MASTER'S DEGREE THESIS

MSc in Applied Mathematics

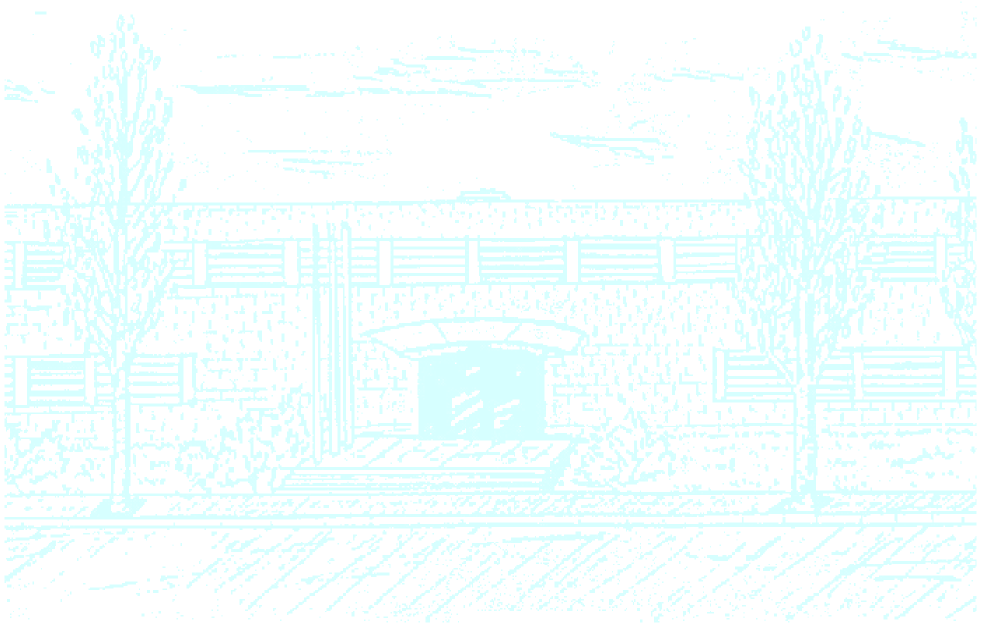
Title: Two-way Replacement Selection

Author: Xavier Martínez Palau

Advisor: Josep Lluís Larriba Pey / David Domínguez Sal

Department: Computer Architecture

Academic year: 2009-2010



Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Contents

Chapter 1

Introduction

Sorting is a fundamental computing problem that has received a lot of attention over the years. There are many situations where sorting is needed, ranging from just a single file to be read from disk, sorted, and written into disk again, to the case of sorting in the context of complex database queries, where different operators provide data to the sort operator, forcing different read and write steps. As a result, there is a large number of sorting algorithms, each with its own properties. In the case of in memory sorting, the two most widely used algorithms are quicksort and mergesort. Quicksort was developed in 1960 by C. A. R. Hoare, at the age of 26, and published in 1962 [5]. Mergesort was invented by John von Neumann in 1945, see [6] page 159.

A sorting algorithm is an algorithm that sets an order to the elements of a list. Typical sorting algorithms like mergesort and quicksort need all the elements to be stored in the memory of the computer being used to sort them. Some applications, the most typical being databases, need to sort more items than fit in memory. Thus, a new class of sorting algorithms is needed: one that sorts amounts of data that do not fit into the memory available to the computer. These algorithms are named *external sorting algorithms.* External sorting is a more complicated problem than regular sorting, also named *internal sorting* in contrast to external sorting, since it involves more input/output operations. One of the most commonly used external sorting algorithms is replacement selection (RS), introduced by Goetz in 1963 [3].

Replacement selection is commonly used because it has a good performance and is optimal if the input is already sorted. Replacement selection is based on the merge paradigm. This paradigm separates the sorting in two phases. The first phase consists in the generation of sorted lists of elements from the input, which are stored in secondary storage, typically hard disks. This lists are called *runs.* The second phase merges all the runs generated in the first phase in one sorted list containing all elements from the input. This second phase is faster to complete when the first phase generates few runs, which means they have long length.

However, replacement selection has several drawbacks. The most prominent drawback of RS is that the average length of the runs generated is not stable for all input distributions. This means that with structured inputs, the average run length may be very different. For example, if the input is already sorted, it generates only one run and the merge phase is immediate. However, if the input is sorted in reverse order, RS generates runs with minimal length and leads to the worst case performance.

Several modifications and alternatives to replacement selection have been proposed over the years. However, none of these proposals aim to generate longer runs. We propose a new algorithm that generalizes and improves replacement selection. The goal of this new algorithm is to consistently generate longer runs than RS, and address its drawbacks. We call this new algorithm two-way replacement selection (2WRS), which generalizes replacement selection and significantly improves its performance.

In this document, we propose the new two-way replacement selection technique, and analyze its performance in three different ways. First, we prove that 2WRS works better than RS for structured inputs, and performs as well as RS for random inputs. We also construct statistical models using analysis of variance (ANOVA) methods, to analyze the performance of the algorithm with respect to its configuration parameters. Later, we perform tests that compare the time needed by 2WRS to sort several sets of data with the time needed by replacement selection, and conclude that the new proposal is preferable to RS because 2WRS is more stable and generally faster.

1.1 Objectives

This document proposes a new external sorting algorithm, based on replacement selection, that aims at improving the time needed to sort amounts of data larger than the available memory by generating larger runs. The objectives of this work are summarized as follows:

- To understand the problem of external sorting, the typical algorithms used in this case, and the circumstances that motivate finding new solutions to this problem.
- To propose a new external sorting algorithm, two-way replacement selection that generates longer runs than replacement selection.
- To formally prove that 2WRS performance is at least as good as that of RS.
- To analyze 2WRS and the effects of its configuration, and confirm, with the aid of statistical methods (ANOVA), that it effectively generates longer runs for certain input distributions.
- To design a set of experiments that analyze time performance of replacement selection and two-way replacement selection.
- To confirm that the generation of larger runs implies better time performance of the whole sorting algorithm.

1.2 Organization

The rest of the document is organized as follows:

- Chapter 2 introduces the concept of external sorting, presenting the problem to be solved.
- Chapter 3 thoroughly explains the replacement selection technique and discusses some variations proposed over the years. Section 3.6, which sets a system of differential equations that models the behavior of RS, is a contribution.
- Chapter 4 introduces our algorithm, two-way replacement selection. This entire chapter is an original contribution.
- Chapter 5 proves some properties of RS and 2WRS and presents a statistical analysis of the performance of RS and 2WRS. This statistical analysis is used to select optimal 2WRS configurations for different input data distributions. The theorems in Section 5.1 regarding 2WRS and the statistical analysis of 2WRS in Section 5.2 are contributions.
- Chapter 6 analyzes and compares the time performance of RS and 2WRS.
- Chapter 7 summarizes and concludes the present document.

Chapter 2

External Sorting

External sorting is a term to refer to a class of sorting algorithms that can handle large amounts of data. The elements that are ordered by a sorting algorithm are referred to as records, and we will use this nomenclature in the rest of this document. When there are more records than those that fit in the main memory of the computing device used to sort the records, external sorting is required. Since more memory is needed, these algorithms rely on the use of slow external memory. Historically, magnetic tapes were used as external storage devices; nowadays, hard drives are used. Accessing and modifying external memory is much slower than doing the same with internal memory, so the execution time of external sorting algorithms is heavily influenced by the number of accesses to external memory. Thus, when analyzing the performance of an external sorting algorithm, one must consider the amount of input/output operations in addition to the algorithmic complexity of the algorithm.

The problem of external sorting arises typically in databases, because they are specialized applications for handling huge amounts of data. When databases perform operations with data, it is frequently necessary to sort part of the data as part of a more complex operation. For example, when joining data from two different tables, or grouping rows having common values, it is necessary to perform sort operations on data, although the sort is not explicitly requested. In most cases, the amount of memory available to the processes performing operations on the database is much less than the amount of data stored in it. In these cases, an external sorting algorithm is needed.

Two of the most commonly used generic approaches to external sorting are the *merge* and *distribution* paradigms. The merge paradigm consists on sorting records as they are read from the input, generating several independent sorted lists of records, that are merged into the final sorted list in a second phase. The distribution paradigm read records from the input and distributes them into *buckets* according to the range to which they belong, so that the ranges assigned to buckets do no overlap pairwise. Each bucket is then recursively sorted, using any sorting algorithm. The final sorted list is formed by concatenating the sorted contents of each bucket.

2.1 Mergesort

One of the most commonly approaches to external sorting is *External mergesort*, which consists of two phases, the *run generation* phase and the *merge* phase. The first phase generates several sorted lists of records, called *runs*, and the second phase merges the runs into the final sorted list of records.

Replacement selection, detailed in Chapter 3, is a sorting algorithm that performs the run generation phase of external merge sort. As such, it can be combined with any algorithm for the merge phase. It was introduced by Goetz in [3] and since then several modifications and alternatives have been proposed. Some of these modifications are explained in Section 3.7. When replacement selection was first introduced, it was not immediately used in commercial applications, because it was considered a complicated algorithm because of the computing limitations of the time. In 1998, Larson and Graefe experimentally compared different memory management algorithms during run generation when ordering variable length inputs, showing that replacement selection is a viable algorithm for commercial database systems [9].

2.1.1 Run generation phase

In the run generation phase, data is read from the input to generate subsets of ordered records. These subsets are called *runs.* Runs are generated using main (internal) memory, and written to external memory (disk). After all input records are distributed in runs, the run generation phase ends and the merge phase starts.

There are several methods used to generate the runs, most of them being based on internal sorting algorithms. For example, the main memory can be filled with records from the input and then sorted using any internal sorting algorithm (mergesort, quicksort, etc.) Using this method, called *Load-Sort-Store*, the run length is always equal to the size of the main memory, except for maybe the last run. Another more advanced algorithm is *replacement selection* (RS), which is described in Chapter 3. Using RS, the run length is approximately equal to twice the size of the internal memory when the input data is randomly distributed.

Generating longer runs means having fewer runs to merge in the next phase, which in turn allows for shorter execution times. RS is more complicated than a Load-Sort-Store solution, but overall less time is needed to sort records. Our algorithm, two-way replacement selection, presented in Chapter 4, generates longer runs than RS, as analyzed in Chapter 5. Chapter 6 shows that this results in faster execution times.

2.1.2 Merge phase

The purpose of the merge phase is to merge the runs generated in the previous phase into a unique run containing all input records. Two different ways of performing this phase are *k-way merge* and *polyphase merge.*

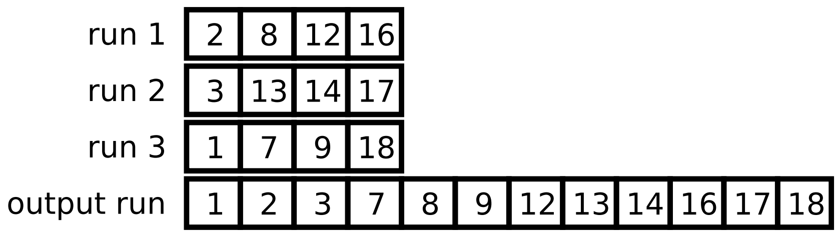


Figure 2.1: Three runs ready to be merged and the run resulting from the merge.

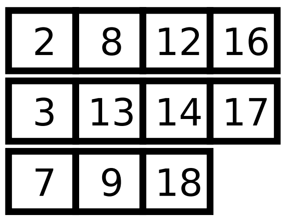


Figure 2.2: The three runs of Figure 2.1 after the first record has been removed and put in the output run.

k-way merge

A k-way merge combines k runs into one sorted run. This process reduces the number of runs to merge by $k-1$, and is repeated until only one run is left. The simplest example is the 2-way merge, where two sorted runs are merged into one. In a two way merge, we only need to know the first two records of each run. The smallest of these two records is the smallest record overall. This record is put in the output run and removed from the corresponding run. This process is repeated until the two initial runs are empty. The k-way merge behaves identically, but at each step the smallest of k records is selected and removed from its run.

As an example, consider that we want to merge 3 runs in a 3-way merge. We start with the 3 runs shown in Figure 2.1. At each step, we do the following:

1. Consider the first record of each run: {2, 3, 1}

2. Take the smallest record, 1, add it to the output run and delete it from the original run. At this point, the output run is {1} and the state of the three runs to be merged is shown in Figure 2.2.

The process of merging the three runs of Figure 2.1 is as follows:

- The three top records are {3, 2, 1}. Remove 1 from the third run and put it in the output: {1}.

- The three top records are {3, 2, 7}. Remove 2 from the second run and append it to the output: {1, 2}.

- The three top records are {3, 13, 7}. Remove 3 from the first run and append it to the output: {1, 2, 3}.

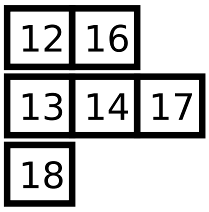


Figure 2.3: The three runs of Figure 2.1 after the sixth merge step. The output run at this point is: {1, 2, 3, 7, 8, 9}.

- The three top records are {8, 13, 7}. Remove 7 from the third run and append it to the output: {1, 2, 3, 7}.
- The three top records are {8, 13, 9}. Remove 8 from the first run and append it to the output: {1, 2, 3, 7, 8}.
- The three top records are {12, 13, 9}. Remove 9 from the third run and append it to the output: {1, 2, 3, 7, 8, 9}. The state of the three runs being merged at this point is shown in Figure 2.3.
- The three top records are {12, 13, 18}. Remove 12 from the first run and append it to the output: {1, 2, 3, 7, 8, 9, 12}.
- The three top records are {16, 13, 18}. Remove 13 from the second run and append it to the output: {1, 2, 3, 7, 8, 9, 12, 13}.
- The three top records are {16, 14, 18}. Remove 14 from the second run and append it to the output: {1, 2, 3, 7, 8, 9, 12, 13, 14}.
- The three top records are {16, 17, 18}. Remove 16 from the first run and append it to the output: {1, 2, 3, 7, 8, 9, 12, 13, 14, 16}. The first run is now empty, the merge follows as a 2-way merge instead of a 3-way merge.
- The two top records are {17, 18}. Remove 17 from the second run and append it to the output: {1, 2, 3, 7, 8, 9, 12, 13, 14, 16, 17}. Now, the second run is also empty, only the third run remains non-empty.
- The records of the las run, {18}, are appended to the output and the final run is obtained: {1, 2, 3, 7, 8, 9, 12, 13, 14, 16, 17, 18}.

The run that results of merging the three initial runs is {1, 2, 3, 7, 8, 9, 12, 13, 14, 16, 17, 18}.

Polyphase merge

Polyphase merge is a merging algorithm designed to minimize the amount of I/O [2]. It was developed when the more commonly used external storage devices were magnetic tapes. The algorithm starts with $k + 1$ tapes, one of them empty and the rest containing runs. At each step it performs a k-way merge, writing the output run in the initially empty tape. This step is repeated until any tape becomes empty. This emptied tape is then reused as the output tape. The whole process is repeated until all runs have been merged into one. Polyphase

	Tape 1	Tape 2	Tape 3	Tape 4	Tape 5	Tape 6
Step 0	8	10	3	0	8	11
Step 1	5	7	0	3	5	8
Step 2	2	4	3	0	2	5
Step 3	0	2	1	2	0	3
Step 4	1	1	0	1	0	2
Step 5	0	0	1	0	0	1
Step 6	1	0	0	0	0	0

Table 2.1: Example of a polyphase merge with 6 tapes. The table shows the number of runs stored in each tape after the execution of each step has completed.

merge is most efficient way of merging the runs when the number of output runs in each tape is different.

As an example, suppose that we have 6 magnetic tapes (or other external storage devices, numbered from 1 to 6), each containing a number not necessarily equal of runs, for example { 8, 10, 3, 0, 8, 11}. Table 2.1 shows the status of each tape at each step of the polyphase merge. The tape number 4 is the only one empty at the beginning, so it will be the output tape of the runs that result of performing 5-way merges of the runs stored in the other tapes. After 3 merges, tape number 3 becomes empty, since it had 3 runs at the beginning. The step ends here and tape number 3 becomes the new output tape. The algorithm ends when all tapes are empty except for one, that contains a unique run.

2.2 Distribution sort

Another approach to external sorting, different from the merge paradigm, is *distribution sort.* A *distribution* sort is a sorting algorithm where records are distributed to several intermediate data structures, named *buckets*, and then sorted and gathered together to form the final sorted list. These sorting algorithms are characterized by the fact that the ranges of values that each data structure accepts do not overlap between them, in contrast with external merge-sort, where the ranges of each run can overlap. As a result, the merge phase is unnecessary in a distribution sort, and the sorted contents of each bucket are concatenated to form the final sorted list of records.

External distribution sort algorithms are an adaptation of the internal sorting algorithm called *bucket sort.* Bucket sort partitions the records in several disjoint sets, called *buckets*, with the property that records in one bucket are smaller than records in the next one. Once the records have been distributed in buckets, the contents of each bucket are sorted using any sorting algorithm. Since each bucket stores records smaller than following bucket, the content of the buckets can be sorted independently and concatenated to obtain a sorted list of records. A schematic working of the algorithm would be as follows:

1. Set up a number of buckets and define the range of values of records accepted by each one.

2. Distribute: put each record into the bucket where it belongs.

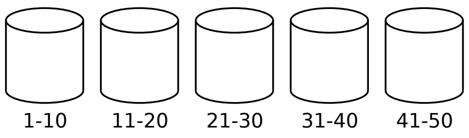


Figure 2.4: Five empty buckets with their corresponding range of accepted records.

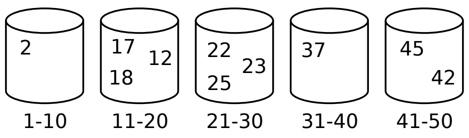


Figure 2.5: Five buckets filled with records.

3. Sort the contents of each bucket.

4. Concatenate the buckets to obtain the final list of sorted records.

The sorting algorithm used in Step 3 can be any other sorting algorithm. Using bucket sort again at this step results in a recursive version of bucket sort similar to *radix sort*. Radix sort is a sorting algorithm that can sort a number of records in linear time. Details on radix sort can be found in [6]. Using this recursive bucket sort with only two buckets corresponds to the *quicksort* algorithm.

Variants of bucket sort use different strategies to select the range of records that go into each bucket. The simplest and most common variant is to distribute the possible range of values uniformly among all the buckets. For example, let us consider the problem of sorting the following set of integers: {37, 2, 45, 22, 17, 12, 18, 23, 25, 42}. If we want to use 5 buckets, we set up 5 buckets, as shown in Figure 2.4. We know that all records are integers between 1 and 50, so each bucket has a range of 10 integers. Then we follow the second step: the integers are distributed, each one in its corresponding bucket. The result of this step is shown in Figure 2.5. Next, the contents of each bucket must be sorted, using any sorting algorithm, possibly bucket sort again. Finally, the records in each of the buckets are concatenated to obtain the final sorted list: {2, 12, 17, 18, 22, 23, 25, 37, 42, 45}.

Distribution sort operates similarly to bucket sort, but each bucket is stored in a disk file, since the main memory is not large enough to hold all buckets at once. If the contents of a bucket fit into the internal memory, an internal sorting algorithm can be used when recursively sorting its contents (Step 3). However, if the contents of a bucket do not fit into the internal memory, an external sorting algorithm needs to be used. This external sorting algorithm can be any one, possible a distribution sort or a mergesort. Therefore, mergesort algorithms are also used as part of distribution sort algorithms.

In order to make distribution sort efficient in terms of needed I/O operations and total execution time, it is necessary that the different buckets have a similar amount of records. If the range of possible records for each bucket is uniformly distributed along the total range of possible records, clustering of the records affects distribution sort negatively: if many records are close, they will all go to a single bucket. If the bucket sizes are not distributed uniformly, the recursion will take a long time to finish for the buckets with a larger number of records. The problem of choosing good intervals for each bucket is similar to the problem of choosing the pivot in the quicksort algorithm. There are several techniques that try to balance the load across the buckets, detailed in [6] and [14].

Chapter 3

Replacement Selection

Replacement Selection (RS) is an external sorting algorithm, based on the merge paradigm, introduced by Goetz [3]. The objective of RS is to sort a stream of records as they come (usually from secondary storage), producing another stream of released data records called *run*, which is sorted. Generated runs are always at least as large as the available memory, so it is at least as good as Load-Sort-Store in terms of generated run length.

This chapter is organized as follows: Section 3.1 details the data structure known as *heap*, Section 3.2 explains *heapsort*, a sorting algorithm upon which RS is based, Section 3.3 details the RS algorithm, Section 3.4 presents a pseudocode version of RS, Section 3.5 shows a proof of the average length of the runs generated by RS with random uniformly distributed input records. Section 3.6, which is an original contribution, closes the chapter presenting the construction of a mathematical model of RS.

3.1 Heaps

Before introducing RS, we describe some previous concepts. The RS algorithm uses a tree-based data structure called *heap.* A tree is a subtype of a more general entity called *graph.*

A *graph* G is an ordered pair of sets $G = (V, E)$. The elements of the set V are called *vertices* or *nodes* and the elements of the set E are called *edges*, and are subsets of V of size 2. Two edges $u, v \in V$ are called *adjacent* if the pair (u, v) is in the set E. We will only consider *simple* graphs, which are those that do not have nodes adjacent to themselves and each edge appears at most once. Figure 3.1 shows an example of a graph, where nodes have been labeled using integers from 1 to 7.

Given a graph $G = (V, E)$ and two edges $u, v \in V$, a *path* between u and v is a sequence $\{u, a_1, \ldots, a_n, v\}$ such that $a_i \in V$ for $1 \leq i \leq n$ and $(u, a_1) \in E$, $\{(a_i, a_{i+1}) \in E | 1 \leq i \leq n-1\}$ and $(a_n, v) \in E$. In other words, a path is a sequence of nodes in which every pair of consecutive nodes are adjacent. The *length* of a path P is defined as the number of nodes in the path minus 1, $\text{length}(P) = |P| - 1$. A path is called *simple* if each node appears at most once. Two edges $u, v \in V$ are *connected* if there is a path between u and v. A graph is said to be *connected* if every pair of nodes are connected. A *cycle* is a path

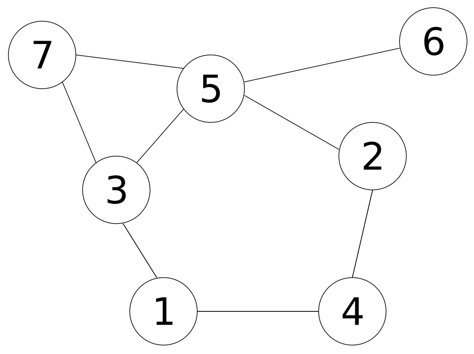


Figure 3.1: A graph.

between a node and itself.

We will use a subtype of graph called *tree*. A *tree* is a connected graph without cycles. There are several other definitions of tree, however, all of them are equivalent. For instance, a tree is a connected graph where there is a unique path between every pair of nodes. For the interested reader, more definitions and properties of trees can be found in [1].

A convenient form of describing a tree is classifying the nodes following the symmetric relation *parent of* and *child of*. In order to define the relation, we select an arbitrary node of the tree and designate it as the *root node*. When then have what is called a *rooted tree*. The *parent node* of a node u is the node connected to it in the unique path between the root node and u. A *child node* of a node u is a node whose parent is u. Note that in a rooted tree, the root has no parent and all other nodes have a single parent. If a node does not have any children, it is said to be a *leaf node*.

In a rooted tree, the depth of a node u is the length of the path between u and the root. The *height* of the tree is the maximum of the depth of each node.

An important set of trees are *binary trees*, which are trees with the property that each node has at most two children. If all nodes have exactly two children except possibly one, then the tree is called *complete* binary tree.

A *data structure* is a way to store and organize data so it can be accessed. A tree-based data structure maps a set of records to the set of nodes of a tree assigns a record to each node of a tree. A *heap* is a tree-based data structure that stores a set of records having a total order, denoted by $\leq$. There are several variations of the heap data structures. The most common, the *binary heap*, uses a complete binary tree. A heap stores records in the nodes of the tree satisfying the *heap property*, namely, that if a node v is a child node of u, then these records are such that $u \leq v$. This means that the record stored at the root node, called *top record*, is always the smallest record according to the total order defined on the records. If the order relation is changed from $\leq$ to $\geq$, the heap is called *max heap* because the top record is the greatest record stored in the heap. If the order relation is the usual $\leq$, the heap is sometimes called *min heap* to distinguish it from max heaps. Figure 3.2 shows an example of a completely filled max heap.

Replacement selection stores the records in a *binary min heap* in memory.

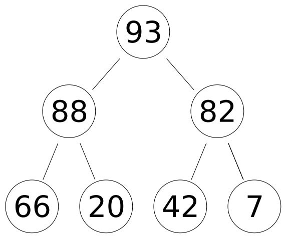


Figure 3.2: A completely filled binary max heap.

3.1.1 Operations

Heaps implement two operations, adding a record to it and popping the top record.

Adding a record

In order to add a record to a binary heap, a procedure named *upheap* is used. The upheap procedure is the following: the record is added at the bottom level of the heap, keeping the heap binary and complete. However, it is possible that the new record violates the heap property. A sequence of swaps is needed to restore this property. The process starts in the new node, which is compared to its parent. If they are in the wrong order, that is, if the heap condition is not satisfied, they are swapped. This comparison with the parent node goes on until the node is in the correct order with respect to its parent or the root node of the tree is reached.

It is possible to prove by induction the correctness of the upheap procedure: when the upheap procedure ends, the resulting tree is a heap, that is, the heap condition is satisfied everywhere in the tree. The induction hypothesis is that after the k-th step, the heap condition is satisfied in for every record in the last $k+1$ levels of the tree. When the $k+1$-th step begins, the heap condition may only be violated for a record in the level $k+2$ from the end, per the induction hypothesis. Only the record in this level that belongs to the path between the root and the new record can violate the heap condition, because the rest of the heap is identical to the heap before the new inserted node. The $k+1$ step swaps this record with one of its children if necessary, so after completion of this step, the heap condition is met everywhere in the last $k+2$ levels of the heap. Before the first step, the last level consists of records without children, so the heap condition is vacuously met, completing the proof.

As an example, consider the heap shown in Figure 3.2. This heaps stores integers with the total order $(\mathbb{N}, \geq)$, which means that it is a max heap. The process of adding a new record, 91, is depicted in Figure 3.3. When the new record is inserted, the 91 is first added at the end of the heap. Since the last level of the heap is already full, a new one is created with the new record, as shown in Figure 3.3(a). Now, the new record has to be compared with its parent. Since it is a max heap, the heap condition says that parent records need to be larger than child records, and in this case this condition is not met, since $66 < 91$, so they are swapped. The current state of the heap is shown in Figure 3.3(b). The heap condition must be checked with the new parent of 91, which is 88, and the condition is still not met. 91 is now swapped with 88 and the result is shown in

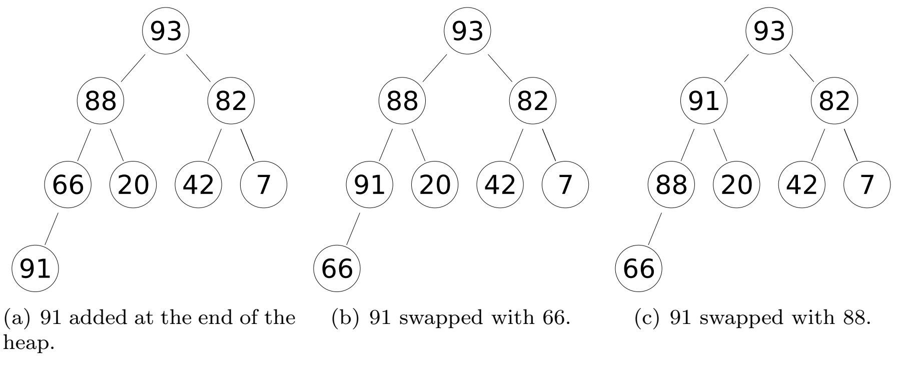


Figure 3.3: The process of adding the record 91 to the heap shown in Figure 3.2.

