Improving the Performance of Concurrent Sorts in Database Systems

by

Weiye Zhang

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1997

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Most research on sorting has been focused on improving single sort performance. This thesis focuses on improving overall system throughput when multiple sorts (or other operations) are running concurrently, competing for the same resources. This is the normal environment in a database system.

A dynamic memory adjustment technique is proposed for external mergesort which adjusts sort space at run time in response to actual input size and available memory space. It balances memory allocation among concurrent sorts so that more sort jobs are done entirely in main memory. This significantly increases system throughput and reduces average response time.

Several read-ahead strategies which reduce disk seeks during merging are studied. Three strategies, called equal buffering, simple clustering, and clustering with atomic reads, effectively reduce disk seeks. The latter two exploit existing order in the input data much better than the first. A set of formulas are derived for estimating the performance improvement resulting from these read-ahead strategies. They provide close estimates for uniformly distributed random data.

The amount of data transferred between main memory and disk is determined by the merge pattern, i.e., the order in which runs are merged. For the case when the sort space remains fixed throughout the merge phase, we derive formulas for calculating the optimum merge cost and provide methods for choosing the best merge width and buffer size. For the case when the sort space is adjustable between merge steps, four merge strategies are proposed and studied. Two are found promising for practical use.

To my parents

Acknowledgements

I am deeply indebted to Professor Per-Åke Larson for his valuable guidance, constant encouragement, patience and support throughout the course of this thesis research. Many ideas in this thesis were sparked from discussions with him. Whenever I was lost, he helped me find the path again. Without his time and effort, this thesis would never have been completed. I am fortunate to have learned from him not only about doing research and writing but also many other aspects of life.

I would like to express my great appreciation to Professors Ian Munro and Kenneth Salem for their guidance. Professor Salem has been watching my progress over the years and has given me much useful advice. Professor Munro provided many helpful suggestions for my thesis research. I am thankful to Dr. Balakrishna Iyer, Professor Bruno Preiss, and Professor Forbes Burkowski for reading this thesis and providing insightful comments.

I gratefully acknowledge financial assistance for my research from the Department of Computer Science, Faculty of Mathematics Graduate Scholarships, Ontario Graduate Scholarships, and the IBM Graduate Student Internship Program. I thank the Center for the New OED and Text Research for providing computing facilities for my experiments. I am grateful to Wendy Rush, Debbie Mustin, Jane Prime, and Ursula Thoene who often helped me beyond their duties.

Special thanks go to Matthew Huras and K.C. Tin at IBM Toronto Lab for giving me a chance to verify my ideas in an industrial environment. The time that Mike Winer spent on explaining the DB2 system is truly appreciated. Thanks also go to James Hamilton and Bruce Lindsay for their encouragement of the work.

Through the journey of my study, many friends offered great help. Weipeng Yan, Qiang Zhu, Glenn Paulley, and Gopi Krishna Attaluri always listened to my problems patiently and provided their suggestions. Lin Yuan helped me with

statistical and mathematical questions. Arunprasad Marathe, Mei Zhou, and many other friends enriched my life with their friendship.

Last but not least, I would like to express sincere gratitude to my wife, Yunqi Sun, who has always been very understanding and patient since the beginning of my Ph.D. studies. When disappointment and depression cloaked me, she lifted my spirits and strengthened my optimism. My daughter accompanied me through my writing of this thesis. She is cute, lovely, and "supportive" in every way. Finally, I am grateful to my parents who led me to the road of science.

Trademarks

IBM are trademarks of International Business Machines Corporation.

Oracle is a trademark of Oracle Corporation.

UNIX is a trademark of The Open Group.

Ordinal and Nsort are trademarks of Ordinal Technology Corporation.

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Problem and Research Goals 2

1.3 Thesis Outline 5

2 Related Work 7

2.1 Internal Sorting 7

2.2 External Sorting 9

2.2.1 External sorting algorithms 10

2.2.2 Techniques for improving external mergesort 12

3 Sort Design and Sort Testbed 20

3.1 A Three-Phase Mergesort Algorithm 21

3.2 Bottlenecks 23

3.3 Reducing Run Input/Output Cost 25

3.4 Sort Testbed 27

3.4.1 Sort testbed components 27

3.4.2 Configurations and test parameters 30

4 Dynamic Memory Adjustment **35**

4.1 Problems with Memory-Static Algorithms 35

4.2 A Memory-Adaptive Mergesort 39

4.3 Memory Adjustment Mechanism 40

4.4 Memory Adjustment Policy 42

4.4.1 System sort space 43

4.4.2 Sort stages 44

4.4.3 Memory adjustment bounds 46

4.4.4 Waiting 48

4.4.5 Fairness 49

4.5 Experimental Results 50

4.5.1 Single sort performance 50

4.5.2 Concurrent sorts 52

4.6 Summary 58

5 Read Ahead during External Merge **59**

5.1 Strategies for Read Ahead 61

5.1.1 Fixed buffering 61

5.1.2 Extended forecasting 62

5.1.3 Simple clustering 66

5.2 Read Ahead for Concurrent Jobs 72

5.3 Performance on Partially Sorted Input 83

5.4 Estimate of Improvement 88

5.4.1 Estimate of simple clustering 89

5.4.2 Estimate of clustering with atomic reads 93

5.4.3 Estimate of equal buffering 96

5.4.4 Estimate of clustering with disk disturbance 97
5.4.5 Estimate of external merge time 100
5.5 Clustering and Buffer Size 102
5.6 Summary 106

6 Merge Patterns **108**
6.1 Tree Representation of Merge Patterns 109
6.2 Memory-Static Merge 111
6.2.1 Optimum merge cost for equal size runs 112
6.2.2 Optimum merge cost for variable length runs 114
6.3 Optimum Merge with Clustering 122
6.4 Merge Width and Buffer Size 128
6.5 Memory-Adaptive Merge 131
6.5.1 Dynamic merge strategies 131
6.5.2 Memory usage patterns 135
6.5.3 Comparisons of the merge strategies 136
6.5.4 Implementation issues and possible improvements 141
6.6 Summary 142

7 Conclusion **143**
7.1 Contributions 143
7.2 Future Work 145
7.2.1 Dynamic memory adjustment 145
7.2.2 I/O improvement 148
7.2.3 Summary 149

A Variable Run Lengths **150**

A.1 Run lengths from sort testbed 150

A.2 Run lengths from triangular probability distribution 151

B Specification of ST-15150W Disk Drive **152**

Bibliography **154**

List of Tables

3.1 Sort system configuration and test parameters 31

3.2 TPC-D sort sizes, scale factor 1.0 33

3.3 Sort job characteristics 34

4.1 Sort stages 44

4.2 Default values for memory usage bounds 48

4.3 Number of external sorts (out of 100 sorts) 55

6.1 Optimum buffer sizes for the examples 129

List of Figures

2.1 External mergesort 11
2.2 Merge patterns for six runs 16

3.1 Three-phase external mergesort 21
3.2 Design of the sort testbed 27

4.1 Problems with memory-static sort 37
4.2 Memory usage of a sort in a commercial DBMS 38
4.3 Sort memory usage bounds 47
4.4 Single sort performance 51
4.5 Concurrent sorts performance (D1, D2) 53
4.6 Concurrent sorts performance (D3, D4) 54
4.7 System performance of D2 56
4.8 System performance of D3 57
4.9 System performance of D4 57

5.1 Consumption sequences 64
5.2 Feasibility of read sequence 67
5.3 Comparison of read-ahead strategies 71
5.4 Feasibility of read sequence 74

5.5 Comparison of read strategies (single sort without disturbance) 79
5.6 Effects of disk disturbance (single sort) 80
5.7 Experiments with multiple concurrent sorts 82
5.8 Disk seeks on partially sorted data 85
5.9 Merge time on partially sorted data 86
5.10 Comparing disk seeks of read strategies 87
5.11 Comparing merge time of read strategies 87
5.12 Ideal consumption sequence for random data 90
5.13 Ideal read sequence 90
5.14 Modeling simple clustering 92
5.15 Varying the input size (simple clustering) 93
5.16 Ideal read sequence 93
5.17 Modeling clustering with atomic reads 95
5.18 Varying the input size (clustering with atomic reads) 95
5.19 Modeling the effects of disk disturbance 99
5.20 Estimate of external merge time 101
5.21 Number of clusters as a function of buffer size 104
5.22 Merge time affected by buffer size 105

6.1 Tree representation of a merge pattern 110
6.2 Merge cost for 50 runs of size 1 113
6.3 Tree representation for encoding 116
6.4 An example of encoding with prefix constraint 116
6.5 Optimum merge tree for equal size runs 118
6.6 Merge cost for variable-length runs 121
6.7 Analysis of merging with clustering 125

6.8 Analysis of merging with clustering 126
6.9 Optimum merge with clustering (fixed buffer size: 8K) 127
6.10 Analysis of merge width and buffer size 130
6.11 Effects of buffer sizes 130
6.12 Memory usage changing patterns 135
6.13 Comparison of merge strategies 140

A.1 Triangular probability distribution for run length 151

Chapter 1

Introduction

1.1 Motivation

Sorting is a frequently used operation in database systems. It is used not only to produce sorted output, but also in many sort-based algorithms, such as grouping with aggregation, duplicate removal, sort-merge join, ANY and ALL operations, as well as set operations including union, intersect, and except [Gra93] [IBM95]. Sorting can also improve the efficiency of algorithms like nested-loop joins and row retrieval via an index.

Sorting speeds have improved dramatically over the past few years. The most recent results are for NOW-Sort, developed at University of California, Berkeley [ADADC+97], Nsort, developed by Ordinal Technology Corp [NKG97], and AlphaSort, developed at Digital Equipment Corporation [NBC+94]. These sorts were designed to break previous sort benchmark records, such as MinuteSort [NBC+94] and Datamation Benchmark [AEA85], which are disk-to-disk sorts with no limit on system resources. The researchers focused on speeding up sorting by using enough memory to sort the data entirely in memory, using as many disks as needed to over-

come I/O bottlenecks, and using multiple processors or a network of workstations to sort the data in parallel.

In a real application system, however, resources are limited and shared by multiple jobs running concurrently. Improving overall system performance is more important than just improving the performance of a single sort running in isolation. Multiple jobs running concurrently in a system will compete for system resources. Large data sets that cannot fit entirely into available memory have to be sorted externally, introducing problems entirely different from those encountered when aiming to break benchmark records. The bottlenecks will be different. The issue of balancing resource usage among concurrent jobs must be addressed. Furthermore, sorting in database systems is normally not a disk-to-disk operation, because operators of a query are often pipelined. The sort input is obtained from another operator and the sorted output is sent to a different operator. When the input is obtained from a fast provider and the output is sent to a fast consumer, source data input and sorted data output are not bottlenecks in sort processing.

This thesis concentrates on sorting issues in database systems, especially when there are multiple sorts running concurrently in the system. We assume that resources, particularly memory resources, are limited. Therefore large sort jobs may have to be done by external sorting due to shortage of memory. The goal is to improve *overall* system (sort) performance by making better use of main memory and I/O resources.

1.2 Problem and Research Goals

When memory resources are limited, external sorting is required to sort large data sets. In this case, I/O time for transmitting intermediate data normally dominates

the sort time. As a result, the amount of available memory may affect the sort time dramatically. Many existing systems rely on static memory allocation, that is, work space is allocated when the sort operation starts and remains unchanged until it finishes. The problem with this approach is that some operations may waste memory while others are starved for memory. There are two reasons why this may happen. First, the input data size may be unknown or poorly estimated at the time a sort starts. Second, in a multiuser environment the workload changes continuously resulting in varying demands on the total memory available in the system. Overall performance can be improved by using algorithms that enable operations to adjust their memory usage at run time in response to the actual size of their inputs and fluctuations in total memory demand. The first goal of this thesis is to have more sorts done in memory by dynamically adjusting the memory usage of sort jobs.

External mergesort is the most commonly used algorithm for external sorting. It has a run formation phase, that produces sorted runs, and a merge phase, that merges the sorted runs into sorted output. During merging, run blocks are consumed in a particular sequence and are usually read in that order. However, researchers have found that disk seek time can be reduced by reading the run blocks in a different order if extra memory is available [Zhe92] [ZL96b] [ECW94]. Several read-ahead strategies have been proposed to reschedule read orders, but these strategies were designed for single sorts, i.e., no concurrent jobs access the disk at the same time. This motivated the study of strategies for concurrent sort jobs as well as estimation of the performance improvement of read strategies. The second goal of this thesis is to find good read-ahead strategies taking into account concurrent jobs, and to estimate the improvement resulting from these strategies.

When runs have to be merged in multiple steps, the amount of data transferred

between main memory and disk is determined by merge pattern, i.e., the order in which runs are merged. Knuth described a method for constructing an optimum merge pattern under the condition that the sort space remains fixed during the whole merge phase [Knu73]. However, no one has ever given the cost of an optimum merge. One of the goals of this thesis is to build cost models for the optimum merge. The models help investigate the relationship between optimum merge and read order scheduling, and the relationship between merge width and buffer size.

Given a fixed amount of memory, we can use all buffers to provide the maximum merge width or leave some buffers for read order scheduling. We can also use large buffers to reduce disk seeks, but it results in fewer buffers. Using large merge width can merge more runs in each step, which reduces the amount of data transmitted, while using more buffers for read ahead and/or using large buffers can reduce disk seeks. So there is a tradeoff between data transfer time and disk seek time. To find the optimum merge width with read order scheduling and the optimum buffer size taking into account merge width are also the goals of this thesis.

If sort space is adjustable during the external merge phase, an optimum merge pattern cannot be guaranteed due to unpredictable memory changes. The last goal of this thesis then, is to find reasonable strategies for adjusting the merge width dynamically.

In summary, the ultimate goal of this thesis is to achieve better sort throughput in an environment where multiple sorts (possibly with other jobs) are running concurrently, competing for the same resources (memory, disks, or both). By making better use of memory resources, we try to have more sorts done entirely in memory; we use extra memory for read order scheduling to reduce disk seeks; and we select proper merge patterns to reduce the amount of data transferred between disk and main memory. Memory resources will be better utilized by dynamically adjusting

the work space of sort jobs.

1.3 Thesis Outline

The rest of this thesis is structured as six chapters.

Chapter 2 reviews previous research on sorting, including recent interests in internal sorting, external sorting algorithms, and techniques for improving external mergesort.

Chapter 3 serves as a starting point for the following chapters. It first introduces a three-phase external mergesort algorithm, followed by an analysis of the bottlenecks in sort processing. It then introduces techniques to reduce run input/output cost, including dynamically adjusting sort memory space, rescheduling read order of run blocks, and choosing proper merge patterns, which are the main topics of this thesis. To evaluate the proposed techniques and confirm our analysis results, a sort testbed was implemented. The design of the testbed is also described in this chapter.

Chapter 4 studies techniques for dynamic memory adjustment. It begins with an analysis of the problems introduced by a memory-static sort, then gives a memory-adaptive mergesort algorithm. We propose a memory adjustment mechanism and a policy that balances memory usage among concurrent sorts. The technique enables sorts to adapt their memory usage to the actual input size and available memory space, and enables concurrent sorts to cooperate with each other when they compete for memory resources. Experimental results show that this technique allows more sort jobs to be done entirely in memory which significantly improves overall system performance.

Chapter 5 presents a set of read-ahead strategies aimed at reducing disk seeks

during external merging. These strategies include fixed-buffering, extended forecasting, simple clustering, and clustering with atomic reads. The last strategy is designed for concurrent sort jobs. It helps retain performance when disk contention is high. Besides comparing the strategies on random data, we investigate the effects of partially sorted input. To estimate the performance improvement, a set of formulas is obtained from analysis of these strategies. The accuracy of the estimates is confirmed by experimental results. We also study the problem of how to choose the buffer size.

Chapter 6 explores merge patterns with the goal of reducing the amount of data transferred between disk and main memory when runs are merged in multiple steps. For the case that sort space remains fixed during the external merge phase, a formula is derived for calculating the exact optimum merge cost for equal size runs, while a lower bound and an upper bound on the optimum merge cost are given for variable length runs. Approximation formulas are provided for both cases. Based on these results, we study the relationship between optimum merge and read order scheduling, and the relationship between merge width and buffer size, then propose methods to determine the optimum merge width and the optimum buffer size. For the case that a sort is able to adjust its memory usage between merge steps, four strategies are proposed for memory-adaptive merge: lazy merge, eager merge, improved eager merge, and optimistic merge. They are compared along with memory-static merge. The improved eager merge strategy and the optimistic merge strategy are promising for practical use.

Chapter 7 summarizes the main contributions of this thesis, and discusses problems and possible solutions for future research.

Chapter 2

Related Work

Sorting is a fundamental problem in computer science. It has been extensively studied for several decades. This chapter surveys some work in the literature related to the issues studied in the thesis. For internal sorting, this survey focuses on the recent interests in sort performance enhancement rather than the sorting algorithms. For external sorting, the survey covers two sorting algorithms: external mergesort and external distribution sort. This is followed by techniques for improving external mergesort, including algorithms for run formation, read ahead for merging, merge patterns, and a dynamic memory adjustment technique.

2.1 Internal Sorting

Internal sorting deals with data sets which can be sorted entirely in main memory. Many algorithms have been invented for internal sorting, including insertion sort, selection sort, bubble sort, quicksort, bucket sort (distribution sort), radix sort, mergesort, and heapsort, etc. [Knu73] [CLR89] [Man89]. Quicksort and bucket sort are two algorithms commonly used in practice.

Early studies of internal sorting focused on time and space complexity. Nowadays, most researchers in the sort community appear to direct their effort towards sort benchmarks. More attention is paid to issues in computer architecture. People try to speed up sorting by exploiting all system resources: processors, cache, memory, and I/O [NBC+94][ADADC+97].

Internal sorting algorithms typically perform a sort job in three steps: data input, sorting, and result output. The three steps are performed sequentially. Nyberg et al. proposed the AlphaSort algorithm which uses quicksort to sort data in small buffers then uses a tournament tree to merge the sorted buffers [NBC+94]. In this algorithm, data input can be overlapped with sorting of the buffers, and the result output can be overlapped with merging. Consequently, the CPU and I/O resources are better utilized and the sort elapsed time is reduced.

If input data is read from disk and result data is written to disk, disk I/O tends to be the bottleneck. This problem can be solved by striping data across many disks[1]. Data striping can balance the workload among multiple disks, which allows parallel reading and writing, and thus increases the effective disk bandwidth [SGM86] [Kim86].

For the sorting step, the major cost comes from memory accesses. Processor speeds continue to increase faster than memory speeds causing an algorithm's cache behavior to become increasingly important. The latency of accessing data from cache is much smaller than from memory. Cache miss penalties have a great influence on sort performance so that cache locality becomes an important factor in sort algorithm design. Among the classic sorting algorithms, quicksort has good cache

[1][BGK90] mentioned a sort on a 100-processor 100-disk system, while DeWitt, Naughton, and Schneider used 32 processors, 32 disks, and 224M of memory [DNS91] for their sorting experiments. The I/O bottleneck was overcome by striping data across many disks to get sufficient I/O bandwidth.

locality since it accesses memory in sequential patterns. Moreover, because of its divide-and-conquer strategy, the data set is divided recursively into smaller pieces, eventually small enough to fit into cache. Many memory accesses can be avoided. So quicksort was employed to construct the cache-efficient AlphaSort in [NBC+94]. LaMarca and Ladner studied the influence of caches on several sorting algorithms, including heapsort, mergesort, quicksort, and radix sort [LL97a] [LL97b]. They showed that radix sort has poor cache locality, and therefore performs worse than other algorithms. To improve cache locality of the sorting algorithms, they used a d-heap (a d-ary tree) for heapsort, employed multiway merging for mergesort, and proposed multi-partitioning for quicksort. All modified algorithms perform better than the original algorithms due to lower cache miss rate. The modified heapsort is outperformed by the modified mergesort and quicksort. Unfortunately, they did not compare their algorithms with the AlphaSort algorithm which sorts small buffers using the quicksort algorithm followed by a multiway merge.

Many researchers are working on sorting using multiprocessors and distributed systems, producing many sort benchmark records [ADADC+97] [NBC+94] [GT92] [DNS91] [BGK90]. Parallel sorting and distributed sorting have been studied extensively from both theoretical and practical perspectives. There are many interesting problems in this area [FL96] [Gra90] [ID90] [Qui88] [BBW88] [RSS85] [Akl85] [BDHM84] [BBDW83]. However, the topic is outside the scope of this thesis, and will not be investigated here.

2.2 External Sorting

This section describes two commonly used external sorting algorithms: external mergesort and external distribution sort. We discuss several important techniques

for improving external mergesort and show the limitations of these techniques.

2.2.1 External sorting algorithms

External sorting refers to sorting very large data sets that cannot fit into main memory. Many algorithms have been developed for external sorting, most of them derived from techniques used in internal sorting. Early studies of external sorting focused on using tapes as secondary storage, while current research concentrates on disk-based algorithms.

External mergesort is a well-known algorithm for external sorting [Knu73]. It consists of two phases: a run formation phase and a merge phase. During the first phase, the data to be sorted is divided into smaller sets that can be sorted in main memory. Each set is sorted and then stored on external storage. These sorted data sets are called *runs*. In the merge phase, the runs are merged into sorted output. Figure 2.1 shows the two phases of external mergesort. Elapsed time of the two phases is usually used as a measure of sort performance. When data input and result output are fast, reading and writing run data becomes the bottleneck.

Among all the external sorting algorithms, external mergesort is the most thoroughly studied algorithm. Aggarwal and Vitter claimed that mergesort is an optimal external sorting method (up to a constant factor) in the total number of I/O operations required [AV88]. Many techniques have been developed to increase its efficiency. We will discuss some of the important techniques and their limitations in Section 2.2.2.

Distribution sort [Kwa86] is also called distributive sort [Ver89] or bucketsort [Knu73]. This internal sorting method has been applied to external sorting to provide an external distribution sort. External distribution sort also consists of two phases. During the first phase, it distributes the input data into a set of range-

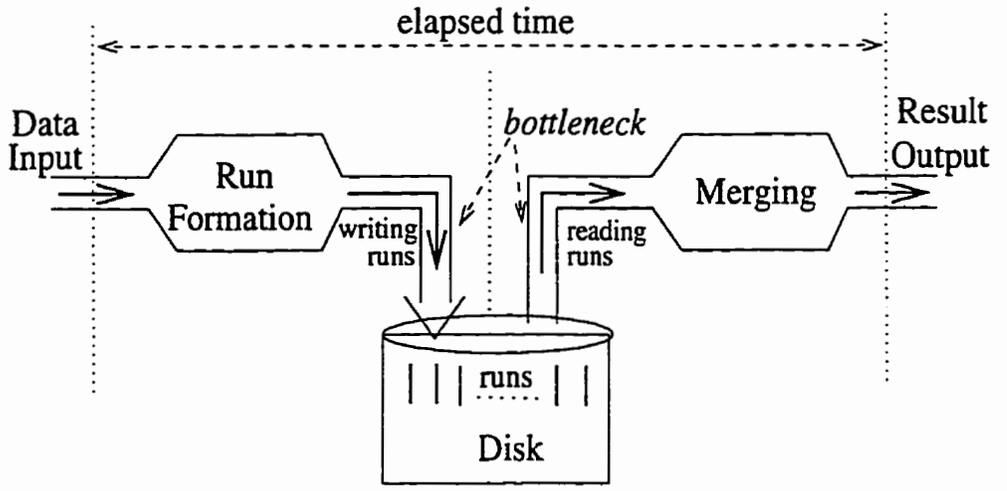


Figure 2.1: External mergesort

disjoint relatively ordered buckets. This process is repeated on the buckets until each bucket is small enough to be sorted in memory. At the second phase, the data in each bucket is sorted internally and the final result is formed by combining all the buckets.

The main issue for this algorithm is finding some way to distribute data evenly among the buckets, thereby reducing the recursive distributions on the buckets. Sampling is commonly used for this purpose. There are many variations of external distribution sort. For example, Cunto et al. proposed an *in situ* distributive external sorting algorithm, which recursively distributes a file into m subfiles. The partition is based on a random sample of size $km - 1$. The values of m and k depend on the data size and the track size of the disk to be used. The analysis of the algorithm is presented in [CGMP91].

There is a duality between merge sort and distribution sort, with a correspondence between externally ordered buckets and internally ordered runs, between

distribution passes and merge passes, etc. (see [Knu73] and [Kwa86]). Although there are similarities between these two external sorting algorithms, the behavior of the algorithms and the research issues are totally different. It is claimed in [LV85] that distribution sort cannot compete with mergesort, unless particular disks (e.g., associative secondary storage) are used. One of the reasons is that an infeasible amount of memory is required to finish the distribution in one pass for large data files.

Besides external mergesort and external distribution sort, there are many other algorithms for external sorting, such as external quicksort [GBY91] [Ver88], external tag sort [Kwa86], external heapsort [WT89], and external bubblesort [DL92]. These algorithms are derived from the corresponding internal sorting algorithms. Since they require more I/O operations than external mergesort, these algorithms are rarely used in practice.

2.2.2 Techniques for improving external mergesort

External mergesort is the most commonly used algorithm for external sorting. Many techniques have been developed to improve its performance, including algorithms for run formation, read ahead for merging, optimum merge patterns, and dynamic memory adjustment.

Run formation

The simplest way to create a run is to fill all the available memory with input records, sort them using some internal sorting algorithm (e.g., quicksort), and then write the run to external storage. The size of the run is the same as the size of available memory for the internal sorting. All runs generated are the same size,

except the last one. Quicksort has good cache locality, but it is hard to overlap CPU processing and I/O. CPU and I/O resources are not fully utilized.

Replacement selection is one of the most well-known methods used for run formation. To create a run, this method first fills all available memory with records and organizes them into a tournament tree (normally using a heap). A loser tree is better than a winner tree since it requires fewer key comparisons for updating the tree structure [Knu73]. The record with the smallest key is then removed from the top of the tree and written to a run file. A new record from the input is inserted into the tree. If the key of this new record is smaller than the key of the last record written out, the new record is marked "dead" and otherwise left unmarked. In comparisons among records in the tree, marked records are always considered "larger" than the unmarked records. Then the currently smallest record is removed and written to the current run file. A new record is inserted into the tree again. When there are no unmarked records left in the tree, the current run is closed, a new run is started, and all records are unmarked.

Replacement selection provides perfect overlap of data input, internal sorting, and run output. In addition, it can produce runs larger than the available memory size. The average run length for random data is twice the size of available memory [Knu73]. Several variants of this algorithm produce even longer runs [Knu73] [Kwa86], but they require several read passes of the input data. The benefit from the longer runs may not fully compensate the extra cost at the run formation phase. The major problem with replacement selection is its poor cache performance when the tournament tree is large. Only a small part of the tree resides in cache. When the record with the smallest key is removed, a new record is added into the tree. Each replacement selection step traverses the tree from the bottom to the top. The traversal path in one step is likely to be different from the path in the next step,

resulting in random memory accesses and many cache misses. One way to improve the cache performance of replacement selection is to have as many parent-child node pairs fit in the same cache line as possible. This can reduce cache misses by a factor of two or three; however, according to [NBC+94], quicksort is still more attractive.

Read ahead for merging

The merge phase requires at least one buffer for each run. If there are no extra buffers, reading will stop during merging until a buffer becomes free. Then merging stops during reading. The overlap of reading and merging requires additional buffers. Several buffer allocation schemes have been proposed.

Double buffering is a commonly used scheme. Two input buffers are used for each run to achieve better overlap between reading and merging. One block from each run is read into memory and the merge process starts. Then the second block of each run is read in during merging. After that, as soon as a buffer is emptied, the next block of that run is read into memory. Salzberg strongly advocates this technique to achieve "perfect overlapping" of merging and reading [Sal89].

Knuth proposed the *forecasting* technique which uses only one extra buffer for read ahead [Knu73]. By comparing the last key of each block in memory, it is easy to decide which block will be emptied first. The next block of that run will be read into the extra buffer.

Both double buffering and forecasting can achieve complete overlap of merging and I/O. In this case, the elapsed time of the merge phase is normally dominated by the I/O time. The time required to read a block of data from disk consists of two parts: disk seek time (including rotational latency) and data transfer time. Two techniques have been proposed for reducing the total seek time: increasing the

buffer size and changing the read order. Increasing the size of each buffer reduces the total disk seek time simply by reducing the number of reads. The effects of this technique are discussed in [Sal89].

The idea of changing the read order was introduced by Zheng and Larson [Zhe92] [ZL96b]. Extra buffers make it possible to read data blocks in an order that is different from the order they are consumed during merging. We can then try to read them in an order that minimizes the total seek time (taking into account that the number of buffers is limited). Zheng and Larson introduced the concepts *consumption sequence* and *read sequence* and proposed a heuristic for computing near-optimal read sequences. However, the proposed method has one deficiency: it relies on the physical location of the run blocks, which is often unknown for modern disks. Estivill-Castro and Wood continued this research and proposed an algorithm that groups adjacent run blocks together to reduce the number of disk seeks, assuming that run blocks of the same run are stored in adjacent locations on disk [ECW94]. So this algorithm does not rely on the physical location of run blocks. However, both methods were designed for single sorts. If there are other jobs accessing the run disk at the same time of merging, the disk head may move away randomly after each disk read. Disk seek time will not be reduced as expected.

Merge patterns

When runs cannot be merged in a single pass, the merge cost can be measured by the amount of data transferred. This is determined by the merge pattern. As an example, Figure 2.2 shows three merge patterns for six runs. The maximum merge width is 4. Each circle represents a run (either an initial run, or a run created by merging), and the number in the circle represents the run length. The three merge patterns result in different merge costs.

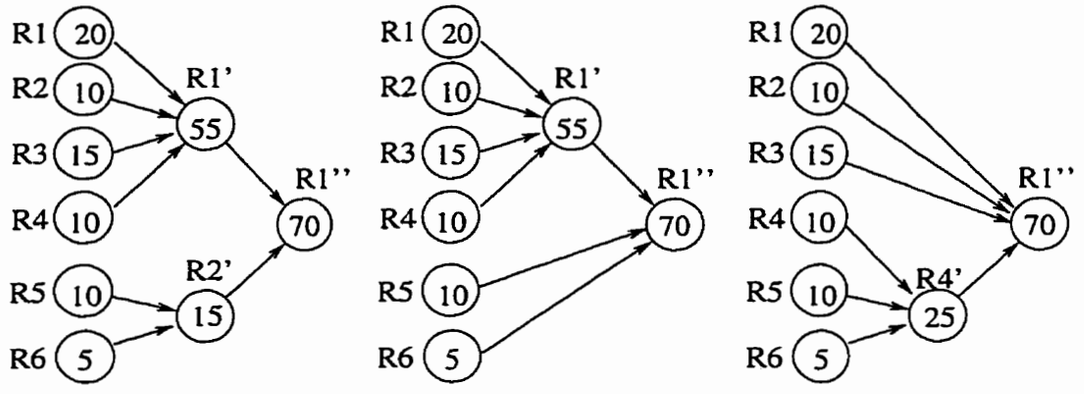


Figure 2.2: Merge patterns for six runs

Figure 2.2(a) shows a straightforward merge method using multiple passes. All runs are merged into larger runs before going to the next merge pass. In this example, two passes are needed to finish the merge. All the data (70 blocks) are read into memory and written to disk once before the last merge pass.

In fact, runs need not be merged in passes. The only requirement is that each merge step must reduce the number of runs. Some records may be involved in many merge steps while others may be involved in only a few steps. Figure 2.2(b) shows a merge pattern using multiple steps. In this example, the last two runs (R5,R6) need not be merged until the last merge step. So only the first 4 runs (totally 55 blocks) are read and written before the last merge step. The merge cost is lower than the cost of merging in passes.

There are many valid merge patterns. Which one transfers the least amount of data? Under the assumption that the maximum merge width remains fixed, this problem has a very simple solution, as described by Knuth in [Knu73] (pp.365-366): "An optimum pattern for this situation can be constructed without difficulty using

Huffman's method, which may be stated in merging language as follows: 'First add (1-S) mod (P-1) dummy runs of length 0. Then repeatedly merge together the P shortest existing runs until only one run is left.' ". S denotes the number of initial runs and P is the maximum merge width. Instead of adding dummy runs, Harold Lorin proposed an algorithm to calculate the number of runs for the first merge step ([Lor75] pp.287). The algorithm is described as below:

```
K = (S-1) mod (P-1)
if ( K > 0 ) M = K + 1 else  M = P
merge the M shortest runs in the first step
merge the P shortest runs in each following step
```

Figure 2.2(c) gives an optimum merge pattern for the six runs. Only 25 blocks are read and written before the last merge step.

