Memory Management during Run Generation in External Sorting

Per-Åke Larson
Microsoft
PALarson@microsoft.com

Goetz Graefe
Microsoft
GoetzG@microsoft.com

Abstract

If replacement selection is used in an external mergesort to generate initial runs, individual records are deleted and inserted in the sort operation's workspace. Variable-length records introduce the need for possibly complex memory management and extra copying of records. As a result, few systems employ replacement selection, even though it produces longer runs than commonly used algorithms. We experimentally compared several algorithms and variants for managing this workspace. We found that the simple best fit algorithm achieves memory utilization of 90% or better and run lengths over 1.8 times workspace size, with no extra copying of records and very little other overhead, for widely varying record sizes and for a wide range of memory sizes. Thus, replacement selection is a viable algorithm for commercial database systems, even for variable-length records.

Efficient memory management also enables an external sort algorithm that degrades gracefully when its input is only slightly larger than or a small multiple of the available memory size. This is not the case with the usual implementations of external sorting, which incur I/O for the entire input even if it is as little as one record larger than memory. Thus, in some cases, our techniques may reduce I/O volume by a factor 10 compared to traditional database sort algorithms. Moreover, the gradual rather than abrupt growth in I/O volume for increasing input sizes significantly eases design and implementation of intra-query memory management policies.

Keywords

Sorting, merge sort, run formation, memory management, variable length records, replacement selection, last-run optimization

1. Introduction

External mergesort is used in virtually all large-scale sorting applications, including database systems, with the optimization to sort small inputs entirely in memory. Runs are typically formed by a "load-sort-store" algorithm, i.e., an algorithm that repeatedly fills memory with input records, sorts them using some in-memory sort algorithm such as quicksort, and stores the sorted records as a run. If the sort input is larger than the available memory, even if only slightly larger, most implementations of external mergesort write the entire input to multiple run files on disk and merge those runs to produce the final, sorted output. This stark discontinuity in the sort algorithm's cost function creates significant problems in optimizing intra-query memory allocation and in query optimization, in particular cost estimation. In this paper, we describe implementation techniques that result in a much smoother cost function as well as in less I/O – in other words, in graceful degradation. Incidentally, one of the oft-claimed advantages of hash-based query evaluation algorithms using hybrid hashing and dynamic destaging [4][3] over sort-based algorithms is graceful degradation for inputs slightly larger than memory; our techniques overcome most of this disadvantage.

The essence of our technique is to retain in memory as much as possible of the previous (e.g., first) run while generating the next (e.g., second) run. If the input is only slightly larger than memory, all records in the second run and many of the records in the first run are never written to disk. Memory must be managed carefully while records from the first run are pushed out of the workspace and records for the second run are gathered. Note that records are expelled from and replaced in the in-memory workspace one by one, not "wholesale" as in traditional load-sort-store algorithms. While conceptually simple, this approach requires some form of memory management if input records vary in length, because replacing records may be larger than replaced records.

Variable-length fields and records are used extensively for almost all human-readable information except numeric values. Many database systems do not allocate record space for *NULL*-valued fields, thus creating variable-length records even if all fields are fixed-size numeric fields. Finally, non-traditional data types tend to be large and of variable length, although they also tend not to be part of sort opera-

tions. Padding all variable-length fields and records to their maximal length is a very inefficient and wasteful option. Any database algorithm that fails to accommodate variable-length records is of limited practical value.

We investigated two memory management algorithms: a standard *best fit* algorithm and a modified version of *next fit*. The best fit algorithm was found to perform very effectively with little space and CPU overhead. In particular, once a record has been inserted into the workspace, it is not moved until it is removed from memory and written to a run file. Our experiments demonstrate that best fit performs very well even for records whose lengths differed by as much as a factor ten. This, then, solved the problem of exploiting all memory to retain records in memory and made possible a sort that degrades gracefully when the input turns out to be only slightly larger than memory.

The same memory management problem is encountered if replacement selection is used to generate initial runs. In replacement selection, individual records are removed from the workspace and replaced by new records from the sort input, and records for two runs are present in the workspace at the same time. There are several advantages to using replacement selection, as outlined in a later section, but the problem of managing variable-length records has misled some researchers, including ourselves [2], to dismiss replacement selection in favor of load-sort-store algorithms. Given the excellent performance of the best-fit algorithm, we reconsider and resolve this argument in favor of replacement selection. Almost needless to say, graceful degradation and replacement selection can be combined, with cumulative benefits.

2. Related work

Sorting is one of the most extensively studied problems in computing. There is a vast literature on internal sorting, but less on external sorting. Knuth [6] provides extensive coverage of the fundamentals of sorting, including excellent descriptions of replacement selection and mergesort. Some sort algorithms are adaptive in the sense that they do less work if the input exhibits some degree of pre-sortedness. Mergesort with run formation by replacement selection has this property but mergesort with run formation by load-sort-store does not, i.e., the amount of work is always the same, even when the input is already sorted. Estivill-Castro and Wood [1] provide a survey of adaptive sort algorithms.

There is also a large body of research dedicated to dynamic memory allocation. Knuth [5] describes several algorithms, including first fit, next fit and best fit. A recent survey with an extensive bibliography can be found in [10]. While the basic ideas are directly applicable to the present study, there are two important differences between managing the workspace for a sort operation and managing the heap for a programming language such as Lisp or Java. On one hand, the size of the workspace is fixed and cannot be enlarged but, on the other hand, any number of records can be purged from the workspace at any time, albeit only in the order determined by their sort keys.

3. Preliminaries and assumptions

We first list some general assumptions and then describe specific assumptions about records, memory, data structures etc. that are most relevant to this study.

3.1 General assumptions

Unless otherwise indicated, the sort order is assumed to be ascending. In the experiments, the input is completely random with respect to the sort key.

We presume that the sort operation is supplied with input records from a stream of records, to model the typical situation in database query processing. This stream may originate from a simple scan or a complex query. A stream supplies its record one at a time and does not provide memory for more than the most recently supplied record. Thus, the sort operation must copy records into its workspace. Moreover, the number of records in the stream is not known with any useful accuracy until the end of the stream is reached.

In our brief discussion of merge patterns, we presume that individual run files can be scanned individually and in any order, as dictated by the merge optimization. It does not matter, for the purposes of this discussion, whether each run is a distinct database or operating system file, or just a distinct collection of pages or of groups of contiguous pages within a single database or operating system file. It also does not matter whether disk space is freed and recycled immediately after it has been scanned in a merge step, or only after one or all merge steps have been completed. While these are important practical concerns, they are orthogonal to the subject of this study.