Figure 3.3(c). The new parent of 91 is 93, which is larger, so the heap condition is satisfied and the process of adding the new record to the heap ends. As it can be checked in Figure 3.3(c), the heap condition is satisfied for all records in the heap, as it is assured by the correctness of the upheap procedure.

The complexity of adding a record depends on the number of swaps. This number is bounded by the height of the tree. If there are n records stored on the heap, the depth size is $O(\log n)$. So, assuming that two records can be swapped in constant time, which can be done by swapping values when the record size is small or by swapping pointers, and that the time needed to compare two records is negligible in front of the time needed to swap two records, the process of adding a record to the heap has a time complexity of $O(\log n)$.

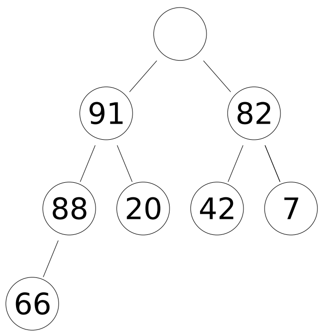
Popping the top record

In order to delete the root record of the heap, a procedure named *downheap* is used. The downheap procedure is the following: the root record is replaced with the last record on the last level of the tree. The record at the root node is then compared to its children, and if the heap condition is not met, it is swapped with the largest of the two children (smallest in a min heap). This is repeated until the heap condition is met or the last level of the tree is reached.

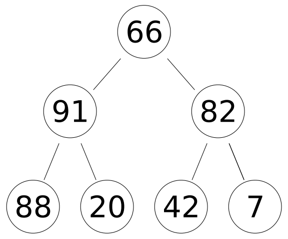
The proof of correctness of the downheap procedure is very similar to the proof of correctness of the upheap procedure, but changing the induction hypothesis by "After the k-th step, the heap condition is satisfied everywhere in the first $k + 1$ levels of the tree".

As an example, consider removing the top record of the heap in Figure 3.3(c). The process is depicted in Figure 3.4. The first step is to remove the top record, 93, and put in its place the last record of the last level, 66 in this case. The result of this step is shown in Figure 3.4(b). Now, the heap condition is not satisfied, because 66 is not greater than both its children, so it is swapped with the largest of the two child records, 91. The heap is now as shown in Figure 3.4(c). The second step is repeated, and since the heap condition is again not met, 66 is swapped with the largest of its two child records, 88. The heap is now as shown in Figure 3.4(d). Since 66 has no children the process ends here. It can be checked that in the end the heap condition is satisfied for all records.

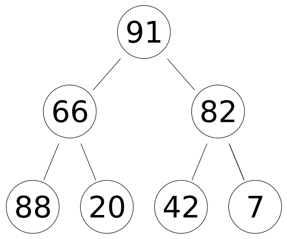
The number of swaps to be done is at most the height of the tree, so deleting



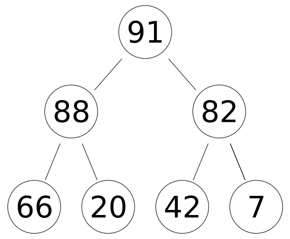
(a) The top record is removed.



(b) The last record of the last level, 66, is put at the top position.



(c) 66 swapped with 91.



(d) 66 is swapped with 88. The downheap ends.

Figure 3.4: The process of deleting the top record of the heap shown in Figure 3.3 (c).

the top record has a time complexity of $O(\log n)$, the same as adding a record.

The action of deleting the top record of a heap is referred to as *popping* the top record. Note that this operation always retrieves the maximum element of the record set stored in the max heap (minimum in a min heap).

3.1.2 Implementation

Heaps are stored contiguously in memory as arrays because of efficiency. The *array* is one of the simplest data structures: the records are stored as a sequence and they are accessed through a set of integer indexes. An *n-dimensional array* uses n integers to index each record. The simplest case is the *one-dimensional array*, in which records are indexed using only one integer. Usually if the array has n records stored, the index is between 0 and $n-1$. Vectors and matrices are typically implemented as one and two-dimensional arrays, and hence arrays are also referred to as vectors and matrices.

In order to use a one-dimensional array to store a heap, each node is labeled with an integer, starting with 0 for the root node and assigning the numbers in order level by level, left to right. If a node has the label i, its parent node has the label $\left\lfloor \frac{i-1}{2} \right\rfloor$, and its children nodes have labels $2i+1$ and $2i+2$. This is true for every complete binary tree with any number of nodes. Thus, a heap is stored in memory using an array of records, having each record indexed by the integer indicated by its label in the heap. In Figure 3.5, we depict a complete

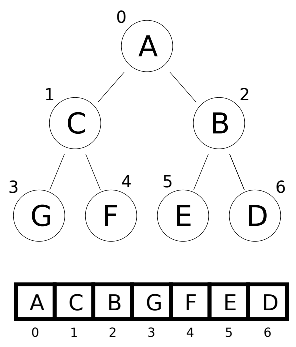


Figure 3.5: A binary complete tree with 3 levels and its associated array.

binary tree, representing a heap, and the array storing the heap.

Since the operations of access and modification of records in an array can be executed in constant time, this implementation allows for $O(\log n)$ time-complexity addition and deletion of records in the heap.

3.2 Heapsort

Heapsort is an internal sorting algorithm that uses a heap to sort records initially stored in an array. Heapsort is implemented with the aid of a heap in addition to the input array[1]. The heap is used to sort the records while the array stores them already sorted.

The algorithm performs two steps. The first step consists of inserting all n records into the heap, one by one. Once this process is finished, the top record of the heap is the smallest of the n records. This record is removed from the heap and put in the first position of the array, because it is the smallest record due to the heap property. The new top record of the heap is the second smallest record. This record is removed from the heap and inserted into the second position of the array. This process is repeated n, times: at each step, the top record of the heap is removed and inserted into the next empty position of the heap. Once the heap is empty, the array contains the n records already sorted.

In Section 3.1.1, it is shown that, in a heap with n records, the operations of adding and deleting a record require $O(\log n)$ time. When sorting n records using heapsort, each record is inserted and deleted one time, and written to the output array. When the i-th record is inserted, the heap has i records, so the insertion cost is $O(\log i)$. When this same record is removed, the heap has also size i, so the removal cost is also $O(\log i)$. Writing a record to the array has a constant cost, that is, $O(1)$. Thus, heapsort has a worst-case running time of

$$\sum_{i=1}^{n} (2 \cdot O(\log i) + O(1)) = \sum_{i=1}^{n} O(\log i) \leq \sum_{i=0}^{n} O(\log n) = O(n \log n)$$

[1]There are implementations of heapsort that use the input array to store the heap. However, for clarity, we use a separate array to store the heap

3.3 Replacement Selection

Replacement Selection (RS) is a run generation algorithm based on heapsort, that can be applied if the set of records to be sorted does not fit into memory. It uses a heap to store records. At each step, the top record of the heap is removed and put in the output, like heapsort does. Once a record has been removed, a new record is read from the input and inserted into the heap.

The main difference between heapsort and RS is the fact that, at each step, a new record is inserted into the heap. Another difference is that the output is written directly into a run, which is stored on disk, in order to have more internal memory available for the heap. This is possible because each time a record is removed from the heap, it has to be appended to the output and it is not needed again during the run generation, so there is no need to store all the sequence of output records in internal memory. The runs are written sequentially to disk, and there is no random access. This is a limitation of magnetic tapes, as they only allow sequential reading or writing. Hard disks allow random access, but sequential reading and writing is much faster.

If a record read from the input is smaller than the last record used as output in the current run, it is not be possible to use it as part of the current run, because records being output are already larger. This situation arises with RS and not with heapsort because RS adds new records to the heap. In this case, the record is marked as belonging to the next run. When a record marked is inserted into the heap, it is put at the bottom of the heap. In order to do this, it is considered that records belonging to the next run are larger than all records belonging to the current run. Therefore, when the top record of the heap belongs to the next run, all records stored in the heap also belong to the next run. The reasoning is the following: suppose that we had another record belonging to the current run in the heap. This record is smaller than the top record. Thus, the heap condition would be violated somewhere along the path that joins the top record with the record belonging to the current run. Since this is an impossible situation, if the top record belongs to the next run, so do the rest of the records stored in the heap.

The generation of the current run ends when the top record belongs to the next run, and then a new run is started. The algorithm proceeds using the same strategy until there are no more records to read from the input.

3.4 Pseudocode

Algorithm 1 shows the pseudocode for the main loop of RS. In the first phase, method *heap.fill* loads the first records from the input into the heap. This step is the same first step of heapsort: no records need to be marked as belonging to the next run since the output is still empty.

Then, the main loop is executed while the heap is not empty. First, a record is output to make room for a new one. This is done by method *output*. Next, a record is read from the input. If the record is smaller than the last output record, it is marked as belonging to the next run, else it is marked as belonging to the current run. Next, the top record of the heap is removed with *heap.pop* and the record read from the input is inserted in the buffer with *heap.insert(current)*. This method checks whether the record to be inserted belongs to the current or

Algorithm 1 RS(heapSize)

Require: The maximum size of the heap *heapSize*.
Ensure: Each run is sorted.

```
let current a pair of integers containing a value for a record and the run to
  which it belongs.
let heap a min heap, of maximum size heapSize.
let currentRun an integer.
let nextOutput an integer.
heap.fill(inputBuffer);
currentRun = 0;
while heap.size() > 0 do
    nextOutput = heap.pop()
    write(nextOutput);
    //Read next value
    if input.read(current.value) then
        if current.value < nextOutput then
            current.run = currentRun + 1;
        else
            current.run = currentRun;
        end if
        heap.insert(current);
    end if
    //Start next run?
    if heap.top().run > currentRun then
        currentRun = 1 + currentRun;
    end if
end while
```

next run, and inserts it accordingly.

Finally, when the top record of the heap belongs to the next run, i.e., it is too small to be part of the current run, the current run ends. Since all records stored in the heap are larger than the top record, if the top record belongs to the next run, all other records also belong to the next run. The whole process starts again, and new runs are generated until the input is fully read.

3.5 Run length

The length of the runs generated by a run generation algorithm like RS has an impact on the performance of the merge phase. If we consider a k-way merge as the merging algorithm, older machines stored runs using tapes, so the value of k is bounded by the number of tapes used to store runs. Newer computer are able to perform a k-way merge for small and large values of k, but there is an optimum value of k in terms of performance, as shown in Subsection 6.1.1. Thus, if a run generation algorithm creates larger runs, the total number of runs to be merged decreases, and the merge phase has a shorter execution time.

When the input records follow a random uniform distribution, RS generates runs that have an average length equal to the size of the available memory.

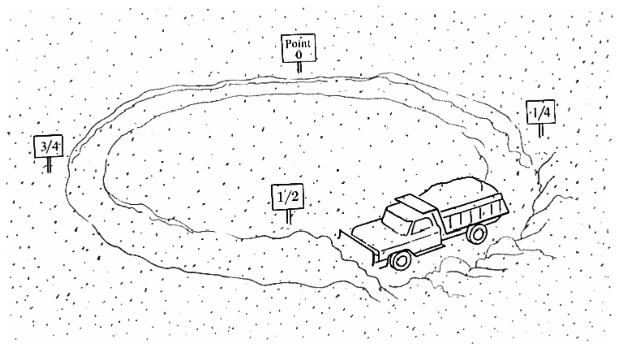


Figure 3.6: Circular road with a snowplow.

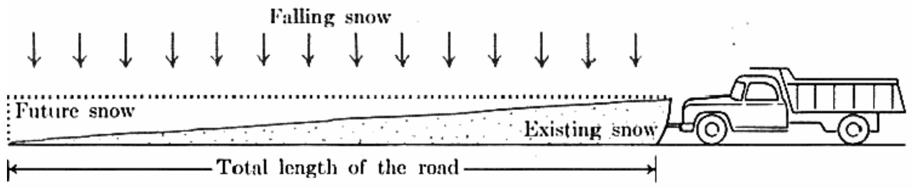


Figure 3.7: Stabilized situation.

There is a proof of this fact in [6]. This proof is intuitive because it builds a physical model for the problem, and it is reproduced here.

The proof considers a circular road on which snow flakes drop at a constant rate. A snowplow is continually clearing the snow, as shown in Figure 3.6. Once the snow is plowed, it disappears from the system. Points on the road may be designated with a real number $0 \leq x < 1$. A snowflake falling on x represents an input value of x. That is, we consider input records to be between 0 and 1, without loss of generality.

The snowplow represents the output of RS, and has a speed inversely proportional to the height of the snow it encounters. The situation is balanced so the quantity of snow in the road is P at all times (so P corresponds to the total memory available). The generation of a run finishes when the snowplow passes through $x = 1$.

After operating for a while, this system will approach a stable situation where the snowplow has a constant speed, because of the circular symmetry of the road. This means that the snow is at a constant height in front of the snowplow and decreases linearly in front of it. It follows that the amount of snow removed at each revolution is $2P$ (see Figure 3.7). The first triangle represents the snow which is already on the track and has size P, and the second one represents snow that will fall while the snowplough is running, and it also has size P.

3.6 Mathematical Model

In this section, we generalize Knuth's model by constructing a set of equations that model the behavior of the replacement selection algorithm in order to be able to analyze it mathematically. The purpose of having a mathematical model is to have a formal tool to easily analyze the behavior of the algorithm with different inputs.

We assume that the data in memory are real values between zero and one, and model the contents of the memory with density distributions. Let $t \geq 0$ be the time and $0 \leq x < 1$ a variable indicating a sorting key. Let $m(x,t)$ be a two variable function. On a given instant $t = t_0$, $m(x, t_0)$ indicates the contents of the memory as a density distribution. So, for instance, if at the beginning the memory is filled with data uniformly distributed between 0 and 1, then $m(x, 0) = 1$; or, if the data is linearly decreasing, $m(x, t_0) = 2 - 2x$, as it is the case when the situation has stabilized on Knuth's proof (previous section).

The percentage of total memory containing values between x_1 and x_2 at $t = t_0$ is $\int_{x_1}^{x_2} m(x, t_0)\, dx$. The percentage of memory used at any given moment $t = t_0$ is given by the integral $\int_0^1 m(x, t_0)\, dx$. To fix that the total amount of data doesn't exceed available memory, equation 3.1 must be met. In the ideal situation, equality holds and all memory is used at all times.

$$\int_0^1 m(x,t)\, dx \leq 1 \tag{3.1}$$

Let $p(t) \geq 0$ be a real valued function, which corresponds to the value being output at instant t. Since values in memory are modeled to be between zero and one, the floor of $p(t)$ is subtracted from $p(t)$ when used as the argument x of $m(x,t)$. This models the periodic behavior of RS that generates multiple runs to form a sorted set. Each interval defined by $p(t) \in [n, n+1)$ represents the time spent building the n-th run. The derivative of $p(t)$ can be seen as the velocity with which p advances to complete the current run and models the snowplough from the previous section.

If we consider that the throughput of the algorithm is constant, that is, that the number of values being output per unit of time is constant, $\frac{dp}{dt}(t)$ is inversely proportional to the value of the function $m(p(t) - \lfloor p(t) \rfloor, t)$. This means that the larger $m(p(t) - \lfloor p(t) \rfloor, t)$ is, the larger is the probability to find elements in memory in the range $(p(t) - \lfloor p(t) \rfloor - \varepsilon, p(t) - \lfloor p(t) \rfloor + \varepsilon)$ and thus the output values, given by $p(t)$ increase slowly, so $\frac{dp}{dt}(t)$ is smaller. That means that the derivative of p is inversely proportional to the value of m, as Equation 3.2 shows. $p(t) - \lfloor p(t) \rfloor$ models the value being output at instant t, so it is a strictly increasing function. If $m(p(t_0) - \lfloor p(t_0) \rfloor, t_0) = 0$ then $p(t)$ has a jump discontinuity at $t = t_0$ and it jumps from $p(t_0)$ to $\inf_{x > p(t_0)}\{x | m(x - \lfloor x \rfloor, t_0) > 0\}$.

$$\frac{dp}{dt}(t) = \frac{k_1}{m(p(t) - \lfloor p(t) \rfloor, t)} \tag{3.2}$$

The throughput of the algorithm, which corresponds to the number of records being output per unit of time, can be calculated from Equation 3.2 as Equation 3.3 shows.

$$\text{Throughput} = m(p(t_0) - \lfloor p(t) \rfloor, t_0) \cdot \frac{dp}{dt}(t_0) = k_1 \tag{3.3}$$

As the records are output, they are cleared from memory. The record being output at instant $t = t_0$ is given by $p(t_0)$. Just after records are output, they are cleared from memory. This means that $m\left(p(t_0), t\right)$ has limit zero as t approaches t_0 from the right. The function $m(x, t)$ has a jump discontinuity at $x = p(t) - \lfloor p(t) \rfloor$ and the limits in Equation 3.4 hold.

$$\begin{cases} \lim_{t \to t_0^+} m\left(p(t_0) - \lfloor p(t_0) \rfloor, t\right) & = 0 \\ \lim_{t \to t_0^-} m\left(p(t_0) - \lfloor p(t_0) \rfloor, t\right) & = m(p(t_0) - \lfloor p(t_0) \rfloor, t_0) \end{cases} \tag{3.4}$$

In order to model data that enters the system, we define $\text{data}(x)$, $0 \leq x < 1$ to be the distribution that input data follows. At any given instant t, $\text{data}(x)$ gives the rate of increase of $m(x, t)$, so $\frac{\partial m}{\partial t}$ is proportional to $\text{data}(x)$, as Equation 3.5 states. For example, for uniformly distributed data, $\text{data}\,(x) = 1$.

$$\frac{\partial m}{\partial t}(x, t) = \text{c}(t) \cdot \text{data}\,(x) \tag{3.5}$$

The flow of data joining the system is given by the expression in Equation 3.7, where $\int_0^1 \text{data}(x)\,dx$ is substituted by a constant, k_2.

$$\begin{aligned} \text{Input flow} &= \int_0^1 \frac{\partial m}{\partial t}(x, t_0) \\ &= \int_0^1 \text{c}(t_0) \cdot \text{data}(x)\,dx \qquad (3.6) \\ &= \text{c}(t_0) \int_0^1 \text{data}(x)\,dx \\ &= \text{c}(t_0) \cdot k_2 \qquad (3.7) \end{aligned}$$

If we equal the input flow (Equation 3.7) to the throughput (Equation 3.3) we obtain that $\text{c}(t)$ is the constant function k_1/k_2, as Equation 3.8.

$$\text{c}(t) = \frac{k_1}{k_2} \tag{3.8}$$

If we fix k_1 as an arbitrary constant corresponding to the throughput of the algorithm and $k_2 = \int_0^1 \text{data}(x)\,dx$, the derived model is:

$$\frac{dp}{dt}(t) = \frac{k_1}{m\left(p(t) - \lfloor p(t) \rfloor, t\right)} \tag{3.9}$$

$$\begin{cases} \lim_{t \to t_0^+} m\left(p(t_0) - \lfloor p(t_0) \rfloor, t\right) = 0 \\ \lim_{t \to t_0^-} m\left(p(t_0) - \lfloor p(t_0) \rfloor, t\right) = m(p(t_0) - \lfloor p(t_0) \rfloor, t_0) \end{cases} \tag{3.10}$$

$$\frac{\partial m}{\partial t}(x, t) = \frac{k_1}{k_2} \cdot \text{data}\,(x) \tag{3.11}$$

$$\int_0^1 m\,(x, t)\; dx \leq 1 \tag{3.12}$$

3.6.1 Estimating run length

The model from previous section can be applied to estimate the run length for a given distribution. Each run is built during the time interval comprised between the passing of $p(t)$ between two consecutive integers. The length of a run (with respect to the total available memory) corresponds to the path integral of $m(x,t)$ along the line described by $p(t)$.

We illustrate the process for the case of uniformly distributed input data. $\text{data}(x) = 1$, $k_2 = \int_0^1 \text{data}(x)\,dx = \int_0^1 1\,dx = 1$, and fixing $k_1 = 1$, a solution to the system of equations is given by:

$$p(t) = \frac{t}{2}$$

$$m(x,t) = \begin{cases} 2 - 2x + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right), & \text{if } x \geq \frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor \\ -2x + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right), & \text{if } x < \frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor \end{cases}$$

In this solution, data is distributed in memory in triangular form. At the start of each run, the memory contents distribution is $2 - 2x$, and at any given $t = t_0$, it is distributed starting at $p(t_0)$ and it decreases linearly, up to $x = 1$, and from $x = 0$ to $x = p(t)$, where the limit $\lim_{x\to p(t_0)^-} m(x,t_0) = 0$.

All equations of the model must be checked. The first one, Equation 3.9 is a simple calculation:

$$\begin{aligned} m\left(p(t) - \lfloor p(t)\rfloor, t\right) &= m\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor, t\right) \\ &= 2 - 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) \\ &= 2 \end{aligned}$$

$$\frac{dp}{dt}(t) = \frac{1}{2} = \frac{1}{m\left(p(t) - \lfloor p(t)\rfloor, t\right)}$$

Equation 3.10 is verified as follows:

$$\begin{aligned} \lim_{t\to t_0^+} m(p(t_0) - \lfloor p(t_0)\rfloor, t) &= \lim_{t\to t_0^+} -2\left(p(t_0) - \lfloor p(t_0)\rfloor\right) + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) \\ &= \lim_{t\to t_0^+} -2\left(\frac{t_0}{2} - \left\lfloor\frac{t_0}{2}\right\rfloor\right) + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) \\ &= 0 \end{aligned}$$

$$\begin{aligned} \lim_{t\to t_0^-} m(p(t_0) - \lfloor p(t_0)\rfloor, t) &= \lim_{t\to t_0^-} 2 - 2\left(p(t_0) - \lfloor p(t_0)\rfloor\right) + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) \\ &= \lim_{t\to t_0^-} 2 - 2\left(\frac{t_0}{2} - \left\lfloor\frac{t_0}{2}\right\rfloor\right) + 2\left(\frac{t}{2} - \left\lfloor\frac{t}{2}\right\rfloor\right) \\ &= 2 \\ &= m(p(t_0) - \lfloor p(t_0)\rfloor, t_0) \end{aligned}$$

To check the third equation, we calculate the partial derivative $\frac{\partial m}{\partial t}$ using that $\left\lfloor\frac{t}{2}\right\rfloor$ is constant piecewise, and its derivative is zero almost everywhere:

$$\frac{\partial m}{\partial t}(x,t) = 1 = \text{data}(x)$$

Finally, the last equation is:

$$\begin{aligned}
\int_0^1 m\left(x,t\right)\,dx &= \int_0^{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor} m\left(x,t\right)\,dx + \int_{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^1 m\left(x,t\right)\,dx \\
&= \int_0^{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor} -2x + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\,dx+ \\
&\quad + \int_{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^1 2-2x + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\,dx \\
&= \int_0^{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor} -2x + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\,dx+ \\
&\quad + \int_{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^1 -2x + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\,dx+ \\
&\quad + \int_{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^1 2\,dx \\
&= \int_0^1 -2x + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\,dx+ \\
&\quad + \int_{\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^1 2\,dx \\
&= -x^2 + 2x\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right)\Bigg|_{x=0}^{x=1} + \\
&\quad + 2x\Big|_{x=\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor}^{x=1} \\
&= -1 + 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right) + 2 - 2\left(\frac{t}{2}-\left\lfloor\frac{t}{2}\right\rfloor\right) \\
&= 1
\end{aligned}$$

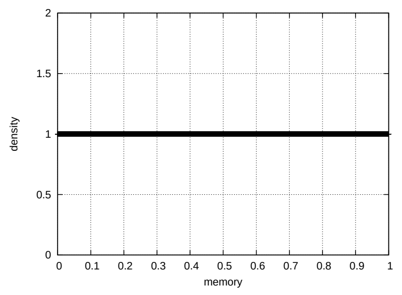
In this case, the whole memory is used at all times, because the equality holds.

Each time the value of $p(t)$ is an integer, the generation of a run ends and the next run is generated. $p(t) = t/2$, so the n-th run begins at $t = 2n$ and ends at $t = 2n + 2$. The size of the n-th run corresponds to the following path integral, using $m\left(p(t), t\right) = 2$ and $p'(t) = 1/2$:

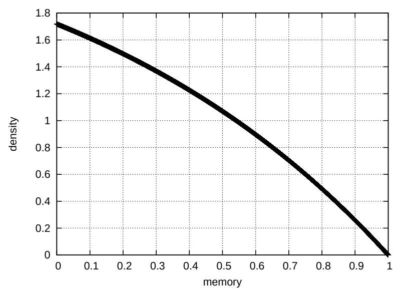
$$\begin{aligned}
\int_{p(t)} m(x,t) &= \int_{2n}^{2n+2} m\left(p(t),t\right)p'(t)dt \\
&= \int_{2n}^{2n+2} 2\cdot\frac{1}{2} \\
&= 2n+2-2n \\
&= 2
\end{aligned}$$

which means that the length of all runs is twice the memory available. This solution corresponds to the stabilized situation of Knuth's proof in Section 3.5.

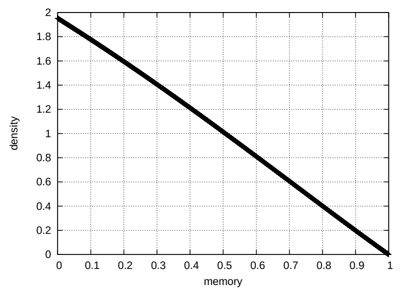
We also verify that starting with uniform memory contents distribution, the solution approaches the stable solution given. We have not been able to find an



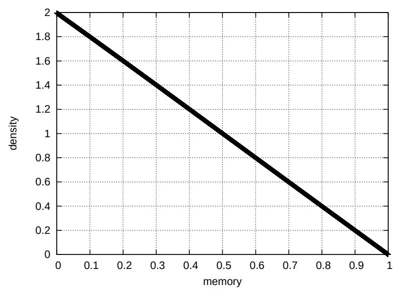
(a) Density distribution of memory contents before the algorithm starts.



(b) Density distribution of memory contents after the generation of the first run.



(c) Density distribution of memory contents after the generation of the second run.



(d) Density distribution of memory contents after the generation of the third run.

Figure 3.8: Evolution of density distribution of memory contents after completion of the first three runs. Data in memory at the beginning and input data is uniformly distributed.

analytic solution for the system of differential equations, so we solved it numerically implementing a version of the classical Runge-Kutta method [8] adapted to this particular system. For the initial condition $m(x, 0) = 1$, which corresponds to having data in memory uniformly distributed when the algorithm starts, we found that the system rapidly converges to the stable solution provided. Figure 3.8 shows the evolution of the density distribution of the contents of the memory just before the generation of a run starts. The stable solution has $m(x, 2n) = 2 - 2x$. After the completion of the second run, the density distribution is almost $2 - 2x$, and after the generation of the third run, the density distribution is indistinguishable from the theoretical $2 - 2x$.

3.7 Related Work

Since replacement selection was introduced, several modifications and alternatives have been proposed, and in 1998 Larson and Graefe showed that RS is a viable algorithm for commercial database systems [9]. Moreover, a recent survey on sorting in database systems pointed out that replacement selection is one of the most used techniques for external sorting in databases [4].

In this section we present some of the variations and techniques based on RS that have been proposed over the years. We should note that all modifications and improvements of RS can be readily applied to two-way replacement selection without change, so 2WRS also benefits from all these changes, while emphasizing on the generation of larger runs.

3.7.1 Batched replacement selection

Larson introduced a modified version of replacement selection called batched replacement selection, a cache conscious version that also works for variable length records [10]. In his paper, Larson first introduces a version of replacement selection that can be used with variable length records. In this version, the heap is not of constant size. When a record is removed from the heap and output, it is possible that the next record does not fit in memory. In this case, more than one record is output without introducing new records. It is also possible that several new records fit together in the newly created space. In this case, several records are added to the heap without removing any other record.