Although the optimum merge pattern can be simply constructed, no one has ever given formulas for calculating the cost of an optimum merge. Unfortunately, this method does not apply to the case when the sort space, and therefore the maximum merge width, may change during merging.

For a given amount of memory, the number of buffers is inversely proportional to the buffer size. The maximum merge width increases as buffer size decreases. Large merge width minimizes the amount of data transmitted, while small buffers increases disk seeks. Then what is the optimum buffer size for a given memory space? This is a tradeoff between transfer time and disk seek time. Graefe has studied this problem and shows that the optimum buffer size can be obtained by minimizing

$$(t + s/C)/(\ln(M/C)) \ ,$$

where t is the transfer time per page, s is the average seek time, M is the memory

size, and C is the buffer size [Gra90]. This result is derived from an approximate formula for data transfer size, which may underestimate the real transfer size by over 20%. Finding the optimal buffer size from this result is complicated. Graefe resorted to full enumeration by testing each physically possible buffer size.

Dynamic Memory Adjustment

Sorting is a memory intensive operation whose performance is greatly affected by available memory. System performance improves radically when the data is processed entirely in memory. Whether external sorting can be avoided or not depends on the input data size and available memory size. If a sort uses a predefined constant amount of memory, extra memory available in the system will not be used. On the other hand, a small sort may not use all the space allocated. The extra space allocated to this sort is wasted and cannot be utilized by other jobs in the system. Memory utilization is low in both cases. Can this problem be solved if a sort allocates an exact amount of space determined by the input data size? Unfortunately, no. First of all, the input data size is often unknown, especially when the data is pipelined from another operator in the query. Secondly, with multiple jobs running in a system, the available memory changes dynamically. There may be only a small amount of memory space available when a sort starts, but more memory may become available during sorting. Allocating a fixed amount of memory for sorting prevents the sort from using the newly available memory space.

Dynamic memory adjustment for sorting is a technique to solve the above problem. It was first studied by Pang, Carey and Livny [PCL93a]. They proposed memory adjustment strategies for external mergesort. For the run formation phase, they considered quicksort and replacement selection. When quicksort is used, adjustments can only be done when a run has been finished and output. Data input,

internal sorting, and run output cannot be overlapped. When replacement selection is used, memory adjustments can be done by expanding or shrinking the tournament tree. But it has been found in [NBC+94] that replacement selection has poor cache behavior, which degrades CPU utilization. For the merge phase, they proposed a method to dynamically adjust sort space (merge width) during merging, basically by stopping the current merge step and starting the next merge step in a new merge pattern. The problem is that frequent adjustment within a merge step may be expensive. Also in their study, they did not consider unknown input size and the effects of several sorts running concurrently.

Further research on dynamic memory adjustment for sorting was done in my previous work at IBM Toronto Lab [ZL96a]. A memory-adaptive sort (MASORT) was designed making a sort adapt its memory usage to both input size and available memory space. The method aims to balance the memory usage among concurrent sorts, but only limited cases (two concurrent sorts and three concurrent sorts) were studied.

Up to now, dynamic memory adjustment techniques have been applied only to external mergesort. However, it is possible to apply the idea to other external sorting algorithms, such as external distribution sort.

Chapter 3

Sort Design and Sort Testbed

Resources required for sorting include CPU, memory, and disk(s). Sort performance is mostly affected by the utilization of these limited resources. Our sort design aims to improve sort performance by exploiting these resources. The goal is to increase system sorting throughput and reduce average response time. In this chapter, we first introduce a three-phase mergesort algorithm followed by an analysis of the bottlenecks in sort processing. Since run input/output is normally the bottleneck of external mergesort, we describe several ways to reduce run input/output cost, including dynamically adjusting sort memory space, rescheduling run read orders, and using proper merge patterns. These techniques are aimed at improving sort performance by utilizing memory and disk resources better. They are the main topics of this thesis and will be discussed in detail in Chapters 4 to 6. In the last section, we describe a sort testbed, which has been used to experimentally study the effects of the techniques proposed.

3.1 A Three-Phase Mergesort Algorithm

External mergesort is the most commonly used algorithm for external sorting. It consists of two phases: a run formation phase and a merge phase. The standard algorithms for run formation are quicksort and replacement selection. However, both algorithms have drawbacks: replacement selection suffers from poor cache performance, and quicksort does not overlap sorting and input/output. Following [NBC+94] we therefore opt for a two-phase algorithm for run formation, which results in a three-phase external mergesort algorithm as shown in Figure 3.1.

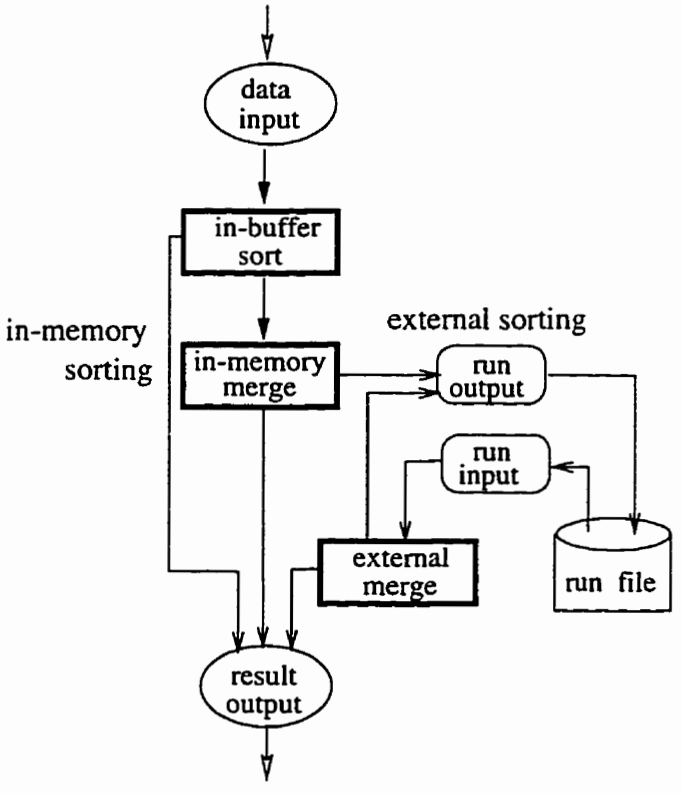


Figure 3.1: Three-phase external mergesort

This algorithm has three phases: an *in-buffer sort* phase which sorts data within each buffer, an *in-memory merge* phase which merges sorted buffers, and an *external merge* phase which merges sorted runs. Each phase involves input/output operations and sort/merge processing. By separating CPU processing and I/O operations, we get seven sort steps: data input, in-buffer sort, in-memory merge, run output, run input, external merge, and result output.

If input data fits into one buffer, sorted output will be produced directly from the in-buffer sort phase. If the input data is larger than one buffer but fits into available memory, sorted output will be produced from the in-memory merge phase. If the input data does not fit into the available memory, the sort process will go through the first two phases multiple times to produce runs, and may execute several external merge steps. Sorted output will be produced from the last merge step in the external merge phase.

In this algorithm, the in-buffer sort can use any internal sorting algorithm to sort the data within a buffer. Quicksort is adopted in our implementation. The buffer size is selected small enough to fit into second level cache (on-board cache), which improves cache performance even if some other algorithm is used for in-buffer sorting. The in-memory merge and the external merge use tournament trees to merge the sorted buffers or sorted runs.

We choose this sorting algorithm because it has several desirable characteristics.

First, the algorithm has good cache performance. This is because it sorts data in small buffers using quicksort and then merges the sorted buffers and sorted runs using multiway merging. Both quicksort and multiway merging have been found to have good cache locality [LL97a] [NBC+94].

Second, the algorithm allows almost full overlap of CPU and I/O operations, which helps improve CPU and disk utilization. Many sort steps can be overlapped,

including data input and in-buffer sort, in-memory merge and result/run output, in-memory merge and data input (for the next run), run input and external merge, as well as external merge and result output. These overlaps can be achieved by using separate I/O agents (processes, threads) for I/O operations. If sort/merge steps and input/output steps are fully overlapped, a sort is completely CPU bound or completely I/O bound, and sort performance is determined either by CPU time or by I/O time.

Third, the algorithm results in smooth I/O operation because it uses multiway merging while producing initial runs, reading run blocks, and writing intermediate runs. It also allows sort jobs to use large buffers (I/O unit) to transfer intermediate data between main memory and disk.

Fourth, this algorithm supports an incremental sorting style. A sort can allocate new space after a buffer is full and before or after the buffer is sorted. This makes it possible for a sort to adapt its memory usage to unknown input data sizes (see Chapter 4).

3.2 Bottlenecks

Any sort step may become the bottleneck of a sort. It depends on system configuration, input data size, and other operators in the query which requires the sort.

- *Data input* becomes the bottleneck when input data is from either a disk or an operator that provides the data slower than the in-buffer sort is able to process the data.

- *In-buffer sort* becomes the bottleneck when input data originates from a fast provider. This could be an operator or multiple disks.
- *In-memory merge* becomes the bottleneck when it produces sorted data that is sent to a fast consumer, either an operator or multiple disks.
- *Run output* becomes the bottleneck when writing run data is slower than in-memory merge processing.
- *Run input* becomes the bottleneck when reading run data is slower than external merge processing.
- *External merge* becomes the bottleneck when both run input and result output are faster than the merge processing.
- *Result output* becomes the bottleneck when the sorted output is sent to an operator or a disk that consumes the data slower than the merge processing.

In a multiuser environment, available memory is limited and may vary continuously. Although main memories are becoming very large, data size increases even faster, especially in database applications and information retrieval. Large data sets have to be sorted with external merge. Since processors are much faster than disks, input/output is still the most common bottleneck unless many disks are used to stripe the data on disks. Run data input/output is a major cost in external sorting.

In database systems, operators of a query are often pipelined. The input data for sorting is usually from an operator rather than directly from disk. The sorted output is often sent to another operator, rather than written to a disk. The overall performance of a query is affected by the sort operator when it becomes the bottleneck, that is, the data input and result output are both fast sufficiently that they

are not the bottleneck. Therefore, we assume that the bottleneck for (external) sorting is run data input/output, and the major cost of sorting is determined by the cost of writing and reading run data blocks. Source data input and sorted result output are not considered in this thesis, although they are important in disk-to-disk sorting problems.

3.3 Reducing Run Input/Output Cost

Sort performance improves radically if input data can be sorted entirely in memory, in which case there is no run input/output cost. Whether external sorting can be avoided or not depends on the input data size and available memory size. Static memory allocation either wastes memory space or fails to make full use of memory. In this thesis, we propose a dynamic memory adjustment technique to improve memory utilization. It supports run-time adjustment of in-memory work space for external mergesort. The goal is to have more data sets sorted completely in memory, thereby improving overall system performance, especially when multiple sorts are running concurrently in a system. Our technique enables sorts to adapt their memory usage gracefully to the actual input size and available memory space. Compared to static memory allocation, this technique wastes less memory. In addition, a large sort may expand to use all available memory resources (for sorting). Both sort throughput and response time can be improved significantly by using this technique. A *memory-adaptive sort* was implemented using this technique in the sort testbed. It was compared to a sort without dynamic memory adjustment, called *memory-static sort*. Details are given in Chapter 4.

If data sets are larger than the available memory, they have to be sorted with one or more merge steps. The run I/O cost can be reduced by reducing data transfer

time, disk seek time, or both.

During external merging, available extra memory can be used for reading run blocks which are not required immediately but can be read with less disk seek time. This is achieved by a read ahead technique which reschedules the read order of run data blocks. The run blocks are read in such an order that the disk seek time is reduced and therefore the sort performance is improved. We can also use large block size, or I/O transfer unit, to reduce disk seeks. Details of these techniques are discussed in Chapter 5.

If memory is very small or the number of runs is large, the runs have to be merged in multiple steps. Part or all of the data will be read from and written to disk multiple times. Both data transfer time and disk seek time are affected by the order in which runs are merged. Choosing a proper merge pattern can reduce data transfer cost as well as disk seeks. When the available memory changes dynamically in the system, sort space may change from one merge step to another. An optimum merge pattern cannot be guaranteed in such cases. Heuristic strategies are developed in this thesis to deal with dynamic merges. Details are given in Chapter 6.

In summary, we attempt to reduce run I/O cost by

- reducing the number of external sorts using dynamic memory adjustment technique;
- reducing disk seeks by exploiting extra memory for read ahead and/or larger units of I/O;
- reducing the total amount of data transferred between memory and disk by choosing proper merge patterns.

3.4 Sort Testbed

3.4.1 Sort testbed components

To evaluate the ideas and techniques proposed in this thesis, a sort testbed was implemented. The sort testbed simulates (part of) a database environment, as shown in Figure 3.2. It includes a sort job initiator, a memory space manager, a disk space manager, asynchronous I/O support, a disk access disturbance facility, and the sort system. When provided with system configuration parameters and sort test parameters, the testbed generates and executes a sequence of sort jobs and collects performance results.

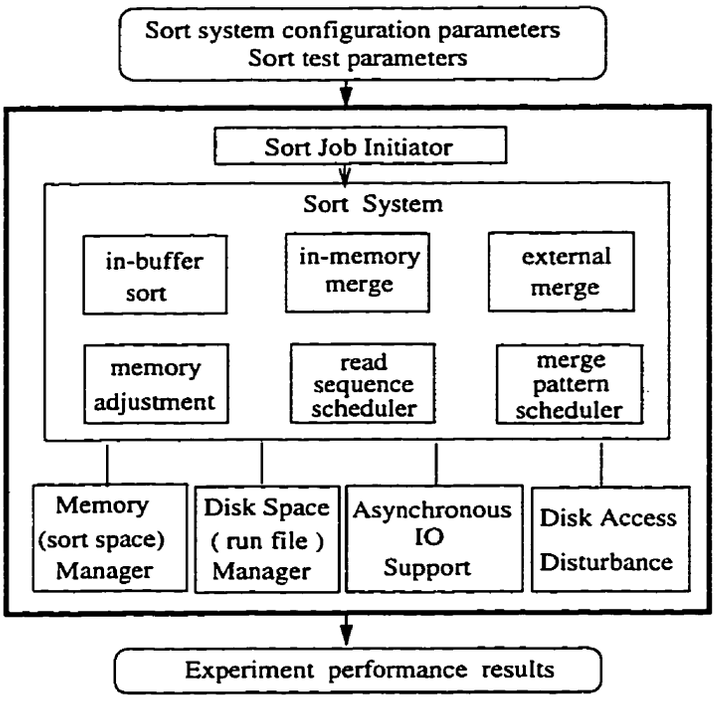


Figure 3.2: Design of the sort testbed

The *memory manager* is similar to a work space manager that manages the work space of all operators in a database system but, in our case, it manages only the system sort space.

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

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

The *disk access disturbance* module simulates other jobs which access the disk(s) storing the run file as an external sort is running. It reads a small chunk of data (4K) from a random position in the raw partition of the disk. The purpose is to move the disk head away from its current position.

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

The *sort system* implements the sort mechanism with the ideas and techniques proposed in this thesis. It includes in-buffer sort, in-memory merge, external merge, memory adjustment, read sequence scheduler, and merge pattern scheduler. The system is multi-threaded with each sort job running as a separate thread. Using threads reduces context switch cost and makes it easier for concurrent jobs to share resources.

The *in-buffer sort* sorts a set of pointers pointing to the records in a data buffer. If the input data fits in one data buffer, the sorted records will be collected from the buffer using the sorted pointers, and the sort job is finished. The *in-memory merge* merges sorted buffers, while *external merge* merges sorted runs. Both may produce either a run or the final result, i.e., the sorted output. The memory usage of a sort

is allowed to change during sorting. The *memory adjustment* component makes the decision of adjustment based on a pre-defined memory-adjustment policy. The *read sequence scheduler* produces a better read sequence of run blocks to reduce disk seek time. When runs cannot be merged in a single step, the *merge pattern scheduler* determines the merge width of each merge step.

Quicksort is used for in-buffer sorting. When there are a lot of equal keys, the performance of quicksort degrades dramatically. There are many techniques to solve this problem [Weg85], but none of them were implemented in this testbed. Sorting of data with equal keys is beyond the scope of this thesis. Tournament trees are used for multiway merging, both during in-memory merge and external merge. A loser tree is used because it has better performance than a winner tree for updating the tree structure [Knu73].

In the implementation, record pointers and data records are stored in a contiguous memory space, which is called a *memory adjustment unit*. The front part is allocated for pointers, and the remaining part is allocated for data records. Data records are 64 bytes long with a randomly generated 10 byte key.

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

In this testbed, one disk is used for storing runs. To fully utilize CPU and I/O resources, two I/O agents are used for the disk. If the sort is completely I/O bound, there is always an I/O request in the I/O queue, which will keep the disk busy all the time. If the processing is CPU bound, one I/O agent is enough, while the other

I/O agent is idle all the time. In both cases, CPU time and I/O time are fully overlapped.

Two output buffers (double buffering) are reserved for each sort in the testbed, one for collecting output data and one for writing. If the sort process is completely I/O bound or completely CPU bound, the two buffers will be enough to overlap CPU and I/O time. But if processing speed and I/O speed change dynamically (because of system workload and data input speed, etc.), sort processing and I/O may wait for each other alternatively. Extra output buffers may help to reduce the wait time in this case. This issue is not investigated further in this thesis and therefore not considered in the implementation.

Extra buffers are also used for reading to overlap CPU and I/O time. During run formation, at least one buffer is used for read ahead until not enough memory is available. During external merging, with one buffer for each run involved in merging, a minimum of two buffers are used for read ahead. CPU (merging process) and I/O (reading) time are fully overlapped if the sort processing is completely I/O bound or completely CPU bound.

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

3.4.2 Configurations and test parameters

The machine used for all experiments reported in this thesis is a Dec Alpha 3000/500S with a clock rate of 150 MHz and a 512 Kb off-chip cache. Run data is stored on a

single disk, a Seagate ST-15150W (see Appendix B for its specification). Table 3.1 lists the configuration parameters of the sort system and their default values.

Table 3.1: Sort system configuration and test parameters

Parameter	*Default*
Sort system parameters	
system sort space	32 M bytes
one sort space limit	4 M bytes
sort buffer size	64 K bytes
run block size	32 K bytes
I/O agents per disk	2
maximum concurrency	10
maximum merge width	no limit
read disturbance rate	0
Test data parameters	
number of sorts	100
random seed	97
overlap of key ranges	1
concurrency degree	10
sort size distribution	D3

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

Sort buffer size is the size of a data buffer for in-memory sort/merge. The unit of memory adjustment is a data buffer plus the space for additional data structure for sorting. Most modern systems provide large second level cache (on-board cache). Buffers should be selected small enough to fit into this cache.

Run block size is the buffer size for external merge and also the I/O transfer unit.

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

of active sorts drops below the limit.

Maximum merge width limits the merge width so as to study the effect of merge patterns.

Read disturbance rate is used for generating disk disturbance requests which affect the performance of read-ahead strategies for external merge. It is the probability of disturbance for a run read request. Before each run read request, a random number within [0, 1) is generated. If the number is smaller than the disturbance rate, a disturbance request is issued. The disk head is moved away resulting in a disk seek for the run read request.

The sort jobs in each experiment run is determined by sort test parameters. *Number of sorts* is the total number of sort jobs for a test run. *Random seed* is the seed for the random number generator used to generate input data.

Overlap of key ranges is used for generating partially sorted input. The keys for each run are generated randomly from a range. This parameter controls the overlap of the key ranges between two consecutive runs. Default value of this parameter is 1, in which case the key ranges of all runs are fully overlapped, which produce completely random data. Decreasing this value increases the presortedness of input data. When it is 0, the key ranges of all runs are not overlapped. In this case, the keys between the runs are already in sorted order, but the keys in the input for each run are not sorted.

Within each test run, a fixed number of sort jobs are always running concurrently, which is controlled by the *concurrency degree*. If the concurrency degree is n, n sort jobs would be submitted to the sort system initially and as soon as one is finished another one would be submitted.

The input size of a sort job is randomly drawn from a specified sort size distribution. To get some basis for deciding on a distribution of sort sizes, we analyzed

the sorts generated when running the TPC-D benchmark queries [Raa95]. More specifically, we analyzed the execution plan used by a major commercial DBMS for each of the 17 queries on a 1 Gb TPC-D database with 26 indexes. We found a total of 55 sorts with the size distribution shown in Table 3.2.

Table 3.2: TPC-D sort sizes, scale factor 1.0

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

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

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

Table 3.3 shows the five sort job sets used for experiments. D0 is used for single sort experiments. The sort size can be changed to any size required for testing. D1 is from execution plans of a set of queries on a small database in our system. D3 is based on the result of our analysis of the queries in the TPC-D benchmark. D2 is a case between D1 and D3, while D4 contains larger sorts than D1 to D3. They reflect several types of workload. D1 represents a type of workload that contains

only small sorts, which will test whether memory-adaptive sort will degrade system performance when available memory is so large that dynamic memory adjustment is not necessary. Both memory-static sort and memory-adaptive sort will sort all the data sets in memory. D4 represents a type of workload that contains large sorts that cannot be sorted in memory even with all available memory in the system. Both memory-static sort and memory-adaptive sort require external merging to finish the large sort jobs. D2 and D3 are cases between D1 and D4. Experiments over these cases will give us some idea of the behavior of the sort algorithms, even though they do not cover all possible situations.

Table 3.3: Sort job characteristics

Sort size distribution D0: for 1 sort					
Sort Size	10M				
Frequency	100%				
Sort Data Set D1: 100 sorts					
Sort Size	50K	600K	1M	2.5M	
Frequency	62%	27%	7%	4%	
Sort Data Set D2: 100 sorts					
Sort Size	60K	1M	3M	5M	10M
Frequency	10%	20%	60%	5%	5%
Sort Data Set D3: 100 sorts					
Sort Size	17K	380K	2M	7M	16M
Frequency	27%	35%	20%	7%	11%
Sort Data Set D4: 100 sorts					
Sort Size	60K	3M	5M	50M	100M
Frequency	10%	55%	30%	3%	2%

Chapter 4

Dynamic Memory Adjustment

Because of fluctuations in memory demand and unknown input size, sort jobs should have the capability to adjust their memory allocation during execution. This chapter begins with a discussion of memory-static sorts, then proposes a memory-adaptive mergesort, followed by details of the memory adjustment mechanism. The main part of the chapter is the design of a policy for memory adjustment. The goal is to reduce the number of external sorts by making better use of memory resources, thereby reducing sort elapsed time and improving system throughput.

4.1 Problems with Memory-Static Algorithms

A memory-static sort algorithm allocates memory space when a sort starts and keeps it fixed until the sort is finished. To prevent a sort from allocating too much memory in the system, there is frequently a fixed upper limit of memory space for a single sort. Memory-static algorithms may allocate memory space in several ways. If the input size is unknown and there is no estimate, it has to allocate memory space using some default size. If the input size is known or estimated, the sort can

allocate space according to this size and the memory limit of a single sort to reach the best sort performance.

Estimates of intermediate result size may be off by as much as one or two orders of magnitude [IC91]: either under-estimated in which case the estimated size is much lower than the real size, or over-estimated in which case the estimated size is much larger than the real size. Figure 4.1 shows the behavior of a static sort in four cases: no estimate, correct estimate, under-estimate, and over-estimate. For each case, there are two sorts: one is a small data set which can be sorted in memory, and the other is a large data set which has to be sorted using external merging by memory-static sort. In the diagram, the height indicates memory usage, while the width indicates sort time. The *single sort space limit* is the maximum memory space that can be allocated to one sort by the static sort algorithm. With dynamic memory adjustment, sort memory space will not be limited by this value. Solid lines shows the performance of static sort while dotted lines show how the performance can be improved by dynamic memory adjustment, either by reducing sort time, or by reducing memory space without affecting the sort time.

For small data sets, if there is no estimate or the input size is over-estimated, part of memory may be wasted during sorting (a, g). If the input size is under-estimated, and even though the data can be sorted in memory, an external merge may occur, which degrades the sort performance greatly (e). For large data sets, under-estimating the input size increases the risk of merging with multiple steps (f).

The memory usage of a static sort is always limited by the single sort space limit. If there is only one sort in the system, the extra space cannot be used by the sort because of this limit. However, the extra space may help to sort some large data sets in memory without the external merge phase, which will save a lot of sort

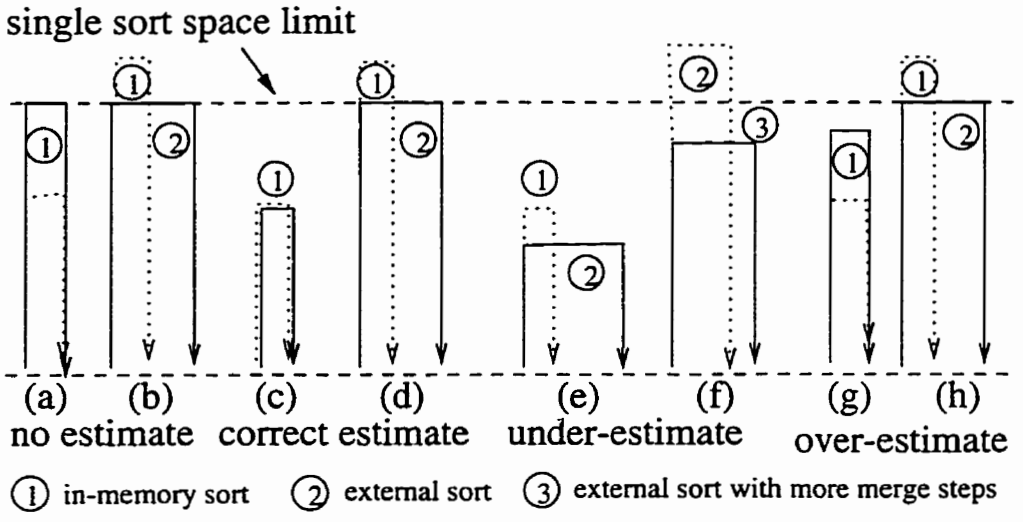


Figure 4.1: Problems with memory-static sort

time (b, d, h).

When the single sort limit is set high, more large sorts can be sorted in memory. However, small sorts will waste more memory space, and fewer sorts are allowed to run concurrently so that large sorts may block small sorts for a long time. This increases the average response time. When the limit is set low, small sorts will waste less memory and more sorts are able to run at the same time, but more large sorts may be sorted with external merge.

Some sorts used in commercial systems are able to adjust sort space during execution, but with very limited ability. For example, some sort algorithms are able to change sort memory space between the run formation phase and the external merge phase. A sort can use less memory for the external merge, but it cannot allocate more space. If the query optimizer provides an estimate of the input size, a sort can allocate memory space according to this size when the sort starts. Figure 4.2 shows the memory usage of a sort algorithm used in a major commercial

DBMS. The drop in memory usage during sorting is the start of the external merge phase.

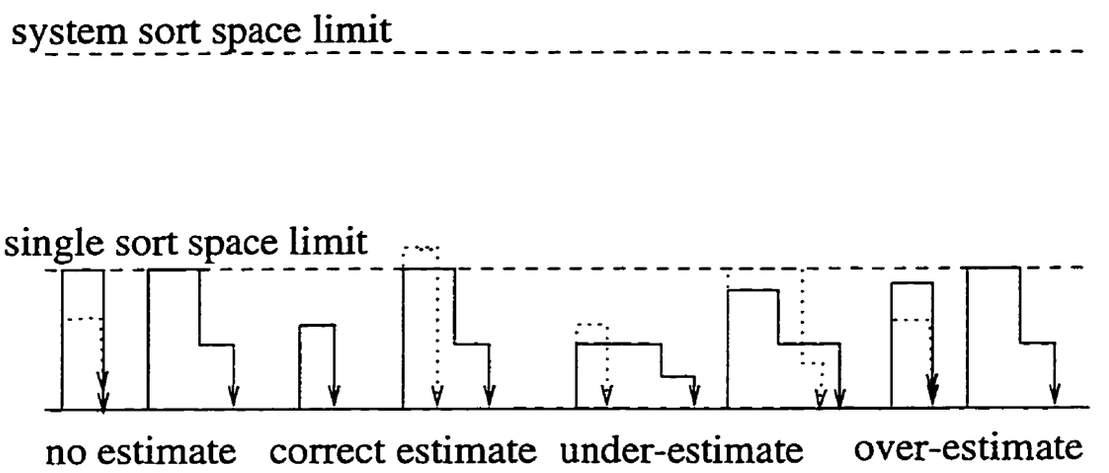


Figure 4.2: Memory usage of a sort in a commercial DBMS

There is a sort space limit at the database system level. It is defined by a system configuration parameter. Each sort is limited by the *single sort space limit*, which is also defined by a system configuration parameter. Since memory usage may change when the sort enters the external merge phase, the memory will be used more efficiently compared to the memory-static sort. However, it changes at most once during sorting and does not consider the memory requirement of other sorts in the system. All the problems of memory-static sort still exist: if the query optimizer does not provide an estimate or provides a poor estimate of the input size, sort performance will be affected; free memory in the system cannot be used to improve sort performance, since a sort cannot allocate more memory space than the single sort space limit.

4.2 A Memory-Adaptive Mergesort

We first need a sort that is able to adjust its memory usage during execution. Pseudo-code given below illustrates a memory-adaptive mergesort based on the sort algorithm introduced in Section 3.1. It shows at which points a sort is able to adjust its memory usage. This algorithm processes input data incrementally, making it possible for a sort to adapt its memory usage to both the actual input sizes and memory fluctuations. (Several places are labeled to be referenced in Section 4.4.2.)

Algorithm *memory-adaptive sort* :

```
// In-Buffer Sort Phase
while there is more input & memory space
    read data into a buffer
    sort the buffer
    [check/adjust memory] ----- (s1)
endloop
// In-Memory Merge Phase
if no more input & this is the first run
    merge buffers to produce output and stop ----- (s2)
if no more memory or this is the last run
    merge buffers
    write the sorted data into a tmp table
    if there is more input
        [check/adjust memory] ----- (s3)
        go to In-Buffer Sort Phase
// External Merge Phase
```

```
[check/adjust memory]
while max merge width < number of runs
    merge a number of shortest runs
    [check/adjust memory] ----- (s4)
merge runs to produce output  ----- (s5)
```

The data in each buffer can be sorted using any internal sorting algorithm. Therefore, many sort algorithms can be modified to produce memory-adaptive versions.

The basic idea of this adaptive sort is to increase memory usage when the system has extra space and additional space will speed up the sort, and to reduce memory usage when the system experiences memory shortage and some memory used by the sort is not critical to sort performance. How to adjust memory usage is implemented by the mechanism of memory adjustment, while the timing and amount of adjustment are determined by a memory adjustment policy. The details of the memory adjustment mechanism and policy are explained in the following two sections.

4.3 Memory Adjustment Mechanism

In-buffer sort phase

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

it proceeds to the in-memory merge phase.

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

In-memory merge phase

During an in-memory merge, the sorted data is written to a temporary file as a run. As buffers become empty, they can be either released (if the system is short of memory) or used for loading data for the next run. Whether a buffer is to be released or kept is a policy decision. It is not necessary to increase memory space during this phase.

External merge phase

The exact number of runs and amount of data are known when a sort enters this phase. The sort structure is changed from the data structure for run formation (in-memory sort/merge) to the data structure for external merge. If the number of runs is small, we attempt to allocate enough memory to complete the sort with a single merge step.

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

Memory usage can also be adjusted by changing the size of input buffers and, thereby, the merge fan-in. Increasing the buffer size reduces disk overhead (total seek time and latency) because fewer I/O requests are needed to transfer the same

amount of data. However, this option is not considered in our implementation; we always use a fixed buffer size. Normally, we use 32Kb buffers because we experimentally found that increasing the buffer size further yields only marginal benefits.