While our analysis never considers explicitly more than a single thread or process, our techniques can readily be applied to parallel sorting, either by applying them within each thread or by sharing data structures among threads with suitable protection mechanisms added.

3.2 Assumptions about records and memory

We assume that record lengths may vary and that records cannot be divided into fragments within the sort operation. The memory management problem considered here is trivial if all records have the same length or can be divided into fragments. We also presume that there is no correlation between sort key and record length.

We also assume that the sort operation is limited to its workspace, i.e., it must not rely on any additional memory for large data structures or on additional I/O buffers provided by the database system or by the operating system. If

such memory or buffers were available, we believe they ought to be made part of the sort operation's workspace, because external sort algorithms have been invented specifically to minimize and direct their I/O operations more effectively than any general-purpose buffer replacement policy could. The only exception is a single output buffer for the run currently being written. Whether this one page is subtracted *a priori* from the available workspace or presumed to be provided outside the workspace does not affect the policies for managing variable-length records within the workspace. This one page could actually be a very small fixed set of pages to enable asynchronous I/O to one or more disk devices; this fixed amount of space is omitted from our analysis.

Most database systems' internal memory allocation mechanisms do not support allocation of arbitrarily large contiguous memory regions. Therefore, we presume that the sort operation's workspace is divided into multiple extents. Each extent is a contiguous area in memory. Extent sizes may vary from a single disk page, e.g., 8 KB, to very large, e.g., multiple megabytes. We assume that records cannot span extent boundaries.

An extent consists of a header and sequence of segments. The header stores the extent length and possibly some other information of small and fixed size, e.g., the anchor of a linked list of free segments. Segments are of two types, record segments or free segments. A record segment holds exactly one record. Each segment, whether it holds a record or is currently free, starts with a one-bit field indicating the segment type and a field indicating its length. The segment type is needed when a valid record is deleted, and its neighboring segment is inspected to determine whether it is a free segment and can be merge with the newly freed segment. Free segments also contain some number of pointers needed for free space management; exactly how many depends on the algorithms used, so this issue will be discussed later. By using this method, information required to manage free segments is captured and maintained within the workspace, not in data structures allocated outside of and in addition to the workspace.

Free segments never abut; there always is at least one record segment between two free segments. If a record segment is freed that abuts a free segment, the free segment is extended to include the newly freed space. If a record separating two free segments is deleted, all three segments are merged into a single free segment. A newly initialized extent contains only one free segment, covering the entire extent.

Note that a free segment must be large enough to store the type field, the segment length and all its required pointers. When inserting a record into a free segment, the leftover piece may be too small to accommodate the fields required in a free segment. If so, the complete space occupied by the free segment is assigned to the record. The net effect is that record segments may be slightly larger than minimally required. To reduce the space wasted in this way, it is important to keep the minimum size of free segments as low as possible.

Records are inserted into and deleted from the workspace as run formation progresses. At any point, the workspace contains records from at most two runs: the run currently being output and the next one. To insert a record, a sufficiently large free segment must be found. This can either be an existing free segment, one created by relocating other records, or one created by deleting records from the workspace and outputting them to the current run. (This last point is what makes our memory management problem unique: if necessary, we can always "manufacture" free space by outputting records.) When a record is output, its segment becomes free. A newly freed segment is merged with adjacent free segments whenever possible.

The actual sort operation operates on pointers to the records in the workspace. Thus, once a record has been assigned to a location in the workspace, it can remain there until the record is written out to a run file. The sort algorithms *per se* do not require moving records; only the free space management algorithm may require record movement.

4. Run formation algorithms

4.1 Load-sort-store

Most database systems implement some variant of the following run formation algorithm. Fill the workspace with records, extract into an array pointers to all records in memory, sort the entries in the array (on the sort key of the records pointed to), scan forward through the array outputting records into a run file, and erase all records in memory. Repeat this until all input records have been processed. Any in-memory sorting algorithm can be used for the pointer sort in step two, with quicksort being a popular choice. All runs created will be of the same length, except possibly the last run and the effect of wasted space due to variable-length records in fixed-length pages. If the complete input fits in memory, the records are not output as a run but passed directly to the consuming operator. For obvious reasons, we call algorithms of this type load-sort-store run formation algorithms.

4.2 Last run optimization

Even when the input is only slightly larger than the workspace, a typical load-sort-store algorithm writes the entire input to disk. Thus, the sort algorithm's cost function has a stark discontinuity at the point when input sizes grow beyond the memory size. We can overcome this problem by never outputting more records than necessary to accommodate incoming records. When a record arrives, we have to find a free entry in the pointer array and also a sufficiently

large free segment in the workspace. If there are no free entries in the pointer array, one can be created by outputting the next record in the current run. How to find a suitable free segment is the focus of this study and is discussed in detail in the section on memory management.

At any time during run generation, in particular at the time when the end of the input stream is encountered, the tail of the pointer array will contain pointers to records in the current run while the front will contain pointers to records in the next run. Three pointers into this array are needed: one pointing to the next available slot in the array, one pointing to the first remaining slot in the current run, and one pointing to the last slot in the current run. When the current run ends, we run an in-memory sort and start a new run. When the input ends, memory will be completely filled with records, except for fragmentation between variable-length records. Some of the records represent the tail of the run that is currently being written to disk while the remaining records belong to the final run. Combining these records can be made part of the first merge step in the external sort, without ever writing these records to any run file on disk.

In the best case, this "last run" optimization saves writing and reading an entire memory load of records. However, depending on the desired fan-in of the first merge step, some memory might have to be used as input buffers. Thus, if the fan-in in the first merge step is close to or even equal to the maximal fan-in, the beneficial effect of the last run optimization should be minimal. A later section details our experimental observations.

4.3 Replacement selection

The previous algorithm always adds an incoming record to the next run, never to the run currently being output. Replacement selection is based on the observation that if we keep track of the highest key output so far, we can easily decide whether an incoming record can still be made part of the current run or has to be deferred to the next run. Adding a record to the current run is expensive if we use a pointer array. Instead, replacement selection uses a priority heap, also called a selection tree, and adds the run number as the first field of the sort key. For details, in particular a very efficient organization of the heap as a "tree of losers," see [6]. The end result is a simple and very efficient algorithm for run formation that produces runs that are, on average, twice the size of employed memory. This algorithm was invented in the early 1950s and has remained virtually unchanged since 1958.

