Buffering and Read-Ahead Strategies for External Mergesort

Weiye Zhang
University of Waterloo
weiyez@microsoft.com

Per-Åke Larson
Microsoft Research
palarson@microsoft.com

Abstract

The elapsed time for external mergesort is normally dominated by I/O time. This paper is focused on reducing I/O time during the merge phase. Three new buffering and read-ahead strategies are proposed, called equal buffering, extended forecasting and clustering. They exploit the fact that virtually all modern disks perform caching and sequential read-ahead. The latter two also collect information during run formation (the last key of each run block) which is then used to preplan reading. For random input data, extended forecasting and clustering were found to reduce merge time by 30% compared with traditional double buffering. Clustering exploits any temporal skew in input runs to further reduce the number of seeks.

Authors' current address: Microsoft, One Microsoft Way, Redmond, WA 98052-6399, U.S.A.

1 Introduction

Sorting is a frequent operation in database systems. It is used not only to produce sorted output, but also in many sort-based algorithms, such as grouping with aggregation, duplicate removal, sort-merge join, as well as set operations including union, intersect, and except [Gra93] [IBM95].

**Proceedings of the 24th VLDB Conference
New York, USA, 1998**

External mergesort is the most commonly used algorithm for large-scale sorting. It has a run formation phase, which produces sorted runs, and a merge phase, which merges the runs into sorted output. This paper focuses on how to improve I/O performance during the merge phase because this phase is typically I/O bound. We assume that the merge pattern (what runs to merge when) and the amount of memory available for merging have already been decided. Our goal here is is to develop buffering and read-ahead strategies that reduce the I/O time for individual merge steps. Other issues related to sorting, such as, selecting merge width, merge pattern, and balancing memory among competing sorts, are discussed in more detail elsewhere [Zha97], [ZL97].

Double buffering and forecasting (see [Knu73]) are the standard buffering and read-ahead strategies used for merging. Suppose we are merging n runs. Double buffering divides the available memory into $2n$ buffers of equal size and assigns two buffers to each run. A run uses one buffer for read-ahead and one for merging. Forecasting divides the available memory into $n + 1$ buffers, assigns one buffer to each run, and uses one as an unassigned read-ahead buffer. Whenever a buffer becomes empty, a read is immediately issued to fill it. The read is always from the run that will be the first one to run out of data in memory. This can be determined by comparing the keys of the last record from each full buffer.

The standard approach to improving I/O performance is simply to increase buffer size, as advocated, for example, in [Sal89]. In this paper we propose and analyze three alternative buffering and read-ahead strategies that all perform better than double buffering and forecasting. Experimental results show a 30% reduction in merge time compared with double buffering.

The three new strategies are called equal buffering, extended forecasting, and block clustering. Equal buffering is an enhanced version of double buffering.

Extended forecasting is based on standard forecasting. Both exploit the fact that virtually all modern disk drives perform caching and sequential prefetch. Extended forecasting also avoids disk wait times by using two read-ahead buffers.

During merging, run blocks are consumed in a particular sequence and are usually read in that order. Extra buffer space makes it possible to read data blocks in an order that is different from the order in which they are consumed during merging. We can then try to read them in an order that minimizes total I/O time. This idea was proposed by Zheng and Larson [Zhe92] [ZL96], including a heuristic for computing read sequences. Estivill-Castro and Wood [ECW94] 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 adjacent on disk. In this paper we propose and analyze a new algorithm, called block clustering, for computing read sequences. The new algorithm does not require knowledge about physical disk layout and is designed to work better when there are multiple jobs competing for the run disk(s).

The rest of this paper is organized as follows. Section 2 provides some background on techniques used in modern disk controllers and on disk reading. Section 3 describes the new read-ahead strategies. Section 4 and 5 analyze the performance of these strategies for random input and for skewed input, respectively. Formulas are derived for estimating merge time and experimental results are provided.

2 Preliminaries

Disks are getting smarter

Modern disks are no longer simple, dumb devices. Disk controllers have become quite sophisticated with a considerably amount of memory and processing power. This section outlines four, widely used, techniques that affect the performance characteristics of disks: multiple zone recording, command reordering, and caching with sequential prefetch and write reordering. Reference [Cor97] provides a good introduction to current disk technology.

Traditionally, all tracks on a disk had the same number of sectors. This is no longer true: virtually all modern disks use multiple zone recording. The disk is divided into zones and all tracks within a zone have the same number of sectors but zones closer to the center of the disk (shorter tracks) have fewer sectors than zones closer to the edge (longer tracks). The outermost tracks may have up to twice as many sectors as the innermost tracks. This means that the data transfer rate is no longer independent of the position on disk, i.e. how fast data can be read depends on its location on the disk.

More advanced disk controllers have a queue for incoming commands. If the queue contains multiple read or write commands, the controller does not necessarily serve them in the order of arrival. It may reorder them to reduce seek time and/or rotational latency. This is in addition to any reordering that might be done by the operating system's I/O scheduler.

The current generation of disk controllers typically have 0.5MB - 2MB of cache memory. The cache is used for two main purposes, namely, sequential prefetch and write reordering. When a disk has completed a read request, it may continue reading forward on the same or even subsequent tracks. If the application is reading a file sequentially, part or all of the data requested by its next read is already in the disk cache and can be returned almost instantly. There are many variations in how this is implemented, for example, exactly what triggers sequential prefetch and how far ahead prefetching is allowed to proceed.

