MONTRES : Merge ON-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems

Arezki Laga, Jalil Boukhobza, Frank Singhoff, and Michel Koskas

Abstract—External sorting algorithms are commonly used by data-centric applications to sort quantities of data that are larger than the main-memory. Many external sorting algorithms were proposed in state-of-the-art studies to take advantage of SSD performance properties to accelerate the sorting process. In this paper, we demonstrate that unfortunately, many of those algorithms fail to scale when it comes to increasing the dataset size under memory pressure. In order to address this issue, we propose a new sorting algorithm named MONTRES. MONTRES relies on SSD performance model while decreasing the overall number of I/O operations. It does this by reducing the amount of temporary data generated during the sorting process by continuously evicting small values in the final sorted file. MONTRES scales well with growing datasets under memory pressure. We tested MONTRES using several data distributions, different amounts of main-memory workspace and three SSD models. Results showed that MONTRES outperforms state-of-the-art algorithms as it reduces the sorting execution time of TPC-H datasets by more than 30 percent when the file size to main-memory size ratio is high.

Index Terms—Sorting, SSD, performance, external mergesort, I/O cost, database, storage, DBMS

1 Introduction

We live in a time when volumes of data to be processed are growing exponentially. Indeed, dealing with terabytes or even petabytes of data becomes a reality [1]. On the other hand, even though DRAM technology has undergone impressive improvements in terms of size, capacity and performance in the past decade, many studies predict that its scaling trends are close to their limit and we will reach a plateau over the next few years [2]. As a consequence, memory capacities can no longer keep up with this data explosion, and designing new data processing algorithms becomes a necessity.

Sorting is one of the most fundamental data processing problems in computer science [3]. While well-studied algorithms like quick-sort and heap-sort are efficient when processing data in main-memory, we need to fall back on external sorting algorithms when the volume of data is too large to fit in the main-memory. Nowadays, as data volumes are several times larger than the main-memory capacity, increasing use is being made of external sorting algorithms relying on secondary storage [4]. External sorting algorithms are fundamental for data processing applications. They are widely used by Database Management Systems (DBMS) for query resolution and index creation [5].

- *A. Laga, J. Boukhobza, and F. Singhoff are with the University Bretagne Occidentale, UMR6285 Lab-STICC, Brest 29238, France. E-mail: {arezki.laga, boukhobza, singhoff}@univ-brest.fr.*
- *M. Koskas is with INRA (Institute of Agronomical Research), Paris 75015, France. E-mail: michel.koskas@agroparistech.fr.*

Manuscript received 30 Sept. 2016; revised 5 May 2017; accepted 9 May 2017. Date of publication 18 May 2017; date of current version 14 Sept. 2017. (Corresponding author: Jalil Boukhobza.) Recommended for acceptance by R.M. Rodriguez-Dagnino. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2706678

External sorting algorithms are composed of two phases: a *run generation phase* and a *run merge phase* [4]. The *run generation* phase splits data into chunks to fit in the main-memory, sorts them into the memory and writes the sorted chunks into intermediate files called *runs*. The *run merge* phase merges the generated runs and writes the sorted data into an output file. External sorting algorithms perform several read/write operations [6], their performance is highly dependent on the way they manage I/Os [4].

Traditional external sorting algorithms were designed based on hard disk drive (HDD) performance model. The performance of read and write operations are symmetric, while sequential I/Os are more efficient than random ones.

In the last decade, flash-based solid-state drives (SSDs) have emerged as the next-generation storage devices and are now an alternative to HDDs. SSDs have been adopted in a wide range of areas thanks to their high I/O bandwidth and short access latency (at the expense of a higher cost)[7]. In fact, SSDs offer a new performance model, with nearly symmetric random/sequential read performance and generally asymmetric read/write performance [8]. This performance model requires external sorting algorithms to be adapted.

State-of-the-art external sorting optimizations for SSDs commonly follow one of two optimization paths. The first one consists in modifying the run generation phase in order to reduce the number of write operations. To achieve this, the authors of [9], [10], [11] try to directly generate some sorted data in the first phase either by relying on efficient random reads [10], [11] to search for minimal values or by scanning the overall file many times [9] in the aim of avoiding costly write operations. The second path consists in optimizing the run merge phase by relying on the data distribution and using more random reads to reduce the sorting effort (number of comparisons) [12]. All these

TABLE 1
Notations

Notation	Definition
N	Input file size in blocks
M	Main memory size in blocks
B	Number of values in one block
D	Data distribution in one block
P	Amount of saved write operations
R	Unitary read operation cost
W	Unitary write operation cost

optimizations benefit from SSD capabilities, mainly by efficient random read operations. Unfortunately, this is achieved at the expense of excessive memory utilization or too many additional read operations.

The main-memory working space allocated for the sorting algorithm is restricted by the DBMS. This restriction depends on the size of the main memory allocated to the DBMS and the requirements of concurrent database operators. Data volume to sort can potentially be several times larger than the main memory working space allocated for the sorting process. Then, the performance of external sorting algorithms needs to scale under memory pressure. Unfortunately, state-of-the-art algorithms [9], [10], [11], [12] do not scale well in this respect (see Table 2).