Larson argues that when adding or deleting a record, a branch of the heap is explored, and this branch is independent of previous explored branches. Thus, the records near the top record are accessed frequently and kept in the cache, but records closer to the leaves are not, and generate cache misses. Batched replacement selection reduces the number of cache misses by reducing the size of the heap. Incoming records are not immediately inserted into the heap, and instead are kept sorted in intermediate buffers called miniruns. When a record is removed from the heap, the next record from its minirun is inserted.

Batched replacement selection introduces another modification in order to further reduce cache misses. The main loop of replacement selection outputs one record, and then reads one record from the input. The main loop of batched replacement selection outputs a batch of 1000 records, and then fills the freed memory with a batch of records read from the input.

The experiments show that batched replacement selection generates fewer cache misses, executes fewer instructions, needs less comparisons and runs faster than replacement selection.

3.7.2 Reading strategy

The typical merge phase implementation uses one buffer for each run being merged. In this case, reading new records from the runs cannot be overlapped with the processing. The reading is on hold until one buffer becomes empty, and the processing has to wait for the emptied buffers to be filled.

One reading strategy, proposed by Knuth [6], is *forecasting.* Forecasting uses an additional buffer. When one block from each run being merged is in memory, the last key of each block is compared to determine which buffer will be emptied first. The additional buffer is filled with records from that run, while the algorithm is already merging runs.

Another strategy is *double buffering*, proposed by Salzberg [13]. This strategy allocates two buffers for each input run. One buffer from each run is filled with records, and the merging starts while the rest of the buffers are filled. When one buffer becomes empty, the other one already has more records, so the processing continues while the other buffer is filled. The drawback of this strategy is that it either needs more memory, or each buffer is smaller, which means that more disk readings are necessary.

Zheng and Larson introduced a new reading strategy, called *planning*, for improving the performance of external mergesort [18]. The planning strategy precomputes the order in which data blocks are to be read during the merge phase.

The planning reading strategy is based on the following observation. If there are many more buffers than runs, the extra buffers can be used for reading some blocks of data that are not immediately required by the merge algorithm but that are located contiguously in disk after a read block of data. The cost of reading these extra blocks is minimal, since there is no seek or rotational overhead. If these blocks are needed soon by the merge algorithm, the buffers can be quickly reused for reading more contiguous data blocks.

Planning uses a heuristic algorithm to decide the order in which each data block will be read from disk. The heuristic tries to minimize the total time needed to read the data blocks. The time needed to read two different data blocks depends on the relative position in which they are stored on disk, which is known by the heuristic. The heuristic algorithm also ensures that the situation where all buffers are filled with immediately unneeded data does not arise.

The heuristic decides the order in which data blocks are to be read. In order to do so, it constructs a read sequence and increases it by adding one block at a time. The new record is added at the end of the read sequence, and is moved to an earlier position if it can be read together with another block residing in the same disk cylinder. This heuristic does not take into account seek distance or rotational delay.

The experimental results show that this new reading strategy consistently outperforms forecasting and double buffering, due to the reduction of the number of disk seeks needed to read all data. Also, beyond a certain point, forecasting and double buffering do not benefit from additional buffer space, while planning continues to benefit.

3.7.3 Dynamic memory adjustment for external mergesort

When the input size is unknown or available memory space varies during the execution of a sorting algorithm, static memory allocation either wastes memory space or fails to make full use of all available memory. In these cases, dynamic memory allocation is necessary. Zhang and Larson presented a new method for run time adjustment of in-memory workspace for external mergesort and a policy for allocating memory when multiple sorts compete for memory resources [17].

The external sorting algorithm that the authors consider in the article is one based on the merge paradigm with the three following phases:

- *In-buffer sort*: during this phase, the sort process uses buffers to store data. The contents of each buffer are sorted using any internal sorting algorithm. When the process runs out of free memory, it tries to allocate some more space for buffers. If successful, this phase continues. Else, the next phase begins.

- *In-memory merge*: during this phase, the buffers are merged into one sorted run. As buffers are emptied, they may be freed if the system is short of memory or filled with data for the next run.

- *external merge*: in this phase, the runs are merged into the final sorted list of records.

Moreover, the sort process is divided in seven stages, from the beginning, when the sort is waiting to start, to the end, when the process is merging runs into the final sorted list.

When there are several sort processes being executed concurrently, the memory adjustment policy decides whether to give more memory to precesses in the first phase, and whether the processes can keep buffer memory or not in phase two. The policy makes the decisions based on total available memory, the memory already assigned to each process, and the stage in which the sort process is. In five situations, a process may choose to wait for memory when the request for memory is denied:

1. The process is about to start.
2. The process is generating the first run in memory, and has the minimum memory possible assigned.
3. The process is generating the first run in memory, but has more than minimum memory assigned.
4. The process is processing the rest of the runs during the in-buffer sort phase.
5. The process is before an external merge step.

The processes are prioritized in the following order: 1, 3, 5, 4 and 2. The reasoning is the following. Sort processes in situation 1 have the most priority to give a chance to very small sorts to finish. Processes in situation 2 have low priority because they use few memory. Processes in situation 3 have high priority to give them a chance to proceed to further stages. Finally, Processes in situation 5 have priority over processes in situation 4 because they hold more memory and are closer to completion.

The policy defines minimum and maximum memory values for the first run, for the rest of the runs and for the external merge phase. When a process fails to allocate more memory, it can either continue sorting with the already allocated memory or, if it has not reached the maximum memory, wait for the needed memory to be available. When it decides to wait, it does so with the assumption that the total sorting time will be shorter because having more memory the latter sorting stages finish faster. When memory is freed, the policy assigns it to waiting processes, giving priority to processes depending on the sorting stage they are on and the memory already allocated by them.

The experiments performed by the authors showed that this memory adjustment policy significantly improves sort throughput and response times when compared to static memory allocation.

3.7.4 Sorting hierarchical data

In 2008, Koltsidas, Müller and Viglas introduced a new variation of replacement selection for sorting hierarchical data, for example XML[2], which they call Hermes [7].

[2] *Extensive markup language*, a set of rules to encode hierarchical data

Hierarchical data is structured in levels arranged in a tree-like data structure. The tree is stored in disk similarly to how an XML file is. A node is specified by a pair of opening and ending tags, and its children are specified within. The data stored in a node is specified just after the opening tag.

Each node of the tree storing the hierarchical data stores a key. Nodes having the same parent are sorted by the value of their key. The sorting process consists on sorting the children of the root node by key value, and then recursively repeating this process for each child node. The algorithm proposed by the authors reads the file and stores the tree in memory. When the memory is filled, the algorithm selects the subtree with minimum key between those that have been completely read, applies replacement selection at each node to obtain a sorted run of its children, and outputs the entire subtree to the current run.

Once the whole tree has been processed, the generated runs are merged to form the final output. The result is a file like the input one, but where the children of each node are specified in the correct order.

The authors prove that the average length of a run is twice the size of available memory, like with replacement selection. The experimental results compared Hermes with Nexsort, another external sorting algorithm for hierarchical data. The results show that Hermes outperforms Nexsort, being 8.5 to 10.8 times faster.

3.7.5 Compression of records

Recently, Yiannis and Zobel studied the possibility of compressing sets of records during the run generation phase in order to reduce disk and transfer costs of external sorting by reducing the number of runs generated, and proposed new compression techniques adapted to sets of records [16].

Compressing records has two advantages. One is that it can reduce the I/O costs of transferring uncompressed data if the overhead of compressing and decompressing the data plus the I/O costs of transferring the compressed data is smaller. Also, having the records compressed in memory allows to generate longer runs, reducing the cost of the merge operation.

Compression techniques that work on records meet several strong constraints: they must allow individual access to records, record reordering and they must use a low amount of memory, since memory is needed by the sorting algorithm. Also, the compression and decompression needs to be fast enough so the whole sorting process is faster overall than if not using compression. In order to be able to sort the records after compression, the sorting key is not compressed, and only the values of the records are compressed. Such values are assumed to be strings of characters.

Their most successful compression technique is based on a ternary search trie[3]. This trie is used to store statistics on common strings observed in the data. Common strings are substituted by bytewise codes, using shorter codewords for the most common strings. This technique is parametrized to to balance memory requirements, compression efficiency and compression effectiveness.

The results obtained by the authors show that, in the best case, the total sorting time is reduced by about 36%, and the temporary disk space used by about 60%.

[3]A *trie* is a tree based data structure that stores ordered data, usually strings.

Chapter 4

Two-way Replacement Selection

We propose a modification of replacement selection, which we call *Two-way Replacement Selection* (2WRS), characterized by the use of two heaps instead of one and two intermediate buffers, named *input* and *victim* buffers. The purpose is to generalize RS and to be able to tailor it to different distributions of the input data in order to maximize the run length.

One bad thing about RS is that it produces runs of infinite length when the input is already sorted, but when it is sorted in reverse order the runs are of the minimal length possible. While the two situations are symmetrical, the performance of the algorithm is extremely different. By having two heaps, 2WRS behaves identically when the input is already sorted as well as when it is sorted in reverse order, a symmetry that RS lacks.

4.1 Algorithm

Figure 4.1 shows a functional diagram of 2WRS. Records read from the input are inserted into the input buffer, which is a FIFO queue. Records from the input buffer can go to one of two heaps or to the victim buffer. The victim buffer is explained in Section 4.3. One of the two heaps stores records greater than those already in the output run, exactly as RS does. This heap is called *TopHeap* and is a min heap. The new heap, called *BottomHeap*, stores records smaller than those already output and is a max heap. Records from both heaps and from the victim buffer are finally stored in four streams. Streams are FIFO lists of sorted records. The TopHeap is a min heap, meaning that each new record output by it will be larger than the preceding one, so the stream generated by the TopHeap, Stream 1, is increasing. Similarly, the BottomHeap is a max heap, so the stream associated with it, Stream 4 is decreasing. Streams 2 and 3 are explained in Section 4.3. Each stream contains records in a fixed range, in a way that the four different ranges do not overlap pairwise. Thus, the final output run is generated concatenating the contents of the four streams.

The two heaps occupy together a fixed amount of memory, but the size of each one changes during the execution of the algorithm. If both heaps are stored in memory as an array, as explained in Section 3.1, they must be stored

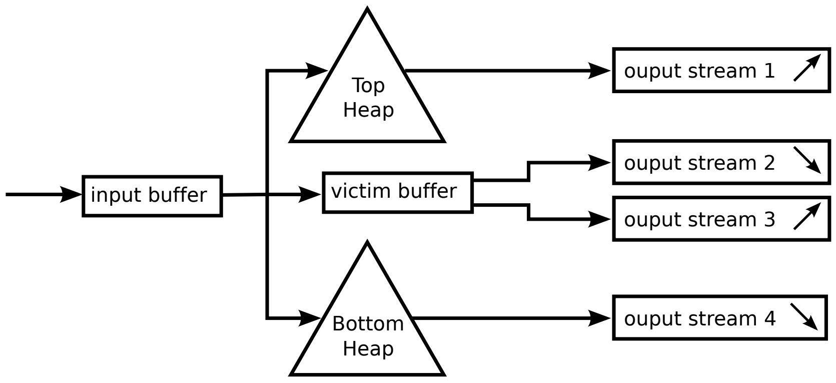


Figure 4.1: Functional diagram of 2WRS.

as dynamic data structures. This means that memory is dynamically allocated and freed continually, adding an overhead to the execution time of 2WRS. In order to avoid this, both heaps are stored contiguously in a single static array. One heap is stored starting at position 0 and growing with increasing memory indexes and the other one starting at the end and growing with decreasing memory indexes. This way one of them can grow larger at the expense of the other one shrinking.

The two heaps are stored contiguously in memory, one starting at position 0 and growing with increasing memory indexes and the other one starting at the end and growing with decreasing memory indexes. This way one of them can grow larger at the expense of the other one shrinking.

In order to illustrate how two heaps are stored in one array, let us consider, as an example, the two heaps shown in Figure 4.2. The one in the left is a max heap, corresponding to a BottomHeap, while the other one is a min heap, corresponding to a TopHeap. In the same figure we also depict how they are stored in memory, each in a different array. Figure 4.3 shows the same two heaps and the way they are stored in a single array following the storage structure used by 2WRS described here. The thicker line shows the point in the array that separates both heaps. In the actual implementation, this separation does not physically exist. Instead, the size of each heap is known. In this case, knowing that the BottomHeap has seven records allows to logically separate the two heaps in the array. Note that the only difference between the two-array and the single-array storing methods is that the TopHeap is stored in the array after the BottomHeap in reverse order.

As an example of how one heap can grow at the expense of the other one shrinking, we consider the following scenario: if the top record of the BottomHeap, 33, is removed, the resulting situation is shown in Figure 4.4. The size of the BottomHeap has been reduced by one unit, and now there is an empty position in the array storing the heaps. This position corresponds to a record removed in the BottomHeap, and it is positioned next to the end of the subarray containing the BottomHeap. Thus, another record can be inserted in it. But this empty position also lies at the end of the TopHeap, next to the record 77, so a new record can be inserted in the TopHeap. If the record 53 is inserted into the TopHeap, the heaps are now as in Figure 4.5.

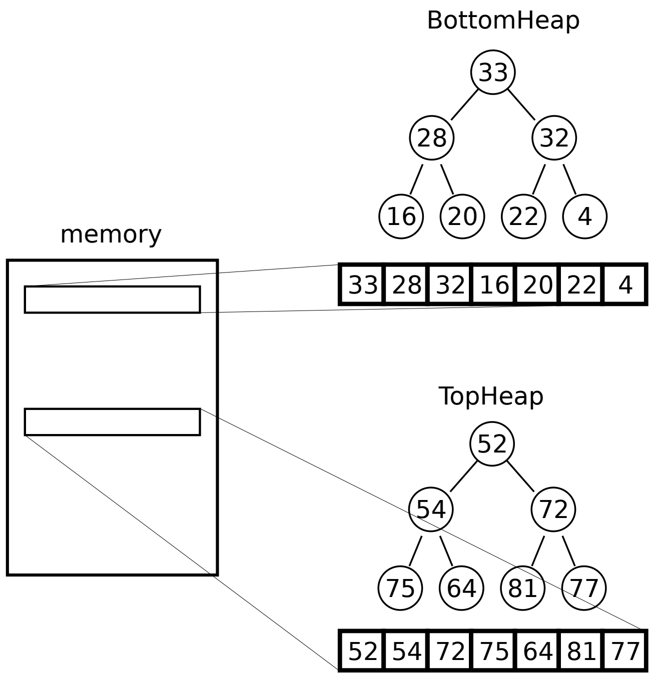


Figure 4.2: Two heaps and their naïve array representation.

If the TopHeap grows to occupy the whole memory while the BottomHeap is kept at size 0, the algorithm is equivalent to RS.

The general idea is similar to having two separate RS algorithms working together, one gives larger records as output while the other one gives a decreasing output.

4.2 Input buffer and Heuristics

The performance of 2WRS is very dependent on how the memory is partitioned between the TopHeap and the BottomHeap. For the algorithm to work in a balanced way, meaning that each heap is used in a way that maximizes the run length, it is crucial to choose a good first output record. This is because the first record given as output marks a division: records greater than it belong to the TopHeap, and smaller records belong to the BottomHeap.

If we do not know anything about the input data, the first record may go at any one of the two heaps, and performance may be suboptimal. This is why an input buffer is introduced. A part of the memory is filled with input records in the same order they are read, and this buffer is used to ample the input data and infer its distribution.

The input buffer works as a FIFO queue of records. When a record read from the input can be inserted into both heaps, the contents of the input buffer are used to choose which heap will store the record. We have proposed and analyzed several different heuristics for this:

- *Random*: chooses one heap at random, where the record will be stored.

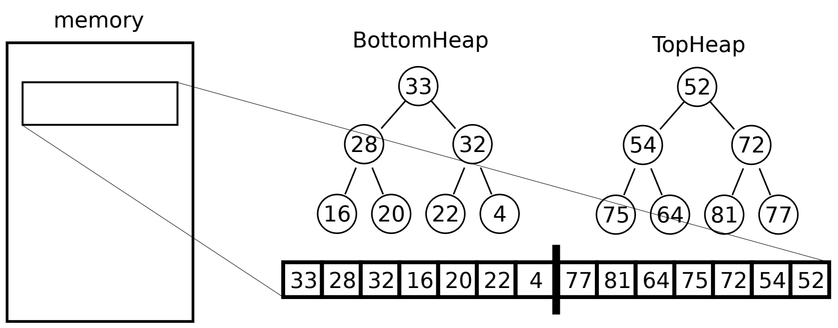


Figure 4.3: The two heaps of Figure 4.2 and their single-array representation, as used by 2WRS.

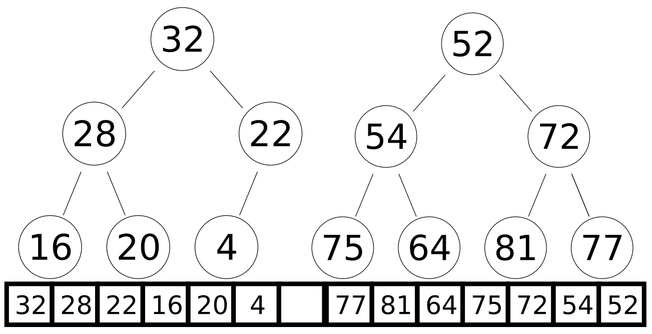


Figure 4.4: The two heaps of Figure 4.3 and their single-array representation, after removing the top record of the first heap.

- *Alternate*: records are assigned to the BottomHeap and TopHeap alternatively. If a record is inserted in the BottomHeap, the next one will be inserted in the TopHeap, and vice versa.
- *Mean*: compares the record to be inserted with the mean of the records in the input buffer. If the mean is smaller, the record is stored in the TopHeap, else it is stored in the BottomHeap.
- *Median*: behaves similarly to the *Mean*, but the next record is compared with the median of the records.
- *Useful*: keeps a dynamic track of the usefulness of each heap. The usefulness of a heap is measured as the number of records output by that heap divided by its size. New records are stored in the most useful heap.
- *Balancing*: records are stored in the smallest of the two heaps. When a run starts, if one heap has more records than the other one, records are popped from the large heap and inserted into the small one until both heaps contain the same number of records.

It should be noted that not all of the heuristics make use of the input buffer, namely, *Random*, *Alternating*, *Useful* and *Balancing*. In Section 5.2 we compare the different heuristics and conclude which are preferable for different input distributions.

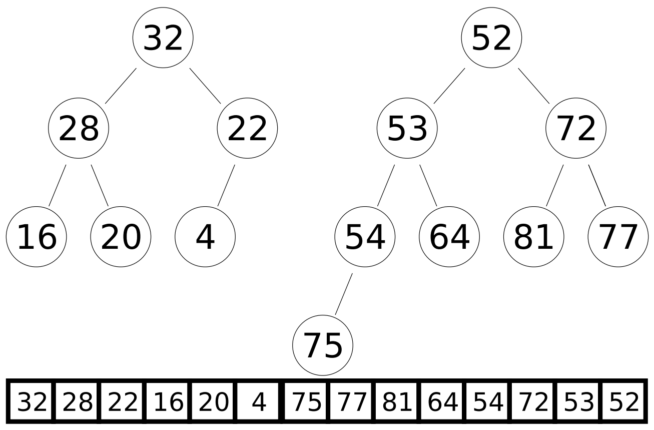


Figure 4.5: 53 has been inserted into the TopHeap.

Output Heuristics

When the algorithm outputs a records of the current run, it is taken from one of the two heaps. If the two heaps are not empty, the record to be output can be chosen from either heap. In order to select from which heap the record will be popped, a second heuristic, called *output heuristic* is used. As with the input heuristic, we study several alternatives:

- *Random*: chooses one heap at random. The record is popped from that heap and written to the corresponding output stream.
- *Alternate*: chooses the heaps in an alternating fashion. First, a record is popped from the BottomHeap, and the next one from the TopHeap.
- *Useful*: using the same usefulness measure as the *Useful* input heuristic, the record is popped from the most useful heap.
- *Balancing*: keeps both buffers the same size. Thus, the record is popped from the larger heap.
- *Min distance*: the first output is chosen at random, and for the following ones, the closer record in absolute value to the first output is selected.

Note that with six input and five output heuristics, there is a total of 30 different combinations of heuristics to choose from, which are analyzed in Chapter 5.

4.3 Victim buffer

The *victim buffer* is a pool of memory dedicated to store and sort records that can not be stored into either heap as records belonging to the current run. At the start of the algorithm, the output of each heap is stored in separate files, corresponding to Streams 1 and 4 in Figure 4.1. Since Stream 1 is increasing and Stream 4 is decreasing, these streams will not have any record that lies in the range comprised between the greatest record in Stream 1 and the smallest record in Stream 4. With the introduction of the victim buffer, if a new record read from the input is in this valid range, then it will be stored in the victim buffer instead of being marked as belonging to the next run and stored in one of the heaps, as would normally happen.

When the victim buffer is full, the records are sorted. The largest gap between consecutive records is selected as the new valid range for the victim buffer. Records smaller than the new valid range are stored in Stream 3, which is increasing, and records greater than it in Stream 2, which is decreasing. The next time the victim buffer is full, the same operations are performed, but the records can be appended to the two streams already created, instead of having to create new streams. This is because the records in the victim buffer are in the range between the largest record of Stream 3 and the smallest record of Stream 2.

Additionally, the victim buffer performs a second task in the beginning of the algorithm. Since the victim buffer is empty in the beginning, 2WRS writes temporarily the output records to the victim buffer instead of to Streams 1 and 4. When the victim buffer is full, 2WRS finds the largest gap in the buffer, which will be the valid range of the victim buffer, and flushes the records to Streams 1 and 4. This allows to choose a valid range different from the first gap between the top records of both heaps. Choosing a larger valid range makes the victim buffer more useful, since the probability of a record belonging to an interval is larger when the interval is larger.

4.4 Pseudocode

Algorithm 2 shows the pseudocode for the main loop of 2WRS. The algorithm first fills both heaps with records obtained from the input. This is done by method *doubleHeap.fill*. When a record can be stored in both heaps, this function uses the input heuristic to decide which heap is used to store the record.

The main loop is executed while the two heaps are not empty. First, a record is released to make room for a new one. This is done by method *victimBuffer.output*, which pops the top record from either the TopHeap or BottomHeap using the output heuristic if necessary.

Next, a record is obtained from the input buffer. Method *victimBuffer.fit* checks whether the current record is inside the gap currently processed by the victim buffer and, if so, stores it and returns *true*. Otherwise, it does nothing and returns *false*. Note that at the beginning of each run, while the victim buffer has not been completely filled, this function always returns *false*. While this method returns *true*, new records are read from the input buffer. When the record read can not be put in the victim buffer, the method returns *false* and the record is inserted into one heap by method *doubleHeap.insert*. This method inserts the record into one of the heaps, using the first heuristic when necessary.

Finally, method *doubleHeap.nextRun* returns *true* when the top record of both heaps belong to the next run, meaning that the current run reached an end, since all records in memory also belong to the next run. In this case the records stored in the victim buffer are written to disk and the next run starts, with an empty victim buffer.

The main loop of this algorithm is very similar to the main loop of replacement selection, shown in Section 3.4. The only difference is that 2WRS checks

Algorithm 2 2WRS(inputBuffer, heapSize, victimBufferSize)

Require: An input buffer *inputBuffer*, the maximum combined size of the heaps *heapSize* and the victim buffer size *victimBufferSize*.
Ensure: The generation of several ordered runs.

```
1: let current a pair of integers containing a value for a record and the run to
   which it belongs.
2: let doubleHeap a pair of heaps, a max heap and a min heap, of maximum
   total size heapSize.
3: let victimBuffer a victim buffer of size victimBufferSize.
4: let currentRun an integer.
5: doubleHeap.fill(inputBuffer);
6: currentRun = 0;
7: while doubleHeap.size() > 0 do
8:     output(doubleHeap);
9:     if inputBuffer.read(current.value) then
10:        current.run = currentRun;
11:        while victimBuffer.fit(current.value) do
12:            inputBuffer.read(current.value);
13:        end while
14:        doubleHeap.insert(current);
15:    end if
16:    if doubleHeap.nextRun(currentRun) then
17:        currentRun = 1 + currentRun;
18:        victimBuffer.flush();
19:    end if
20: end while
```

whether the current record can be placed in the victim buffer and keeps reading new records while this is the case.

4.5 Example

In order to show the general behavior of the algorithm, a small example is presented here. In this example the system has enough memory to store 22 records, which are assigned to the 2WRS data structures as follows: 4 to the input buffer, 4 to the victim buffer, and 14 to the heaps. We apply the *Mean* input heuristic and the *Random* output heuristic. The input data is

$$\{40, 50, 39, 51, 38, 52, 37, 53, 36, 54, 35, 55,$$
$$34, 56, 33, 57, 32, 58, 44, 39, 59, 60, 61, \ldots\}$$

This input data alternates records sorted with records sorted in reverse order, leaving a gap for the victim buffer to use.

In the beginning, 2WRS fills the input buffer. Then, the buffer contains the four first records of the input, $\{40, 50, 39, 51\}$. Now, the algorithm reads the first record from the input buffer, which is 40. This record can go into either heap, because both are empty. Since 40 is not greater than the mean of the input buffer contents (45), it is pushed into the BottomHeap. A new record,

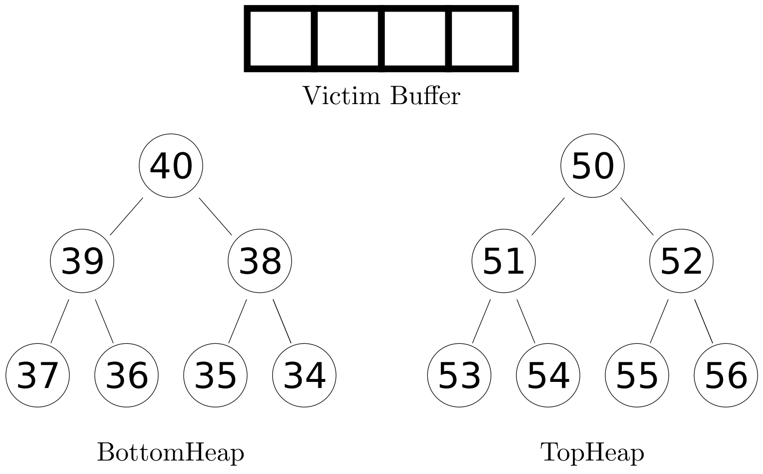


Figure 4.6: The two heaps after they are filled. The victim buffer and the output are still empty.

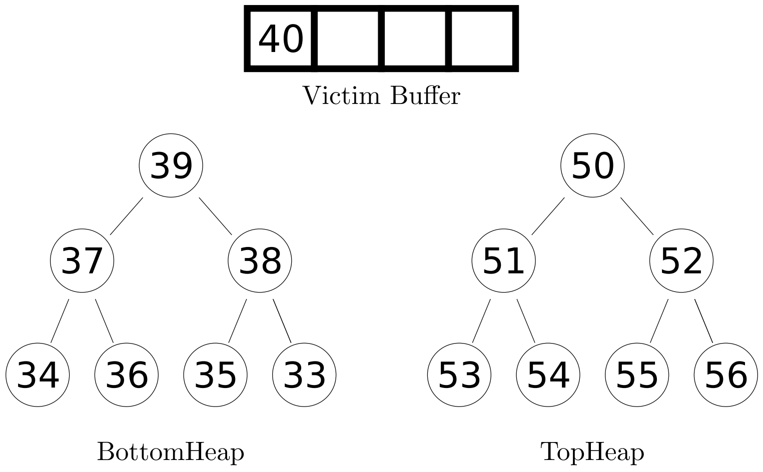


Figure 4.7: The two heaps after the first record is put in the victim buffer.

38, is inserted into the input buffer FIFO and we pop the head, 50. This time, 50 greater than the mean (44.5), so it is inserted into the TopHeap. The next record is 39, which goes to the BottomHeap because it is smaller than the top record of the BottomHeap, 40. This process is iterated until the heaps are full. In Figure 4.6 we show the content of the heaps at this point.

