TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Optimizing Sort in Hadoop using Replacement Selection

by

Pedro Martins Dusso

A thesis submitted in partial fulfillment for the degree of
Master of Science

in the

Fachbereich Informatik
AG Datenbanken und Informationssysteme

May 2014

Declaration of Authorship

I, Pedro Martins Dusso, declare that this thesis titled, 'Optimizing Sort in Hadoop using Replacement Selection' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"I like design to be semantically correct, syntactically consistent, and pragmatically understandable. I like it to be visually powerful, intellectually elegant, and above all timeless."

Massimo Vignelli

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstract

Fachbereich Informatik
AG Datenbanken und Informationssysteme

Master of Science

by Pedro Martins Dusso

This thesis implements and evaluates an alternative sorting component for Hadoop based on the replacement-selection algorithm. Hadoop is an open source implementation of the MapReduce framework. MapReduce's popularity arises from the fact that it provides distribution transparency, linear scalability, and fault tolerance. This work proposes an alternative to the existing load-sort-store solution which can generate a small number of longer runs, resulting in a faster merge phase. The replacement selection algorithm usually produces runs that are larger than available memory, which in turn reduces the overall sorting time. This thesis first describes different sorting algorithms for in-memory sorting and external sorting; secondly, it describes Hadoop and Hadoop's map tasks and output buffer strategies; thirdly, it describes four sorting implementation alternatives, which are presented from the most simple to the most complex. Furthermore, this work analyzes the performance of these four alternatives in terms of execution time for run generation and merging, and the number of intermediate files produced. Finally, we provide a critical analysis with respect to Hadoop's default implementation.

Acknowledgements

In Portuguese we say "diga-me com quem andas e eu te direi quem és", which I semantically translate as *wer mit den Weisen umgeht, der wird weise.* I would like to express my very great appreciation to Professor Theo Härder for the continuous support since my first days in Kaiserslautern. I always had everything that was necessary to study and work — a great environment filled with marvelous and intelligent people. Professor Härder's career and work are a huge inspiration for me, and I can only hope to cause a small portion of the great impact that his work has caused in our science and in our society. The foundations are laid, now it's our turn. I would also like to offer my special thanks to my advisor, colleague, tutor, friend, roommate, sports and drinking partner Caetano Sauer. I have learn from him philosophy, linguistics, politics, astronomy, mathematics, photography, history, and also a little of computer science. We probably have not "solved out" all of our problems, but I'm sure we have much harder questions to answer today than when we met. My special thanks are extended to Vítor Uwe Reus, a friend which I do not hesitate to call a brother, and who has shared with me not only our work space but also many trains and bottles. I believe the sayings in the beginning of this paragraph are true, because merely walking next to these men made myself a much better individual.

My most beloved thanks goes to Carolina Ferret, who filled with love and care not only our first house here in Germany but my life as a whole. Her endeavor coming with me while I as pursuing this title was a huge love act, and which I will always be thankful to. Traveling, eating, listening to music, dancing, and all the best things in life are even better when we can share them with someone we love and I share with her. I will never be able to be grateful enough to my family, which never get tired of asking when I was coming back. To my mother, Cláudia, for her infinite patience and infinite strength. To my father, Eduardo, for the clear path to follow. To my brother, Henrique, for being so different to me but so similar at same time. For all friends I made during this journey, *Ich hoffe wir auf den Straßen der Welt treffen.*

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 2

2 Sorting 4
- 2.1 In-memory sorting . . . 4
  - 2.1.1 Quicksort . . . 6
  - 2.1.2 Heapsort . . . 7
  - 2.1.3 Mergesort . . . 8
- 2.2 External memory sorting . . . 9
  - 2.2.1 Run generation with quicksort . . . 11
  - 2.2.2 Run generation with Replacement Selection . . . 12
  - 2.2.3 Comparison . . . 13
  - 2.2.4 Merge . . . 14

3 Hadoop 18
- 3.1 Motivation . . . 18
- 3.2 Running a MapReduce job . . . 19
- 3.3 Map and Reduce tasks . . . 20
  - 3.3.1 The Map Side . . . 22
- 3.4 Pluggable Sort . . . 28

4 Implementation 31
- 4.1 Development and experiment environment . . . 31
- 4.2 Record Heap . . . 32

4.3 Pointer Heap 36
4.3.1 Memory Manager 37
4.3.2 Selecting and replacing 38
4.3.3 Memory Fragmentation 41
4.3.4 Comparing Records 42
4.4 Prefix Pointer Heap 43
4.5 Prefix Pointer with Custom Heap 44

5 Experiments **48**
5.1 Asynchronous Writer 48
5.1.1 Single disk 49
5.1.2 Dual disks 50
5.2 Ordered, Partially Ordered and Reverse Ordered 51
5.3 Real World Jobs 58

6 Conclusions **60**

Bibliography **63**

List of Figures

2.1 Quicksort procedure . . . 6
2.2 Array and equivalent complete binary tree . . . 8
2.3 Heapsort procedure . . . 9
2.4 Mergesort procedure . . . 10
2.5 Multiway merge with four runs . . . 14
2.6 Four-way merging . . . 15
2.7 Multiway merge with four runs with buffer . . . 16
2.8 Merging twelve runs into one with merging factor of six . . . 17

3.1 How Hadoop runs a MapReduce job . . . 20
3.2 Map and reduce tasks in detail . . . 21
3.3 In-memory buffers: key-value and metadata . . . 24
3.4 Kvoffsets buffer sorted by partition . . . 25
3.5 Two spill files and the resulting segments for each partition . . . 27
3.6 Merge tree with 20 segments and merge factor 10 . . . 28
3.7 The map task execution . . . 29

4.1 Testing setup of micro-benchmarks . . . 32
4.2 Comparison between Hadoop's `MapOutputBuffer` and our custom `RecordHeap` buffer . . . 34
4.3 Java's heap space in two different scenarios . . . 35
4.4 A possible state of a memory buffer with its corresponded memory manager 38
4.5 Flowchart describing the collection method of `PointerHeap` . . . 39
4.6 Flowchart describing the flushing method of `PointerHeap` . . . 40
4.7 Comparison between Hadoop's `MapOutputBuffer` and our custom `PointerHeap` buffer . . . 40
4.8 Fragmentation levels depending on the input data . . . 42
4.9 Comparison between Hadoop's `MapOutputBuffer` and our custom `PointerHeap` buffer . . . 43
4.10 Percentage of comparisons decided with key prefix and full key for random strings and English words . . . 44
4.11 `MetadataHeap` fields . . . 45
4.12 Comparison between all output buffers . . . 46
4.13 Comparison between `PrefixPointerHeap` and `PPCustomHeap` . . . 47
4.14 Comparison between `PrefixPointerHeap` and `PrefixPointerCustomHeap` 47

5.1 The circular buffer and the writer thread . . . 49
5.2 Run generation with a single disk or dual disks . . . 50

5.3 Comparison between output buffers using synchronous and asynchronous writer with one hard disk 50
5.4 Comparison between output buffers using synchronous and asynchronous writer with two hard disks 51
5.5 Baseline experiment, executed over a random version of lineitem table . . 52
5.6 Lineitem table sorted by orderkey column 53
5.7 Partially ordered file by shipdate results 54
5.8 Partially ordered file by receiptdate results 54
5.9 Lineitem table sorted by second date column 55
5.10 Number of records in the buffer and number of free blocks 56
5.11 Number of occupied blocks 57
5.12 Length of raw records and rounded records from lineitem table 57
5.13 Join of lineitem and order tables, using a buffer size of 100MB 58
5.14 Join of lineitem and order tables, using a buffer size of 16MB 59

List of Tables

2.1 Comparison of Storage Technologies 10
2.2 Run generation with replacement selection 12

4.1 Sample from EnglishWords and RandomStrings input files. 41

Para Eduardo e Cláudia

Chapter 1

Introduction

1.1 Motivation

It has being a decade since the first papers about the so called MapReduce framework were published [1–3] by Google. During this period, data management research on *big data* moved from the laboratories to the reality of working clusters with thousands of machines, where a volume of data in the order of petabytes is processed daily running some implementation of the MapReduce framework. Most of this popularity can be credited to MapReduce's open source version, called *Hadoop*, released by Yahoo! and today supported by the Apache Foundation. Hadoop runs on commodity hardware, making it easy and cheap to deploy; it provides both a programming model and a software framework for analysis of distributed datasets, which frees the developers (or, using a modern term, the *data scientists*) from the technical aspects of distributed execution over a cluster, fault tolerance, process communication, and data sharing, letting them focus on the analytical algorithms needed to solve the existing problems.

During the last years, Hadoop is evolving thanks to the contribution of its users — which are not only persons but also companies like Cloudera, Yahoo, Facebook, IBM, and others. Nevertheless, there is a lot of room for improvement in two main landscapes: the first is the development of additional tools that enhance the abstraction of the framework in different levels and integrate it with other data management tools. In [4], J. Kobielus anticipates that massively parallel data processing systems will not make traditional warehousing architectures obsolete; instead, they will supplement and extend the data warehouse in strategic roles such as extract/transform/load (ETL), data staging, and preprocessing of unstructured content. Working in this direction, the data management community can achieve an integrated (but diverse) platform for multi-structured data.

Approaches like [5, 6] try to address these issues with a higher-level data-processing system on top of Hadoop.

The second landscape where one can improve Hadoop is by modifying or customizing the middleware itself. The framework itself is not hard to install and use, but it presents challenges for developers of extensions and customizations. Changes are slow, and many of the new features are implemented by third-party entities (a customization due a particular necessity) and integrated into the main branch. This highly heterogeneous and decoupled development environment makes the code hard to understand and modify, which was one of the main challenges in this thesis. Hadoop was first released in 2007 and in 2012 its second (alpha) version became public, with major features such as federation of the underlying distributed file system, a dedicated resource negotiator, and many performance improvements.

1.2 Contribution

This thesis implements and evaluates an alternative sorting component for Hadoop based on the replacement-selection algorithm [7]. Sorting is used in Hadoop to group the map outputs by key and deliver them to the reduce function. The original implementation is based on quicksort, which is simple to implement and very efficient in terms of RAM and CPU. However, data management research has shown that replacement selection may deliver higher I/O performance for large datasets [8, 9]. Sorting performance is critical in MapReduce, because it is essentially the only non-parallelizable part of the computation. The sort stage of a MapReduce job is very network- and disk-intensive, and thus the superiority of quicksort for in-memory processing may not be directly manifested in this scenario. Our goal in this thesis is to evaluate replacement selection for sorting inside Hadoop jobs. To the best of our knowledge, this is the first approach in that direction, both in academia and in the open-source community.

The remainder of this thesis is organized as follows. Chapter 2 reviews algorithms for in-memory and disk-based sorting, focusing on the replacement-selection algorithm. In chapter 3, we present the internals of job execution in Hadoop, focusing on the algorithms and data structures used to implement external sorting using quicksort. The infrastructure presented will serve as basis for our implementation of replacement selection, which will be discussed in Chapter 4. We implemented four versions of the algorithm using different mechanisms for internal memory management and sort processing. Our goal was to start from a simple naive implementation and gradually improve it to deliver the best possible performance. An experimental evaluation of each alternative in comparison with Hadoop's original method is also presented. In Chapter 5, we compare our

replacement-selection method against original Hadoop, using a wide range of scenarios representative of real-world usage. The experiments in this chapter are executed at the level of whole MapReduce jobs, instead of the micro-benchmarks performed in Chapter 4. Finally, chapter 6 concludes this thesis, providing a brief overview of the pros and cons of our solution, as well as discussing numerous open challenges for future research.

Chapter 2

Sorting

Sort algorithms can be classified into two main categories. When the input data set fits in main memory, an *internal* sort can be executed. If not, an *external* sort algorithm is needed. We sort in-memory using sorting algorithms such as quicksort, heapsort, mergesort, and others.

It is necessary to rethink the sorting procedure when the input data set is bigger than the available main memory. Memory devices slower than the computer main memory (e.g., hard disk) must work together to bring the records in the desired ordering. The combination of sorting input subsets in memory followed by an external merge process is the basic recipe of an external sorting algorithm.

A *record* is a basic unit for grouping and handling data. It can represent one single integer number or a more complex data structure, such as a personal address book composed by *name*, *telephone number* and *address* fields. A *key* is a special field (or subset of fields) used as criterion for the sort order. A *value* is the data associated to a specific key, i.e., the rest of the record. Whether key and value are disjoint or one is a subset of the other is irrelevant for our discussion.

2.1 In-memory sorting

In-memory sorting algorithms present different strategies and consequently different complexities, which in turn influence their efficiency. When deciding which in-memory sorting algorithm to use we consider the number of records in the input: for less than one hundred records any algorithm is equally efficient because the input is too small; but when dealing with a number of records between one hundred and the maximum number of records that still fit in main memory an $O(nlog(n))$ algorithm is preferrable

[10]. Before discussing the three main $O(nlog(n))$ sorting algorithms (namely quicksort, heapsort, and mergesort) we introduce some concepts to classify these algorithms. This categorization will help us later to decide which algorithm to use.

A *stable* sort is a sort algorithm which preserves the input order of equal elements in the sorted output; i.e., if whenever there are two records r and s with the same key and with r appearing before s in the original list, r will appear before s in the final sorted output. An *in-place* algorithm is an algorithm which transforms input using a data structure with a constant amount of extra storage space. The input is usually overwritten by the output as the algorithm executes.

We can further classify sorting algorithms with respect to its general operational method. These methods are:

1. *Insertion*: when examining record j assume all previously records $1 \dots j-1$ are already sorted. Then, move the actual record to the correct position, shifting higher ranked records up as necessary.
2. *Exchanging*: systematically interchange pairs of records that are out-of-order until no more such pairs exist.
3. *Selection*: repeatedly select the smallest (or biggest) record from the input. One by one we move the selected record to an output data structure, until the input is empty.
4. *Merging*: combine two or more sorted inputs into a single ordered one.
5. *Distribution*: records are distributed from their input to multiple intermediate structures which are then gathered and placed on the output.

The first algorithm we discuss is quicksort, which is an unstable, in-place, and exchanging sort. Perhaps quicksort is the most used sort algorithm in general, given that it is the algorithm used in the Microsoft .NET Framework [11] and in Oracle Java 7 [12]. Furthermore, it is the sort algorithm used by the Hadoop MapReduce framework. We also discuss heapsort, an unstable, in-place, selection sort, because its selective strategy backed by a heap will be useful when we talk about replacement selection. Finally, we introduce mergesort, a stable, non-in-place, merging sort. Its sort strategy based on merging sorted inputs is used in *external* sorting algorithms, thus reviewing the mergesort algorithm provides a good background to work on in the next section. A comprehensive study of sorting algorithms and techniques is found in [7].

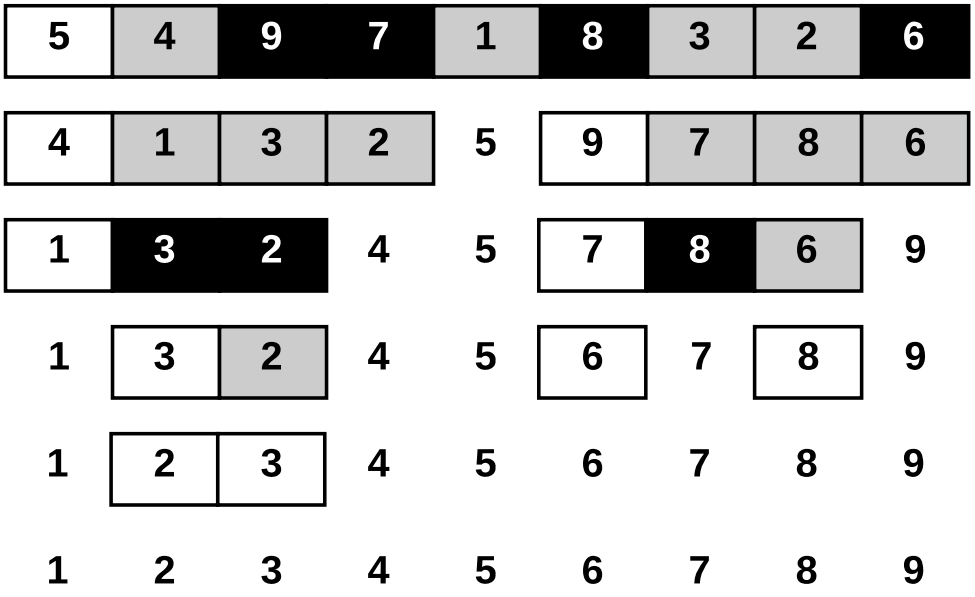


FIGURE 2.1: Quicksort procedure: white boxes represent the current pivot; grey boxes represent the records smaller than the pivot and the black boxes the records bigger than the pivot. Records outside any box are already in their final ordered position.

2.1.1 Quicksort

Quicksort is an exchanging sort algorithm that, on average, makes $O(nlog(n))$ comparisons to sort n records. It has a $O(n^2)$ complexity in the worst case, but it is usually faster than other $O(nlog(n))$ algorithms in practice [10]. Furthermore, quicksort's sequential and localized memory references exhibit superior address locality, which allows exploiting processor caching [13]. Quicksort implementations are usually unstable, but stable versions exist [14].

Quicksort work as follows: given an input of n records, we take a record r (called *pivot*) and move it to the final position p it should occupy in the output. To determine p, we rearrange the other records so that all records with a key value greater than that of r are "to the right" of p (i.e., their positions are greater that p) and the rest to the left. At this point, we have *partitioned* the input into two smaller sets: the set of values smaller than r, and the set of values greater than r. Now the original sorting problem is reduced to these two smaller problems, namely sorting the two smaller partitions. This is a typical application of the *divide and conquer* strategy, and we show an example in Figure 2.1. The algorithm is applied recursively to each subset, until all records are in their final position.

2.1.2 Heapsort

The heapsort algorithm is based on the heap data structure. The heap is usually placed in an array with the layout of a complete binary tree, which is "a binary tree in which every level, except possibly the deepest, is completely filled; at depth n, the height of the tree, all nodes must be as far left as possible" [15]. The binary tree structure is mapped to array indices: for a zero-based array, the root of the tree is stored at index 0. The left and right children of a node in position i are given by $2i + 1$ and $2i + 2$, respectively. Similarly, the parent of i is found $\lfloor (i - 1)/2 \rfloor$. During the construction of the heap structure, the data in the array is in an arbitrary ordered. The positions are then rearranged to preserve the *heap property*: the key of a node i is greater (or less) than or equal to that of its children. To restore the property, the *heapify* operation is invoked to rearrange the array elements. The heapify process can be thought of as building a heap from the bottom up, successively shifting nodes downward to establish the heap property. More details in building the heap structure are found in [7] and [10]. In Figure 2.2 we show a heap (constructed over an array) and its corresponding binary tree.

If we can transform an input set into a heap, we can sort the input simply by always selecting the root element for output until the heap is empty. This makes heapsort a selection-based algorithm. Differently from quicksort, heapsort has an $O(nlog(n))$ average case complexity, regardless of the order of the input dataset. However, it is slower than a tuned quicksort in practice [10].

The algorithm is divided into two parts: first, a heap is built out of the input data; and second, a sorted array is built by repeatedly removing the largest (or the smallest) record from the heap and inserting it into the sorted output. After each removal, the heap is reconstructed, i.e., a heapify is invoked.

Once all records from the input are inserted into the heap, we can start *selecting* the largest record. This record at the root of the tree is swapped with the last leaf node, which is located at the end of the array. The array is then *heapified* again, excluding the last position of the array. This process is repeated for all records in the heap until only the root node remains in the tree. At this point, the records are sorted. This process is illustrated in Figure 2.3, where circle nodes represent records to be sorted, square nodes represent already sorted records, shaded nodes represent the start of a new heapify operation, and finally dashed arrows represent a shift operation (used during the heapify process to rearrange the nodes to keep the heap property).

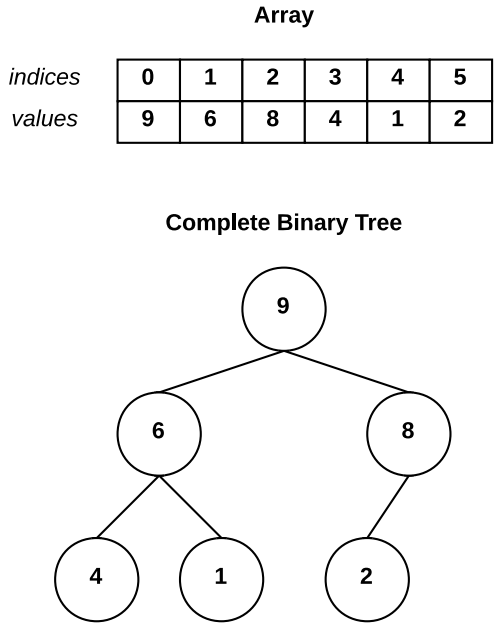


FIGURE 2.2: Array and equivalent complete binary tree

2.1.3 Mergesort

Mergesort is a merging-based sort algorithm where most implementations are stable, differently from quicksort. Mergesort has an average and worst-case performance of $\theta(nlog(n))$, however it demands $\theta(n)$ auxiliary space. *In-place* versions also exist but, due to the constant factors involved, they are only of theoretical interest [16].

Mergesort has two phases: first, we divide the unsorted input set into n subsets of size 1, which are obviously sorted. Second, we combine two or more ordered subsets into a single ordered one. This is called the *merge* phase. In the second step, we repeatedly compare the smallest remaining record from one subset with the smallest remaining record from the other input subset. The smallest of these two (assuming a *two-way* merge, where only two subsets are merged at a time) records is moved to the output. The process repeats until one of the subsets becomes empty. Figure 2.4 shows both of these steps. The output of a merge step can be an intermediary output, which will be further merged with other intermediary outputs or a final output, when all n subsets divided into the first step have been merged into one last sorted set of size n.