When the input ends, replacement selection will be in almost the same state as the previous algorithm, i.e., memory will be completely filled with records belonging to two different runs. However, in replacement selection, it is guaranteed that the key ranges of the two runs in memory do not overlap. Only keys higher than the most recently written record are in the current run, and only keys lower than that record have been deferred to the next run. Thus, these two partial runs can be concatenated without any additional effort in the first merge step.

4.4 Benefits of replacement selection

It is well known that replacement selection produces runs twice as large as the memory employed, which reduces the number of runs and the required merge effort. For very large inputs, since the average merge depth is approximately $log_F W$ for merge fan-in F and W initial runs, having ½ as many runs reduces the merge depth by $log_F 2 = 1/log_2 F$ merge levels, e.g., ¼ merge level for fan-in $F = 16$. More about this in a later section. However, replacement selection has several other, equally important, advantages over load-sort-store.

First, replacement selection exhibits a fairly steady I/O behavior during run generation rather than I/O in great bursts. This improves the utilization of I/O devices. Secondly, it very effectively exploits pre-sorting, i.e., input sequences that are not random but somewhat correlated to the desired sorted output sequence. In particular, if the input is already sorted, a single run will be generated [1]. Sorting is often done as part of grouping with aggregation or for duplicate removal. A technique called early aggregation or early duplicate removal can then be applied, which reduces I/O significantly (see [7] and its references). Early aggregation and early duplicate removal achieve much higher I/O reduction if runs are created by replacement selection, which is the third advantage.

Contrary to an occasionally encountered belief, external sorting using replacement selection does not require more key comparisons than using quicksort, presuming that run generation employs a selection tree and that this selection tree is organized as a "tree of losers" [6]. A tree of losers is a binary tree that is traversed only from leaf to root, never from root to leaf, with one key comparison at each level. The tree is balanced, except that the lowest level might not be complete, in order to permit node counts that are not a power of 2. A traversal from leaf to root thus requires N $log_2 K$ comparisons for N input records passing through a tree of K entries. In fact, if a tree of losers is used both for run generation and for merging, and if the merge pattern is optimized using a simple heuristic, the entire sort requires N $log_2 N$ key comparisons[1]. Similarly, replacement selection requires the same amount of in-memory copying if, as is typical in database query processing, the sort operation has

[1] These formulas are accurate modulo some minor rounding if K is not a power of 2, and if runs of very different lengths must be merged. However, they do not require a substantial correction such as a factor of 2 as would be required by the "text book" implementation of priority heaps that employs root-to-leaf traversals.

a private workspace into which no other operation, e.g., a prior join, can directly deposit records, and if the memory management techniques discussed in this study are used. Thus, comparisons and in-memory copying are very similar for run generation using load-sort-store algorithms and replacement selection.

5. Merge patterns

We assume that a sort continues to use the same workspace during the merge phase that it used during run formation. The space will now be occupied mainly by input buffers. At least one input buffer per run is required but additional buffers may improve I/O performance [9][8][11]. The buffer size, required number of buffers per run, and workspace size determine the maximum fan-in that can be used during merging. We ignore here many such issues, not because they are not relevant in practice, but because those issues are orthogonal to the subject at hand and relevant techniques can readily be combined with the techniques described here. Additional issues include the management of a very large directory of runs, virtual concatenation of runs based on disjoint key ranges, asynchronous I/O operations based on double buffering or forecasting, and the tradeoffs between buffer size and fan-in.

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 run. So given N initial runs, possibly of variable length, and a maximum merge fan-in F, which merge pattern results in the minimum I/O volume? This has a surprisingly simple solution (see [6], pp. 365-366): first add enough dummy runs of length zero to make the number of runs minus one divisible by $F-1$ and then repeatedly merge together the F shortest existing runs until only one run remains. Note that some records will participate in more merge steps than others; thus, it makes sense to define the merge depth as the average number of merge steps each record participates in. If the number of runs is a power of the fan-in, this number will be equal to the integer usually computed as merge depth; however, this number meaningfully can and often will be a fraction.

We make one exception to this merge pattern: the last one or two runs are always included in the first merge step. Here are the reasons for this exception. When reaching the end of the input, the final (short) run and the tail of the previous run are in memory. Normally, the first merge step will not be a full K-way merge because of dummy runs added. We continue outputting records just long enough to free up exactly the space needed by buffers for the real runs included in the first merge step. (If the final run remains entirely in main memory, it does not require an input buffer.) In the worst case, this may force out all the records in memory. Note that the free space has to be consolidated into contiguous buffer-size areas, which will require copying of records.

The input to the first merge step will then consist of (a) some number of (real) runs stored entirely on disk, (b) one run stored partially on disk (front on disk, tail in memory), and (c) one run stored entirely in memory. This requires that input for the first merge step be handled differently than input for subsequent merge steps.

Slightly more records can be retained in memory if initial runs are output in reverse order, i.e., for an ascending sort, records are output in descending order during initial run formation, and vice versa for a descending sort. If runs are output in reverse order, allocation of input buffers needed by the run partially in memory can be delayed until all records of that run currently stored in memory have been consumed during the first merge step. Runs output in reverse order have to be read backwards during merging. This optimization was not used in our experiments.

6. Memory management

As run formation proceeds, records will be inserted into and deleted from the workspace. The process is driven by insertions, i.e., a record is deleted only when space is needed for additional records. To insert a record, we must first find (or make) a free block large enough to hold the record. If no suitable free segment is found, we output the next record and delete it from the workspace. This continues until the new record can be stored. Deletion of a record creates a new free block that must be added to whatever data structure is used for managing free segments. If the new free segment is adjacent to another free segment, the two segments need to be merged.

6.1 First fit and next fit

The simplest search method is a linear scan: check free segments one by one (in some order) and stop when a sufficiently large free segment is found or all free segments have been checked. The search can start either at a fixed position or at the position where the last insertion or deletion took place. These algorithm variants are called *first fit* and *next fit*, respectively. Note that *next fit* requires a pointer to the first free segment to be considered in the next insertion, and that this field must be maintained when segments are merged. The normal implementation is to place all free segments (within an extent) on a doubly-linked list. The list is maintained in address order to simplify merging of adjacent free segments.