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

Wait queues

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

4.4 Memory Adjustment Policy

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

A memory adjustment policy needs some system wide state information, including the number of active sorts, the amount of free memory in the system, the stage of each sort, etc. It also relies on a set of predefined parameters such as memory

adjustment bounds. The objective is to improve system performance (throughput and response time) while at the same time ensuring fair treatment of competing sorts.

4.4.1 System sort space

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

In our adaptive sort two configuration parameters determine total sort space and memory allocation: $SysSortSpace$ and $MemUnit$. $SysSortSpace$ is the limit on total memory space available for sorts. $MemUnit$ is the size of one data buffer plus related sort structures. A sort allocates memory one $MemUnit$ at a time. The value of $SysSortSpace$ is based on the total memory size, while $MemUnit$ is used to tune sort performance. If a system does many small sorts, setting $MemUnit$ low will make use of memory space more efficiently and memory adjustment is more flexible. On the other hand, if a system usually does large sorts, setting $MemUnit$ high will reduce the allocation and deallocation cost, but some memory may be wasted.

4.4.2 Sort stages

For the purpose of memory adjustment, we consider a sort to be in one of seven different stages as listed in Table 4.1. The stages correspond to the sort phases and the specified places (s1 to s5) indicated in the algorithm given in Section 4.2.

Table 4.1: Sort stages

stage	*sort phase*	*explanation*
0		waiting to start
1	in-buffer sort (s1)	first run
2	in-memory merge (s2)	final result
3	in-buffer sort (s1)	remaining runs
4	in-memory merge (s3)	produce runs
5	external merge (s4)	intermediate merge
6	external merge (s5)	final merge

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

Stage 1: The sort is processing the first run during the in-memory sort phase (at s1). It is not known yet if the input will fit completely in memory. Giving a sort in this stage additional memory may be very beneficial if it results in the input being sorted completely in memory.

Stage 2: All input data has been loaded into memory and the sort is in the in-memory merge phase (at s2), i.e., the sort has enough space for an in-memory sort. A sort in this stage is unable to reduce its memory usage. On the other hand, extra memory will not improve the performance of the sort.

Stage 3: The sort is processing the remaining runs during in-memory sort phases (at s1). At this stage it is known that an external merge is necessary. All data will be written to disk and then read during merge again. Additional memory helps to reduce the number of runs, which may reduce the number of external merge steps. If a single merge step is sufficient, extra memory can be used to reduce the disk seeks (see next chapter). However, if this memory space helps some other sort in the system to be done entirely in memory, the total I/O cost of that sort for writing and reading runs is avoided. Thus memory space is less critical to a sort in this stage than it is to a sort in stage 1 or stage 5.

Stage 4: The sort is processing the runs during in-memory merge phases (at s3). Similar to Stage 3, it is known that external merging is necessary.

Stage 5: The number of runs could not be merged in a single step and the sort is performing intermediate merges during this stage (at s4). It checks the available memory before each merge step and adjusts the fan-in accordingly. When there is enough memory to merge all remaining runs in one step, the sort allocates enough space, and immediately goes to the last merge step. Since extra memory will help reduce the amount of I/O, additional memory is very important to a sort in this stage.

Stage 6: The sort merges all remaining runs producing the final output (at s5). Since the amount of data is known at the start of the merge step, the sort is able to allocate exactly the amount of memory needed. One page less of the memory will result in another merge step.

Based on the above analysis, we decided on the following priorities:

1. memory requirements of sorts in stage 0 have the highest priority,

2. memory requirements of sorts in stage 1 or stage 5 have the next highest

priority,

3. sorts in stage 3 can benefit from more memory (by reducing the number of runs) but yield to sorts in stage 1 or stage 5,

4. sorts in stage 4 have the lowest priority,

5. sorts in stage 2 or stage 6 do not change their memory usage.

4.4.3 Memory adjustment bounds

We do not allow a sort to increase or decrease its work space arbitrarily but restrict the size to be within a specified range. The range depends on what stage the sort is in and on the number of active sorts. The main purpose of this restriction is to prevent a sort from monopolizing resources, thereby starving other sorts running at the same time or arriving later. The lower bounds prevent sorts from attempting to run with too few resources. Figure 4.3 illustrates these memory bounds.

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

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

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

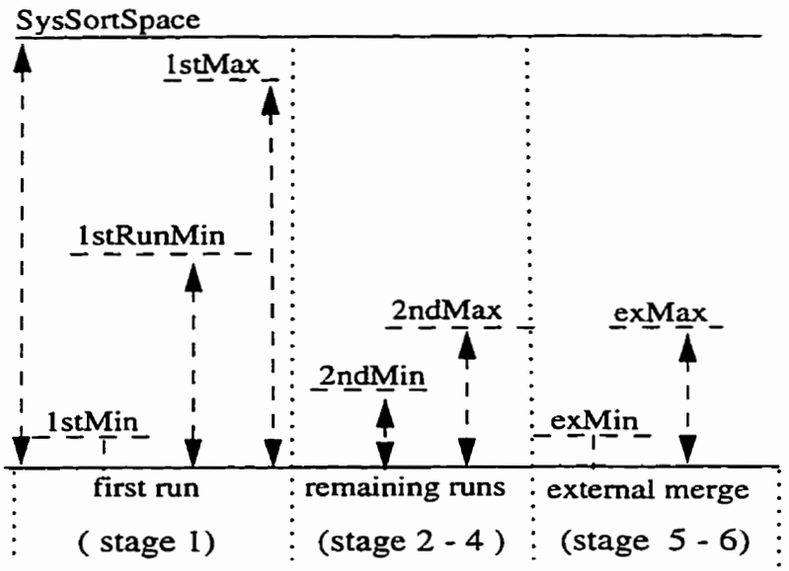


Figure 4.3: Sort memory usage bounds

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

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

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

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

The lower bounds are usually fixed based on system configuration, while the upper bounds depend on total amount of free memory and workload in the system. Table 4.2 list the values used for our experiments, where $freeMem$ is the amount of free memory space (i.e. available memory) for sorting in the system, and $evenShareMem$ is the total sort space size divided by the number of active sorts in the system. Both of them change dynamically as the workload changes.

Table 4.2: Default values for memory usage bounds

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

4.4.4 Waiting

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

In our system, a sort is allowed to wait only if it has not reached the upper bound on memory for its current stage (1stRunMin, 2ndMax, or exMax). Otherwise, it will proceed with the memory it has acquired. A sort may wait in one of five situations:

W1: in stage 0 waiting to start;

W2: in stage 1 with 1stMin space;

W3: in stage 1 with more memory;

W4: in stage 3;

W5: before an external merge step.

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

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

4.4.5 Fairness

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

reflected in our policy:

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

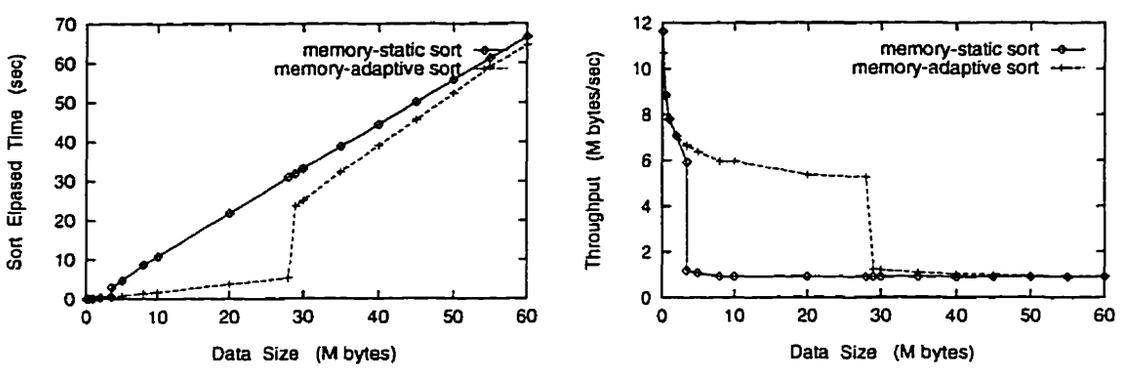
These considerations are addressed by the incremental sorting mechanism, memory priority of sort stages, multi-level priority waiting queues, first-come-first-serve policy for the sorts within each waiting queue, and round-robin scheduling policy for active sort agents.

4.5 Experimental Results

4.5.1 Single sort performance

When there is only one active sort in the system (the single sort case), a static sort is limited by the single sort space limit, while our adaptive sort is able to employ much more space available in the system. If this limit is the same as the system sort space size, sort jobs are not allowed to run concurrently. A small sort will waste memory space, while a large sort may block the following sort jobs for a long time.

Figure 4.4 shows the observed elapsed time of a single sort as a function of input size and the corresponding throughput measured as the amount of sorted data produced per second. Static sort changes from in-memory sort to external sort at an input size of 3,585 Kb, while the adaptive sort changes at an input size of 29 Mb.



(a) Sort elapsed time (b) Sort throughput

Figure 4.4: Single sort performance

For input less than 3585 Kb, both adaptive sort and static sort finish the sort entirely in memory and have the same elapsed time and throughput. For medium size input (3585 Kb - 29 Mb), static sort relies on external merging, while adaptive sort can sort the data completely in memory. The difference in throughput is dramatic, dropping from about 6 Mb/s to slightly over 1 Mb/s. One of the main objectives of memory-adaptive sort is to exploit this difference by trying to complete as many sorts as possible in memory.

Adaptive sort performs slightly better for large inputs (over 29 Mb). The reason is that adaptive sort produces a large run followed by a set of small runs. The run blocks required for external merge are more often from the first run than from other

runs, which reduces the disk seek time. When the input size grows and the number of runs increases, the elapsed time and throughput of the two sort algorithms slowly converge.

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

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

4.5.2 Concurrent sorts

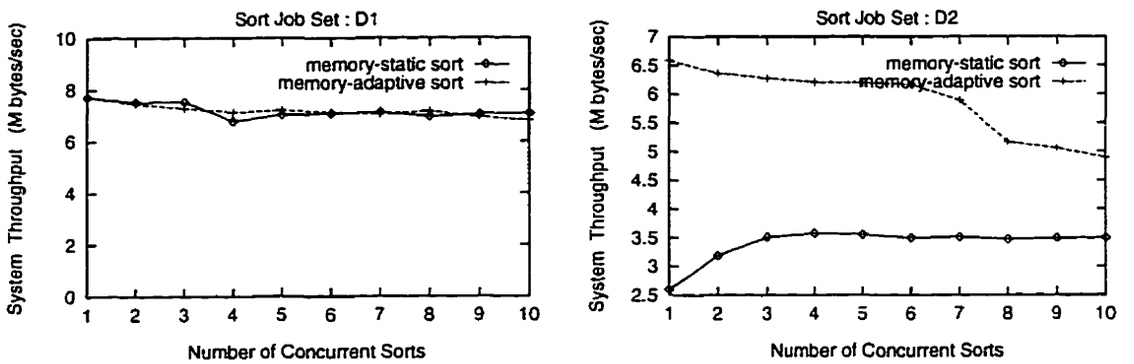
A database system does not have the luxury of running only one sort at a time. Many sorts may be running concurrently, competing for memory and I/O resources. This section reports on experiments investigating the effects of memory adjustment on (sort) system throughput and response time when multiple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

The workload for each experiment consisted of a sequence of sort jobs of varying size. There were 100 sort jobs in each experiment run. The input size of a sort job was randomly drawn from a specified sort size distribution (D1 to D4) given in Section 3.7.2 Table 3.3. For each sort job set, experiments were conducted on concurrency degree 1 to 12 (or the maximum concurrency defined). Memory-static sort and memory adaptive sort were tested in the same conditions.

Unrestricted concurrency

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

Figure 4.5 and Figure 4.6 show the sorted data throughput as a function of the number of sorts running concurrently for workload based on sort size distributions D1 to D4.

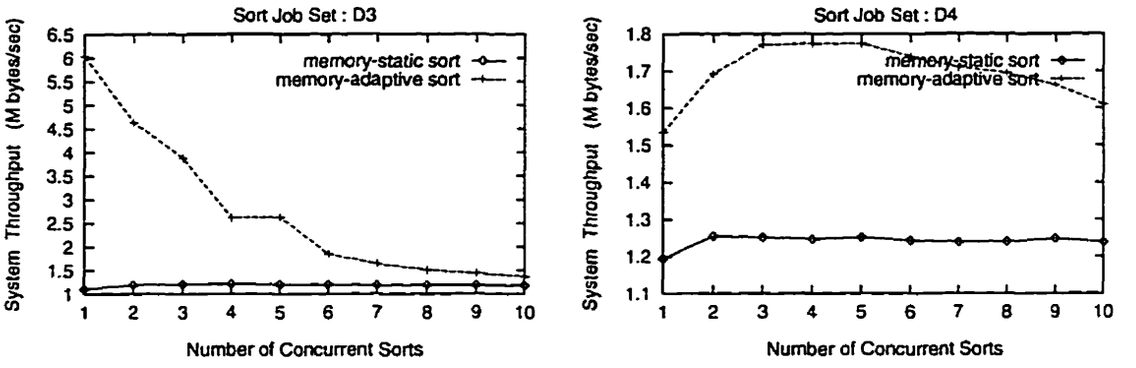


(a) Throughput of D1

(b) Throughput of D2

Figure 4.5: Concurrent sorts performance (D1, D2)

All sorts in D1 are small enough to always be sorted in memory, even with 12 sorts running concurrently. In this case the system is completely CPU bound. Figure 4.5 (a) shows that the two sort methods achieve about the same throughput, which confirms that the overhead of dynamic memory adjustment is minimal. As the number of concurrent sorts increases, throughput decreases only slightly. This is a result of more frequent thread switching which (probably) also results in poorer



(a) Throughput of D3 (b) Throughput of D4

Figure 4.6: Concurrent sorts performance (D3, D4)

cache performance.

For the other three workloads, memory-adaptive sort has significantly higher throughput when the number of concurrent sorts is low (see Figures 4.5 (b) and 4.6). In the best case, the throughput is up to 6 times higher. The difference decreases as the number of concurrent sorts increases because of the increased competition for memory and I/O bandwidth. This shows that memory-adaptive sort works in the sense that, when possible, it exploits available memory to speed up sort jobs and gracefully degrades when the competition for memory space increases.

Only workload D4, see Figure 4.6 (b), shows increased throughput as the number of concurrent sorts increases (up to 4). The few large sorts in this workload are completely I/O bound, leaving free CPU cycles that will only be used (by small sorts) when there are enough sorts active at the same time.

An important objective of memory-adaptive sort is to reduce the number of external sorts. Table 4.3 shows that, when memory space is available, all but the largest sorts are completed entirely in memory. When many sorts run concurrently,

less memory is available for each sort so fewer sorts can be completed in memory and the load on the I/O system increases. This effect accounts for most of the decrease in throughput.

Table 4.3: Number of external sorts (out of 100 sorts)

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

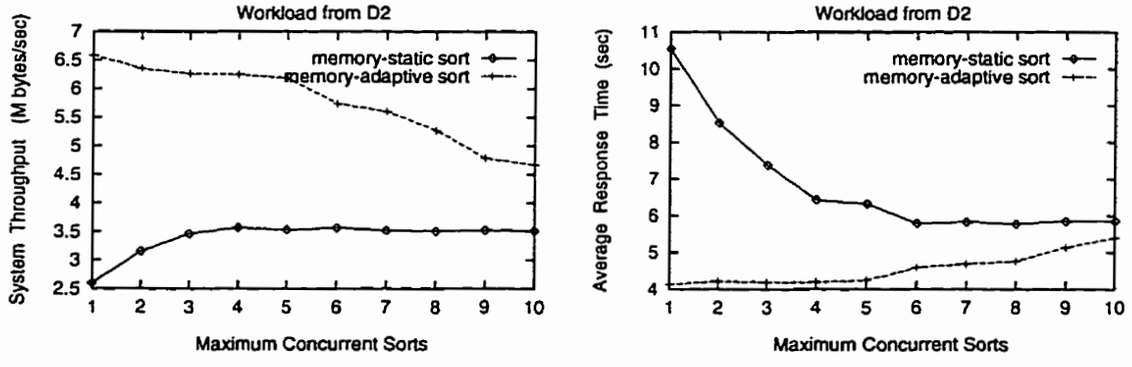
(ma: adaptive sort; st: static sort)

Limiting concurrency

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

In these experiments we had 10 clients repeatedly submitting sort jobs. As soon as a client's previous job finished, it submitted another sort job. In other words, there were always 10 outstanding sort jobs, some being processed and some waiting to start. We then varied the number of sorts being processed concurrently, i.e., the maximum concurrency, and measured throughput and response time. Response time is the average time from when a client submitted a request until the last record in the output arrived.

Figures 4.7 to 4.9 show the throughput and average response time for D2, D3, D4 as the limit on concurrent sorts varies. (Limiting the number on concurrent sorts has no effect on D1 because the sorts are so small.) In all cases, except for D1, memory-adaptive sort achieves both better throughput and response time than static sort.

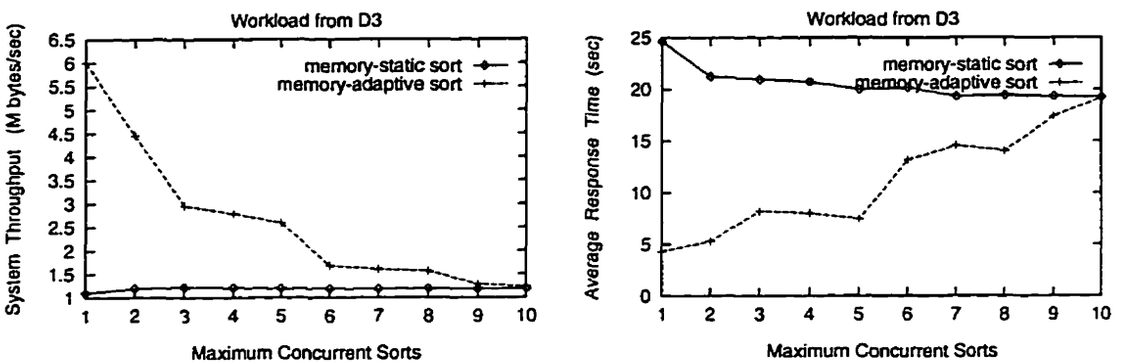


(a) Throughput of D2

(b) Average response time of D2

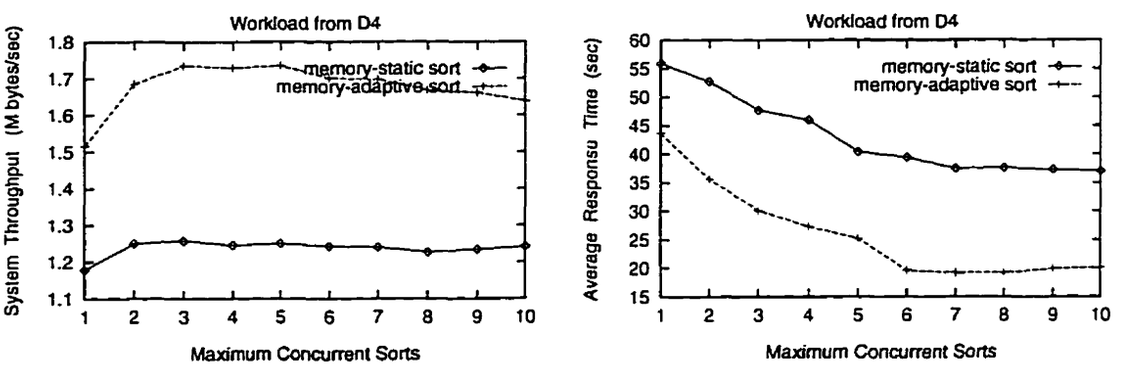
Figure 4.7: System performance of D2

The graphs are best read from right to left. The results for D2 and D3 are very similar because all sorts in these job sets are less then 32 Mb and, hence, can be sorted entirely in memory if run in isolation. As the number of sorts being processed



(a) Throughput of D3 (b) Average response time of D3

Figure 4.8: System performance of D3



(a) Throughput of D4 (b) Average response time of D4

Figure 4.9: System performance of D4

concurrently is decreased, both throughput and average response time improve for memory-adaptive sorts as more and more of the sorts are done in memory. The reverse is true for static sort but the effects of limiting concurrency are much less pronounced.

D4 contains a few large sorts that cannot be completed in memory. In the time it takes to complete a 100 Mb sort, about 24 (6x4) sorts of size 25 Mb can be completed (assuming they can be done in memory). So in this case, processing only one sort at a time is clearly not a good idea. This effect is also visible in the graphs. Figure 4.9 (a) shows that throughput initially increases as the limit on concurrent sorts decreases but then starts dropping (because CPU and memory resources are not fully utilized). Response time, see Figure 4.9 (b), increases steadily as fewer sorts are processed concurrently.

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

4.6 Summary

This chapter proposed a dynamic memory adjustment mechanism and policy based on the three-phase mergesort algorithm introduced in Chapter 3. The technique enables sorts to adapt their memory usage to the actual input size and fluctuations of available memory space.

Our study focused on a memory adjustment policy that takes into account system sort space, sort stages, memory adjustment bounds, waiting, and fairness. The policy balances memory usage among concurrent sorts so that more sort jobs are done entirely in main memory, which improves the overall system (sort) performance. Experimental results showed that sort throughput was improved significantly compared with static memory allocation.

Chapter 5

Read Ahead during External Merge

The purpose of read ahead is usually to overlap CPU and I/O operations. It seems that extra buffers will not help improve performance, once full overlap has been achieved. However, Zheng and Larson [ZL96b] showed that extra buffers can be exploited to reduce disk seek time.

Modern disks have become increasingly complex. Most disk drives have multiple zones, with each zone having different numbers of sectors per track [RW94]. Disk caches also have a great impact on disk performance. Some data may be read from the disk cache rather than the disk. There will be no disk seeks in this case and the transfer rate is much higher. So it is very difficult to calculate the exact cost of each disk access, but two facts remain unchanged:

1. disk seek time and rotational latency still heavily affect the total disk access time for random reads/writes; and

2. sequential access is much faster than random access.

This chapter focuses on improving I/O performance by reducing the number of disk *seeks*. Three read-ahead strategies are considered: fixed buffering, extended forecasting, and clustering. When multiple jobs access the same disk, improvements from simple clustering degrades as disk contention increases. An improved clustering algorithm, called clustering with atomic reads, helps retain performance when disk contention is high. An analysis of these methods results in a set of formulas to estimate the performance improvement, and the accuracy of the estimates is confirmed by experimental results.

Throughout this chapter, we assume that all data blocks of a run are stored contiguously on disk[1]. When several blocks of the same run are adjacent, it is assumed that only one disk seek is required if these blocks are read continuously, even with several read requests. In the sort testbed, this is accomplished by having two I/O agents for each disk. The disk is kept busy and there is little waiting time between the requests. We also assumed that memory for external merging is fixed within each merge step, but it is adjustable between merge steps. So for each merge step, a sort is able to plan for reading in advance based on the given memory size.

This chapter is organized as follows. Section 5.1 describes three types of read-ahead strategies: fixed buffering, extended forecasting, and (simple) clustering. Section 5.2 discusses the problem introduced by concurrent jobs and gives our solution — clustering with atomic reads. Section 5.3 studies the performance of these read-ahead strategies on partially sorted input. Formulas are derived in Section 5.4 for estimating the performance improvement resulting from these strategies.

[1]In the sort testbed, runs are stored in a raw partition of the disk. Normally a partition is a large chunk of contiguous space on disk. This space is managed by the disk manager of the sort testbed, which allocates a contiguous space for a run before the in-memory merge starts, since the run length is already known at this stage. This guarantees that run blocks from multiple sorts will not be mixed on disk.

Section 5.5 studies the the problem of how to choose buffer size when using these strategies, and the last section summarizes this chapter.

5.1 Strategies for Read Ahead

5.1.1 Fixed buffering

Fixed buffering assigns all buffers to runs before a merge step starts. Each buffer is dedicated to a run until the merge step is finished. Buffers can be assigned to runs in many ways. The minimum requirement is that each run must have at least one buffer. These buffers are called *merge buffers*, while others are called *read ahead buffers*. Whenever a merge buffer is empty, it becomes a read ahead buffer, while a read ahead buffer with the next block for that run becomes the merge buffer.

Equal buffering assigns the same number of buffers to each run. *Double buffering* is a special case of this method in which each run has two buffers. Double buffering achieves full overlap of CPU and I/O operations if the process is constantly I/O-bound or constantly CPU-bound.

For random data, the next block to be read is normally from a run different from the run that the latest block was read. So each read requires a disk seek. With more than one read ahead buffer for each run, it is possible to read several (contiguous) blocks with one disk seek. Suppose each run has S buffers, one buffer for merging and $S-1$ buffers for read ahead. If a sort sends a read request whenever a buffer becomes empty, reading still jumps across runs. A better approach is to read $S-1$ blocks from a run when all its buffers but one become empty. So only one disk seek is required for every $S-1$ blocks. The total number of disk seeks is then reduced by a factor of $(S-1)$.

To overlap processing and read time, fixed buffering requires at least twice as many buffers as the number of runs, although it can proceed with fewer buffers. With more buffers, it is able to reduce disk seeks. The problem is that fixed buffering does not fully utilize the buffer space. When $S-1$ buffers become empty, the sort cannot issue read requests for these buffers unless they belong to the same run. If data is not uniformly distributed or the runs are not in equal size, some buffers may stay unused for a long time. For example, when some runs finish much earlier than other runs, their buffer space will be unused until the end of the merge. A simple improvement is to reallocate these buffers to other runs (but this is not strictly fixed buffering any more). Another way is to allocate the buffers proportional to the run length before merge starts. Long runs get more buffers. However, short runs do not necessarily finish earlier. To use buffer space more efficiently, buffers should not be dedicated to a specific run, but serve any run on demand. This is *floating buffering* in contrast to fixed buffering.

5.1.2 Extended forecasting

In this section, we extend the standard forecasting read-ahead strategy and discuss two methods of merging.

Forecasting uses floating buffers, i.e., buffers are not dedicated to specific runs. Traditionally, forecasting uses one extra buffer for read ahead [Knu73]. When one block from each run resides in memory, it can be determined which buffer will be emptied first by comparing the last keys in the buffers. The extra buffer is used for reading the next block from that run.

Because of data distribution and variation of system work load, a merging process may not produce empty buffers at a constant rate. After reading the extra buffer, there may not be any empty buffers. Even if merging is fast enough to pro-

duce empty buffers all the time, the next block to be read is determined on the fly. The disk always has a wait time after finishing each read request. Thus processing and reading are not fully overlapped. To maintain high utilization of the disk, it is necessary to keep one read request in the I/O queue. To achieve this, we must (a) have more than one extra buffer and (b) know which block to read in advance. *Extended forecasting* is an extension of traditional forecasting, which reads the run blocks based on a pre-determined run block sequence and is able to use additional buffers for read ahead. It achieves better overlapping of processing and read time.

The order in which run data blocks are consumed by merging is called the *consumption sequence.* It depends on the external merge algorithm. The standard merge algorithm requires the next block of a run whenever the merge buffer of that run becomes empty. So the next block required depends on when the previous block of the run is finished. The consumption sequence is based on the last key of each run block and can be computed by simulating the merge process.

In the standard merge algorithm, the run block required for merging in fact may not be used immediately. As an example, if the input data is sorted (or reversely sorted), at any time only one merge buffer is really needed while all other merge buffers stay unused. For each run, reading its data blocks can be delayed until all the runs with smaller keys have finished. In general, the read of any block can be delayed until all other blocks with keys smaller than the first key of the block have been read.

Based on this observation, we designed a new merge variant, called *merging with delayed reads*: whenever a buffer becomes empty, the next block required is the one which has the smallest first key among all runs on disk. If the next block to be read is from run X and the merge buffer of run X has not been emptied yet, the merge process is able to proceed while some other runs may not have merge buffers. Those

runs without merge buffers are not involved in merging until their next blocks are read into memory, and there are no key comparisons between these runs and their sibling runs. So the algorithm reduces the number of key comparisons, which is usually the major cost of CPU time for sorting in database applications.

For merging with delayed reads, the read order is determined by the first key of each run block. The consumption sequence can be computed simply by sorting the first keys of the run blocks.

Figure 5.1 shows an example with 3 runs, each with 3 blocks. The block numbers reflect the order of the blocks written (adjacently) on disk. The first key and the last key of each run block are given in the diagram.

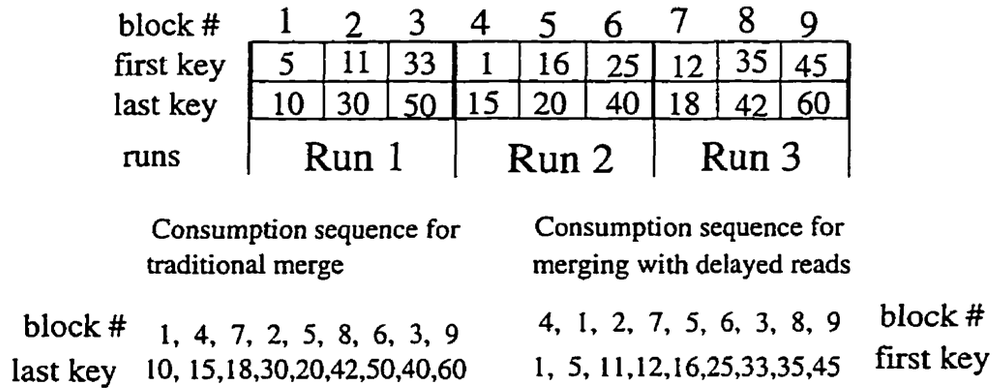


Figure 5.1: Consumption sequences

The consumption sequence for standard merging is based on the last key of each data block. Initially, the first block of each run is required to start the merge process. Block 1 is finished first since it has the smallest last key. The next block of run 1 (block 2) is then required. Then block 4 is finished. The next block to be read is the next block of run 2 (block 5), and so on, resulting in the consumption sequence at the left in the diagram.

The consumption sequence for merging with delayed reads is based on the first keys of the run blocks. The consumption sequence is computed by sorting the first keys as shown at the right in the diagram. During merging, after the first two blocks are read into memory, the merge process can start, since the next block to be read (block 2) is from the same run of the previous block (block 1). Run 3 is not involved in merging until block 1 is finished and block 7 is read into memory. It shows that merging with delayed reads groups more adjacent blocks together than the standard merge algorithm.

For both merging algorithms, once the consumption sequence is determined, extended forecasting reads the run blocks in that order. Extra buffers help improve the overlap of CPU and I/O operations, but cannot reduce disk seeks. However, if the extra buffer space is used to increase the buffer size instead of increasing the number of buffers, disk seeks can be reduced, since large buffers reduce the number of read requests, resulting in fewer disk seeks. Section 5.5 will study the effect of buffer size for some read strategies. Before that section, we assume that buffer size, i.e., I/O transfer unit size, is fixed. We focus on how the number of disk seeks is affected by the number of buffers.

In summary, traditional forecasting uses one extra buffer for read ahead. Merging does not rely on the consumption sequence, but processing and reading may not be fully overlapped. Extended forecasting employs additional buffers to achieve better overlap of CPU and I/O time, but merging relies on a pre-computed consumption sequence. The consumption sequence depends on the merging algorithm. For the standard merge algorithm, the consumption sequence is determined by the last key of each run block, while for merging with delayed reads, the consumption sequence is determined by the first key of each run block. The merging with delayed reads may save some of the key comparisons. For both merging algorithms,

additional buffers improve the overlap of CPU and I/O time, but cannot reduce disk seeks unless the extra space is used to increase the buffer size (I/O unit size).

5.1.3 Simple clustering

Although a merge process consumes run blocks in a particular order, the run blocks can be read in a different order if extra buffers are available. The extra buffers can be used for storing run blocks which are not required immediately but which can be read with less I/O cost (i.e., disk seek time). The sequence in which blocks are read from disk is called the *read sequence*. Let $C = \{C_1, C_2, ..., C_T\}$ be a consumption sequence, where each C_i is a run block. A read sequence $R = \{R_1, R_2, ..., R_T\}$ is a permutation of $\{C_1, C_2, ..., C_T\}$. Throughout this chapter, for any two sequences A and B, we define $A \subseteq B$ to mean that the set of elements in A is a subset of those in B.