In this paper we present a new external sorting algorithm named Merge ON-The-Run External Sorting (MONTRES), which uses both optimization paths. It first upgrades the run generation phase based on three techniques: (1) an ascending block selection that prepares for the run generation while flushing the maximum values to the output file directly, thus reducing the dataset on which the run merge is performed; (2) a continuous run expansion policy that generates the largest runs possible to reduce the number of passes of the run merge phase; and finally, (3) a merge on-the-fly mechanism that permanently checks for values to evict during the run generation phase in order to reduce the dataset size for the run merge phase. In the run merge phase, MONTRES tries to minimize the I/O cost by performing the merge phase in one pass at the expense of a small additional number of read operations.

We tested MONTRES with data from the TPC-H [13] benchmark with three different SSDs. We varied the file size to memory size ratio to test MONTRES under different configurations of memory pressure. We compared MONTRES with state-of-the-art sorting algorithms for SSD and the results show that our approach enhances overall performance by more than 30 percent under high memory pressure.

We present in this paper the following contributions:

- An I/O cost analysis of state-of-the-art external sorting algorithms. This part of our contribution shows why these algorithms do not scale with very large datasets under high memory pressure.
- The new external sorting algorithm, called MONTRES, that benefits from the SSD performance model and is scalable even in case of high dataset size to memory size ratio.

This paper is organized as follows. Section 2 provides some background about state-of-the-art research on external sorting and analyses in terms of I/O operations. In Section 3, we introduce MONTRES. In Section 4, we discuss the I/O cost of MONTRES. Section 5 presents the experimental methodology and results. Finally, Section 6 concludes and outlines some perspectives for future work.

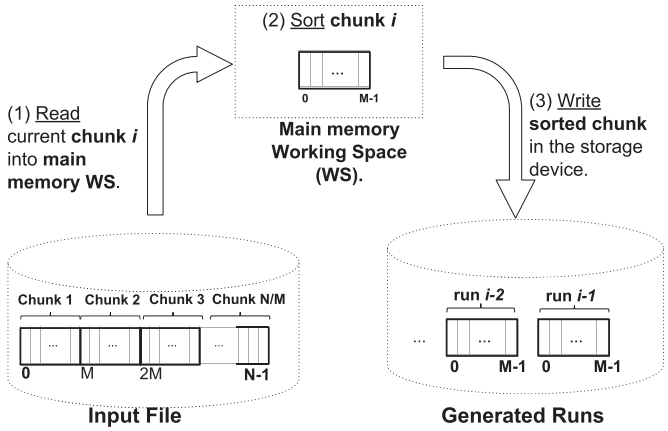


Fig. 1. Illustration of the run generation phase.

2 Background and I/O Analysis of Related Work on External Sorting

This section describes the basic external sorting algorithm named *mergesort* and previous work to improve its performance on flash memory-based storage devices such as SSD. We also present the first contribution, which is the I/O cost analysis of state-of-the-art algorithms. We show that state-of-the-art improvements of the mergesort algorithm do not scale well with large files under memory pressure. This is what motivated us to design MONTRES.

2.1 Mergesort External Sorting

In order to describe and evaluate external sorting algorithms, we use the notation summarized in Table 1. We consider an input file containing n values. These values are grouped in blocks of B elements. The number of blocks in the file is denoted by N. We consider the main-memory working space as the amount of memory allocated by the DBMS to the sorting operator. The main-memory working space can contain M blocks (giving m values). D denotes the distribution of data in the blocks of the input file. As in [11], the distribution of data within a block is given by the number of different values in this block. Finally, R and W denote the unitary read and write costs, respectively.

The mergesort algorithm [3] is one of the most commonly used and studied external sorting algorithms. Chunks of data from the input file handled during the *run generation phase* have to fit in main-memory working space. Thus, each chunk contains M contiguous blocks. Chunks are processed successively (see Fig. 1). They are first loaded into main-memory (Fig. 1 (1)), then sorted using an in-memory sorting algorithm (Fig. 1 (2)), and finally written into the run, an intermediate file in the external memory (see Fig. 1 (3)). The total number of generated runs is equal to $\frac{N}{M}$. The run generation phase results in N read operations on the input file in order to load data and N write operations for writing the generated runs (cache effects are ignored).

The *run merge phase* consists in merging the $\frac{N}{M}$ runs created during the run generation phase (see Fig. 2).