When the two heaps are full, the *Random* output heuristic selects one of the heaps, the BottomHeap, and its top record is put in the victim buffer. Then a new record is inserted into one of the heaps. The top of the BottomHeap is 40, which it is inserted into the victim buffer. The next record is 33, which goes to the BottomHeap. The content of the heaps is shown in Figure 4.7. This process is repeated until the victim buffer is full.

Once the victim buffer is full, it is sorted, as shown in Figure 4.8. The largest gap in the victim buffer is that between 40 and 50, so records that are smaller than (or equal to) 40, i.e. 39 and 40, are written to the output stream 3. The

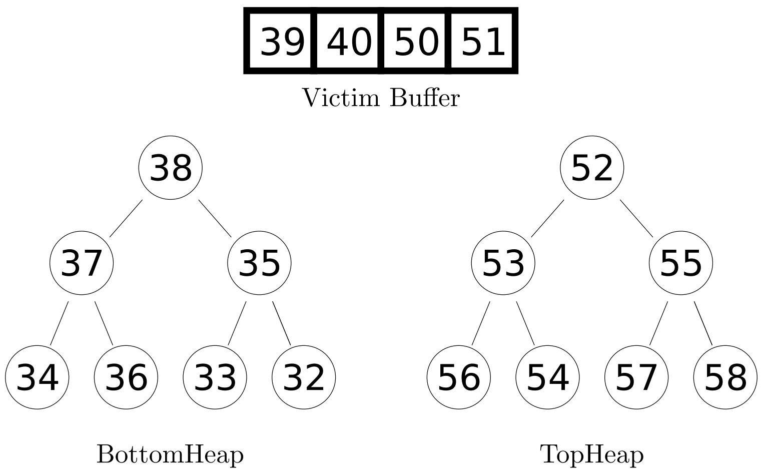


Figure 4.8: The two heaps after the victim buffer is full. The four streams are still empty.

rest of the records (50 and 51) are written to the output stream 2. The victim buffer is now empty, and accepts records between 40 and 50.

Next, the *Random* output heuristic selects one of the heaps, the BottomHeap, and its top record, 38, is written to Stream 4. The next record in the input buffer is 44. It can not be inserted into either heap, but since it is between 40 and 50, it is inserted into the victim buffer. The next record is 39. It is not possible to use it in the current run, but since it is not between 40 and 50, it can not be put in the victim buffer. In this case, it is marked as belonging to the next run and inserted into the BottomHeap, because 39 is not larger than the mean of the contents of the input buffer, which are now $\{39, 59, 60, 61\}$. The content of the heaps now is shown in Figure 4.9. Given that 39 belongs to the next run, it is considered smaller than any other record in the BottomHeap for the current run.

The algorithm continues until all record in both heaps are marked. Then, the current run is finished and the next one is started.

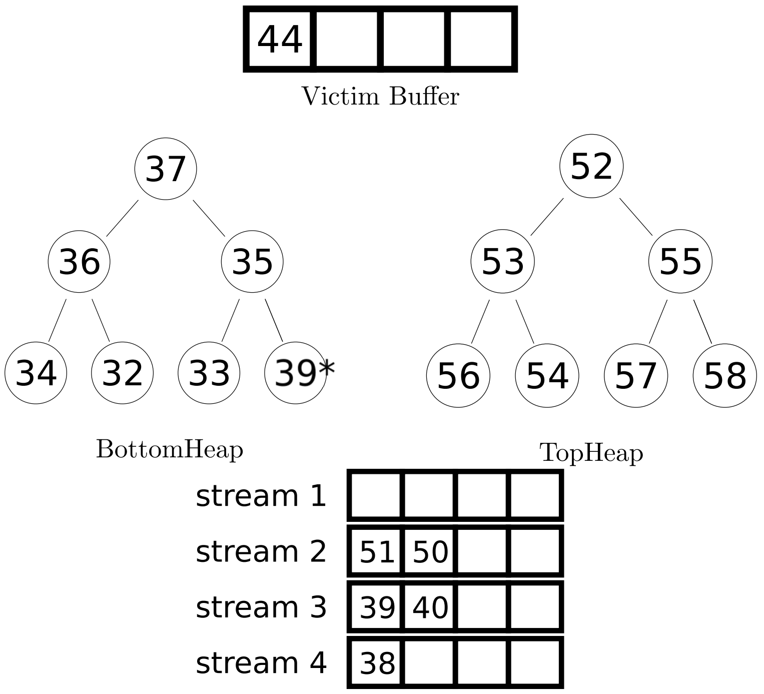


Figure 4.9: The two heaps at the end of this example and the contents of the victim buffer and the four streams. The asterisk next to 39 marks it as belonging to the next run and, as such, it is considered to be smaller than any record in the current run.

Chapter 5

Analysis

In this section we first prove some relevant properties of RS and 2WRS formally, for different characteristic data distributions. This way, we show that 2WRS is able to sort incoming data, generating runs which are, at least, as long as those generated by RS for sorted data (Theorem 2) and longer than RS for other data inputs (Theorems 5 and 6).

Later, in Section 5.2, we analyze the number of runs generated by RS and 2WRS. We study the different combinations of configurations for 2WRS with the aid of statistical models based on the analysis of variance (ANOVA). Since the quality of the models is high, we give recommendations on the best configurations based on these models. In Chapter 6, we pick this recommended configuration and we perform a timing benchmark.

5.1 Theoretical Analysis

Theorem 1. *For inputs already sorted in ascending order, RS generates one run containing all the input records.*

Proof. Since the input records are already sorted, each new record will be larger than all the values in the heap and, thus, it will be possible to insert it into the heap as belonging to the present run. No record will be marked as belonging to the next run. □

Theorem 2. *For inputs already sorted in ascending order, 2WRS generates one run containing all the input records.*

Proof. This proof is the same proof as for Theorem 1. All records obtained from the input are larger than those stored in memory. All the records are stored in the TopHeap, and all they belong to the same run. □

Theorem 3. *For inputs sorted in reverse order, RS generates runs with length equal to the size of the memory.*

Proof. Since the input records are sorted in reverse order, the next record obtained form the input is smaller than all the previous records. Thus, it is not possible to include the new record in the current run when the heap is full. So, the new record is marked as belonging to the next run. When the heap is full every new record belongs to the next run. Once the records belonging to the present run are released, a new run starts and the size of the run is equal to the available memory. □

Theorem 4. *For inputs sorted in reverse order, 2WRS generates one run containing all the input records.*

Proof. The records obtained from the input are smaller than all the records in memory. However, in contrast to RS, those records are inserted in the BottomHeap. Since all the records from the BottomHeap can be used in the current run, all the stream is released in a single run through the BottomHeap. □

Theorem 5. *For inputs consisting of alternating chunks of length k records sorted in ascending order and k records sorted in descending order repeatedly, RS generates runs with an average length around twice the size of the memory m ($m << k$).*

Proof. Let m be the size of the memory. Every chunk of k records sorted in ascending order is placed in the same run, as per Theorem 1.

When the algorithm starts reading records sorted in reverse order, only the first $m/2$ will be included in the current run. The rest of the records sorted in reverse order are put in runs of length m, as per Theorem 2. Therefore, the number of runs generated in a descending section is $\left\lfloor \frac{k-\frac{m}{2}}{m} \right\rfloor = \left\lfloor \frac{k}{m} - \frac{1}{2} \right\rfloor$.

The last m records of a chunk of k records sorted in reverse order ($(k - \frac{m}{2}) \mod m$) are placed in the same run as the following k records sorted in ascending order.

So every chunk of k records sorted in ascending order is included in a run together with $m/2$ records from the next chunk of k records sorted in descending order, plus the last records from the previous run $(k - \frac{m}{2}) \mod m$.

The average run length is then the total number of records divided by the number of generated runs,

$$\frac{2k}{1 + \left\lfloor \frac{k}{m} - \frac{1}{2} \right\rfloor}$$

The denominator of this formula can take the values $\left\lfloor \frac{k}{m} \right\rfloor$ and $\left\lfloor \frac{k}{m} + 1 \right\rfloor$. The formula achieves maximum value when the denominator is minimum. The maximum average run length is then

$$\frac{2k}{\left\lfloor \frac{k}{m} \right\rfloor} \approx 2m$$

□

Theorem 6. *For inputs consisting of alternating chunks of length k records sorted in the reverse order, two-way replacement selection generates runs with an average length equal to k (with an appropriate heuristic[1]).*

Proof. 2WRS behaves identically to RS for the chunks sorted sorted in ascending order, thanks to the TopHeap. For the chunks with records sorted in reverse order, 2WRS captures the trend with the BottomHeap, generating runs of k records, as well. Thus, the average run length is k. □

[1]An appropriate heuristic is one that uses the TopHeap for sorted inputs and the BottomHeap for reverse sorted inputs.

Theorem 7. *There is an input heuristics that makes 2WRS perform at least as good as RS.*

Proof. The heuristic consists on choosing always the TopHeap. □

5.2 Run length analysis

In this section, we design experiments to test the configuration parameters of 2WRS following the analysis of variance (ANOVA) techniques. Through the rest of this chapter we use the conventional statistics nomenclature, where categorical variables are named *factors* and the values they can take are named *levels.*The ANOVA detects which factors are more relevant and it is used to select the optimal configuration for a set of factors. For further details about ANOVA, see Appendix B.

In these experiments, the memory size allocated to the algorithm is fixed to 100K records and the input length is 100MB. Each record is formed by a 4B integer, which means that the input consists of 25 million records. The number of runs and the average run length verify the formula

$$\#\text{runs} \cdot \text{avg.runlength} = \text{memorysize}$$

Since the memory size is fixed to 100K records, it is equivalent to optimize the average run length and minimize the number of runs generated. We selected the number of runs as the response variable because the generated models are better than the ones generated with the average run length as the response variable.

The observations are obtained as a crossed factorial experiment with five factors:

- *Buffer setup*: Three levels are tested: only the input buffer is used, only the victim buffer, and both the input and victim buffers are used.
- *Size of buffer*: There are four levels: 0.02%, 0.2%, 2% and 20% of the available memory is dedicated to the buffers and the rest to the heaps. Note that in all the configurations, the total allocated memory (the addition of the heap and buffer sizes) for 2WRS is always constant.
- *Input heuristic*: We test six levels for the heuristic of the input buffer: *Random*, *Alternate*, *Mean*, *Median*, *Useful* and *Balancing.*
- *Output heuristic*: We test five levels for the output heuristic: *Random*, *Alternate*, *Useful*, *Balancing* and *Min distance.*
- *Data distribution*: We test six different data input distributions. (1) Sorted: The records are already sorted. (2) Reverse sorted: the inputs are sorted in reverse order. (3) Alternating: This dataset is a sequence of increasing intervals followed by decreasing intervals. The number of intervals is set to 50, with 25 increasing and 25 decreasing interleaved intervals. (4) Random: The records are generated following a uniformly random distribution. (5) Mixed: This dataset alternates one record from a sequence of increasing records, with another record of a sequence of decreasing records. (6) Mixed imbalanced: This dataset alternates one record from a sequence of increasing records, with three records of a sequence of decreasing records. We depict these datasets in Figure 5.1.

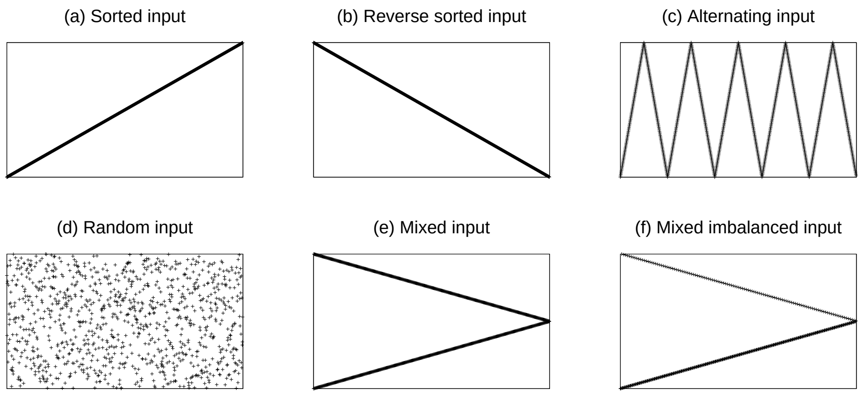


Figure 5.1: Samples of all data inputs used.

These data distributions are presented as basic distributions which can be combined to build more complicated distributions are combinations and concatenations of these basic distributions. For example, a table in a database with two columns, A and B, where the data in A is anticorrelated with B, and sorted by A, would result in a reverse sorted input (Figure 5.1 (b)) when sorting by B. Another example is to have a bidimensional attribute stored as two attributes, A and B. When sorting data by B, the input is a concatenation of sorted inputs (Figure 5.1 (a)).

In our experimental setup, there is a total of 2160 different configurations. In order for the results to have variance, each configuration is tested five times, using five different seeds for the random number generator. Thus, there is a total of 10800 executions of the 2WRS algorithm. If we execute 2WRS with the same input data five times, the result is the same, as it is a deterministic algorithm. Because of this, a uniformly distributed random value is added to each input record. These random values range from 1 to 1000 for a total range of values sorted from 1 to 10^9.

We executed the algorithm in a computer equipped with an Intel Core 2 Duo processor running at 2.40GHz. Each core has 4KB of L2 cache memory and the system has a total of 2GB of RAM. The hard disk is a SATA drive with a capacity of 60GB. The OS of the system is Debian GNU/Linux. A shell script executed each configuration sequentially and saved the results in a disk file, as described in Appendix A.2. The results were imported into the SPSS statistics software.

We observed that the hypothesis of homoscedasticity (homogeneity of variances) is not met. Figure 5.2 shows all the results for each input dataset. It is clear that it is not true that all levels have the same variance, so we decided to split the results in six groups, according to the input dataset used, and construct six different models using the remaining four factors instead of five. Table 5.1 summarizes the factors and their levels. We observed no outliers in any of the six groups. The significance level of all tests performed is 0.05. Each of the following subsections discusses the six models.

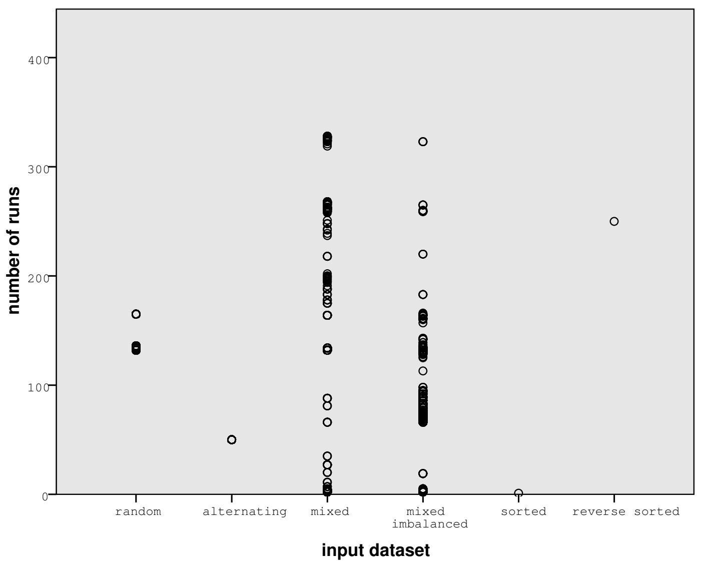


Figure 5.2: Number of runs generated as a function of the input dataset.

5.2.1 Sorted

In accordance to Theorems 1 and 2. When the input is already sorted, both RS and all configurations of 2WRS behave identically, generating a unique run containing all records. This is the optimal behavior of the algorithms. The model is $y = \mu = 1$, since the response variable is constant and independent of any factor. The residuals are all zero and the residual variance is also zero.

5.2.2 Reverse Sorted

When the input is sorted in reverse order, RS has the worst performance possible, generating runs of length equal to the available memory, as we proved in Theorem 3. However, as shown by Theorem 4, all configurations of 2WRS generate a unique run containing all records. The model for 2WRS is identical to the model for sorted input, $y = \mu = 1$. For RS, $y = \frac{100K}{1} = 10^5$, as Theorem 3 proves. There is no dependence on any factor and the residuals are all zero. With this input, 2WRS is a big improvement over RS in terms of average run length, because we jump from worst-case performance to optimal performance for any configuration.

5.2.3 Alternating

With the alternating input, as found in Theorem 6, all configurations of 2WRS build 50 runs with average length 5 times the memory size. As with the sorted and reverse sorted input datasets, the residual variance is zero and the model $y = \mu = 50$ accurately predicts the average run length with this input.

With this input dataset, RS generates runs of length 1.94 times the available memory, which is close to 2.0 as Theorem 5 states. This shows that when

Factor	Level	Description
α_i Buffer setup	$i = 0$	Only the input buffer is used
	$i = 1$	Both buffers are used
	$i = 2$	Only the victim buffer is used
β_j Size of buffers	$j = 0$	0.02% of memory is used for buffers
	$j = 1$	0.2% of memory is used for buffers
	$j = 2$	2% of memory is used for buffers
	$j = 3$	20% of memory is used for buffers
γ_k Input heuristic	$k = 0$	*Random* input heuristic is used
	$k = 1$	*Alternate* input heuristic is used
	$k = 2$	*Mean* input heuristic is used
	$k = 3$	*Median* input heuristic is used
	$k = 4$	*Useful* input heuristic is used
	$k = 5$	*Balancing* input heuristic is used
π_l Output heuristic	$l = 0$	*Random* output heuristic is used
	$l = 1$	*Alternate* output heuristic is used
	$l = 2$	*Useful* output heuristic is used
	$l = 3$	*Balancing* output heuristic is used
	$l = 4$	*Min distance* output heuristic is used

Table 5.1: Factors and their levels.

Factor	SS	D.F.	MSS	F	Sig.	Power
α_i	1.028	2	0.514	6.713	0.001	0.917
β_j	344166.646	3	114722.215	1498694.762	0.000	1.000
γ_k	3.396	5	0.679	8.873	0.000	1.000
π_l	30.491	4	7.623	99.582	0.000	1.000
$\mathcal{R}^2 = 1.0$ $\sigma = 0.277$ $CV = 0.2\%$						

Table 5.2: Summary of the model $y_{ijklm} = \mu + \alpha_i + \beta_j + \gamma_k + \pi_l + \epsilon_{ijklm}$ with random input.

datasets are ordered or partially ordered 2WRS is more effective than RS.

5.2.4 Random

Using a simple model with all the factors and no interactions, $y_{ijklm} = \mu + \alpha_i + \beta_j + \gamma_k + \pi_l + \epsilon_{ijklm}$, the analysis shows that all factors are significant. The result of the analysis of this model is summarized in Table 5.2. The second and third columns are the sums of squares and the number of degrees of freedom. The fourth column, MSS, is the mean of sums of squares, that corresponds to SS divided by the degrees of freedom. The fifth column is the F value necessary to test the significance of the factor. The significance value tells us if the factor is significant, if this values is less than 0.05 it means that the factor is statistically significant in the model. The last column is the power of the hypothesis test obtained.

In this case, the analysis tells us that the model is accurate, since it explains 99.9% of the variance in the response variable, and the coefficient of variability is much less than 5%. The most relevant factor is the size of the buffer, since it has an F value several orders of magnitude higher than the other factors. Because of this, we analyze a simpler model with only this factor, $y_{ij} = \mu + \beta_i + \epsilon_{ij}$.

Factor	SS	D.F.	MSS	F	Sig.	Power
β_j	344166.646	3	114722.215	1201032.325	0.000	1.000

$\mathcal{R}^2 = 1.0 \quad \sigma = 0.310 \quad CV = 0.2\%$

Table 5.3: Summary of the model $y_{ij} = \mu + \beta_i + \epsilon_{ij}$ with random input.

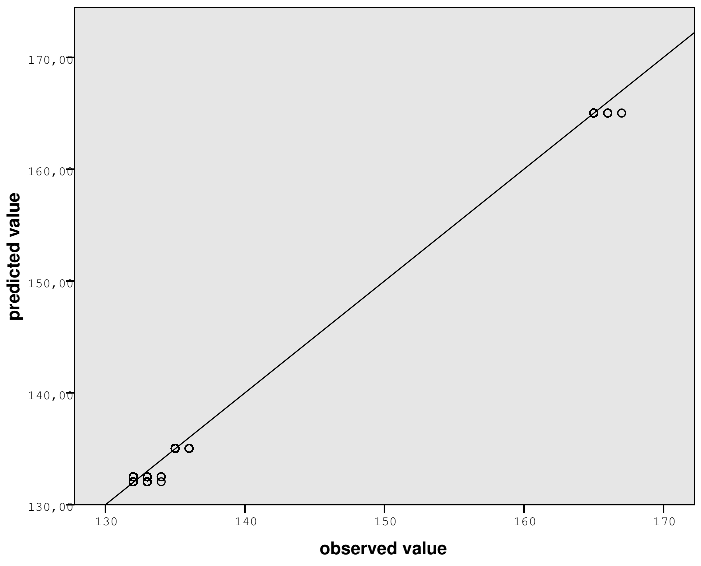


Figure 5.3: Predicted vs. observed values for the model $y_{ij} = \mu + \beta_i + \epsilon_{ij}$.

The ANOVA of this model is summarized in Table 5.3. The model is still very accurate, the values of $\mathcal{R}^2$ and CV have not changed. Therefore, the accepted model for random input is $y_{ij} = \mu + \beta_i + \epsilon_{ij}$.

Figure 5.3 depicts the predicted values versus the observed values of the response variable. There is an accurate match between the two variables, and as a result the value of $\mathcal{R}^2$ is very close to one. Thus, the model with only one factor, the size of the buffer, accurately predicts the average length of the runs generated by 2WRS with random input data.

Since the analysis found that the most important factor is β, the size of the buffers, Figure 5.4 represents the average run length as a function of the size of the buffers. It is observed that there is a linear correlation between the buffer size and the run length. If buffers are allocated, the memory dedicated to the heap diminishes by the percentage of memory dedicated to them. Thus, a configuration with 2% of the memory dedicated to buffers, reduces the run length by just 2%. Furthermore, in our experiments, we measured a very small difference in the length of the runs generated by the configurations with 0.2% and 2% allocated to the buffers, but larger between 2% and 20%.

With random inputs, replacement selection generates runs of length equal to twice the available memory, as it is proved in Section 3.5. The statistical model shows that 2WRS also generates runs with length relative to the available memory close to 2.0. This is to be expected since 2WRS is similar to RS, but

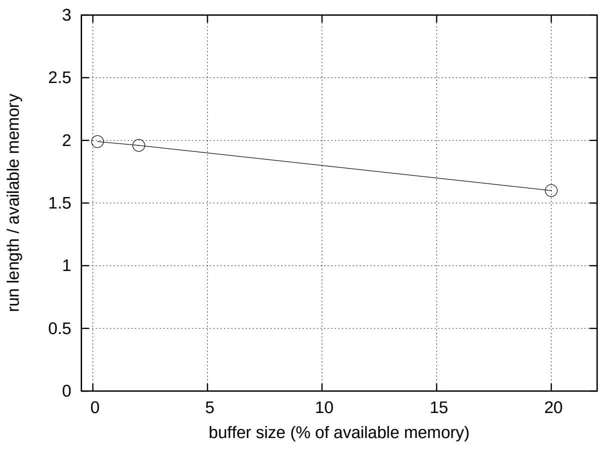


Figure 5.4: Length of runs relative to memory size as function of buffer size for random inputs.

working with two heaps instead of one. Also, since the behavior of random input can not be predicted, the input and victim buffers are of no use, as the statistical analysis shows.

As a conclusion, if the input follows a random uniform distribution and no input or victim buffers are used, 2WRS and RS perform identically in terms of number of runs generated.

5.2.5 Mixed balanced

The number of runs generated by RS with mixed datasets is approximately the same as with random data, around 50. With some of the configurations, 2WRS greatly outperforms RS, generating as few as two runs.

When using a model with no interactions, $y_{ijklm} = \mu + \alpha_i + \beta_j + \gamma_k + \pi_l + \epsilon_{ijklm}$, the value of $\mathcal{R}^2$ is 0.5, which is very low. The next model tested is one with all interactions of first order. The analysis shows an acceptable value of $\mathcal{R}^2$, 0.882, but a very high value of the coefficient CV, greater than 20%. Further analysis reveals that the variance of the response variable is not equal for each level of the factor α, the buffer configuration. Specifically, if the victim buffer is not used, the behavior of the algorithm is very different and exhibits, a larger variance, the runs generated have a length smaller than two times the available memory and a high number of runs are generated. This is shown in Figure 5.5.

In order to obtain a better model, the results corresponding to configurations not using the victim buffer are removed. This does not affect finding an optimal configuration because the configurations removed have bad performance, generating runs with an average length below two times the size of the available memory. After removing the corresponding results, a model with all factors and their first order interactions is analyzed. One of the factors, α, and its interactions is much less significant than the others, having a value of F at least two orders of magnitude smaller than the other factors, as shown in Table 5.4. Also, one of the interactions is not statistically significant. In order to simplify the model, this factor, α, and all of its interactions are removed from the model.

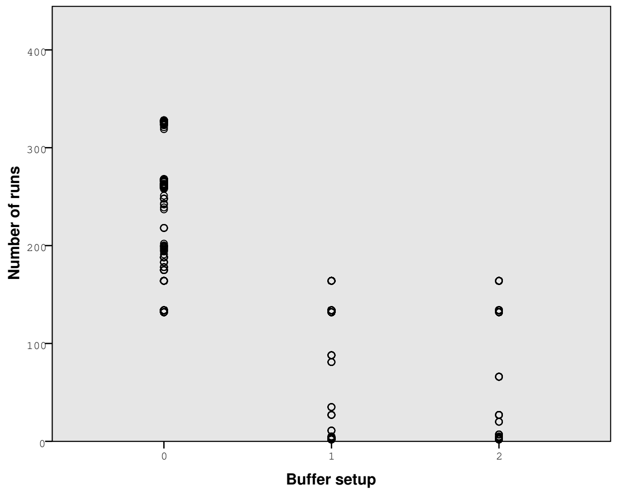


Figure 5.5: Number of runs generated as function of buffer setup (factor α) for mixed balanced input.

Factor	**F**	**Sig.**	**Power**
α_i (Buffer setup)	5.751	0.017	0.669
β_j (Size of buffers)	253.852	0.000	1.000
γ_k (Input heuristic)	10506.618	0.000	1.000
π_l (Output heuristic)	6499.640	0.000	1.000
$(\alpha\beta)_{ij}$	4.081	0.007	0.847
$(\alpha\gamma)_{ik}$	1.931	0.087	0.657
$(\alpha\pi)_{il}$	7.902	0.000	0.998
$(\beta\gamma)_{jk}$	65.683	0.000	1.000
$(\beta\pi)_{jl}$	79.974	0.000	1.000
$(\gamma\pi)_{kl}$	1158.751	0.000	1.000

$\mathcal{R}^2 = 0.990 \quad \sigma = 7.16 \quad CV = 9.7\%$

Table 5.4: Summary of the model with all factors and first order interactions with mixed balanced input.

Factor	**SS**	**D.F.**	**MSS**	**F**	**Sig.**	**Power**
β_j	40051.463	3	13350.488	247.843	0.000	1.000
γ_k	2732713.714	5	546542.743	10146.196	0.000	1.000
π_l	1419558.687	4	354889.672	6588.287	0.000	1.000
$(\beta\gamma)_{jk}$	52148.003	15	3476.534	64.539	0.000	1.000
$(\beta\pi)_{jl}$	54937.100	12	4578.092	84.989	0.000	1.000
$(\gamma\pi)_{kl}$	1214436.523	20	60721.826	1127.260	0.000	1.000

$\mathcal{R}^2 = 0.989 \quad \sigma = 7.34 \quad CV = 9.8\%$

Table 5.5: Summary of the model with factors β, γ, π and their first order interactions with mixed balanced input.