Not all read sequences are useful for merging. Some may result in deadlock between merging and reading. For example, given 10 runs, each with 100 data blocks, and a total of 50 buffers, if the last 5 blocks of each run are read at the beginning in the read sequence, no buffers are left to read the first block of each run, so the merge process cannot proceed. A read sequence is feasible if it guarantees that the merge process terminates. Zheng and Larson [ZL96b] introduced following condition to check the feasibility of a read sequence:

Proposition 5.1 *A read sequence $\{R_1, R_2, ..., R_T\}$ is feasible for consumption sequence $\{C_1, C_2, ..., C_T\}$, if $\{C_1, C_2, ..., C_{k-B+n}\} \subseteq \{R_1, R_2, ..., R_k\}$ for all k such that $B \leq k \leq T$, where B is the number of buffers and n is the number of runs.*

To overlap processing and read time, Estivill-Castro and Wood [ECW94] suggested to have one buffer reserved for reading at all times. Therefore the condition

becomes $\{C_1, C_2, ..., C_{k-B+n}\} \subseteq \{R_1, R_2, ..., R_{k-1}\}$.

Figure 5.2 illustrates the idea with one buffer reserved to overlap the merge processing and read time. At any stage of the merge process when $k \geq B$, $k-1$ blocks have been read into memory, which includes C_1 to C_{k-B+n}. Among them $k-B$ blocks have been consumed by the merge process, while n blocks, one for each run, are being merged. The condition guarantees that merging can proceed while R_k is being read into memory. If merging is fast enough to provide empty buffers before the read of R_k is finished, processing and read time are fully overlapped.

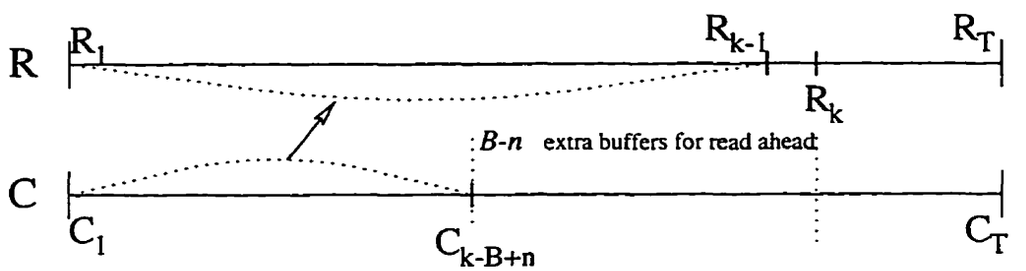


Figure 5.2: Feasibility of read sequence

Proposition 5.1 is based on the standard merge algorithm. With delayed reads, the number of runs involved in merging may be smaller than n. So more than $B-n$ buffers may be available for read ahead. Thus the condition in Proposition 5.1 is sufficient, but not necessary for merging delayed reads.

It is obvious that the consumption sequence is a feasible read sequence if $B > n$ (with at least one buffer for read ahead to overlap merging and reading). It guarantees that the merge process terminates. In fact, the consumption sequence is the read sequence for both traditional forecasting and extended forecasting.

Although there is a finite number of feasible read sequences, it is not known if there is an efficient algorithm to find the optimum sequence with minimum disk seek

time. Finding the optimum sequence by trying all the read sequences is expensive[2]. Research has been focusing on using heuristics.

The following algorithm, which we call *simple clustering*, is a heuristic algorithm for finding a "good" read sequence. The initial read sequence is the consumption sequence for the standard merge algorithm. The first n blocks are the first blocks of the n runs. Beginning from the $(n+1)th$ block in the read sequence, each block is combined with the previous block of the same run if the feasibility of the read sequence is preserved.

Algorithm *simple clustering*
Input: consumption sequence $C = \{C_1, C_2, ..., C_T\}$,
number of buffers B, number of runs n
Output: read sequence $R = \{R_1, R_2, ..., R_T\}$
// C_i *and* R_i *have the same structure: run number field and block address field*
begin
$R := C$; // *Initialize read sequence to be the consumption sequence*
for $i := n + 1$ **to** T
// *Search each previous block to find the one with the same run as R[i]*
for $j := i - 1$ **downto** 1
if $R[j].runNumber = R[i].runNumber$
then exit loop; endif;
endfor;

[2]Here is an example which gives a rough idea on how expensive it is to find an optimum read sequence. A brute-force algorithm implemented on the sort testbed searched all feasible read sequences to find the best one. It took more than 10 hours to get the optimum sequence for 3 runs with a total of 24 blocks. However, using a heuristic algorithm, it took 5 ms to compute a feasible read sequence for a 50M data set with 15 runs and 1600 blocks. The generated read sequence reduced disk seek time by 12 seconds, which is 80% of the disk seek time if the data blocks are read in the consumption sequence.

if $R[i]$ can be moved after $R[j]$ preserving feasibility
then insert $R[i]$ between $R[j]$ and $R[j+1]$; **endif**;
endfor;
end

The algorithm tries to cluster together as many blocks from the same run as possible while preserving feasibility. Each group, called a *cluster*, is a sequence of adjacent blocks from the same run. Since the blocks in a cluster are adjacent, they can be read sequentially which avoids disk seeks, and therefore reduces the total read time. In this algorithm, a cluster is not read with a single read command. It is still read one block at a time, each block using one read command. When the disk drive processes the read requests of a cluster continuously, these adjacent blocks in a cluster will be read sequentially.

To check the feasibility efficiently, we used a free buffer count array $F = \{F_1, F_2, ..., F_T\}$ in our implementation, where F_i is a nonnegative integer which records how many free buffers will be left after R_i is read. The initial value of F_i is $B - n - 1$. ($i = 1$ to T), with 1 buffer reserved for overlapping merging and reading. When a data block R_i is moved after block R_j ($j < i$), F_{j+2} to F_i are reduced by one, since one free buffer is used for reading a data block (R_i) before it is required. To guarantee a feasible read sequence, we need only to keep all F values nonnegative, i.e., a block should not be inserted before a data block which F value is 0. Therefore, for each block R_i, the algorithm needs only check the block R_{i-1} down to the first block R_j where $F_j = 0$. With F values correctly maintained, we can guarantee the feasibility of the read sequence. This method is very efficient compared to using Proposition 5.1 directly to check the feasibility.

The simple clustering algorithm is similar to the *group-shifting* algorithm pro-

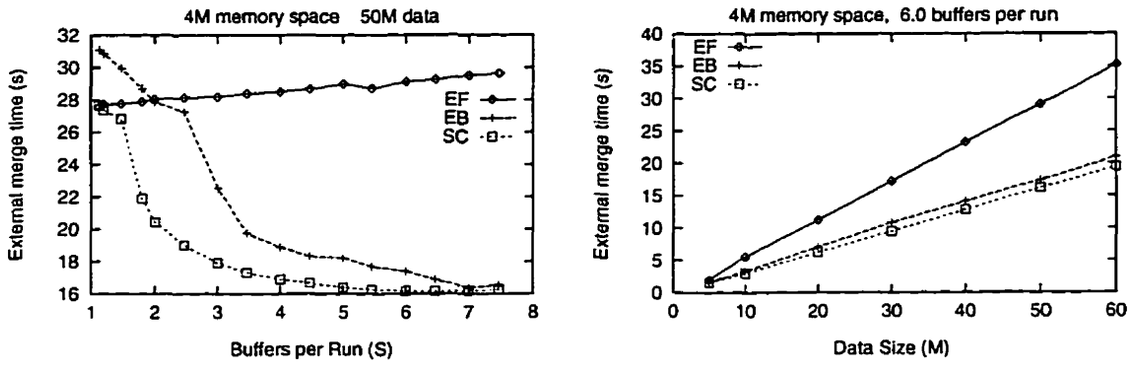
posed by Estivill-Castro and Wood [ECW94]. However, the two algorithms were designed independently. Compared to group-shifting, our algorithm is much simpler and requires fewer scans of the consumption sequence. Experimental results (see Section 5.1.3) show that for random data, the simple clustering algorithm reduces disk seeks as well as the group-shifting algorithm.

Several other algorithms were developed and tested on the sort testbed, including *closest logical distance*, *double clustering*, and a method achieving similar effect of the backward movement in the group-shifting algorithm, but the additional performance improvement is small compared to the simple clustering algorithm. So those methods were not investigated further.

Experimental results

Experiments were run on many data sets. For each data set, each read-ahead strategy was tested, and the experiment was repeated using the minimum number of buffers up to the maximum number of buffers allowed within the memory space limit. The minimum number of buffers is the number of runs plus two. The two extra buffers are used to improve disk utilization by keeping a read request in the I/O queue. Each point plotted in the diagrams represents the average computed from five experiments. The five experiments used five data sets of the same size that were produced using different random seeds. The three-phase external mergesort algorithm proposed in Section 3.1 was used for producing runs and merging. However, dynamic memory adjustment was not used in these experiments. Throughout the experiments, buffer size was 32 K bytes for all read strategies. It was also the run data block size and I/O transfer unit size. For extended forecasting, when the merge phase was started, we sent a read request for each buffer according to the consumption sequence. During merging, as soon as one buffer became empty,

we sent a read request for the next block in the consumption sequence. Simple clustering worked in the same way, but using a read sequence pre-computed from the clustering algorithm. For equal buffering, instead of issuing a read request as soon as a buffer became empty, we sent a set of read requests for a run when all its buffers but one became empty.



(a) Effect of merge buffers (b) Effect on data size

(EF: extended forecasting; EB: equal buffering; SC: simple clustering)

Figure 5.3: Comparison of read-ahead strategies

Figure 5.3 (a) shows the average results for five 50M data sets. The experiments using other data sizes (5M to 100M) produced similar results. Figure 5.3 (b) shows the results of using 6 buffers per run while data size changes from 5M to 60M (the memory limit is not enough to provide 6 buffers per run for larger data set). The figures prompt the following observations:

1. When the number of buffers is less than twice the number of runs, equal buffering performs the worst. The reason is that some runs have two buffers while others have only one buffer. Processing and reading are not fully overlapped.

2. When the number of buffers is more than twice the number of runs, we can reduce disk seeks by using equal buffering. If the number of buffers is over three times the number of runs, a significant amount of merge time is reduced. With seven buffers per run, the merge time is close to the lowest value.

3. Extended forecasting does not benefit from extra number of buffers. Merging is very fast in our experiments so that full overlap of processing and reading is achieved by using two extra buffers. Disk seeks are not reduced by the extra number of buffers assume that the buffer size (I/O unit size) is fixed. In fact performance tends to degrade as the number of buffers increases. The reason is not known, though.

4. Simple clustering makes full use of floating buffers and reduces the disk seek time even with a small number of extra buffers. With twice as many buffers as the number of runs, the merge time is already reduced significantly. Merge time is close to the lowest value using as little as five buffers per run. For all cases, simple clustering outperforms the other strategies.

5.2 Read Ahead for Concurrent Jobs

All previous research on computing read sequences ([ZL96b] and [ECW94]) as well as the discussion in the previous section assumed that only one sort runs in the system. No other jobs access the run disk when the sort is doing an external merge. In reality, a disk drive serves many jobs in the system. Several queries may access the run disk(s) at the same time. Reading the run blocks in a cluster may require more than one disk seek if: (1) while a sort is doing an external merge, other jobs, such as joins, access the same disk, (2) multiple external sorts are running

concurrently and their run blocks are read/written on the same disk, (3) sorted output is written to the same disk where the input runs reside, or (4) a sort merges the runs in multiple steps and stores intermediate runs on the same disk. Many people have suggested using different disks for intermediate runs and input runs so that run read requests and run write requests will not be mixed with each other, but it does not solve the problem when there are multiple sorts accessing the same disks at the same time or when the optimum merge pattern is adopted for merging (see Section 5.2.1).

The clustering technique given in the previous section groups data blocks into clusters expecting that the blocks in each cluster are read sequentially. When a sort's read requests are mixed with I/O requests from other jobs or its own write requests (for intermediate runs or sorted output), a cluster may be broken into several pieces requiring more than one disk seek. Experimental results indicate that with moderate disk disturbance, the simple clustering algorithm still works fine. But when disk contention is high, for example in the extreme case when there is an I/O request from other jobs after each run read request, clustering will not save any disk seeks.

This problem can be solved by *atomic cluster reading*, where an atomic cluster read (or an *atomic read* for short) is composed of a set of uninterruptible data block read requests[3]. A cluster will not be broken into pieces, but the sort has to wait for enough empty buffers before sending an atomic read. To overlap processing and read time, there must be enough buffers for the merge process to proceed and

[3]In the sort testbed, it is supposed that all jobs in the database system send their I/O requests to an I/O request queue, through which the I/O requests are served by the I/O agents. An atomic read is implemented by locking the I/O request queue, sending a set of read requests, then unlocking it. It is also possible to send a set of read requests of a cluster by using UNIX *readv*() command, which is able to read adjacent blocks from disk into several buffers.

enough buffers for I/O to read an additional cluster. This resulted in Theorem 5.2 for feasible read sequence using atomic cluster reading.

Theorem 5.2 *Let B represent the number of buffers, n the number of runs, Q_i a cluster (a set of adjacent run blocks from the same run), and L_i the number of run blocks in cluster Q_i. When clusters are read atomically, a read sequence of N clusters $\{Q_1, ..., Q_N\}$ is feasible for consumption sequence $\{C_1, ..., C_T\}$, if for all k such that $B \leq k \leq T$, $\{C_1, ..., C_{k-B+n}\} \subseteq Q_1 \cup ... \cup Q_{j-1}$ for the largest j such that $\sum_{i=1}^{j} L_i \leq k$.*

Proof: We assume that a merge process is able to proceed only if the first unfinished block of each run resides in memory.

When $k = B$, $\{C_1, ..., C_n\} \subseteq Q_1 \cup ... \cup Q_{j-1}$ and $\sum_{i=1}^{j} L_i \leq B$, which means the first block of each run belongs to the first $j-1$ clusters, and there are enough buffers to read the first j clusters. When cluster Q_j is being read, Q_1 to Q_{j-1} have already been read into memory. Thus C_1, C_2, ... , C_n reside in memory. The merge process can start.

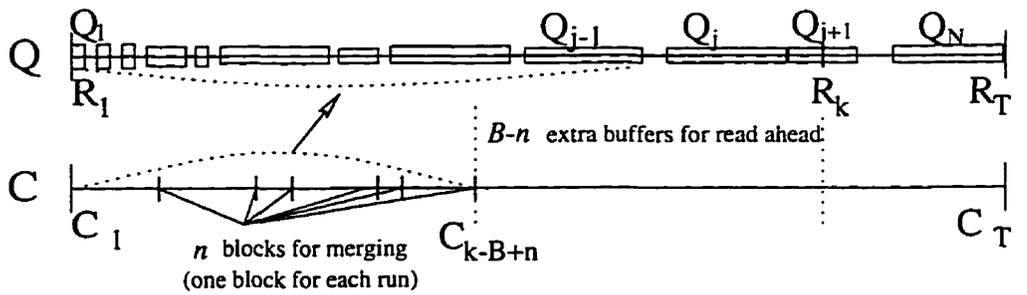


Figure 5.4: Feasibility of read sequence

At any stage when $B < k \leq T$ (as shown in Figure 5.4), among $\{C_1, ..., C_{k-B+n}\}$, n blocks are needed for merging. So $k - B$ blocks must have been consumed by

the merge process. Since $\{C_1, ..., C_{k-B+n}\} \subseteq Q_1 \cup ... \cup Q_{j-1}$, the number of blocks unfinished within $\{Q_1, ..., Q_j\}$ is $\sum_{i=1}^{j} L_i - (k - B)$. Because $\sum_{i=1}^{j} L_i \leq k$, we have $\sum_{i=1}^{j} L_i - (k - B) \leq B$, which means there are enough buffers to store the unfinished blocks in $\{Q_1, ..., Q_j\}$. So after cluster Q_{j-1} is read into memory, the n blocks needed for merging already reside in memory. The merge process can proceed, while there are enough buffers to read cluster Q_j.

The merge process is able to proceed until $k = T$ when all blocks are read into memory. Therefore, the merge process will terminate. So the condition in the theorem guarantees the feasibility of the read sequence. □

Similar to the proposition for simple clustering, Theorem 5.2 is based on the standard merge algorithm. The condition is sufficient but not necessary for merging with delayed reads.

The following algorithm, called *clustering with atomic reads*, is used to compute a feasible read sequence for atomic cluster reading. It is similar to the *simple clustering* algorithm given in the previous section. The initial read sequence is still the consumption sequence for the standard merge algorithm, while each block is a cluster of size 1. Each block is then combined with the previous cluster for the same run if the feasibility of the read sequence is preserved. The major difference between simple clustering and clustering with atomic reads is that the latter records cluster size and uses Theorem 5.2 to check the feasibility of the read sequence. The algorithm returns a sequence of clusters (each cluster with a run number and an address of the first block in the cluster), and returns a cluster size array at the same time.

Algorithm *clustering with atomic reads* :

Input: consumption sequence $C = \{C_1, C_2, ..., C_T\}$,

number of buffers B, number of runs n

Output: read sequence $Q = \{Q_1, Q_2, ..., Q_N\}$, cluster size $L = \{L_1, L_2, ..., L_N\}$

// C_i and R_i have the same structure: run number field and block address field,

// L_i is an integer recording the size of cluster Q_i

begin

$Q := C$; *// Initialize read sequence to be the consumption sequence*

for $i := 1$ **to** T

$L[i] := 1$; *// Set initial cluster size to 1*

endfor;

$lastCluster := n$; *// lastCluster: index of the last cluster before Q[i]*

for $i := n + 1$ **to** T

// Search each previous cluster to find the one with the same run as $Q[i]$

for $j := lastCluster$ **downto** 1

if $Q[j].runNumber = Q[i].runNumber$

then $k := j$; **exit loop**; **endif**;

endfor;

if $Q[i]$ can be combined with $Q[k]$ preserving feasibility

then $L[k]$ ++; *// combine $Q[i]$ with cluster $Q[k]$*

else $lastCluster$ ++; $Q[lastCluster] := Q[i]$; *// Q[i] becomes the new last cluster*

endif;

endfor;

end

To check the feasibility efficiently, a free buffer count array $F = \{F_1, F_2, ..., F_T\}$ was also used to implement this algorithm. F_i records the number of free buffers left after cluster Q_i is read. It is set to $B - n$ initially. When Q_i is combined with cluster Q_j, the values for F_j to $F_{lastCluster}$ are reduced by one. To guarantee that

there are enough buffers to read a cluster while the merge process can proceed, it is required that $F_i \geq L_{i+1}$ for all i. This condition guarantees that the merge process can terminate. For each Q_i, the algorithm needs only check cluster $Q_{lastCluster}$ down to cluster Q_j that $F_j = L_{j+1}$ or $F_{j-1} = L_j$. It is more efficient than using Theorem 5.2 directly.

In simple clustering, whenever a buffer is empty, it is used for reading the next block. When the blocks to be read are adjacent and from the same run, they form a cluster. So a cluster size may be as large as the run length (which happens when the input data is already sorted). For clustering with atomic reads, however, the sort has to wait until there are enough buffers to hold a cluster before issuing an atomic read request. The cluster size is limited by the number of buffers. In fact, the cluster size is restricted by the feasibility of the read sequence. In the above algorithm, the size of each cluster grows to its maximum while preserving the feasibility of the read sequence. Two adjacent clusters are counted as two clusters because they require two atomic reads, even though they may come from the same run and only one disk seek is required. For random data, the next cluster is normally from a run different from the run that the latest cluster was read. So few clusters of the same run are adjacent. Therefore, we can use the number of clusters to approximate the number of disk seeks.

Experimental results

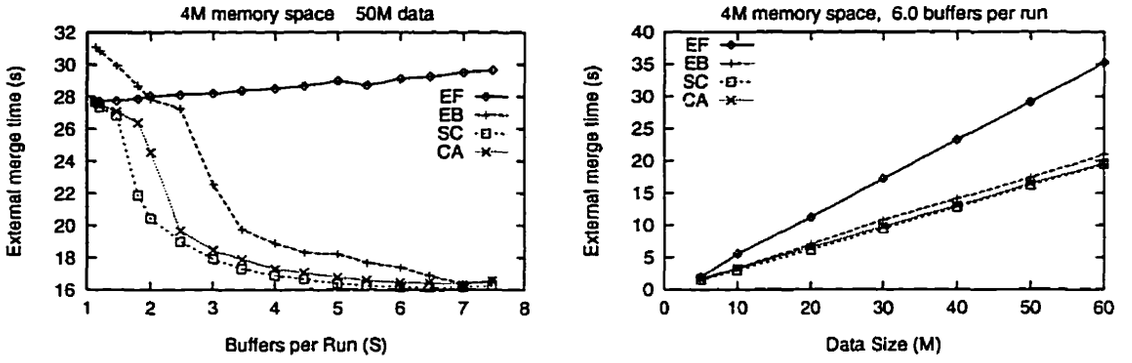
Experiments have been conducted for the following cases:

- single sort, no disk disturbance
- single sort requiring one merge step, with external disturbance
- single sort requiring multiple merge steps, no external disturbance

- multiple sorts, no external disturbance
- multiple sorts with external disturbance

Figure 5.5 shows the experimental results of clustering with atomic reads (CA) compared with previous strategies (with the same data sets), where there is only one sort running in the system. No other jobs access the run disk. Because clustering with atomic reads requires a set of empty buffers to send each atomic read, the average cluster size is shorter than that of simple clustering (see next section). Its performance is not as good as simple clustering, but better than equal buffering and extended forecasting. As the number of buffers increases, both equal buffering and clustering with atomic reads converge to simple clustering, but clustering with atomic reads converges more quickly than equal buffering. Figure 5.5 (b) shows that the result of merging with 6 buffers per run while data size changed from 5M to 60M. Simple clustering and clustering with atomic reads performed almost the same.

During external merge, the run read requests of a sort may be mixed with other disk access activities, which is called *disk disturbance* or disturbance for short. To distinguish it from the disturbance of its own write requests (for intermediate runs or sorted output), the disturbance from other jobs is called *external disturbance*. External disturbance may vary greatly in practice, depending on the system workload. This can be modeled by simply using a disturbance frequency or a probability of disturbance. The sort testbed simulates the external disturbance by sending disturbance requests, each reads a small chunk of data (4K) from a random position on the run disk. The purpose is to move the disk head away from its current position. Disturbance requests are issued according to a given disturbance rate d_r, called the *read disturbance rate* (an input parameter), which is the probability of disturbance for a run read request. Before each run read request, a random number within [0, 1)



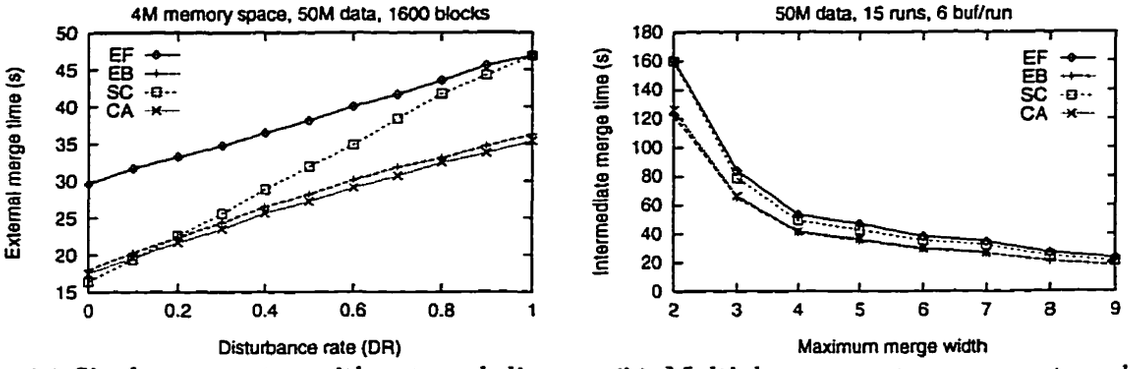
(a) Effect of merge buffers (b) Effect on data size

EF: extended forecasting EB: equal buffering
SC: simple clustering CA: clustering with atomic reads

Figure 5.5: Comparison of read strategies (single sort without disturbance)

is generated. If it is smaller than d_r, a disturbance request is issued. When $d_r = 1$, i.e., the highest disturbance rate, there will be a disturbance request for each run read request. For simple clustering, a disturbance request may appear between any run read requests. Before each run read request, the system will decide whether a disturbance request will be produced or not. For clustering with atomic reads, the disturbance requests can appear only between atomic reads. If the cluster contains m run block read requests, the testbed will generate m random numbers. Whenever there is a number smaller than d_r, a disturbance request is issued. So there may be several disturbance requests between two atomic reads. For a given disturbance rate and a given random seed, the total number of disturbance requests produced is the same for all strategies.

Figure 5.6 (a) shows the impact of external disturbance on the four strategies. The experiment was performed on five 50M (random) data sets. For each data



(a) Single merge step with external disturbance

(b) Multiple merge steps, no external disturbance

Figure 5.6: Effects of disk disturbance (single sort)

set, the run formation phase produced 15 runs with a total of 1600 run blocks. 90 buffers (6 buffers per run) were used during external merge. The disturbance rate changes from the minimum value 0 (no disturbance) to the maximum value 1 (highest disturbance). Since the merge time measured is the elapsed time of the external merge phase, including the time for disturbance requests, it increases as disturbance rate increases.

The equal buffering introduced in the previous section issues a set of read requests for a run when all the buffers of the run but one are emptied. Each set of read requests were implemented by an atomic read, thus its performance is similar to clustering with atomic reads. Extended forecasting reads run blocks in the order of the consumption sequence. For random data, the next block to be read is normally from a run different from the run that the latest block was read. So each read required a disk seek. The total number of disk seeks reached the maximum value and was not affected by the disturbance requests. This resulted in much longer

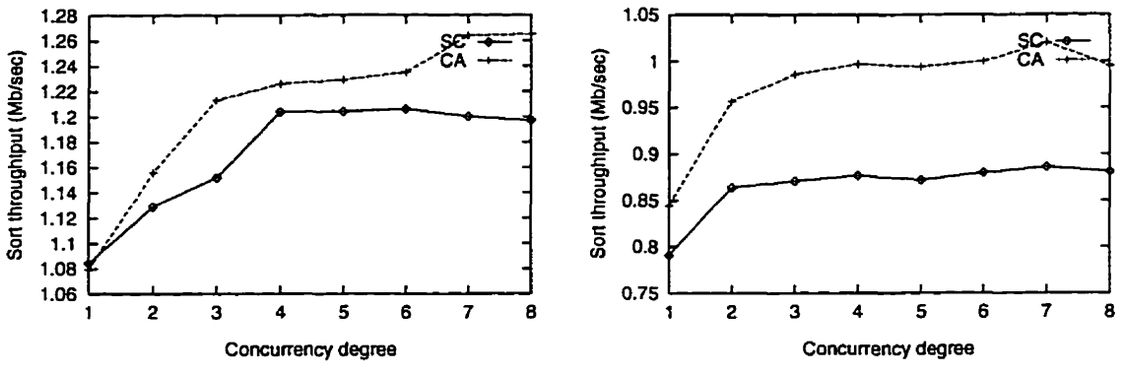
merge time, but the merge time changed at the same rate as for clustering with atomic reads. All these three strategies are not affected by disk disturbances.

The major impact of disturbance is on simple clustering (SC). When there was no disk disturbance, it performed almost the same as clustering with atomic reads (in fact a little bit better). As the disturbance rate increased, the improvement deteriorated. At the highest disturbance rate ($d_r = 1$), simple clustering performed the same as extended forecasting, which means almost all of its reads required a disk seek.

Figure 5.6 (b) shows the results of a sort requiring multiple merge steps. Intermediate runs are stored on the same disk as the input runs. So reading of the input run blocks is mixed with writing of the new generated runs. Since input and output proceed at about the same speed (the small difference is from merging which does not provide empty buffers at a constant rate), so there is a write request after almost each read request. Each read then requires a disk seek. Thus simple clustering is very close to extended forecasting. Both of them are worse than clustering with atomic reads and equal buffering with atomic reads.

Figure 5.7 compares simple clustering and clustering with atomic reads when multiple sorts run concurrently in the system. These experiments were very time consuming. Since extended forecasting always performs the worst and equal buffering is similar to clustering with atomic reads, they were not included in the experiments. Each experiment run consists of twenty 50M sorts, and each sort uses 6 buffers per run during external merge.

Figure 5.7 (a) shows the results of multiple sorts without external disturbance. When the sorts run independently (concurrency degree = 1), both methods have the same performance. As the concurrency degree increases, throughput increases for both of them. This is caused by the overlap of processing and I/O operations



(a) No external disturbance (b) With external disturbance

(SC: simple clustering, CA: clustering with atomic reads)

Figure 5.7: Experiments with multiple concurrent sorts

when multiple sorts reside in the system. It shows that clustering with atomic reads improves faster than simple clustering. The reason is that simple clustering is affected more by mixed requests from multiple jobs.

Figure 5.7 (b) shows the results for multiple sorts with disturbance from other jobs at the same time. The disturbance rate is 0.5. Simple clustering is affected by both other sorts and non-sort jobs (simulated by external disturbances).

The experimental results show that clustering with atomic reads outperforms simple clustering when sorts are affected by disk disturbance, but the degradation of simple clustering is not very significant, unless the disturbance is extremely high. In summary, we offer the following conclusions:

1. The simple clustering algorithm effectively reduces disk seek time for external merge, even with moderate disk disturbance.

2. Clustering with atomic reads is not affected by disk disturbance. It avoids the performance degradation of traditional read strategies. The algorithm is suitable for high disk contention, but it is not as good as simple clustering when there is no disk contention, especially if the number of buffers per run is less than 3 (see Figure 5.5).

5.3 Performance on Partially Sorted Input

All the experimental results in the previous sections are based on random input. In this section, we give some performance results of the read strategies on partially presorted input.

Existing order or presortedness of a data file can be measured in many ways [EC91] [ECW92]. However, most of them cannot be used in our case, because the performance of our read strategies is affected by the existing order of records between runs, rather than the existing order in the input for each run. During the external merge phase, regardless of the existing order in the input of the runs, the records in each run are already sorted during the run formation phase. Zheng and Larson introduced a simple model for producing partially ordered records between runs. The keys in a run i are uniformly distributed in a range Low_i to $High_i$. Each run has a key range of the same length but the key ranges of run i and run $i+1$ are set to overlap. A parameter α controls the overlap of the key ranges for run i and $i+1$ so that $Low_{i+1} = (1-\alpha)High_i + \alpha Low_i$. Setting $\alpha = 1$ produces completely random data. Decreasing α increases the data skew (modeling partially sorted data). Setting $\alpha = 0$ is equivalent to the input file already being sorted [ZL96b].