One important characteristic is the number of sequential streams the controller can keep track of concurrently and how the cache space is divided among streams. In the simplest case, the controller keeps track of only one stream at a time and all of the cache space is dedicated to that stream. A more sophisticated controller might be designed to keep track of a fixed number of streams, say the last four active streams, and have the cache space statically divided among the the streams. Some controllers go even further by dynamically deciding how many streams to keep track of and how to divide the cache space among the streams. From a practical point of view this means that reading concurrently from multiple sequential streams does not necessarily disable prefetching. For example, assume that we are reading sequentially from two files but interleave reads from the two files. Depending on its design, the disk controller may recognize the two sequential streams and start prefetching, say, one track at a time. This reduces the number of disk seeks by, in essence, consolidating small reads into fewer and larger, track-at-a-time, reads.

Write reordering, also known as write caching, means that write requests are acknowledged as soon as the data has been copied into the cache but the actual writing to the disk occurs sometime later. A disk controller that implements write reordering may then perform pending writes in an order that minimizes disk overhead. Note that, typically, there is no guarantee that the cached data will be written to disk in case of power failure – a serious problem in a transactional environment.

These enhancements make it difficult to model the behavior of modern disks. However, it is still the case

that (a) disk seeks and rotational latency heavily affect the total disk access time for random access; and (b) sequential access is much faster than random access.

Reading clusters sequentially

The reading done during merging consists of multiple interleaved, sequential streams, i.e. we read some number of adjacent blocks from one run, seek to another run, read a group of adjacent blocks there, seek to a third run, and so on. The clusters of adjacent blocks read may be of fixed or variable size depending on the buffering and read-ahead strategy.

Assume for the moment that we read clusters of fixed size, say 512KB, and that each cluster is read into a contiguous area in memory. The actual reading can then be done as a single physical read of size 512KB or as a sequence of smaller reads, say 16 reads of size 32KB. What are the effects of increasing the number of physical reads? It will increase CPU time slightly because each read request requires some processing but not the transfer time or the number of seeks because of disk caching and prefetching. However, issuing smaller reads makes data available sooner which may allow the merge to resume sooner. In our example, merging can resume as soon as the first 32KB have arrived instead of waiting for the complete 512KB.

If using smaller reads, it is important to issue them together as a batch. This reduces the chance of interference from other concurrent jobs and improves the chance of benefiting from disk caching and sequential prefetch.

Most operating systems, in fact, break large application read requests into a number of smaller reads. Sometimes this is necessary because the requested data is not adjacent on disk. In other cases, it is because of the space allocation mechanism used for I/O buffers or the file cache. For example, Windows NT breaks large reads into a sequence of 64KB reads.

In a typical database environment, the memory available for input buffers is not contiguous but consists of smaller extents scattered in memory. If the operating system supports scatter-read and gather-write, we can still decide on the size and number of physical reads independently of the extent size [1]. This is the scenario modelled in our sort testbed which was used for the experiments reported later in the paper. In the rest of this paper, memory available for input buffers is assumed to consists of fixed size extents, called buffer pages. Each buffer page occupies a contiguous area in memory but that is not necessarily true for the complete set of buffer pages. In our testbed, buffer pages had a minimum size of 32KB.

[1] On Unix systems scatter-read and gather-write capability is provided through the `readv()` and `writev()` system calls.

3 Buffering Strategies

3.1 Fixed buffering

Fixed buffering assigns all buffer pages to runs before a merge step starts and each buffer page remains dedicated to the same run throughout the merge step. Buffer pages can be assigned to runs in many ways but each run must have at least one page. *Equal buffering* is a policy that assigns the same number of buffer pages to each run.

In addition to buffer space assignment, we must also decide when to trigger reads. Consider a situation when a run has been assigned m buffer pages and assume that they are initially full. As merging proceeds, buffer pages will slowly be emptied, one by one. We must decide at what point to initiate reading, which may range from issuing a read as soon as there is one empty buffer page to waiting until $m-1$ pages have been emptied. The longer we wait, the larger the clusters read, which reduces the number of disk seeks. However, we still want to achieve full overlap of CPU and I/O operations. This implies that there needs to remain enough data in memory when issuing a cluster read for the merge process to continue without interruption until additional data arrives, i.e. until the first read request completes.

The traditional *double buffering* algorithm follows the simple rule of initiating reading when half of the buffer space allocated to a run is empty. We propose a version of equal buffering that one might call equal buffering with lazy triggering of batched reads. That is, reads are triggered when there is only one full buffer page left. When a read is triggered, we issue a batch of $m-1$ smaller read requests. Why wait instead of issuing a read as soon as a buffer page becomes empty? If we issue reads immediately, there will be more seeks because requests for different runs will be randomly mixed. By waiting we create batches of $m-1$ sequential reads, thereby reducing the number of seeks by a factor of (almost) $m-1$. As mentioned earlier, issuing a batch of small reads instead of a single large read does not increase the read time (because of prefetching) and it makes data available to the merge process sooner.

Fixed buffering does not fully utilize the available buffer space. When $m-1$ buffer pages become empty, the sort cannot issue read requests to fill them unless they belong to the same run. To use buffer space more efficiently, buffer pages should not be dedicated to a specific run, but serve any run on demand.