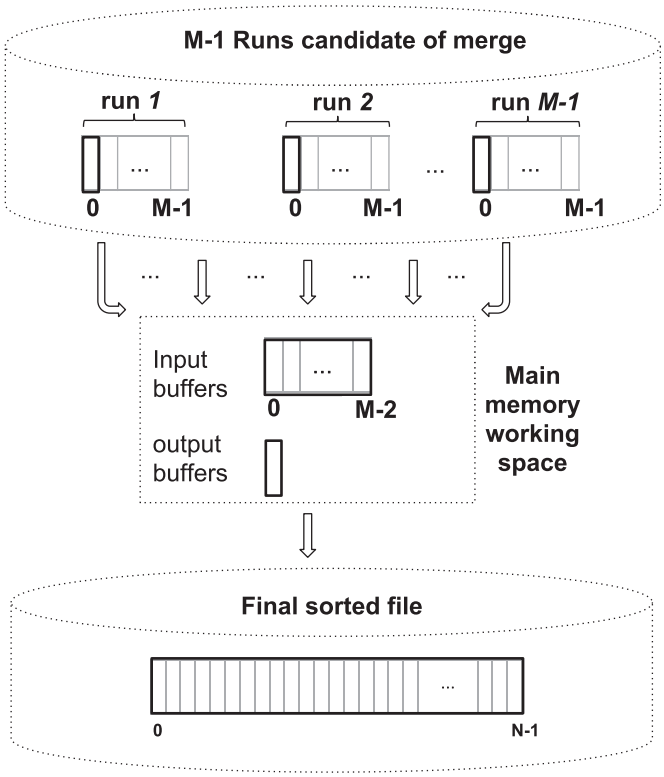


Fig. 2. Illustration of the run merge phase.

The number of runs to be merged at once is limited by the amount M of the main-memory working space. The mergesort algorithm needs $M-1$ buffers as input buffers and uses the remaining one as an output buffer. Each input buffer stores a block from a different run and the output buffer stores merged data to be evicted to the output file.

When the number of generated runs $\frac{N}{M}$ exceeds $M-1$, the merge process needs $\lceil \log_{M-1} \frac{N}{M} \rceil$ passes to generate the final sorted file from all the generated runs. Thus, the run merge phase will result in $N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil$ read and write operations, and the total I/O cost of the traditional mergesort algorithm is given by

$$IO\_cost = N \cdot (R+W) + N \cdot \left\lceil \log_{M-1} \frac{N}{M} \right\rceil \cdot (R+W). \quad (1)$$

The first expression is related to the *run generation phase* and the second is due to the *run merge phase*.

2.2 I/O Analysis of State-of-the-Art Optimizations

External sorting optimizations based on SSD performance models have seen a surge of interest in the past few years. In this section, we investigate upgrades of the external mergesort algorithm for SSD. For each study, we attempt to give the main idea of the algorithm and the theoretical I/O cost (the overall evaluation result is summarized in Table 2).

2.2.1 Natural Page Run

Basic Principle. The optimization introduced in [10] relies on finding data that are already partially sorted to speed-up the run generation phase. To achieve this, the concept of naturally occurring page runs is used. A *natural page run* is a sequence of blocks whose values do not overlap but are not necessarily sorted.

During the run generation phase, this algorithm detects natural page runs in the input file. Values within blocks of the natural page run are not sorted during the run generation phase, instead, an index is created to hold the sequence of these blocks to be sorted during the merge run phase. Indexing natural page runs instead of creating normal sorted runs results in a reduction of the total number of write operations during the run generation phase.

I/O Analysis. Finding natural page runs requires an additional scan of the whole input file. Then, the cost for read operations for the run generation phase is $2 \cdot N \cdot R$ and the cost for write operations is $(N-P) \cdot W$, with P being the number of blocks of natural page runs. The I/O cost of the run merge phase remains unchanged by this optimization. The total I/O cost is given by

$$IO\_cost = 2 \cdot N \cdot R + (N-P) \cdot W + N \cdot \left\lceil \log_{M-1} \frac{N}{M} \right\rceil \cdot (R+W).$$

2.2.2 FSort

Basic Principle. FSort [14] is an external sorting algorithm for flash-based sensor devices with a small memory footprint. FSort reduces the complexity of the merge phase by expanding the size of runs to $2 \cdot M$ (rather than M). FSort uses a replacement selection algorithm to enlarge the size of runs during the run generation phase. This makes it possible to reduce the number of I/O operations of the run merge phase as it reduces the number of passes performed during the merge.

I/O Analysis. By decreasing the number of runs created to $\frac{N}{2 \cdot M}$, the number of merge passes is reduced to $\lceil \log_{M-1} \frac{N}{2 \cdot M} \rceil$. Thus, the overall I/O cost is given by