We considered a modified version of next fit, here called *next fit with single record move*. We observed that search lengths for standard next fit were very high so we decided to limit the number of free segments inspected during a search. If no suitable free segment is found before reaching the limit, our algorithm then considers moving one record. The idea is to enlarge one of the free segments seen during the search by moving the record immediately after it to one of the other free segments seen. If moving one record can-

not create a sufficiently large free segment, the algorithm resorts to purging one or more records (in which case no records are moved). Records are output until a deletion creates a free segment large enough for the new record.

For example, let the record to be inserted be 120 bytes long. If the first free segment inspected is only 100 bytes long, it cannot hold the new record. If the second free segment is only 60 bytes long, it, too, cannot hold the new record. However, if the record immediate after the 60-byte free segment is 80 bytes long, that record can be moved to the first free segment, with 20 bytes left over. Once that record has been moved and its old space of 80 bytes combined with the old free segment of 60 bytes, the new record can be placed, also with 20 bytes left over.

Record segment	Free segment
Type	Type
Length	Length
Pointer to reference on heap	(Forward pointer)
Actual record	Backward pointer

Table 1: Segment structure for next fit

The fields contained in a segment are listed in Table 1. All segments contain a one-bit type field and a length field. In addition, free segments contain a forward pointer and a backward pointer. The forward pointer is not absolutely necessary. Instead of scanning forward looking for a free segment, we can scan backward using the backward pointer. When a record is deleted, the space it occupied is converted into a free segment that must be inserted into the linked list at the appropriate place and, if possible, merged with adjacent free segments. Because every segment contains a type field and a length field, we can locate the first free segment *after* the deleted record by skipping forward, one record segment at a time. The first free segment *before* the deleted record can then be located by following the backward pointer. Once the two free segments on either side have been located, inserting the new free segment into the linked list and deciding whether it can be merged with one or both of them is straightforward. Our implementation used two pointers even though the forward pointer is not absolutely necessary.

Record segments require one pointer, in addition to the type and length fields, because records may move while in memory. The actual run formation algorithm deals with references (pointers) to records, stores the references in some data structure, and shuffles the references. Whenever a record is moved, we must update the corresponding reference, which means that we need to know where that reference is. That's the purpose of the pointer in the record segment, i.e. it keeps track of the location of the reference to the record. As such, it must be updated whenever the reference is moved in memory, which increases the run time.

There are several possible variants of this algorithm that we did not investigate in detail. Our variant of the algorithm does not consider moving a record until it has inspected as many free segments as permitted by the limit, and it does not consider records or free segments beyond those inspected during this initial search. If there are multiple records that could be moved, or multiple free segments to which such a record could be moved, the algorithm chooses the first ones found. Finally, more than one record could be moved in a triangular fashion.

6.2 Best fit

The idea of the best-fit algorithm is to insert a new record into the smallest free segment large enough to hold it. Our implementation of the algorithm relies on a binary tree, using segment size as the key. We make no attempt to keep the tree balanced.

To locate a free segment with the desired property, we first descend the tree looking for an exact match. If a free segment of exactly the right size is found, the search ends. If none is found, we look for the immediate successor of the node where the previous search ended. (Algorithm: remember the last node with a right pointer where we went left; from that node, go right one step and then descend left pointers as far as possible.) If this fails as well, none of the free segments is large enough to store the record. We then output records until a sufficiently large free segment is created.

When deleting a record and creating a new free segment, we still need to be able to locate the closest free segments on either side (to check whether free segments can be merged). For this reason we retain the backward pointer. The forward pointer is not needed.

The binary tree used for searching needs a left and a right pointer. A third pointer, pointing to the parent of a node, is useful but not indispensable. When deleting a free segment, we need to access the parent node. Using the third pointer makes it trivial to locate the parent but we can always find it by searching from the root.

As illustrated in Table 2, a free segment then contains a one-bit type field, a length field, and three pointers (backward, left, right). Two additional pointers, a forward pointer and a parent pointer, are useful but not indispensable. Our implementation includes the forward pointer but not the parent pointer. A record segment contains just the type field and the length field.

Record segment	Free segment
Type	Type
Length	Length
Actual record	(Forward pointer)
	Backward pointer
	(Parent pointer)
	Left child pointer
	Right child pointer

Table 2: Segment structure for best fit

Again, there are several possible variants to this algorithm that we did not investigate. For example, the best-fit search could be combined with moving one or more records from their current location to some free segment. It will become clear in the next section why we did not investigate any "fancy" variants to the basic best-fit algorithm.

7. Experimental results

The results reported in this section are divided into three parts. First, we analyze memory utilization and insertion effort for best fit and next fit with single record move. Second, we analyze the run length achieved by replacement selection when memory is managed by best fit. Third, we analyze what effect this has on the total I/O volume during a sort and, in particular, on the I/O volume for small inputs (up to a few times memory size).

Input data for the experiments was generated randomly as follows. Record keys were drawn from a uniform distribution with values in the range [0, 32768]. Note that the actual key values have no effect on the results. Record lengths were drawn from a triangular distribution with a specified minimum and maximum record length. The probability that a record is of length k is $p(k) = (1- 2(k-l)n)/(n+1)$, $k = l, l+1, \ldots, u$, where l is the minimum record length, u is the maximum record length and $n = u - l$. The average record length is then $l+(u-l)/3$. In the experiments reported here, the average record length is always 200 bytes but we vary the minimum and maximum lengths to investigate the effects of record length variance. Intuitively, one would expect almost any space management scheme to leave more unused space when the record length is more variable.

In all of the following experiments, the indicated memory size and memory utilization compare the size of all data structures, including extent headers, free space lists and trees, segment type and length indicators, etc. with the size of the records only. In other words, all space employed for or wasted due to variable-length records is accurately accounted for.

7.1 Memory utilization and insertion effort

Our initial experiments focused on memory utilization and insertion effort and how they depend on the extent size. The data represent averages computed over 100,000 record insertions into a single extent of the specified size. As we are mainly interested in steady state performance, measurements were only taken after a warm-up period consisting of 10,000 insertions

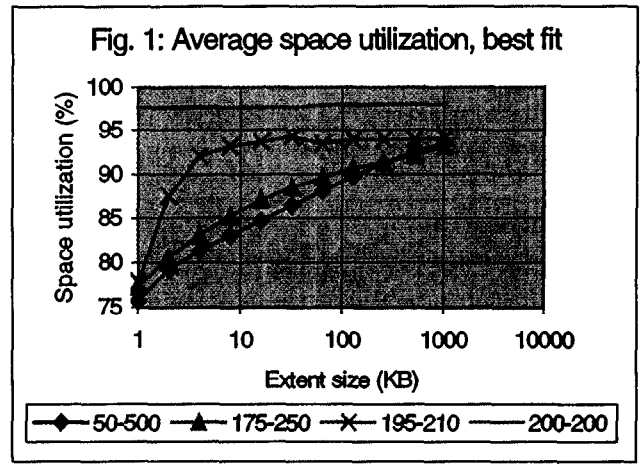


Figure 1 plots the average space utilization observed when managing memory by best fit, that is the percentage of memory occupied by actual records. Each line in this and subsequent graphs represents a range of record lengths (in bytes), and is labeled with minimum and maximum record length. Note the logarithmic scale on the X-axis.