3.2 Extended forecasting

Traditional forecasting uses one merge buffer per run plus one extra buffer for read ahead [Knu73]. When a

block from each run resides in memory, we can determine which buffer will be emptied first by comparing the last keys in the buffers. Whenever a buffer becomes empty, the next block from that run is read, normally using a single large request. We propose two improvements to forecasting: using more than one extra buffer and issuing a batch of small reads instead of a single large read. We call the resulting scheme *extended forecasting.*

Traditional forecasting cannot achieve completely full overlap of reading and merging because the next read cannot start until the current one has completed. The disk always experiences a short wait between read requests. To maintain full utilization of the disk, we need to keep at least one read request in the I/O queue at all times. This requires more than one extra buffer. Furthermore, we must decide from which run to read next before the current read has finished.

Data from each run is read in blocks equal to the buffer size. The order in which run data blocks are consumed by merging is called the *consumption sequence.* 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.

Figure 1 shows an example with three runs, each containing three blocks. The block numbers reflect the order in which the blocks were written to disk.

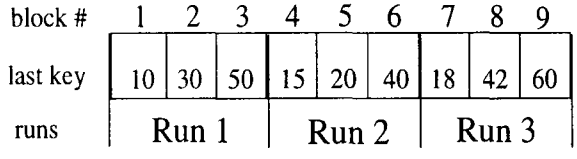

Figure 1: An example of the consumption sequence

The first block of each run is required to start the merge process. Block 1 will finish first because it has the smallest last key. The next block of run 1 (block 2) must then be brought into memory. Block 4 is the next one to finish. The next block to be read is the second block of run 2 (block 5), and so on. The resulting consumption sequence is shown in the diagram.

The consumption sequence is completely determined by the last key of each run block. It can be computed by extracting the last key of each block during run formation and simply sorting the set of extracted keys. Once the consumption sequence has been computed, extended forecasting reads the run blocks in that order.

Our second modification is to perform the actual reading by a batch of small read requests. Suppose each buffer consists of m buffer pages. Instead of waiting until a complete buffer (m pages) from some run has been emptied, we issue a batch of read requests as soon as there are m free pages in total, regardless of which runs the pages belonged to. This way reading starts sooner. We still issue the read requests as a batch to exploit disk prefetching and reduce the chance of interference from other jobs accessing the same disk.

To get merging started as quickly as possible, we initially read just one page-full from each run (instead of a full buffer of m pages).

3.3 Clustering

Although a merge process consumes run blocks in a particular order, the run blocks can be read in a different order if extra buffer pages are available. The extra buffer pages can be used for storing data that is 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 paper, for any two sequences X and Y, we define $X \subseteq Y$ to mean that the set of elements in X is a subset of those in Y.

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 buffer pages, if the last 5 blocks of each run are read at the beginning in the read sequence, no free buffer pages are left to read the first block of each run. The merge process cannot proceed without overwriting some of the full buffer pages and rereading the same data later. A read sequence is feasible if it guarantees that the merge process terminates with each data block having been read exactly once.

It is obvious that the consumption sequence is a feasible read sequence, provided that there are at least as many buffer pages as there are runs. In fact, the consumption sequence is the read sequence used by traditional forecasting and by extended forecasting.

Although there is a finite number of feasible read sequences, it is not known if there is an efficient algorithm for computing the optimum sequence with minimum disk seek time. Finding the optimum sequence by trying all the read sequences is prohibitively expensive. Research has been focusing on using heuristics.

In this paper, we introduce a heuristic algorithm called *(block) clustering* which attempts to group 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. The target buffer pages are normally not adjacent in memory so the actual reading

of a cluster is done by multiple physical reads. However, the cluster is read sequentially so only the first read of the cluster requires a seek.

The read requests for a cluster are issued as a batch so they will not be intermixed with write requests from the same merge, that is, the sort does not interfere with itself. However, other jobs in the system may access the same disk and may interrupt the smooth sequential processing of read batches [2].

To overlap processing and read time, there must be enough full buffer pages for the merge process to proceed and enough free buffer pages for I/O to read an additional cluster. Theorem 1 defines a condition for testing the feasibility of a read sequence.

Theorem 1 *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 as sequential batches, 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 buffer pages 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.

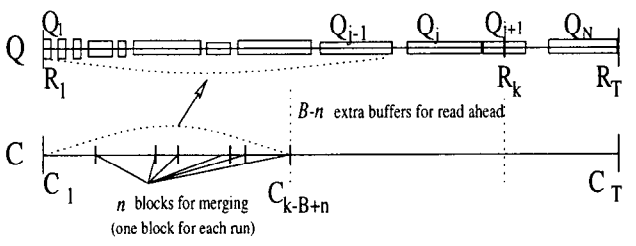

Figure 2: Feasibility of read sequence

At any stage when $B < k \leq T$ (as shown in Figure 2), 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} has been 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 have been read into memory. Therefore, the merge process will terminate. So the condition in the theorem guarantees the feasibility of the read sequence. □