TABLE 2
External Sorting Algorithms I/O Cost

Ref	RGRC	RGWC	RMRC	RMWC	R/W on data
Mergesort [3]	$N \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil \cdot R$	$N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil \cdot W$	Logarithmic Read Write
Natural page run [10]	$2 \cdot N \cdot R$	$(N-P) \cdot W$	$N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil \cdot R$	$N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil \cdot W$	Logarithmic Read Write
FSort [14]	$N \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-1} \frac{N}{2M} \rceil \cdot R$	$N \lceil \log_{M-1} \frac{N}{2M} \rceil \cdot W$	Logarithmic Read Write
MinSort [11]	$N \cdot (1+D) \cdot R$	$N \cdot W$	0	0	$1+D$ Read operations
FAST [9]	$N \cdot \frac{N}{M} \cdot R$	$N \cdot W$	0	0	Quadratic Read
FAST(N) [9]	$N \cdot \frac{Q}{M} \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-1} \frac{N}{Q} \rceil \cdot R$	$N \cdot \lceil \log_{M-1} \frac{N}{Q} \rceil \cdot W$	Quadratic Read
MONTRES	$(2 \cdot N + P_R) \cdot R$	$(N+P_W) \cdot W$	$(N - P_W - G + \alpha) \cdot R$	$(N - P_W - G) \cdot W$	Linear Read Write

$$IO\_cost = N \cdot (R + W) + N \cdot \left\lceil \log_{M-1} \frac{N}{2M} \right\rceil \cdot (R + W).$$

2.2.3 MinSort

2.2.3.1 Basic Principle. Flash MinSort [11] was also designed for embedded devices with a very limited amount of memory. The MinSort algorithm performs in a single phase and avoids the complexity and cost of the run merge phase. MinSort divides the input data into a number of regions (the default region size is one page) and maintains an index on the smallest value for each region. MinSort performs in many passes. On each pass, a number of tuples with the smallest value from the index are sent to the final sorted file and the minimum index is updated.

MinSort uses additional read operations to retrieve the next minimum value from all the regions. In this way, it reduces the amount of write operations by performing the sorting process in a single phase.

I/O Analysis. MinSort performs a first scan of the input file to create the index of minimum values for each region. Then, each region is read d times, where d is the number of different values in the region. Let D be the average of d (different) values for a given input file (averaged across all regions). The read cost for MinSort is $N \cdot (1 + D) \cdot R$ and the write cost is $N \cdot W$. The resulting I/O cost is given by

$$IO\_cost = N \cdot (1 + D) \cdot R + N \cdot W.$$

2.2.4 FAST

2.2.4.1 Basic Principle. FAST [9] is designed for mobile database systems with a limited amount of main-memory. It also performs in a single phase. FAST scans the input file several times. In each pass, it retrieves the m smallest values and outputs them to the final sorted file. Because read operations are generally more efficient than writes in flash memory, FAST performs up to $(\frac{N}{M})$ scans on the input file to save N write operations. The basic FAST algorithm processes files up to $(M \cdot \frac{W}{R})$ in size.

I/O Analysis. The read cost for the basic FAST algorithm is $N \cdot \frac{N}{M} \cdot R$ with $\frac{N}{M} < \frac{W}{R}$ and the cost of write operations is $N \cdot W$, leading to an overall cost of

$$IO\_cost = N \cdot \frac{N}{M} \cdot R + N \cdot W.$$

2.2.5 FAST(N)

Basic Principle. FAST(N) [9] is a generalization of the FAST algorithm that processes files larger than $(M \cdot \frac{W}{R})$ in size. FAST(N) employs the FAST sorting algorithm repeatedly for each chunk of Q blocks (with $M \leqslant Q \leqslant N$), instead of N pages, to generate the initial sorted runs. Then, during the run merge phase, Q sorted runs are merged instead of $M - 1$.

I/O Analysis. The total read cost for the run generation phase is $N \cdot \frac{Q}{M} \cdot R$ and the total write cost is $N \cdot W$. Once all the chunks are sorted and written to the run, the algorithm starts the external run merge phase. FAST(N) creates runs of Q blocks that can be greater than the external mergesort run size. Then, the number of runs is reduced and the merge passes minimized. The number of merge passes is given by: $\lceil \log_{M-1}(\frac{N}{Q}) \rceil$. Thus, the I/O cost for the run merge phase is $N \cdot \lceil \log_{M-1}(\frac{N}{Q}) \rceil \cdot (R + W)$, leading to

$$IO\_cost = N \cdot \frac{Q}{M} \cdot R + N \cdot W + N \cdot \left\lceil \log_{M-1} \frac{N}{Q} \right\rceil \cdot (R + W).$$

Other state-of-the-art work has relied on designing storage system architecture for performance optimization, such as in [15], or better using SSD parallelism through some implementation related optimizations such as Fmergesort [12]. As these studies did not deal with algorithmic optimization of the external mergesort, we consider them as orthogonal to our work but outside the scope of this paper.

2.3 Discussion on I/O Cost of State-of-the-Art Work

In this section, we discuss the I/O cost of state-of-the-art algorithms and analyze their scalability with regard to increasing dataset size and memory pressure. Table 2 summarizes the I/O costs for external sorting algorithms. We use the following abbreviations in the table: *RGRC* for Run Generation Read Cost, *RGWC* for Run Generation Write Cost, *RMRC* for Run Merge Read Cost, and *RMWC* for Run Merge Write Cost. From Table 2, one can see that I/O cost increases according to: (1) the input file size N and (2) the main-memory working space M.