Figure 1 immediately lends itself to several observations. First, if records happen to be of uniform length, the best-fit algorithm keeps memory completely filled but space utilization is slightly less than 100%, because of the type and length fields added to each record. Second, for variable-length records, the larger the extent, the higher the space utilization. This agrees with intuition: the more records and gaps there are in an extent, the better can the *best fit* heuristic insert new records into existing free segments. Third, for record collections of almost uniform length (195-210), little space is lost to fragmentation; 6-7% is hardly worth worrying about. Fourth, even when extents are very small, e.g., 5 times the average record size, or 1 KB, loss is moderate, about 25%. If only 75% of memory are utilized, replacement selection will result in run lengths of about 1.4 times the size of memory, which is still more than what can be obtained by load-sort-store but less than the desirable factor of 2. For extent sizes more typical as a sort operation's workspace, e.g., 256 KB or 1,280 average record lengths, over 90% of the space is utilized. Fifth, there is no significant difference between record collections with moderately and drastically varying record lengths. Moreover, this observation is quite uniform over all extent sizes. Our main conclusion from Figure 1 is that, except in unrealistic cir-

cumstances, *best fit* succeeds in reclaiming free space remarkably well.

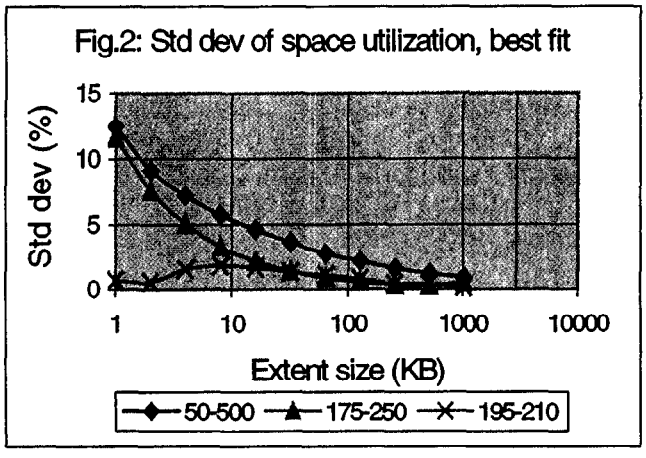


Figure 2 shows the estimated standard deviation of the space utilization. After each insertion, the space utilization (in percent) was observed, giving a series of 100,000 data points. The standard deviation was then computed for these 100,000 observations, giving one of the data points plotted in Figure 2. The standard deviation decreases as the extent size (and number of records stored) increases, confirming the intuitive expectation that the process is more stable for larger extents.

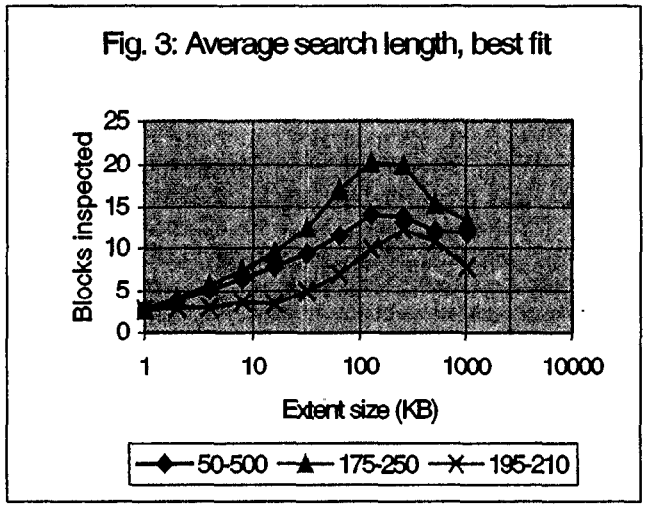
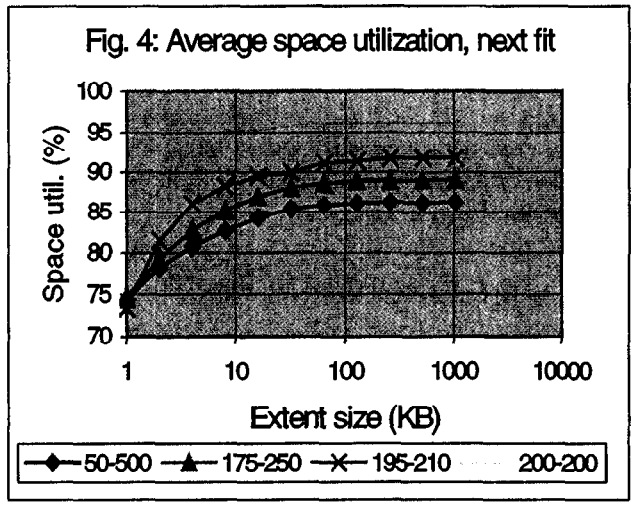


Figure 3 illustrates the search effort when inserting records and looking for a free segment. *Best fit* relies on a binary tree of free segments and the search effort roughly corresponds to the depth of the tree, which increases logarithmically with the number of free segments. Intuitively one would expect the search length to increase steadily with the extent size, but it doesn't. Note that a suitable free segment may be found in the tree without searching all the way to a leaf node. For example, searches that find an exact match finish higher up in the tree. Regardless of the variance in record length, the search effort reaches a maximum somewhere in the range 128 to 256 KB. The main reason for the decrease is that the number of free blocks increases as the extent size increases and a larger fraction of the requests find a free block of exactly the right size.

While we did not measure it specifically, the search depth gives an excellent indication for the CPU time required for the search. Since the binary tree is stored in the free segments, i.e., all over the sort operation's workspace, each node that must be searched corresponds to one cache fault. The CPU effort within each node is just a few instructions, because all that needs to be done are one or two comparisons of integer values indicating record lengths. Maintaining the binary tree is also very inexpensive, given that we did not bother to keep the tree balanced.