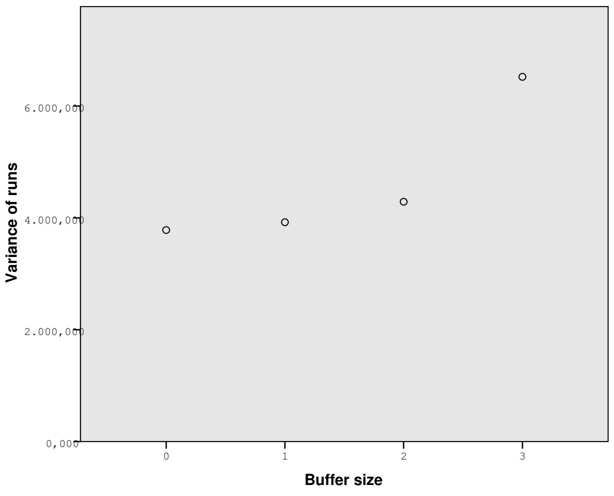


Figure 5.6: Variance of the number of runs generated as function of buffer size (factor β) for mixed balanced input.

Factor	SS	D.F.	MSS	F	Sig.	Power
β_j	7.089	3	2.363	192.961	0.000	1.000
γ_k	590.462	5	118.092	9643.895	0.000	1.000
π_l	306.726	4	76.682	6262.125	0.000	1.000
$(\beta\gamma)_{jk}$	9.645	15	0.643	52.509	0.000	1.000
$(\beta\pi)_{jl}$	11.501	12	0.958	78.271	0.000	1.000
$(\gamma\pi)_{kl}$	266.128	20	13.306	1086.655	0.000	1.000

$\mathcal{R}^2 = 0.988 \quad \sigma = 0.11 \quad CV = 0.15\%$

Table 5.6: Summary of the model with factors β, γ, π and their first order interactions with mixed balanced input and WLS weighting.

The analysis of the model that results from removing the buffer setup and its interactions is summarized in Table 5.5. The model can not be simplified, as all factors are significant and the removal of any of the interactions yields residue distributions far from normal. The coefficient of variance is a little higher than desired, 9.8%, so we try to improve this model.

In order to improve the model, we construct another model applying *Weighted Least Squares* (WLS) instead of the *Minimum Least Squares* (MLS) applied as the parameter estimation method. The WLS sets a weight for each configuration so that configurations with lower variance contribute more to the parameter estimation. Figure 5.6 shows that the variance for the number of runs is not constant for every level of the factor β, the size of the buffers. The WLS weights are defined as $w_i = {}^1\!/\!_{\sigma_i^2}$, where σ_i is the variance of the number of runs restricted to the i-th level of factor β, as shown in Figure 5.6.

The resulting model using WLS is summarized in Table 5.6. The value $\mathcal{R}^2$ is as good as with the model in Table 5.5, and the coefficient of variability is much better, 0.15%. Figure 5.7 shows that the standardized residuals follow a gaussian distribution. The most important factors of the model in Table 5.6 are

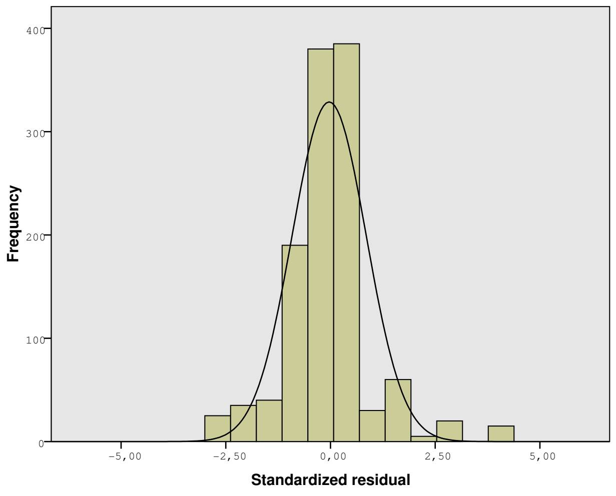


Figure 5.7: Histograms of the residuals of the model in Table 5.6.

	γ_0	$\boldsymbol{\gamma_1}$	$\boldsymbol{\gamma_2}$	$\boldsymbol{\gamma_3}$	γ_4	γ_5
γ_0	-	0.000	0.000	0.000	1.000	0.000
$\boldsymbol{\gamma_1}$	0.000	-	**0.693**	**0.693**	0.000	0.000
$\boldsymbol{\gamma_2}$	0.000	**0.693**	-	**1.000**	0.000	0.000
$\boldsymbol{\gamma_3}$	0.000	**0.693**	**1.000**	-	0.000	0.000
γ_4	1.000	0.000	0.000	0.000	-	0.000
γ_5	0.000	0.000	0.000	0.000	0.000	-

Table 5.7: Significance of the pairwise comparison of input heuristics. Significance level is 0.05.

the two heuristics, γ and π, and their interaction, $(\gamma\pi)_{kl}$.

We performed Tukey's tests for each of the factors in order to compare the mean of the result when grouped according to each level of the factors. The input heuristic has six different levels, and each level is compared to the other five levels, so there is a total of $\binom{6}{2} = 15$ pairwise comparisons. For each of pairs of input heuristics, the null hypothesis states that the mean of the response variable grouped by each heuristic is equal. The test rejects the null hypothesis for all comparisons except for the three comparisons between the *Alternate* (γ_1), *Mean* (γ_2) and *Median* (γ_3) heuristics, and the comparison between *Random* (γ_0) and *Useful* (γ_4). The significance obtained by the test for each pair being compared is shown in Table 5.7. The best performing levels are marked in boldface. The mean of the number of runs generated by each one of the three heuristics *Alternate*, *Mean* and *Median* is less than the same measure for any other heuristic, and the difference of the means is statistically significant. Also, these heuristics achieve the minimum number of generated runs, and according to Table 5.7 γ_1, γ_2 and γ_3 are not statistically different. Thus, optimal configurations for the mixed input dataset use one of these three input heuristics.

Similarly, for the output heuristics, the Tukey test rejects the null hypothesis

	$\boldsymbol{\pi_0}$	π_1	π_2	$\boldsymbol{\pi_3}$	π_4
$\boldsymbol{\pi_0}$	-	0.000	0.000	**0.761**	0.000
π_1	0.000	-	0.000	0.000	0.000
π_2	0.000	0.000	-	0.000	0.000
$\boldsymbol{\pi_3}$	**0.761**	0.000	0.000	-	0.000
π_4	0.000	0.000	0.000	0.000	-

Table 5.8: Significance of the pairwise comparison of output heuristics. Significance level is 0.05.

	$\boldsymbol{(\gamma\pi)_{10}}$	$\boldsymbol{(\gamma\pi)_{13}}$	$\boldsymbol{(\gamma\pi)_{20}}$	$\boldsymbol{(\gamma\pi)_{23}}$	$\boldsymbol{(\gamma\pi)_{30}}$	$\boldsymbol{(\gamma\pi)_{33}}$
$\boldsymbol{(\gamma\pi)_{10}}$	-	**0.792**	**0.442**	**0.321**	**0.442**	**0.321**
$\boldsymbol{(\gamma\pi)_{13}}$	**0.792**	-	**0.615**	**0.464**	**0.615**	**0.464**
$\boldsymbol{(\gamma\pi)_{20}}$	**0.442**	**0.615**	-	**0.820**	**1.000**	**0.821**
$\boldsymbol{(\gamma\pi)_{23}}$	**0.321**	**0.464**	**0.820**	-	**0.821**	**1.000**
$\boldsymbol{(\gamma\pi)_{30}}$	**0.442**	**0.615**	**1.000**	**0.821**	-	**0.820**
$\boldsymbol{(\gamma\pi)_{33}}$	**0.321**	**0.464**	**0.821**	**1.000**	**0.820**	-

Table 5.9: Significance of the pairwise comparison of the interaction between input and output heuristics using only the best input and output heuristics. Significance level is 0.05.

for all pairs of output heuristics except for the *Random* (π_0) and *Balancing* (π_3) heuristics. The significance obtained by the test for each pair is summarized in Table 5.8. Again, these two heuristics have a mean number of runs less than the other heuristics and achieve the minimum number of runs generated for this input dataset.

In order to find the best combination of heuristics, we analyze the interaction $\gamma\pi$, which corresponds to the interaction of the input and output heuristics. Table 5.9 shows the significance obtained by the test comparing each pair of levels, only for the input and output heuristics found to be the best. All values are greater than the significance level, 0.05, which means that the test fails to reject the null hypothesis, which states that the mean of the number of runs grouped by each level is equal. Thus, there is no statistical evidence that one of these levels is better among them, so any of these combinations of heuristics can be considered optimal. Figure 5.8 depicts the mean number of runs generated by 2WRS for each pair of input and output heuristics. In this figure we see how, when using π_1, π_2, π_3 or π_5 as the input heuristics, the choice of a specific output heuristic can significantly improve the performance of the algorithm in terms of number of runs generated.

The minimum number of runs generated with this input dataset is two, which corresponds to an average run length of 125 times the size of the available memory. There are several optimal configurations that generate only two runs when this input dataset is used. These configurations use the *Mean*, *Median* and *Balancing* input heuristics, the *Random* or *Balancing* output heuristics and more than 0.2% of memory for buffers.

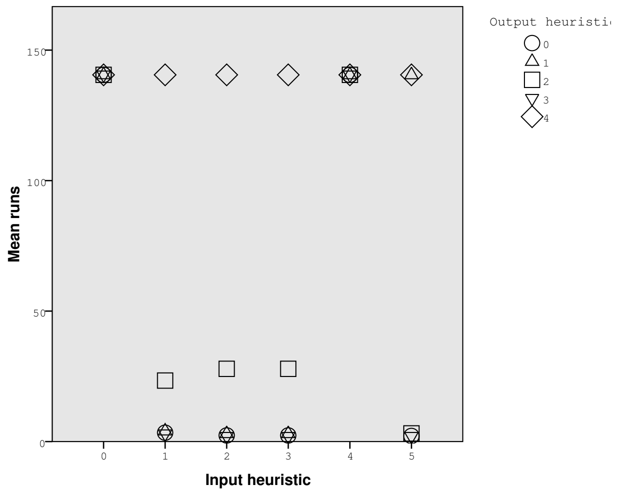


Figure 5.8: Mean of the number of runs generated for each pair of input and output heuristics with mixed input.

Factor	SS	D.F.	MSS	F	Sig.	Power
α_i	2393189.684	2	196594.842	1842,204	0.000	1.000
β_j	100285.913	3	33428.638	313,245	0.000	1.000
γ_k	207097,324	5	41419.465	388.124	0.000	1.000
π_l	370667.639	4	92666.910	868.341	0.000	1.000
$(\alpha\gamma)_{ik}$	786379.369	10	78637.937	736.881	0.000	1.000
$(\alpha\pi)_{il}$	341176.404	8	42647.051	399.627	0.000	1.000
$(\gamma\pi)_{kl}$	381931.848	20	19096.592	178.946	0.000	1.000
$(\alpha\gamma\pi)_{ikl}$	682352.809	40	17058.820	159.851	0.000	1.000

$\mathcal{R}^2 = 0.947 \quad \sigma = 10.33 \quad CV = 12.36\%$

Table 5.10: Summary of the model with all factors and interactions of first and second order of factors α, γ and π and mixed imbalanced input.

5.2.6 Mixed imbalanced

With the mixed imbalanced input, the number of runs generated strongly depends on the levels chosen for each factor of the configuration of 2WRS, as it happens with the mixed balanced input dataset.

With this input dataset, we found that it is necessary to have second order interactions in order to have reasonably good models. A model with all factors and interactions of first and second order showed that interactions that involve the factor β are much less significant, so these interactions are removed in order to have a simpler model. Table 5.10 summarizes this simpler model. This model has a high coefficient of variability, 12.36%.

As with the mixed balanced input, we construct another model applying WLS instead of MLS in oder to improve the model. Figure 5.9 shows that the variance for the number of runs is not constant for every level of the factor β, the size of the buffers. As with mixed balanced input, we define σ_i as the variance of the number of runs restricted to the i-th level of factor β, as shown

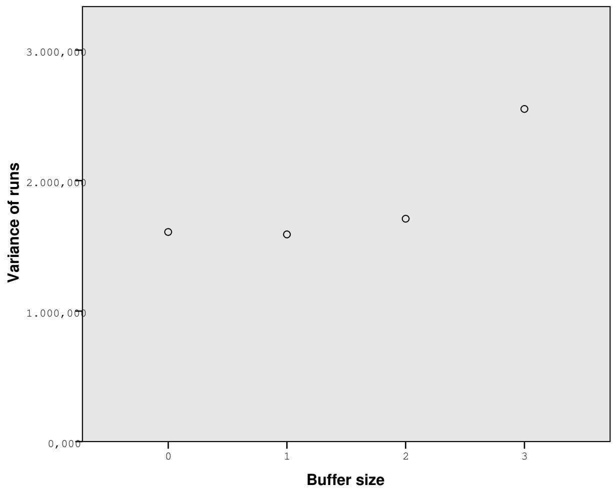


Figure 5.9: Variance of the number of runs generated as function of buffer size (factor β) for mixed imbalanced input.

Factor	SS	D.F.	MSS	F	Sig.	Power
α_i	208.897	2	104.448	1778.920	0.000	1.000
β_j	43.597	3	14.532	247.507	0.000	1.000
γ_k	113.381	5	22.676	386.211	0.000	1.000
π_l	204.340	4	51.085	870.058	0.000	1.000
$(\alpha\gamma)_{ik}$	417.793	10	41.779	711.568	0.000	1.000
$(\alpha\pi)_{il}$	180.412	8	22.552	384.088	0.000	1.000
$(\gamma\pi)_{kl}$	210.125	20	10.506	178.938	0.000	1.000
$(\alpha\gamma\pi)_{ikl}$	360.825	40	9.021	153.635	0.000	1.000

$\mathcal{R}^2 = 0.946 \quad \sigma = 0.24 \quad CV = 0.29\%$

Table 5.11: Summary of the model with all factors and interactions of first and second order of factors α, γ and π and mixed imbalanced input using WLS.

in Figure 5.9.

Table 5.11 summarizes the model using WLS. The value of $\mathcal{R}^2$ is as good as with the model in Table 5.10, but the coefficient of variability is much better, 0.29%. Figure 5.10 shows the histogram of the residuals of this model. There is a bar centered at nine. This bar corresponds to two specific configurations with a high error, meaning that the model is not able to accurately predict the result for them. These two configurations use 0.02% of the memory for buffers. Having very small buffers is similar to having no buffers, but sometimes the algorithm is capable of profiting from the existence of the buffers, albeit very small. Thus, similar configurations having small buffers may perform very differently. We cannot draw any conclusions from the model about these two configurations, so they are not considered when finding optimal configurations. This is not a problem because the performance of these configurations is bad.

The most important factor is α, the buffer setup. The best average of the number of runs generated corresponds to the level where both buffers are used.

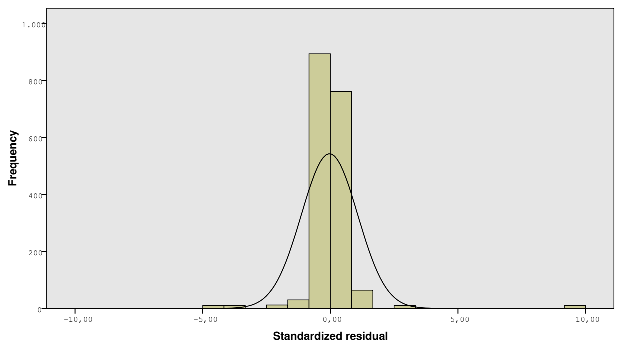


Figure 5.10: Histograms of the residuals of the model in Table 5.11.

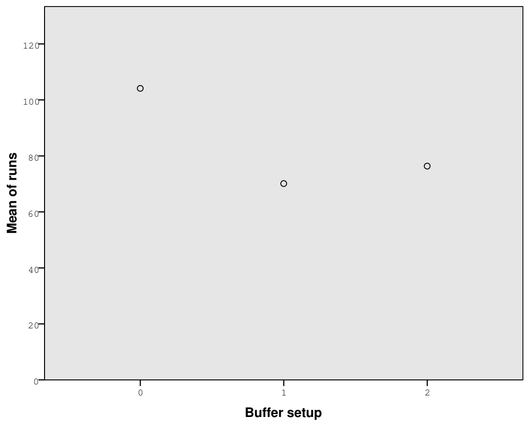


Figure 5.11: Mean of the number of runs generated as a function of buffer setup for mixed imbalanced input.

The analysis shows that there is a statistically significant difference between these value and the values corresponding to using only the input or the victim buffer. These values are depicted in Figure 5.11.

For the size of the buffers, β_j, the best results correspond to using 0.2 and 2% of the memory for them, and this result is better than the one obtained if using 0.02 or 20% of the memory. However, the test does not reject the null hypothesis that the mean of the results obtained when using 0.2 and 2% of memory for buffers are equal.

The input heuristics that achieve the beast average number of runs are *Alternate* (γ_1) and *Useful* (γ_4). The tests confirm that these two levels are statistically different from all the others, but are not able to confirm that these two heuristics are different. However, when looking at the results for the interaction between α and γ, the buffer setup and the input heuristic used, the best configurations are the *Mean* and *Median* heuristics when using both buffers. The tests show that there is statistically significant evidence that these two combinations, $(\alpha\gamma)_{12}$ and $(\alpha\gamma)_{13}$, perform better than any other combination for this interaction. Again, the tests do not reject the null hypothesis that the

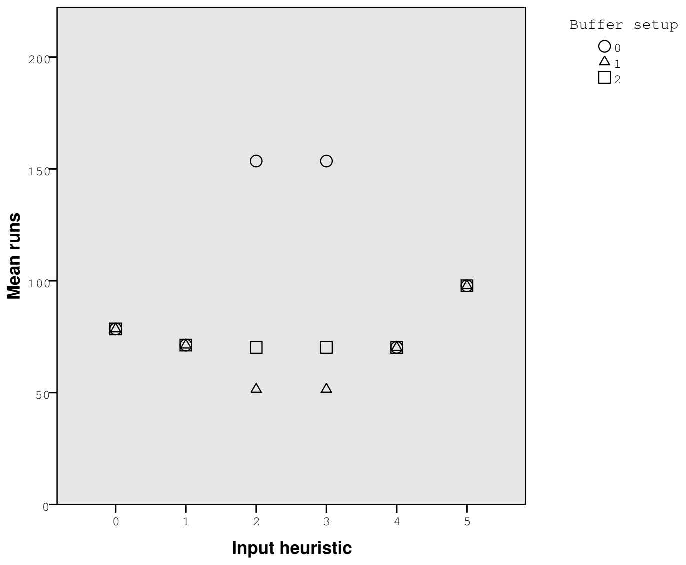


Figure 5.12: Mean of the number of runs generated as a function of input heuristic for each buffer setup for mixed imbalanced input.

	$(\alpha\gamma\pi)_{120}$	$(\alpha\gamma\pi)_{121}$	$(\alpha\gamma\pi)_{130}$	$(\alpha\gamma\pi)_{131}$
$(\alpha\gamma\pi)_{120}$	-	**0.937**	**1.000**	**0.937**
$(\alpha\gamma\pi)_{121}$	**0.937**	-	**0.937**	**1.000**
$(\alpha\gamma\pi)_{130}$	**1.000**	**0.937**	-	**0.937**
$(\alpha\gamma\pi)_{131}$	**0.937**	**1.000**	**0.937**	-

Table 5.12: Significance of the pairwise comparison of the best levels of the interaction between buffer setup, input and output heuristics. Significance level is 0.05.

average number of runs generated by these two combinations is equal. The fact that this interaction is significant is interpreted means that some heuristics take advantage of the presence of the buffers. Figure 5.12 represents the mean of the number of runs generated as a function of input heuristic for each buffer setup. In the figure we see that input heuristics π_0, π_1, π_4 and π_5 have the same average performance for all buffer setups. However, The performance of input heuristics π_2 and π_3 is very different for each buffer setup. If only the input buffer is used, the performance is the worst, but if both buffers are present, the performance of these heuristics increases significantly. This means that these two heuristics profit from the presence of both buffers and increase the performance of 2WRS.

The best output heuristics are *Random* (π_0), *Alternate* (π_1) and *Min distance* (γ_4). The test does not reject the null hypothesis that the performance of each of these three heuristics is equal when compared pairwise, but they are better than any of the other two output heuristics. When looking at the interaction between the buffer setup and the output heuristic, $\alpha\pi$, the best level is *Random* and *Alternate* with using both buffers, that is, $(\alpha\pi)_{01}$ and $(\alpha\pi)_{11}$. Again, the ANOVA test finds no significant difference between these two levels.

Analyzing the interaction between the input and output heuristics, $(\gamma\pi)_{kl}$, shows that there is a large number of levels that achieve the best average number of runs. Specifically using any input heuristic with the *Random*, *Alternate* or *Useful* output heuristics. The interaction of second order between the buffer setup, the input and the output heuristics, $(\alpha\gamma\pi)_{ikl}$, has four levels that achieve the minimum average of number of runs generated. These levels correspond to using both buffers, the *Mean* or *Median* input heuristic, and the *Random* or *Alternate* output heuristics.

The best configurations generate only two runs, which correspond to an average run length of 125 times the size of the memory. There are several of these optimal configurations. These configuration correspond to those found by the statistical analysis of the interaction between the buffer setup and the input and output heuristics, using the *Mean* or *Median* input heuristics, and the *Random* or *Alternate* output heuristics and using more than 0.2% of memory for buffers.

5.3 Conclusions

In order to find a configuration that works well with all inputs, we interpret the results obtained with the models for each input data distribution. The performance of 2WRS for sorted, reverse sorted and alternating datasets does not depend on the configuration. For random inputs, the performance only depends on the percentage of memory dedicated to buffers: the less, the better. The selected recommended configuration is:

- α: use both buffers. There are optimal configurations for each input dataset using both buffers.

- β: use 2% of the memory for buffers. The mixed balanced and mixed imbalanced datasets need this value to be high, but with random inputs, the average length of the run diminishes by the same percentage. 2% is chosen as a value which is not very large, so with random inputs the performance is only slightly worse, but large enough so that performance with mixed and mixed imbalanced is near optimal.

- γ: *Mean* is chosen as the input heuristic. Optimal configurations for sorted, reverse sorted, alternating and random input do not depend on the input heuristic. Optimal configurations for mixed and mixed imbalanced inputs use *Mean* or *Median.*The complexity of calculating the new mean is $O(1)$, and the complexity is $O(\log n)$ for the median. Thus, we prefer the *Mean* input heuristic.

- π: *Random* is selected as the output heuristic. Optimal configurations for sorted, reverse sorted, alternating and random input do not depend on the output heuristic. Optimal configurations for mixed balanced inputs use *Random* or *Balancing*, while for mixed imbalanced inputs the optimal configurations use *Random* or *Alternate. Random* is the only output heuristic common to optimal configurations of all input datasets. We note that no output heuristic is able to outperform *Random*, and we consider it to be the simplest output heuristic.

Input	RS	2WRS cfg 1	2WRS cfg 2	2WRS cfg 3
Sorted	inf	inf	inf	inf
Reverse sorted	1.0	inf	inf	inf
Alternating	1.94	50	50	50
Random	2.0	2.0	1.6	1.96
Mixed balanced	2.0	1.2	125	63
Mixed imbalanced	2.0	1.2	125	63

Table 5.13: Average run length relative to memory size. All three 2WRS configurations use the *Mean* input heuristic and the *Random* output heuristic. The first configuration sets the buffer size to 0.02% and allocates the input buffer. The second and third configurations use both buffers with size 20% and 2% respectively.

In Table 5.13, we have summarized the average run length for all the input sets. In this table, the run length of RS is compared to the best three parameterizations of 2WRS. Two of the 2WRS configurations minimize the number of runs generated for the mixed and random datasets, and the third configuration works reasonably well for all inputs. This last configuration is the one used in all the experiments in Chapter 6.

All in all, the run length analysis concludes that 2WRS creates runs of a length at least equal to RS or better. 2WRS is able to capture partially sorted data such as those in the alternating and mixed datasets, and is optimal with totally sorted data either increasingly or decreasingly.

Chapter 6

Time Performance Analysis

In this section, we test experimentally the performance of 2WRS with respect to RS. In the experiments, we measure the time to generate the runs, as well as the subsequent merge phase. The input datasets are generated following the random, mixed, alternating and decreasing patterns and the sorting time for each strategy is measured. With sorted inputs both algorithms are equivalent so no results are shown for that dataset.

6.1 Experimental setup

All the 2WRS configurations used in the experiments analyzed in this chapter use the *Mean* input heuristic, since this combination generates longer runs overall for all the inputs analyzed as we have seen in Chapter 5. According to the results presented in that chapter, a large buffer benefits mixed datasets, and a tiny buffer benefits random inputs. Therefore, the available memory for the buffers is set to an intermediate value, 2%.

We perform two experiments varying the input length and the memory allocated to the sorting algorithm. In the first experiment, the input is fixed to 1GB and the memory varies from 1k to 1M. Therefore, these tests aim at systems with large inputs (between 3 and 6 orders of magnitude larger) with respect to the memory available. In the second experiment, the memory is fixed to 10k and the input varies from 100MB to 1GB. The fan-in is set to 10 in accordance to 6.1.1.

Setup: We executed the algorithms in a computer equipped with an Intel Core 2 Duo processor running at 2.40GHz. Each core has 4KB of L2 cache memory and the system has a total of 2GB of RAM. The hard disk is a SATA drive with a capacity of 60GB. The OS of the system is Debian GNU/Linux. Given that we want to limit the available memory dedicated to sorting, all files are opened using direct I/O, which bypasses the operating system cache.

6.1.1 Fan in Analysis

The merge phase is computed as a tree of run merges. Depending on the number of files merged simultaneously (i.e. the fan in), the performance of the algorithm varies. In this experiment, we measure the fan in that achieves the

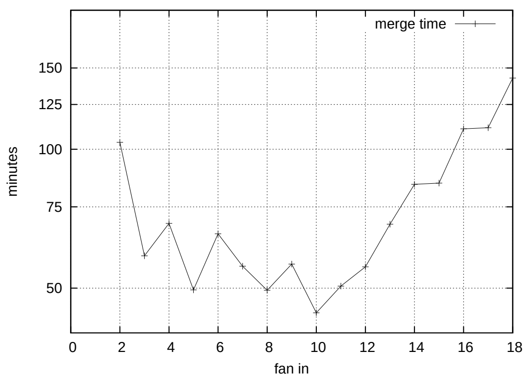


Figure 6.1: Merge time for different values of fan-in.

best performance in the computer used to run the experiments. We generate 400 files, each one of size 16MB, containing integers already sorted following a uniform distribution (i.e. 400 runs), and we merge them. This experiment is independent of the algorithm that generates the runs, and thus is valid for RS and 2WRS.

The fan in is a compromise between two characteristics: (a) the smaller the fan in, the more sequential is the access to the files from disk, but (b) the larger the fan in, the less merge operations are required to end the task.

We observe this tradeoff between the two benefits in Figure 6.1. If the fan in is too small, the algorithm takes more time because it must perform more merge steps. However, if the fan in is too large, the head of the disk performs more seeks and the bandwidth obtained from the disk is smaller. In this experiment, the minimum time is observed for a fan in 10, which means that in each merge step 10 different files are simultaneously merged. That is, during the merge phase, we perform a 10-way merge as explained in Chapter 2.

6.1.2 Storing decreasing records

Due to the way 2WRS works, it generates two streams of sorted records and two streams of reverse sorted records, as detailed in Chapter 4. The latter need to be stored in disk already sorted in order to allow the merge phase to read files sequentially. It is possible to store these stream in disk already sorted, so no modification to the merge phase is needed, as all files read by the merge phase contain records sorted in the order needed. Appendix A.2 details how these decreasing streams are stored in disk files.