Natural page run performs two read passes, during the run generation phase, on the whole input file $2 \cdot N \cdot R$ and tries to save P write operations at the end of this phase. The run merge phase I/O cost (RMRC and RMWC) of this algorithm remains the same as that of mergesort. The natural page run algorithm becomes efficient only when the saved write operations exceed the additional read operations cost, when

$$P \cdot W > N \cdot R. \tag{2}$$

Although some previous studies showed that write operations are at least 5 times slower than reads [16], current SSDs try to address this issue by providing large buffers. In the SSDs tested here, this ratio is between 1 and 2. By using $W = 2 \cdot R$ in the last equation above, it becomes apparent that the inequality $P > \frac{N}{2}$ needs to be satisfied in order for this algorithm to be more efficient than mergesort. This means that half of the data should be natural page runs, which is very uncommon. In addition, the algorithm detects these runs into the main memory. Thus, under memory pressure, only a low proportion of such runs can be found.

Even though it is interesting that sorting algorithms perform well with partially sorted data, many state-of-the-art benchmarks (TPC-H) and real applications have near random data distributions (very low values of P). In these cases, mergesort performs better than natural page run.

FSort I/O cost also depends on the input file size N and the main-memory working space M. This algorithm attempts to reduce the number of merge passes during the run merge phase at the expense of additional CPU operations during the run generation. The FSort run generation phase I/O cost equation is the same as that of the mergesort and performs the run merge with $\lceil \log_{M-1} \frac{N}{2M} \rceil$ passes, thus the I/O cost will be lower or equal to the one of mergesort.

To conclude, from the I/O cost point of view, FSort seems to behave better than mergesort. In the evaluation we conducted, it was selected for comparison with MONTRES.

MinSort I/O cost only depends on the input file size N and the distribution of values over blocks (no M variable in the cost). Despite performing the sorting process in a

single phase, it achieves a high amount of read operations $N \cdot (1 + D)$ but only a single write pass $N \cdot W$.

As MinSort tries to use more read operations to reduce the number of writes, we consider the higher ratio from the tested disks, which is $W = 2 \cdot R$. Thus we obtain the following equation for the I/O cost

$$IO\_cost = N \cdot (3 + D) \cdot R. \quad (3)$$

In order for MinSort to be more efficient than mergesort, the following inequality should be satisfied

$$N \cdot (3 + D) \cdot R < N \cdot (3 \cdot R) + N \cdot \left\lceil \log_{M-1} \frac{N}{M} \right\rceil \cdot (3 \cdot R). \quad (4)$$

Thus, $D < 3 \cdot \lceil \log_{M-1} \frac{N}{M} \rceil$. For the very high ratios of $\frac{N}{M}$ tested in our study (8,192), it turns out that D should be lower than 9 for MinSort to perform better than mergesort. This means that a page of 1,024 values should contain only 9 different values. This type of value redundancy may happen in some cases, but it cannot be generalized.

FAST(N) I/O cost largely depends on the input file size N. It aims to reduce the write cost at the expense of additional reads. During the run generation phase, it generates larger runs than mergesort by performing $\frac{N \cdot Q}{M}$ read operations on the input file with $M \leqslant Q \leqslant N$. Then, RGRC is found in the following interval:

$$N \cdot R \leqslant RGRC \leqslant \frac{N^2}{M} \cdot R. \quad (5)$$

The inequality (5) shows that the RGRC is the same as that of the mergesort $N \cdot R$ in the best case (when $Q = M$). In addition, the inequality shows that the FAST RGRC is quadratic when the chunk size Q is close to N, which is higher than the external mergesort RGRC.

The run merge I/O cost (RMRC and RMWC) depends on the value of the chunk Q. When $Q = M$, the number of runs equals $\frac{N}{M}$ and the run merge phase I/O cost is equivalent to that of mergesort. When $Q = N$, only one run is created and the run merge phase is avoided by performing $\frac{N^2}{M}$ read operations during the run generation phase. This also exhibits a quadratic cost with regards to the file size. To conclude, when the file size is large and memory is low, FAST(N) I/O cost can be the same or higher than that of the mergesort depending on the size of chunks Q. In addition, it is not scalable because its cost evolves quadratically in relation to the file size N.

To conclude, mergesort outperforms MinSort in general cases and has a better scalability for large files in cases with medium to high numbers of different values.

2.4 Summary

Most state-of-the-art external sorting algorithms for SSD were not designed to be scalable with regards to file size and memory pressure. They generally do not perform better than traditional external mergesort. Single phase algorithms like MinSort and FAST(1) were designed to be efficient for mobile devices and sensors and focus more on the flash memory lifetime than on I/O sorting efficiency. As shown above, these algorithms can result in higher I/O cost than mergesort. Two phase algorithms like natural page run and FSort result in I/O costs that are close to those of mergesort when the file size N is several times larger than main-memory working space size M. However, natural page run works mainly in the case of partially sorted data.