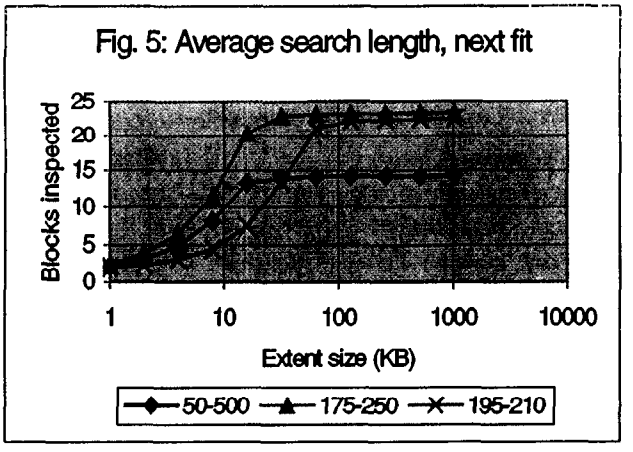
We ran similar experiments using a standard version of next fit, i.e., next fit without record move. The results were not encouraging; search lengths were high and space utilization rather low. In response, we designed the modified version of next fit explained in a previous section, i.e. next fit with single record move.



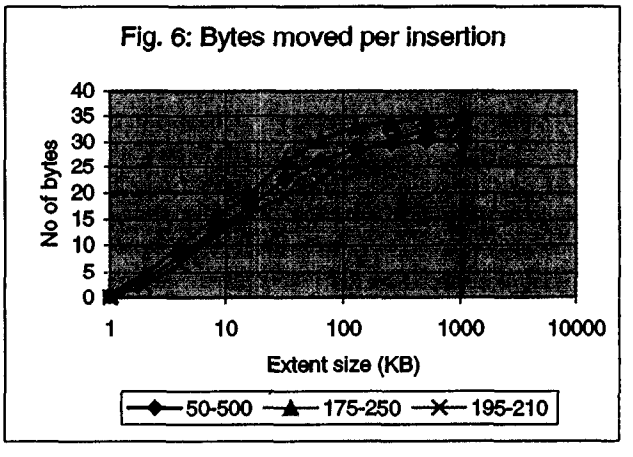
Figures 4 to 6 summarize the performance of next fit with single record move. In these experiments, the search limit was set to 25. The average space utilization (see Figure 4) increases smoothly as the extent size increases. However, it is always lower than the space utilization for best fit. The standard deviation (not shown) was very similar to the standard deviation for best fit shown in Figure 2.

The average search length is plotted in Figure 5. Note that the search was limited to a most 25 blocks. Again, the average search lengths are very reasonable but higher than those for best fit are.

The algorithm occasionally moves a record to enlarge a free block. Figure 6 shows the average number of bytes moved per record inserted, not counting the cost of actually



copying a new record into the workspace. When the extent size is 1 MB, slightly over 30 bytes (about 15% of the average record length) are moved. This is surprisingly low. Upon analysis, we found that about ¼ of the insertions cause a record to be moved. The record moved is always shorter than the record being inserted; therefore, the records moved tend to be shorter than the average record length, which explains the discrepancy. Note that each record has to be copied into and out of the workspace, thus incurring on average 400 bytes of copying cost. Adding 30% on average per record adds less than 10%, and was therefore worth exploring and comparing with alternative algorithms. However, even with the single record move enhancement, next fit does not compete well with best fit in overhead and effectiveness.



Based on these experiments, it is evident that best fit is the preferred algorithm for space management in a sort operation's workspace. However, the next-fit algorithm uses a simpler data structure (a linked list). This may be a consideration in applications or circumstances other than those typically found in database sort algorithms, for example, when managing space shared by multiple threads running in parallel.

7.2 Run sizes

In this section, we report on experiments measuring the size of initial runs generated by replacement selection when space is managed with *best fit*. The figures show the *relative run length*, which is the run length divided by the size of the workspace. Replacement selection produces runs that are, on average, twice the size of memory [6], as is well known for fixed-sized records and random input. When records are of variable length, some memory is wasted so the average run length will certainly be less than two times the size of memory.

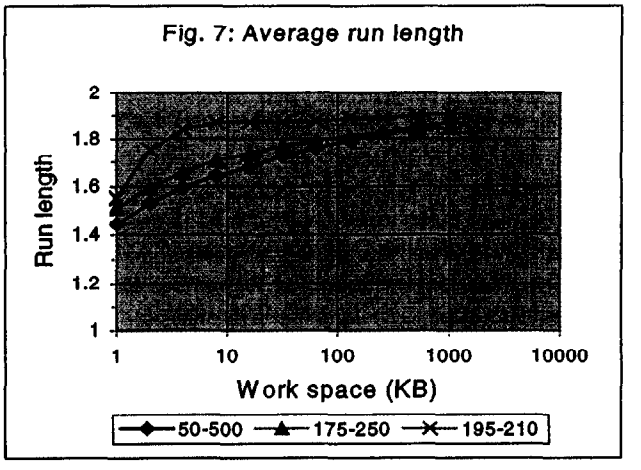
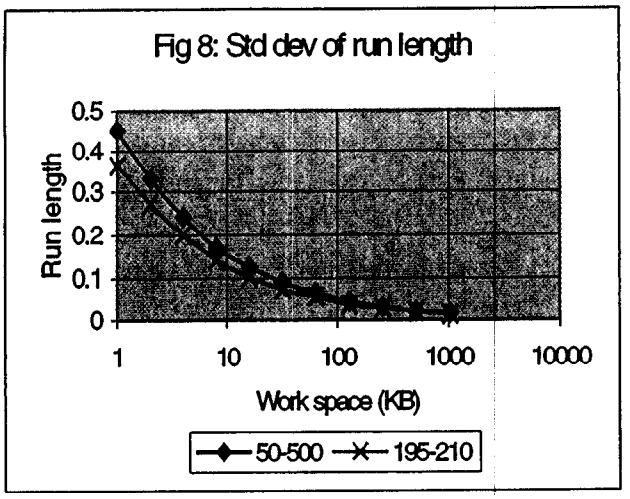


Figure 7 plots the average run length for three different record length distributions and different workspace sizes. As expected, the relative run length increases as the workspace size increases and as the variance in record length decreases. Both have the effect of increasing space utilization, which immediately translates into longer runs. Overall, the relative run lengths are quite remarkable: when the workspace size is 128 KB or higher, we can achieve run lengths over 1.8 time the size of the workspace, that is, within 10% of the average for fixed-length records.