We can cover many possible sorting scenarios with these three sorting algorithms in our tool set. When auxiliary space is not an issue and we desire a tight upper bound for our worst case complexity, we shall use mergesort. When we need to sort in-place or we have knowledge about the unsorted input to explore cache locality, we shall use the quicksort. Heapsort will be of special interest when we detail the replacement selection algorithm

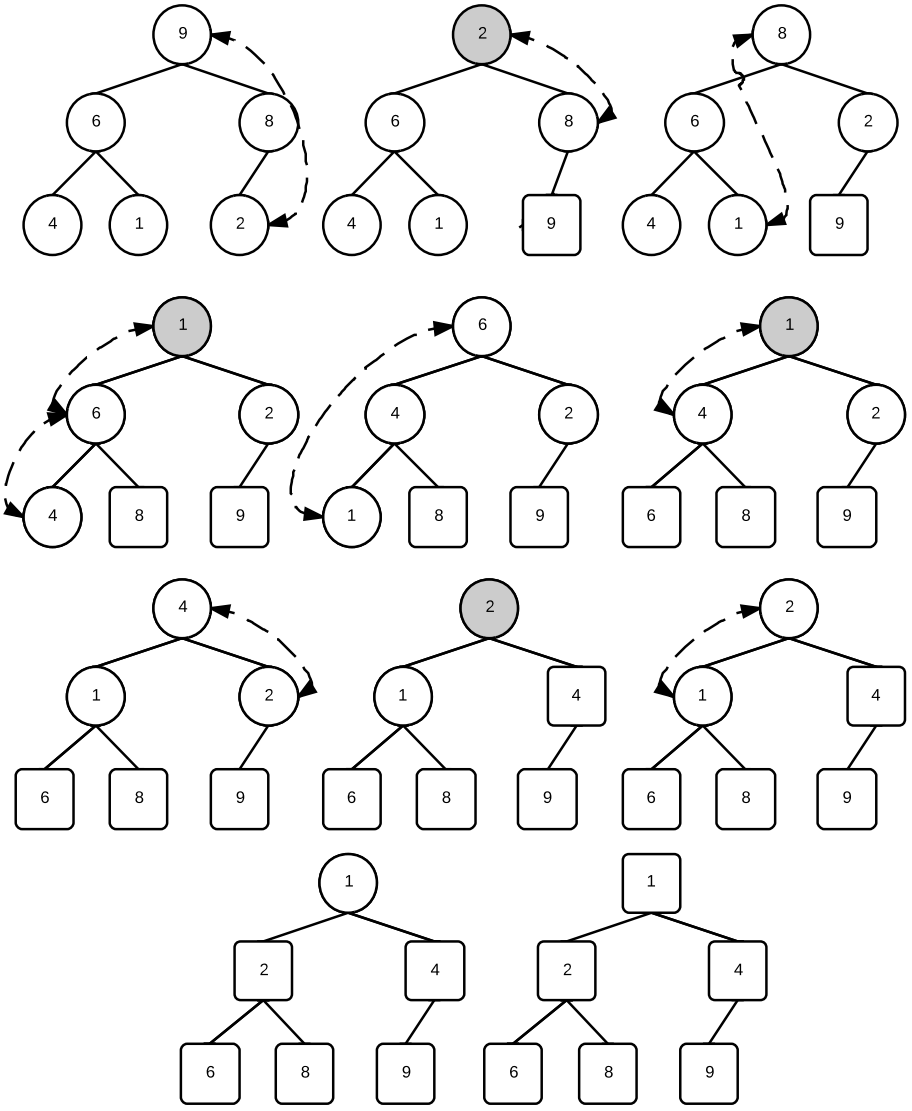


FIGURE 2.3: Heapsort procedure

in the following section, because it borrows the heap and the selection strategy from heapsort.

2.2 External memory sorting

We use *external* sort when the input dataset does not fit in the available main memory. From now on, we should use the terms disk, hard disk, and external memory indistinctly. The same is valid for memory, main memory, and internal memory. Nowadays, hard disks are the most common form of external memory device. If we compare them to main memory, hard disks present the following properties, as seen in [17]:

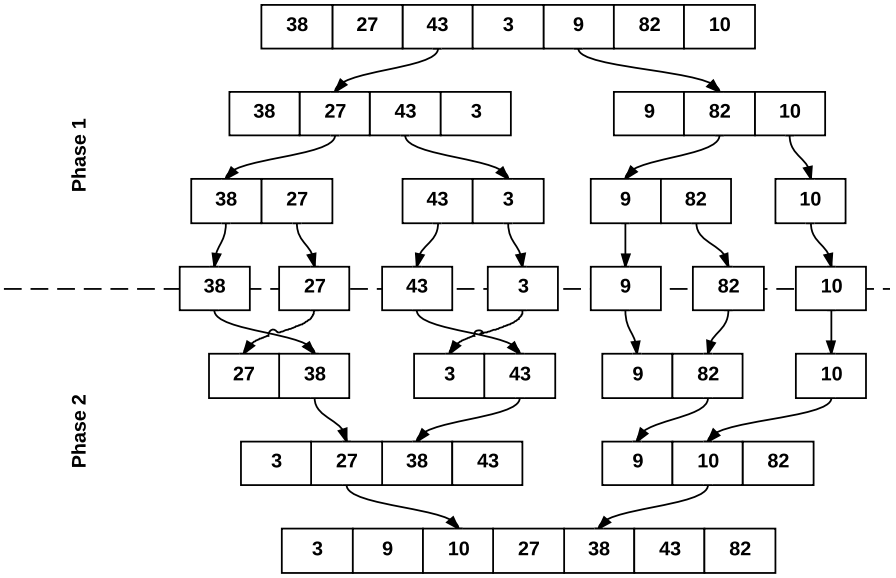


FIGURE 2.4: Mergesort procedure first dividing the input until subsets of size 1 and then merging the subsets into one single ordered output.

Parameter	**DRAM**	**NAND Flash**	**Hard Disk**	**PCM**
Density	1X	4X	N/A	2-4X
Read latency (granularity)	20-50ns (64B)	~25μs (4KB)	~5ms (512B)	~50ns (64B)
Write latency (granularity)	20-50ns (64B)	~500μs (4KB)	~5ms (512B)	~1μs (64B)
Endurance	N/A	$10^4 \sim 10^5$	∞	$10^6 \sim 10^8$

TABLE 2.1: Comparison of Storage Technologies. Adapted from [18].

- The amount of such storage on hard disks often exceeds the amount of main memory by at least two orders of magnitude;
- Hard disks are much slower (up to five orders of magnitude) because they have moving mechanical parts;
- It is beneficial to access more than one item at a time to amortize this time gap between main memory and hard disk.

Often, reading a page from the hard disk takes longer than the time to process it. Thus, CPU instructions stop being the unit to measure cost in the context of external sorting, and we replace it by the number of disk accesses—or I/O operations—performed. This difference makes algorithms designed only to minimize CPU instructions perform not so efficiently when analyzed from the I/O point of view. External memory algorithms must aim to minimize the number of I/O operations. In Table 2.1, we see that reading a $4KB$ page takes ~25μs on flash memory, while reading a $64B$ page from main memory

takes about 20-50ns. For disk reads, a 512B page takes ∼5ms. For comparison with CPU access speeds, in 5ms an average modern CPU can execute $\sim 10^5$ instructions.

Suppose we must sort a file of 5 million records residing on disk, but only 1 million can fit into internal memory at a time. A common solution is to bring each of the five subfiles to main memory, sort it using an internal sort algorithm, and store it back into disk. We call a sorted subfile of records a *run*. These runs are then merged by reading records of each run sequentially into a merge buffer, in a way that guarantees that the buffer contains at least one record of each run at all times. Once the buffer is full, the smallest record is moved to the output file. This process is repeated until there are no records left in all input runs and the buffer is empty. To allow for fast selection of the smallest record in the buffer, a heap data structure is used.

The above process is similar to the mergesort algorithm, but instead of occurring all at the same time in main memory, the input is consumed in chunks that fit in main memory. For a given input file f of size s records and main memory which accommodates m records, where $s > m$, a *multiway merging sort* can be described in two phases: *run generation*, where intermediary sorted subfiles, i.e., runs, are produced, and *merge*, where multiple runs are merged into a single ordered one. With respect to run generation, two strategies are worth discussing: one using quicksort and another using replacement selection as the in-memory sort algorithm.

2.2.1 Run generation with quicksort

Run generation based on the quicksort algorithm is a simple, nevertheless effective, strategy to create the sorted subfiles. We repeat the following steps until the input file is empty:

1. Read records from f into main memory until the available main-memory buffer is full;
2. Sort the records in main memory using quicksort;
3. Write them into a new temporary file (run).

If we assume fixed-length records such that m records fit in main memory, this process is repeated $\frac{s}{m}$ times, resulting in $r := \frac{s}{m}$ runs of size m stored in the disk.

2.2.2 Run generation with Replacement Selection

The replacement selection technique is of special interest because the expected length of the runs produced is two times the size of available main memory ($2m$ in the example above). This estimation was first proposed by E.H. Friend in [19] and later described by E.F. Moore in [20], and is also described in [7]. In real world applications, input data is usually not random (i.e., it often exhibits some degree of *pre-sortedness*). In such cases, the runs generated by replacement selection tend to contain even more than $2m$ records. In fact, for the best case scenario, namely when the input data is already sorted, replacement selection produces *only one* run.

Given a set of tuples $\langle record, status \rangle$, where *record* is a record read from the unsorted input and *status* is a Boolean flag indicating whether the record is *active* or *inactive*. Active records are candidates for the current run, while inactive records are saved for the next run. The idea behind the algorithm is as follows: assuming a main memory of size m, we read m records from the unsorted input data, setting its status to active. Then, the tuple with the smallest key and active status is selected and moved to an output file. When a tuple is moved to the output (*selection*), its place is occupied by another tuple from the input data (*replacement*). If the record recently read is smaller than the one just written, its status is set to inactive, which means it will be written to the next run. Once all tuples are in the inactive state, the current run file is closed, a new output file is created, and the status of all tuples is reset to active.

Step	Memory contents				Output
1	503	087	512	061	061
2	503	087	512	908	087
3	503	170	512	908	170
4	503	897	512	908	503
5	(275)	897	512	908	512
6	(275)	897	653	908	653
7	(275)	897	(426)	908	897
8	(275)	(154)	(426)	908	908
9	(275)	(154)	(426)	(509)	(end of run)
10	275	154	426	509	154
11	275	612	426	509	275

TABLE 2.2: Run generation with replacement selection

We introduce another example from [7] in Table 2.2 to explain in detail the replacement selection algorithm. Assume an input dataset consisting of twelve records with the following values: *061, 512, 087, 503, 908, 170, 897, 275, 653, 426, 154, 509* and *612*. We represent the inactive records in parentheses. A discussion of efficient ways of selecting the smallest record (e.g., using a heap) is postponed to Chapter 4.

In step 1, we load the first four records from the input data into the memory. We select 061 as the smallest then output and replace it for 908. The smallest now is 087, which we move to the output and replace with 170. The just-added record is also the smallest in step 3, so we move it out and replace it with 897. Now we have an interesting situation: when we replace the record 503, the record read from the input is 275, which is *smaller* than 503. Thus, since we cannot output 275 in the current run, we set it as inactive—a state which will be kept until the end of the current run. Steps 6, 7, and 8 proceed normally until we move out record 908, which is replaced by 509. At this point, in step 9, all records in memory are inactive. We close the current run, revert the status of all records to active, and continue the algorithm normally.

In this small example, we produce a first run with twice the size of the available memory and a second smaller one. In Table 2.2, steps 12, 13, and 14 will output *426, 509,* and *612*. We calculate this by $(tr)modulo(2m) = (12)modulo(2 \times 4) = 5$.

2.2.3 Comparison

Replacement selection is a desirable alternative for run generation for two main reasons: first, it generates longer runs than a normal external-sort run-generation procedure. As Knuth remarks in [7] "the time required for external merge sorting is largely governed by the number of runs produced by the initial distribution phase". A second advantage of this algorithm is that reads and writes are performed in a continuous record-by-record process, and as such can be carried out in parallel. This is particularly advantageous if a different disk device is used for writing runs. In this case, the I/O cost of run generation is half of that observed when using quicksort. Furthermore, heap operations can be interleaved with I/O reads and writes asynchronously.

A potential disadvantage of replacement selection compared to quicksort is that it requires memory management for variable-length records. When a record is removed from the sort workspace, the free space it leaves must be tracked for use by following input records. If a new input record does not fit in the free space left by the last selection, more records must be selected until there is enough space. On the other hand, the smallest records generate fragmented space. In quicksort, such memory management is not necessary, because the sort procedure is carried out in-place and one memory-sized chunk at a time.

These characteristics will be evaluated empirically in detail in Chapter 4.

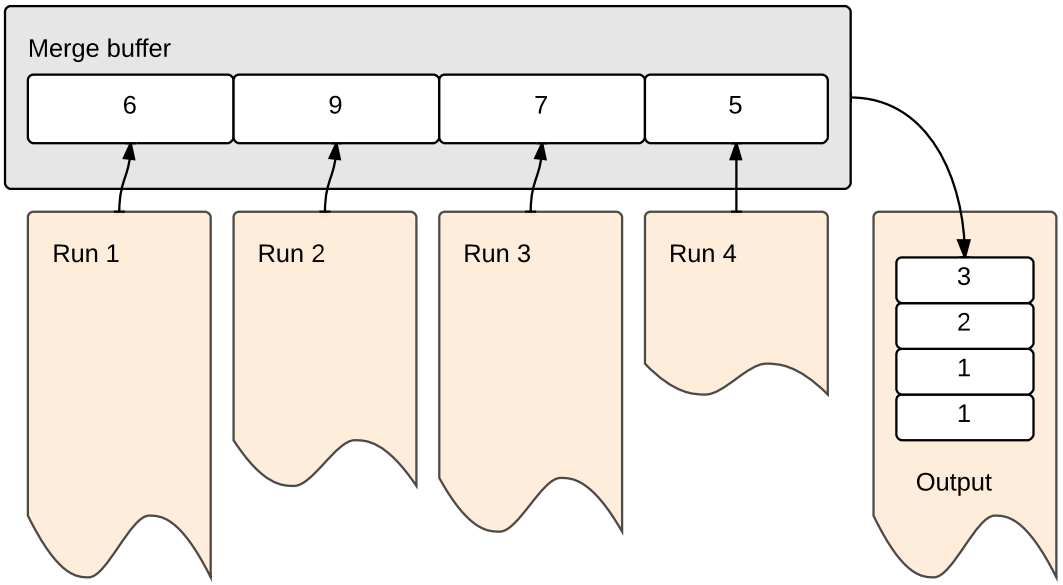


FIGURE 2.5: Merging four runs into one output file.

2.2.4 Merge

We turn our attention now to the second phase of external sorting, namely the *merge phase*. The objective of this phase is to create a final sorted file from the existing runs. Figure 2.5 shows a possible execution of the merge phase. In this example, four runs are generated from the initial unsorted input file. These runs are being read, record by record, into a merging buffer in main memory. The smallest record in the buffer is the one with key value *5*, and is the next one moved to the output file, which in turn is a new run. Then, we move a new record from the top of *run 4* into the place where record *5* was located. The execution repeats until only one run is left, which is the sorted output.

In order to select the smallest current record, at least r comparisons must be performed, since this is the number of runs produced. In [7], Knuth advises making this selection by comparing the records against each other (in $r-1$ comparisons) only for a small number of runs. When the number of records is larger than a certain threshold (Knuth says 8 in [7]), the smallest record can be determined with fewer comparisons by using a *selection tree*. In this case, a heap data structure can be employed, in the exact same way as done in run generation with replacement selection. With a selection tree we need only $log(r)$ comparisons, i.e., the height of the tree. Consider now an example of a four-way merging with a two-level selection tree in Figure 2.6 (adapted from [7]). We replace the smallest record by its successor record in its run at every step after this, and update the tree accordingly.

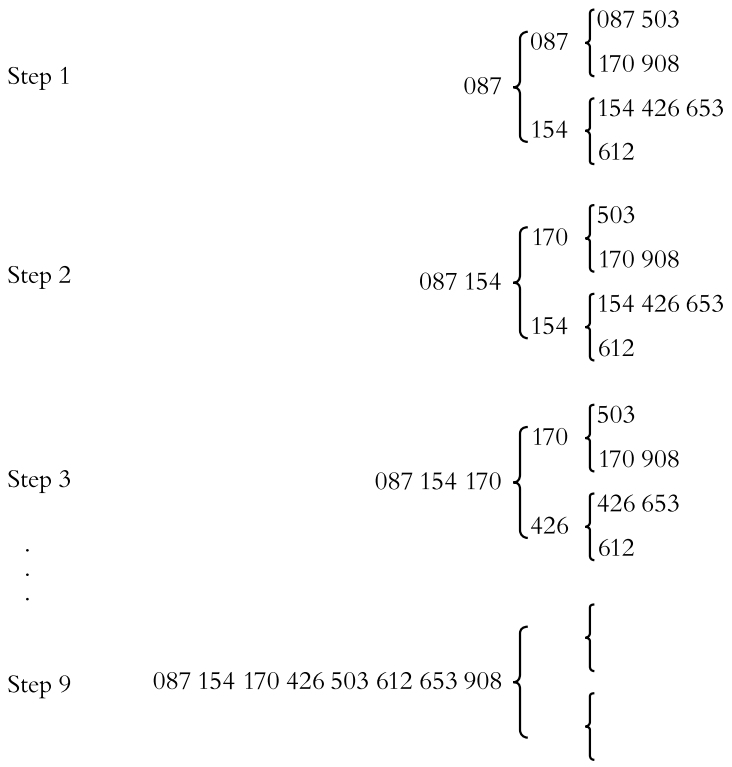


FIGURE 2.6: Four-way merging with a two-level selection tree

Complete the tree by executing the necessary comparisons in the first step. Replace the smallest record by its successor record in its run at every step after this, and update the tree accordingly. This two-level selection tree yields 087 as the smallest record in the first step. Move the record to the output and update the three positions where it appears in the tree. Step 2 outputs 154 as smallest record, and the three positions containing 154 are updated. We repeat this process of *selecting* and *replacing* while the runs are not empty.

A second improvement over the naïve procedure presented is to take advantage of read and write buffers. Given r runs and a main memory of size m, one read buffer of size $\frac{m}{r+1}$ can be used for each input run. This way, there is still $m - (\frac{m}{r+1})r$ space left for a write buffer. Figure 2.7 shows an example of this idea: the main memory can hold 20 records at time; there are four reading buffers (one for each produced run) of 4 records each and one writing buffer capable of holding also 4 records at time. Select the smallest record in memory from the top of the reading buffers (represented by the hexagons) and move it to the write buffer (represented by the shaded rectangles inside the main memory). When a read buffer is empty read the next top four records from the corresponding run. When the write buffer is full, all records contained in it are written to the output file.

The second phase of the external sort executes only $\theta(n)$ I/O operations. Despite this good result, the algorithm cannot sort inputs of arbitrary size as it is. If the first phase

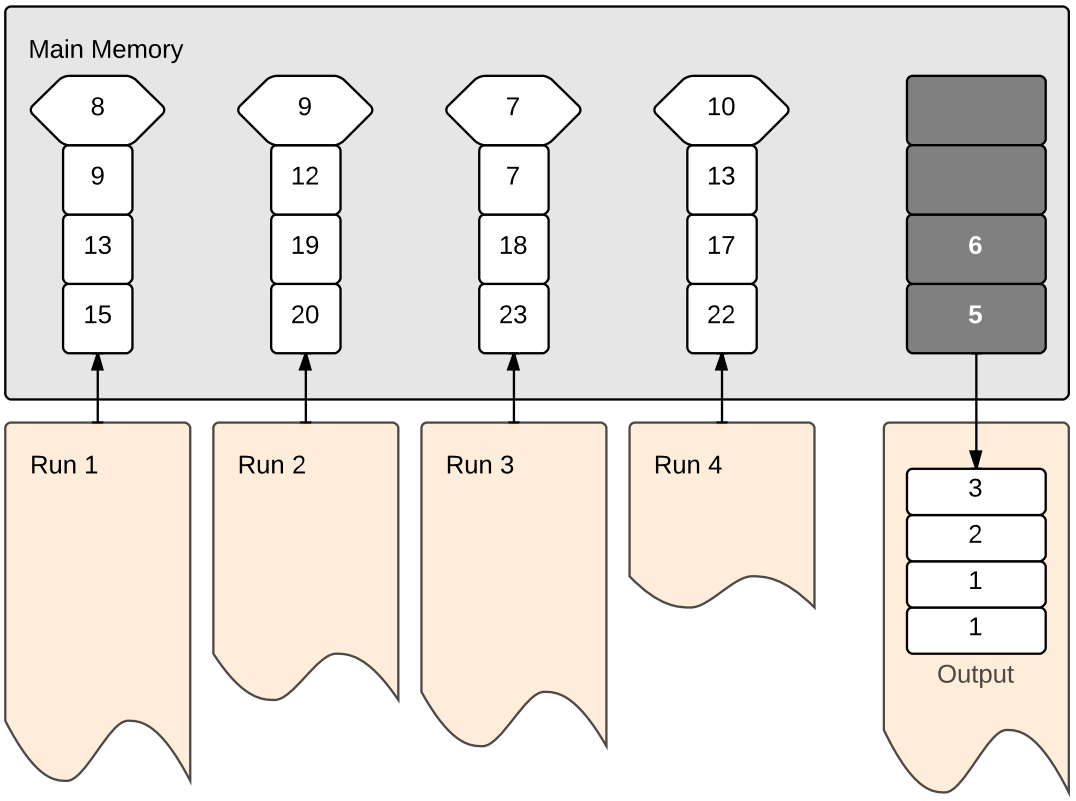


FIGURE 2.7: Merging four runs into one output file using read and write buffers.

of the algorithm produces more than $m-1$ runs ($\frac{s}{m} > m-1$), then we cannot merge these runs in a single step. A natural solution for this limitation is to repeat the merging procedure on the merged runs, producing a *merge tree*. At each iteration we merge $m-1$ runs into a new sorted run. The result is $\frac{s}{m} - (m-1) + 1$ runs for the next iteration—the total number of runs minus the merged runs in this turn plus the new merged run.