Our sort testbed was modified to generate partially sorted input based on the

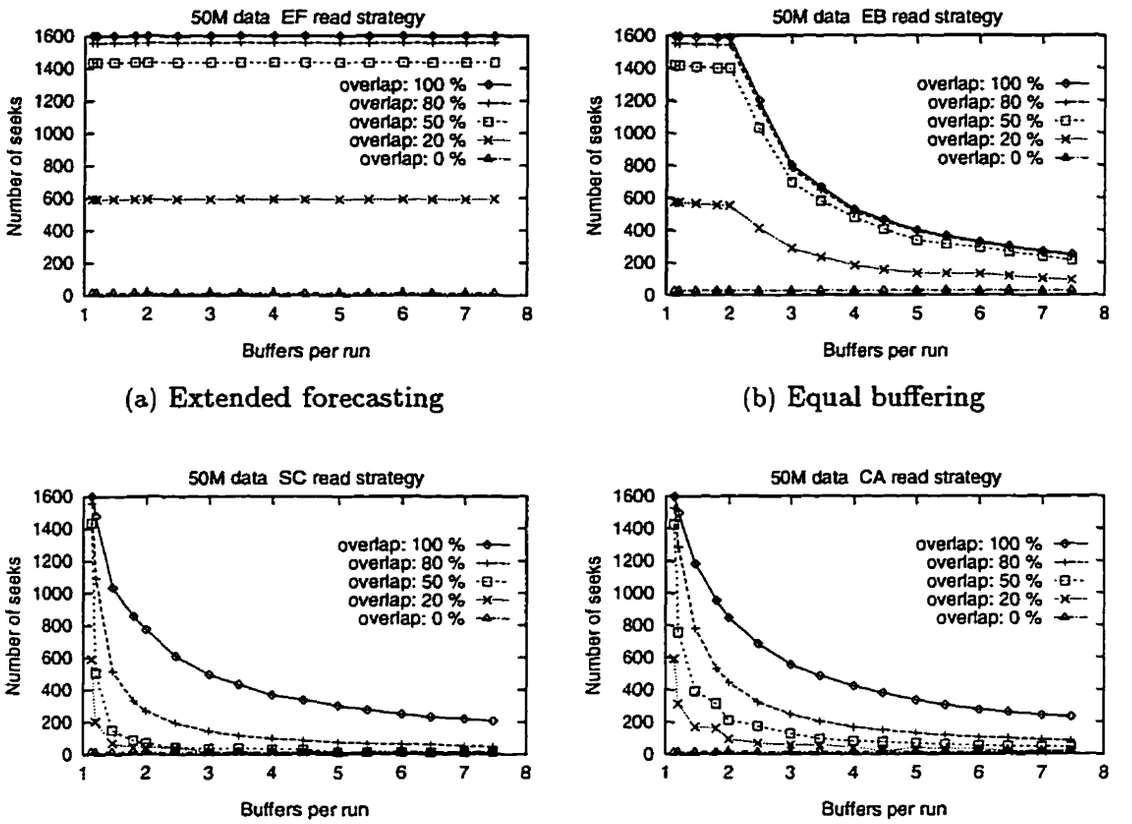
above model. α is set to the value of input parameter *overlap of key ranges*. A set of experiments was performed using each read strategy on 50M data sets. The overlap of key ranges, i.e., the α value, varied from 0 to 1 (100%). 4M memory space was used for sorting, resulting in 15 runs. Merge buffer size was 32K, and the number of buffers per run varied from the minimum number (one buffer per run plus 2 read ahead buffers) to the maximum number allowed by available memory.

Figure 5.8 shows experimental results for five data sets, each representing partially sorted input controlled by the overlap of key ranges. It plots the number of disk seeks as a function of the number of buffers per run for each read strategy. For each set of adjacent blocks (or a set of adjacent clusters) of the same run, only one disk seek is counted for reading[4]. Thus when $\alpha = 0$, only one disk seek is counted for reading each run, resulting in the minimum 15 disk seeks.

The results show that all read strategies perform better on partially sorted input. However, extended forecasting does not benefit from additional buffers. Equal buffering can save disk seeks by using more than two buffers per run, while the two clustering strategies can reduce the number of disk seeks even with a small number of buffers. For the clustering strategies, when the input data is nearly sorted, i.e., the overlap of key ranges is small, the number of disk seeks is close to the minimum with far fewer buffers for read ahead.

Figure 5.9 shows the external merge time as a function of the number of buffers per run for each read strategy. The merge time is mostly consistent with the results for disk seeks shown in Figure 5.8. When the input data is nearly sorted, equal buffering and the clustering strategies can reduce disk seek time to the minimum with far fewer buffers. For equal buffering, the number of disk seeks remains

[4]In practice, more disk seeks are required sometimes even with sequential read. For example, a disk seek may be needed when the data crosses cylinder boundaries. However, the number of these disk seeks is normally small. So they are ignored here.



(a) Extended forecasting

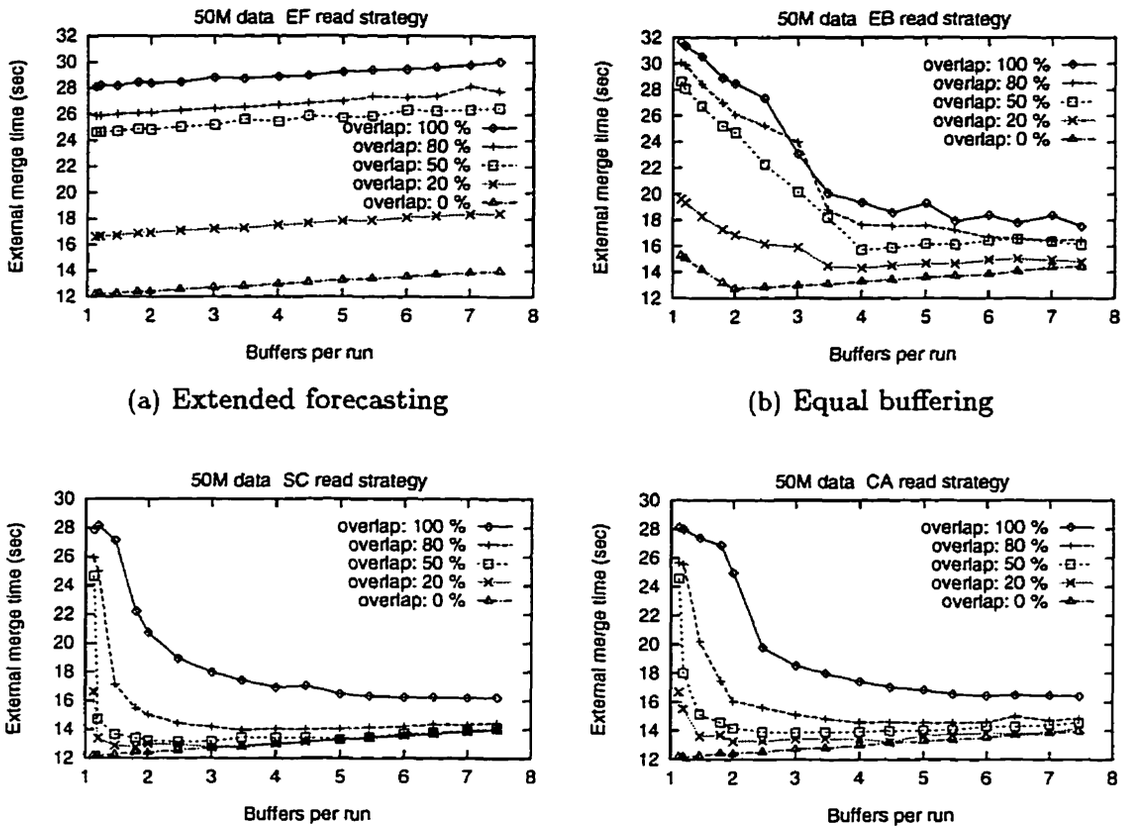
(b) Equal buffering

(c) Simple clustering

(d) Clustering with atomic reads

Figure 5.8: Disk seeks on partially sorted data

constant with less than two buffers per run. This is because some runs have two buffers while others have only one buffer so the sort can use at most one buffer for read ahead for each run. Within this range, merge time decreases as the number of buffers increases, since CPU time and I/O time are better overlapped with more buffers. Extended forecasting does not benefit from extra number of buffers. In fact the merge time tends to increase as the number of buffers increases. This also happens to other strategies when they reach the minimum merge time. Some



(a) Extended forecasting

(b) Equal buffering

(c) Simple clustering

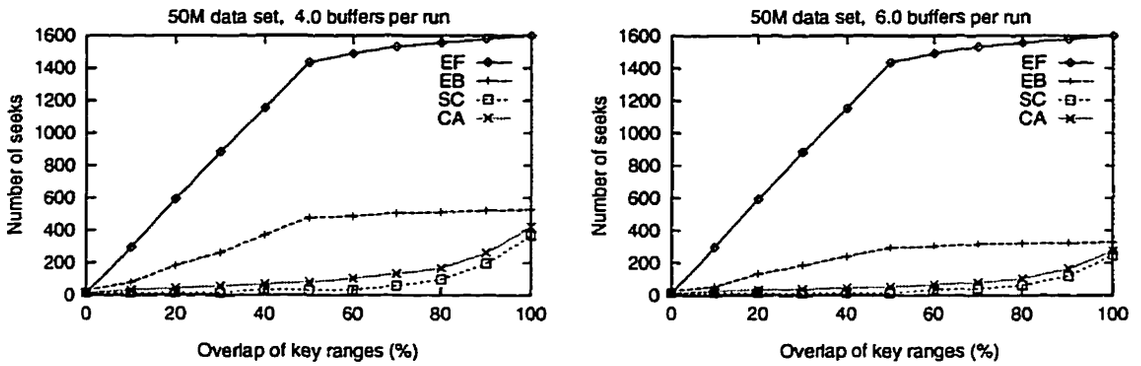
(d) Clustering with atomic reads

Figure 5.9: Merge time on partially sorted data

factors other than disk seeks play a role here. Cache behavior may be one of the reasons, since extra buffers reduce cache locality.

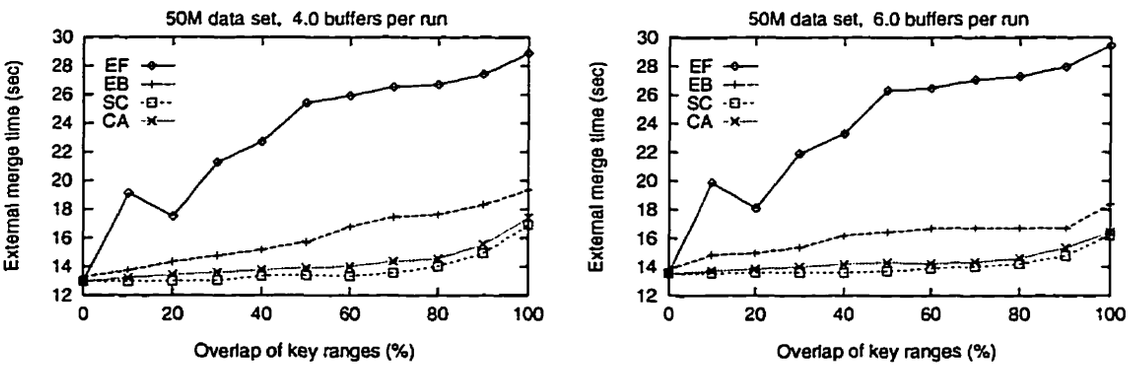
Figure 5.10 compares the read strategies with 4 buffers per run and 6 buffers per run respectively. It shows the number of disk seeks as a function of overlap of key ranges. Figure 5.11 gives the corresponding merge time.

Extended forecasting and equal buffering benefit little from partially sorted data until the overlap of key ranges is below 50%, in which case only two runs are



(a) Number of disk seeks (4 buf/run) (b) Number of disk seeks (6 buf/run)

Figure 5.10: Comparing disk seeks of read strategies



(a) External merge time (4 buf/run) (b) External merge time (6 buf/run)

Figure 5.11: Comparing merge time of read strategies

involved in merging at each stage of the merge process. Unlike these two strategies, the two clustering strategies benefit from partially sorted data starting from a large overlap of key ranges. As the overlap of key ranges decreases, the merge time decreases. At 80% overlap of key ranges (close to random data), the merge time is

almost minimal. Experimental results indicate that the clustering strategies exploit existing order in the input better than equal buffering and extended forecasting. Why extended forecasting has a higher merge time at 10% overlap than at 20% is not known at this time.

In summary, all read strategies benefit from existing order in the input. However, the clustering strategies exploit it better than equal buffering and extended forecasting. For nearly sorted data, far fewer buffers are required to minimize the external merge time.

5.4 Estimate of Improvement

Previous research evaluated the performance effects of read strategies experimentally. In this section, we build approximate models to estimate the performance improvement resulting from the read strategies. We use two factors to measure the performance improvement: average cluster size and read reduction factor.

Definition 5.4.1 *Suppose the number of run blocks is T and the number of clusters is N. Then the average cluster size CS is defined as T/N, and the read reduction factor RF is defined as $1 - N/T$.*

The average cluster size is the average number of blocks in a cluster. When the average cluster size is large, more blocks are grouped together and fewer disk seeks are required to read the run blocks. It reflects the ability of a read strategy to group data blocks into clusters.

For uniformly distributed random data, there are few data blocks adjacent in the consumption sequence when the number of runs is not too small. T disk seeks are required for the consumption sequence, while only N disk seeks are needed

for the read sequence after clustering. The number of disk seeks saved is $T - N$, and the percentage of the saving is $(T - N)/T$, which is $1 - N/T$. Therefore the read reduction factor reflects the amount of improvement from using a clustering technique. From the definition, we have:

$$RF = 1 - \frac{1}{CS} . \tag{5.1}$$

Since extended forecasting does not save any disk seeks from the extra number of buffers, the analysis is focused on simple clustering, clustering with atomic reads, and equal buffering. We also estimate the performance improvement of simple clustering under disk disturbance. For partially sorted input, we do not have approximation models for now. It is left for future work.

5.4.1 Estimate of simple clustering

CS and RF can be greatly affected by the ordering of the input data. For the purpose of estimation, we assume that the sort keys are uniformly distributed, all runs are the same size, and there is only one sort doing a single step merge with no disk disturbance. The approximate model is derived based on the assumption that all runs are equal, in the sense that the probability of each run being required for a block by the merging process is the same, and the blocks of each run have the same opportunity to be clustered, i.e., to be combined with the previous block of the run. In such a situation, we assume that the consumption sequence is close to the ideal consumption sequence shown in Figure 5.12. n is the number of runs and within each sequence of n blocks, there is one block from each run.

When simple clustering is used, each block is combined with the previous block of the same run if feasibility is preserved. Therefore, block C_{n+1}, C_{n+2}, ... , C_{2n}

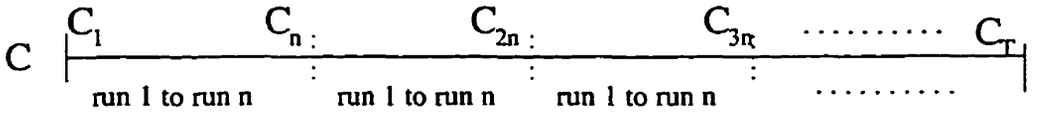


Figure 5.12: Ideal consumption sequence for random data

are combined with C_1, C_2, ... , C_n respectively and form n clusters. Each cluster contains two blocks. Then C_{2n+1}, C_{2n+2}, ..., C_{3n} are combined with these clusters. The cluster size grows until the feasibility cannot be preserved. The remaining blocks will be combined to form the second set of clusters, and so on. Since each block of each run has the same opportunity to be combined with the previous block of the same run, the clusters are the same size, which results in the ideal read sequence shown in Figure 5.13, where Q_i represent a cluster, which is a sequence of adjacent blocks from the same run. Within each sequence of n clusters, there is one cluster from each run.

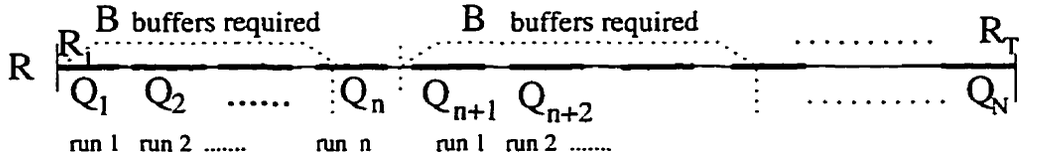


Figure 5.13: Ideal read sequence

To guarantee feasibility, the merge process should be able to proceed while an extra buffer is used to overlap the merge processing and read time. The first $n-1$ clusters and the first block of Q_n are required for the merge process to start, while a buffer is required to read the second block of Q_n. Suppose the cluster size is CS, then at least $(n-1)*CS+2$ buffers are required. Whenever a buffer is empty, it is used to read the next block of cluster Q_n. Based on the equality of the runs, their

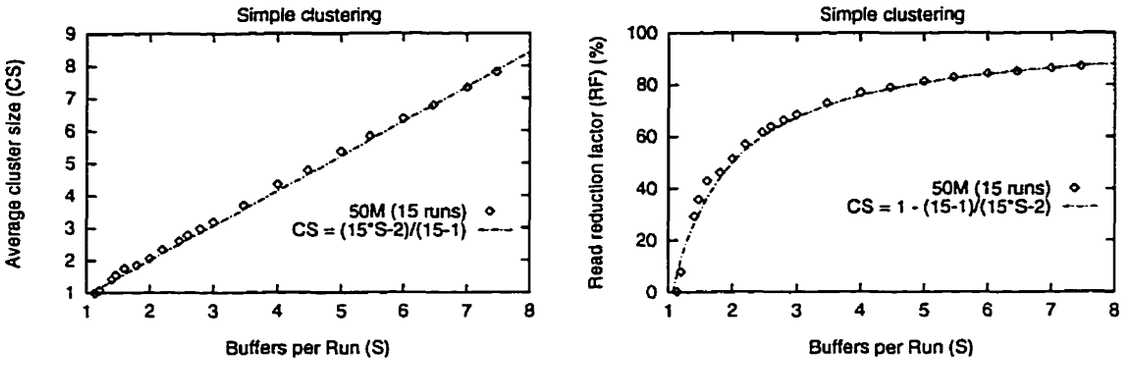
blocks are consumed at the same speed (see the ideal consumption sequence). As Q_1 is finished, enough buffers have been emptied to read Q_{n+1} into memory. The merge process is able to continue. The same holds when Q_2 is finished. When Q_n is finished, there are exactly 2 buffers for Q_{2n}, one to store the first block of Q_{2n} so that the merge process is able to proceed, and another one to read the next block of Q_{2n}. This procedure repeats until the last set of clusters are read into memory. So the $(n-1)*CS+2$ buffers are enough for the merge process to continue until it terminates. Suppose there are B available buffers and S buffers per run (i.e., $S = B/n$), then $B = n*S = (n-1)*CS+2$. So we have:

$$CS_s = \frac{B-2}{n-1} = \frac{n*S-2}{n-1} . \tag{5.2}$$

$$RF_s = 1 - \frac{n-1}{B-2} = 1 - \frac{n-1}{n*S-2} . \tag{5.3}$$

Although these formulas are derived from the ideal case, they provide good estimates of clustering if sort keys are uniformly distributed. Figure 5.14 shows experimental results on 50M data sets with fixed 4M memory space (which resulted in 15 runs, 14 of them are 3584 K and the last run is 1024 K). Sort keys are 10 byte random character strings. Experiments were performed on 10 random data sets. The differences of the results from these random data sets are less than 3%. All the experimental results are very close to the estimated values (the solid line).

Experiment were also performed on data sets with different sizes (20M to 100M). The results for a 20M data set and a 80M data set are shown in Figure 5.15. When the number of runs is small, merging may require more than one block from one run, then from another run. For example, with two runs, the probability of the next block coming from the same run as the block that was just read is 0.5. If there are

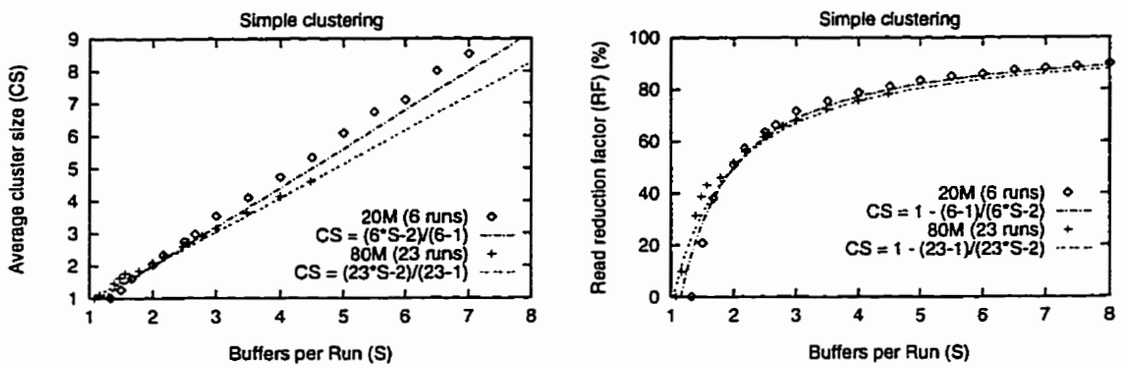


(a) Average cluster size (CS_s) (b) Read reduction factor (RF_s)

Figure 5.14: Modeling simple clustering

ten runs, the probability is 0.1. So there may be clusters already in the consumption sequence, especially for a small number of runs. After the clustering algorithm is applied, the resulting cluster size tends to be greater than the cluster size resulting from the consumption sequence without clusters. Since our formulas are derived based the ideal consumption sequence (without clusters in it), they underestimate the cluster size, especially for a small number of runs. This is reflected in the diagram for the 20M data set (with 6 runs), but the experimental results are still close to the estimated values. The read reduction factor (RF) shows that the number of disk seeks is reduced by over 80% when $S \geq 6$ in all these cases.

When a sort produces variable-length runs (e.g., using memory-adaptive sort), especially when the run lengths differ greatly, experimental results indicated that the average cluster size is larger than the estimated size from formula 5.2, and the performance is better than the estimates based on equal runs. One reason is that data blocks are more often from the longer runs than the shorter runs, which increases the cluster size. But the analysis becomes complicated and is left for



(a) Average cluster size (CS_s) (b) Read reduction factor (RF_s)

Figure 5.15: Varying the input size (simple clustering)

further work.

5.4.2 Estimate of clustering with atomic reads

Similar to the analysis of simple clustering, the estimate for clustering with atomic reads is also based on the ideal consumption sequence and the ideal read sequence in which clusters have the same size. Within each sequence of n clusters, there is one cluster from each run.

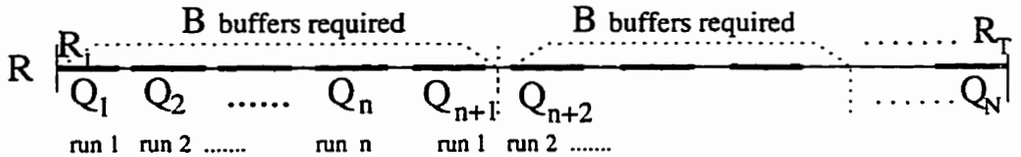


Figure 5.16: Ideal read sequence

For clustering with atomic reads, the sort sends the read requests of a cluster

as an atomic read. $n * CS$ buffers are required to keep the first n clusters so that the merge process can start, while CS buffers are required for Q_{n+1} to overlap the merge processing and read time (as shown in Figure 5.16). We assume that all the runs are consumed at the same speed. When there are enough buffers for the next cluster, another atomic read is issued. By the time the first n clusters are finished, there are enough buffers to keep Q_{n+1} to Q_{2n+1}. So the merge process is able to continue with the run blocks in Q_{n+1} to Q_{2n}, while there are enough buffers for Q_{2n+1} being read at the same time. Thus the merge process is able to terminate with $(n+1) * CS$ buffers. Then we have $B = n * S = CS * (n+1)$, which gives us the following formulas.

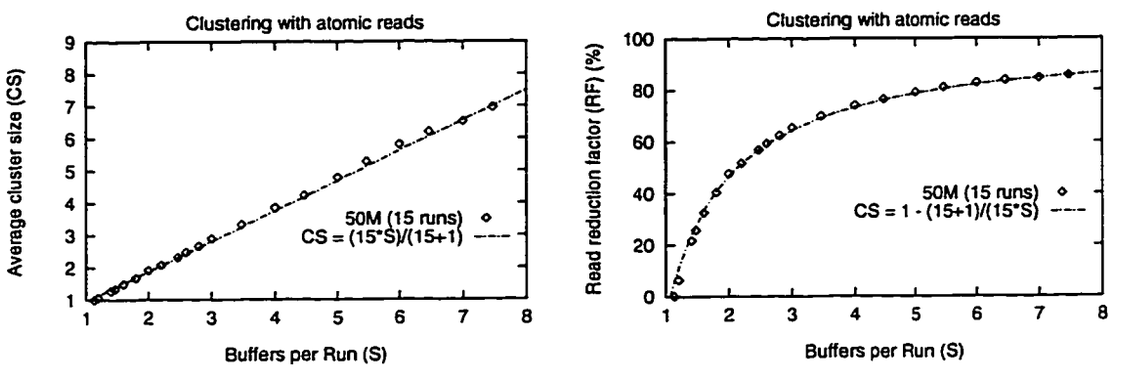
$$CS_a = \frac{B}{n+1} = \frac{n * S}{n+1} . \tag{5.4}$$

$$RF_a = 1 - \frac{n+1}{B} = 1 - \frac{n+1}{n * S} . \tag{5.5}$$

These formulas, derived from the ideal case, again provide good estimates of the effects of clustering with atomic reads when sort keys are uniformly distributed. Figure 5.17 and Figure 5.18 show experimental results for the same data sets used for simple clustering. They are all very close to the estimated values. In fact, the estimates fit the experimental results even better than for simple clustering.

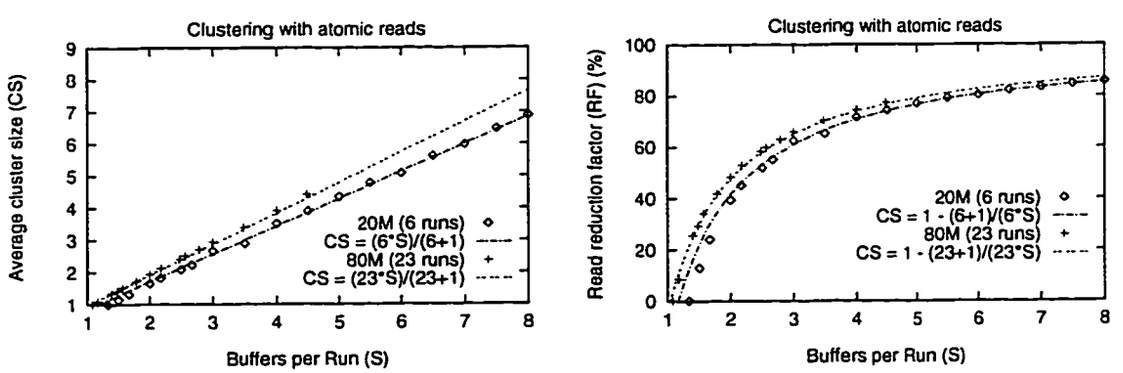
Similar to simple clustering, when a sort produces variable-length runs, experimental results indicated that the average cluster size for clustering with atomic reads is larger than the estimated size using formula 5.4, and the performance is better than the estimate based on equal runs.

From formula 5.2 and formula 5.4, we find that $CS_s = 2 + (S-2)/(1-1/n)$



(a) Average cluster size (CS_a) (b) Read reduction factor (RF_a)

Figure 5.17: Modeling clustering with atomic reads



(a) Average cluster size (CS_a) (b) Read reduction factor (RF_a)

Figure 5.18: Varying the input size (clustering with atomic reads)

and $CS_a = S/(1 + 1/n)$. Therefore,

- As the number of runs n increases, the average cluster size of simple clustering CS_s decreases (when $S > 2$), but the average cluster size of clustering with atomic reads CS_a increases;

- $CS_s > CS_a$ when $S > 1 + 1/n$. Simple clustering results in larger average cluster size, and therefore performs better than clustering with atomic reads (if there is no disk disturbance).

- $CS_s > S$ and $CS_a < S$ (when S > 2). As the number of runs increases, CS_s and CS_a converge to S, which means the effect of both methods becomes close. When the number of runs is large enough, they result in the same average cluster size S, and therefore the same performance improvement ($RF = 1 - 1/S$).

5.4.3 Estimate of equal buffering

Equal buffering can be considered as a special clustering algorithm which uses fixed buffers. It sends an atomic read request when a set of empty buffers (for a run) is available. Each run owns S buffers and uses one buffer for merge. If $S \geq 2$, whenever $S - 1$ buffers of a run become empty, the sort sends an atomic read of $S - 1$ blocks for that run. So the average cluster size is $S - 1$. Then we have:

$$CS_e = \begin{cases} S-1 & \text{if } S \geq 2 \\ 1 & \text{if } 1 \leq S < 2 \ . \end{cases} \tag{5.6}$$

$$RF_e = \begin{cases} 1 - \frac{1}{S-1} & \text{if } S \geq 2 \\ 0 & \text{if } 1 \leq S < 2 \ . \end{cases} \tag{5.7}$$

Normally the number of blocks of a run is not an exact multiple of $S - 1$, so the size of the last cluster of the run is less than $S - 1$. The average cluster size is therefore slightly smaller than the estimate from formula 5.6. When run lengths are large compared to the number of runs, the difference is minimal. Experimental results confirm that the average cluster size and the read reduction factor are almost

identical to the estimates.

From formula 5.2 and 5.6, we can get $CS_s > CS_c$ when $S + n > 3$. Since $n \geq 2$, and $S > 1$ (with at least two extra buffers for read ahead), the condition always holds. So simple clustering produces larger clusters and performs better than equal buffering (if there is no disk disturbance).

From formula 5.4 and 5.6, we can get $CS_a > CS_c$ when $S < n+1$. So clustering with atomic reads performs better than equal buffering when $S < n + 1$. When the number of buffers is small and many buffers are available so that $S > n + 1$, equal buffering may outperform clustering with atomic reads, provided that the processing and read time are fully overlapped. However, if the data is not uniformly distributed, clustering (floating buffering) will exploit the existing order in the input data, but equal buffering (fixed buffering) cannot.

5.4.4 Estimate of clustering with disk disturbance

Clustering with atomic reads and equal buffering with atomic reads issue their block requests in atomic reads, which cannot be affected by disturbance requests. So their average cluster size and read reduction factor are not affected by disk disturbance.

With simple clustering, the improvement deteriorates as disturbance increases. Suppose the disturbance rate is d_r, the average cluster size of simple clustering without disturbance is CS_s, T is the number of run blocks, and N is the number of clusters after clustering without disturbance. For each block, the probability of requiring a disk seek (after clustering) is $P_c = N/T = 1/CS_s$. For each read request, the probability of having a disturbance request is d_r. Then the probability of a block having both a disk seek for the cluster and a disturbance request in front of it is $P_{cd} = (1/CS_s) * d_r = d_r/CS_s$. Each block after a disturbance request requires a disk seek. So for any block, the probability of requiring a disk seek is:

$$P_r = P_c + P_d - P_{cd} = 1/CS_s + d_r - d_r/CS_s = (CS_s * d_r - d_r + 1)/CS_s$$

Among T blocks, $T * P_r$ of them require disk seeks. So the number of clusters is $T * P_r$, and thus the average cluster size with disturbance is $T/(T * P_r) = 1/P_r$. Then we have:

$$CS_d = \frac{CS_s}{CS_s * d_r - d_r + 1} . \tag{5.8}$$

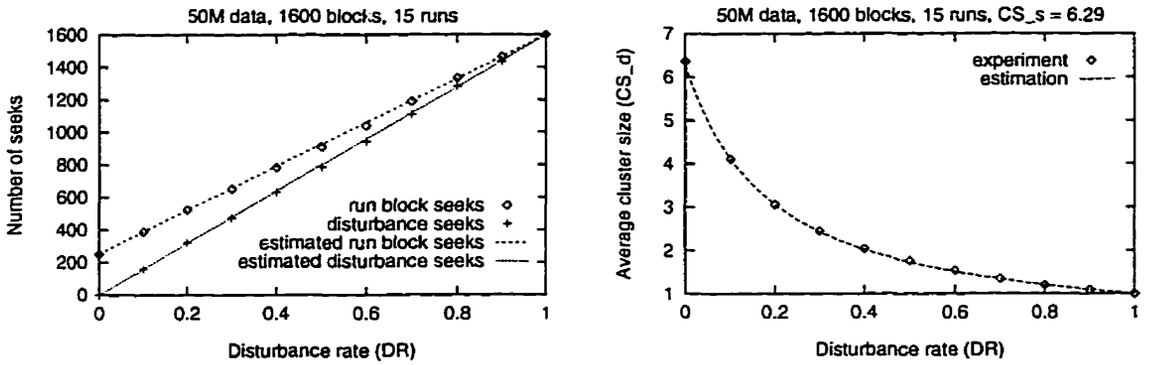
$$RF_d = 1 - \frac{CS_s * d_r - d_r + 1}{CS_s} . \tag{5.9}$$

Here are some special cases to confirm the formulas:

- With the highest disturbance, $d_r = 1$, $CS_d = CS_s/(CS_s * 1 - 1 + 1) = 1$. The average cluster size degrades to 1 when there is a disturbance request before each run block read request.
- Without disturbance, $d_r = 0$, $CS_d = CS_s/(CS_s * 0 - 0 + 1) = CS_s$. Average cluster size is not changed when there is no disk disturbance.
- If $CS_s = 1$, $CS_d = 1/(1 * d_r - d_r + 1) = 1$. If each block in the read sequence already requires a disk seek, disturbance will not (actually cannot) add more disk seeks.