The standard deviation of the relative run length is plotted in Figure 8. It decreases very rapidly as the workspace increases, meaning that even for moderately sized workspaces run lengths are very stable. Interestingly, it is affected very little by variance in record length.

In our experiments the sort operation's workspace consisted of a single extent of the indicated size, i.e., the workspace was not divided into smaller extents. Dividing the workspace into smaller extents has minimal effect, as long as the extents are not too small. The only change is that records and free segments cannot cross extent boundaries, resulting in a slightly lower memory utilization, as was discussed in the previous section. Another way to think of it is to divide a single large extent into smaller units by inserting (header-size) dummy records at fixed intervals. The dummy records are never moved or deleted and therefore prevent two free



segments on either side from being merged. As long as there are only a few dummy records, their effect on memory utilization (and run lengths) will be negligible.

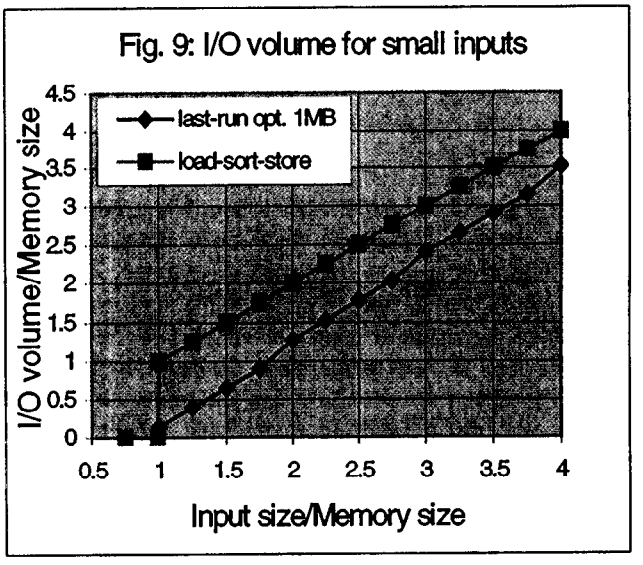
Last run optimization

In addition to record-by-record replacement of variable-length records in a sort operation's workspace during run generation using replacement selection, the memory management techniques explored in this study can also be used to enable the last run optimization. In this section, we report on experiments comparing load-sort-store algorithms with and without the last run optimization.

For simplicity we use the term *I/O volume* for the total amount of intermediate data written to disk during sorting. The initial runs are included but neither the original input nor the final output are; we assume that the original input and the final output are passed directly from and to neighboring operators in the query plan and do not have to be read from or written to disk. Given that all algorithms require almost identical amounts of record comparisons and in-memory record movement, the amount of I/O is an appropriate indicator of overall sort performance that is orthogonal to specific implementations of I/O operations and their performance.

We assume that the sort continues to use the same memory during the merge phase as it did during run formation. The simplest way to reduce disk overhead (seek time and latency) during merging is to use large I/O buffers and issue large read/write operations. Our results are based on a buffer size of 64 KB with two input buffers per run. The buffer size, the number of buffers per run, and total amount of memory determine the maximum merge fan-in: with double buffering, 1 MB of memory is enough for a 8-way merge and ½ MB for a 4-way merge. The effects of forecasting and selective read-ahead are not considered here.

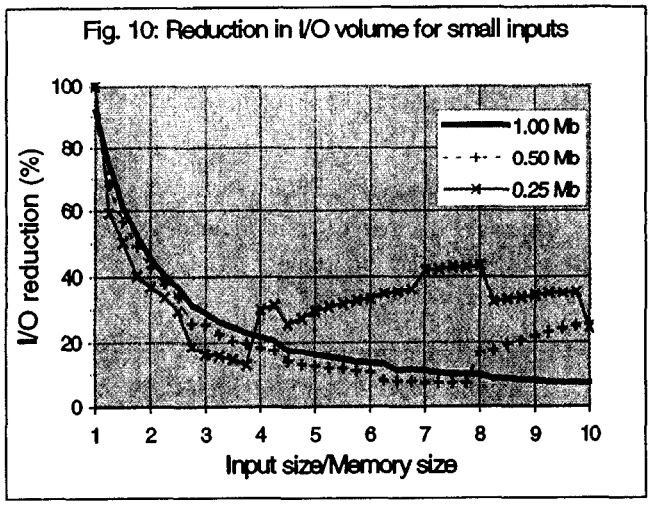
Figure 9 shows the I/O volume produced by load-sort-store run generation with and without the last-run optimization. It clearly demonstrates how the last run optimization completely avoids the sudden increase when the input size slightly exceeds the memory size. Even for input sizes a few times larger than memory, the reduction in I/O volume is still substantial. This difference slowly decreases, and completely vanishes when all memory is required even for the first merge step, i.e., when the input size approaches the memory size times the merge fan-in, a quantity sometimes referred to as the square of memory size. In the example shown in Figure 9, this occurs at 8 MB.



7.4 Overall sort performance

So far we have seen that both replacement selection and the last run optimization can be made to work even for variable-length records. But how does this affect overall sort performance? The experiments reported in this section provide a partial answer by comparing the amount of intermediate data (run data) that has to be written to disk using two different run formation algorithms: replacement selection with last run optimization and load-sort-store. Note that we did not apply last-run optimization to load-sort-store run formation, i.e., all runs were completely written to disk before merging began, because this is the way most systems handle run formation today.