At each iteration, we call the total number of possible merges at the same time the merge *factor*. It is constrained by the size of available main memory: given by $\frac{s}{m} - 1$. For a total number of runs r and assuming that $r > factor$, the first iteration should attempt to bring r to be divisible by the factor, to minimize the number of merges. The goal is to reduce the number of runs from r to factor, and then to 1 (the final output). In [21], the number of merges executed in the first iteration is:

$$factor_0 = ((r - factor - 1) modulo (factor - 1)) + 2^1 \tag{2.1}$$

Figure 2.8 show this merging strategy in practice for $r = 12$ and $factor = 6$. In the first iteration we should merge $((12 - 6 - 1) modulo (6 - 1)) + 2 = (5) modulo (5) + 2 = 2$

[1]The author distinguishes between *final factor* and *normal factor*. In the query processing context, some operations may consume the output of other operations, and memory must be divided among multiple final merges. We shall not make such distinction in this text.

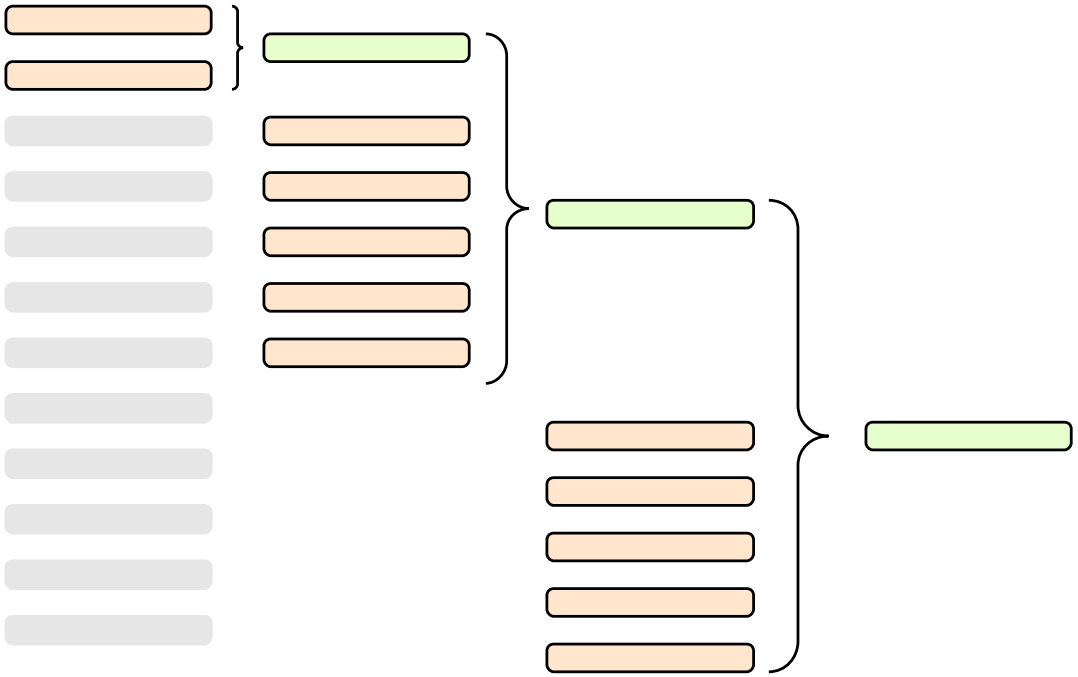


FIGURE 2.8: Merging twelve runs into one with merging factor of six.

runs. Other merging strategies, such as cascade and polyphase merges, exist and can be found in [7].

The analysis and implementation presented in Chapters 4 and 5 will focus exclusively on the run generation phase. Since the algorithm used for merging is independent of what is used for run generation, we evaluate sort performance using the same merge algorithm, namely the one implemented by the Hadoop framework.

Chapter 3

Hadoop

3.1 Motivation

According to a 2008 study by International Data Corp [22], since 2007 we produce more data than we can store. This huge amount of data was conventionally named big data. Thus the challenge nowadays is not the complexity of the problems to process, but the amount of information to take into account when doing so. In order to solve these data-problems while addressing the inherent problems of distributed computing at the same time, a radical new programming model for processing large data sets with a parallel, distributed algorithm on a cluster—called *MapReduce* [23]—was conceived. A MapReduce program is defined by two functions: a *map* and a *reduce* function. The map function emits records as intermediate key-value pairs, where normally one key is associated to a list of values. The reduce function merges all related intermediate values with the same key to an output value. The MapReduce framework addresses issues as parallel execution, data distribution, and fault-tolerance transparently.

Hadoop is an open source and widely used implementation of MapReduce. It is a software framework for storing, processing, and analyzing big data. Hadoop partitions large data files across the cluster using HDFS (Hadoop Distributed File System). Data replication increases availability and reliability: if one machine goes down, another machine has a copy of the required data available. The distinction between processing nodes and data nodes tends to disappear as approximately all nodes in the cluster can store and process data. Distributed programs are simpler to write because remote procedure calls are transparently handled by Hadoop. The programmer only writes code for the high-level map and reduce functions. Fault tolerance is achieved by re-assigning failed executions to a different node; nodes which recover can rejoin the cluster automatically.

3.2 Running a MapReduce job

In this section, we overview Hadoop's architecture and present the entities involved during the execution of a MapReduce job in Hadoop. This overview will provide the basis for a detailed analysis of map and reduce tasks in the following sections.

Figure 3.1 (adapted from [24]) shows the whole process. A *Client* submits a MapReduce job to the `JobTracker`, which is responsible for coordinating job execution. The map and reduce functions are spawned as *tasks* to each computing nodes in the cluster, each one running the `TaskTracker` service. The numbered arrows correspond to the following steps:

1. Run job: creates a new instance of the `JobClient` class;
2. Get new job ID: asks the `JobTracker` for a new ID for the current job;
3. Copy job resources: resources include the job JAR file (a Java package containing the program to be run), a configuration file, and the computed input splits[1]. An input split is a part of the input that is processed by a single map [24]. They are copied to a directory named after the job ID on the `JobTracker`'s file system;
4. Submit job: after the `JobClient` finished all preparation steps, it tells the `JobTracker` that the job is ready for execution;
5. Initialize job: the `JobTracker` will enqueue the incoming job, and schedule it according to its current load;
6. Retrieve job splits: this step is necessary so the `JobTracker` can determine the number of map tasks it should create (one for each split). Reduce tasks are created conform configured;
7. Heartbeat: more than just telling the `TaskTracker` is alive, the heartbeat messages tell the `JobTracker` whether the `TaskTracker` is ready to execute more tasks. If so, the `JobTracker` communicates that through a special return value in the message;
8. Retrieve job resources: the `TaskTracker` brings configuration file and the job JAR file and creates a new TaskRunner object;
9. Launch: the new TaskRunner launches a new JVM;
10. Run: the child process keeps the `TaskTracker` updated about the task status until its execution finishes.

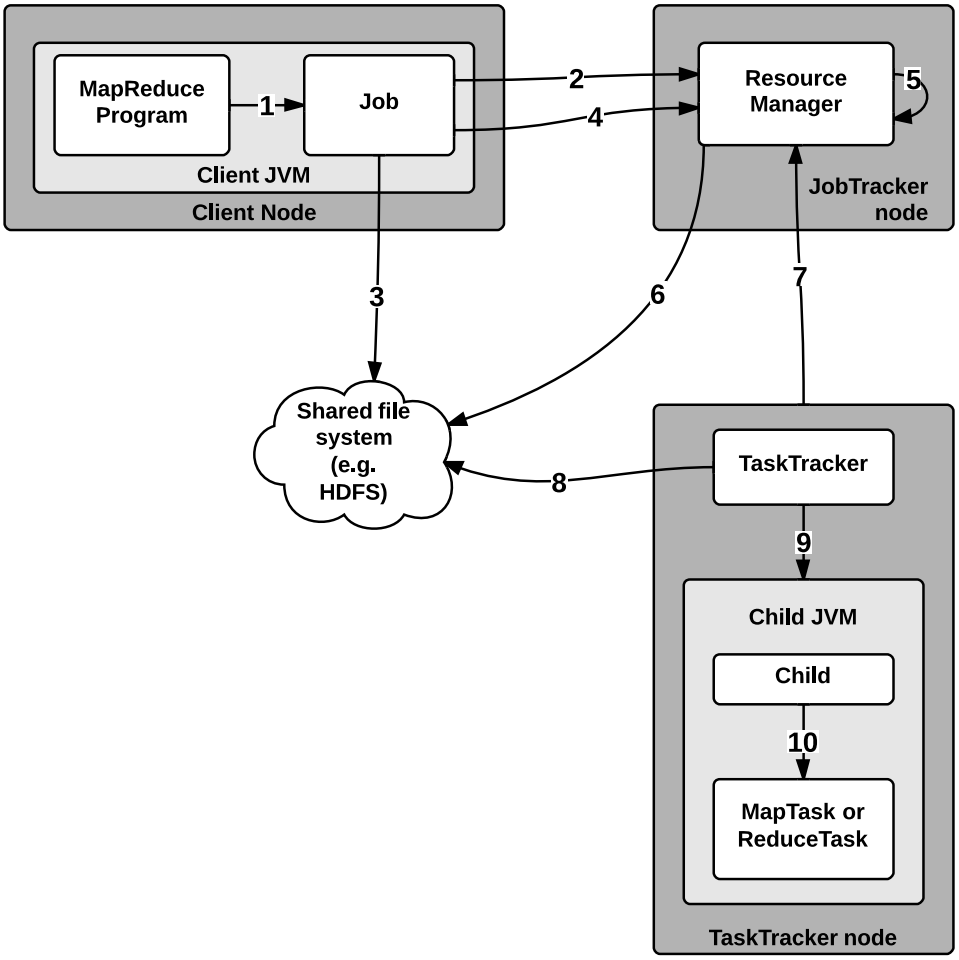


FIGURE 3.1: How Hadoop runs a MapReduce job.

After the job starts to run, the `TaskTracker`'s child process will keep consuming input splits to feed map tasks. When these tasks are finished, their results are stored locally, on the node's hard disk and not on HDFS. These results are intermediary key-value records which the reduce phase will bring together.

3.3 Map and Reduce tasks

We now *zoom* inside the *MapTask* and *ReduceTask* components in the `TaskTracker` node of Figure 3.2. Internally, a map task is responsible for more than only running the map function specified by the programmer. In order to make its intermediary results available for the reduce phase, it must organize and prepare these temporary results in a format that can be consumed by reduce tasks.

[1]T. White states in [24] that "If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program".

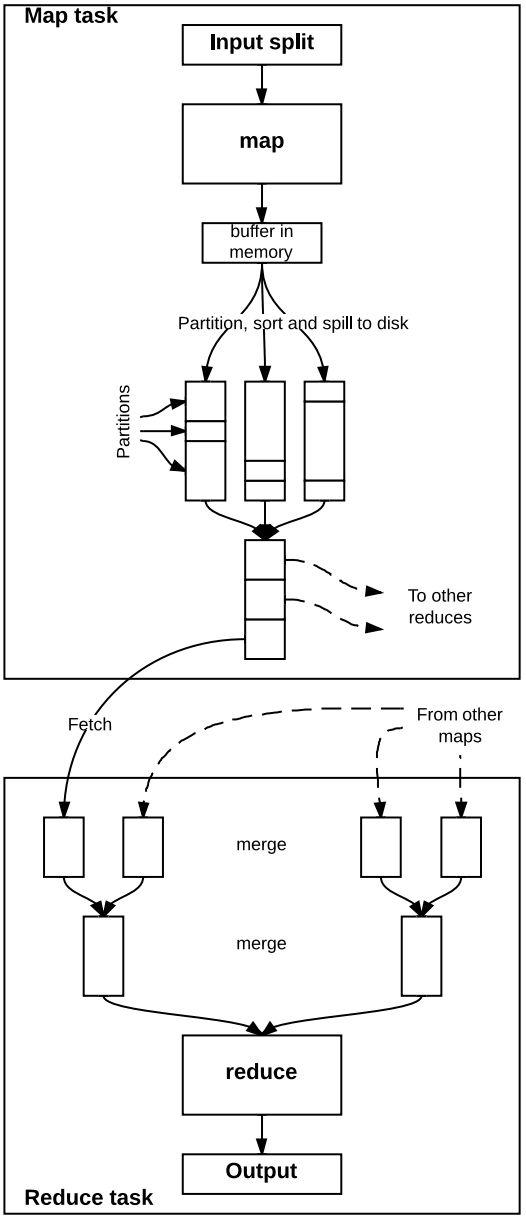


FIGURE 3.2: Map and reduce tasks in detail.

The following process happens pipelined, i.e., as soon as one step finishes the next can start using the output emitted by the former. The map function emits records (key-value pairs) while it is processing its input split, and this records are separated into partitions corresponding to the reducers that they will ultimately be sent to. However, the map task does not directly write the intermediary results to the disk. The records stay in a memory buffer until they accumulate up to a certain minimum threshold, measured as the total bytes occupied in the *kvbuffer*; this threshold is default configured as 80% of the *kvbuffer* size. When the buffer reaches this threshold, before the map task flushes the records to disk, it sorts them by partition and by key. When the records are sorted, the map task finally writes them to the disk in a file—called *spill*. Every time the memory

buffer reaches the threshold and the map task flushes it to the disk a new spill is created. Because the input split normally is bigger than the memory buffer, when the map task has written the last key-value, there could be several spill files. Naturally, the map task merges these spills into one single file. Just like the spill files, this final output file is ordered by partition and within each partition by key.

MapReduce ensures that the same reducer receives all output records containing the same key. Figure 3.2 represents this by the dashed arrows going from map to reduce tasks. The attentious reader may ask how do reducers know which `TaskTracker`s to fetch map output from? The answer is in the heartbeat communication system. As soon as a task finishes, it informs its parent `TaskTracker`, which will forward the notification to the `JobTracker`. After the completion of a map task, each reduce tasks will copy its assigned slice into its local memory. However, as long as this *copy* phase is not finished, the reduce function may not be executed, since it must wait for the complete map output.

Normally, the copied portion does not fit into the reduce task's local memory, and it must be written to disk. Once all map outputs are copied, the *merge* phase begins. As we noted in Section 2.2.4, if we have more than $m - 1$ map task outputs, the reduce cannot merge the intermediary results of all maps at the same time. The natural solution is to iteratively merge these spills, as we illustrated in Figure 2.8. We shall give more details about the Hadoop merging strategies in the following sections. Once we have a single merged file, the reduce function is finally called for each key-value of this file and outputted to the shared file system, HDFS.

This whole infrastructure is necessary to accomplish the design goals of MapReduce: distribution, parallelism, scalability and fault tolerance. The user code (map and reduce functions) is a small part in the whole process. Since this work is focused on run generation for external sorting, we shall concentrate on the map task, which is where sorted runs are initially produced.

3.3.1 The Map Side

While a map task is running, its map function is emitting key-value records to an in-memory buffer class called `MapOutputBuffer`[2]. If not explicit specified, all the entities discussed during this section are located inside this class. In the previous section we stated that these records are separated into partitions corresponding to the reducers

[2]This is true when the MapReduce job has at least one reducer. When there is no reduce phase, a `DirectMapOutputCollector` is used which directly writes the records to the disk as the job's final output.

that they will ultimately be sent to. This is done by a *partitioner* component, which assigns records output by the map tasks to reducers based on a partitioning function. This function can be customized by a user-defined partitioning function, but T. White states in [24] that "normally the default partitioner—which buckets keys using a hash function—works very well".

After the record receives a partition, the *collector* adds the record to an in-memory buffer: an unordered key-value buffer (*kvbuffer*). Hadoop keeps track of the records in the key-value buffer in two metadata buffers for accounting (*kvindices* and *kvoffsets*). *Kvbuffer* is a byte array that works as the main output buffer: the keys and values of the records are serialized into this buffer. The amount of memory reserved for the *kvbuffer* is calculated by:

$$io.sort.mb - (\frac{io.sort.mb \times io.sort.record.percent}{16}) \tag{3.1}$$

Hadoop's default value for the *io.sort.mb* property is 100MB and for the *io.sort.record.percent* is 0.05%. This configuration yields a *kvbuffer* of $104,529,920$ bytes. The accounting buffers (or metadata buffers) are auxiliary data structures used by Hadoop to efficiently manipulate the key-value records without actually having to move then in the *kvbuffer*. They are composed of two integer arrays. The first array, *kvindices*, has three integer values (4 bytes each): *partition*, *key start* (a pointer to where the key starts in the *kvbuffer*), and *val start* (a pointer to where the value starts in the *kvbuffer*). The second array, *kvoffsets*, manages a further level of indirection pointing to where each $\langle partition, keystart, valstart \rangle$ tuple starts in the *kvindices* (also 4 bytes integer pointer). An illustration of how these buffers are used is given in Figure 3.3, which shows 4 key-value pairs already serialized in the *kvbuffer*, where each key is 8 bytes long and each value is 11 bytes long. The *kvindices* buffer keeps track of where each record's key and value starts and also to which partition that record belongs to. Finally in the *kvoffsets* buffer there is second round of pointers indicating where each of the $\langle partition, keystart, valstart \rangle$ tuples are located in the *kvindices*. The number of records in the metadata buffers is calculated by:

$$\frac{io.sort.mb \times io.sort.record.percent}{16} \tag{3.2}$$

The default configuration values result in a capacity to store metadata of $327,680$ records, i.e., 5% of the reserved space. We call attention for the obvious although important difference between the buffers: the *kvbuffer* is measured in bytes, while accouting buffers are measured in records. As we shall see in the following paragraphs, the spill procedure is triggered when one of these buffers—the *kvbuffer* or the *kvindices*—reaches

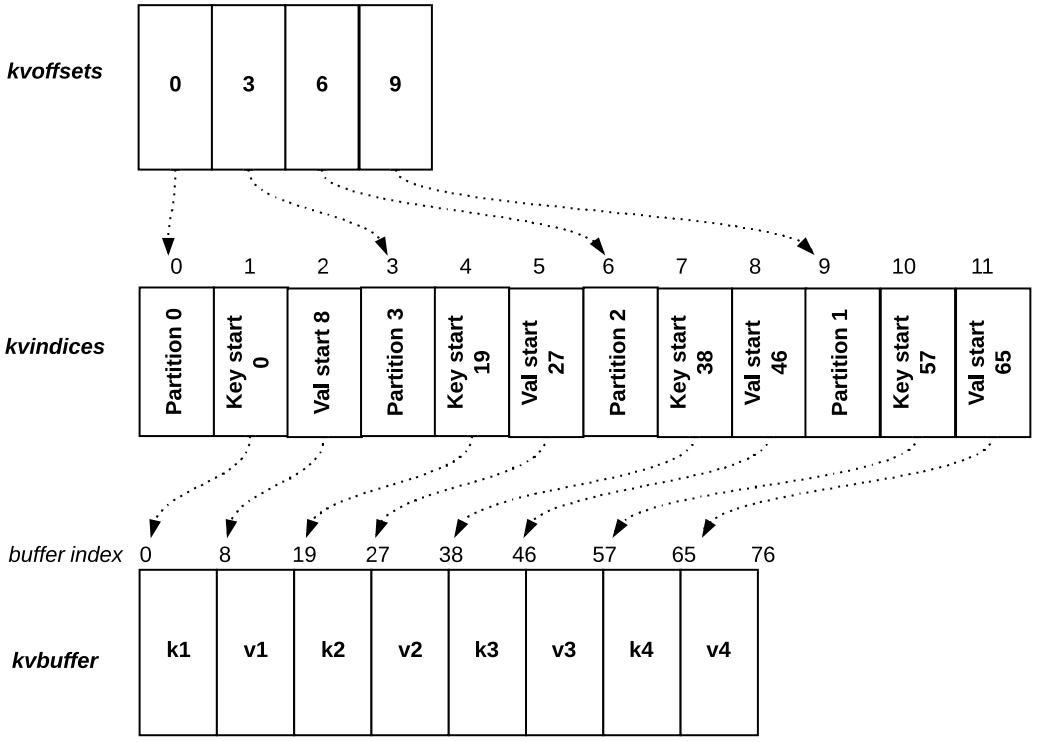


FIGURE 3.3: In-memory buffers: key-value (kvbuffer) and metadata (kvindices and *kvoffsets*).

a configurable threshold. The default configuration expects a record with average size around 320 bytes. However, if the average size is much smaller than the originally expected, the *kvbuffer* will never get full enough to trigger the spill procedure. But because the metadata buffer threshold is measured in number of records, and now more records fit in the *kvbuffer* at the same time (because they are smaller), *kvindices* will trigger the spill all the time. This is a waste of resources: we have an approximately empty byte buffer and a full accounting buffer to track the records in it. In order to avoid this misuse of the available space, given by the *io.sort.mb* property, we can simply increase *io.sort.record.percent*. To optimize this value, Hadoop provides useful statistics generated during job execution by means of *counters*. Two counters are useful here: *map output bytes*, the total bytes of uncompressed output produced by all maps in the job; and *map output records*, the number of map output records produced by all the maps in the job. Thus we can calculate the average size of a record by $\frac{MapOutputBytes}{MapOutputRecords}$.