6.2 Random

In Figure 6.2, we plot the performance with respect to the memory buffer size. The time spent generating the runs is detailed with empty circles and squares for RS and 2WRS, respectively, and the total time needed to sort the records is depicted with solid circles and squares. The same applies to the rest of the plots in this chapter. We observe that the total time needed by the two algorithms is

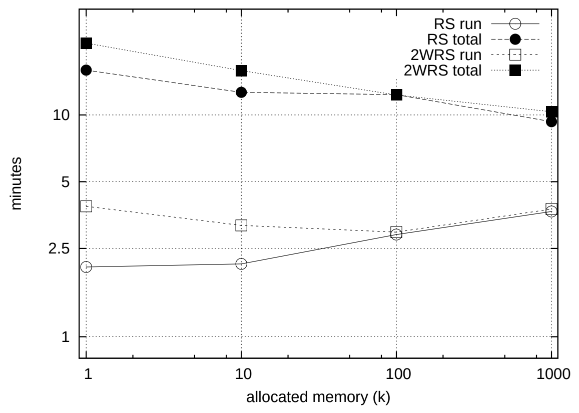


Figure 6.2: Run generation and total sorting times for random input as a function of available memory.

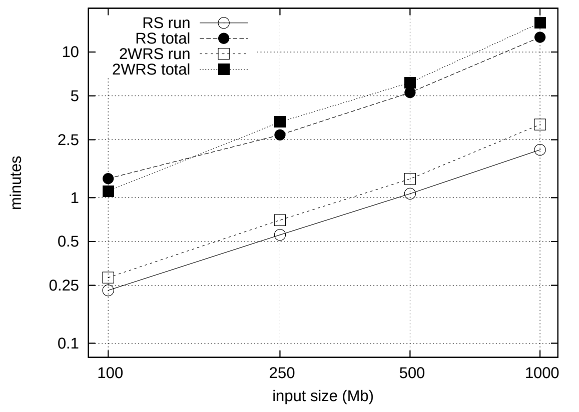


Figure 6.3: Run generation and total sorting times for random input as a function of input size.

very similar. This is due to the fact that it is not possible to predict the behavior of random input data. 2WRS has slightly worse performance during the run building phase for some configurations because the logic of 2WRS is slightly more complex than for RS, due to the two heaps and the multiple streams. However, the difference between both algorithms is tiny, and thus the use of either RS or 2WRS is equivalent for random inputs.

We plot the scalability of the algorithms with respect to the input length in Figure 6.3. Here, we observe a similar pattern to that described for the previous plot, where both algorithms consume a similar time. Furthermore, we observe that both algorithms scale identically when the input size grows.

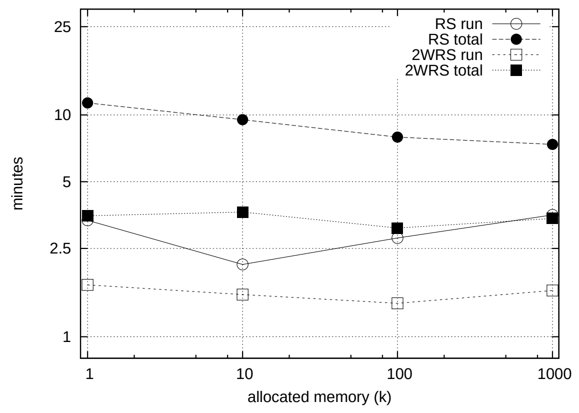


Figure 6.4: Run generation and total sorting times for mixed input as a function of available memory.

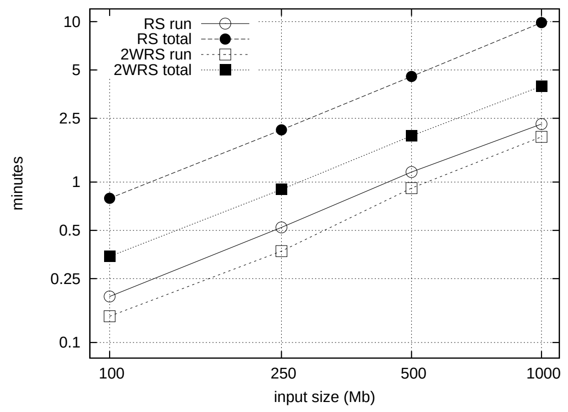


Figure 6.5: Run generation and total sorting times for mixed input as a function of input size.

6.3 Mixed

In the run length section, we found that 2WRS creates runs significantly larger than RS for mixed datasets. Figure 6.4 confirms it, because independently of the memory size, 2WRS is approximately three times faster than RS. This is because 2WRS generates less runs for the mixed dataset, and so the merge phase is much faster than with RS. We also see that, as the amount of allocated memory increases, both algorithms need less time to sort the data, since the runs generated are longer, and thus less merge phases are needed.

In Figure 6.5, we represent the scalability of both algorithms with the input. The advantage of 2WRS over RS for mixed data is sustained as the input data grows, and for all input sizes an approximate speedup of 3 is maintained. We note that for this dataset even the run generation of 2WRS is faster. This is because the heaps are not used and most of the computational time is spent sorting the victim buffer. Since the victim buffer uses a standard library sort,

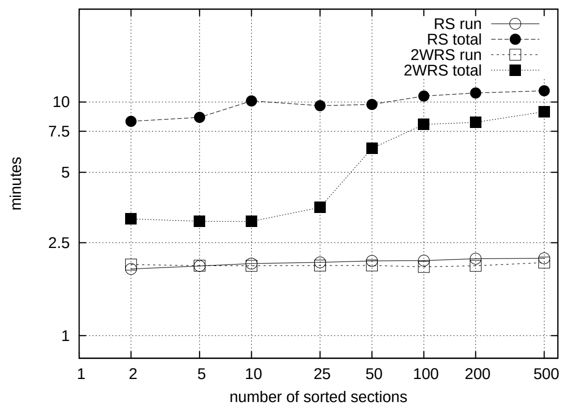


Figure 6.6: Run generation and total sorting times for alternating input as a function of the number of sorted and reverse sorted sections.

which is optimized for efficient in memory sorting, it is faster than RS that applies a heapsort.

6.4 Alternating

The complexity to sort the alternating dataset is dependent on the number of increasing and decreasing intervals for a fixed input size. If there are very few intervals, the dataset is similar to the sorted dataset, but if there are many intervals then it becomes closer to the random dataset. In this experiment, we fix the memory allocated to the algorithms to 10k and the input size to 1GB, and we vary the number of increasing and decreasing sections. In Figure 6.6, we depict the sorting time for both algorithms.

For a small number of sorted sections, 2WRS performs much better than RS, achieving up to an approximate speedup of 3. We observe that although the run phase takes the same time for both algorithms, the merge phase is significantly shorter for 2WRS because of the fewer number of runs. 2WRS is able to include the sections sorted in a single run in reverse order, whereas RS creates multiple runs for these sections. As the number of peaks increases, the sorted sections are shorter. Then, both algorithms asymptotically tend to need the same amount of time for sorting the data, although 2WRS still performs better. In the extreme case, if the number of peaks tends to infinite, the dataset would resemble a random input and both algorithms would spend the same execution time.

6.5 Reverse Sorted

In Figure 6.7, we plot the time spent by both algorithms to order reverse sorted data, as the size of the input grows. We observe that the run generation time is similar for both algorithms, but overall 2WRS gets a better performance than RS for all input sizes. This is because the run generation phase of 2WRS needs the same time as the same phase of RS but generates more runs, and the merge

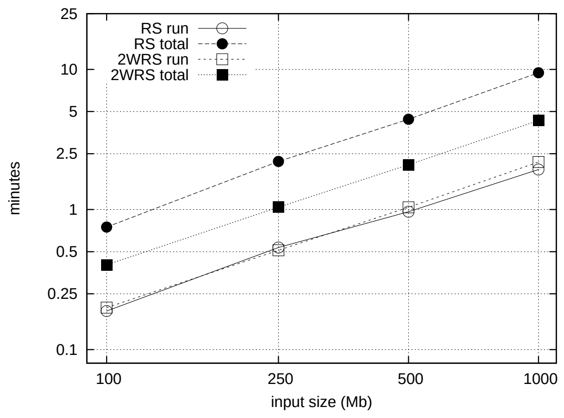


Figure 6.7: Run generation and total sorting times for reverse sorted input as a function of input size.

phase that is much faster when using 2WRS because it has to merge less runs than RS.

With this input data, replacement selection generates runs of length equal to the size of the memory, which is the worst possible performance for RS in terms of run length. 2WRS generates a unique run containing all records, as with sorted input. The result is that the merge phase is immediate with 2WRS, and much faster to finish.

The scalability of both algorithms is similar, showing parallel trends, that indicate a constant speedup, which is in this case 2.5.

6.6 Conclusions

In this chapter we found that the optimal fan-in of the merge phase in the machine used to execute the algorithms has a value of 10, independent of the algorithm used in the run generation phase. The performance of each value of the fan-in is depicted in Figure 6.1.

From the experiments performed we conclude that 2WRS works as well as RS with random input, as seen in Figures 6.2 and 6.3. With mixed input, Figures 6.4 and 6.5 show that 2WRS is about three times faster than RS. With alternating input, the performance of 2WRS depends on the number of alternating sections, as Figure 6.6 shows. When this number is low, 2WRS is up to three times faster than RS, and the performance decreases as the number of alternating sections increases, approaching the performance of RS. When the input is sorted in reverse order, we found that 2WRS is 2.5 times faster than RS, as depicted in Figure 6.7.

Chapter 7

Summary, Conclusions and Future Work

This document presents a new external sorting algorithm named two-way replacement selection. External sorting refers to the problem of sorting amounts of data large enough so that do not fit into the main memory of the computing device used to sort it. Therefore, external sorting algorithms are forced to used storage devices, usually hard disks, that have slower access times than internal memory.

In Chapter 3, we thoroughly present one of the most widely used external sorting algorithms, replacement selection. Replacement selection stores records being sorted in a heap. At each step the smallest record in the heap is output and another record is read from the input and inserted into the heap. If the new record is smaller than the last output record, it cannot be used as part of the current run and is marked as belonging to the next run. The generation of a run finishes when all records in memory are marked as belonging to the next run.

We proposed a mathematical model of RS in the form of a system of differential equations. With the aid of this model we proved that there is a stable situation when the input data follows a uniform random distribution, corresponding to a periodic solution of this system. The model confirms that when starting with the memory filled with uniformly distributed data, the solution approaches the previous stable solution as the time increases.

Replacement selection has an important drawback: when the input is sorted it generates only one run, but if the input is sorted in reverse order the performance is the worst, generating runs with length equal to the available memory. In order to solve this, we introduce two-way replacement selection in Chapter 4. This algorithm uses two heaps instead of one. This second heap solves the problem of replacement selection: 2WRS generates only one run when the input is sorted as well as when it is sorted in reverse order.

2WRS also has an input and a victim buffer, and an input and an output heuristic. The input buffer stored records from the input in the same order as they are read. The contents of this buffer are used by the input heuristic to decide which of the two heaps stores each input record when it can be stored in both. The output heuristic choses at each step from which heap is the next output record removed. The victim buffer stores records that can not be stored by either heap as part of the current run but belong to a specific range of values which can be part of the current run.

In Chapter 5, our formal analysis proves that 2WRS is able to generate runs at least as large as the runs generated by RS. We also proved that with structured input distributions, 2WRS generates runs of a length much larger than that of the runs generated by RS.

We performed experiments with six different input distributions: sorted, reverse sorted, alternating, random, mixed balanced and mixed imbalanced. These basic input distributions are analyzed as the basic building blocks of more complicated real distributions. For example, a table in a database with two columns A and B which have anticorrelated distributions, results in a reverse sorted input when sorting both columns by B. Also, if two columns store the flat and door numbers of addresses sorted by flat number and then by door number, the input distribution when sorting all data by door number is a concatenation of sorted inputs.

Aside from the input data distribution, the 2WRS algorithm can be parametrized to adapt to different input distributions. The four parameters are the buffer setup, the percentage of memory used by the buffers and which input and output heuristics are used. We executed each configuration of 2WRS with each input data distribution and analyzed statistical models of 2WRS from the experiments. Statistical analysis allows studying heuristics that are difficult to analyze formally, and allows analyzing configurations of the algorithm which are also difficult to formalize. With the aid of the statistical analysis we found optimal configurations for each input dataset. We also find a configuration that is not optimal but the performance is very good for all input datasets: using both buffers, *Mean* as the input heuristic, *Random* as the output heuristic, and 2% of the memory for buffers.

Regarding the individual analysis of the optimal configuration for each input distribution, we found that for the sorted, reverse sorted and alternating input datasets, the number of runs generated is independent of the 2WRS configuration. For the random input distribution, the number of runs generated depends only on the percentage of memory reserved for the buffers, the lower the better. We also found a configuration that is optimal for the mixed balanced and mixed imbalanced input datasets: using both buffers, *Mean* as the input heuristic, *Random* as the output heuristic, and 20% of the memory for buffers.

Chapter 6 uses the recommended configuration of 2WRS and compares the time performance of RS and 2WRS for different inputs, concluding that 2WRS performs similarly to RS with random input, and better with structured inputs, achieving speedups between 2.5 and 3.0.

We conclude that the addition of another heap along with the input and victim buffers consistently improves the performance of replacement selection, both in terms of number of runs generated and total time needed to sort data. This improvement is especially significant when the input is sorted in reverse order or when it is a combination of reverse sorted data with other distributions, such as the alternating and the mixed input datasets.

In Section 3.7 we present some modifications and improvements of replacement selection that have been proposed over the years. These modifications, such as the compressing of records or the new reading strategy can be readily applied to 2WRS in order to further improve its performance.

2WRS may be used in any application requiring external sorting, and particularly everywhere where RS is currently used. The most prominent application of external sorting is database management systems. Databases usually use several processes to perform concurrent operations. As a result, each of these processes operate with a low amount of memory. Also, several operations that are usually performed in databases require sorting of data. For example, the removal of duplicate entries, joining two or more tables, or intersections of tables. With our proposals, these operations can be performed in less than a third of the time that replacement selection needs.

7.1 Future work

There are two paths which can be explored to improve two-way replacement selection, one being theoretical and the other one experimental.

It would be interesting to obtain analytical solutions to the system of differential equations presented in Section 3.6 for other input distributions, and to find general analytic solutions, if possible.

Mathematical models are interesting because they are a tool used to find and prove properties of the algorithm. For example, they can be used to test the suitability of 2WRS for a task that knows the input distributions. In our case, there is the possibility to prove of the results without finding specific solutions to the system of differential equations. For example, finding bounds to the length of the runs for certain families of distributions. The run length is the path integral of $m(x, t)$ along the path defined by $(p(t), t)$. If this value is common to several solutions, it may not be necessary to find $m(x, t)$ or $p(t)$ explicitly for all initial solutions in order to compute the value of this integral.

Experimentally, it is interesting to find new heuristics that can improve performance of 2WRS, both with general and with specific input distributions. If the input distribution is unknown, an adaptive heuristic that tries to determine the distribution that the input data is following and adapts the decisions accordingly may improve significantly the average run length. Also, this heuristic may adjust itself according to the changes detected on the input data distribution. We note that this heuristic could take advantage of the model that we present in this project in order to switch dynamically its behavior.

From a practical perspective, the results derived from the work can be readily applied to autonomic computing procedures. Autonomic computing refers to those systems that are able to monitor the state of the system and react to the changes in order to improve the performance. This type of systems are used by databases that use query optimizers to find the best execution plan for a query. In our case, the query optimizer knows the distribution of the data at certain intermediate nodes of the execution tree, and can fix the parameters of 2WRS to minimize the execution time of the sorting processes involving those nodes.

Bibliography

[1] B. Bollobás. *Modern graph theory.* Springer Verlag, 1998.

[2] RL Gilstad. Polyphase merge sorting: an advanced technique. *AFIPS Joint Computer Conferences*, pages 143–148, 1960.

[3] M. Goetz. Internal and tape sorting using the replacement-selection technique. *Communications of the ACM*, 6(5):201–206, 1963.

[4] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3), 2006.

[5] CAR Hoare. Quicksort. *The Computer Journal*, 5(1):10, 1962.

[6] D. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, 2nd edition, 1998.

[7] I. Koltsidas, H. Müller, and S.D. Viglas. Sorting hierarchical data in external memory for archiving. *Proceedings of the VLDB Endowment archive*, 1(1):1205–1216, 2008.

[8] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[9] P. Larson and G. Graefe. Memory management during Run Generation in external Sorting. In *SIGMOD*, pages 472–483. ACM, 1998.

[10] P.A. Larson. External sorting: run formation revisited. *IEEE TKDE*, 15(4):961–972, 2003.

[11] D. Mongomery. *Design and Analysis of Experiments.* Wiley, 5th edition edition, 2000.

[12] T.K. Moon. *Error correction coding: mathematical methods and algorithms.* Wiley-Blackwell, 2005.

[13] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27(3):195–215, 1989.

[14] J.S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.

[15] S.B. Wicker and V.K. Bhargava. *Reed-Solomon codes and their applications.* IEEE, 1999.

[16] J. Yiannis and J. Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, 2007.

[17] W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, pages 376–385, 1997.

[18] L. Zheng and P. Larson. Speeding up external mergesort. *IEEE TKDE*, 8(2):322, 1996.

Appendix A

Files storing decreasing records

As seen in Chapter 4, 2WRS stores streams of records in decreasing order. That means that if records need to be sorted in the usual order, from smallest to greatest, two of the streams generated by 2WRS will have records ordered from greatest to smallest. For performance reasons, reading backwards the stream is not convenient because the hardware of disks is optimized to read data forward. On the other hand, the impact of writing backwards is less severe because the operating system uses the disk cache and does not need to flush synchronizedly the write operations.

In order to allow the merge phase to read all streams forward, streams with records in decreasing order are written to disk in a special way. In this appendix, we describe this strategy along with the internal workings of hard disks and files.

A.1 Hard disks

A *hard disk drive*, usually called *hard disk*, *hard drive* or *HDD*, is a non-volatile storage device. This means that data stored in a hard disk will not be erased if the power supply to the drive is interrupted. In a hard disk, data is digitally encoded in rotating rigid plates with magnetic surfaces. These plates rotate at a speed typically between 5400 and 15000 rpm. The data stored in a hard disk is encoded with an error correcting code, usually a Reed-Solomon code [15] or a low-density parity-check code (LDPC) [12], depending on the manufacturer.

Each of the platters of a hard disk is separated in concentric circular strips, called *tracks*. Each track is divided in smaller parts called *sectors*. This organization of platters is shown in Figure A.1. Each sector of a hard disk can store the same number of bits, most commonly 512. A sector is the smallest storage unit in a hard disk. This means that when reading (or writing) data from (to) a hard disk, the data read (or written) must be a multiple of the sector size.

In order to provide access to the hard disk, operating systems provide *file systems*. A file system is a set of data structures and an API[1] for applications

[1]Application Programming Interface, an interface implemented by software to be able to interact with hard disks

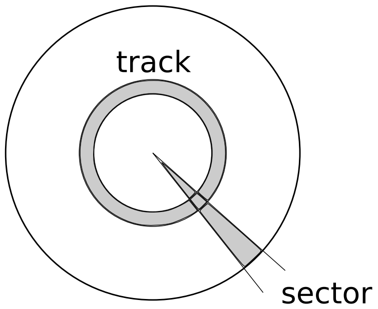


Figure A.1: Organization of a hard disk platter.

to store, organize, manipulate and retrieve data in a storage device. A *page* is the minimum amount of data that the file system can read or write. The size of a page depends on the file system being used, and when using hard disks, it is an integer multiple of the sector size. The default file system in Linux-based operating system, like the Debian GNU/Linux distribution used in our experiments, is the third extended file system, or *ext3*. This file system defines the default page size to be 4 kbytes.

When the hard disk need to read a page, it does so following three steps:

1. The read/write head of the disk is moved to the corresponding track.
2. The head waits for the sector to pass below it.
3. The head reads the sector.

If the operating system wants to read several contiguous sectors, Steps 1 and 2 are performed only once, because sequential sector can be read as they pass below the head. However, if the pages need to be read in reverse order, the head need to wait for the platter to do a full revolution until the previous sector passes under it. Thus, the throughput of the hard disk is maximized when performing sequential readings.

A.2 File format specification

In order to store the data supplied by a stream while reversing the order of the records, a file is created with a fixed size of k pages. The data records read from the stream are written to the file starting at the end, that is, the last position of the last page, and continuing backwards until the first page is reached. When the file is full, a new one is created in the same manner.

In order to minimize the number of input/output operations with the hard disk, a special output buffer is used with each file. This buffer has the same size as a disk page. When a record from a stream is output, it is written to this buffer instead immediately to the file. When the buffer is full, its contents are flushed to the corresponding page of the file. The memory space needed to store the buffer is taken from the memory dedicated to the 2WRS algorithm, but the performance of the algorithm is not affected because, typically, the amount of memory available to a sorting algorithm is several orders of magnitude larger than the size of a file page.

The first page of each file does not contain any data coming from the stream. Instead, it has a header with the following information:

- *Number of files*: the number of files that have been created to store all records from the stream.
- *Number of pages*: the number k of pages that each file has.
- *Starting page and position*: the page and the position within that page where the data begins. This should be page number two and first position for all files except possibly the last one.

Only a small percentage of the first page is used, but if the value of k is large, the amount of unused space is insignificant when compared to the total file size.

In order to know the order in which the files have been created, we use a naming system that assigns each file the same name followed by a different number. With this naming system, it is possible to open directly any of the files created to store the stream, and read the whole sequence in the desired order.

Once all the files have been created, the records can be read in non decreasing order starting at the last file created and ending at the first one.

The value of k should be chosen large enough so that not a large number of files are created, since closing and opening files adds an unnecessary time overhead. But if the value of k is chosen to be very large, it is possible that most runs fill only a small portion of the file, and lots of hard disk space will be wasted containing huge temporary files. Thus, when deciding the value of k, one must consider the number of records to be sorted, the expected run length, and the hard disk space available. This value can also be adapted during the execution of the 2WRS algorithm and it can change between runs. We use a value of $k = 1000$ to ensure that few files are created each run, which corresponds to 40MB files.

Appendix B

ANOVA

This Appendix briefly summarizes the ANOVA techniques used in Chapter 5. For a more extensive and comprehensive introduction to ANOVA, see [11].

The *Analysis of Variance* (ANOVA) is a collection of statistical models used to analyze the effect of different *factors* in the variance of a result or measure, called *response variable*. A *factor* is a categorical variable, that is, the values that can be assigned to it are labels. For example, a rock can be categorized as igneous, sedimentary or metamorphic. Each possible label is called *level* in statistics. The ANOVA technique used with categorical variables is the equivalent of linear regression used with continuous variables.

There are several types of ANOVA, depending on the experiment being analyzed. We use the *factorial ANOVA*. In a factorial ANOVA, there are two or more factors, each with its own levels. The experiment is repeated for all the combinations of levels multiple times, generating a fixed number of observations for each configuration. This type of experiment is known as *full crossed factorial experiment*.

ANOVA partitions the variance observed in the response variable into components due to the different factors, and estimates the contribution of each level of each factor, indicating which are significant. The generated model allows the experimenter to draw conclusions on the influence of each factor and its interactions in the system outcome.

B.1 Factors

In statistics, a *factor* is a categorical variable. A *level* is each of the values that a factor can take. In ANOVA, the factors are the explanatory variables. The term explanatory variable is preferred over independent variable to avoid confusion with statistically independent variables. Likewise, the response variable is not referred to as the dependent variable.

If the levels of a factor are fixed, they are considered constants, and the factor is called a *fixed effects factor*. In contrast, if the levels of a factor are selected at random, they are considered observations of a random variable and the factor is called a *random effects factor*. Random effects factors are used when the number of levels of a factors is too large to perform an experiment for each level. In this case, the levels are selected following a uniform random distribution.

A model where all factors are of the same type is called a *fixed effects model*, if the factors are fixed effects factors, or a *random effects model*, if the factors are random effects factors. A model that has both types of factors is called a *mixed effects model*. All the models discussed in Chapter 5 are fixed effects models, since we decide the levels of each factor. Fixed effects models are preferred because they are more robust than random effects models.

Given two factors, A and B, they can be *crossed* or *nested*. The two factors are crossed if every level of A occurs with every level of B in the experiment. The factor A is nested within B when the levels of A are different for each level of B. Given two crossed factors, it is possible to quantify the *interaction* between them. Two factors interact when the effect of at least one level of A on the response variable is dependent on at least a level of B. Note that it does not make sense to talk about interactions between nested factors.

Two of the most commonly used ANOVA models are the *one-way ANOVA*, where there is only one factor, and the *n-way ANOVA*, which is a generalization he one-way ANOVA for multiple factors.

B.2 One-way ANOVA

The One-way ANOVA is the simplest of the ANOVA models. It is used when there is only one factor, A with a different levels. The model is the following:

$$y_{ij} = \mu + \alpha_i + \epsilon_{ij}$$

where $0 \leq i < a$ and $0 \leq j < n$, n being the number of experiments performed for each level, y_{ij} is the response variable for the j-th experiment when the factor A takes the level α_j, μ is the expected value of y if no information about the level of the factor is known, α_i is the effect of the i-th level of the factor A, and ϵ_{ij} is the error, and corresponds to the effect of the variables not included in the model.

The parameters of the model are μ and α_i, $1 \leq i \leq a$. The objective is to estimate the parameters and find an estimator for y, $\hat{y}_i = \hat{\mu} + \hat{\alpha}_i$, and thus be able to predict the outcome of the system for any level of the factor.

The *raw residuals* are the estimated values of the errors,

$$\hat{\epsilon}_{ij} = y_{ij} - \hat{y}_i = y - (\hat{\mu} + \hat{\alpha}_i)$$

In order to be able to compare the residuals of different models, the *standardized residuals* are defined as the raw residuals divided by the square root of the estimated variance:

$$\hat{\epsilon}_{ij}^{std} = \frac{\epsilon_{ij}}{\sqrt{\hat{\sigma}^2}}$$

The variance of the raw residuals is estimated using the following estimator:

$$\hat{\sigma}^2 = \frac{\sum_{i=0}^{a-1} \sum_{j=0}^{n-1} (y_{ij} - \hat{y}_{ij})^2}{N - a}$$

The *coefficient of determination*, denoted by $\mathcal{R}^2$, is the proportion of variability in the explanatory variable that is accounted for in the model, and thus

$0 \leq \mathcal{R}^2 \leq 1$. Usually, a model is considered good when this coefficient is greater than 0.7, which means that more than 70% of the variability in the data is explained by the model. This coefficient is calculated using the following formula:

$$\mathcal{R}^2 = \frac{n \cdot \sum_{i=0}^{a-1} (\bar{y}_{i.} - \bar{y}_{..})^2}{\sum_{i=0}^{a-1} \sum_{j=0}^{n-1} (y_{ij} - \bar{y}_{i.})}$$

Another important coefficient of the model is the *coefficient of variation*, CV, that is a normalized measure of the dispersion of a random variable. It is defined as the ratio of the standard deviation to the mean, and is usually given as a percentage:

$$\mathrm{CV}(\%) = 100 \cdot \frac{\sigma}{\mu}$$

In the ANOVA model it is applied to the response variable, and calculated as

$$\mathrm{CV} = 100 \cdot \frac{\sqrt{\hat{\sigma}^2}}{\hat{\mu}}$$

A good value of this coefficient is considered to be below 5%.

The ANOVA model tests if a factor contributes to the response variable. If that is the case, the factor is said to be a *significant factor*, and otherwise, it is immediately removed from the model. A factor is significant if the parameter of any of its levels is proven to be statistically different from zero, that is, if $\alpha_i \neq 0$ for any i. This is verified using an F-test to decide whether $\hat{\alpha}_i$ is statistically different from zero.

Additionally, it is also possible to check if two levels of a significant factor are equivalent, which means that their contribution to the outcome is not statistically different. This can be achieved with a Student's t-test or a Tukey's test over the different levels of the factor.