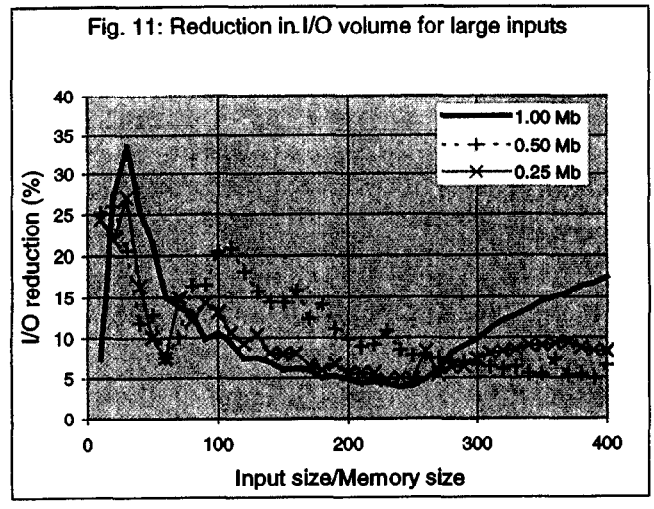
Figures 10 and 11 show the reduction in I/O volume achieved by using replacement selection (with best fit and last-run optimization) compared with the I/O volume for the standard implementation using load-sort-store run formation. In these experiments, the record length varied between 100 and 400 bytes. Figure 10 shows the reduction for small inputs, i.e., inputs less then 10 times the memory size. When the input is only slightly larger than memory, the last-run optimization will retain almost all of the input data in memory, thereby reducing the I/O volume by almost 100%, i.e., almost to nothing. Note that a decrease by 90% is a



reduction by as much as an order of magnitude. As the input size increases, the benefits of the last-run optimization gradually diminish. However, the reduction is still about 20% for inputs as large as four times the memory size. Notice the jumps at four and eight times memory size. They occur where the standard implementation shifts from a single merge step to two merge steps. Because replacement selection produces fewer runs, it can still complete the sort with a single merge, shifting to multiple merge steps at about twice the input size.

For large inputs, shown in Figure 11, the reduction is the result of fewer runs and therefore fewer merge steps; the effect of the last-run optimization is minimal. The highest reduction is observed when the standard sort adds one more merge step. For example, with 1 MB of memory (16-way merge), up to 16+15=31 runs can be merged in two steps. At 32 runs, three merge steps are required: first a two-way merge (to reduce the number of runs to 31), followed by two 16-way merge steps. Runs 31 and 32 are now written twice to disk, once during run formation and once after the first merge step. However, because replacement selection produces fewer runs, inputs of size 32 times memory can still be sorted using only two merge steps.

The steady increase in the line for 1 MB sort memory after about 250 MB of input is also caused by the larger number of runs resulting from load-sort-store run formation. With a fan-in of 16, at most 256 runs can be merged in two complete passes over the data. The first complete pass will consist of 16 merge steps producing 16 runs and the final merge step will merge these 16 runs. In other words, each record will be written to disk exactly twice. As soon as the number of runs exceeds 256, some records will be written more than twice to disk. Load-sort-store produces runs of the same size as memory so as the input size increases beyond 256, more and more records will be written three times to disk. Replacement selection produces longer runs, about 1.85 times memory size, and will not exceed 256 runs until the input size reaches approximately 1.85x256 = 473.6. This explains the increasing reduction with input sizes in the range 256 to 474 times the size of memory.



8. Summary and conclusions

In this study, we have investigated memory management techniques for variable-length records during run formation for external merge sort. This led to a non-intuitive (at least for us!) result, which in turn enabled to two important improvements of external sort algorithms.

Our most important finding is that even if record lengths vary widely, fairly high memory utilization can be achieved with very little space and processing overhead. All required data structures can be "hidden" in the unused portions of the sort operation's workspace, and search effort per record insertion is a few hundred instructions at the most, using very limited search and no compaction.

Based on this result, we have reversed our former belief that variable-length records are a strong and convincing argument against replacement selection for run generation in external mergesort. Managing variable-length records is not a big problem. The simple best-fit algorithm solves this problem very well.

The longer runs produced by replacement selection reduce the total I/O volume by reducing the number of merge steps required. This fact alone is sufficient reason to favor replacement selection over load-sort-store run formation but replacement selection offers several additional advantages. First, the I/O activity during run formation is spread very evenly, thus making the most effective use of I/O devices. Second, it adapts naturally to pre-sortedness in the input by producing longer runs than expected for random input. In particular, it will produce only one run if the input is already sorted. Third, when combined with early aggregation for evaluation of group-by queries and duplicate elimination, it achieves close to maximal reduction in I/O volume.

The same memory management technique can be employed to build sort algorithms that spill incrementally as needed to accommodate further input records, resulting in a smooth

cost function very similar to that of hybrid hashing with dynamic destaging. Differently than most implemented sort algorithms, if our sort algorithm's input is only slightly larger than the available memory, only a small part of the input is ever written to disk. Thus, our algorithm significantly improves sort performance for input sizes between memory size and a few times memory size, in addition to simplifying cost calculation and memory management during query processing, which are additional very important aspects of this technique. Finally, graceful degradation and doubled run sizes can easily be combined.

Memory management problems very similar to the ones studied here also arise in areas unrelated to sorting. Within the field of database management, two areas come to mind: free space management within disk pages and caching of variable-length records or objects. Our solution is directly applicable to these cases.

We believe that the techniques presented here will relatively quickly (as these things go!) find their way into commercial database systems. In fact, they are currently being prototyped for sorting variable-length records in Microsoft SQL Server.

9. References

[1] Vladimir Estivill-Castro, Derick Wood: A Survey of Adaptive Sorting Algorithms. Computing Surveys 24(4): 441-476 (1992)

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

[3] Goetz Graefe, Sort-Merge-Join: An Idea whose Time Has(h) Passed? Proc. Data Engineering Conf. 1994: 406-417.

[4] Masaya Nakayama, Masaru Kitsuregawa, Hash-Partitioned Join Method Using Dynamic Destaging Strategy. Proc. VLDB Conf. 1988: 468-478

[5] D. E. Knuth: The Art of Computer Programming, Volume 1 (2nd Ed.), Addison-Wesley, 1973.

[6] D. E. Knuth: The Art of Computer Programming, Volume 3. Addison-Wesley, 1973.

[7] Per-Åke Larson: Grouping and Duplicate Elimination: Benefits of Early Aggregation, Manuscript submitted for publication, 1997. Available from the author.

[8] Vinay S. Pai, Peter J. Varman: Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis. Proc. Data Engineering Conf., 1992: 273-282

[9] Betty Salzberg: Merging Sorted Runs Using Large Main Memory. Acta Informatica, 27(3): 195-215 (1989)

[10] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles: Dynamic Storage Allocation: A Survey and Critical Review. In International Workshop on Memory Management, Kinross, Scotland, UK, 1995. Also http://www.cs.utexas.edu/users/oops/papers.html.

[11] LuoQuan Zheng, Per-Åke Larson: Speeding up External Mergesort. IEEE Transactions on Knowledge and Data Engineering, 8(2): 322-332 (1996)