The following algorithm computes a feasible read sequence. The consumption sequence is taken as the initial read sequence, with each block forming a cluster of size one. A block is then combined with the previous cluster for the same run as long as the feasibility of the read sequence is preserved. 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 block clustering
Input: consumption sequence $C = \{C_1..C_T\}$,
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
// Initialize Q to be the consumption sequence
$Q := C$;
for $i := 1$ **to** T
$L[i] := 1$; *// Set initial cluster size to 1*
endfor;
// lastCl: index of the last cluster before Q[i]
$lastCl := n$;
for $i := n + 1$ **to** T
// Search each previous cluster to find
// the one with the same run as Q[i]
for $j :=$ *lastCl* **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 *// combine Q[i] with cluster Q[k]*
$L[k]$ ++;
else *// Q[i] becomes the new last cluster*
lastCl ++;
$Q[lastCl] := Q[i]$;
endif;

[2] In our sort testbed, all sorts in the system send their I/O requests to an I/O request queue which is served by I/O agents (two per disk)in FIFO order. To make sure that a read batch is not intermixed with reads or writes from other concurrent sorts, the queue is first locked, the batch of read requests added to the queue, and the queue unlocked.

endfor;
N := lastCl;
end

To check feasibility efficiently, our implementation makes use of a free buffer count array $F = \{F_1, F_2, ..., F_T\}$. F_i records the number of free buffer pages 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_{lastCl} are reduced by one. To guarantee that there are enough buffer pages to read a cluster while the merge process can proceed, it is required that $F_i \geq L_{i+1}$ for all i. For each Q_i, the algorithm needs only check cluster Q_{lastCl} down to cluster Q_j that $F_j = L_{j+1}$ or $F_{j-1} = L_j$. This technique is more efficient than applying Theorem 1 directly.

4 Performance for Random Input

In this section we derive formulas for estimating the performance of the different read strategies. It is assumed that input data is randomly distributed and all runs have the same length. The number of cluster reads initiated is used as an approximate measure of the number of disk seeks (regardless of how many physical reads are issued). Here is the notation used in the rest of this paper.

D : input size,
n : number of runs,
b : number of buffer pages available,
p : buffer page size,
M : total memory space used for buffers, $M = bp$,
N : number of cluster reads,
R : average size of a cluster read ($D = NR$),

Subscripts D, E, F, and C represent double buffering, equal buffering, extended forecasting, and clustering respectively.

4.1 Estimating number of cluster reads

Double buffering

Double buffering divides the available buffer space equally among the n runs so, on average, a run gets pb/n bytes. Provided every run has at least two buffer pages, that is, $b \geq 2n$, cluster reads will have an average size of $pb/2n$. Otherwise, a cluster read will cover exactly one buffer page. This gives us the following formula for the total number of cluster reads:

$$N_D = \begin{cases} D/(pb/2n) & \text{if } b \geq 2n \\ D/p & \text{if } n \leq b < 2n \end{cases} . \quad (1)$$

Equal buffering

The average number of buffer pages per run is b/n. (More precisely, some runs are assigned $\lfloor b/n \rfloor$ buffer pages and some $\lceil b/n \rceil$ buffer pages but the average is b/n.) If a run has t buffer pages, $t \geq 2$, its cluster reads will be for $t - 1$ pages. Otherwise, its reads will always be for one page. If all runs have more than two buffer pages each, that is, if $b \geq 2n$, the average cluster read will therefore of size $b/n - 1$ pages and otherwise one page. This gives us the following formula for the total number of cluster reads:

$$N_E = \begin{cases} D/(p(b/n - 1)) & \text{if } b \geq 2n \\ D/p & \text{if } n \leq b < 2n \end{cases} . \quad (2)$$

Extended forecasting

Extended forecasting uses two read-ahead buffers and divides the available buffer space equally among the n runs and the two read-ahead buffers. So the space assigned to each run is $pb/(n+2)$, which is also the average size of read clusters. This gives us the following simple formula for the total number of cluster reads:

$$N_F = D/(pb/(n+2)) = (n+2)D/(pb) . \quad (3)$$

Clustering

When input data is randomly distributed and all runs are of the same length, data from all runs will be consumed at the same rate during merging. In such a situation, the consumption sequence can be approximated by an ideal consumption sequence shown in Figure 3. n is the number of runs and within each sequence of n blocks, there is one block from each run.

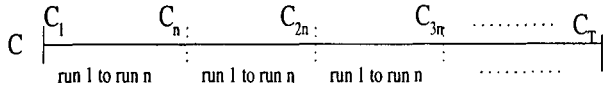

Figure 3: Ideal consumption sequence.

During clustering, a block is combined with the previous block of the same run as long as feasibility is preserved. Therefore, blocks C_{n+1}, C_{n+2}, ..., C_{2n} are combined with blocks C_1, C_2, ..., C_n, respectively and form n clusters. Each cluster contains two blocks. Then blocks C_{2n+1}, C_{2n+2}, ..., C_{3n} are combined with these clusters. The cluster size grows until feasibility can no longer be preserved. The remaining blocks will be combined to form a second set of clusters, and so on. The resulting clusters are all of the same size. This yields the ideal read sequence shown in Figure 4, where Q_i represent a cluster, i.e., a sequence of adjacent blocks from the same run. Within each sequence of n clusters, there is one cluster from each run.