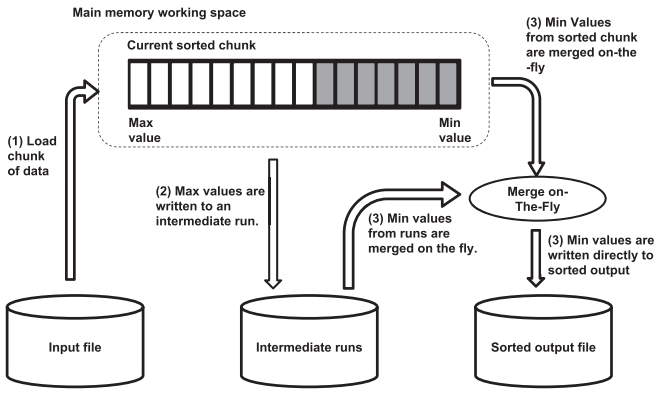


Fig. 3. Run generation phase of MONTRES.

In this paper, we describe a scalable sorting algorithm, named MONTRES, tailored for SSD performance model.

3 MONTRES: Merge On-the-Run External Sorting

Similarly to the traditional mergesort algorithm, MONTRES consists of a run generation phase and a run merge phase. The *run generation phase* includes three main mechanisms (see Fig. 3): (1) an ascending block selection algorithm for optimized run creation, (2) a continuous run expansion policy, and (3) a merge on-the-fly mechanism:

1) *Ascending block selection*. The objective of this technique is to use random reads on SSDs to select blocks in ascending order according to their minimal value for the run generation step. Doing this makes it possible to evict some minimal values directly to the sorted file, and prepares for the two following optimizations.
2) *Continuous run expansion policy*. The objective of this optimization is to create runs that are as large as possible. Doing this allows the run merge phase to be reduced as seen earlier. So, rather than separately creating each run from its respective chunk in the run generation phase, we continuously expand pre-existing runs, when possible, with large values from ongoing chunks. This optimization is efficient when a part of the data is partially sorted.
3) *Merge on-the-fly mechanism*. The objective of this optimization is to reduce the run merge phase effort by reducing the number of values to merge. This is done by continuously detecting and then evicting small values to the sorted file throughout the run creation phase at the expense of some additional read operations.

Thanks to the above mentioned techniques, MONTRES both reduces the amount of intermediate sorted data (runs) and expands the size of runs, thus decreasing the run merge phase I/O cost.

During *the run merge phase*, MONTRES is designed to proceed in a single pass rather than $\lceil \log_{M-1} \frac{N}{M} \rceil$ passes. This is achieved by continuously retrieving minimum values from the generated runs and outputting them to the final

Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8
45	36	36	02	12	08	45	71
17	89	58	54	25	09	39	69
18	77	17	63	06	10	41	77
25	44	12	11	00	11	35	85

Fig. 4. File content example.

sorted file at the expanse of an additional, yet reasonable, number of read operations.

3.1 Run Generation Phase

The run generation algorithm of MONTRES aims to generate intermediate sorted files called runs. MONTRES, tries to decrease the volume of the intermediate data by outputting part of the values to the final sorted file, and also by attempting to increase the size of the runs.

3.1.1 Ascending Block Selection

The ascending block selection mechanism starts by building an index (called *min-index*) of the minimum values contained in each block of the input file. To achieve this, MONTRES performs a first scan of the whole input file in order to detect the smallest values of each block, giving N values, (see lines 4, 5 and 6 in Algorithm 1). The created index is then sorted in ascending order of minimum values of the blocks it contains (see line 7 in Algorithm 1).

Algorithm 1. Run Generation Algorithm

```
1  size_t : N, M;
2  data_type : Block[B];
   /* Build min-index on input file                    */
3  for I ← 0 to N do
4     Block = ReadNextBlockFromInputFile();
5     min = minimum(Block);
6     InsertIndexEntryToMinIndex(Block.id, min);
7  sortMinIndex()
8  while NOT EMPTY(min-index) do
9     LoadMFirstBlockToMainMemoryWS();
10    SortChunkInMemory();
11    next_min_value ← NextMinFromMinIndex();
12    WriteSortedDataGreaterThanNextMin();
13    RunExpansionPolicy();
14    MergeOnFly(next_min_value);
```

Once the index is created, chunks are successively loaded and sorted into main-memory according to min-index, beginning with blocks containing minimum values (see line 9 in Algorithm 1). In doing this, we seek two optimizations: (1) as minimum values are already detected, they can be directly evicted to the final sorted file, (2) then, we rely on data values locality and assume that if a block contains a small value, then there is a high probability that it contains neighboring small values, which can consequently be also directly evicted into the final sorted file.

The loaded data are sorted using an in-memory sorting algorithm creating a sorted chunk (see line 10 in Algorithm 1). Then, sorted values in main-memory that are smaller than the next minimal value from the next block of the min-index (called $next\_min\_value$) can be directly evicted into the sorted file, while higher values are written to the current run (see lines 11, 12 in Algorithm 1). Note that, to optimize I/Os, the eviction is done at the end of each run generation together with the on-the-fly merging, which will be discussed in Section 3.1.3.