B.3 Hypotheses

The ANOVA models assumes some hypotheses that must be met in order to build a reliable model. There are three such hypotheses:

- *Independence*: the residuals must be statistically independent. If the residuals are not statistically independent, it means that the model is biased for certain configurations of factors and thus it is not reliable. This hypothesis is verified plotting the standardized residuals as a function of the response variable or the predicted values. If any pattern is observed, it means that the residuals are not independent. Besides, the experiments are performed in a way that ensures that observations constitute a real sample of the response variable.

- *Normality*: the distribution of the residuals is normal. This is verified plotting an histogram of the residuals and checking if they follow a bell curve. On a fixed effects model, moderate departures from normality are of little concern.

- *Homoscedasticity (equality of variance)*: If the response variable is grouped according to the levels of one factor, the variances of each group are all equal. This is tested graphically plotting the response variable with respect to each one of the factors, which shows that the variance is equal for all levels of the factors.

Usually, the model is constructed and then the hypothesis are verified, validating the model.

These assumptions imply that the standardized residuals are independently, identically and normally distributed in a fixed effects model, and follow a $N(0,1)$ distribution.

B.4 Significance of a factor

A factor A is significant if at least one of its levels contributes to the response variable, i.e., if there is at least one i such that $\alpha_i \neq 0$. The significance of a factor is verified using a statistical hypothesis test, concretely an F-test. A hypothesis test has two hypotheses: the *null hypothesis*, H_0, and the *alternative hypothesis*, H_1, which is the negation of H_0 and the hypothesis one tries to prove. Thus, in the case of the significance of a factor, the hypotheses are:

$$H_0 : \forall i, \alpha_i = 0$$
$$H_1 : \exists i, \alpha_i \neq 0$$

Performing the hypothesis test consists on trying to reject the null hypothesis. The null hypothesis is rejected when there is enough statistical evidence that it is false. If a test fails to reject the null hypothesis it does not mean that there is statistical evidence of its validity. In order to perform the test, it is necessary to know the distribution that a particular statistic, called *test statistic* follows when H_0 is true.

The *error of the first kind* is defined as rejecting the null hypothesis when it is true. When performing a hypothesis test, it is asked that the probability of making an error of the first kind is less than a parameter, called *significance level*, and denoted usually by the letter α. This parameter verifies $P(\text{reject } H_0 | H_0 \text{ is valid}) \leq \alpha$.

An observation of the test statistic is obtained from the data obtained during the experiment. If the observation falls within the central part containing $(1-\alpha)\%$ of the distribution, the null hypothesis is accepted, else, it is rejected.

The *error of the second kind* is defined as accepting the null hypothesis when it is false. The *power* of a test is the probability of rejecting the null hypothesis when it is false, which is the same as the probability of not making an error of the second kind. Of all the tests available, the one with the highest power is preferred.

In order to decide if the null hypothesis is accepted or rejected, the variability due to the factor A is compared with the residual variability. If these variabilities are statistically different, we conclude that factor A is significant and the null hypothesis is rejected.

The total variability is defined as the sum of squares of deviations, usually named total sum of squares and denoted by SS_T. This sum of squares is decomposed in two components, the sum of squares associated to the factor A,

SS_A, and the sum of squares not explained by the model and associated to the error, SS_E, as follows:

$$SS_T = \sum_{i=0}^{a-1}\sum_{j=0}^{n-1}(y_{ij} - \bar{y}_{..})^2 = n \cdot \sum_{i=0}^{a-1}(\bar{y}_{i.} - \bar{y}_{..})^2 + \sum_{j=0}^{a-1}\sum_{i=0}^{n-1}(y_{ij} - \bar{y}_{i.})^2 = SS_a + SS_E$$

These two components can be compared if they are divided by their degrees of freedom, which are $a-1$ for SS_A and $N-a$ for SS_E. Thus, the *mean sums of squares* (MSS) are defined as

$$MSS_A = \frac{SS_A}{a-1}$$

$$MSS_E = \frac{SS_E}{N-a}$$

Due to the hypotheses of the ANOVA model, and in particular due to the normality hypothesis, MSS_E is a sum of squares of standardized independent normal variables, and thus MSS_E follows a χ^2 distribution with $N-a$ degrees of freedom, $MSS_E \sim \chi^2_{N-a}$. Moreover, if H_0 is true, MSS_A is also the sum of standardized independent normal variables, and thus MSS_A follows a χ^2 distribution with $a-1$ degrees of freedom, $MSS_A \sim \chi^2_{a-1}$. Also, the quotient of MSS_A and MSS_E follows a Fisher-Snedecor distribution,

$$F_0 = \frac{MSS_A}{MSS_E} \sim F_{a-1,N-a}$$

We define $F_{\alpha,a-1,N-a}$ as the value for with the distribution $F_{a-1,N-a}$ leaves $(1-\alpha)\,\%$ of the observations to the left, that is,

$$\int_0^{F_{\alpha,a-1,N-a}} F_{a-1,N-a}(k)dk = 1-\alpha$$

Thus, if the calculated value F_0 is greater than $F_{\alpha,a-1,N-a}$ the null hypothesis is rejected and the factor A is considered to be significant.

B.5 Parameter estimation

The one-way ANOVA mode, $y_{ij} = \mu + \alpha_i + \epsilon_{ij}$, has $a+1$ parameter to estimate: $\mu, \alpha_i, 0 \leq i < a$. The μ term is common for all configurations and corresponds to the expected outcome of the system if no information about the configuration is available. Thus, this term is the average of all observations $\mu = \bar{y}_{...}$.

The rest of the parameter can be estimated using one of various techniques. One of the most commonly used techniques is the *Minimum Least Squares* (MLS), which minimizes the sum of the squares of the error terms, which is

$$\sum_{i=0}^{a-1}\sum_{j=0}^{n-1}\epsilon_{ij}^2 = \sum_{i=0}^{a-1}\sum_{j=0}^{n-1}(y_{ij} - \mu - \alpha_i)^2$$

The solutions to this optimization problem correspond to the solutions of a system of linear equations. This system has one degree of freedom, so an additional equation, consistent with the system, is needed. Usually the constraint

$\sum \alpha_i = 0$ is added, which leads to the solution $\hat{\mu} = \bar{y}_{..}$, $\hat{\alpha}_i = \bar{y}_{i.} - \bar{y}_{..}$. This condition states that the deviations around the mean due to factor A add to zero. Another possible condition, used by the SPSS software is to set one of the parameters equal to zero. The specific choice of the additional constraint is not important, because the fundamental values analyzed by ANOVA are the differences among the levels of each factor, $\alpha_i - \alpha_j$. These values do not depend on the added condition.

If the response variable has different variance when restricted to each level of a factor, i.e., if the hypothesis of homoscedasticity does not hold for one factor, it is possible to use the ANOVA analysis, using the *Weighted Least Squares* (WLS) instead of the MLS. WLS weights each configuration so that configurations that exhibit greater variance contribute less to the estimation of the parameters, and vice versa. This is justified because observations with less variance are more accurate. The function to be minimized when using WLS is

$$\sum_{i=0}^{a-1}\sum_{j=0}^{n-1} \epsilon_{ij}^2 \cdot w_i = \sum_{i=0}^{a-1}\sum_{j=0}^{n-1} \left(y_{ij} - \mu - \alpha_i\right)^2 \cdot w_i$$

where w_i is defined as the inverse of the variance of the response variable for the i-th level of the factor A, $w_i = {}^1\!/\!_{\sigma_i^2}$. The method used to find the estimations of the parameters is analogous to the one used in the MLS case.

B.6 N-way ANOVA

The *n-way ANOVA* is a generalization of the one-way ANOVA model, used when there is more than one factor influencing the outcome of the experiment, as it is usually the case. The n-way ANOVA model allows to estimate the influence of each of the factors and the relation between them. This model also allows to find a configuration that leads to an optimal value of the response variable.

An example of a complete model with three factors, A, B and C, with a, b and c levels respectively, is:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \epsilon_{ijkl}$$

where $1 \le i \le a$, $a \le j \le b$, $1 \le k \le c$ and $1 \le l \le n$. In ANOVA models with more than one factor there is a new type of parameter, called *interaction.* This new parameter corresponds to the joint effect on the response variable of two or more factors. For example, $(\alpha\beta)_{ij}$ corresponds t the interaction between the i-th level of the factor A and the j-th level of the factor B. An interaction between two levels A and B indicates that the effects of the levels of A depend on the levels of B. Interactions with more than two factors are usually avoided in models because they are difficult to interpret.

The residuals in an n-way ANOVA model are defined analogously as they are defined in a one-way ANOVA. The coefficients of determination and variation are also calculated for these models. The tests that check for statistical significance of a factor are also generalized to work with n-way ANOVA models. Factors and interactions that are not statistically significant are always removed from the model. If the contribution to the response variable of a significant factor or interaction is small in comparison with other contributions, it is possible to remove the term with a small influence from the model, if the model still verifies the assumptions and scores a good enough value of $\mathcal{R}^2$. This is done because if two different models explain the data, the simplest one is preferred. This is known as principle of parsimony.

Appendix C

Conference paper

As a result of this work, a research article has been written and submitted to the *36th International Conference on Very Large Data Bases* (VLDB2010), and is pending approval at the time of writing. The VLBD conference is a top tier conference, equivalent in impact and prestige to reputable journals. The full article text is included in this appendix.

Two-way Replacement Selection

Xavier Martinez-Palau, David Dominguez-Sal, Josep Lluis Larriba-Pey
DAMA-UPC
{xmartine,ddomings,larri}@ac.upc.edu

ABSTRACT

The performance of external sorting is highly dependant on the length of the runs generated. One of the most commonly used run generation strategies is Replacement Selection (RS) because, on average, it generates runs that are twice the size of the memory available. However, the length of the runs generated by RS is downsized for data with certain characteristics, like inputs sorted inversely with respect to the desired output order.

The goal of this paper is to propose and analyze two-way replacement selection (2WRS), which is a generalization of RS obtained by implementing two heaps instead of the single heap implemented by RS. The appropriate management of these two heaps allows generating runs larger than the memory available in a stable way, i.e. independent from the characteristics of the datasets. Depending on the changing characteristics of the input dataset, 2WRS assigns a new data record to one or the other heap, and grows or shrinks each heap, accommodating to the growing or decreasing tendency of the dataset. On average, 2WRS creates runs of at least the length generated by RS, and longer for datasets that combine increasing and decreasing data subsets. We tested both algorithms on large datasets with different characteristics and 2WRS achieves speedups at least similar to RS, and over 2.5 when RS fails to generate large runs.

1. INTRODUCTION

Sorting is in the heart of many high performance processes [5]. Some of those processes require a sorted dataset as their final output (e.g. sort names alphabetically) or as a partial computational step (e.g. a sort merge-join). Since datasets are typically large, the selection of a good out of core sorting algorithm has an important impact on the performance of the final application. This motivates that many benchmarks emphasize a good performance of the out of core sorting operation. For example, in the 80's Anon et al. proposed a 100MB sort benchmark, which focused on the objective to sort the dataset in the minimum time possible[1]. Even though the computer growth has outdated this particular benchmark, the sorting operation has been popular as a benchmark along the years and it is still evaluated nowadays, for instance, sorting up to 100 TB of data or sorting the maximum number of records in one minute [9].

In the context of database management systems, the data to be sorted is often generated by other operators in a streamed fashion at irregular moments of time. This is the case for database workloads, where join operators keep generating tuples irregularly in time feeding sometimes a final sort [4].

In addition, database management systems (DBMSs) assign a memory quantum to each operation involved in a query, limiting the amount of global memory that a sort operation may use to process the whole dataset to be sorted. This raises two more issues: first, sorting becomes frequently out of core in DBMSs; and second, sorting must take advantage of the limited memory assigned by the DBMS. Out of core sorting implies that during the process, sorted runs are generated and stored in disk. It is not until all the runs are created, that they can be read again and merged to generate the final sorted dataset [5].

In summary, three features are desirable for a external sort operation in a DBMS: (a) it should be able to start sorting data before all the input is generated; (b) it should be efficient with already sorted data because a previous operation may have already sorted or partially sorted the incoming data; and (c) the memory consumption should be predictable.

Replacement Selection (RS) has played a prominent role in these complex situations because it fulfills most of the stated features. First, unlike other sorting methods, it is able to sort data in a streamed fashion, using one heap to perform the job. Second, it generates runs which double the size of the memory available for random data, and infinite runs for already sorted data. This reduces the number of runs, allowing the merge process to reduce its fan in, and the chances to perform multiple I/O passes during the merge phase. Finally, although it is not the fastest in-memory sorting strategy, it offers a good trade off for its value: it generates smaller I/O by creating larger runs at the cost of possibly more in-core computational effort, compared to other faster in-core methods[3].

Replacement Selection, however, has one important drawback: it is not able to generate runs larger than the memory available for inversely sorted input data. This creates unpredictable situations and leads to low performance in undesired cases. This paper solves this problem, proposing Two-way Replacement Selection, a general strategy that allows to obtain runs which are at least the size of those generated by RS and, in many cases, more than double the size of the memory available for sorting no matter the dataset, improving the good features mentioned above for RS.

Two-way replacement selection (2WRS) implements two heaps that adapt to the data characteristics, one intends to capture the growing values and the other one intends to capture the decreasing values. The strategy is to place each newly arrived record in the correct heap. But not only that, the heaps grow or shrink depending on the nature of the data. So, in case there are more growing than decreasing data, it grows the growing data heap, and shrinks the decreasing data heap, and conversely. We set two heuristics, random and the mean average of the input heap records, for the decision of which heap should store the new record, and grow or shrink.

We also study the effect of buffering before and after the heaps without changing the total size of the memory used for 2WRS. Our study shows that the use of buffers before the insertion of data in the heaps, and the use of buffers after a decision is taken to store a data record in disk, is beneficial performance-wise. Also, we show that 2WRS scales for growing datasets, and improves RS no matter the characteristics of the dataset. The reason for the general improvement of 2WRS is the larger runs generated, and the small complexity added compared to RS.

The paper is organized as follows. In Section 2 we explain Replacement Selection, in Section 3 we introduce Two-way Replacement Selection, in Section 4 we formally prove the performance of 2WRS, in Section 5 we analyze the run length obtained for different configurations of 2WRS, in Section 6 we analyze the time performance of RS and 2WRS with the best configuration obtained from the previous section and in Section 7 we give an overview of the methods used to improve external sorting and in particular the performance of RS. Section 8 ends the paper with some conclusions and future work.

2. REPLACEMENT SELECTION

Replacement Selection (RS) is an external sorting algorithm introduced by Goetz in [2]. The objective of RS is to sort a stream of records as they come (usually from secondary storage), producing another stream of released data records called "run", which is sorted. The algorithm allocates memory that works as an intermediate buffer and stores a window of streamed data. This buffer is managed as a heap: upon the arrival of a new record, RS releases the first record in the sorting order, and stores the new record in the heap in sorted order.

This strategy has an important drawback due to the limited memory in computers. For example, when sorting in ascending order, any new record introduced in the heap that is smaller than the last flushed record to the stream of released records cannot be included in the active run: it would not be possible to place such record among the already flushed records. These records are kept at the bottom of the heap, marked as "next run" records, until they fill all the heap. Note that those records preserve an order for the next run but are smaller than all the values in the present run.

When the algorithm closes the current run, it starts a new run which contains all the records that could not be included in the closed run. This breaks the incoming flow of data into multiple runs that are merged in a final phase to create a final single run.

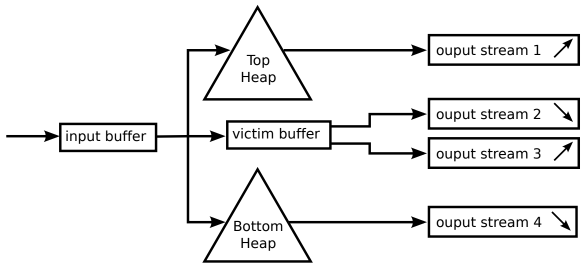


Figure 1: Functional diagram of 2WRS.

The merge phase is strongly dependent on how many runs have been generated before. Runs are stored sequentially in disk, and in order to improve the merging speed it is desirable to have the longest possible runs. Thus, the ideal situation would be when a single run is generated, which, for instance, occurs when the input is already sorted.

The theoretical analysis of RS found that, for input streams following a uniform distribution, the average run length is twice the size of the available memory [5]. However, for some entries such as when the values of the record set are sorted inversely, the runs are the same size of the available memory, generating the largest amount of I/O possible for the memory assigned.

The objective of this paper is to get the most out of RS, i.e. generate runs above double the size of the memory available, no matter the incoming dataset characteristics. This is achieved with two-way replacement selection which is explained below.

2.1 Pseudocode

Algorithm 1 shows the pseudocode for the main loop of RS. In the first phase, method *heap.fill* loads the first records from the input into the heap.

Then, the main loop is executed while the heap is not empty. First, a record is output to make room for a new one. This is done by method *output*. Next, a record is read from the input. If the record is smaller than the last output record, it is marked as belonging to the next run, else it is marked as belonging to the current run. Next, the top record of the heap is removed with *heap.pop* and the record read from the input is inserted in the buffer with *heap.insert(current)*.

Finally, when the top record of the heap belongs to the next run, i.e. it is too small to be part of the current run, the current run ends. The process starts again, and a new run is generated until the input is fully read.

3. TWO-WAY REPLACEMENT SELECTION

Two-way replacement selection (2WRS) implements two heaps that we call TopHeap and BottomHeap, instead of one as in RS. The objective of the two heaps is that they cooperate to obtain longer runs: the TopHeap and BottomHeap capture increasing and decreasing sequences of values respectively. This architecture resembles two cooperating RS algorithms working together, which output their result into streams 1 and 4, as depicted in Figure 1. Therefore, stream 1 is a sequence of increasing values and stream 4 is a sequence of decreasing values that do not overlap.

Algorithm 1 RS(heapSize)

Require: The maximum size of the heap *heapSize*.
Ensure: Each run is sorted.

```
1: let current a pair of integers containing a value for a
   record and the run to which it belongs.
2: let heap a minheap, of maximum size heapSize.
3: let currentRun an integer.
4: let nextOutput an integer.
5: heap.fill(inputBuffer);
6: currentRun = 0;
7: while heap.size() > 0 do
8:     nextOutput = heap.pop()
9:     output(nextOutput);
10:    //Read next value
11:    if input.read(current.value) then
12:        if current.value < nextOutput then
13:            current.run = currentRun + 1;
14:        else
15:            current.run = currentRun;
16:        end if
17:        heap.insert(current);
18:    end if
19:    //Start next run?
20:    if heap.top().run > currentRun then
21:        currentRun = 1 + currentRun;
22:    end if
23: end while
```

In order to decide which heap should be populated with every new input record, we take a decision based on an input buffer and a heuristic. The input buffer is an array of records that acts as a regular I/O buffer, which loads data items from disk and releases them like a FIFO. When a new record has to be inserted in one of the heaps, 2WRS chooses one of the two heaps at random, and the top record is released to the corresponding stream. In an independent decision, the heuristic samples the input buffer and selects which heap, either the TopHeap or the BottomHeap, will store the record chosen by the FIFO structure. If the heuristic chooses the heap that released the top record, then the two heaps preserve their size. Otherwise, one heap grows and the other one shrinks.

We propose and compare the following two heuristics:

Random Every record is pushed into a heap selected at random. With this heuristic, the expected size of the heaps is the same.

Mean If the record to be pushed is larger than the mean of the records in the buffer then it is inserted into the TopHeap. Otherwise, it is inserted into the BottomHeap. This heuristic captures a rough approximation of the data distribution and balances the heaps according to the distribution.

By the nature of streams 1 and 4, the smallest value in the increasing stream and the largest value in the decreasing stream, create a gap between them. If the input stream contains values which are in between these two, it is possible to sort them and keep them in another sorted sequence. We solve this by using the victim buffer and creating two streams, numbers 2 and 3. So, the victim buffer will be a sorted window of records between the smallest value in the increasing stream (stream 1) and largest value in the decreasing stream (stream 4). Whenever an input record is between the largest and the smallest value in the victim buffer, it is stored in the victim buffer. When the victim buffer is full, it is sorted and flushed to streams 2 and 3. The distribution of records between those streams is such that maximizes the gap between the last records inserted in those streams. This way, as shown in Figure 1, streams 2 and 3 will grow until the largest and smallest values in the victim buffer do not fit any input record among them. The initialization of the victim buffer is done when the BottomHeap and TopHeap are just filled at the beginning of the algorithm, taking the top records of both heaps. An example is given in Appendix A.

Among other situations, the use of streams 2 and 3 and the victim buffer are very beneficial for convergent series. The output streams 1 and 4 define an interval of values (those which fall among both streams) that cannot be released in the current run. Therefore, the use of the victim buffer captures this trend.

When a record cannot be inserted in the TopHeap, BottomHeap or the victim buffer, the record is marked as belonging to the next run and it is inserted in the corresponding heap with a special mark indicating that it belongs to the next run, similarly to RS. The records marked as belonging to the next run are considered to be larger (smaller) than all the records belonging to the current run in the TopHeap (BottomHeap).

When the available space for the heaps is full, the algorithm removes one record at random from either the TopHeap or the BottomHeap, and writes the record to stream 1 or 4, respectively. This procedure does not affect the relative size of the heaps, since on average each heap will be selected half of the time. The process is iterated until no more records can be removed from none of the heaps (because all the records are marked as belonging to the next run) and then the whole algorithm is restarted with a new run.

The algorithm creates its result in four different output streams, as opposed to RS, which only uses one stream. However, those four streams are consecutive and non overlapping among them: (1) streams 1 and 3 are sorted in ascending order, (2) streams 2 and 4 are sorted in descending order; (3) any four records x, y, z, w from streams 1, 2, 3 and 4 respectively hold: $x \leq y \leq z \leq w$. Therefore, it is immediate to generate the final run as the sequence of the data generated by streams 4, 3, 2 and 1 in this order.

2WRS behaves identically when the input is already sorted as well as when it is sorted in reverse order because it takes advantage of the TopHeap or the BottomHeap, respectively. In both cases, the runs are of infinite size. Furthermore, the presence of the victim buffer also allows 2WRS to generate runs of infinite size for convergent series, which contain sequences of values that keep approximating. In summary, 2WRS is able to detect structured increasing or decreasing inputs and benefits from their regularity to build longer runs.

3.1 Pseudocode

Algorithm 2 shows the pseudocode for the main loop of 2WRS. The algorithm, first fills both heaps with records obtained from the input. This is done by method *doubleHeap.fill*. When a record can be stored in both heaps, this

Algorithm 2 2WRS(inputBuffer, heapSize, victimBufferSize)

Require: An input buffer *inputBuffer*, the maximum combined size of the heaps *heapSize* and the victim buffer size *victimBufferSize*.
Ensure: The generation of several ordered runs.

```
1: let current a pair of integers containing a value for a record and the run to which it belongs.
2: let doubleHeap a pair of heaps, a maxheap and a minheap, of maximum total size heapSize.
3: let victimBuffer a victim buffer of size victimBufferSize.
4: let currentRun an integer.
5: doubleHeap.fill(inputBuffer);
6: currentRun = 0;
7: while doubleHeap.size() > 0 do
8:     output(doubleHeap);
9:     if inputBuffer.read(current.value) then
10:        current.run = currentRun;
11:        while victimBuffer.fit(current.value) do
12:            inputBuffer.read(current.value);
13:        end while
14:        doubleHeap.insert(current);
15:    end if
16:    if doubleHeap.nextRun(currentRun) then
17:        currentRun = 1 + currentRun;
18:        victimBuffer.flush();
19:    end if
20: end while
```

function uses the heuristic to decide which heap is used to store the record.

The main loop is executed while the two heaps are not empty. First, a record is released to make room for a new one. This is done by method *victimBuffer.output*, which pops the top record from either the TopHeap or BottomHeap at random.

Next, a record is obtained from the input buffer. Method *victimBuffer.fit* checks whether the current record is inside the gap currently processed by the victim buffer and, if so, stores it and returns *true*. Otherwise, it does nothing and returns *false*. Note that at the beginning of each run, while the victim buffer has not been completely filled, this function always returns *false*. While this method returns *true*, new records are read from the input buffer. When the record read can not be put in the victim buffer, the method returns *false* and the record is inserted into one heap by method *doubleHeap.insert*. This method inserts the record into one of the heaps, using the first heuristic when necessary.

Finally, method *doubleHeap.nextRun* returns *true* when the top record of both heaps belong to the next run, meaning that the current run reached an end, since all records in memory also belong to the next run. In this case the records stored in the victim buffer are written to disk and the next run starts, with an empty victim buffer.

The main loop of this algorithm is very similar to the main loop of replacement selection, shown in Section 2.1. The only difference is that 2WRS checks in the main loop whether the current record can be placed in the victim buffer and keeps reading new records while this is the case.

4. ANALYSIS

In this section we demonstrate the properties of RS and 2WRS. This way, we show that 2WRS is able to sort incoming data, generating runs which are, at least as long as those generated by RS for random data (Theorem 2) and longer than RS for other data inputs (Theorems 4 and 6).

THEOREM 1. *For inputs already sorted in ascending order, RS generates one run containing all the input records.*

PROOF. Since the input records are already sorted, each new record will be larger than all the values in the heap and, thus, it will be possible to insert it into the heap as belonging to the present run. No record will be marked as belonging to the next run. □

THEOREM 2. *For inputs already sorted in ascending order, 2WRS generates one run containing all the input records.*

PROOF. The same proof as for Theorem 1. All records obtained from the input are larger than those stored in memory. All the records are stored in the TopHeap, and all they belong to the same run. □

THEOREM 3. *For inputs sorted in reverse order, RS generates runs with length equal to the size of the memory.*

PROOF. Since the input records are sorted in reverse order, the next record obtained form the input is smaller than all the previous records. Thus, it is not possible to include the new record in the current run when the heap is full. So, the new record is marked as belonging to the next run. When the heap is full every new record belongs to the next run. Once the records belonging to the present run are released, a new run starts and the size of the run is equal to the available memory. □

THEOREM 4. *For inputs sorted in reverse order, 2WRS generates one run containing all the input records.*

PROOF. The records obtained from the input are smaller than all the records in memory. However, in contrast to RS, those records are inserted in the BottomHeap. Since all the records from the BottomHeap can be used in the current run, all the stream is released in a single run through the BottomHeap. □

THEOREM 5. *For inputs consisting of alternating chunks of length k records sorted in ascending order and k records sorted in descending order repeatedly, RS generates runs with an average length around twice the size of the memory m ($m << k$).*

PROOF. Let m be the size of the memory. Every chunk of k records sorted in ascending order is placed in the same run, as per Theorem 1.

When the algorithm starts reading records sorted in reverse order, only the first $^m/_2$ will be included in the current run. The rest of the records sorted in reverse order are put in runs of length m, as per Theorem 3. Therefore, the number of runs generated in a descending section is $\left\lfloor \frac{k-\frac{m}{2}}{m} \right\rfloor = \left\lfloor \frac{k}{m} - \frac{1}{2} \right\rfloor$.

The last m records of a chunk of k records sorted in reverse order ($(k - \frac{m}{2}) \bmod m$) are placed in the same run as the following k records sorted in ascending order.

So every chunk of k records sorted in ascending order is included in a run together with $^m/_2$ records from the next chunk of k records sorted in descending order, plus the last records from the previous run $(k - \frac{m}{2}) \bmod m$.