Figure 5.19 shows the experimental results for simple clustering when the disk disturbance rate changes. The left diagram shows the disk seeks for disturbance requests and the disk seeks for run blocks. Given the total number of blocks T, the number of disturbance seeks is estimated by $T * d_r$, while the number of run block seeks is estimated by $T * P_r$. The experimental results are very close to the estimates. As the disturbance increases, the number of disturbance requests

increases, and the number of run block seeks increases at the same time. When the disturbance rate reaches 1, the number of run block seeks equals the number of run blocks, i.e., each run block requires a disk seek. The right diagram shows the average cluster size of simple clustering as the disturbance rate changes. It is very close to the estimates obtained from formula 5.8. When the disturbance rate is 1, the average cluster size degrades to 1, which means there is no improvement from clustering.



(a) Number of disk seeks

(b) Average cluster size (CS_d)

Figure 5.19: Modeling the effects of disk disturbance

When a sort is doing intermediate merge and writes the output run on the same disk as the input runs, the input data size is the same as the output data size. If input and output buffers are of the same size, the number of write requests is the same as the number of read requests. If read requests and write requests are mixed completely evenly, namely, there is a write request before each read request (except the first one), the disturbance rate is 1. Normally, read/write requests will not be mixed exactly in this way. Some read requests may be issued and served continuously. The disturbance rate is then smaller than 1. If output and input use

different buffer sizes, especially when large output buffers are used, there will be fewer write requests. The disturbance rate will be smaller so that the degradation of clustering will be less.

When multiple sorts are running concurrently, it is almost impossible to predict the disturbance rate for a particular sort. The analysis becomes much more difficult and is not investigated further.

5.4.5 Estimate of external merge time

Modern disks have become complicated, making it difficult to precisely predicate the I/O elapsed time. Roughly, the I/O time can be estimated by data transfer time and disk seek time (including rotational latency) as expressed in formula 5.10:

$$T = t * D + s * N \ , \tag{5.10}$$

where D is data size (Mb), t is the data transfer time for 1M data (sec/Mb), N is the number of disk seeks, s is the average disk seek time (with rotational latency), and T is the total elapsed time for accessing the data.

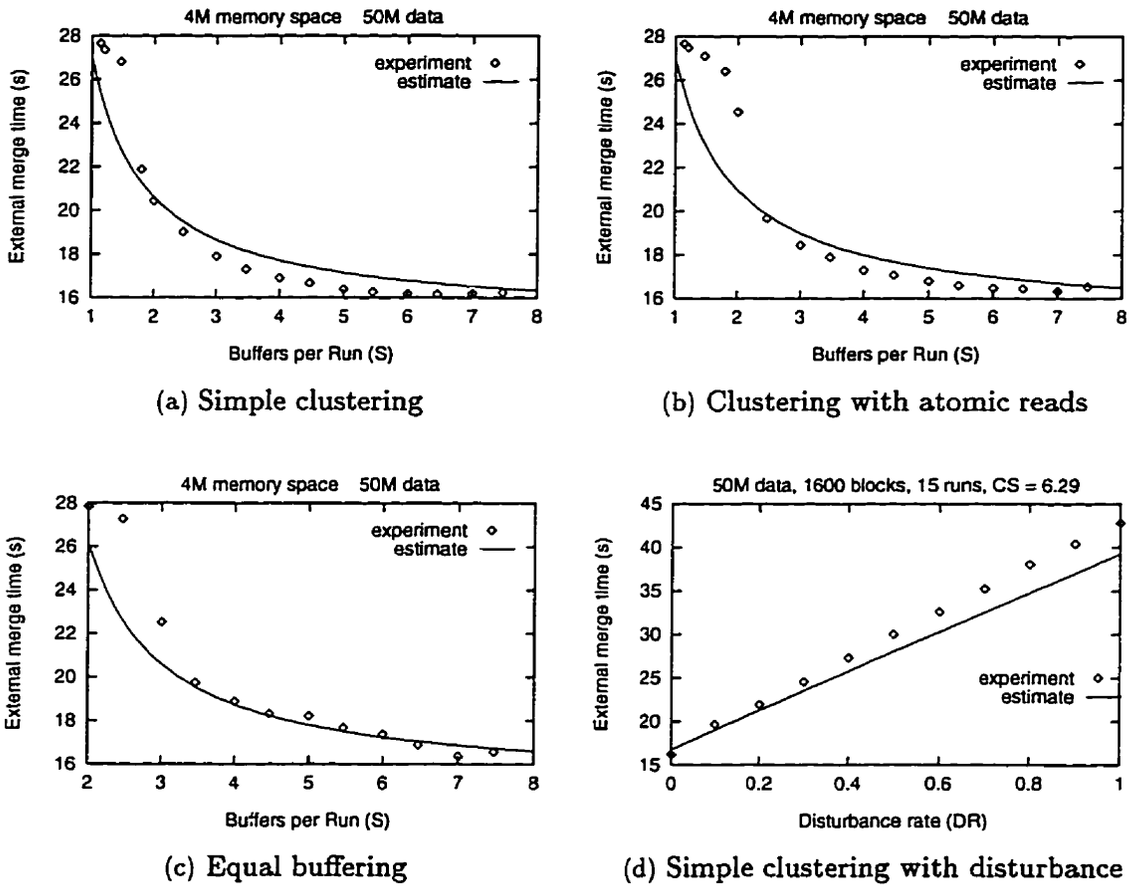
The sort testbed uses a 500M raw partition on one disk, a Seagate ST-15150W. Experimentally, it was found that $t \approx 0.3$ sec/Mbytes and $s \approx 0.007$ sec.

For a one pass merge, the amount of data to be read is the same as the input data size, while the number of disk seeks can be approximated by the number of clusters. Suppose the run block size (merge buffer size) is b, the number of blocks is D/b, and thus the number of clusters is $D/(b * CS)$. Then we have:

$$T = t * D + s * D/(b * CS) = D * (t + s/(b * CS)) \ . \tag{5.11}$$

Figure 5.20 shows experimental results for a set of 50M data sets, and compares

the results with estimates from formula 5.11, where CS is replaced by the average cluster size of each algorithm.



(a) Simple clustering

(b) Clustering with atomic reads

(c) Equal buffering

(d) Simple clustering with disturbance

Figure 5.20: Estimate of external merge time

For the clustering strategies, when the number of buffers per run $S = 2$, the average cluster size is close to 2 (according to formulas 5.2 and 5.4). For equal buffering, when the number of buffers per run $S = 3$, the average cluster size is close to 2 (according to formula 5.6). Figure 5.20 (a) to (c) show that when the average cluster size is over 2, the experimental results and the estimates are close,

but with some differences. When the average cluster size is less than 2, some clusters contain only one block and some contain 2 blocks. The real improvement is much less than estimated. Clearly, factors other than disk seeks play an important role here, but we do not know which factors.

Figure 5.20 (d) shows the the experiment of simple clustering with disk disturbance. The elapsed time collected during external merge includes the part of data access for disturbance requests. The estimated value is the sum of the merge time (based on formula 5.11 and formula 5.8) and the time for disturbance requests based on formula 5.10, where the number of disturbance requests (or seeks) is the number of run blocks multiplied by the disturbance rate, and the data size is the disturbance read size (4K) multiplied by the number of disturbance requests. It shows that the experimental results and the estimates are close when the disturbance rate is low ($d_r < 0.4$). With a high disturbance rate ($d_r \geq 0.4$), the experimental results diverge from the estimates. For this set of experiments, when $d_r \geq 0.4$, the average cluster size $CS_d \leq 2$. So the reason may be the same as the reason for the three clustering algorithms when average cluster size is less than 2.

5.5 Clustering and Buffer Size

The purpose of clustering is to reduce the number of disk seeks when reading run blocks during external merge. Using large buffers also saves disk seeks. However, given a fixed amount of memory, the number of buffers is inversely proportional to the buffer size. Large buffer size results in fewer buffers. The number of buffers per run, S, becomes smaller, thus the average cluster size decreases and the number of disk seeks increases. With clustering, how is the number of disk seeks affected by buffer size?

Suppose D is the amount of input data, M is the size of available memory for external merging, and b is the run block size (merge buffer size). The number of merge buffers is $\lfloor M/b \rfloor \approx M/b$, the number of run blocks is close to $\lceil D/b \rceil \approx D/b$,[5] and the number of clusters $CN \approx (D/b)/CS = D/(b * CS)$. Let n be the number of runs, then $S = (M/b)/n = M/(n * b)$. From formulas 5.2, 5.4, and 5.6, we then have:

Simple clustering:	$CN_s = D/(b*(n*S-2)/(n-1)) = D*(n-1)/(M-2*b)$	
Clustering with atomic reads:	$CN_a = D/(b*n*S/(n+1)) = D*(n+1)/M$	
Equal buffering:	$CN_e = D/(b*(S-1)) = D/(M/n-b)$	(if $S \geq 2$)
	$CN_e = D/b$	(if $S < 2$)

When $S = 2$, $b = M/(2*n)$. So the formula for equal buffering can also be expressed as:

$$CN_e = D/(M/(2*n) + |M/(2*n) - b|) .$$

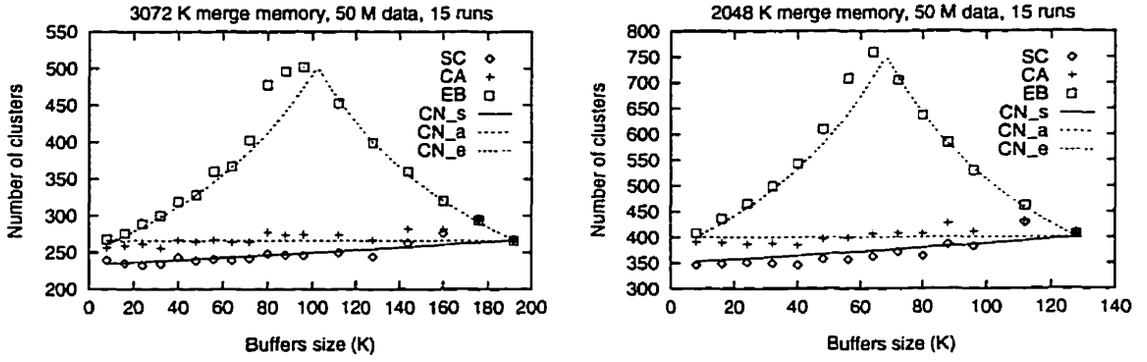
From these formulas we observe that:

- For simple clustering, the number of clusters increases as buffer size increases. When the available memory M is much larger than the buffer size b, the change will not be significant.
- For clustering with atomic reads, the number of clusters is independent of buffer size.
- For equal buffering, when $b \leq M/(2*n)$, the number of clusters increases as buffer size increases. When $b > M/(2*n)$, the cluster size equals the buffer size. So the number of clusters decreases as buffer size increases.

[5] Suppose the run lengths are r_i for n runs ($i = 1$ to n). The exact number of run blocks is $\sum_{i=1}^{n} \lceil r_i/b \rceil$. Since the last block of each run may not be full, the total number of run blocks may be greater than $\lceil D/b \rceil$ blocks.

These results are based on equal size runs. For variable length runs, the results might be different.

Figure 5.21 plots experimental results for a 50M data set (15 runs) with 3M and 2M merge memory respectively. The observed results (plotted as points) are compared to the estimates from the above formulas (plotted as lines). The experimental results are close to the estimates. The differences are mainly caused by rounding down the number of buffers when the buffer size is not an exact divisor of the memory size.



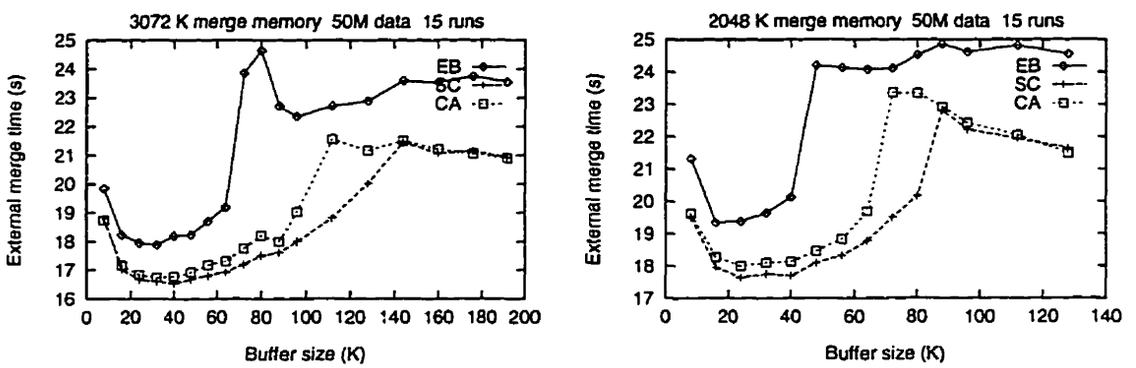
(a) Merge memory space = 3M

(b) Merge memory space = 2M

Figure 5.21: Number of clusters as a function of buffer size

Based on the analysis of the number of clusters, the smallest buffer size would appear to be the best choice. However, our experiments showed that the smallest buffer size did not result in the lowest merge time. Figure 5.22 shows the merge time in the above experiments. When the buffer size is very small, the number of clusters is also small, but the merge time is fairly high. The major reason is that our I/O time estimate (formula 5.10) does not take into account the small overhead for each read request. When a cluster contains many read requests, although there

is only one disk seek, the total overhead becomes noticeable. The experimental results show that this happened when the average cluster size is over 8, but it may not be true for all situations.



(a) Merge memory space = 3M (b) Merge memory space = 2M

Figure 5.22: Merge time affected by buffer size

When the buffer size increase to some point, there is a sharp increase in the external time. It was found that the average cluster size drops below 2 exactly at these points. The experimental results in the previous section (Figure 5.20) show that the three algorithms do not perform as well as the estimates when the average cluster size is smaller than 2, which results in the poor performance of large buffers here.

So the experiment shows that I/O performance is affected not only by disk seeks, but also by other factors, some of which may be device dependent. Experimentally, two conditions may be used as a guideline to choose buffer size:

1. the buffer size should be selected small enough so that the average cluster size is greater than 2;

2. the buffer size should be selected large enough so that the average cluster size is less than 8.

Within this range, the merge time is not greatly affected by the buffer size.

5.6 Summary

This chapter focused on reducing disk seeks during merging, taking into account the overlap of processing and read time, and concurrent jobs. We presented three read strategies: fixed buffering, extended forecasting, and clustering.

Fixed buffering assigns buffers to runs statically, with each buffer dedicated to a run. It reduces disk seeks by sending a set of read requests for a run each time when all this run's buffers but one become empty. The strategy does not rely on the consumption sequence.

Forecasting uses floating buffers for read ahead. Extended forecasting uses more than one additional buffer, and so achieves better overlap of CPU and I/O time than the traditional forecasting which uses only one extra buffer. However, merging for extended forecasting relies on a pre-computed consumption sequence that is determined by the last key or the first key of each run block depending on the merge algorithm (standard merging or merging with delayed reads).

Clustering exploits floating buffers to read run blocks in an order different from the consumption sequence. Blocks from the same run are grouped into clusters for reading, which reduces disk seeks. Simple clustering results in the largest (average) clusters, but its performance deteriorates when there is disk disturbance from other concurrent jobs. An improved method, clustering with atomic reads, retains performance better in this case. It performs better than other strategies when disk disturbance is high. Experimental results on partially sorted input showed that the

two clustering strategies exploit any existing order in the input better than equal buffering and extended forecasting.

Formulas were derived to estimate the performance improvement of the read strategies. The accuracy of the estimates is confirmed by experimental results. We also study the effect of buffer size, resulting in a guideline for choosing buffer size when using the read strategies.

Chapter 6

Merge Patterns

When the available memory is very small or the data sets are very large, runs may have to be merged in multiple steps. The amount of data transferred between disk and main memory is determined by the merge pattern, which also affects disk seeks. The goal of this chapter is to reduce the I/O transmission cost when multiple merge steps are required to complete a sort.

When the sort space remains constant, it is known how to construct an optimum merge pattern. However, no one has ever given the cost of an optimum merge, and this has motivated the study of merge cost in this chapter. The results are used to analyze the relationship between merge width and clustering, as well as the relationship between merge width and buffer size. When the sort space is adjustable during external merge, an optimum merge pattern is not guaranteed. Four merge strategies are proposed for memory-adaptive merge: *lazy merge, eager merge, improved eager merge,* and *optimistic merge.* The chapter is organized as follows. Section 6.1 describes tree representation of merge patterns. Section 6.2 studies the optimum merge patterns of memory-static merge, and derives formulas for calculating the optimum merge cost. Section 6.3 and 6.4 study the relationship

between optimum merge and clustering, and the relationship between merge width and buffer size. Strategies for memory-adaptive merge are proposed in Section 6.5, and the last section summarizes this chapter.

6.1 Tree Representation of Merge Patterns

Perhaps the simplest merge pattern is 2-way merge which merges two runs in each step. It has been studied extensively, especially for merging with tapes [Knu73]. With today's high performance disks and large memory space, multiway merging is more often used. Merge patterns can be represented as trees with initial runs as external "leaf" nodes and output runs as internal nodes. The root node represents the last run, i.e., the final result, which may be sent to some other operator rather then written to disk. Figure 6.1 shows a merge pattern for 6 runs. The number in each node is the length of the run. Each internal node corresponds to a merge step. The length of an output run is the sum of the lengths of its input runs.

Since each internal node corresponds to a merge step, the length of an output run equals the amount of data read in the merge step. Therefore, the sum of all the output run lengths is the total amount of data read during the external merge. Suppose there are m output runs (including the final result) whose lengths are l_1, l_2, ..., l_m, then D_r, the total amount of data read during the external merge, equals $\sum_{i=1}^{m} l_i$.

For each external node, its height, i.e., the length of the path from the root to the node, represents how many times the data of the initial run is involved in a merge step. In other words, the height represents how many times the data is read from disk into memory. Thus the total amount of data to be read can be computed from lengths of the initial runs and their heights. Suppose there are n initial runs

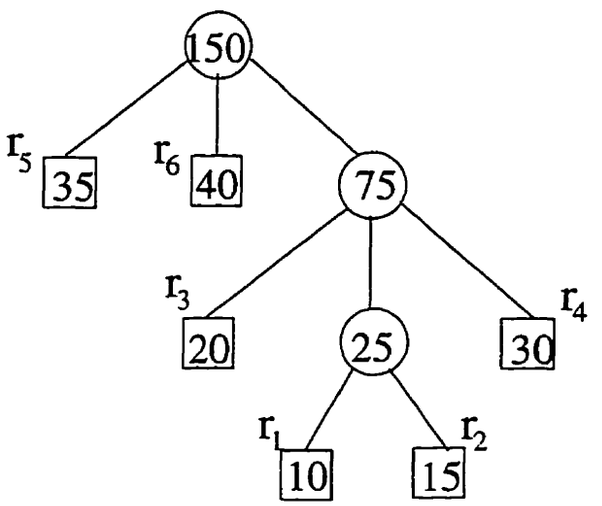


Figure 6.1: Tree representation of a merge pattern

whose lengths are r_1, r_2, ... , r_n and their heights are h_1, h_2, ..., h_n. Then we have

$$D_r = \sum_{i=1}^{n} h_i * r_i \ . \tag{6.1}$$

The right side of the formula is also called the *weighted external path length* of the tree.

When the buffer size b, which is also the I/O unit size, is fixed, the number of run blocks D_r/b is proportional to D_r. Without clustering, reading each block requires a disk seek. Therefore, the number of disk seeks is also proportional to D_r. The amount of data read during the external merge, D_r, is used as a measure of external merge cost throughout this chapter. Because the source data input and sorted result output are not considered in this thesis, the total I/O cost is only from reading and writing run blocks. As the amount of run data written is the same as the amount of data read, the total amount of data transferred is $2 * D_r$, and its

transfer time plus disk seek time is the total I/O cost.

6.2 Memory-Static Merge

Memory-static merge does not change sort space size during the external merge phase. The maximum merge width is fixed for all merge steps. Runs can be merged in passes. Each pass produces a set of new runs from the set of existing runs, and all the data will be read from and written to disk once per pass. Suppose the maximum width is w, the number of runs is n, and the input data size is D. Each pass reduces the number of runs by a factor of w, so $\lceil \log_w n \rceil$ passes are required to finish the merge. Thus the total amount of data read by merging in passes is $D * \lceil \log_w n \rceil$.

However, it is not necessary to merge runs in passes. There are many valid merge patterns. The only requirement is that each merge step must reduce the number of runs so that we eventually end up with a single, completely sorted run. Given n initial runs, possibly of variable length, and a maximum merge width w, which merge pattern will result in the minimum data transmission? Under the assumption that the maximum merge width remains fixed this problem has a very simple solution (see [Knu73] pp.365-366). An optimum merge pattern can be constructed using Huffman's technique. The first step is to add $(1-n) \bmod (w-1)$ dummy runs of length zero, and then repeatedly merge together the w shortest remaining runs until only one run remains. Figure 6.1 actually shows an example of an optimum merge pattern. The first merge step merges r_1 and r_2, as well as a dummy run of length 0 which is not shown in the diagram. Other merge steps always merge the shortest remaining runs.

6.2.1 Optimum merge cost for equal size runs

Most run formation algorithms, except replacement selection, produce equal size runs if sort space size is fixed during run formation. Even though the last run usually is shorter than the previous runs, the situation is close to the equal size runs when the number of runs is not too small. The following theorem gives a formula for calculating the exact amount of data read during the external merge phase.

Theorem 6.1 *Given maximum merge width w and n initial runs all of the same length r, by optimum merge, the total amount of data read during the external merge phase equals*

$$D_r = r * (h * n - \lfloor (w^h - n)/(w-1) \rfloor), \qquad where\ h = \lceil \log_w n \rceil\ . \tag{6.2}$$

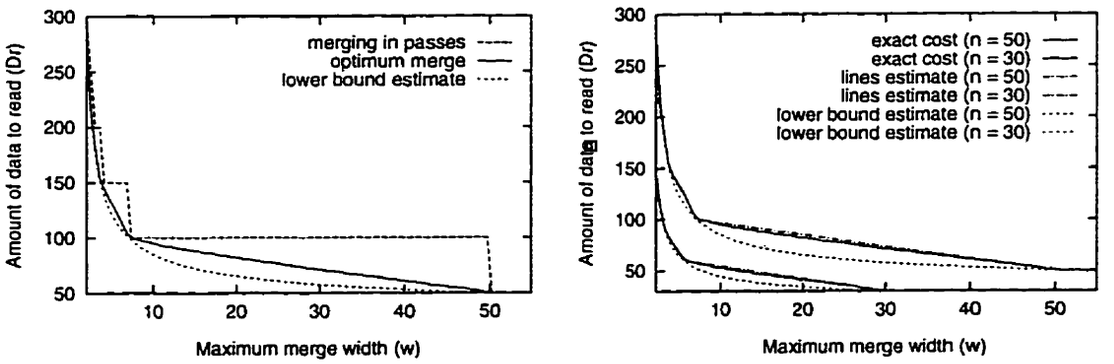
Proof: Since all runs have the same length, the optimum merge pattern corresponds to a tree with minimum external path length. Knuth shows that a complete w-ary tree results in the minimum external path length. He states that the minimum external path length of a w-ary tree is $h * n - \lfloor (w^h - n)/(w-1) \rfloor$, where $h = \lceil \log_w n \rceil$ ([Knu73] pp. 365-366). From Formula 6.1, $D_r = \sum_{i=1}^{n} h_i * r_i$, we have $D_r = \sum_{i=1}^{n} h_i * r = r * (\sum_{i=1}^{n} h_i)$, where $\sum_{i=1}^{n} h_i$ is the external path length of the tree. □

Formula 6.2 provides the exact cost of an optimum merge for equal size runs, but it is too complicated to be used for further analysis. The formula can be approximated by setting $h \approx \log_w n$. Then we have:

$$D_r^{eb} = r * n * \log_w n = D * \log n / \log w \ . \tag{6.3}$$

where D is the input data size (the base of the logarithm is arbitrary), and D_r^{eb} is an estimate of the exact cost. It will be shown in the cost analysis for variable-length runs that Formula 6.3 is a lower bound for the merge cost for equal size runs. So we call it the *lower bound estimate.*

Although this approximation is simple, it underestimates the merge cost. The difference can be as large as 20% of the real cost. Figure 6.2 (a) shows an example of merging 50 runs with run size of 1. The maximum merge width w changes from 2 to 50. The diagram gives the exact cost of an optimum merge and the lower bound estimate, as well as the cost for merging in passes.



(a) Merge cost and approximate

(b) Comparison of estimates

Figure 6.2: Merge cost for 50 runs of size 1

A better approximate is to use a line segment for each range of merge width during which $h = \lceil \log_w n \rceil$ does not change. From Formula 6.2 we can derive that when $w = n^{1/k}$ (k = 1, 2, 3, ...), $h = k$, and

$$D_r = r * (k * n - \lfloor((n^{1/k})^k - n)/(n^{1/k} - 1)\rfloor = k * (r * n) = k * D \quad .$$

When $w = n^{1/(k+1)}$, $D_r = (k+1) * D$. So for $n^{1/(k+1)} \leq w < n^{1/k}$, we can estimate the cost using a line segment defined by two points: $(n^{1/k}, k * D)$ and $(n^{1/(k+1)}, (k+1) * D)$. The function can be derived from the following equation:

$$\frac{w - n^{1/(k+1)}}{D_r - (k+1) * D} = \frac{n^{1/k} - n^{1/(k+1)}}{k * D - (k+1) * D} \quad .$$

So

$$D_r = (k+1) * D - \frac{D * (w - n^{1/(k+1)})}{n^{1/k} - n^{1/(k+1)}} \quad ,$$

When $w \geq n$, the runs will be merged in a single step. The amount of data to be read is the same as the input size D.

We now have a second approximate formula, called *line estimate*:

$$D_r^{el} = \begin{cases} D & \text{if } w \geq n, \quad (\text{where} \quad n \geq 2) \\ D * (k + 1 - \frac{w - n^{1/(k+1)}}{n^{1/k} - n^{1/(k+1)}}) & \text{if } n^{1/(k+1)} \leq w < n^{1/k},\ k = 1, 2, 3, \ldots \end{cases} \quad (6.4)$$

Figure 6.2 (b) shows the line estimate costs of merging 50 runs and 30 runs. The line estimates are much closer to the exact cost than the lower bound estimates.

6.2.2 Optimum merge cost for variable length runs

Replacement selection produces initial runs of variable length. If there is existing order in the input data, run lengths may vary greatly. If a sort is able to adjust its memory space during run formation, run lengths may also vary, and they vary greatly when the sort space experiences dramatic changes from one run to another.

For runs of variable length, the exact formula for calculating the merge cost is

not known, although the optimum merge pattern can be simply constructed with Huffman's technique. The following theorem gives an upper bound and a lower bound on the optimum merge cost.

Theorem 6.2 *Given maximum merge width w, n initial runs, and run lengths r_1, r_2, ... , r_n, if the runs are merged in an optimum merge pattern, then*

$$\sum_{i=1}^{n}(r_i * \log_w \frac{D}{r_i}) \leq D_r \leq (h * n - \lfloor (w^h - n)/(w-1) \rfloor) * D/n \ , \qquad (6.5)$$

where $h = \lceil \log_w n \rceil$, $D = \sum_{i=1}^{n} r_i$, and D_r is the total amount of data read during the external merge phase.

Proof: The lower bound is derived from a coding theorem which gives a lower bound for the average code length for encoding a source alphabet with each character associated with a probability. ([Man87] [YY83]). Let $\{a_1, a_2, ..., a_n\}$ be a source alphabet and $\{c_1, c_2, ..., c_k\}$ be an encoder's alphabet. Each character in the source alphabet will be encoded by a code word, which is a sequence of code letters from the encoder's alphabet. Any encoding schema can be expressed in a *k-ary* tree as shown in Figure 6.3. Each node has at most k children. Level 1 gives the first letter of a code word, level 2 gives the second letter of a code word, and so on.

For encoding with prefix constraint, no code word can be a prefix of another code word, which means no code word is in the path from the root to another code word. All code words are leaf nodes of the tree. A code word's length l_i is the length of the path from the root to the leaf node. Figure 6.4 shows an example of encoding source alphabet $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ with encoder's alphabet $\{c_1, c_2, c_3\}$. It gives the tree representation of the encoding and the code word for each source character. The prefix constraint is satisfied.

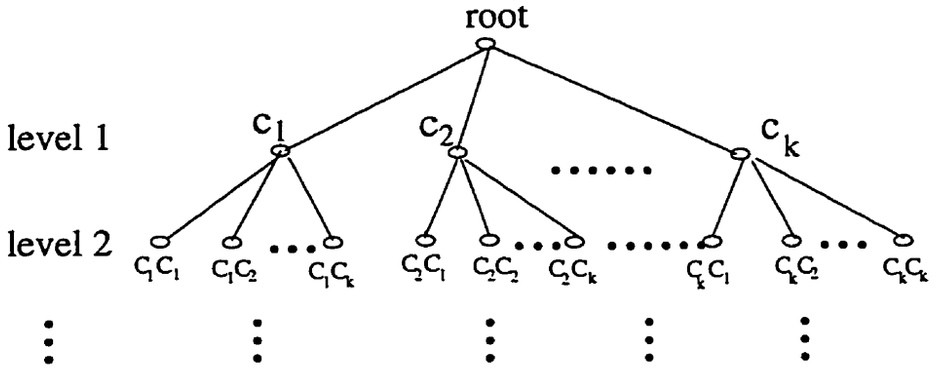


Figure 6.3: Tree representation for encoding

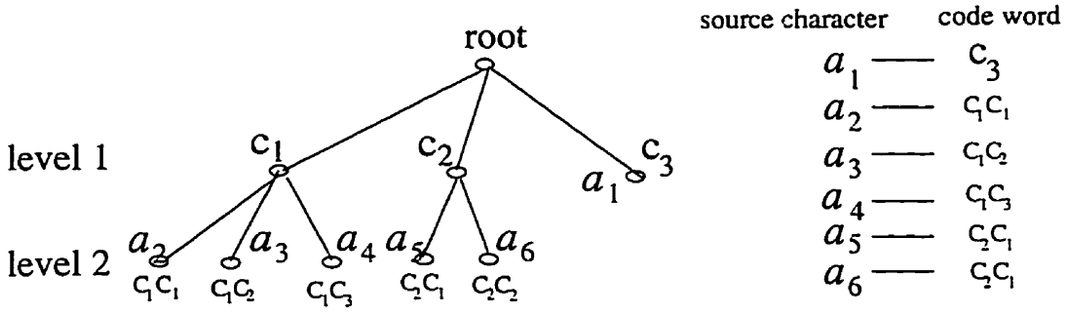


Figure 6.4: An example of encoding with prefix constraint

Given a source alphabet $\{a_1, a_2, ..., a_n\}$ with probability distribution $P(a_i) = p_i$ $(1 \leq i \leq n)$, the average code word length $\bar{l} = \sum_{i=1}^{n} p_i * l_i$, which is the weighted path length of the tree. Huffman's technique is often used to minimize the average code word length. It is shown in [Man87] that for encoding with prefix constraint

$$\bar{l} \geq \frac{H}{\log k} ,$$

where $H = \sum_{i=1}^{n} p_i * \log \frac{1}{p_i}$, and k is the size of the encoder's alphabet. This can

be rewritten as

$$\frac{\sum_{i=1}^{n} p_i * \log \frac{1}{p_i}}{\log k} \leq \sum_{i=1}^{n} p_i * l_i \ .$$

In the merging problem, the optimum merge pattern corresponds to the tree constructed using Huffman's technique. Each initial run is at a leaf node of the merge tree. The maximum merge width w corresponds to the size of the encoder's alphabet k, the number of runs corresponds to the size of the source alphabet, and the path length h_i of an initial run corresponds to the code length l_i. Suppose the input data size is D, which is the sum of the initial run lengths, i.e., $D = \sum_{i=1}^{n} r_i$. The merge cost $\sum_{i=1}^{n} r_i * h_i$ can be rewritten as $D * \sum_{i=1}^{n} (r_i/D) * h_i$, where r_i/D is between 0 and 1, and $\sum_{i=1}^{n} r_i/D = 1$. In the optimum merge pattern, the runs of shortest lengths r_i are merged first and they have the largest path length h_i in the merge tree. Similarly in Huffman encoding, the characters with the smallest probabilities p_i are constructed first and have the longest cord words. Both optimum merge and Huffman encoding try to put off the costly part, long runs or characters with high probabilities, so that they appear at high levels in the tree. The long runs will be involved in fewer merge steps, and the characters with high probabilities will have shorter code words. So the value r_i/D corresponds to the probability p_i in constructing merge trees. Then we have

$$\begin{aligned}
\frac{\sum_{i=1}^{n} (r_i/D) * \log \frac{1}{r_i/D}}{\log w} &\leq \sum_{i=1}^{n} (r_i/D) * h_i \\
\frac{\sum_{i=1}^{n} r_i * \log \frac{D}{r_i}}{\log w} &\leq \sum_{i=1}^{n} r_i * h_i \\
\sum_{i=1}^{n} (r_i * \log_w \frac{D}{r_i}) &\leq D_r \ .
\end{aligned}$$

To prove the upper bound, let D_{re} be the exact cost of the optimum merge for equal size runs whose lengths are the average length of the given runs. We will construct a merge tree for the given runs and prove that the merge cost is less than or equal to D_{re}. Since an optimum merge pattern results in the minimum merge cost, its cost is no more than the cost of our constructed tree. Then we can claim that the cost of the optimum merge pattern is less than or equal to D_{re}.