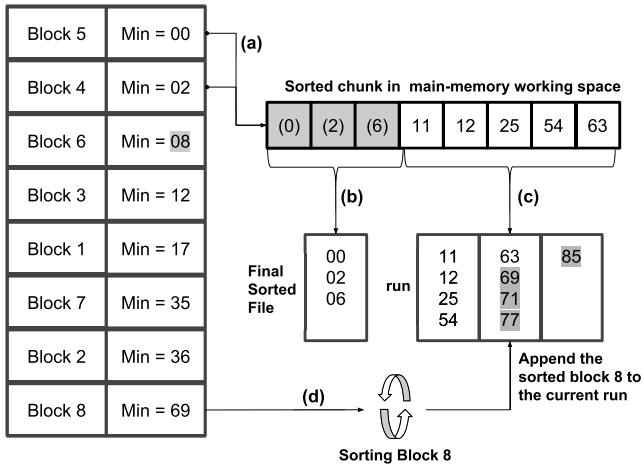


Fig. 5. Building min-index, starting the run generation.

Example 1. Fig. 4 shows an example of a file to sort. The input file contains $N = 8$ blocks, where each block stores $B = 4$ values. The main-memory working space is limited to $M = 2$ blocks. First, file blocks are scanned sequentially in order to retrieve the smallest value for each block. These minimum values associated with the corresponding block IDs are used to create the *min-index* presented in the left side of Fig. 5. $Block_5$ and $Block_4$ containing the lowest values in the min-index are loaded into the main-memory working space (operation (**a**)) to form a chunk of data in main-memory. This chunk is sorted using an in-memory algorithm. Values in the sorted chunk lower than or equal to the next lowest value in the min-index (here $next\_min\_value = 08$) will be later evicted to the final sorted file (operation (**b**)). The remaining values are written to the storage device to form a sorted run (operation (**c**)).

3.1.2 Continuous Run Expansion Policy

As previously discussed, the merge performance complexity largely depends on the number of generated runs. The proposed run creation policy aims to reduce the number of generated runs by expanding their size.

Once the current run has been generated, MONTRES tries to retrieve, from the *min-index*, blocks having all values higher than the maximum value in the current run. Retrieved blocks are loaded into main-memory, sorted, and appended to the generated run.

Example 2. In the previous example, all values of $Block_8$ are higher than the maximum value of the generated run (which is 63, see Fig. 5). So, $Block_8$ is loaded into the main-memory, sorted, and the resulting block is appended to the run (operation (**d**) in Fig. 5).

3.1.3 Merge On-the-Fly Mechanism

As previously described, for each iteration of the run generation algorithm, thanks to the ascending block selection mechanism, values in the main-memory that are lower than or equal to $next\_min\_value$ are evicted to the sorted file.

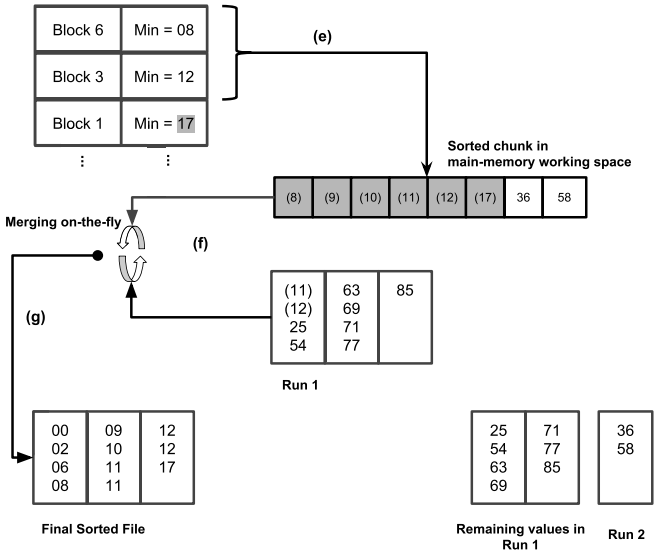


Fig. 6. Example of merge on-the-fly.

For this eviction to take place, we use a merge algorithm on all previous runs containing values lower than or equal to $next\_min\_value$. To do this, MONTRES loads one block from each of these runs, performs the merge operations, evicts the data and so on (on all blocks containing smaller values). In the worst case scenario, the memory occupation of the merge on-the-fly mechanism happens when we generate the last run (in case merging on-the-fly requires to read one block of each previous run) and this is equal to $\frac{N}{M} - 1$.

Values to merge from the ascending block selection and merge on-the-fly mechanism are grouped and evicted together to optimize I/O cost. By doing this, MONTRES reduces the amount of candidate data for the merge phase.