The collector serializes the key-value records in the *kvbuffer*. It also adds metadata (the position where the key and value start in the serialized buffer) about *kvbuffer* to *kvindices* and adds metadata (where each metadata tuple $\langle partition, keystart, valstart \rangle$) about *kvindices* to *kvoffsets*. When the threshold of accounting buffers or record buffer

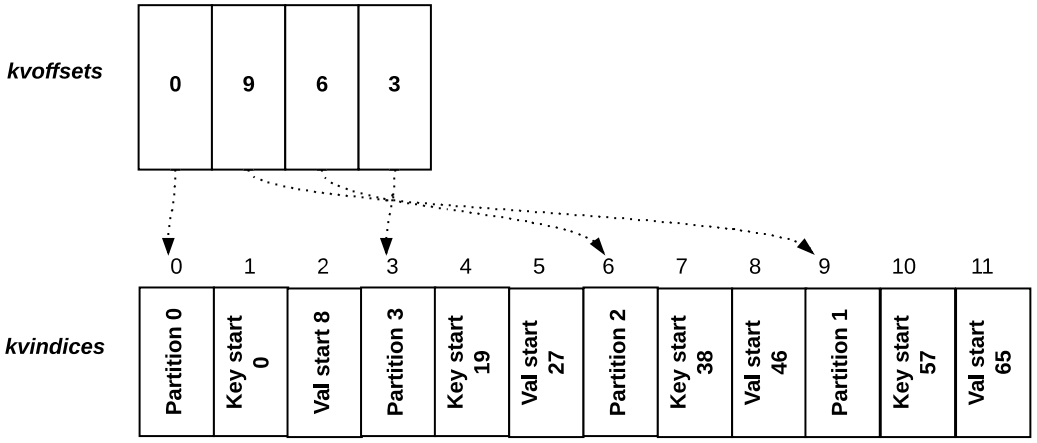


FIGURE 3.4: Kvoffsets buffer sorted by partition.

is reached, the buffers are spilled to disk. The *io.sort.spill.percent* configuration property determines the threshold usage proportion for both the key-value buffer and the accounting buffers to start the process of spilling to disk [24]. The default value of this property is 80%, which is used as a *soft limit*. The collector will continue to serialize and keep track of new records emitted by the map function while a spill thread is working until the buffers are full, when the *hard limit* is reached. The collector then suspends its activity then until the spill thread has finished.

The purpose of the second accounting buffer is to improve CPU performance. We mention in section 3.3 that records should be ordered by partition and, within each partition, by key. For the run generation inside map tasks, Hadoop uses quicksort. C. Nyberg et al. explore optimization techniques with respect to sorting in [13]. Two of these techniques include the minimization of cache misses and the sorting of only pointers to records rather than the whole records. They adopted quicksort in [13] because "it is faster because it is simpler, makes fewer exchanges on average, and has superior address locality to exploit processor caching". These techniques are also employed by Hadoop—the second accounting buffer holds the pointer values exchanged by the sort algorithm. This so-called *pointer sort* technique is better than a traditional *record sort* because it moves less data. When sorting *kvoffsets*, quicksort's *compare* function determines the ordering of the records accessing directly the partition value in *kvindices* through index arithmetic. But quicksort's *swap* function only moves data in the *kvoffsets*.

We return to Figure 3.3 as an example. The records in the *kvindices* buffer are not ordered by partition: these are 0, 3, 2 and finally 1. Initially, *kvoffsets* is sorted with respect to the order which the records where inserted into *kvindices*. Sorting it results in the state shown in Figure 3.4.

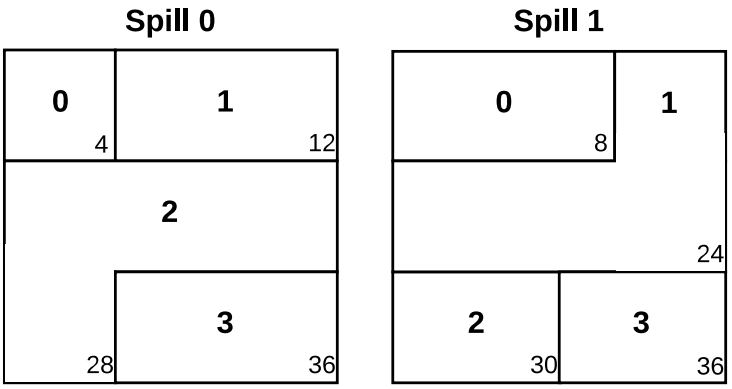
After sorting the in-memory buffer, as shown in Figure 3.4, the spill thread writes the buffer contents to disk. A new spill file is written every time the spill threshold is reached.

When the map function completes processing the input split and finished emitting the key-value records, one last spill is executed to flush the buffers. The map task then has several spill files in its local disk, which must be merged into a final output file that becomes available for the reducers. Hadoop implements an iterative merging procedure, where the property *io.sort.factor* specifies how many of those spill files can merged into one file at a time.

The map side merge work as follows: first an array with all spill files is obtained. In the case where there is only one spill file, that spill is the final output; otherwise, two final output files are created: one for the serialized key-value records (*final.out*), and another for pointing where each partition starts in the former (*final.out.index*). As a header, $numberOfPartitions \times headerLenght$ bytes are reserved in the beginning of both files. The $numberOfPartitions$ is equal to the number of reducers configured for the job; the $headerLenght$ is a constant value set in 150 bytes. First, an output stream is created to write the final file. Then, for each partition, a list of segments to be merged is created and, for each spill, a new segment is added to the segment list. Each segment correspond to a "zone" in each spill file corresponding to the current partition. Figure 3.5 illustrates the idea. We now call the merge method in the *Merge.class*, passing the segment list. Note: we are still inside the loop over partitions. This means that as many mergings as the number of partitions will be executed.

The merge procedure calculates the number of segments to merge in the first pass using equation 2.1. We reintroduce the term $factor$ (abbreviated to f), now in the context of Hadoop. Its value is configured through the *io.sort.f* property, and it corresponds to the maximum number of streams to merge at once. Assuming we have more than f segments, the first merge pass brings the total number of runs to an amount divisible by $f - 1$ (each pass transforms f runs into one) to minimize the number of merges which, in turn, minimizes disk access. We refer to the number of runs to be merged in the first pas as the *first factor*. All the subsequently passes will then merge f streams.

First-factor segments are copied from the original segment list to an auxiliary list (called *segments-to-consider*). Then, all segments in this list are added to a priority queue. The heap which implements the priority queue sort the segments ascending by size, thus smaller segments sort first. All records from the priority queue are written into a temporary file, and the heap is cleaned afterwards. This temporary file is a new, merged segment, which is added to the list of segments to be merged. After some clean-up routines are executed, the second round of merge starts.



Segments

Spill		Partition 0	Partition 1	Partition 2	Partition 3
	0	0 .. 4	4 .. 12	12 .. 28	28 .. 36
	1	0 .. 8	8 .. 24	24 .. 30	30 .. 36

FIGURE 3.5: Two spill files and the resulting segments for each partition.

At some point, there will be fewer segments than f, wich means the merge reached the last merge level. When this occurs, the segments are already in the heap and, instead of creating another temporary file, we simply return the priority queue back to the *MapTask.class*. Each segment is pulled from the priority queue and is written to the final file. Here, the loop around the partitions finally ends. Some housekeeping functions are executed, and the next iteration —with the next partition— can begin.

We show one example of a merge tree in Figure 3.6. Assume for a given partition (say 0) we built a segment list of size 20 (we have at least 20 spill files with records that belong to partition 0) and an *io.sort.factor* of 10. Equation 2.1 yields $((20-10-1) modulo (10-1))+2 \therefore (9 modulo 9)+2 \therefore 2$, which means we must merge two segments in the first round. Given the (sorted ascending by file length) segments list, we get the first two and merge them. Then, we merge the next 10 segments. At this point, we have 8 segments from the original list and 2 temporary ones, resulting from the previously merged segments. As expected, this sums exactly to 10 segments, which are now merged in the final file.

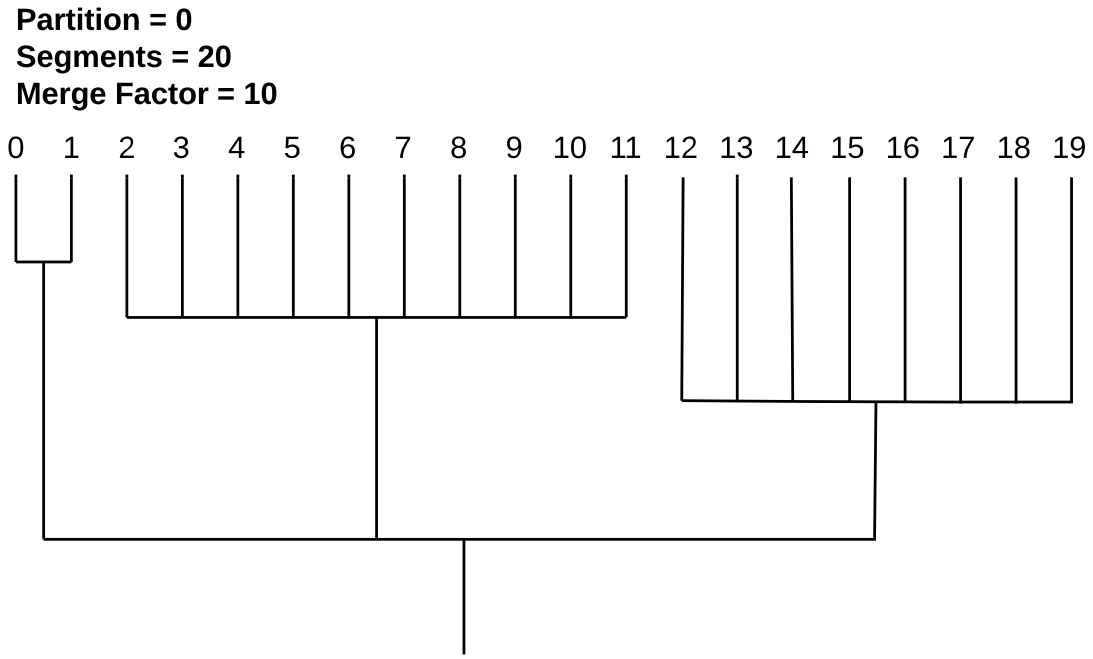


FIGURE 3.6: Merge tree with 20 segments and merge factor 10.

This concludes the map task, after which the `TaskTracker` sends a heartbeat informing the `JobTracker`. Next time a reducer consults the `JobTracker`, it will be informed that in a given `TaskTracker` there are map outputs ready to be copied. We close this chapter presenting figure 3.7, which summarizes the map-side processing we have discussed so far.

3.4 Pluggable Sort

The `MapOutputBuffer` class is Hadoop's default implementation of in-memory sort and run generation, which was fixed since its first version. However, the second stable version of Hadoop (2.x) enables the customization of this procedure through an interface called *pluggable sort*. It allows replacing the built-in run generation logic with alternative implementations. This means not only having the possibility to customize which data structures and buffer implementations to use, but also which sort algorithm. The pluggable sort was proposed by SyncSort [25] and discussed and develop in Apache's JIRA issue tracker under codename MAPREDUCE-2454 [26].

A custom sort implementation requires a *MapOutputCollector* implementation class, configured through the property *mapreduce.job.map.output.collector.class*. Since all pluggable components run inside job tasks, they can be configured on a per-job basis [27]. Likewise, custom external merge plugins can also be customized for reduce tasks.

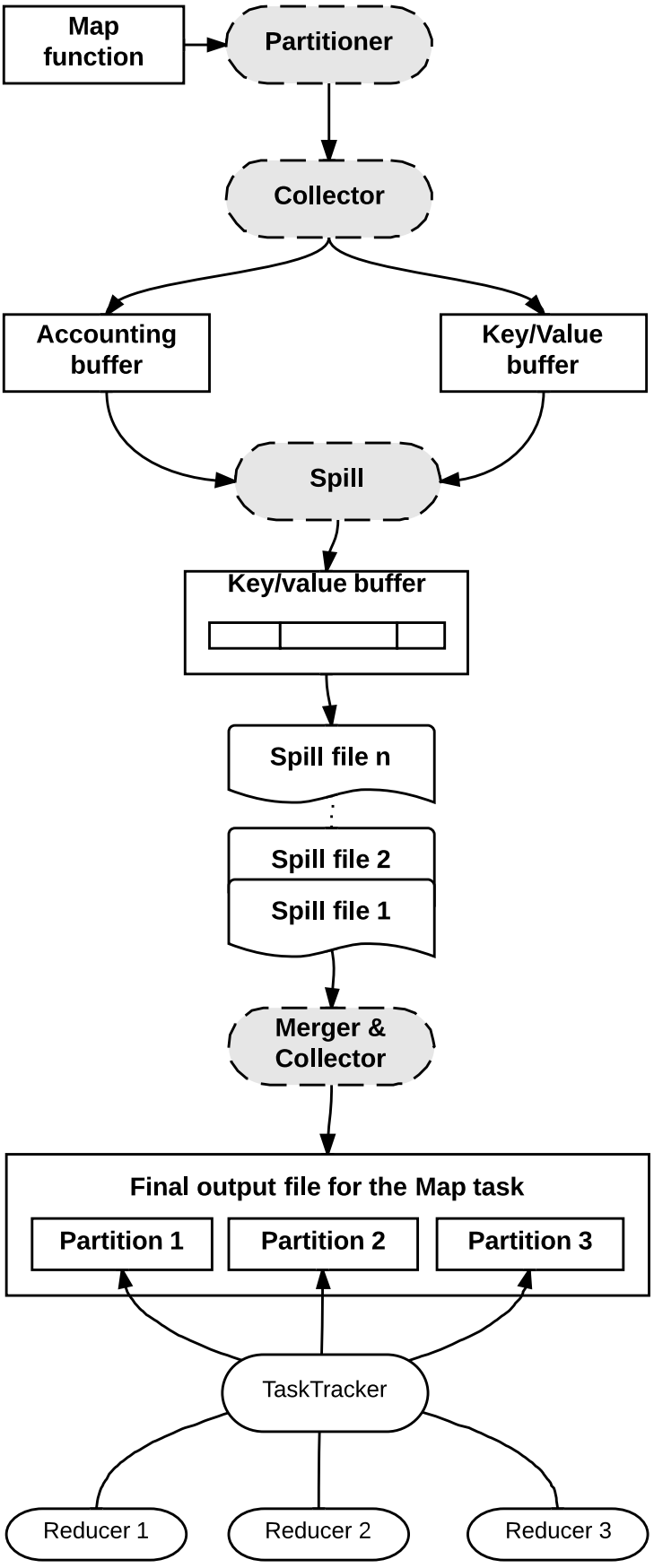


FIGURE 3.7: The map task execution.

Implementing the MapOutputCollector interface demands the implementation of three methods: `collect`, `flush`, and `close`. The `collect` method receives a key, a value, and a partition and is responsible for storing it into an internal buffer. The `flush` method is called when the map function finishes processing its input split and there are no more key-value pairs to emit. When the map task calls this method, it is giving the output buffer a chance to spill any record left in the buffer, and telling the buffer to merge all spill files (or, in the case there is only one spill, rename it). The `close` method simply performs some housekeeping procedures, closing streams and freeing class fields.

Using this extensibility mechanism, we implement run generation using the replacement-selection algorithm, as an alternative to the original quicksort. Because of Hadoop's own big data nature, having multiple spill files is the rule and not an exception. Thus, not only an alternative output buffer has to take care of sorting in-memory keys and partitions but also merging these multiples sorted spills into one single, locally-ordered file. It has to carefully consider data structures to store the records and algorithms to manage and reorder these records. It should be clear that this merge is only a local merge (performed by each map task). A second, cluster-wide merge performed in the reduce side will merge the locally-ordered files that each map task has processed. This global merge, however, is beyond the scope of this work, where we focus exclusively in the map side task sorting and merging. In the next chapter, we introduce our own replacement-selection-based implementation of `MapOutputBuffer`

Chapter 4

Implementation

This Chapter describes four implementation approaches for run generation in Hadoop using replacement selection. Starting with a naïve approach, where memory management is left under complete control of the Java virtual machine, each implementation introduces an addition in complexity in order to improve performance

During the discussion, we compare the current algorithm with Hadoop's original output buffer. Our objective in this Chapter is to construct an output buffer which can generate a small number of larger runs if comparable to Hadoop as fast as possible. The lesser number of spills created by replacement selection leads to a faster merger phase [28]. This suggests that if we decrease the difference in the run generation phase, possibly the total execution time of our custom buffer will be shorter than Hadoop's buffer execution time. The best trade-off between generating faster runs or generating less spill files is one of the questions this work tries to solve.

4.1 Development and experiment environment

The tests in this Chapter were executed in a machine with the following configuration:

- Architecture: Intel(R) Xeon(R) 64bits CPU X3350 @ 2.66GHz with 4GB of main memory and 465GiB hard disk;
- Operating System: Ubuntu 12.04 LTS Linux (3.0.0-12-generic-pae) i386 GNU/Linux
- Java Version: 1.7.0_25 OpenJDK

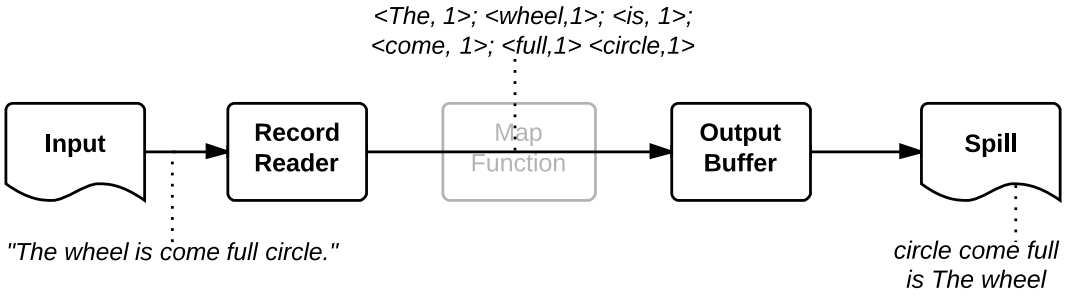


FIGURE 4.1: The setup of the tests performed in this Chapter. We feed the output buffer with records directly from the input reader.

In order to isolate the performance analysis to the run generation phase, we perform micro-benchmarks instead of complete Hadoop jobs. We show an example of the set up in Figure 4.1, only replacing the random strings for some pleasant text. Records are read directly from an input file and fed into Hadoop's *collect* method, simulating the output of the Map phase. The collect method triggers run generation and the merge of local output files, after which the Reduce phase starts. In our experiments, the process stops when runs are generated and timestamps are collected to measure run generation execution time.

The input is a 9GB file with randomly generated strings varying length between 80 and 160 bytes (one string per line). The in-memory buffer is 16MB, which yields a ratio of 562.5 between input and buffer size. The keys in our experimental job are the full strings, and the values are 1.

4.2 Record Heap

The Record Heap implementation is characterized by a simplistic and direct implementation of replacement selection, fully relying on the Java Virtual Machine (JVM) to perform memory management and on container classes of the Java Standard library. The `RecordHeap` class implents the `MapOutputCollector` interface, and is the class *plugged* into the Hadoop's job. Besides it, there are other two important classes: the `HeapEntry` class and the `NaiveReplacementSelection` class. The first class encapsulates the record's key, value, partition and current status (a Boolean flag). The former class implements the replacement selection algorithm and has the following main fields:

- A priority queue of `HeapEntry` objects: each $\langle key, value, partition \rangle$ tuple is encapsulated in a `HeapEntry` object and put into `PriorityQueue` instance, which is Java's default priority queue. In this implementation, both the record buffer and

the selection tree entities are implemented into one single object — this queue. As the priority queue is supported by a heap, we shall use both terms indistinctly;

- A `HeapEntry` comparator: records are sorted by status (*active* < *inactive*), partition, and key within the heap.
- A `HeapEntry` extra object: where we save the last output record, used to compare and decide if the incoming record belongs to the current run or not;
- A Boolean flag for the current active status: we detected that all the records are inactive when the status of the top element pulled from the `PriorityQueue` is different from the current status flag. If so, we simply set the current active status to the top element's status. This saves the work of iterating through the heap to flip all statuses.
- A writer: this is a synchronous writer to output the records as spill files. We open a stream when we create a new run and keep appending key and values into it. Hadoop creates index objects of where each partition begins and ends (begin plus length) in the spills (as shown in Figure 3.5). We made sure to mimic Hadoop's file format to be able to use the same merger later.

The process works as follows: as the map function emits new key/value pairs, the map task pushes them into the output buffer through the collect method. The collect method calls an *add* method in the `NaiveReplacementSelection` object which encapsulates the key, the value, and the partition in a new `HeapEntry` object and puts it into the heap. Java's default `PriorityQueue` implementation does not limit the number of elements it can contain[1], thus we control the maximum number of HeapEnty objects in the heap with a variable *max_buffer_size*. The priority queue will have a constant number of elements from the moment the buffer is full (i.e., the number of elements in the heap is equal to the buffer maximum size) for the first time until the moment when the map tasks call the flush method.

Once the heap is full, we have to make space for the incoming records. Pulling the priority queue yields the lowest possible record, which is written out. Then, the incoming record is compared to the saved record: if the new record is bigger than the saved record, it still belongs to the current run. Then, its status is set to active and the record is added to the heap. Otherwise, add it with inactive status. At any time, if the pulled record's status is not equal to the current active status, close the run, flip the current active status and start a new run.

[1]The `PriorityQueue` grows as needed, doubling it size when it is has less than 64 elements and increasing its size in 50% otherwise. It is bounded only by the heap size of JVM.