For our clustering algorithm, the sort sends the read requests of a cluster as a batch. The average cluster size is the average read size R_C. Thus $n * R_C$ buffer

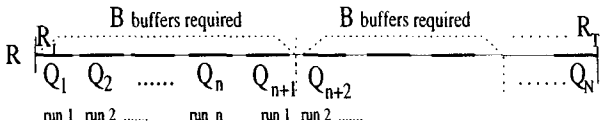

Figure 4: Ideal read sequence.

space is required to keep the first n clusters so that the merge process can start, while R_C buffer space is required for Q_{n+1} to overlap the merge processing and read time (as shown in Figure 4). We assume that all the runs are consumed at the same rate. When there are enough free buffer pages for the next cluster, another batch of reads is issued. By the time the first n clusters are finished, there are enough buffer pages 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 buffer pages to read cluster Q_{2n+1} at the same time. Thus the merge process is able to terminate with $(n+1) * R_C$ buffer space. Then we have $pb = R_C * (n+1)$, that is, $R_C = pb/(n+1)$. This results in the following formula for estimating the total number of cluster reads:

$$N_C = D/R_C = (n+1)D/(pb) \ . \tag{4}$$

Numerical results

Figure 5 show theoretical and experimental results for the four read-ahead strategies. The case shown in the graph is for an input size of 50MB, divided into 15 runs of the same size. Records were 64 bytes long with a randomly generated key of 10 bytes. Each point plotted in the diagram represents the average computed from five experiments, using different input sets.

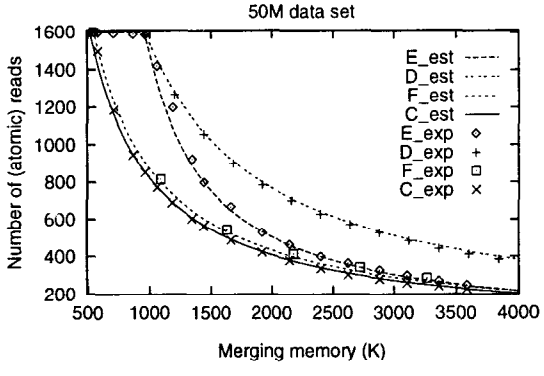

Figure 5: Theoretical and observed number of cluster reads.

It is difficult to distinguish the different line types in the figure but the experimental results match the theoretical results very closely, even for clustering. Clustering has the fewest cluster reads for all memory sizes. Its cluster reads (seeks) are slightly fewer than for extended forecasting but the difference is minimal. Not surprisingly, double buffering results in the most seeks, about twice as many as clustering and extended forecasting. Equal buffering performs quite well when the amount of buffer memory is reasonably large.

4.2 Estimating merge time

Merging is normally I/O bound so the elapsed time of a merge step is determined by the I/O time. The traditional disk model estimates I/O time by adding transfer time and (average) seek time. The traditional formula is

$$T = tD + sN \ , \tag{5}$$

where T is total elapsed time (sec), D is data size (MB), t is the transfer time for 1M data (sec/MB), N is the number of disk seeks, and s is the average disk seek time (including rotational latency). Given the additional complexity of modern disks, this simple model may no longer be an acceptable approximation.

As mentioned previously, we approximate the number of seeks with the number of cluster reads, i.e. we assume that adjacent clusters are from different runs. Our sort testbed uses a 500MB raw partition on one disk, a Seagate ST-15150W. Experimentally, we found that $t \approx 0.3$ sec/Mbytes and $s \approx 0.007$ sec.

We ran experiments on many randomly generated data sets to collect timing data. For each data set, each read-ahead strategy was tested, and the experiment was repeated using different memory sizes. Each point plotted in the diagrams represents the average computed from five experiments. The five experiments used different data sets of the same size that were produced using different random seeds. Except for standard double buffering and standard forecasting, the memory available for input buffers was divided into buffer pages of size 32KB. When a cluster was read using a batch of read requests, each request read 32KB. For standard double buffering and standard forecasting, memory was contiguous. Each buffer was assigned a contiguous area and each read request was of the same size as the buffer.

Numerical results

Figures 6 to 9 show estimated and observed results for input of size 50MB. The estimates were computed using formula 5, where N is replaced by the number of cluster reads given by formulas 2, 3, and 4, respectively. Experiments using other data sizes (5M to 100M) produced similar results.

Figures 6 and 8 each plot results from two series of experiments. The difference is in the size of physical reads: the two series labelled D-exp1 and F-exp1 used contiguous buffer space and a single physical read (the

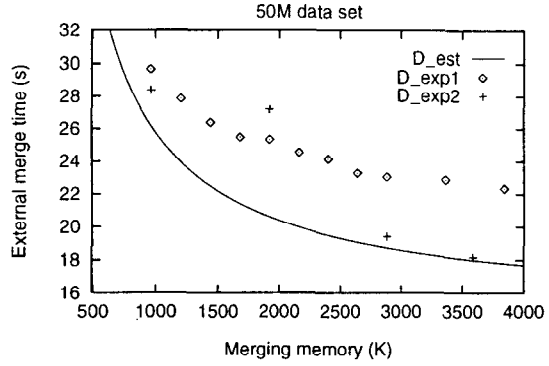

Figure 6: Merge time for double buffering.