The average run length is then the total number of records divided by the number of generated runs,

$$\frac{2k}{1 + \lfloor \frac{k}{m} - \frac{1}{2} \rfloor} \quad (1)$$

The denominator of this formula can take the values $\lfloor \frac{k}{m} \rfloor$ and $\lfloor \frac{k}{m} + 1 \rfloor$. The formula achieves maximum value when the denominator is minimum. The maximum average run length is then

$$\frac{2k}{\lfloor \frac{k}{m} \rfloor} \approx 2m \quad (2)$$

□

THEOREM 6. *For inputs consisting of alternating chunks of length k records sorted in the reverse order, two-way replacement selection generates runs with an average length equal to k (with an appropriate heuristic*[1]*).*

PROOF. 2WRS behaves identically to RS for the chunks sorted sorted in ascending order, thanks to the TopHeap. For the chunks with records sorted in reverse order, 2WRS captures the trend with the BottomHeap, generating runs of k records, as well. Thus, the average run length is k. □

5. RUN LENGTH ANALYSIS

In this section, we test the configuration parameters of 2WRS following the analysis of variance (ANOVA). The ANOVA detects which variables are more relevant and it is used to select the optimal configuration for a set of variables (for further details about ANOVA, see [10]). Our output variable will be the length of the runs, and hence, the variable to be optimized. In these experiments, the memory size allocated to the algorithm is fixed to 100K records and the input length is 1GB. Each record is formed by a 4B integer.

The observations are obtained as a crossed factorial experiment with four variables:

- *Buffer setup:* We test three configurations: only input buffer, only victim buffer, and both input and victim buffer.
- *Size of buffer*: We set three configurations: 0.2%, 2% and 20% of the available memory are dedicated to the buffers and the rest to the heaps. Note that in all the configurations, the total allocated memory (the addition of the heap and buffer sizes) for 2WRS is always constant.
- *Heuristic:* We test two configurations for the heuristic of the input buffer: random and mean.
- *Data distribution:* We test five different data input distributions. (1) Sorted: The records are already sorted. (2) Reverse sorted: the inputs are sorted in reverse order. (3) Alternating: this dataset is a sequence of increasing intervals followed by decreasing intervals. The number of intervals is set to 50, with 25 increasing and 25 decreasing interleaved intervals. (4) Random: The records are generated following a uniformly random distribution. (5) Mixed: This dataset alternates one record from a sequence of increasing records, with another record of a sequence of decreasing records. We depict these datasets in Figure 2. In order to add some randomness to the experiments a uniformly distributed random value is added to each input. These random values range from 1 to 1000 for a total range of values sorted from 1 to 10^9.

[1]An appropriate heuristic is one that uses the TopHeap for sorted inputs and the BottomHeap for reverse sorted inputs.

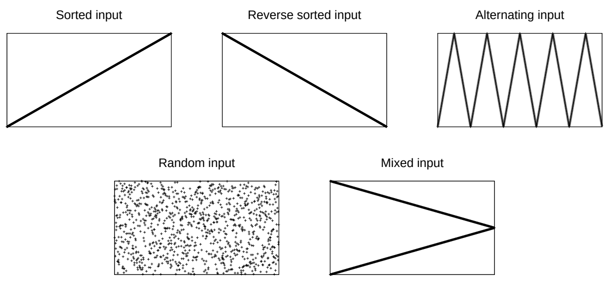


Figure 2: Samples of all data inputs used.

Input	RS	2WRS cfg 1	2WRS cfg 2	2WRS cfg 3
Sorted	inf	inf	inf	inf
Reverse sorted	1.0	inf	inf	inf
Alternating	1.94	50	50	50
Random	2.0	2.0	1.6	1.96
Mixed	2.0	1.2	16.5	2.24

Table 1: Average run length relative to memory size. All three 2WRS configurations use the mean heuristic. The first configuration sets the buffer size to 0.02% and allocates the input buffer. The second and third configurations use both buffers with size 20% and 2% respectively.

In Table 1, we summarize the average run length for all the input sets. In this table, we show the run length of RS compared to the best three parameterizations of 2WRS. We report two of the configurations that maximize the run length for the mixed and random datasets, and a third configuration that works reasonably well for all inputs.

For the remaining configurations (sorted, reversed and alternating), we have found that all configurations of 2WRS are optimal independently of the configuration: 2WRS generates runs of infinite size for the sorted and the reverse sorted datasets, and 2WRS builds runs of length 50 times the memory size for alternating sequences of length 50. On the other hand, RS is only able to generate runs of one size equal to the memory available for the reverse sorted and equal to approximately twice the memory for the alternating data, as found in Theorems 5 and 6. This shows that when datasets are ordered or partially ordered 2WRS is more effective than RS.

The configuration parameters, and specially the buffers and the heuristic, have an important effect on the mixed datasets. Although not shown in Table 1, the configurations without both the input and victim buffers show poor run lengths. Additionally, we computed the average run length

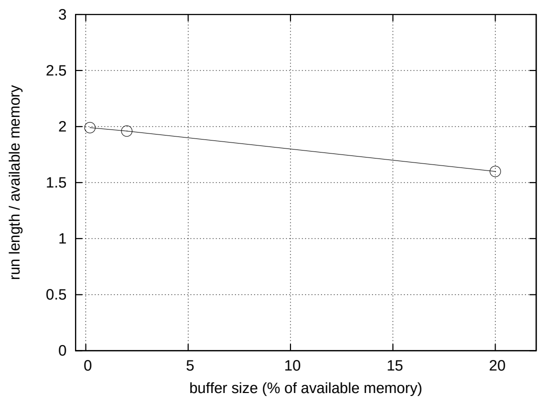


Figure 3: Length of runs relative to memory size as function of buffer size for random inputs.

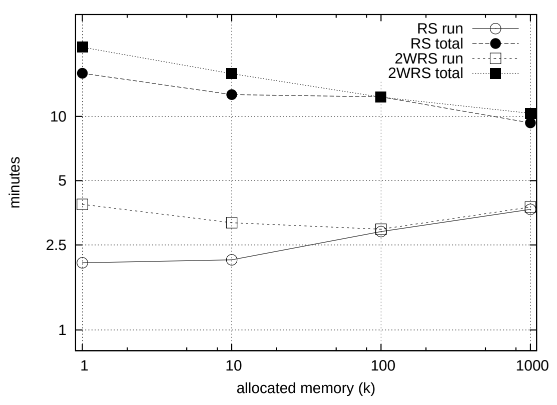


Figure 4: Run generation and total sorting times for random input as a function of available memory.

with respect to the heuristic used, which were 3.0 for mean and 1.85 for random. A t-student test confirmed that with a significance level of 0.05, these two averages are different. Therefore, we conclude that the mean heuristic is better than the random one.

With respect to the random distribution, we found that 2WRS is as good as RS because none of them is able to take advantage of any pattern in the distribution. In our results, we observed that for random distributions, there is a linear correlation between the buffer size and the run length. If buffers are allocated, the memory dedicated to the heap diminishes by this percentage. Thus, a configuration with 2% of the memory dedicated to buffers, reduces the run length by just 2% for random distributions, as shown in Figure 3. Furthermore, in our experiments, we measured a very small difference in the length of the runs generated by the configurations with 0.2% and 2% allocated to the buffers, but larger between 2% and 20%.

All in all, our run length analysis concludes that 2WRS creates runs of a length at least equal to RS or significantly better. 2WRS is able to capture partially sorted data such as those in the alternating and mixed datasets, and is optimal with totally sorted data either increasingly or decreasingly. Regarding the configuration, we found that the presence of buffers and the mean heuristic is very important because they generate longer runs for the datasets with more complex structures.

6. PERFORMANCE ANALYSIS

In this section, we test experimentally the performance of 2WRS with respect to RS. In all our experiments, we account for the time to generate the runs, as well as the subsequent merge phase. We generate different datasets following the random, mixed, alternating and decreasing patterns and we measure the sorting time for each strategy. We do not show the results for the sorted dataset because RS and 2WRS are equivalent.

All the 2WRS configurations used in the experiments analyzed in this section use the mean heuristic, since this combination generates longer runs overall for all the inputs analyzed as we have seen in Section 5. According to the previous section, a large buffer benefits mixed datasets, and a tiny buffer benefits random inputs. Therefore, we set the available memory for the buffers to an intermediate value, which is 2%.

We perform two experiments varying the input length and the memory allocated to the sort algorithm. In the first experiment, we fix the input to 1GB and vary the memory from 1k to 1M. Therefore, out tests aim at systems with large inputs (between 3 and 6 orders of magnitude larger) with respect to the memory available. In the second experiment, we fix the memory to 10k and we vary the input from 100MB to 1GB.

Setup: We execute the algorithms in a computer equipped with an Intel Core 2 Duo processor running at 2.40 GHz. Each core has 4 KB of L2 cache memory and the system has a total of 2 GB of RAM. The hard disk is a SATA drive with a capacity of 60 GB. The OS of the system is Debian GNU/Linux. Given that we want to limit the available memory dedicated to sorting, we open all files using direct I/O, which bypasses the operating system cache. 2WRS generates output streams containing decreasing sequences of records.

Fan in analysis: The merge phase is computed as a tree of run merges. Depending on the number of files merged simultaneously (i.e. the fan in), the performance of the algorithm varies. In an experiment detailed in Appendix B, we measured the best fan in for merging runs in our experimental setup, and we obtained that the optimal fan in is equal to 10 runs. Thus, in all the following experiments with RS and 2WRS, we use this fan in for the merge phase.

6.1 Random

In Figure 4, we plot the performance with respect to the memory buffer size. The time spent generating the runs is detailed with empty circles and squares for RS and 2WRS, respectively, and the total time needed to sort the records is depicted with solid circles and squares. The same applies to the rest of the plots in this section. We observe that the total time needed by the two algorithms is very similar. This is due to the fact that it is not possible to predict the behavior of random input data. 2WRS has slightly worse performance during the run building phase for some configurations because the logic of 2WRS is slightly more complex than for RS, due to the two heaps and the multiple streams. However, the difference between both algorithms is tiny, and

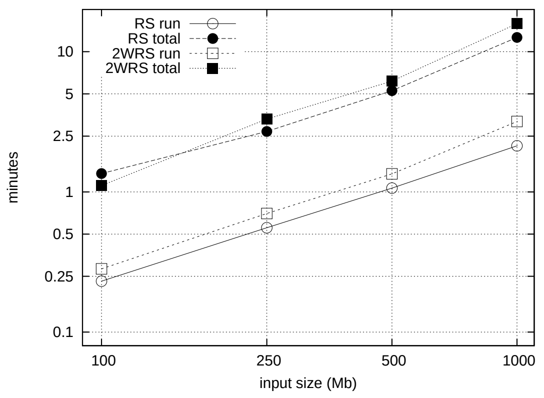


Figure 5: Run generation and total sorting times for random input as a function of input size.

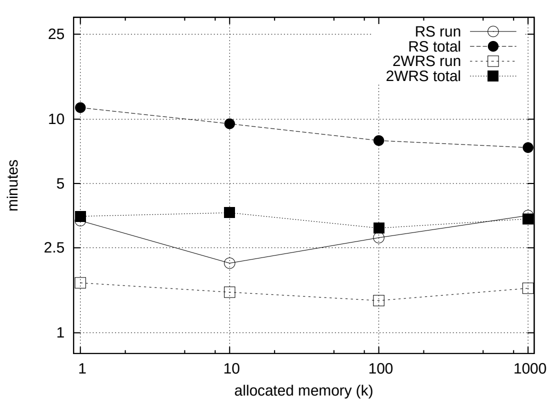


Figure 6: Run generation and total sorting times for mixed input as a function of available memory.

thus the use of either RS or 2WRS is equivalent for random inputs.

We plot the scalability of the algorithms with respect to the input length in Figure 5. Here, we observe a similar pattern to that described for the previous plot, where both algorithms consume a similar time. Furthermore, we observe that both algorithms scale identically when the input size grows.

6.2 Mixed

In the run length section, we found that 2WRS creates runs significantly larger than RS for mixed datasets. Figure 6 confirms it, because independently of the memory size, 2WRS is approximately three times faster than RS. This is because 2WRS generates less runs for the mixed dataset, and so the merge phase is much faster than with RS. We also see that, as the amount of allocated memory increases, both algorithms need less time to sort the data, since the runs generated are longer, and thus less merge phases are needed.

In Figure 7, we represent the scalability of both algorithms with the input. The advantage of 2WRS over RS for mixed data is sustained as the input data grows, and for all input sizes an approximate speedup of 3 is maintained. We note that for this dataset even the run generation of 2WRS is

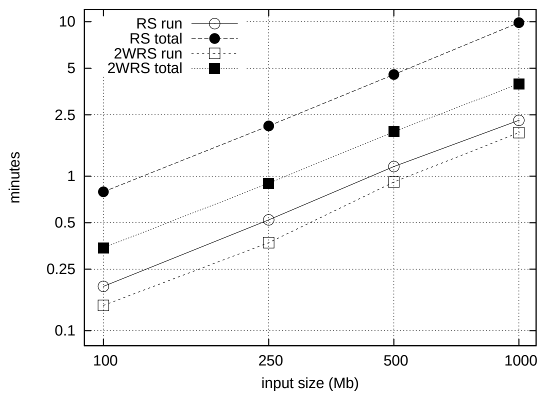


Figure 7: Run generation and total sorting times for mixed input as a function of input size.

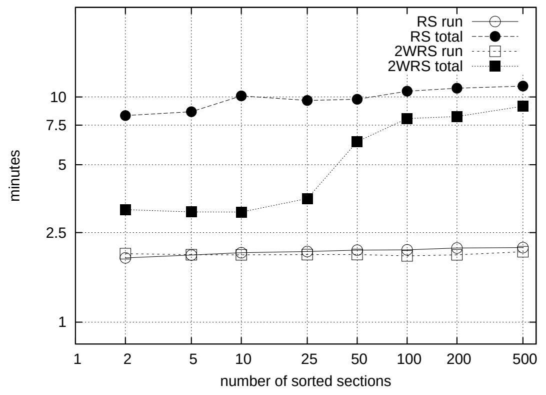


Figure 8: Run generation and total sorting times for alternating input as a function of the number of sorted and reverse sorted sections.

faster. This is because the heaps are not used and most of the computational time is spent sorting the victim buffer. Since the victim buffer uses a standard library sort, which is optimized for efficient in memory sorting, it is faster than RS that applies a heapsort.

6.3 Alternating

The complexity to sort the alternating dataset is dependant on the number of increasing and decreasing intervals for a fixed input size. If there are very few intervals, the dataset is similar to the sorted dataset, but if there are many intervals then it becomes closer to the random dataset. In this experiment, we fix the memory allocated to the algorithms to 10k and the input size to 1GB, and we vary the number of increasing and decreasing sections. In Figure 8, we depict the sorting time for both algorithms.

For a small number of sorted sections, 2WRS performs much better than RS, achieving up to an approximate speedup of 3. We observe that although the run phase takes the same time for both algorithms, the merge phase is significantly shorter for 2WRS because of the fewer number of runs. 2WRS is able to include the sections sorted in a single run in reverse order, whereas RS creates multiple runs for these sections. As the number of peaks increases, the sorted

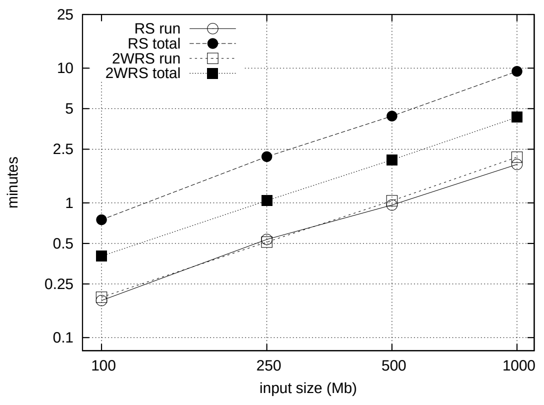


Figure 9: Run generation and total sorting times for reverse sorted input as a function of input size.

sections are shorter. Then, both algorithms asymptotically tend to need the same amount of time for sorting the data, although 2WRS still performs better. In the extreme case, if the number of peaks tends to infinite, the dataset would resemble a random input and both algorithms would spend the same execution time.

6.4 Reverse Sorted

In Figure 9, we plot the time spent by both algorithms to order reverse sorted data, as the size of the input grows. We observe that for all input sizes 2WRS gets a better performance than RS. The scalability of both algorithms is similar, showing parallel trends, that indicate a constant speedup, which is in this case 2.5.

7. RELATED WORK

Sorting is a basic computing problem that has received a lot of attention over the years. The basis of most external sorting algorithms is a two step process that in the first phase generates runs as long as possible, and in the second phase it merges the runs. Often, the run generation phase is based on some internal sorting algorithm. In particular, replacement selection is based on heapsort, which is analyzed in [5].

In 1998, Larson and Graefe experimentally compared different memory management algorithms during run generation when ordering variable length inputs, showing that replacement selection is a viable algorithm for commercial database systems [8]. Moreover, a recent survey on sorting in database systems pointed out that replacement selection is one of the most used techniques for external sorting in databases [3].

Replacement selection was introduced by Goetz in [2] and since then several modifications and alternatives have been proposed. For instance, Larson introduced a modified version of RS called batched replacement selection, a cache conscious version that also works for variable length records [7]. More recently, Koltsidas, Müller and Viglas introduced a new variation of replacement selection for sorting hierarchical data (e.g. XML files) [6].

There have been several proposals to improve the performance of the merge phase, but no emphasis has been placed on the generation of larger runs in the general case. Zheng and Larson introduced a new reading strategy for external mergesort that consistently performs better than double buffering and forecasting [12]. This technique uses heuristics to precompute the order in which data blocks will be read during the merge phase.

Yiannis and Zobel studied the possibility of compressing sets of records during the run generation phase in order to reduce disk and transfer costs of external sorting by reducing the number of runs generated, and proposed a new compression technique adapted to sets of records [11].

We should note that all modifications and improvements of RS can be readily applied to 2WRS without change, so 2WRS also benefits from all these changes.

8. CONCLUSIONS

In this paper, we propose Two-way Replacement Selection (2WRS), which is a generalization of Replacement Selection (RS). 2WRS allows to deal with increasing, decreasing and mixed inputs, obtaining runs of optimal size, and significantly longer than RS. Moreover, this improvement does not penalize the length of the runs of 2WRS for random distributions, the length of which is similar to those of RS. Besides, the additional complexity generating the runs is amortized by a faster merge phase, which turns into a much faster total execution time.

Our results have been tested for different input sizes and space dedicated to the sorting operation. We have been able to sort datasets with strong memory limitations (6 orders of magnitude larger) three times faster than the regular RS. Furthermore, we obtain similar speedups with different scaleups of the input.

Additionally, the amount of memory allocated to 2WRS can be fixed beforehand as with RS, which makes our proposal also suitable for DBMSs. Finally, 2WRS maintains the heap and run generation architecture of RS that allows for improvements already proposed in the literature for RS, which include variable key support, read ahead strategies or hierarchical data sorting among others.

9. REFERENCES

[1] Anon et al. A measure of transaction processing power. *Datamation*, 31:112–118, 1985.

[2] M. Goetz. Internal and tape sorting using the replacement-selection technique. *Communications of the ACM*, 6(5):201–206, 1963.

[3] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3), 2006.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[5] D. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, 2nd edition, 1998.

[6] I. Koltsidas, H. Müller, and S. Viglas. Sorting hierarchical data in external memory for archiving. *Proceedings of the VLDB Endowment archive*, 1(1):1205–1216, 2008.

[7] P. Larson. External sorting: run formation revisited. *IEEE TKDE*, 15(4):961–972, 2003.

[8] P. Larson and G. Graefe. Memory management during Run Generation in external Sorting. In *SIGMOD*, pages 472–483. ACM, 1998.

[9] Minute sort web page. http://sortbenchmark.org.

[10] D. Mongomery. *Design and Analysis of Experiments.* Wiley, 5th edition edition, 2000.

[11] J. Yiannis and J. Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, 2007.

[12] L. Zheng and P. Larson. Speeding up external mergesort. *IEEE TKDE*, 8(2):322, 1996.

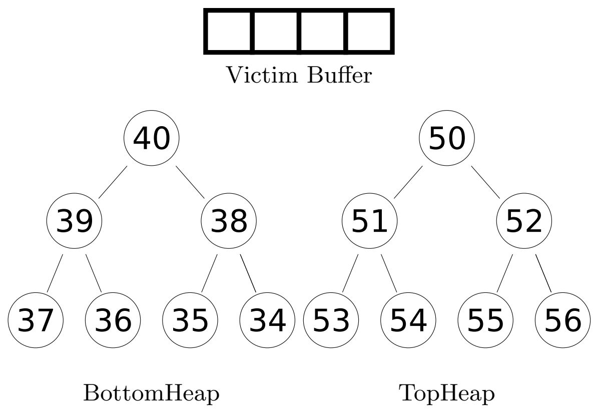


Figure 10: The two heaps after they are filled. The victim buffer and the output are still empty.

APPENDIX

A. EXAMPLE

In order to see the general behavior of the algorithm, a small example is presented here. Assume that we have an input and a victim buffer of length 4 each, and 14 units of memory for the heaps. The input data will be

$$\{40, 50, 39, 51, 38, 52, 37, 53, 36, 54, 35, 55,$$

$$34, 56, 33, 57, 32, 58, 44, 39, 59, 60, 61, \dots\}$$

This input data alternates records sorted with records sorted in reverse order, leaving a gap for the victim buffer to use.

At the start, the first thing to do is to fill the input buffer. Then, the buffer contains the four first records of the input, $\{40, 50, 39, 51\}$. Now, the algorithm reads the first record from the input buffer, which is 40. This record can go into either heap, because both are empty. Since 40 is not greater than the mean of the input buffer contents (45), it is pushed into the BottomHeap. A new record, 38, is inserted into the input buffer FIFO and we pop the head, 50. This time, 50 greater than the mean (44.5), so it is inserted into the TopHeap. The next record is 39, which can only go to the BottomHeap because it is smaller than the top record of the BottomHeap, 40. We iterate this process until the heaps are full. In Figure 10 we show the content of the heaps at this point.

When the two heaps are full, the top record of one of the heaps is chosen at random (in this case we take the BottomHeap) and put in the victim buffer. Then a new record is inserted into one of the heaps. The top of the BottomHeap is 40, which it is inserted into the victim buffer. The next record is 33, which goes to the BottomHeap. The content of the heaps is shown in Figure 11. This process is repeated until the victim buffer is full.

Once the victim buffer is full, it is sorted, as shown in Figure 12. The largest gap in the victim buffer is that between 40 and 50, so records that are smaller than (or equal to) 40, i.e. 39 and 40, are written to the output stream 3. The rest of the records (50 and 51) are written to the output stream 2. The victim buffer is now empty, and accepts records between 40 and 50.

Next, a new heap is selected randomly (in this case the

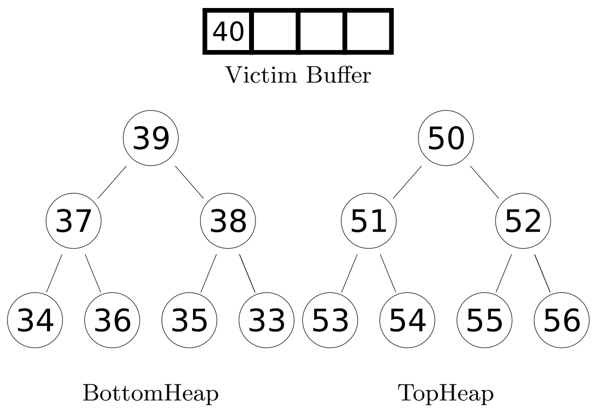


Figure 11: The two heaps after the first record is put in the victim buffer.

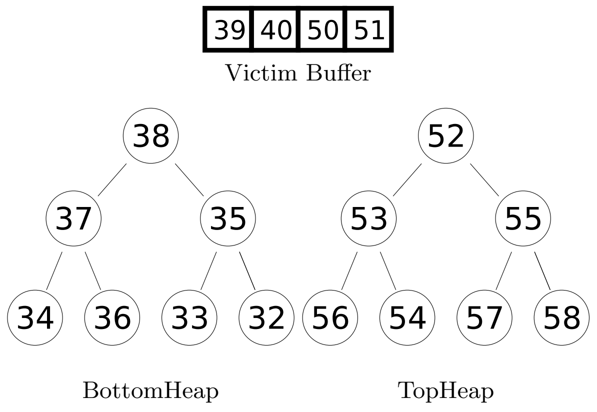


Figure 12: The two heaps after the victim buffer is full.

BottomHeap) and its top, 38, is written to the output stream 4. The next record in the input buffer is 44. It can not be inserted into either heap, but since it is between 40 and 50, it is inserted into the victim buffer. The next record is 39. It is not possible to use it in the current run, but since it is not between 40 and 50, it can not be put in the victim buffer. In this case, it is marked as belonging to the next run and inserted into the BottomHeap, because 39 is not larger than the mean of the contents of the input buffer, which are now $\{39, 59, 60, 61\}$. The content of the heaps now is shown in Figure 13. Given that 39 belongs to the next run, it is considered smaller than any other record in the BottomHeap for the current run.

The algorithm continues until all record in both heaps are marked. Then, the current run is finished and the next one is started.

B. FAN-IN ANALYSIS

In this experiment, we measure the fan in that achieves the best performance in our computer. In our experiment, we generate 400 files, each one with size 16MB, which contain integers already sorted following a uniform distribution (i.e. 400 runs), and we merge them. This experiment is independent of the algorithm that generates the runs, thus is valid for RS and 2WRS.

The fan in is a compromise between two characteristics:

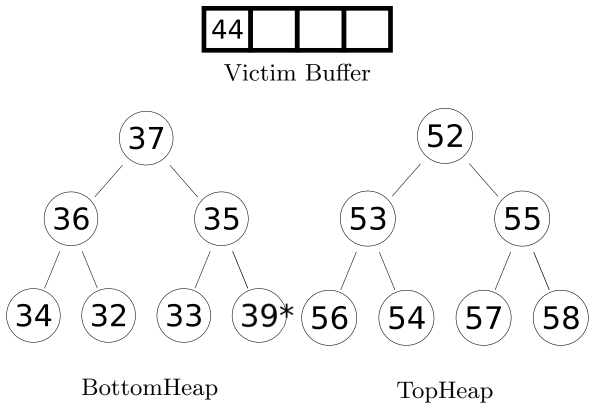


Figure 13: The two heaps at the end of this example. The asterisk next to 39 marks it as beloning to the next run and, as such, it is considred to be smaller than any record in the current run.

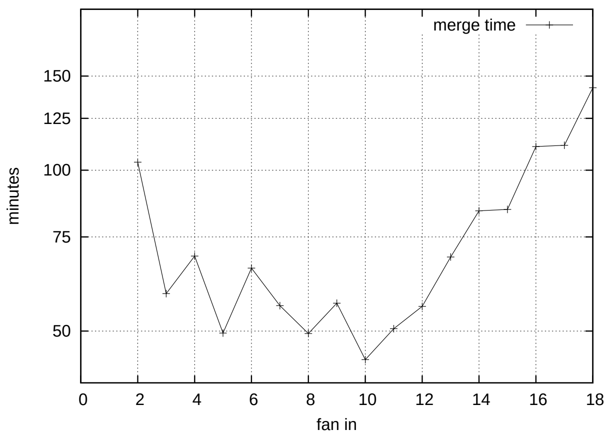


Figure 14: Merge time for different values of fan-in.

(a) the smaller the fan in, the more sequential is the access to the files from disk, but (b) the larger the fan in, the less merge operations are required to end the task.

We observe this tradeoff between the two benefits in Figure 14. If the fan in is too small, the algorithm takes more time because it must perform more merge steps. However, if the fan in is too large, the head of the disk performs more seeks and the bandwidth obtained from the disk is smaller. In our experiments, the minimum time was observed for a fan in 10, which means that in each merge step 10 different files are simultaneously merged.

C. STORING DECREASING RECORDS

Due to the way 2WRS works, it generates two streams of sorted records and two streams of reverse sorted records. The latter need to be stored already sorted in order to allow the merge phase to read files sequentially.

In order to do this, when the file is first created, a fixed amount of pages is allocated on disk. The records are then stored starting at the end of the file, one page at a time, until the first page of the file is reached. When the first page is reached, a new file is created and filled in the same way. The first page of the file stores information, as where does the data start and how many files have been generated to store that run. Note that the generation of several files does not affect negatively the fan-in, since they are read in sequential order. The only overheads are the closing of one file and opening of the next one, and possibly one seek in the first file to go to the page where the data starts.