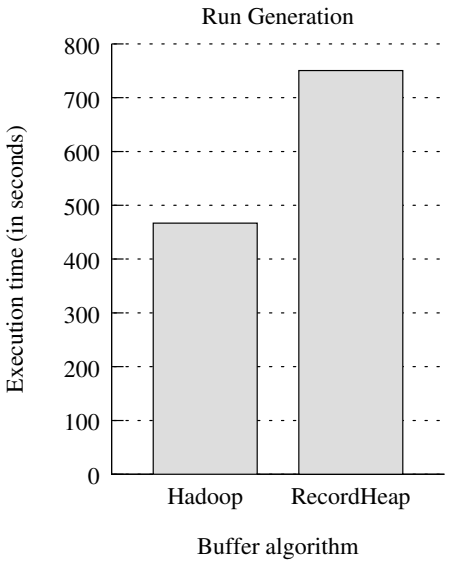


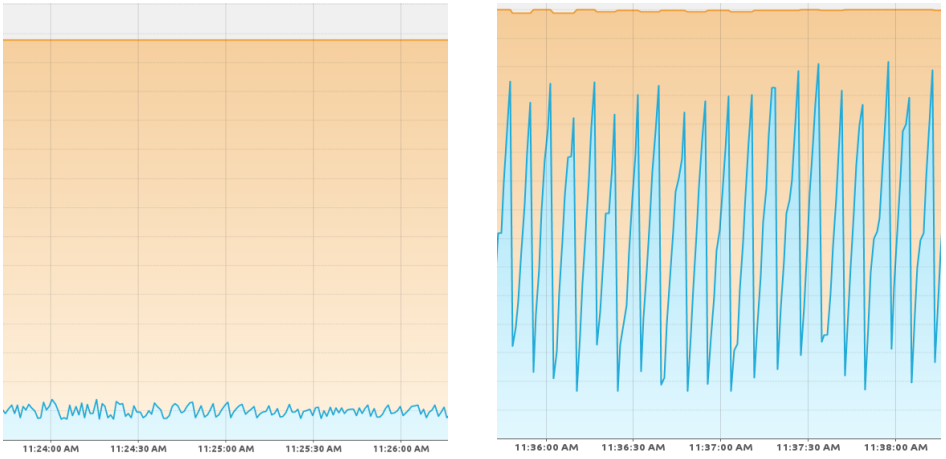
FIGURE 4.2: Comparison between Hadoop's `MapOutputBuffer` and our custom `RecordHeap` buffer.

This process repeats while the map function emits new key/value pairs to the collect method. Once the input is finished, the MapTasks calls `RecordHeap`'s flush method. The process to flush the priority queue is similar to the previous one. The just pulled record is compared to the saved record to decide if it belongs to the current run or not. If so, the record is outputted and the saved record is updated. If not, the run is closed and a new one is started. This executes while the priority queue is not empty. Figure 4.2 shows a comparison between Hadoop's default output buffer and the `RecordHeap` implementation, as we can see Hadoop's run generation is 60% faster, which is approximately 300 seconds. We shall now investigate what is the cause of this difference.

Both the key and the value in Hadoop are generic objects which may assume any of the job's configurable types, like `Text`, `LongWritable`[2], and `IntWritable`. Hadoop reuses this pair of objects when feeding the map function with data from the input split. The `RecordReader` class provides a record iterator with some extra capabilities, like parsing a CSV input for example. The map task uses a `RecordReader` to generate record key-value pairs, which it passes to the map function. The `RecordReader` instantiates a key object and a value object when the mapper function first asks for records. Writable types, in general, are backed up by a byte array and provide a *set* method, which copies the received set method's argument into the byte array. When these objects arrive to the collect method in the `MapOutputBuffer` class, they are serialized into the *kvbuffer* (i.e., their bytes are copied to the `MapOutputBuffer`'s byte array). When the mapper asks for more records from the `RecordReader`, the existing key and value objects are reused — the data read from the input is *set* into these objects. This saves Hadoop from creating new objects for each record read, which is not true for the `RecordHeap` implementation.

Recall that the buffer and the selection tree of the `RecordHeap` are one single object, an instance of Java's `PriorityQueue` class strongly typed with `HeapEntry` objects. Adding these recycled key and value objects to the buffer provokes an erroneous behavior: since

[2] `Writable` is the interface created to handle data serialization in Hadoop. It has two methods: one for writing to a DataOutput binary stream and one for reading from a DataInput binary stream [24]. We shall discuss serialization in more detail in a further Section.



(A) Memory monitor during execution of `MapOutputBuffer`.

(B) Memory monitor during execution of `RecordHeap`.

FIGURE 4.3: Java's heap space in two different scenarios. In foreground at the lower part of the figure there is the used heap, and the background there is the heap size.

the key and the value are instantiated in the `RecordReader` only once, their objects hold the same reference during the whole execution (until they are destroyed). Thus, every "new" record key-value pair added to our buffer points to the same object. This results in all `HeapEntry` instances in the `PriorityQueue` having the same key and value as the last added one. The natural solution to this problem is to create new objects every time the `RecordReader` reads data from the input. It turns out that the price to pay in terms of performance for creating so many objects is expensive.

In these tests we decrease the input file to 2GB and used the VisualVM 1.3.7 profiler tool [29] with the Visual Garbage Collection Monitoring Tool plugin [30] to monitor the execution. Figure 4.3 compares the JVM heap space during the execution of a test with Hadoop's `MapOutputBuffer` class and our `RecordHeap` output buffer.

The right side of the figure (4.3b) shows the `RecordHeap` executing over the same input, but having an entirely different memory footprint. The used Java's heap space constantly reaches 600MB, in contrast with left side of the figure (4.3a) where the used heap is stable at 50MB. It is not the high memory consumption which overloads the system and brings performance down, but instead the high volume of memory allocations and deallocations in JVM's heap of short-time objects. One can wonder that each memory peak correspond to one run and, when a spill finishes, a lot of memory is freed at once. However, we recall that the number of records in the buffer stays approximately constant during all the collection phase. For every incoming record only one existing record is selected and replaced in the buffer. What happens in fact is that every time we replace a record (appending it to the current open run) it is polled from the priority queue; since we are constantly inserting new records in the buffer, we are also constantly removing

other records. This constantly removal of objects leaves a thousand of objects allocated in the heap, but without any reference to them. At some point[3] the garbage collector runs and, due to the high amount of orphaned objects in the heap, it has a lot of work.

Despite the overall bad results of this naïve implementation, the adopted strategy seems promising: we have already saved some time in the merging phase. In the next Section, we try to solve the excess of garbage collection by splitting the buffer from the selection tree. The first will be responsible only for storing records, while the former will provide access to the records as well information about the current run, the current partition and other necessary fields to execute the algorithm.

4.3 Pointer Heap

Hadoop reuses the same key-value pair[4] to send the data from the input to the buffer. It does not have to create a new pair for each record it emits because the key and the value are serialized into the buffer. This means the records are read, turned into a byte array, and copied to the buffer. To follow this strategy and avoid creating a massive number of objects, we split the solution from the previous Section: a new byte array will mimic `MapOutputBuffer`'s kvbuffer behavior, being a placeholder for the serialized keys and values, while in a priority queue we insert only metadata about the current record and current spill. This metadata includes the current run number, the record partition, and *pointers* to where the values are located in the buffer — which lead us to name this solution `PointerHeap` buffer.

Replacement selection is based on the premise that when we select the smallest record from the buffer, we can replace it with the incoming record. However, this is only valid if all the records have the same size, i.e., they are fixed-length. This is not true for our experiments and often not true in real world jobs, specially when dealing with text. Manage efficiently the space in the buffer when records are of variable length becomes a necessity. To address this issue, we implemented a memory manager entity based on the design proposed by P. Larson in [9]. This memory management is not an issue in `MapOutputBuffer`'s quicksort strategy because the records are appended in the end of the buffer as they come and, when they are removed, they are all remove at once (when a spill is triggered). Thus, Hadoop is free from this extra overhead we shall detail now.

[3] The Java's garbage collector is multi-strategy, complex entity based on the concept of *generations*. GC runs first in young generations, and objects that survive (i.e., another object still holds a reference to them) are promoted to older generations. A common strategy to trigger garbage collection is to run the collector when memory allocation memory fails. Further references can be found in [31–33].

[4] In a real world job, where the map function is actually executed, there is one key-value pair transmitting data between the `RecordReader` and the map function and another pair between the map function and the in-memory buffer.

4.3.1 Memory Manager

The buffer is divided into *extents*, and the extents are divided in *blocks* of a predetermined size. The block sizes are spaced 32 bytes apart, which result in blocks of 32, 64, 96, 128, and so on. The extent size is the largest possible block size. In our implementation, we used extents of 8KB. For each block size we keep a list containing all the free blocks referent to that size. The number of free lists is given by the extent size divided by the block size, thus $8 \times 1024/32 = 256$. The memory manager provide two main methods: one to *grab* a memory block big enough to hold a record of a given size, and one to *restore* a block which is not in use anymore (the record in that address was spilled).

Grabbing a memory block big enough to hold a given record means to locate the smallest (we want to avoid waste at maximum) free block larger than the record size. The grab method work as follows: round up the record size to next multiple of 32 ($\lceil recordSize/blockSize \rceil * blockSize$). Find the index of the resulting rounded size in the free lists ($roundedSize/blockSize - 1$). Check if the list at the calculated index has a free block: if it does, return it. Otherwise, increment the index and look in the next list (which will be 32 bytes bigger). If we only find a free block bigger than the rounded record size, we take only what is needed and immediately return the excess to its appropriate list. For instance, in the initial case where there is only one free block of 8192 bytes (the extent size), suppose the memory manager must grab a block for a record of size 170. The rounded size of 170 is 192; because all other lists are empty, the manager gets the 8192 block from its list. To avoid a major wasting, the 192 first bytes of the 8192 block are returned, and the other 8000 are placed in its appropriate list. When the record is spilled from the buffer and its memory block becomes free, we return the block to the appropriate list.

We illustrate the concepts of the memory manager in Figure 4.4, which shows a buffer of 8K with some restored blocks. All lists start initially empty except the one referent to the extent size. As the blocks are grabbed and restored, the lists are populated with smaller blocks. In this example we have 10 blocks of variable sizes. The block sizes are written inside the blocks in the memory buffer (lower part of Figure) and over the list they belong in the memory manager (upper part of Figure). As smaller records are restored, the whole manager tends to become faster, having the lists of block size closer to the record average size more populated.

However, this can lead to an exceptional situation. Without lost of generality, imagine that the memory manager is continuously being asked for records of 32 bytes. After 256 block requests, the amount of blocks with 32 bytes in a 8KB extent, say 4 contiguous (i.e., they are physically located one after the other) blocks are restored. At this point,

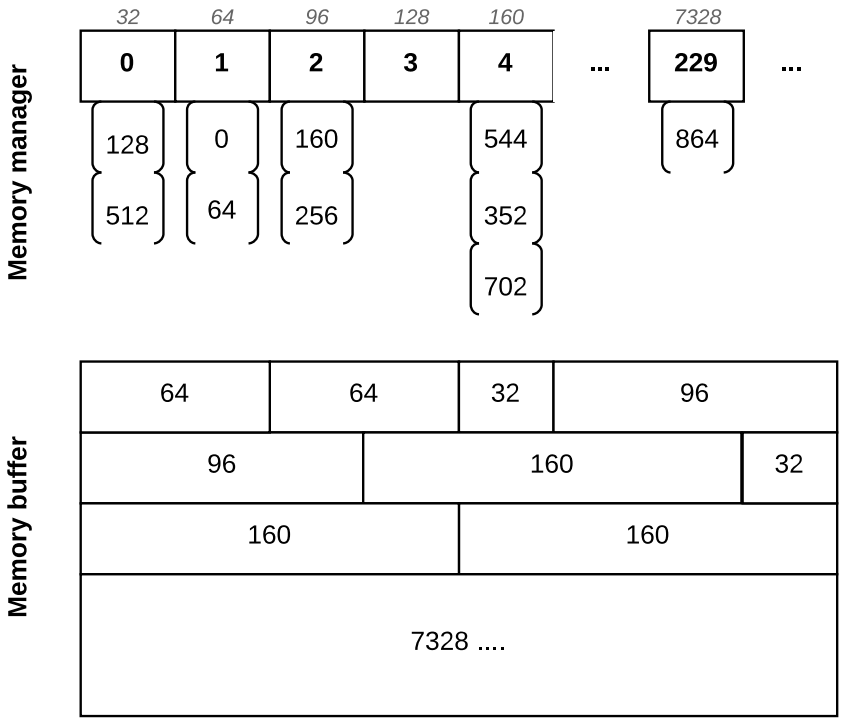


FIGURE 4.4: A possible state of a memory buffer with its corresponded memory manager.

the memory manager has 128 bytes of free memory fragmented in four blocks of 32 bytes. If this stream of small records is interrupted by a bigger record with 96 bytes, the memory manager will not find any block sufficiently large for that record — despite having free memory enough to answer the request.

This situation can be generalized for the whole buffer (across extents), and is provoked because we do not coalesce free neighbor blocks. This is not a trivial task since neighbor blocks can be located in different lists, and we skipped it in the implementation of our buffer for simplicity. In our implementation, we detect this situation when the selection tree has flushed all its records (trying to free enough space in the buffer), but there still no space large engough for the incoming record. When this happens, we restart the memory manager. This forced flush breaks the continuous behavior of the run generation, creating smaller runs. Implementing block coalescing is a feature which should be addressed in a future work.

4.3.2 Selecting and replacing

With a memory manager, we can now serialize the key-value pairs into the buffer and control the selection and the replacement of records with the selection tree. The `HeapEntry` class from the previous implementation evolved into a `LightHeapEntry`, a new class that

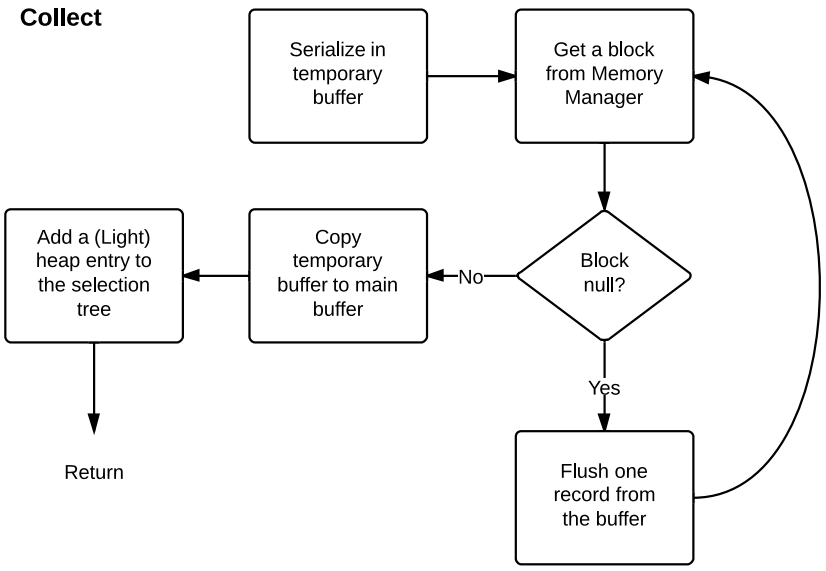


FIGURE 4.5: The loop executes until it has successfully added the current key-value pair into the buffer. It may be necessary to flush a lot of records from the buffer until we can add the current one.

contains four integers (run, partition, key length, and record length) and a (also new) `Block` object, which encapsulates the information from allocated slot in the memory manager in three integers (addres, list index, and rounded record length). This results in only 28 bytes of information, plus the Java object overhead. We also change how the current run is controled. Instead of having a Boolean flag and changing the current active status when creating a new run, we simply control it with natural numbers, starting in 0. If a record should not belong to the current run, its run value is equal to the current run plus one. When the selection tree returns as its smallest record a `LightHeapEntry` with run value bigger than the current run, we *reset* the run, i.e., properly close the current run file and create a new one.

Figure 4.5 introduces the new collect strategy in a flowchart. The key-value pair is serialized first in a temporary buffer because we must know the length in bytes of the record (key length plus value length) to grab a memory block for that size, knowledge which is not available a priori. The process works as follows: Ask the memory manager for a block and check if the returned result is not null. If not, copy the temporary buffer to the main buffer, construct the record's light heap entry and add the entry to the selection tree. Otherwise, the buffer is full and one record must be removed. Flush one record (flush process is detailed in Figure 4.6) and ask the memory manager again. The loop runs until it has created a free slot of sufficient size to add the record to the buffer (or the selection tree become empty and the memory manager is restarted). When flushing a record, select the top record from the selection tree. If the tree is empty, restart the memory manager and return. Otherwise, check if the selected record's run

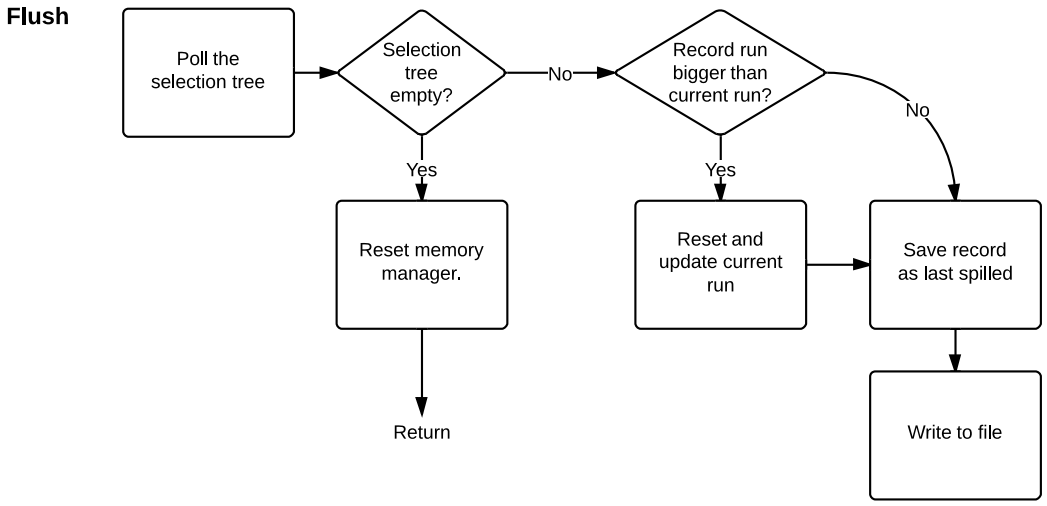


FIGURE 4.6: Whenever the collect method fails to find memory space for an incoming record, it calls this record flush method.

is bigger than the current run: if it is, reset the spill file, update the current run and continue. If the selected record still belongs to the current run, save it as last spilled record and send it to the file writer.

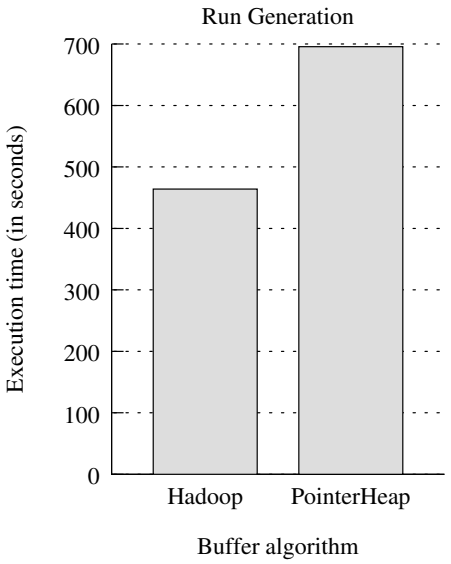


FIGURE 4.7: Comparison between Hadoop's `MapOutputBuffer` and our custom `PointerHeap` buffer.

`PointerHeap` results already show some improvement over `RecordHeap`. Figure 4.7 compares the `PointerHeap` with the original Hadoop. We decreased the run generation execution time to 695 seconds (from the 750 seconds of `RecordHeap`), but we are still too far from `MapOutputBuffer`'s time: 464 seconds. The use of such a data structure to control the record allocation in the buffer such as the memory manager implies in an inherent percentage of wasted space in the buffer. Because there is a *minimum* block size (in our main experiments 32 bytes), any record smaller than that will have at least the size of this minimum block available and, because it is smaller, the difference between block size and record size is wasted — what we refer as *fragmentation* hereafter.

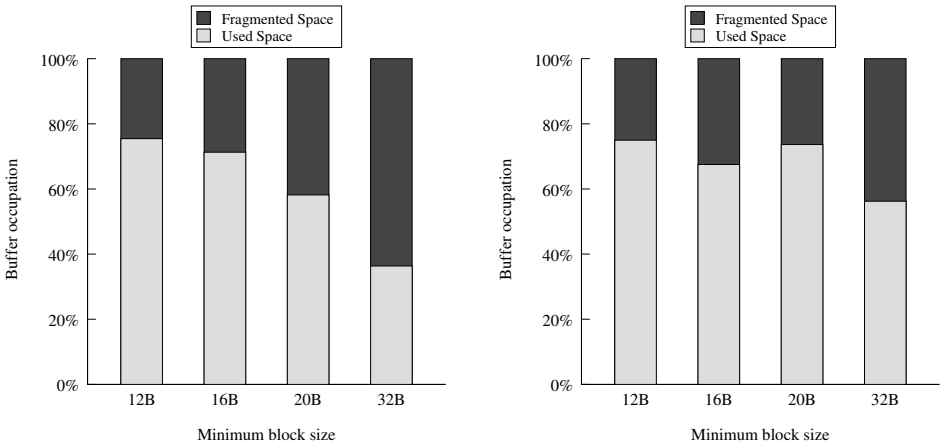
4.3.3 Memory Fragmentation

We now introduce other two inputs used specificaly to test the effects of memory fragmentation. The EnglishWords input has about 3000 different words in English, repeated in order to build an input file with 14 million records (approximately 100MB). The RandomStrings input contains 6.5 million distinct random-generated records varying from 8 to 16 bytes, repeated in order to build an intput file with also 14 million records. The former input has approximately 180MB, and it is bigger than the EnglishWords because the records are in average bigger: English words that span 12 bytes or more are not common. We show a sample of both inputs in Table 4.1. The RandomStrings differ from the main input used on the other tests of this Chapter only in the length of the strings, which here are 10 times smaller.