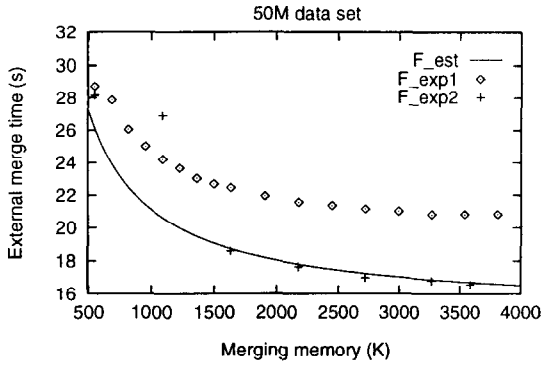

Figure 8: Merge time for extended forecasting.

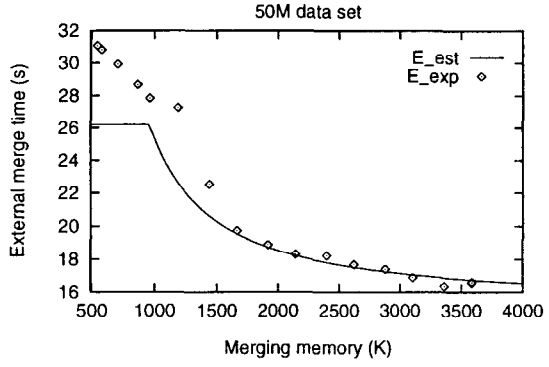

Figure 7: Merge time for equal buffering.

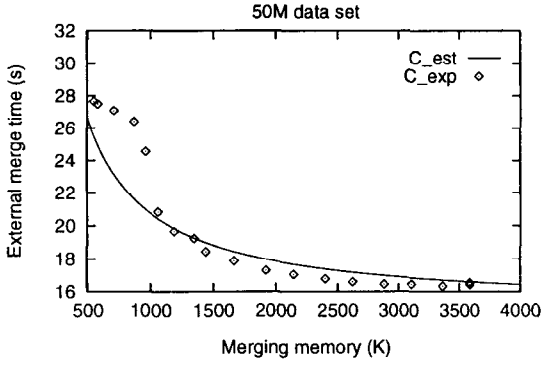

Figure 9: Merge time for clustering.

traditional implementation), while the two series labelled D-exp2 and F-exp2 used batches of 32KB reads. Using batches of small reads instead of a single large read improved performance but only for batches of three reads or more. Using two reads was actually slightly slower than a single read. We have no good explanation for this effect.

The agreement between predicted and observed merge time is excellent for all schemes when the amount of buffer memory is sufficiently large to allow read batches of size three or more. (This happens at different memory sizes for different schemes.) For single reads or batches of size two, the model underestimates the I/O time. The most logical explanation is that the disk controller starts prefetching after it has seen two sequential reads.

The estimated results of the four strategies were combined in one diagram (Figure 10) to facilitate comparison. The results confirm the observations in the previous section. Double buffering is the slowest. Clustering is the fastest, followed closely by extended forecasting. Equal buffering is in between. Equal buffering converges to clustering quickly as the available memory increases, but double buffering does not. The observed performance improvement of clustering over double buffering with large reads is substantial, around 30 % for all memory sizes.

5 Performance for Skewed Input

The analysis and experiments in the previous section assumed completely random input. In this section, we study the performance of the read-ahead strategies when runs exhibit *temporal skew.*

Records within a run are sorted so the ordering of the input for a run does not affect merge performance. However, merge performance will be affected if sort keys are distributed in such a way that the merge process concentrates on a subset of the runs for some time period and then shifts to another subset of runs for another time period. The number of active runs actually involved in merging is smaller than the total number of runs. Active here means contributing records to the output. In the extreme case, there might be only one active run at any given time. We call this phenomenon *temporal skew.*

Zheng and Larson [ZL96] introduced a simple model for runs with temporal skew. 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

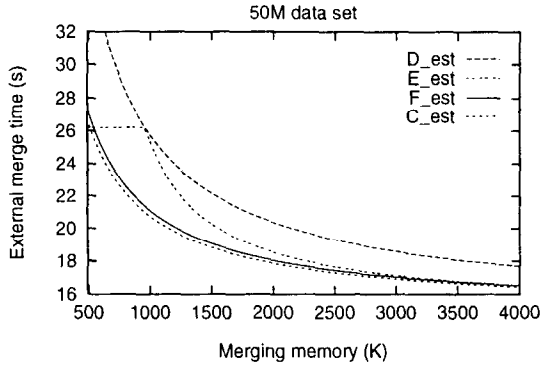

Figure 10: Comparing estimated merge times.

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 temporal skew. Setting $\alpha = 0$ is equivalent to the input data being sorted.

The key difference between random input and skewed input is in the number of active runs, i.e. the runs involved in merging. Let n_a denote the number of active runs. For the simple model of temporally skewed data described above (as shown in Figure 11), the number of active runs can be computed as follows:

- if $1/(1-\alpha) \geq n$, then $n_a = n$;
- if $1/(1-\alpha) < n$ and $\alpha \geq 0.5$, $n_a = 1/(1-\alpha)$;
- if $\alpha < 0.5$, $n_a = 2$, while for each run, a fraction of size $(1-2\alpha)$ of the run is not overlapped with any other runs.