According to Knuth, a complete w-ary tree gives an optimum merge pattern if all of the initial runs are the same length ([Knu73] pp.365-366). Figure 6.5 shows a general optimum merge pattern of n equal size runs with maximum merge width w, where $(1-n) \bmod (w-1)$ dummy runs of length 0 are added. The lengths of output runs are always larger than the lengths of initial runs. All of the initial runs are at either the bottom level or the second to last level.

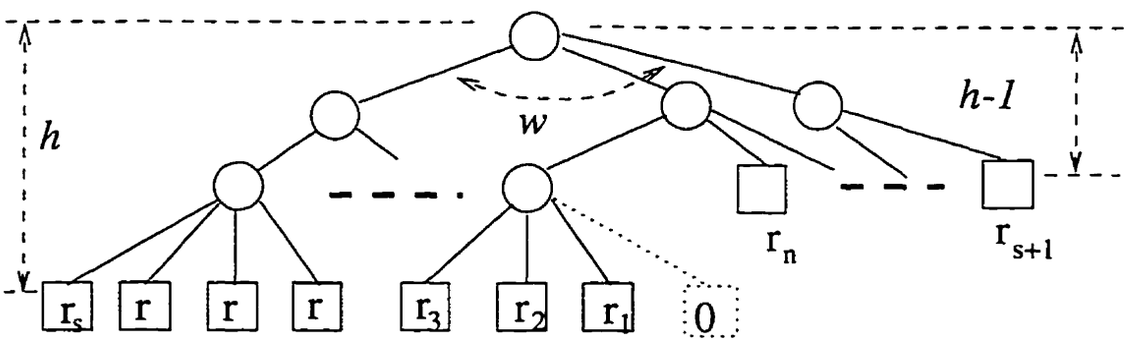


Figure 6.5: Optimum merge tree for equal size runs

Suppose the height of the tree is h, the length of the runs is r, where $r = D/n = (\sum_{i=1}^{n} r_i)/n$, and there are k runs at the bottom level. From Figure 6.5 we can see that the cost of the optimum merge is

$$\begin{aligned} D_{re} &= k*r*h+(n-k)*r*(h-1) \\ &= n*r*(h-1)+k*r\ . \end{aligned}$$

Since $r=(\sum_{i=1}^{n} r_i)/n$,

$$D_{re} = (h-1)*\sum_{i=1}^{n} r_i + k*(\sum_{i=1}^{n} r_i)/n\ .$$

Now we use this merge pattern to merge the variable-length runs. Suppose the run lengths are r_1, r_2, ..., r_n, $r_1 \leq r_2 \leq \cdots\cdots \leq r_n$, and the smallest k runs are at the bottom level. Let D_{rv} denote the merge cost for these variable-length runs, we have

$$D_{rv} = h*\sum_{i=1}^{k} r_i + (h-1)*\sum_{i=k+1}^{n} r_i = (h-1)*\sum_{i=1}^{n} r_i + \sum_{i=1}^{k} r_i\ .$$

Then we have:

$$\begin{aligned} D_{re}-D_{rv} &= k*(\sum_{i=1}^{n} r_i)/n - \sum_{i=1}^{k} r_i \\ &= (k*\sum_{i=1}^{n} r_i - n*\sum_{i=1}^{k} r_i)/n \\ &= (k*\sum_{i=k+1}^{n} r_i - (n-k)*\sum_{i=1}^{k} r_i)/n \\ &\geq (k*(n-k)*r_{k+1} - (n-k)*k*r_k)/n \\ &= k*(n-k)*(r_{k+1}-r_k)/n \\ &\geq 0\ . \end{aligned}$$

Thus $D_{rv} \leq D_{re}$. The cost of the constructed merge pattern for the n runs

is less than or equal to D_{re}. Since the optimum merge pattern of the n runs of variable length has the minimum cost, its cost is less than or equal to D_{rv}, which is less than or equal to D_{re}. Therefore the cost D_{re} is an upper bound. According to Theorem 6.1,

$$D_{re} = r * (h * n - \lfloor (w^h - n)/(w-1) \rfloor) = (h * n - \lfloor (w^h - n)/(w-1) \rfloor) * D/n \ .$$

This completes the proof. □

Based on the lower bound in the above theorem, when all the initial runs have the same length, we have

$$D_r \ \geq \ \sum_{i=1}^{n} (r * \log_w \tfrac{n*r}{r}) \ = \ (\sum_{i=1}^{n} r) * \log_w n \ = \ r * n * \log_w n \ ,$$

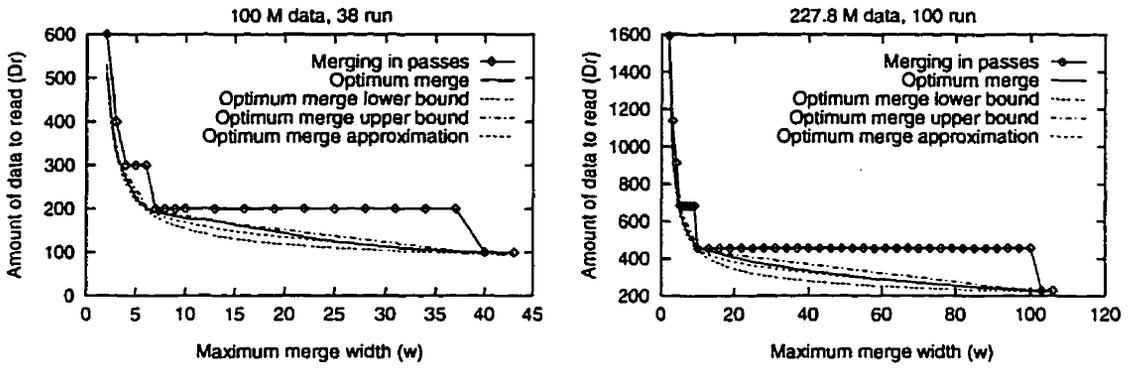
which is the lower bound estimate for equal size runs (Formula 6.3). Therefore, the lower bound estimate mostly underestimates the exact cost of the optimum merge. They are equal only when $w = n^{1/k}$ for an integer k.

Since no formula is available for calculating the exact cost of merging variable-length runs, we approximate it by using the average of the lower bound cost and the upper bound cost, that is, by

$$D_r^{ev} = \frac{\sum_{i=1}^{n} (r_i * \log_w \frac{D}{r_i}) + (h * n - \lfloor (w^h - n)/(w-1) \rfloor) * D/n}{2} \ , \qquad (6.6)$$

where $h = \lceil \log_w n \rceil$, and $D = \sum_{i=1}^{n} r_i$.

Figure 6.6 shows examples of two sets of variable-length runs. The left diagram is based on a set of runs collected from an execution of memory-adaptive sort on the sort testbed. The run lengths are listed in Appendix A.1. The right diagram is based on a set of runs between 1M and 5M whose lengths were randomly drawn from a triangular probability distribution. The run lengths are listed in Appendix A.2.



(a) 100M data, 38 runs (b) 227.8 data, 100 runs

Figure 6.6: Merge cost for variable-length runs

Each of the diagrams plots the exact optimum merge cost, the estimated cost, the upper bound and the lower bound on the optimum merge cost, as well as the cost of merging in passes. The exact optimum merge costs are calculated from an optimum merge algorithm and the estimates are obtained from Formula 6.6.

The diagrams show that the estimates are very close to the exact costs. The cost of merging in passes is usually much higher than the cost of optimum merge. They meet at the points when the number of merge passes changes, i.e., at $w = n^{1/k}$ for each integer $k > 0$. The diagrams also show that within a large range $w \in [\sqrt{n}, n]$, the merge cost changes slowly, but when the merge width is smaller than $\sqrt{n}$, the merge cost increases very fast as merge width decreases. This is also true for equal size runs (see Figure 6.2).

Using multiple disks

Many people have suggested using different disks for input runs and output runs. The purpose is to overlap run input and run output [Sal89], and make run output

fast by exploiting sequential writes. To achieve this goal, two disks are enough if runs are merged in passes. But it has been shown that merging in passes usually results in higher data transmission cost than the optimum merge. With optimum merge, the shortest runs are selected for each merge step. These runs may reside on any disk used for initial runs and intermediate runs. If the maximum merge width is w, input runs may reside on w disks. So $w + 1$ disks are required to guarantee that there is always a disk for intermediate runs which does not contain any input runs. Since the maximum merge width can be large, it is quite possible that a system does not provide as many disk as required.

When multiple sorts access disks at the same time, even if there are enough disks to separate intermediate runs and input runs of a particular sort, the run read/write requests of all the sorts are mixed. Since the work load on each disk may vary, using different disks for input runs and output runs may not help improve performance. Instead, I/O performance is more affected by the utilization of all disks. Balancing the workload among disks is more important. *Data striping* is one of the techniques to solve this problem by spreading each data file across the disks [SGM86] [Kim86].

6.3 Optimum Merge with Clustering

Given a fixed amount of memory, we can use all buffers to maximize the merge width which minimizes data transmission cost. We can also merge a smaller number of runs while using some buffers to cluster run blocks thus reducing disk seeks. The decision is a tradeoff between data transfer time and disk seek time. Then for a given memory size, or a given number of buffers, what is the optimum merge width, taking clustering into account?

In this section and the next section, we use the following notation:

D_r : amount of data read during the external merge phase

N_r : number of disk seeks for reading run blocks

N_w : number of disk seeks for writing run blocks

M : size of available memory space for external merging

b : run block size, which is also the merge buffer size

B : number of merge buffers ($M = b * B$)

w : maximum merge width ($2 \leq w \leq B - 2$)

S : number of buffers per run ($S = B/w$)

D : input data size

n : number of initial runs

t : transfer time for 1M data

s : average seek time.

Since the total amount of data to be transferred is twice the amount of data to be read during the external merge phase, the total I/O cost, including the cost for writing initial runs, is

$$T = t * 2 * D_r + s * (N_r + N_w) \; . \tag{6.7}$$

Assuming that all initial runs have the same length and clustering with atomic reads is used for clustering run blocks, the total number of run blocks to be read is close to D_r/b. The average cluster size CS is estimated by $CS_a = B/(n+1)$ from Formula 5.4, where the number of runs n is the maximum merge width w, since only w runs are merged each time except the first merge step. Then we have:

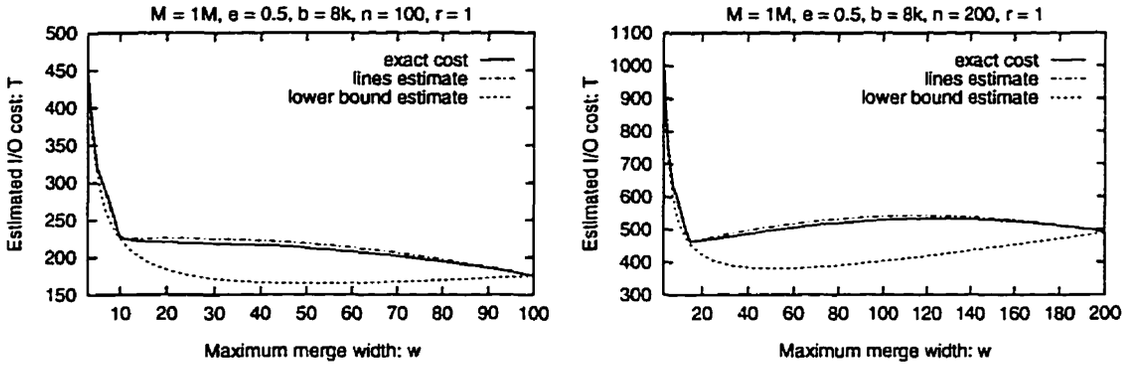
$$N_r \;=\; (D_r/b)/CS_a$$

$$
\begin{aligned}
&= (D_r/b)/(B/(w+1)) \\
&= D_r * (w+1)/(b*B) \\
&= D_r * (w+1)/M \ . \qquad (6.8)
\end{aligned}
$$

The number of disk seeks for writing run blocks may vary greatly depending on how much the run writing is interrupted by other disk activities as well as its own run read requests. Two extreme cases are sequential writes and random writes, in which N_w is 0 and D_r/b respectively. For the general case, we introduce a value d_w as the *write disturbance rate* , so that $N_w = d_w * D_r/b$, where $0 \leq d_w \leq 1$. For sequential writes $d_w = 0$ and for random writes $d_w = 1$. Basically, this value reflects the degree of disk contention. When many external sorts run concurrently or many other jobs access the run disk, it is high. When the disk workload is low or multiple disks are used, it is low. In general we have

$$
\begin{aligned}
T &= t * 2 * D_r + s * (N_r + d_w * D_r/b) \\
&= 2 * t * D_r + s * D_r * (w+1)/M + s * d_w * D_r/b \\
&= (2 * t + s * (w+1)/M + s * d_w/b) * D_r \ .
\end{aligned}
$$

The amount of data D_r can be replaced by the exact cost Formula 6.2, but further analysis will become very complicated. The lower bound estimate (Formula 6.3) is simple, but sometimes it underestimates the cost significantly. Figure 6.7 gives two examples of 100 M and 200 M data sets with run size of 1M. Using lower bound estimate for D_r, the optimum merge width is about 50 in both cases. However, if we use the exact cost formula (Formula 6.2) for D_r, we found that the optimum merge widths are 100 and 14, respectively.



(a) 100 M data set (b) 200 M data set

Figure 6.7: Analysis of merging with clustering

Line estimates are very close to the exact formula. Using Formula 6.4 to estimate D_r, we have

$$T = (2 * t + s * d/b + (s/M) * (w + 1)) * r * n * (k + 1 - \frac{w - n^{1/(k+1)}}{n^{1/k} - n^{1/(k+1)}}) \ , \quad (6.9)$$

where $n^{1/(k+1)} \leq w < n^{1/k}$, $k = 1, 2, 3, \ldots$.

This function is composed of a set of functions, each is determined by k. To minimize T, we can find the optimum merge width within each range, and then find the lowest among them. To find the optimum merge width within each range, the above formula can be rewritten in the following format:

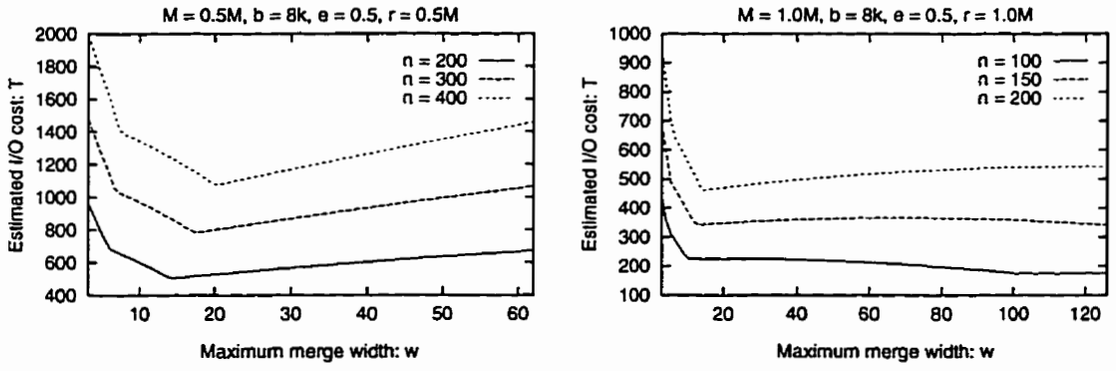
$T = a * w^2 + b * w + c$, where $a = -(s/M) * \frac{r*n}{n^{1/k} - n^{1/(k+1)}} < 0$.

Thus the function has a maximum value, but not a minimum value. Since w is bounded by $n^{1/k}$ and $n^{1/(k+1)}$, T will be minimal at one of these two points. As a result, we need only check merge width $w = n^{1/k}$ ($k = 1, 2, 3, \ldots$ and $w \geq 2$) to

get the optimum merge width.

Normally the number of runs is not very large. When the merge width is smaller than $\sqrt{n}$, the amount of data to be transferred increases very fast as the merge width decreases (see Figure 6.2). At merge width $n^{1/3}$ and smaller, we can hardly get better performance. The check for $k \geq 3$ is virtually unnecessary. In addition, with two buffers for read ahead, merge width is restricted by $B - 2$. So in practice we need only check two merge widths: $\sqrt{n}$ and $min\{n, B-2\}$.

Several examples are plotted in Figure 6.8. Three data sizes and two memory sizes are selected. The merge buffer size is 8K and the write disturbance rate d_w is 0.5[1]. The diagrams are plotted using Formula 6.9. They show that the optimum merge width is either $\sqrt{n}$ or $min\{n, B-2\}$.



(a) 0.5 M memory (b) 1.0 M memory

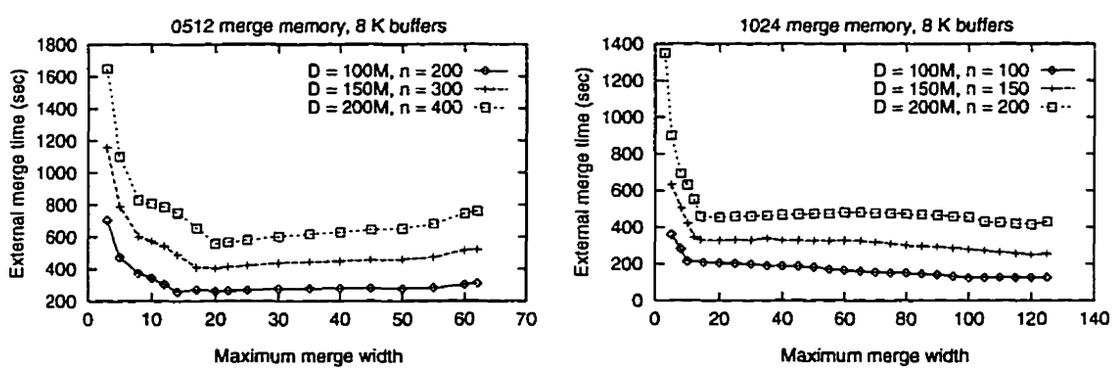
Figure 6.8: Analysis of merging with clustering

Because of the complexity of modern disks, the estimate of I/O cost T may

[1]In many cases the optimum merge width is not very sensitive to d_w. Even though the value of d_w may not be precisely predicted, the optimum merge width selected is still the right one. Through our analysis, we set the value of d_w at 0.5.

not be precise. When the cost at $\sqrt{n}$ and the cost at $min\{n, B-2\}$ are close, the optimum merge width selected may be the wrong one, but it does not affect the performance greatly because the I/O costs at both merge widths are about the same.

Figure 6.9 shows experimental results corresponding to the above examples. The optimum merge widths are about the same as that determined by the formula. For merging the 200M data set with 1M memory, the optimum merge width is 14 according to the formula, but it is close to 120 in the experimental result. However, the experimental result shows that the I/O costs at the two merge widths are very close. Even though the merge width 14 is not optimal, it is still a good choice.



(a) 0.5 M memory (run size = 0.5M) (b) 1.0 M memory (run size = 1M)

Figure 6.9: Optimum merge with clustering (fixed buffer size: 8K)

The above analysis and experimental results are based on equal size runs. For variable-length runs, no models are available to estimate average cluster size, making the analysis impossible for now.

6.4 Merge Width and Buffer Size

With a fixed amount of memory, the number of buffers is inversely proportional to the buffer size. The maximum merge width increases as buffer size decreases. Large merge width minimizes data transmission cost, while small buffers increases the disk seeks. The question is, for a given memory size, what is the optimum buffer size? This is also a tradeoff between transfer time and disk seek time.

Goetz Graefe studied this problem based on the lower bound estimate of data transfer size while assuming that the same amount of memory is used for both the run formation and the merge phase, and that the runs have the same lengths [Gra90]. There are three problems with his results: first, the optimum buffer size selected based on his formula is sometimes far away from the real one because of the poor estimate; second, his result shows that the optimum buffer size is independent of data size, which is not true; and third, because of the complexity of his formula, he suggested a check of all physically possible buffer sizes to find the optimum buffer size. It will be shown in this section that we can find the optimum buffer size by checking far fewer buffer sizes

To simplify the analysis, we assume that all buffers are used to increase merge width and runs are merged without clustering. So the number of disk seeks for reading run blocks is about D_r/b, where D_r is approximated by line estimate (Formula 6.4). Then we have

$$\begin{aligned} T &= t * 2 * D_r + s * (N_r + N_w) \\ &= t * 2 * D_r + s * (D_r/b + d_w * D_r/b) \\ &= (2 * t + (1 + d_w) * s/b) * D_r \end{aligned}$$

$$= (2 * t + (1 + d_w) * s/b) * r * n * (k + 1 - \frac{(M/b) - n^{1/(k+1)}}{n^{1/k} - n^{1/(k+1)}}) , \quad (6.10)$$

where $M/n^{1/k} < b \leq M/n^{1/(k+1)}$, $k = 1, 2, 3, \ldots$.

Similar to the analysis of merging with clustering, we can prove that the optimum value is one at the points $M/n^{1/k}$, k = 1, 2, 3, ... With minimum two buffers for read ahead, the minimum number of buffers is 4. Thus the maximum buffer size is $M/4$. So we have

$$b \leq M/4 \Longrightarrow M/n^{1/k} \leq M/4 \Longrightarrow n^{1/k} \geq 4 \Longrightarrow 1/k \geq \log_n 4 \Longrightarrow k \leq \log_4 n .$$

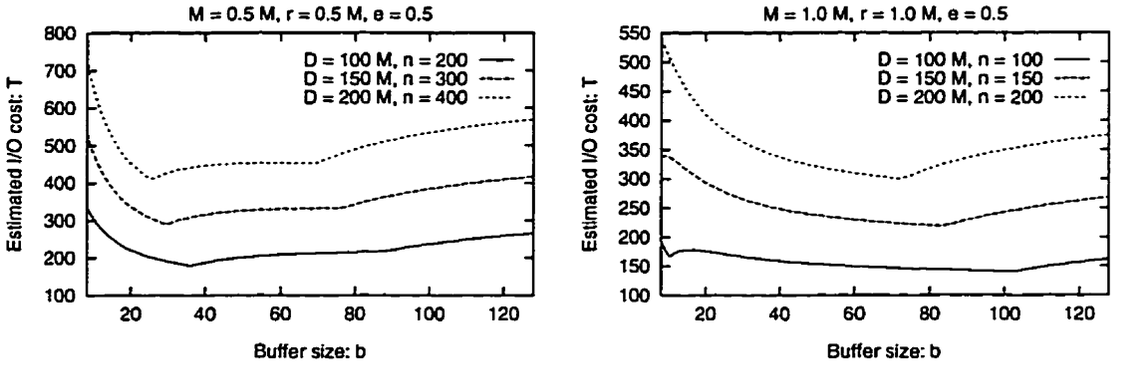
For merging 1000 runs, the maximum value of k is 4. So values larger than 4 are virtually never needed. Normally, the buffer size is selected as some multiple of the page size to improve I/O efficiency. Assume the page size is P (4K or 8K). For each k, checking both $\lceil (M/n^{1/k})/P \rceil * P$ and $\lfloor (M/n^{1/k})/P \rfloor * P$ will give us a better result.

Several examples are given in Figure 6.10. They are plotted using Formula 6.10. All the optimum merge widths are close to $M/\sqrt{n}$. Table 6.1 lists the number of runs for each case and the optimum buffer size rounded up or down to a multiple of the page size (8K).

Table 6.1: Optimum buffer sizes for the examples

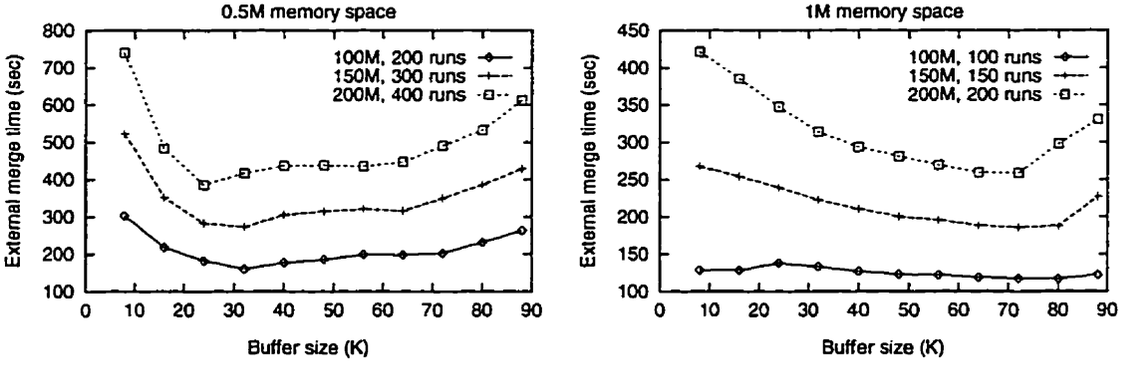
	M = 0.5 M			M = 1.0 M		
D	100M	150M	200M	100M	150M	200M
n	200	300	400	100	150	200
$\sqrt{n}$	14	17	20	10	12	14
b_{opt}	40K	32K	24K	104K	80K	72K

Figure 6.11 shows experimental results corresponding to the above examples. The optimum buffer sizes are almost the same as those shown in Figure 6.10 and



(a) 0.5 M memory (b) 1.0 M memory

Figure 6.10: Analysis of merge width and buffer size



(a) 0.5 M memory (run size = 0.5M) (b) 1.0 M memory (run size = 1M)

Figure 6.11: Effects of buffer sizes

Table 6.1.

For variable-length runs, the data transfer size is close to the upper bound given in Theorem 6.2, and therefore close to the line estimate. So we can use the above method to determine the optimum merge width, although the result may not be the

exact optimum one. If precise buffer size is desired, we can evaluate each possible buffer size using Formula 6.10, while the data transfer size D_r is estimated by the approximate formula for variable-length runs (Formula 6.6).

When clustering is taken into account, both buffer size and merge width are variable. The optimization becomes more complicated. Since only two checks are needed to find the optimum merge width for a given buffer size, a straightforward strategy is to find the optimum merge width for each physically possible buffer size and compare their costs.

6.5 Memory-Adaptive Merge

In the previous sections, we assumed that the available memory space for merging remains constant. With memory-adaptive sort, the memory usage of a sort may change from one merge step to another, which means that the maximum merge width changes dynamically. Since the memory change is unpredictable, it is impossible to plan an optimum merge in advance. However, the following two facts still hold:

- Merging m runs always reduces the total number of runs by $m - 1$.
- Merging the shortest runs transmits less data than merging any other runs.

6.5.1 Dynamic merge strategies

To make merging adapt to the memory changes in the system, we devised four merge strategies. All of them merge the shortest remaining runs in each merge step. The strategies focus on how to determine the merge width for each merge step.

Suppose n is the number of existing runs before a merge step, w is the maximum merge width allowed by the available memory, and m is the actual merge width which has to be determined ($m \leq w$). The values of w and m may change from one merge step to another, while n is reduced by $m - 1$ after each merge step.

Lazy merge : merge the smallest number of runs if the existing runs cannot be merged in a single step, i.e., if $n \leq w$, $m = n$; otherwise, $m = 2$.

This strategy tries to do minimal work in each merge step and postpones the costly merge(s) as long as possible, hoping that the system will soon have enough space to merge the remaining runs in a single step.

One of the best cases of this strategy is when, after the first merge step, there is enough memory to merge the remaining runs in one step. These two merge steps result in the minimum data transmission.

However, if the available memory is decreasing and $w < n$ after each merge step until $n = 2$, only two runs are merged each time, which results in the maximum number of merge steps and high cost in data transmission.

The major problem of this strategy is that it does not make full use of the available memory resource to reduce the merge cost.

Eager merge : merge as many runs as possible each time, i.e., if $n \leq w$, $m = n$; otherwise, $m = w$.

This strategy works eagerly by utilizing all the available memory. The number of runs is reduced as much as possible in each merge step, resulting in the minimum number of merge steps.

If $w = n$ in the last merge step, memory resources are fully utilized for each merge, and the amount of data transmission is minimal.

If only two runs are left for the last merge while more memory is available, the

memory space is not fully utilized.

The major problem of this strategy is that the merge width of the last merge step may be very small, which means the transfer cost of the second to last merge step or previous steps is high. As a result, the total transfer cost is higher than it is in an optimum merge.

Improved eager merge : merge as many runs as possible in each step until the available memory is large enough to merge the remaining runs in two steps, then keep the sort space fixed and do an optimum merge, i.e., if $n \leq w$, $m = n$; if $w < n < 2 * w$, $m = n - w + 1$; otherwise, $m = w$.

This strategy tries to merge fewer runs in the second to last merge step by making the merge width of the last merge step wide. Since the last merge step always reads a fixed amount of data, i.e., the total run data, which is independent of the merge width, minimizing the I/O cost of the second to last merge step will reduce the total transfer cost.

If $n - w + 1 = w$ in the second to last merge step, memory resources are fully utilized for each merge, and the amount of data transmission is minimal.

If the second to last merge step merges only two runs, the I/O cost can be further reduced by merging fewer runs in the third last merge step and making the merge widths of the last two merge steps wide. However, we have to keep the sort space fixed for the last three merge steps. In the extreme case, we can keep sort space fixed for the whole merge phase and do an optimum merge. The strategy degrades to a memory-static merge. In this case, newly available memory in the system is not utilized at all.

One major problem of improved eager merge as well as the previous strategies is that they do not merge the runs in an optimum pattern if the available memory

remains constant.

Optimistic merge : always do an optimum merge based on the currently available memory space. More specifically, at the beginning and after each sort space change, add $(1 - n) \bmod (w - 1)$ dummy runs of length zero, and merge w shortest runs, including the dummy runs. In the following merge steps, always merge w shortest runs.

The philosophy of this strategy is that whatever has been done is done and the work can be done better only from now on. It tries to optimize the remaining merge steps based on the current sort space, hoping that the transfer cost of merging the remaining runs will be minimized. It produces an optimum merge pattern within each time period during which there are no memory fluctuations. Whenever sort space changes, the sort moves to a new merge pattern, that is optimum for the new sort space. Usually some dummy runs are added after memory adjustment. So the number of runs actually merged or the real merge width in the first merge step after memory adjustment is normally smaller than w. Memory is not fully utilized for this merge step.

The best case of this strategy is that no dummy runs are added so that the available memory is always fully utilized, and the amount of data transmission is minimal. It results in an optimum merge if the available memory does not change during the merge phase.

The worst case is that w changes after each merge step and $w - 2$ dummy runs are added. Only two runs are merged each time, which results in the maximum number of merge steps and expensive data transmission.