EnglishWords	**RandomStrings**
affect	0OfSweAIIpaDAieg
anything	1mZFGt7c
beyond	2cJjZKtB7Zi8
classic	bu7wj0Rl3dz1
cover	CkFBuAF5K87EI8gM
final	dUMrhuK5Usi
junior	dWwOsvA9q3
lift	EMrwn4BDtygq
mix	JAAOpqcNj
pick	jjotMUcJE

TABLE 4.1: Sample from EnglishWords and RandomStrings input files.

In Figure 4.8 we present the fragmented memory in two scenarios: in Figure 4.8a using the EnglishWords input and in Figure 4.8b using the RandomStrings input. In the left Figure, we can observe that the bigger the minimum size of the memory blocks, the bigger the fragmentation is. Indeed, as Table 4.1 shows, real words are commonly small. In the right Figure, we see that 32 bytes are also too long: recall the strings' length vary from 8-16 bytes. However, using block sizes of 16 bytes did not bring the best results, as one would expect. The reason behind this is the serialization strategy employed. Hadoop's Text type serializes also the length of the string right before it to be able to deserialize it correctly later on. We also added a zero-byte at the end, to ensure correct lexicographic order in binary string comparison.



(A) Memory fragmentation for common words in English as key.

(B) Memory fragmentation for random strings as key.

FIGURE 4.8: Different choices for the minimum block size used in the memory manager reflect different fragmentation levels depending on the input data.

4.3.4 Comparing Records

Logically speaking, the replacement selection algorithm does not need a priority queue — recall we can select the smallest record in the buffer by linearly comparing a temporary record with the other records, and updating it in the case we found a smaller one. We use a priority queue to support the selection tree strategy, which yields the smallest record at the cost of the logarithm of the number of records. However, once we decided for it, an important characteristic of our priority queue-based solutions is how to compare the entries residing on it. From Section 3.3 we know the key-value pairs must be sorted first by partition and, within each partition by key. In the naïve solution, the key is available in the `HeapEntry` class, but as a generic object. It is the job's programmer responsibility to provide a comparator for the keys when creating the job. This comparator is configurable and available via the MapTask class. Because Java's priority queue accepts a custom comparator object, our only job is to instantiate the configured comparator and pass it as an argument at the moment we create the priority queue.

In the `PointerHeap` class we adopted a different strategy to compare two records. The `LightHeapEntry` implements the `Comparable` interface, which forces the class to implement a *compareTo* method. Here we first compare the runs of the records and, if they are equal, the partition and, if still equal, finally we compare the keys. But because the entries in the selection tree do not contain keys anymore, the compare method must uses the memory block address and key length field to access the real value of key in the main buffer. To allow this access, we moved the `LightHeapEntry` class inside `PointerHeap` as an inner class. Because the keys are already serialized, the comparison between them

is realized at the byte level. However, when dealing with large keys this can be cumbersome. Our next alternative implementation tries to address this issue bringing a small part of the key to the heap entry, aiming to speed up the comparison between records.

4.4 Prefix Pointer Heap

We start this Section quoting Nyberg et al. [13]: "Pointer sort has poor reference-locality because it accesses records to resolve key comparisons". All selection tree should fit in the on-chip data cache (D-cache) in an ideal scenario. But accessing the whole keys from the buffer to resolve record comparisons provoke a lot of cache misses, because the heap entry objects keep being replaced by the keys. Nyberg et al. suggest the use of a prefix of the key rather than the full key. In our implementation, we create an extra 8 bytes field in the `LightHeapEntry` class contained in the `PrefixPointerHeap` class to store the key prefix. We present the results in Figure 4.9. Comparing keys using only the first eight bytes makes the execution time of the run generation phase decrease again; the `PrefixPointerHeap` only needs 614 seconds, which is 81 seconds, or 12%, less than `PointerHeap`.

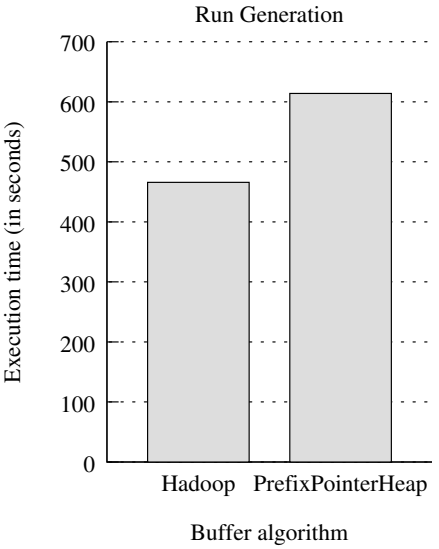


FIGURE 4.9: Comparison between Hadoop's `MapOutputBuffer` and our custom `PointerHeap` buffer.

All the implementation logic described in the previous section is still valid for the `PrefixPointerHeap`, and we just changed the methods which compare the records. The comparators now (after comparing the run and the partition value) check if the prefix of the keys are equal — which means byte-comparing only 8 bytes. If the records can be distinguished at that point, then the compare method will not have to access the keys residing in the buffer. Otherwise, the comparator access them like in the pointer version. One issue associated to the use of key-prefix is that it may not be a good discriminator of the key [13]. This means that the comparator will have to access the keys to decide (Nyberg et al. use the expression "degenerate to pointer sort") but with the overhead of calling lower-level comparison methods for the first eight bytes. In Figure 4.10 we show the percentage of comparisons decided using only the key prefix and the percentage of comparisons which had to read the full key. The results are clear: for

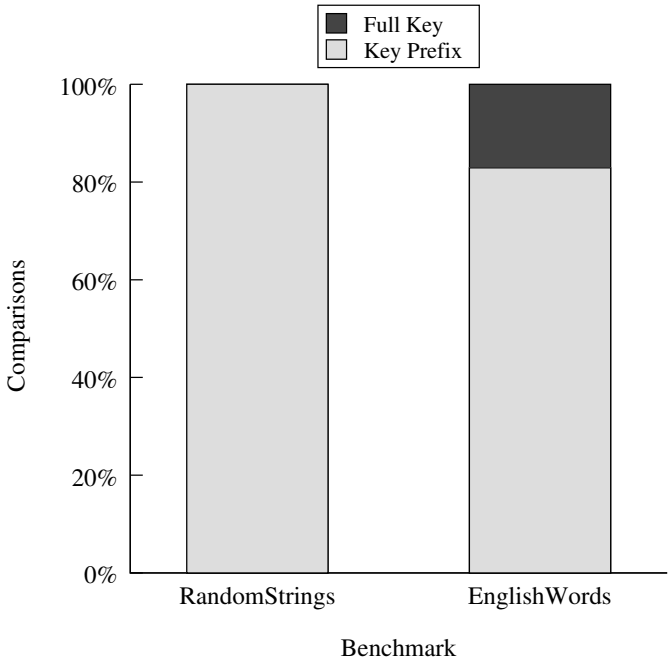


FIGURE 4.10: Percentage of comparisons decided with key prefix and full key for random strings and English words.

artificial keys (possibly not only our random strings but also Table ids, log time stamps, etc.) the key prefix can solve all comparisons without recurring to the full key in the buffer. In smaller and more natural inputs like words (which appear not only in word count jobs but also in inverted index and grep jobs for instance), the key prefix enhances the comparison, but for some keys only eight bytes are not enough to solve ties. For instance, the words *beautiful* and *beautifully* are only distinct at the tenth and eleventh byte.

4.5 Prefix Pointer with Custom Heap

A review of our path up to this point shows that at first we let the JVM manage all our data structures: because there is no distinction between the buffer and the selection tree in the `RecordHeap`, everything was managed by Java's `PriorityQueue`. The excessive time (over 5 times greater than the original output buffer) garbage collecting heap-entry objects lead us to split the buffer and the selection tree in two different data structures. Our second approach — the `PointerHeap` — custom managed the memory buffer where the records are serialized but still left the management of the selection tree for the JVM. The `PrefixPointerHeap` class brought back a small part of the key to the heap entries, but did not advance in any aspect related to memory management. We now present our fourth and last output buffer implementation, which takes control over the heap where

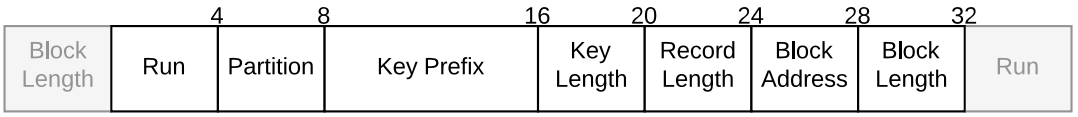


FIGURE 4.11: In the `MetadataHeap`, each record correspond to 32 bytes in the byte array.

the selection tree is based. We shall call it `PrefixPointerWithCustomHeap`, and refer to it by its abbreviation: `PPCustomHeap`.

Our objective is to eliminate at maximum the creation of objects in JVM's heap space during run time, and allocate every needed array or object as soon as possible. One of the main advantages of `PointerHeap` over `RecordHeap` was the serialization of keys and values in a byte buffer of fixed sized. To achieve the same result but for the selection tree, we implemented a custom heap (called `MetadataHeap` hereafter) which employs a byte array as the placeholder for the heap entries. We could stretch the serialization strategy and serialize `LightHeapEntry` objects in the byte array; but due the design decision to avoid allocating objects, we removed the `LightHeapEntry` and the `Block` classes and moved its fields inside the `MetadataHeap` in the form of offsets. We illustrate the idea in Figure 4.11. We use Java's `ByteBuffer` class to wrap the byte array, which provides methods such *putInt* and *getInt*, as well similar methods to set and get byte arrays, longs, and other data types. Now, when we add a key-value pair to the kvbuffer, we add its metadata "heap entry object" by directly writing the run, partition, key prefix, key length, record length, block address, and block length into the `MetadataHeap` byte array. With this design we can directly control how much memory the selection tree will consume, and allocate all of it right in the begin.

Figure 4.12 shows the results from all previous output buffers plus the result for the prefix pointer with custom heap (`PPCustomHeap`). We also abbreviate the `PrefixPointerHeap` to `PPHeap`. Because the difference between these two looks negligible, we decided to run another round of tests only with `PPHeap` and `PPCustomHeap`.

We executed a paired observation with 20 experiments on each of the two algorithms. With a 90% confidence interval the `PPCustomHeap` is faster than `PrefixPointerHeap` for the given input, with a synchronous writer reading and writing to the same HDD. Figure 4.13 show the comparison between run generation (figure 4.13a) and merge (figure 4.13b). However, due to the small difference in these results, we probed again both output buffers. We changed from our artificil random-generated string file to the lineitem table from TPCH benchmark [34], which we explain and use extensively in the next chapter. For now, suffices to say that the lineitem table aims to mimic real world workloads. We

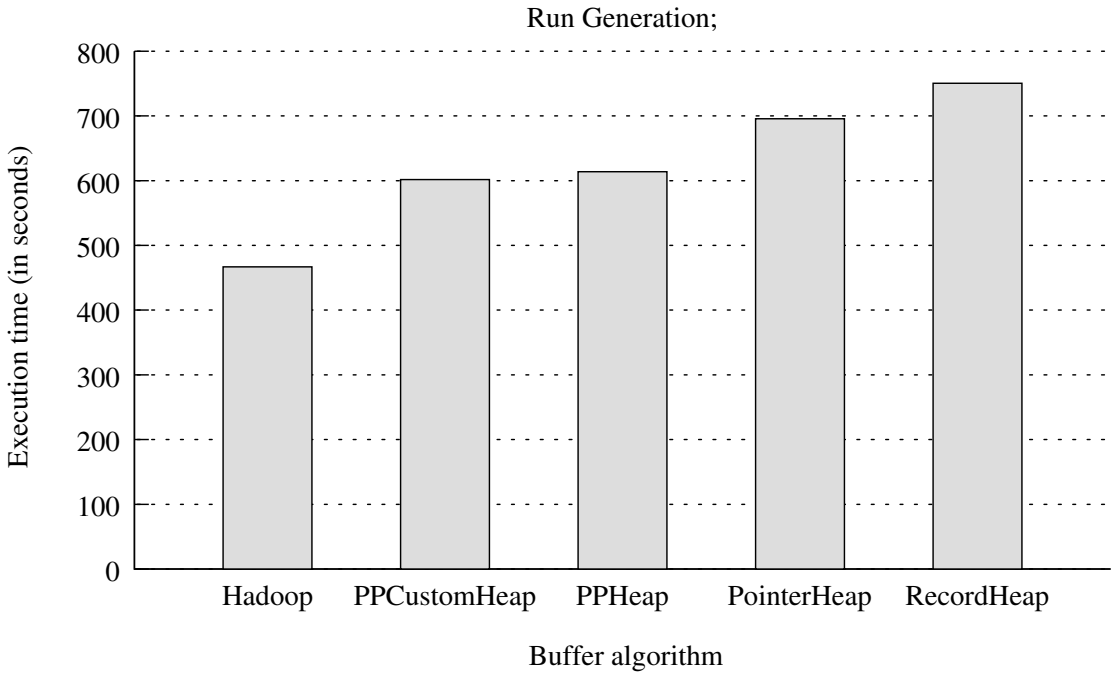
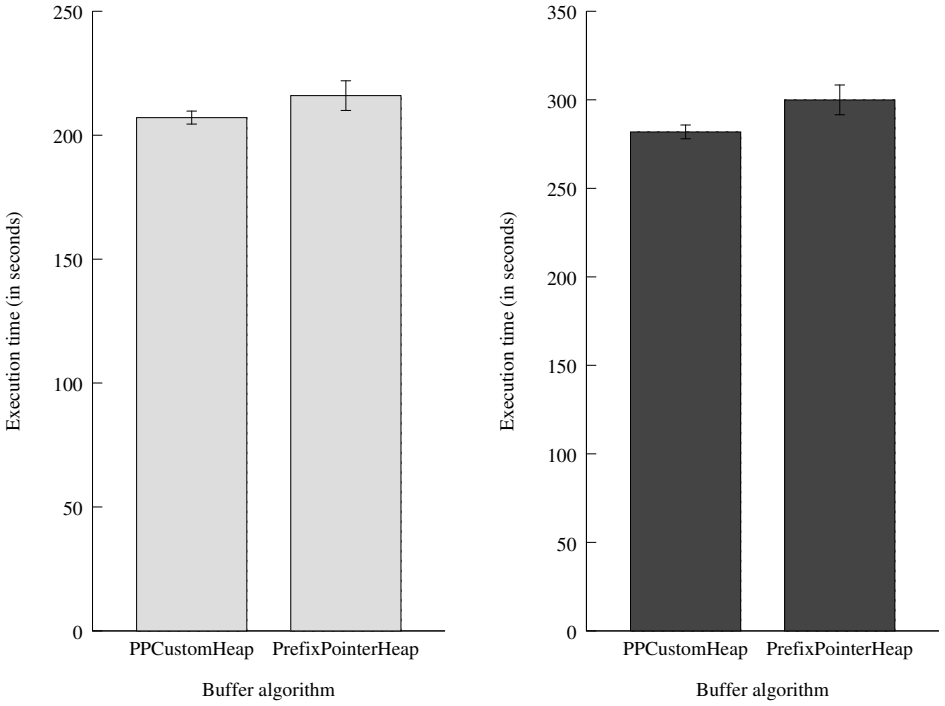


FIGURE 4.12: Comparison between all output buffers.

present the results in Figure 4.14, where we can observe that the `PrefixPointerHeap` (represented as PPHeap in the figure) performed better than the `PPCustomHeap`. The small advantage in artificial input, the worst result in a real input, and the relatively inflexibility of the `MetadataHeap` class may suggest that the `PPCustomHeap` is not worth it. However, under strictly memory constraints having an output buffer class which can be rigid controlled may be a necessity. A deeper study comparing how both classes perform in a myriad of inputs remains as an open question to be addressed in future research.

The evolutionary aspect of this Chapter aimed to show a progression from the most naïve to a fully customized implementation of replacement selection as the main algorithm to handle the in-memory output buffer of the map side task in Hadoop. In the next Chapter, we continue experimenting only with `PPCustomHeap` class and probe it against Hadoop's `MapOutputBuffer`. We address different scenarios, closer to the workloads Hadoop jobs perform on a daily basis.



(A) Run generation. (B) Merge.

FIGURE 4.13: Comparison between `PrefixPointerHeap` and `PPCustomHeap`. Although statistically different, both systems present approximately the same performance.

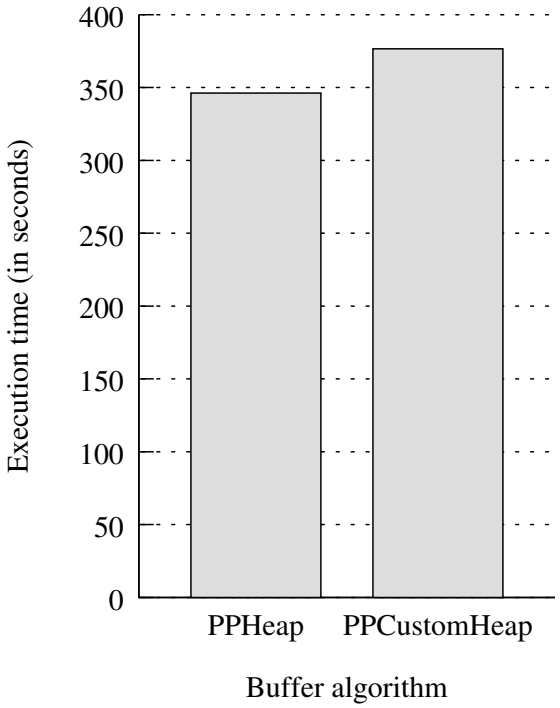


FIGURE 4.14: Comparison between `PrefixPointerHeap` (PPHeap) and `PrefixPointerCustomHeap` (PPCustomHeap) using the lineitem table as input.

Chapter 5

Experiments

This chapter evalautes the performance of replacement selection on the broader context of real-world Hadoop jobs. In the first section, we introduce and discuss an asynchronous writer, which can achieve I/O overlapping when two hard disks are used. In the second section we experiment with inputs ordered, partially ordered, and sorted in reverse order. Finally, in the last section, we execute MapReduce programs representative of problems solved in real-world deployments. We run experiments for Hadoop using its default output buffer (`MapOutputBufer`) and for replacement selection using the `PrefixPointerCustomHeap` class. We shall refer to them by its main algorithm: quicksort in the first and replacement selection in the former.

5.1 Asynchronous Writer

Recall from section 3.3.1 that Hadoop's `MapOutputBufer` triggers its spill thread when its key-value buffer or the accounting buffer reaches a determined threshold (default set at 80% of buffer occupation). When triggered, the spill thread starts consuming records from the buffer. Two situations may occur here: the spill thread writes the records fast enough so that when the collect method reaches the end of the buffer, it can continue writing in the begin of it, because the beginning records were already written out. On the other hand, if the map function emits key-value pairs faster than the spill thread writes the ones in the buffer out, we reach the hard limit of the buffer. If this happens, the collection process is suspended until the spill thread finishes.

In the previous chapter, all output buffers used a synchronous writer. The synchronous writer opens a file when the run starts, and when the output buffer needs to spill one record to create a free slot in the buffer, it calls an append method from the writer.

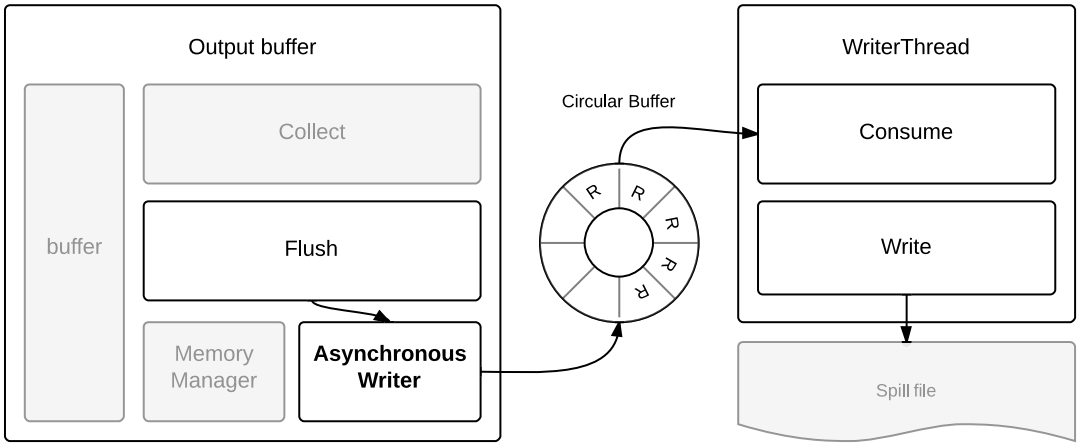


FIGURE 5.1: The circular buffer and the writer thread.

The caller then has to wait until the writer has finished appending the record and have returned. This design, however, does not exploit one of the good characteristics of replacement selection, which is that the run generation process is continuous, alternately consuming input records and producing run files [28]. Thus we abstracted the writer into a single `Writer` interface and implemented two writer classes: `SynchronousWriter` and `AsynchronousWriter`. This allows us to switch between writers with ease, configuring it at job creation time.

Our asynchronous writer follows the producer-consumer pattern, where the output buffer class is the producer and new class called `WriterThread` is the consumer. When the output buffer wants to spill a record, it calls the writer method. In the asynchronous object, however, the writer method does not directly append the key-value pair to an open data stream, but instead it adds this pair to a circular buffer. The circular buffer is a virtually infinite data structure with two pointers: one at the begin and one at the end. The producer (the asynchronous object) adds new records to the circular buffer at the end of it, while the consumer (the writer thread) consumes records from the begin of the buffer. We illustrate this process in Figure 5.1. We create two experiments to test the asynchronous writer: the first uses the same hard disk to read the input and write the output folder, while the second reads from one disk and writes in another one. The setups are illustrated in Figure 5.2, which we adapted from [28].