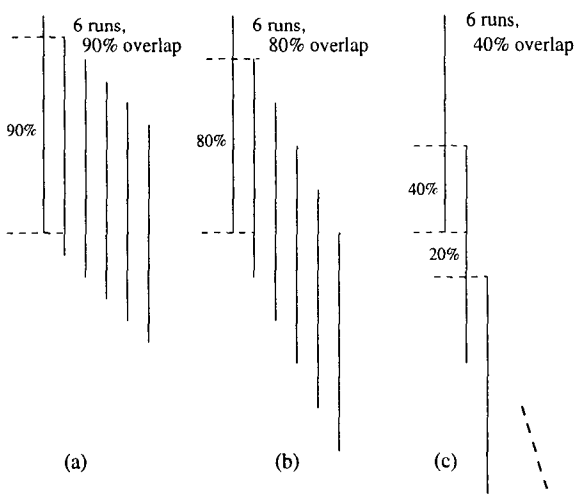

Figure 11: Example of temporal skew and active runs

Equal buffering

For equal buffering the cluster read size equals $R_E = p(b/n-1)$ when $b \geq 2n$, and $R_E = p$ when $n < b < 2n$. The number of clusters read is D/R_E. However, if $\alpha < 0.5$, a fraction $(1-2\alpha)$ of each run is not overlapped with any other runs. The data in this portion will be read sequentially so it is best treated as a single cluster (one seek only). The remaining 2α fraction of the data is read at the read size R_E. We approximate the number of reads for the $(1-2\alpha)$ portion by $(1-2\alpha)n$. When $\alpha = 0.5$, no extra reads are added, while when $\alpha = 0$, n reads are required, one for each run. Putting it all together we get the following estimate for the number of clusters read:

$$N'_E = \begin{cases} D/R_E & \text{if } \alpha \geq 0.5 \\ 2\alpha D/R_E + (1-2\alpha)n & \text{if } 0 \leq \alpha < 0.5 \end{cases} \quad (6)$$

where $R_E = p(b/n-1)$ when $b \geq 2n$, and $R_E = p$ when $n < b < 2n$.

Double buffering

Double buffering can be viewed as a special case of equal buffering. It divides the available buffer space equally among the runs and each read fills half of the buffer space assigned to a run. Provided every run has at least two buffer pages, that is, $b \geq 2n$, cluster reads will have an average size of $pb/2n$. Otherwise, a cluster read will cover exactly one buffer page. The cluster read size equals $R_D = pb/2n$ when $b \geq 2n$, and $R_D = 1$ when $n < b < 2n$. Therefore,

$$N'_D = \begin{cases} D/R_D & \text{if } \alpha \geq 0.5 \\ 2\alpha D/R_D + (1-2\alpha)n & \text{if } 0 \leq \alpha < 0.5 \end{cases} \quad (7)$$

where $R_D = pb/2n$ when $b \geq 2n$, and $R_D = p$ when $n < b < 2n$. (In our experiments, R_D was rounded down to a multiple of the page size.)

Extended forecasting

For extended forecasting, the cluster read size is $R_F = pb/(n+2)$. The rest of the argument is exactly the same as for double buffering. We get the following formula:

$$N'_F = \begin{cases} (n+2)D/(pb) & \text{if } \alpha \geq 0.5 \\ 2\alpha(n+2)D/(pb) + (1-2\alpha)n & \text{if } 0 \leq \alpha < 0.5 \end{cases} \quad (8)$$

Clustering

For clustering, we estimate the cluster size by the formula derived for random input but replacing the number of runs n by the number of active runs n_a, that is, $R_C = pb/(n_a+1)$. The function for n_a splits into three ranges:

- If $1/(1-\alpha) \geq n$, $n_a = n$;
- If $1/(1-\alpha) < n$ and $\alpha \geq 0.5$, $n_a = 1/(1-\alpha)$;
- If $\alpha < 0.5$, $n_a = 2$ for a fraction of size 2α and $n_a = 1$ for the rest (the non-overlapping part).

Combining this with formula 4, we get the following estimate for the number of clusters:

$$N'_C = \begin{cases} (n+1)D/(pb) & \text{if } 1/(1-\alpha) \geq n \\ (1/(1-\alpha)+1)D/(pb) & \text{if } 1/(1-\alpha) < n \ \&\ \alpha \geq 0.5 \\ 6\alpha D/(pb) + (1-2\alpha)n & \text{if } \alpha < 0.5 \end{cases} \quad (9)$$

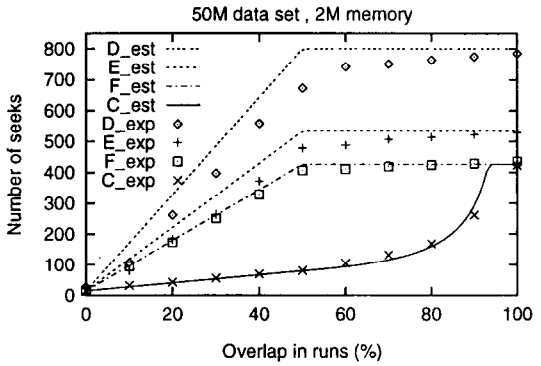

Figure 12: Estimated and observed number of seeks for skewed input when using 2MB of memory for buffers.