The major problem of this strategy is that the sort has to adjust to the initial merge width in the first merge step after each memory adjustment. This merge

width might be small, which results in poor performance when sort space changes frequently.

6.5.2 Memory usage patterns

Given a merge strategy, the memory usage of a sort in the system follows its own pattern. From Figure 6.12 we can see some features of the four strategies. The dotted lines represent changes of the available memory in the system, including the memory space occupied by this sort. The solid lines represent changes of the sort space occupied by this sort during its merge phase[2], while the dashed lines represent the amount of memory actually used by this sort. For lazy merge and eager merge, the amount of memory occupied by the sort is the same as the amount of memory actually used.

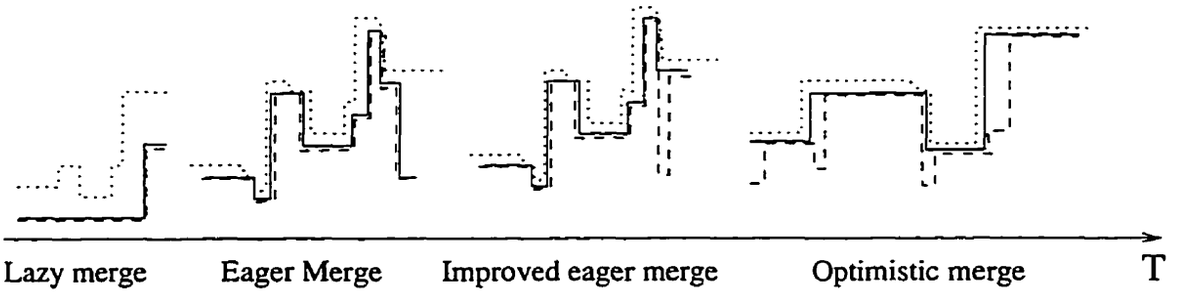


Figure 6.12: Memory usage changing patterns

Lazy merge uses the minimum merge space until there is enough space to merge the remaining runs in a single step. So the last merge step uses more memory than the previous ones.

[2]In the memory-adaptive algorithm, a sort at merge phase does not use up all the available memory in the system. Some memory is reserved for high priority sorts and incoming sorts in order to improve overall system performance.

Eager merge adapts itself to the memory fluctuations. So does improved eager merge except that it keeps the sort space fixed in the last two merge steps in order to reduce transfer cost. For improved eager merge, the merge width is usually smaller than the maximum merge width in the second to last merge step. The extra memory occupied can be used for clustering which reduces disk seek time for reading run blocks.

For optimistic merge, the sort usually merges a small number of runs for one step and then merges the remaining runs with the maximum merge width during which the memory space remains steady. Similar to improved eager merge, when merging a small number of runs, the extra memory occupied can be used for clustering to reduce disk seeks.

6.5.3 Comparisons of the merge strategies

Each strategy has its advantages and disadvantages. A strategy may perform better than others in one situation, but worse in another situation. The four adaptive merge strategies and optimum static merge are compared based on their total transfer cost. Assume that the number of initial runs is larger than the maximum merge width when the external merge phase starts, so that the runs have to be merged in multiple steps.

(a) *No memory fluctuation* : The maximum merge width w does not change during the whole merge phase.

Optimistic merge ends up being an optimum merge that exploits all the available memory space. The I/O cost for the sort is minimal. Optimum static merge performs exactly the same as the optimistic merge, if they use the same amount of memory space.

The other strategies may also produce an optimum merge, but only in rare cases. Since the last merge step always reads the same amount of data, improved eager merge has lower transfer cost than eager merge. Lazy merge merges two runs each time and more runs in the last step, so it requires the most merge steps and performs the worst.

(b) *Increasing memory* : The available memory starts from a small space and keeps increasing. The maximum merge width w increases after one or more merge steps.

Memory-static merge does not benefit from newly available memory, since the sort merge space is fixed. The performance of optimistic merge is determined by the frequency of the memory changes and the merge width of the first merge step after each memory change (or the number of dummy runs added). Suppose the merge width for each first step after a memory change is half of the maximum merge width on average, if w changes after each merge step, only half of the available memory is used. If the available memory changes less often, optimistic merge will perform better.

Eager merge outperforms improved eager merge if more runs are left to the last merge step than the runs merged in the second to last merge step. Otherwise, the transfer cost of using eager merge is higher than the cost of using improved eager merge.

Lazy merge always merges two runs until the merge width is greater than the number of remaining runs. It performs well if a large amount of memory is soon available to merge the remaining runs in a single step. If not, it will require many merge steps making transfer cost high.

(c) *Decreasing memory* : The available memory starts from a large space and

keeps decreasing. The maximum merge width w decreases after one or more merge steps.

Memory-static merge does not use the extra space when the available memory is large, and does not release its sort space when the system is short of memory. Memory-adaptive merge is able to release part of the sort space to improve overall system performance.

Similar to (b), the performance of optimistic merge is affected by the frequency of the memory changes. It may perform very poorly if the available memory changes after every merge step, but it is coming closer to be an optimum merge as the frequency of memory change decreases.

Eager merge and improved eager merge performs well in this case by making full use of memory resources at each stage. Improved eager merge has lower cost by keeping the sort space fixed in the last two merge steps.

Lazy merge merges two runs and reduces the number of runs by 1 in each step. The maximum merge width may decrease at the same time. It does not utilize the available memory space when the space is large. Generally, it is not a good strategy, although it works well in special cases, such as $n = w + 1$ under the condition that memory space does not shrink in the next merge step.

(d) *Increasing/decreasing memory* : This is the general case. The available memory increases and decreases as the system workload changes. It is impossible to predict how the maximum merge width w will change, and it is difficult to tell which strategy will perform best. However, some facts are true:

- Optimum static merge minimize I/O cost given fixed memory space. However, it cannot utilize extra memory available in the system, and does not reduce its sort space on behalf of other jobs in the system.

- Lazy merge uses minimum sort space until the last merge step. The extra space in the system is used only when the available memory is large enough to merge all the remaining runs in one step. Therefore, it does not make full use of memory resources. Its performance may be worse than memory-static merge in some cases.

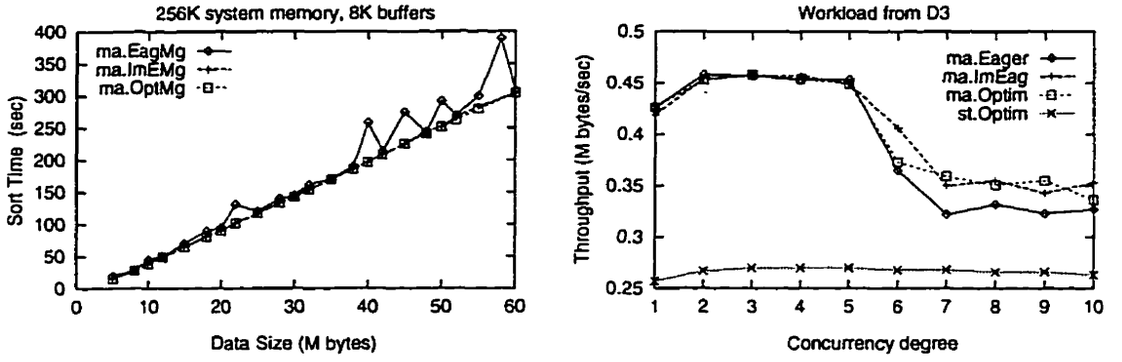
- Eager merge makes full use of memory resources. It adapts itself to memory changes freely, either using newly available space or releasing part of its space. However, there may be fewer runs left for the last merge step, which means higher cost in the second to last merge or previous merges.

- Improved eager merge is similar to eager merge, but it reduces the cost in the second to last merge step.

- The performance of optimistic merge is close to an optimum merge when memory fluctuation is small, but its performance degrades on frequent memory changes.

Figure 6.13 shows two sets of experiments. Diagram (a) gives the elapsed time of single sorts in the case of very small system memory space (256K). It reflects the situation when the available memory does not changes during the merge period. Optimum static merge is not included since it performs the same as optimistic merge if it uses the same amount of memory. The performance of optimistic merge and improved eager merge are almost the same, while eager merge is occasionally worse than the two of them.

Diagram (b) gives the system throughput of multiple sorts based on data set D3 used in Section 4.5.2. The system memory space used is 2M and the merge buffer size is 8K. When the concurrency degree is 1, each sort job run independently without memory fluctuations. For memory-static sort, the single sort space



(a) Single sort (b) Multiple sort

Figure 6.13: Comparison of merge strategies

limit is 256 K so that at most 8 sorts are able to run concurrently in the system. With memory-adaptive sort, more sorts may be able to run concurrently. As the concurrency degree increases, the available memory in the system may change more often.

The diagram shows that the performance of all the strategies increases when the concurrency degree changes from 1 to 2. The reason is that I/O time and CPU time are overlapped when multiple sorts are running in the system. When the concurrency degree increases further, the performance of static sort does not change much, since each sort always uses the same amount of memory. As a result, its transfer cost is always the same. Memory-adaptive sorts consistently perform better than memory-static sorts because of full utilization of memory resources. Most of the time, improved eager merge and optimistic merge are better than eager merge. Experiments on other data sets produced the similar results, but it does not mean that improved eager merge and optimistic merge are the best strategies in all cases.

From both the analysis and the experiments we can see that: (1) memory-static merge and lazy merge do not make full use of memory resources, so their performance tends to be poor; (2) eager merge may lose performance in the second to last merge step or previous merge steps; (3) improved eager merge and optimistic merge are promising strategies, the former is a good choice if the available memory often changes, while the later is better when the available memory is stable.

6.5.4 Implementation issues and possible improvements

Although the memory-adaptive sort is able to adjust its sort space after each merge step, the amount and frequency of adjustments may affect the merge performance differently. If an adaptive merge changes a sort space whenever available memory changes, we have extra overhead due to frequent changes of sort space, while minor changes of the sort space may not improve the performance. Therefore, some memory adjustments should be avoided.

From the cost analysis of memory-static sort (Section 5.1), we know that the transfer cost changes slowly within the merge width $[\sqrt{n}, n]$. However, it changes very fast when the merge width is less than $\sqrt{n}$. As to when we should adjust a sort space, one possible policy is to increase sort space only when $w < \sqrt{n}$. Another policy is to increase sort space if the maximum merge width can be doubled, such that the sort is able to use more space even if $w > \sqrt{n}$. The second policy was adopted in our implementation. Other policies are also possible.

In order to reflect fairness among concurrent sorts, a *fair share* amount of memory is defined as the total sort memory space divided by the number of active sorts. If a sort has less memory than the fair share memory size, and the fair share memory is large enough to merge the remaining runs in a single step, the sort will wait for extra memory to do the last merge, rather than proceed with its current smaller

space. When the system is short of memory, a sort will release part of its space while keeping the amount of space close to this fair share amount. These policies are also used in our implementation.

Another issue is that the same amount of memory may improve system performance differently when it is used for different sorts. Sorts with many runs but very small space may require memory more urgently than other sorts. It might be helpful to prioritize these sorts and divide the available memory among them. This is a more delicate problem, which requires further study.

6.6 Summary

This chapter explored merge patterns aimed at reducing the amount of data transferred between disk and main memory.

For the case when the sort space remains constant throughout the external merge phase, we derived a formula for calculating the exact cost of optimum merge for equal size runs, and gave a lower bound and an upper bound on the optimum merge cost for variable length runs. We also provided some approximation formulas. From the analysis of optimum merge, clustering, and buffer size, we proposed methods to determine the optimum merge width and the optimum buffer size.

For the case that a sort is able to adjust its memory usage between merge steps, we considered four merge strategies: lazy merge, eager merge, improved eager merge, and optimistic merge. Experimental results showed that the last three strategies perform better than memory-static merge, while the last two are promising for practical use.

Chapter 7

Conclusion

The goal of this thesis was to improve overall system performance by better utilizing memory and I/O resources for sorting. It can be achieved by dynamically adjusting sort memory space, rescheduling run block read orders, and using merge patterns that reduce I/O.

7.1 Contributions

The main contribution of this thesis is a memory-conscious design for sorting, which takes into account fluctuation in available memory and concurrent sort jobs in the system. By using the proposed techniques, memory resources can be better utilized, thereby improving system sort throughput.

A dynamic memory adjustment technique was proposed for sorting. This technique adjusts sort space at run time in response to input data size and available memory space. It balances memory allocation among concurrent sorts to reduce the number of external sorts, and this improves overall system performance. A memory-adaptive mergesort was designed and implemented using this technique.

We experimentally showed that this technique enables sorts to adapt their memory usage gracefully to the actual input size and fluctuations in available memory space. Sort throughput was improved significantly compared with static memory allocation. Many ideas developed for memory adjustment are important not only to this memory-adaptive mergesort, but also to other memory-adaptive algorithms (see next section).

To reduce disk seek time during external merge, a set of read-ahead strategies were considered. The strategies include *equal buffering, extended forecasting, simple clustering,* and *clustering with atomic reads.* Extended forecasting improves overlapping of CPU and I/O time, but it does not reduce disk seeks. The other three strategies effectively reduce disk seeks. Simple clustering performs the best when there is no disk disturbance from other jobs running in the system, while equal buffering and clustering with atomic reads improves performance without being affected by the disk disturbance. The two clustering strategies exploit the existing order in the input better than equal buffering and extended forecasting. An analysis of these strategies resulted in formulas for estimating the performance improvement. These formulas provide close estimates for uniformly distributed random data. When sort keys are partially sorted, the improvements of the strategies are better than the improvements on random data, and therefore better than the improvements estimated using these formulas. Based on the formulas, we analyzed the tradeoff between using more buffers for read ahead and using large buffers, and provided guidelines for selecting proper buffer size.

The amount of data transferred between main memory and disk is determined by the merge pattern. When the sort space remains constant throughout the external merge phase, it is known how to construct an optimum merge pattern. This thesis provides a formula for calculating the exact cost of optimum merge for equal size

runs, and provides a lower bound and an upper bound on the optimum merge cost for variable length runs. In both cases, approximate formulas are also provided which closely estimate the exact costs. Based on these formulas, we analyzed the tradeoff between using large merge width and using more buffers for read ahead, and the tradeoff between using large merge width and using large buffers. We gave methods to choose the optimum merge width and optimum buffer size.

When sort space is adjustable during external merge, an optimum merge pattern cannot be guaranteed. Four merge strategies were considered for memory-adaptive merge: *lazy merge, eager merge, improved eager merge,* and *optimistic merge.* The last three strategies make better use of memory resources and normally perform better than static merge. The last two are promising for practical use. Improved eager merge is a good choice if the available memory often changes, while optimistic merge is better when the available memory is relatively stable.

7.2 Future Work

7.2.1 Dynamic memory adjustment

Chapter 5 presented one policy for dynamic memory adjustment, taking into account system sort space, sort stages, memory adjustment bounds, waiting, and fairness. Other policies can be employed and more factors can be taken into consideration. For example, each job in the system may have its own priority, and this should affect its memory allocation. High priority jobs should get larger memory space or get memory space sooner than low priority jobs. This can be achieved by setting higher memory adjustment bounds for high priority jobs and putting high priority jobs into a high priority wait queue respectively. Accordingly, the policy

may require more level wait queues.

In this thesis, the memory usage of a sort is adjusted by changing the number of buffers. We can also adjust buffer size dynamically, especially in the merge phase. Before each merge step, we can release some buffers or allocate new buffers, then divide some or all buffers of the sort into smaller ones, or combine small buffers into a larger one if their memory spaces are adjacent. The number of buffers will affect the merge width and read order scheduling. The adjusted buffer size based on the available memory may result in better performance. However, new policies are required to determine whether buffers should be divided or combined, and run block size should be as small as the smallest buffer size. One buffer may contain several run blocks. Thus sort space management, the merging algorithm, read order scheduling, and the memory adjustment policy will become complex.

Although the memory adjustment mechanism and policy were designed based on the three-phase sort algorithm introduced in Section 3.1, most of the ideas are applicable to other sort algorithms. For example, the (internal) distribution sort algorithm can be used for run formation. It distributes sort keys into buckets and then sort each bucket. Each bucket is composed of a variable number of small buffers. The sort will be able to adjust its memory space by dynamically changing the number of buffers in each bucket. We can also apply the memory adjustment technique if replacement selection is used for run formation phase. Memory adjustments can be done by expanding or shrinking the selection heap. This approach was adopted in [PCL93a], but they did not consider the input data size and the effects of several sorts running concurrently.

All the above methods are based on external mergesort, but dynamic memory adjustment can be extended to other external sort algorithms. External distribution sort is a good candidate. During distribution, we can dynamically change bucket

size or the number of buckets in response to input data size and available memory. If the input is larger than available memory, we can output data from one or more buckets. The bucket(s) to be chosen can be determined by the adjustment policy. After the first pass distribution, the data sizes of all the buckets are already known. Large buckets that cannot fit into memory require a second distribution, while small buckets can be sorted in memory. Based on available memory, we can either sort small buckets with the keys ready for output, or choose a large bucket for further distribution. Choice of bucket to be processed is a policy decision. When all buckets are smaller than the available memory, part of the sort space can be released to the system. Sort space can be adjusted before sorting each bucket or before each distribution step. All the decision issues will be managed by the memory adjustment policy. Many ideas developed for adaptive mergesort, such as sort stages, memory adjustment bounds, and waiting, will still be useful, but the details of the memory adjustment mechanism and policy will be very different.

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

The technique is more important to hash join algorithms. Memory adjustment for hash joins has been studied by [ZG90], [PCL93b], and [DG94]. However, their work focused on how a single join can use extra space or release part of its space to affect I/O transfer unit size. They did not take into account the memory requirements in different stages of a join and did not consider balancing memory allocation among concurrent joins. We can develop memory-adaptive joins based on the ideas proposed in this thesis, making concurrent joins that are competing for

memory resources cooperate with each other, therefore improving overall system performance. In addition, we can develop policies to balance memory allocation among all types of memory-adaptive jobs (sorts and joins).

7.2.2 I/O improvement

Chapter 6 proposed several strategies to reschedule the read order of run blocks and provided formulas to estimate the performance improvement. However, all the formulas are derived based on random input data and equal size runs. When the input data is partially sorted, the clustering algorithm exploits the existing order by using floating buffering, but the estimating of the performance improvement is difficult. How to estimate the performance effects of different read strategies for variable-length runs and partially sorted data is still an open question.

All the analysis and experiments of various read strategies were based on the assumption that runs are stored on a single disk. However, using multiple disks is common in modern systems. Data striping allows parallel reads and writes to increase disk bandwidth and improve overall disk utilization. It is not known yet how our read strategies will behave on multiple disks along with the data striping technique. New strategies may be needed for multiple disks, taking data striping into consideration. The goal is not only to reduce disk seeks but, more importantly, to balance the workload among multiple disks and maximize the parallelism of I/O operations.

During run formation for the last run, instead of writing the entire run to disk, we can keep part of the run in memory if extra space is available. This will reduce the amount of data transferred between main memory and disk. Depending on the available memory and the last run size, we can keep in memory part of the last run, the entire last run, or the entire last run plus part of the second to last run. The

other part of sort space is used for runs residing on disk during external merging. Since part of the memory is used to keep the entire last run and perhaps part of the second to last run, some space may stay unused until the very end of external merging. There is a tradeoff between keeping the runs in-memory and using the memory for clustering run blocks.

7.2.3 Summary

The techniques for dynamic memory adjustment and I/O improvement can be studied further in the following areas:

1. New policies for dynamic memory adjustment applied to external mergesort;
2. Memory adjustment for other sort algorithms, such as distribution sort;
3. Memory adjustment for other memory intensive jobs, such as joins;
4. Memory adjustment policies for different types of memory-adaptive jobs;
5. Performance estimate for variable-length runs and partially sorted data;
6. Read strategies for multiple disks along with the data striping technique;
7. Partial writing during run formation.

Appendix A

Variable Run Lengths

A.1 Run lengths from sort testbed

Following run lengths were collected while a 100M data set was sorted on the sort testbed using memory-adaptive sort. The total available memory in the system is 4M and sort buffer size is 64K.

```
Data size (Mb): 100.0
Num of runs: 38
Run sizes (Mb):

 3.438 0.312 0.375 0.562 0.750 0.938 1.125 1.312 1.500 1.688
 1.875 2.062 2.250 2.438 2.625 2.812 3.000 3.188 3.375 3.438
 3.438 3.438 3.438 3.438 3.438 3.438 3.438 3.438 3.438 3.438
 3.438 3.438 3.438 3.438 3.438 3.438 3.438 2.500
```

A.2 Run lengths from triangular probability distribution

Replacement selection is a popular algorithm for run formation since it is able to produce runs larger than the available memory size. Usually it produces runs of variable length depending on the existing order in the input data. However the run length distribution is not known.

Here we assume that the probabilities of longer runs are smaller than the probability of shorter runs. The following run lengths were randomly drawn from a triangular probability distribution (as shown in Figure A.1). The run length is between 1M and 5M.

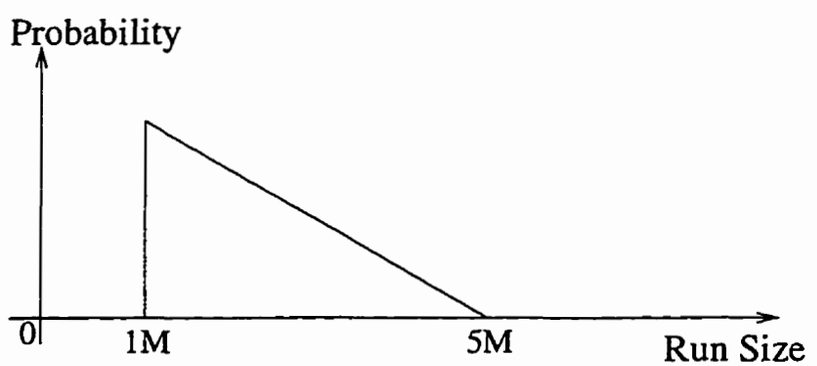


Figure A.1: Triangular probability distribution for run length

```
Data size (Mb): 227.8
Num of runs: 100
Run sizes (Mb):
 2.1 1.2 1.5 2.2 4.0 1.2 2.7 1.3 2.0 1.1 1.0 1.8 1.5 1.7 4.5 2.2 3.0
 2.5 3.0 3.1 3.3 1.2 2.5 1.6 1.6 3.8 2.1 1.8 2.4 3.1 3.9 3.5 3.5 2.6
 3.0 2.3 1.8 1.7 1.3 3.3 1.8 2.0 4.4 1.1 1.3 4.2 2.9 1.8 3.1 3.0 4.1
 1.0 1.6 3.0 1.4 2.3 1.0 4.1 1.6 1.0 1.9 3.8 2.3 1.1 2.3 1.4 2.1 1.4
 2.0 1.8 3.5 1.9 1.7 1.7 1.0 1.2 2.1 2.7 1.1 1.8 3.0 1.4 1.6 1.4 1.5
 1.5 3.6 1.0 2.5 1.8 2.6 2.9 4.0 1.4 3.3 4.2 3.0 1.9 2.6 3.2
```

Appendix B

Specification of ST-15150W Disk Drive

(From http://www.seagate.com/cgi-bin/view.cgi?/scsi/st15150w.txt)

ST-15150W

UNFORMATTED CAPACITY (MB)	5048
FORMATTED CAPACITY (xx SECTORS) (MB)	4294
AVERAGE SECTORS PER TRACK	107 rounded down
ACTUATOR TYPE	ROTARY VOICE COIL
TRACKS	77,931
CYLINDERS	3,711 user
HEADS PHYSICAL	21
DISCS (3.5 in)	11
MEDIA TYPE	THIN FILM
RECORDING METHOD	ZBR RLL (1,7)
INTERNAL TRANSFER RATE (mbits/sec)	47.4 to 71.9
EXTERNAL TRANSFER RATE (mbyte/sec)	20 Sync
SPINDLE SPEED (RPM)	7,200
AVERAGE LATENCY (mSEC)	4.17
BUFFER	1024 KByte
Read Look-Ahead, Adaptive, Multi-Segmented Cache	

```
INTERFACE _____________________________SCSI-2 FAST WIDE
                                        ASA II
BYTES PER TRACK _______________________64,773 average
SECTORS PER DRIVE _____________________8,388,315
TPI (TRACKS PER INCH) _________________4,048
BPI (BITS PER INCH) ___________________52,187
AVERAGE ACCESS (ms) (read/write)_______8.0/9.0
  Drive level without controller overhead
SINGLE TRACK SEEK (ms) ________________0.6/0.9
MAX FULL SEEK (ms) ____________________17/19
MTBF (power-on hours) _________________800,000
POWER DISSIPATION (watts/BTUs) Active ____15/51
                                 Idle ______12/41
POWER REQUIREMENTS: +12V START-UP (amps) _2.18
                    +12V TYPICAL (amps) __0.83 idle
                    +5V START-UP (amps) __1.0
                    +5V TYPICAL (amps) ___0.76 idle
                    IDLE (watts) _________14
LANDING ZONE (cyl) ____________________AUTO PARK
IBM AT DRIVE TYPE _____________________0 or NONE

Physical:
                  Height (inches/mm):     1.63/41.4
                  Width  (inches/mm):     4.00/101.6
                  Depth  (inches/mm):     5.97/151.6
                  Weight    (lbs/kg):     2.3/1.04
```

Already low-level formatted at the factory with 9 spare sectors per cylinder and 1 spare cylinder per unit.

ZBR = Zone Bit Recording = Variable sectors per track

Bibliography

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, 1997.

[AEA85] Anon-Et-Al. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, 1985. Also in *Readings in Database Systems*, M.H. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[Akl85] Selim G. Akl. *Parallel Sorting Algorithms.* Academic Press, Toronto, 1985.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[BBDW83] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Transactions on Database Systems*, 8(3):324–353, Dec. 1983.

[BBW88] Micah Beck, Dina Bitton, and W. Kevin Wilkinson. Sorting Large Files on a Backend Multiprocessor. *IEEE Trans. on Computers*, 37(7):769–778, July 1988.

[BDHM84] Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon. A Taxonomy of Parallel Sorting. *ACM Computing Surveys*, 16(3):287–318, Sept. 1984.

[BGK90] B.A.W Baugsto, J.F Greipsland, and J. Kamerbeek. Sorting Large Data Files on POMA. In *Proceedings of COMPAR-90 VAPPIV*, pages 536–547, Sept. 1990. Springer Verlag Lecture Notes No. 357.

[CGMP91] Walter Cunto, Gaston H. Gonnet, J. Ian Munro, and Patricio V. Poblete. Fringe Analysis for Extquick: An *in situ* Distributive External Sorting Algorithm. *Information and Computation*, 92(2):141–160, June 1991.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1989.

[DG94] Diane L. Davison and Goetz Graefe. Memory-Contention Responsive Hash Joins. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 379–390, 1994.

[DL92] W. R. Dufrene and F. C. Lin. An Efficient External Sort Algorithm with no Additional Space. *The Computer Journal*, 35(3):308–310, Mar. 1992.

[DNS91] D. Dewitt, J.F. Naughton, and D.A. Schneider. Parallel Sorting on a Shared-nothing Architecture using Probabilistic Splitting. In

Proc. of the International Conference on Parallel and Distributed Information Systems, pages 280–291, 1991.

[EC91] Vladimir Estivill-Castro. *Sorting and Measure of Disorder.* PhD thesis, University of Waterloo, 1991.

[ECW92] Vladimir Estivill-Castro and Derick Wood. A Survey of Adaptive Sorting Algorithms. *ACM Computing Surveys*, 24(4):442–475, Dec. 1992.

[ECW94] Vladimir Estivill-Castro and Derick Wood. Foundations of Faster External Sorting. In *Proceedings of the Fourteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 414–425, 1994.

[FL96] James D. Fix and Richard E. Ladner. Sorting by Parallel Insertion on a One-Dimensional Sub-Bus Array. Technical Report UW-CSE-96-09-02, University of Washington, Sept. 16 1996.

[GBY91] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1991.

[Gra90] Goetz Graefe. Parallel External Sorting in Volcano. Technical Report CU-CS-459, University of Colorado at Boulder, 1990.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GT92] Goetz Graefe and Shreekant S. Thakkar. Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor. *Software — Practice and Experience*, 22(7):495–517, July 1992.

[IBM95] IBM. *DATABASE 2, Administration Guide for common servers, Version 2.* IBM, June 1 1995.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. of ACM SIGMOD Conf.*, pages 268–277, May 1991.

[ID90] Balakrishna R. Iyer and Daniel M. Dias. System Issues in Parallel Sorting for Database Systems. In *Proc. of Int. Conf. on Data Engineering*, pages 246–255, Feb. 1990.

[Kim86] M.Y. Kim. Synchronized Disk Interleaving. *IEEE TOCS*, 35(11):978–988, Nov. 1986.

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

[Kwa86] Sai Choi Kwan. *External Sorting: I/O Analysis and Parallel Processing Techniques.* PhD thesis, University of Washington, 1986.

[LL97a] A. LaMarca and R.E. Ladner. The Influence of Caches on the Performance of Sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium ' on Discrete Algorithms*, pages 370–379, Jan. 1997.

[LL97b] Anthony LaMarca and Richard E. Ladner. The Influence of Caches on the Performance of Heaps. Technical Report UW-CSE-96-02-03, University of Washington, Jan. 6 1997.

[Lor75] Harold Lorin. *Sorting and Sort Systems.* The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1975.

[LV85] Eugene E. Lindstrom and Jeffrey Scott Vitter. The Design and Analysis of BucketSort for Bubble Memory Secondary Storage. *IEEE Trans. on Computers*, C-34(3):218–233, Mar. 1985.

[Man87] Masud Mansuripur. *Introduction to Information Theory*, pages 29–35. Prentice-Hall Inc., 1987.

[Man89] Udi Manber. *Introduction to Algorithms, A Creative Approach.* Addison-Wesley, 1989.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 233–242, 1994.

[NKG97] Chris Nyberg, Charles Koester, and Jim Gray. Nsort: a Parallel Sorting Program for NUMA and SMP Machines, *http://www.ordinal.com/white/whitepaper.shtml.* Technical report, Ordinal Technology Corp., June 5, 1997.

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

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

[Qui88] M. J. Quinn. Parallel Sorting Algorithms for Tightly Coupled Multiprocessors. *Parallel Computing*, 6:349–367, June 1988.

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

[RSS85] Doron Rotem, Nicola Santoro, and Jeffery B. Sidney. Distributed Sorting. *IEEE Trans. on Computers*, C-34(4):372–375, Apr. 1985.

[RW94] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, Mar. 1994.

[Sal89] Betty Salzberg. Merging Sorted Runs Using Large Main Memory. *Acta Informatica*, 27:195–215, 1989.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk Striping. In *Proc. of Int. Conf. on Data Engineering*, pages 336–342, 1986.

[Ver88] A. Inkeri Verkamo. External Quicksort. *Performance Evaluation*, pages 271–288, Aug. 1988.

[Ver89] A. Inkeri Verkamo. Performance Comparison of Distributive and Mergesort as External Sorting Algorithms. *The Journal of Systems and Software*, pages 187–200, Oct. 1989.

[Weg85] Lutz M. Wegner. Quicksort for Equal Keys. *IEEE Trans. on Computers*, C-34(4):362–367, Apr. 1985.

[WT89] Lutz M. Wegner and Jukka I. Teuhola. The External Heapsort. *IEEE Trans. on Software Engineering*, 15(7):917–925, July 1989.

[YY83] A. M. Yaglom and I. M. Yaglom. *Probability and Information*, pages 160–161. D. Reidel Publishing Company, 1983.

[ZG90] Hansjorg Zeller and Jim Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 186–197, 1990.

[Zhe92] Luo Quan Zheng. Speeding up External Mergesort. Master's thesis, University of Waterloo, 1992.

[ZL96a] Weiye Zhang and Per-Åke Larson. A Memory-Adaptive Sort (MASORT) for Database Systems. In *Proceedings of the 1996 IBM CAS Conference (CASCON'96)*, pages 194–207, Nov. 1996.

[ZL96b] Luo Quan Zheng and Per-Åke Larson. Speeding up External Mergesort. *IEEE Trans. on Knowledge and Data Engineering*, 8(2):322–332, Apr. 1996.