5.1.1 Single disk

This experiment, which uses one single disk as source and destination, aims to test if the run generation becomes faster simply by using an asynchronous writer. Indeed, we can observe a small improvement in two cases (`PointerHeap` and `PPCustomHeap`), but

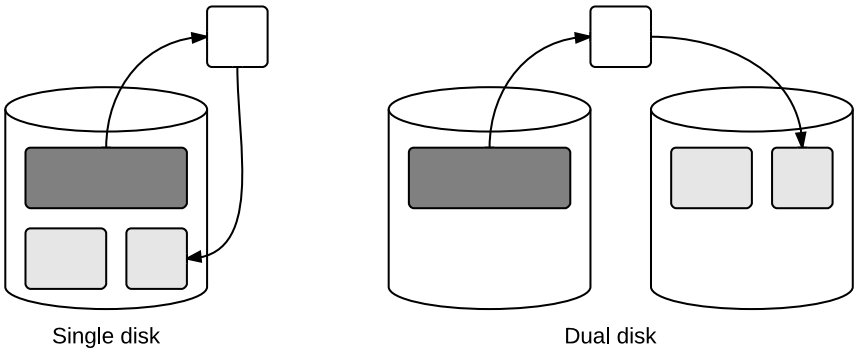


FIGURE 5.2: Run generation with a single disk or dual disks.

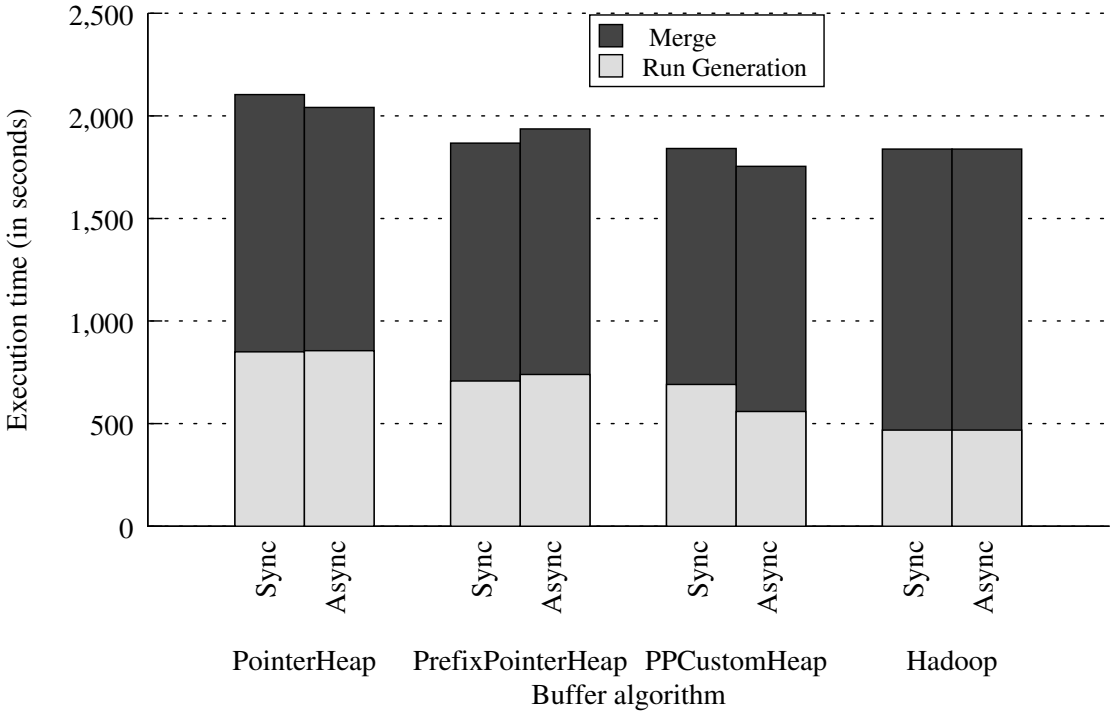


FIGURE 5.3: Comparison between output buffers using synchronous and asynchronous writer with one hard disk.

in `PrefixPointerHeap` the synchronous writer performed better. Due to hard-disk's physical-constraints (as arm movement and seek time), writing at the same time as reading is impossible. At some point, either the read or the write will have to wait fore the other to finish. Thus, we conclude that when there is only one disk available, using an asynchronous writer does not bring any significant benefit.

5.1.2 Dual disks

This experiment, which uses one disk as source and another disk as destination, aims to exploit the continuous run generation characteristic of replacement selection and

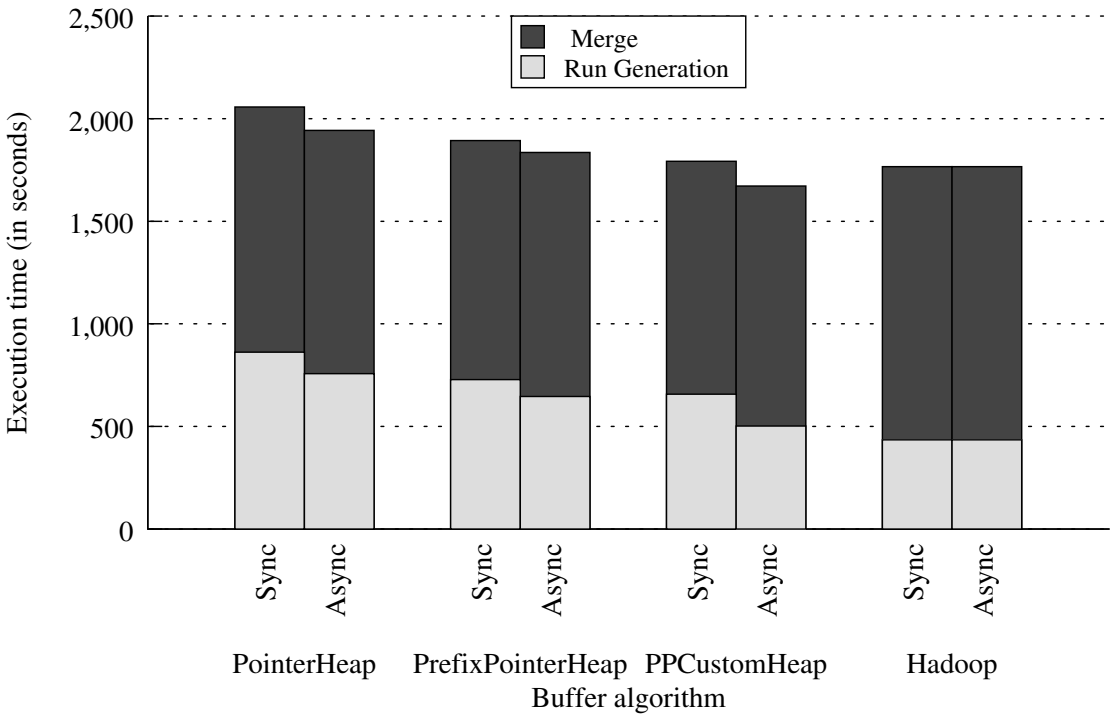
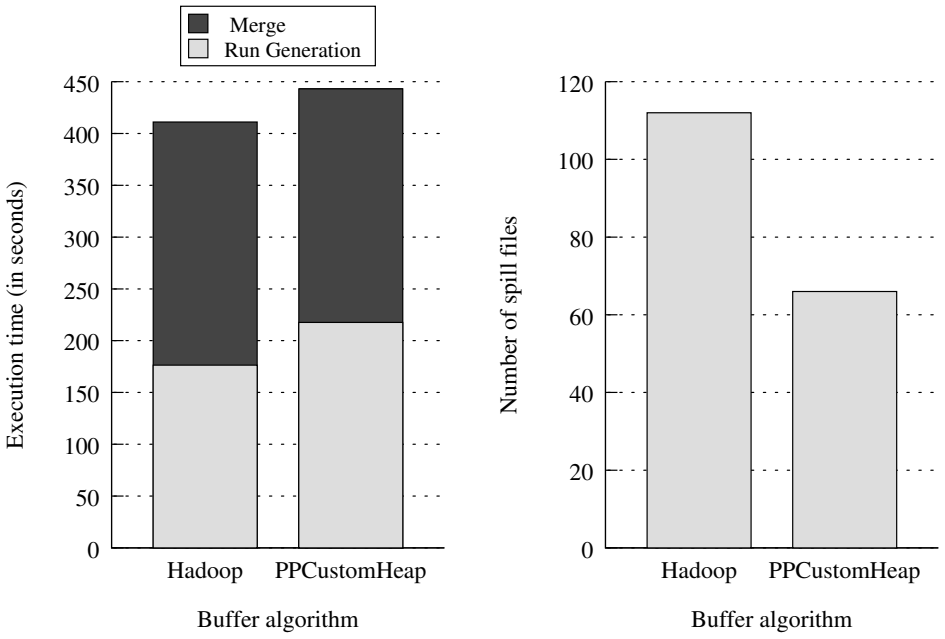


FIGURE 5.4: Comparison between output buffer using synchronous and asynchronous writer with two hard disks.

described in [28], where reads and writes overlap as the input is consumed and the output is produced. Using this configuration, `PPCustomHeap` performs faster than Hadoop because of the savings in the merge phase resulted from the smaller number of spill records, even with the run generation being slightly longer. Because the number of spills are smaller than the merge factor, no further intermediary files were necessary during the merge phase. With two disks, all output buffers performed better using a separate writer thread than using the synchronous writer. Hadoop, however, does not take any advantage of this configuration, because the quicksort algorithm exhibits a fixed read-process-write cycle that does not allow I/O overlapping.

5.2 Ordered, Partially Ordered and Reverse Ordered

In this section, we explore the impact of preexisting sort order in the input file and how replacement selection reacts with respect to that. We used as input the *lineitem* table from the TPC-H benchmark [34]. From the columns of lineitem table, three are of special interest for this work: *orderkey*, *shipdate*, and *receiptdate*. The buffer size used in these experiments was 50MB, and for our custom output buffers we also discounted 10% of that space for auxiliary data structures. First we present a comparison between Hadoop and `PrefixPointerCustomHeap`. This experiment was executed over a random version lineitem table, i.e., we sorted the table by another field and them used as key



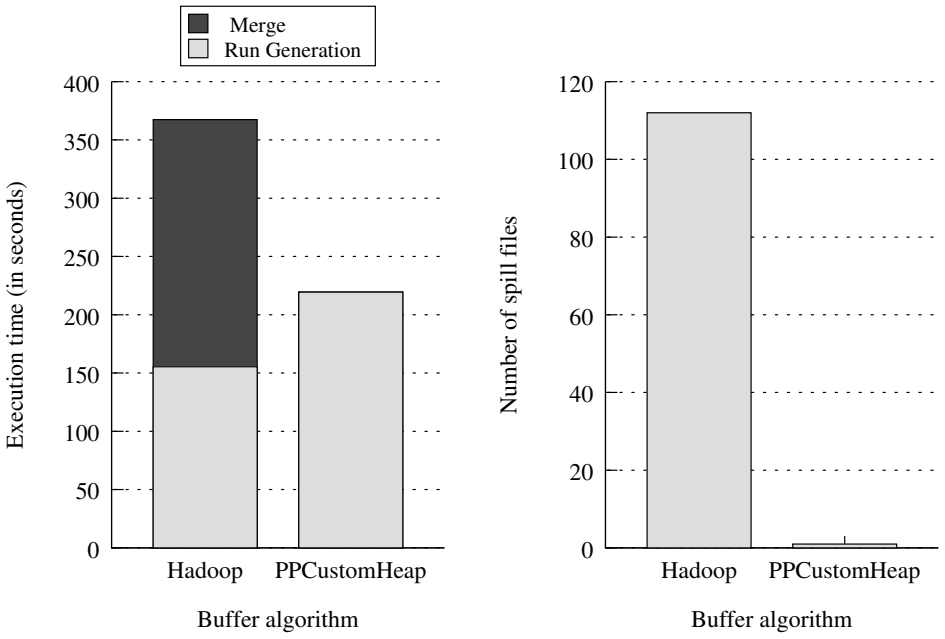
(A) Execution time. (B) Number of spill files.

FIGURE 5.5: Baseline experiment, executed over a random version of lineitem table.

the orderkey column. Results are presented in Figure 5.5. We can see Hadoop performs slightly better because its run generation phase was faster, and its merge phase was only 10 seconds (4%) slower than replacement selection.

V. Estivill-Castro and D. Wood confirmed mathematically in [35] that the length of the runs created by replacement selection increases as the order in the input file increases. In the best case, i.e., where the input file is already sorted, replacement selection is guaranteed to produce only one run. Indeed, Figure 5.6 confirms this expectation, showing the results of processing the lineitem table sorted by the orderkey column. Quicksort typically does not benefit much from incidental ordering in the input [28]. It may seem that such scenario has no practical value, because sorting an already-sorted file is a waste of resources. However, in fact, the input order for large datasets cannot be determined beforehand, especially in big-data scenarios—even if a previous exploratory analysis shows all records in a certain order, it cannot be guaranteed that the ordering indeed holds for the complete (unknown) dataset. Replacement selection offers more robustness in this scenario because it significantly reduces the overhead of sorting.

A scenario which is more common in practice is *partial sortedness*. When an input file has little disorder, the lengths of the generated runs will be longer [35]. In the first experiment, we use as input the lineitem table sorted by *shipdate*, using the column *receiptdate* as new sorting key. The two columns have high correlation, since the dates of receipt and of shipment are likely close to one another. Results are presented in Figure 5.7. If the records are distant by an offset smaller than the total number of records



(A) Execution time. (B) Number of spill files.

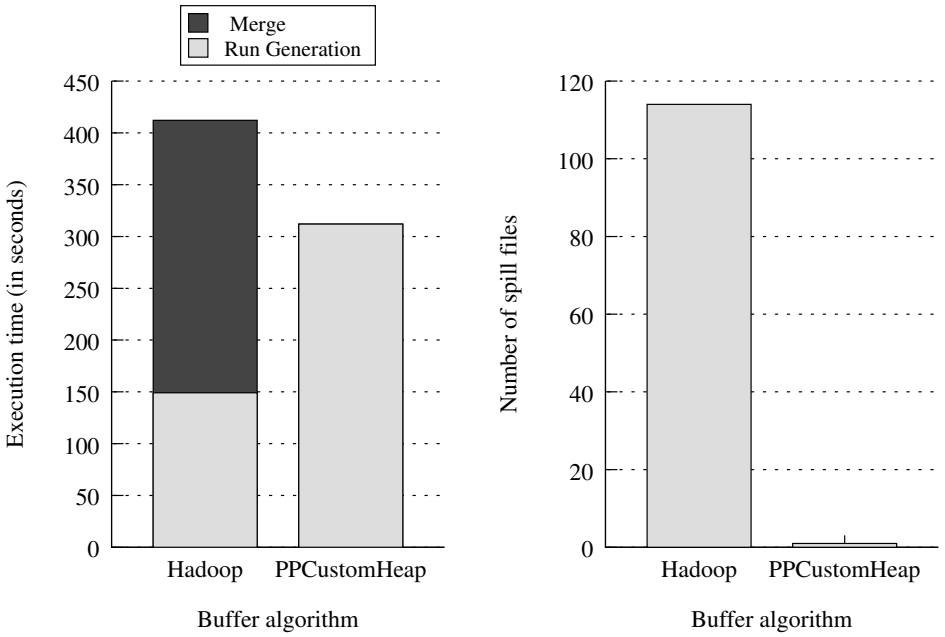
FIGURE 5.6: Replacement selection presents its best case scenario for already sorted inputs, when it produces only one run.

the buffer holds, replacement selection will act like a sorting slide-window. However, defining the sortedness degree of an input on real time and the optimal ratio between this degree and the size of the output buffer remains as an open question.

For completeness, we present the second experiment with partially orderness, where the input is ordered by the *receiptdate* column and *shipdate* is used as key (the opposite of the previous experiment). The results were similar, and we present them in Figure 5.8.

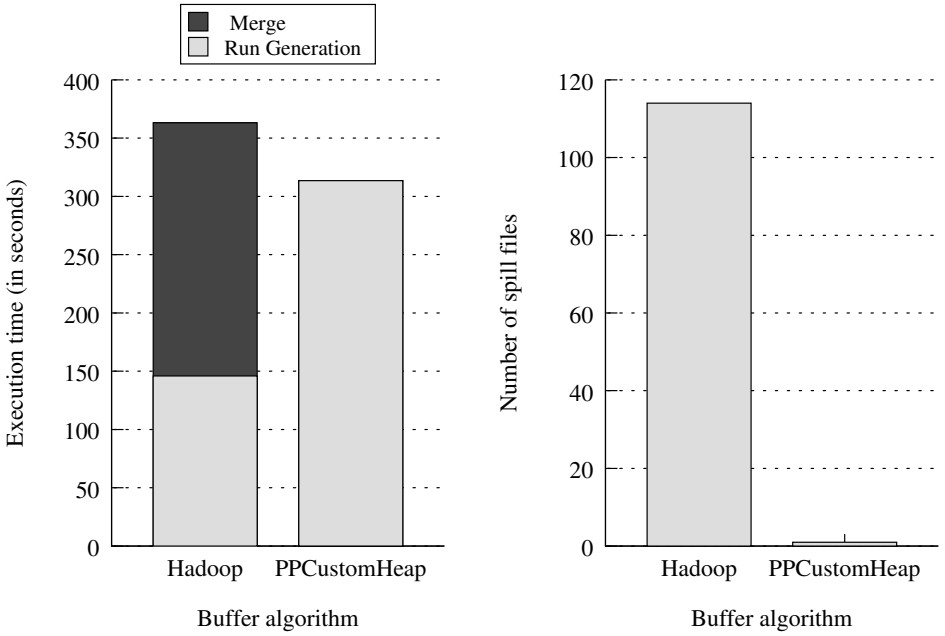
Finally, we present the last and perhaps the most interesting ordering experiment. We sorted in reverse order the lineitem table by the orderkey column. This is the worst case for replacement selection, where it produces as many runs as a quicksort-based run generation.

Suppose we have a buffer which can hold n records and an input file sorted in reverse order where there are no duplicated records. After the first n records are inserted into the buffer, there will be no space for the $n + 1$th record, which triggers the selection process. We selected the smallest record which, in this case, is the record n. Because the file is reversed, we know a priori that the record $n + 1$ is smaller than the record n, which means that the position of the latter is less than the position of the former in the final output. But because record n has just been written out, record $n + 1$ must belong to the next run. When we flush all n records from the buffer, the $n + 1$ record will be the current smallest. At this time, the record $2n + 1$ must be inserted into the buffer



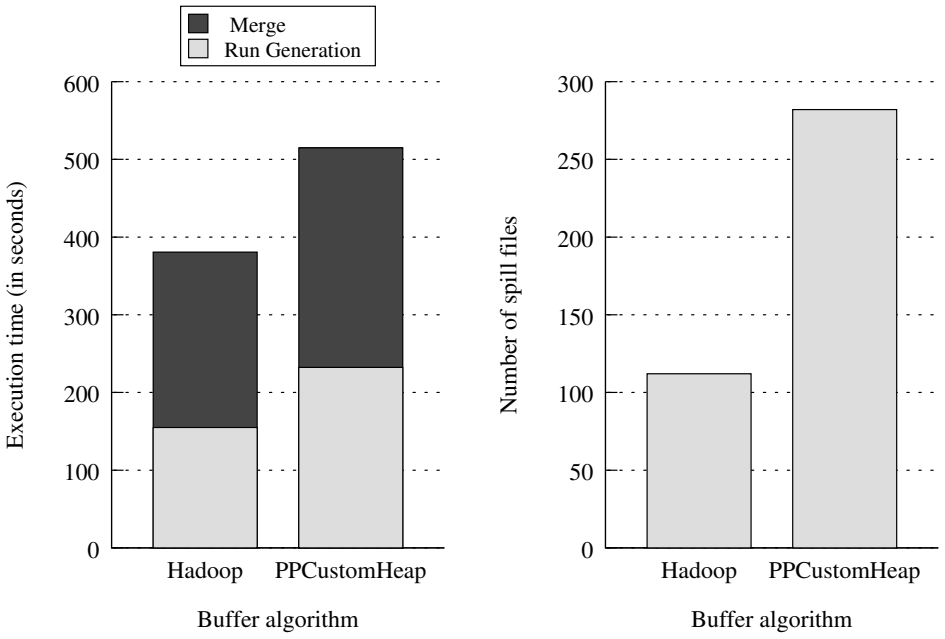
(A) Execution time. (B) Number of spill files.

FIGURE 5.7: Partially ordered file by shipdate results.



(A) Execution time. (B) Number of spill files.

FIGURE 5.8: Partially ordered file by receiptdate results.



(A) Execution time. (B) Number of spill files.

FIGURE 5.9: Partially ordered file by receiptdate results.

and the process repeats. Thus, for each n records (the buffer size) we will have one spill file.

Nevertheless, this is not what we see in Figure 5.9, where in the right (Figure 5.9a) we can see a huge number of spill files created by `PPCustomHeap`. When we analyzed the spills, we noticed that, after an initial normal behavior, their size (i.e., the number of records in each run) decreased rapidly until a certain average value and stagnated there. This leaded us to plot the graph of Figure 5.10, where the number of records in the heap, normally fluctuating during the selection and replacement, suddenly drops. At the same time, the number of free blocks (of 128 and 160 bytes) in the memory manager increases. Figure 5.11 shows another perspective of this situation, namely the amount of occupied blocks during the run generation.