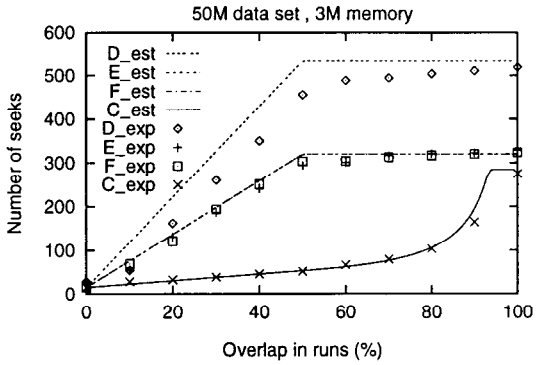

Figure 13: Estimated and observed number of seeks for skewed input when using 3MB of memory for buffers.

Numerical results

Our sort testbed is able to generate temporally skewed input based on the above model. We ran a series of experiments for each read strategy on 50MB input, varying the overlap of key ranges, i.e., the α value, from 0 to 1 (100%). 4MB of memory space was used for run formation, resulting in 15 runs. Buffer pages and physical reads were of size 32KB.

Figures 12 and 13 show the theoretically estimated and experimentally observed number of seeks (more precisely, number of cluster reads). The agreement is very close. Double buffering, equal buffering and extended forecasting do not exploit temporal skew unless the skew is so severe that there is only one active run some of the time. Clustering, on the other hand, adapts quickly to temporal skew and reduces the number of seeks.

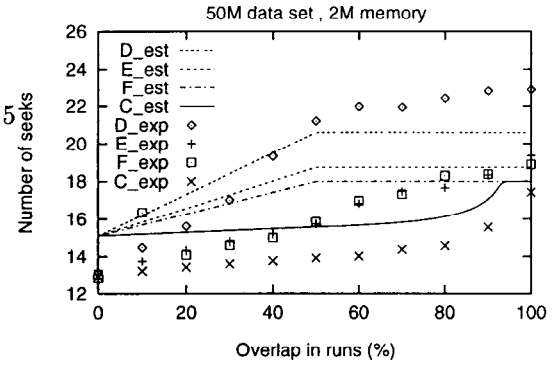

Figure 14: Estimated and observed merge time for skewed input when using 2MB of memory for buffers.

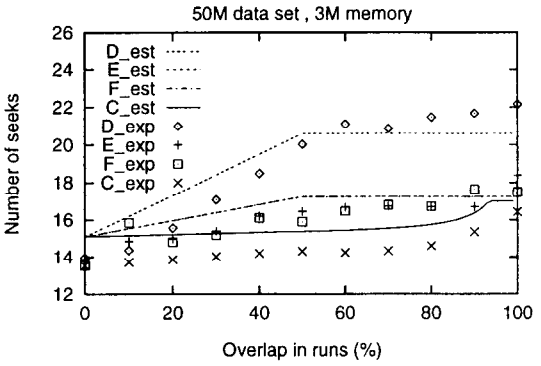

Figure 15: Estimated and observed merge time for skewed input when using 3MB of memory for buffers.

Figures 14 and 15 show estimated and observed merge times. Considering the difficulty of modeling modern disks, the agreement is reasonable: the general shape is the same but the observed merge times are lower than estimated. Part, but not all, of the discrepancy is explained by reduced seek time. When the number of active runs decreases, they cover a smaller part of the disk so seeks are shorter and faster. However, these results confirm the conclusion that clustering is able to exploit temporal skew in the input to reduce merge time. The other strategies benefit only from severe temporal skew.

6 Conclusion

This paper introduced three buffering and read-ahead strategies aimed at reducing disk seeks during the merge phase of external mergesort. They achieve bet-

ter performance than traditional double buffering or forecasting by exploiting the fact that modern disks do caching and sequential prefetch and by preplanning reads. Preplanning is based on retaining the last key of each (say 32KB) run block during run formation, from which the block consumption sequence can be computed.

Equal buffering is based on double buffering and does not do any preplanning. It requires about 50% fewer disk seeks then double buffering when the amount of buffer memory is large.

Extended forecasting achieves better overlap of I/O and merge processing than traditional forecasting. It uses two read-ahead buffers and requires access to the consumption sequence. Clustering reduces seeks by preplanning the read order (using a heuristic algorithm) based on the consumption sequence.

Based on theoretical modelling and experiment results, we found that

- clustering has the best performance; it reduced merge time by about 30% compared with standard double buffering;
- extended forecasting performs almost as well as clustering on random input but not on input with temporal skew;
- equal buffering does not require any preplanning and always performs better than double buffering, in particular when memory size increases.

One mystery remains unsolved: why changing from single large reads to batches of smaller reads reduced I/O time for double buffering and (extended) forecasting.

Our analysis and experiments made the simplifying assumption that runs are stored on a single disk. However, using multiple disks and striping is common in modern systems. We have not yet investigated how the proposed buffering and read-ahead strategies perform in such an environment.

References

[Cor97] Quantum Corporation. Storage basics. *Document at http://www.quantum.com /src/storage_basics/*, Oct. 1997.

[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.

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

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

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

[Sal89] Betty Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[Zha97] Weiye Zhang. *Improving the Performance of Concurrent Sorts in Database Systems.* PhD thesis, University of Waterloo, 1997.

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

[ZL96] Luo Quan Zheng and Per-Åke Larson. Speeding up external mergesort. *IEEE Trans. on Knowledge and Data Engineering*, 8(2):322–332, Apr. 1996.

[ZL97] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 376–385, 1997.