One possible cause for this behavior is the fragmentation of the memory as described in Section 4.3.1. During the run generation phase, after the buffer is full for the first time we need to select one record and replace it for the incoming record. Records residing in the buffer will be spilled as long as a memory block big enough to hold the incoming record cannot be found. In this case, after a flush of more than twenty thousand records, enough memory was freed, and thus we can allocate the incoming record. If the next record is as big as the previously one, all those blocks freed in the last flush are not useful, and we will have to flush records until another block with the same size as the incoming block is selected (which here, in the worst case, is the previously record). This results in a decrease of the usable space of the buffer: despite having multiple free blocks,

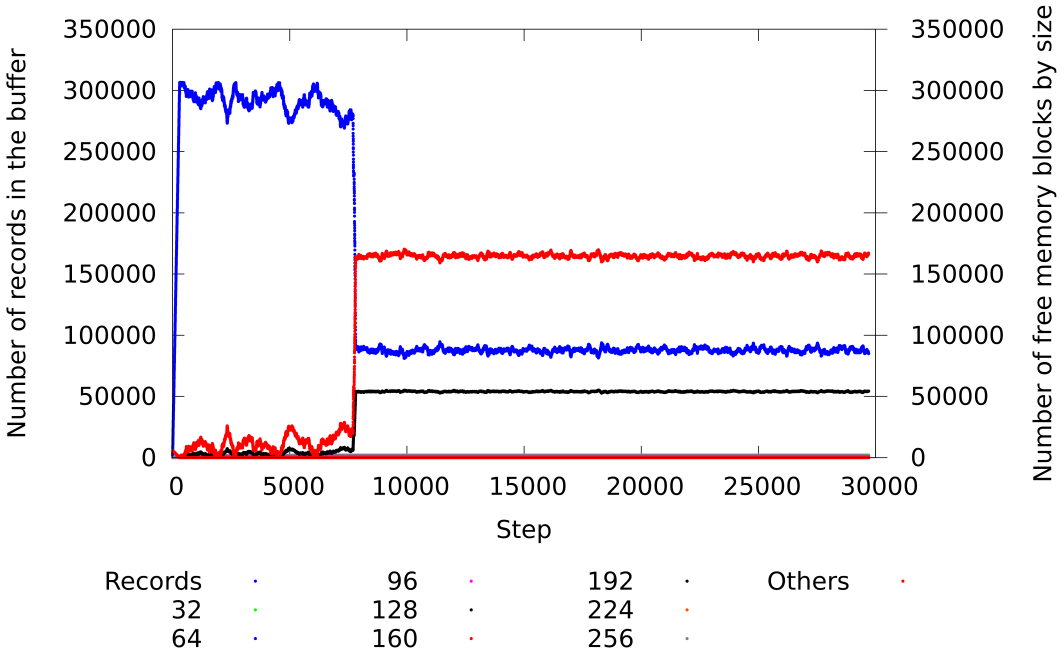


FIGURE 5.10: The total number of records in the buffer reaches 300 thousand when the buffer gets full, but after approximately 8 thousand steps it drops to bellow 10 thousand records.

all of them are too small to contain the new records except a few large ones. If all the next incoming records are large, the buffer is constrained by the number of large blocks.

One possible explanation to the Figure 5.10 and Figure 5.11 is that, when ordered in reverse order, the average size of the lines in the lineitem table tends to grow as the orderkey decreases, i.e., we have bigger records in the end of input if compared to the records from the beginning. But this is not true, as we can see in Figure 5.12, where we plot the raw size (key length plus the value length) of the records with the dots and crossed them with their rounded block size (black horizontal lines). The figure shows a homogeneous distribution of the line sizes in the lineitem table, which disproves the hypothesis.

We could not explain this erroneous behavior, which only occurs when the input is in reverse order. Therefore, we leave it for future work to investigate the problem and find further possible causes. Because the problem also occurs with `PrefixPointerHeap`, we excluded the possibility of a bug in the `PPCustomHeap` implementation. The problem is likely caused by our simplified implementation of the memory manager. The reset mechanism proved to be a reasonable alternative to the coalescence of free blocks in the circumstances described in Section 4.3.1. However, cases like this indicate that the robustness of a proper memory-management implementation are crucial to achieve sorting performance.

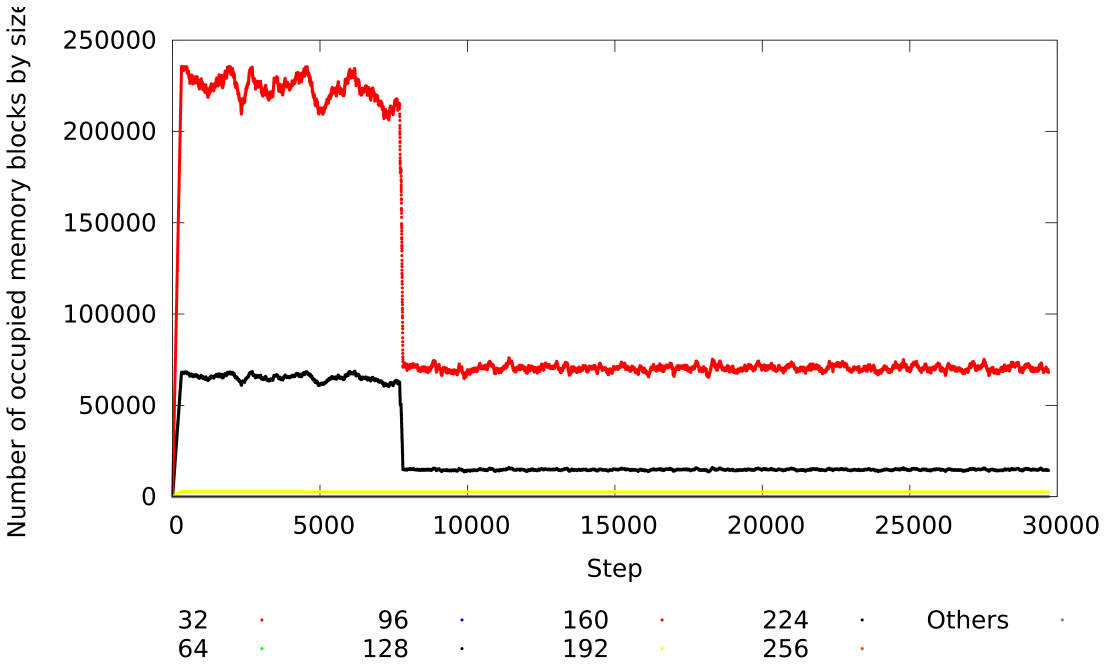


FIGURE 5.11: A split from the records line in Figure 5.10 shows the majority of occupied blocks are 160 bytes long, just followed by 128 bytes blocks.

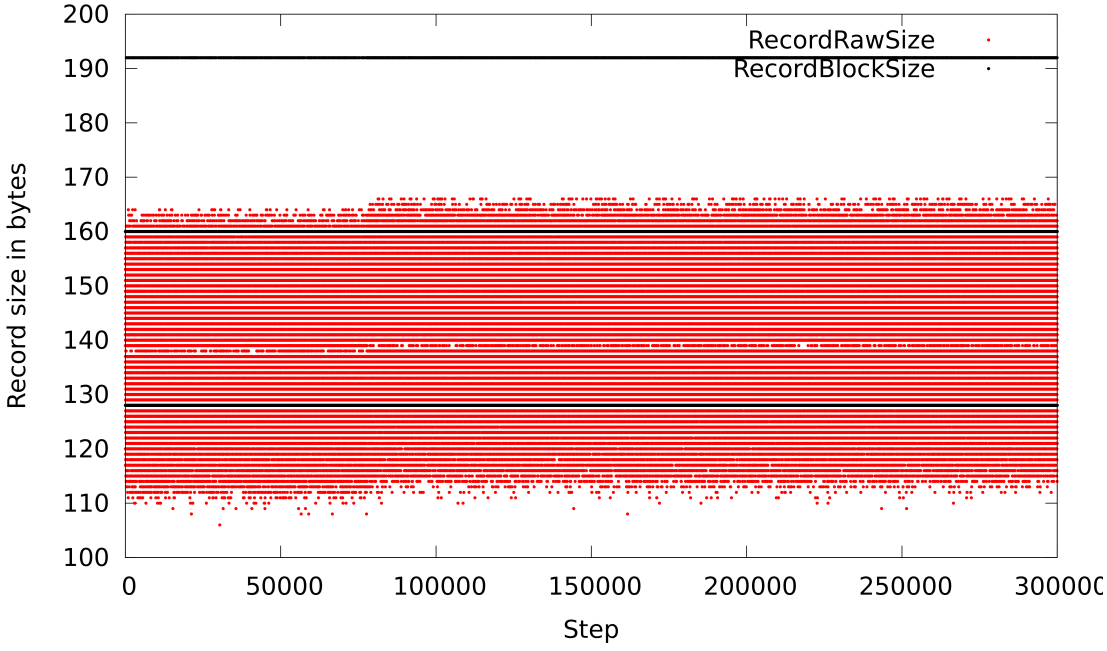


FIGURE 5.12: Length of raw records and rounded records from lineitem table.

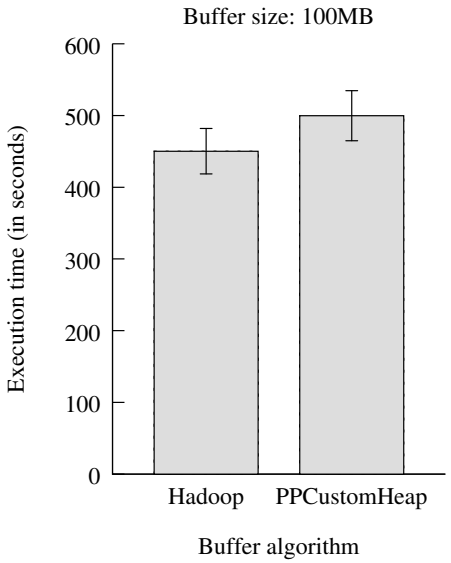


FIGURE 5.13: Join of lineitem and order tables, using a buffer size of 100MB.

5.3 Real World Jobs

To conclude the experiments, we created a test scenario where a join of two tables must be performed. Joins are a common operation in data management systems, but in the MapReduce programming model they are a clumsy task to do. The table which are being joined are the already described lineitem together with a orders table, also from the TPCH benchmark. The lineitem table size is 1GB, and the order table size is 600MB. The tables are being joined by the orderkey field, present in both tables.

This experiments were performed in a small cluster running Hadoop 2.4.0 with six nodes: one master, running the `NameNode` and the `ResourceManager`, and five slaves, where the `DataNode` and `TaskTracker` were being executed. The master has a hardware configuration identical to the machined used in the experiments of Chapter Section 4.1, while the slave machines are composed of an Intel(R) Xeon(R) CPU X3440 @ 2.53GHz with 4GB of main memory and 750GiB hard disk. The operating system and java version are also identical to the ones described in Section 4.1. We measured only the total execution time (end time minus start time) of the jobs, in order to interfere the least as possible with the execution.

First, Figure 5.13 present the execution time of Hadoop and `PPCustomHeap` with a buffer size of 100MB. Hadoop performance was faster in average, because the ratio between the table sizes and the buffer size (10 for lineitem table and 6 for order table) is low. This results in fewer spills than the merge factor for both systems, and the total execution time ends being determined by the run generation time.

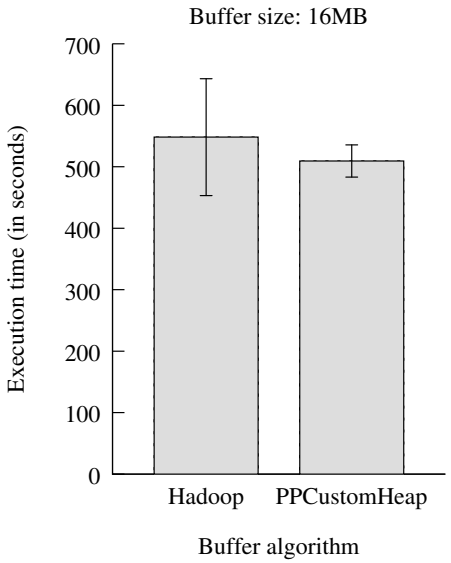


FIGURE 5.14: Join of lineitem and order tables, using a buffer size of 16MB.

For the second test, we decreased the buffer size to 16MB, which yields a 62.5 with the lineitem table and a 37.5 ratio with the order table. The results are presented in Figure 5.14, where the `PPCustomHeap` was faster, because even with the small buffer, the total number of runs produced are not bigger than the merge factor. Despite a small apparently small difference, the replacement selection advantaged of minimizing the number of runs if combined with a proper buffer size configuration, can be significantly for massive data sets.

Chapter 6

Conclusions

This thesis described the implementation and evaluation of an alternative sorting component for Hadoop based on the replacement-selection algorithm. Sorting performance is critical in MapReduce, because it is essentially the only non-parallelizable part of the computation. Thus, every second one could save reflect across the whole cluster.

Our goal with this thesis was to evaluate replacement selection for sorting inside Hadoop jobs, specifically in the map task side. The new versions of Hadoop allow permit an alternative sorting implementation to be plugged to the framework's code, and we took advantage of that during our work.

We first described different in-memory sorting alternatives, as quicksort, mergersort, and heapsort. After that, we introduced external sorting, because Hadoop produces several small sorted files which it has to merge. We explained that the external sorting is divided in two phases: the first phase, called run generation, creates this small sorted files from some input; the second phase, called merge, merges this files into one single ordered file.

The original implementation, based on quicksort, is simple to implement and very efficient in terms of RAM and CPU. During this work, the original implementation was always faster in the run generation phase. However, we demonstrated that under certain conditions, the replacement-selection alternative performs faster in the merge phase, which, depending on the savings' size, can lead to a smaller total execution time compared to the original solution. The main conditions studied during this work were the follow:

- Ratio between input size and buffer size: the major savings in the merge phase occur when replacement selection produces fewer runs than the merge factor. Replacement selection normally produces runs twice as large as the available memory, which results in fewer longer runs.
- Orderness: when an input file has little disorder, replacement selection generates runs with even longer lengths. Quicksort, in turn, does not benefit much from incidental ordering in the input.

Despite the stimulating results, there still a lot to be done. The following tasks are only a small sample of all the possible future work:

- Coalescing of memory blocks: promote coalescing of small adjacent free memory blocks into one larger block. The police of when this should happen also should be studied. An eager coalescing would ignore the fact that the block fragments converge to the average size of the records. A lazy coalescing, triggered only after some iterations of record flushes, perhaps could avoid the fragmentation problem we discussed before while not affecting the performance of the memory manager.
- Memory manager: avoid high fragmentation levels by probing the input in order to decide the best minimum size for the memory blocks;
- Run control: in the `RecordHeap` we use a Boolean flag to keep track of the current active status, which we changed in the following implementations to an integer. Bring back the flag but as a single byte (specially for the `MetadataHeap` case) saves 3 bytes for each entry;
- Partition: the same logic from the run control applies to the partition field in the heap entries. It is uncommon to have a high number of partitions, and one single byte could address up to 128 of them. This yields another 3 bytes saving;
- Sort the heap in the last flush: in the `PointerHeap` and `PrefixPointerHeap` classes, when the flush method is called and we must poll all records from the heap, we get the `LightHeapEntry` array from the priority queue, sort it and send each record to the writer. Because the array behinds the `MetadataHeap` is not an array of objects but simply a byte array, apply this optimization is not a trivial task. One could, however, customize a quick sort which would swap groups of 32 bytes as records.
- Map side merge: to have an honest comparison, we made sure our writer used the same codification as Hadoop uses. This made possible to reuse all the *mergeParts* method from the original buffer. But we believe its possible to write a more efficient

merger which would take advantage of the smaller-in-quantity but longer-in-size spill files, exploiting different merge strategies as show in [7, 21, 28].

Work in low-level Hadoop components is a fruitful and rewarding experience. Nowadays, hundreds of companies base their solutions in Hadoop to address problems in fields like artificial intelligence, machine learning, information retrieval, speech processing, data management, natural language processing, machine translation, and others. Every change made in the core of the framework is reflected across many clients. After some code polishing and further testing, we are confident that the work presented in this thesis may be integrated in the framework, adding a handful tool to be used in terms of sorting.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, page 10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 6:137–149, 2008. URL http://dl.acm.org/citation.cfm?id=1327492.

[3] J Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 2010. URL http://dl.acm.org/citation.cfm?id=1629198.

[4] James Kobielus. Hadoop: Nucleus of the Next-Generation Big Data Warehouse, 2012. URL http://ibmdatamag.com/2012/08/hadoop-and-data-warehousing/.

[5] Caetano Sauer. XQuery Processing in the MapReduce Framework. *Master Thesis*, 2012. URL http://csauer.net/MScThesis_CaetanoSauer.pdf.

[6] Caetano Sauer and Theo Härder. Compilation of Query Languages into MapReduce. *Datenbank-Spektrum*, 13(1):5–15, January 2013. ISSN 1618-2162. URL http://link.springer.com/10.1007/s13222-012-0112-8.

[7] DE Knuth. *The Art of computer programming, Volume 3: Sorting and searching (1973)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:The+Art+of+Computer+Programming,+Volume+3:+Sorting+and+Searching#8http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:The+Art+of+computer+programming,+Volume+3:+Sorting+and+searching+(1973)#1.

[8] Per-Å ke Larson and Goetz Graefe. Memory management during run generation in external sorting. *Proceedings of the 1998 ACM SIGMOD international*

conference on Management of data - SIGMOD '98, pages 472–483, 1998. doi: 10.1145/276304.276346. URL http://portal.acm.org/citation.cfm?doid=276304.276346.

[9] P.-a. Larson. External sorting: Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, July 2003. ISSN 1041-4347. doi: 10.1109/TKDE.2003.1209012. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1209012.

[10] Steven S Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998. ISBN 0-387-94860-0.

[11] Microsoft. Array.Sort Method (Array) in .NET Framework 4.5, 2014. URL http://msdn.microsoft.com/en-us/library/6tf1f0bc.aspx.

[12] Oracle. Java 7 Class Arrays, 2014. URL http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html.

[13] C Nyberg, T Barclay, and Z Cvetanovic. AlphaSort: A RISC machine sort. *ACM SIGMOD ...*, pages 233–242, 1994. URL http://dl.acm.org/citation.cfm?id=191884.

[14] Dalia Motzkin. A stable quicksort. *Software: Practice and Experience*, 1981. URL http://onlinelibrary.wiley.com/doi/10.1002/spe.4380110604/abstract.

[15] Vreda Pieterse and Paul E. Black. Dictionary of Algorithms and Data Structures [online], 2011. URL http://www.nist.gov/dads/HTML/completeBinaryTree.html.

[16] Jyrki Katajainen and TA Pasanen. In-place sorting with fewer moves. *Information Processing Letters*, 70(1):31–37, 1999. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/S0020-0190(99)00038-1. URL http://www.sciencedirect.com/science/article/pii/S0020019099000381.

[17] TH Cormen, CE Leiserson, RL Rivest, and Clifford Stein. *Introduction to algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511. URL http://euler.slu.edu/~goldwasser/courses/loyola/comp363/2003_Spring/handouts/course-info.pdf.

[18] Shen Gao, Jianliang Xu, B He, Byron Choi, and H Hu. PCMLogging: reducing transaction logging overhead with PCM. *Proceedings of the 20th ACM ...*, pages 2401–2404, 2011. URL http://dl.acm.org/citation.cfm?id=2063977.

[19] Edward H Friend. Sorting on Electronic Computer Systems. *J. ACM*, 3(3): 134–168, July 1956. ISSN 0004-5411. doi: 10.1145/320831.320833. URL http://doi.acm.org/10.1145/320831.320833.

[20] PB McCauley. Sorting method and apparatus, 1989. URL https://www.google.com/patents/US2983904http://www.google.com/patents/US4809158.

[21] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2), 1993. URL http://dl.acm.org/citation.cfm?id=152611.

[22] John F Gantz and Stephen Minton. The Diverse and Exploding Digital Universe An Updated Forecast of Worldwide. *IDC White Paper*, 36(10):20–24, 2008. URL http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008. ISSN 00010782. doi: 10.1145/1327452.1327492. URL http://portal.acm.org/citation.cfm?id=1327492.

[24] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 1st edition, 2012. ISBN 0596521979, 9780596521974. URL http://books.google.com/books?hl=en&lr=&id=drbI_aro20oC&oi=fnd&pg=PR5&dq=Hadoop:+The+Definitive+Guide&ots=tZCnxdjVe2&sig=HGg7oRotdi_dBz6zOuaGcgnqZlc.

[25] Tendu Yogurtcu. Hadoop MapReduce: to Sort or Not to Sort, 2014. URL http://blog.syncsort.com/2013/02/hadoop-mapreduce-to-sort-or-not-to-sort/.

[26] Mariappan Asokan. MAPREDUCE-2454 Allow external sorter plugin for MR - ASF JIRA, 2014. URL https://issues.apache.org/jira/browse/MAPREDUCE-2454.

[27] The Apache Software Foundation. Hadoop MapReduce Next Generation - Pluggable Shuffle and Pluggable Sort, 2014. URL https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html.

[28] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys*, 38(3):10–es, September 2006. ISSN 03600300. doi: 10.1145/1132960.1132964. URL http://portal.acm.org/citation.cfm?doid=1132960.1132964.

[29] Oracle. VisualVM, 2014. URL http://visualvm.java.net/.

[30] Inc. Sun Microsystems. visualgc - Visual Garbage Collection Monitoring Tool, 2003. URL http://www.oracle.com/technetwork/java/visualgc-136680.html.

[31] Bill James Gosling, Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® Language Specification, 2013. URL http://docs.oracle.com/javase/specs/jls/se7/html/index.html.

[32] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791.

[33] Brian Goetz. Java theory and practice: Garbage collection and performance, January 2004. URL http://www.ibm.com/developerworks/library/j-jtp01274/index.html.

[34] Transaction Processing Performance Council. TPC BENCHMARK H - Decision Support Benchmark. Technical report, 2013. URL http://www.tpc.org/tpch/spec/tpch2.17.0.pdf.

[35] V Estivill-Castro and D Wood. Faster External sorting and nearly sortedness. 1994. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:External+Sorting+and+Nearly+Sortedness#2http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Faster+External+sorting+and+nearly+sortedness#1.