Example 3. Fig. 6 shows the processing of the second chunk from the *min-index* and the input file. The next $M = 2$ blocks from the *min-index*, $Block_6$ and $Block_3$, are loaded into the main-memory working space, then sorted. Values in the sorted chunk that are lower than or equal to $next\_min\_value = 17$ are merged on-the-fly (operation (**f**)) with values 11 and 12 from the previous run, and directly evicted to the final sorted file (Operation (**g**)). Note that if there were more runs already sorted, MONTRES would have looked for values smaller than $next\_min\_value = 17$ in all of them.

3.1.4 Preparing the Run Merge Phase

In order to prepare for the run merge phase, blocks of the intermediate runs are indexed according to the minimum value they contain (in ascending order) and their position in the run (a run may have several blocks). This index is created when outputting runs into the storage device. We name this data structure the *run-index*.

To conclude this first phase, the run generation of the MONTRES algorithm results in the following three outputs:

- *Sorted data*: Part of the data that will be stored into the final sorted file.
- *Sorted runs*: Another part that will be stored into runs still to be merged.
- *run-index*: An index of blocks sorted in ascending order according to minimal values.

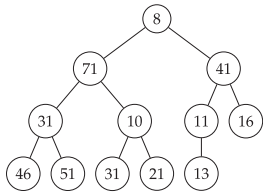


Fig. 7. Min-Max heap data structure for the run merge phase.

3.2 Run Merge Phase

The run merge phase aims to produce the final sorted file based on intermediate runs.

With the traditional run merge algorithm, several merge passes ($\lceil \log_{M-1} \frac{N}{M} \rceil$) are required to generate the final sorted file when the number of runs generated ($\frac{N}{M}$) exceeds the number of input buffers in the main-memory ($M - 1$).

Instead of performing several merge passes, MONTRES relies on the created *run-index* to load minimal values into the main-memory working space by performing random read operations on the run files. Then, those values are evicted to the sorted file and this process carries on until there are no more values to merge. The run merge is realized in one pass at the expense of some additional read operations.

3.2.1 Min-Max Heap

The merge process sorts the runs data using a tree-based data structure called a heap [17]. Nodes of the data structure obey the min-max heap property: each node at an even level in the tree has a value smaller than all of its descendants, while each node at an odd level in the tree has a value larger than all of its descendants (see Fig. 7). We chose this data structure because it makes it possible to retrieve minimum and maximum values with a constant time. Our implementation of the min-max heap used of traditional insertion and deletion functions for this data structure.

3.2.2 One-Pass Data Merge Processing

The merge process starts by selecting $M - 1$ blocks (1 block is used for the output) containing the smallest values from run-index (see Algorithm 2). Selected block minimum values are inserted into the min-max heap with a pointer to the corresponding blocks.

Algorithm 2. Run Merge Algorithm

```
1 min-max-heap : heap;
2 index : run-index; int : nbBlockInMemory = 0;
   heap ← getBlocksFromRunIndex(M − 1);
   nbBlockInMemory ← M − 1;
   nextMin ← getNextMinFromRunIndex();
3 while NOT EMPTY(heap) do
4    nbFreedBlock ← OutputValuesFromHeapLowerThanNextMin();
       nbBlockInMemory− = nbFreedBlock;
5    if nbBlockInMemory < M-1 then
6       heap ← getBlocksFromRunIndex( M - 1 - nbBlockInMemory);
        nbBlockInMemory ← M − 1;
7    else
8       nbFreedBlock ← MergeInputBlockInMainMemory();
        nbBlockInMemory− = nbFreedBlock;
        heap ← getBlocksFromRunIndex(M − 1 − nbBlockIn
        Memory);
9    nextMin ← getNextMinFromRunIndex();
```

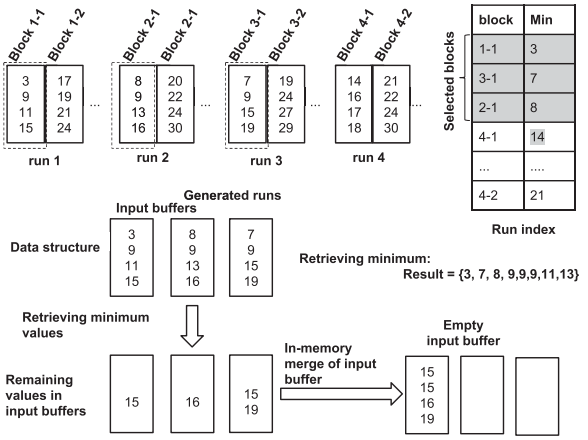


Fig. 8. Example of run merge phase with four runs and three buffers.

Once the data are inserted in the tree, MONTRES retrieves all values smaller than the next minimum value in the run-index from the tree, and appends them to the final sorted file. Once this has been done, MONTRES needs to load the next block(s) (according to run-index) to carry on the merge process.

If one or several buffers are fully emptied by the merge process, next blocks in the run-index are loaded into the free buffers and inserted into the tree. If there is no buffer available (line 7, Algorithm 2), two options are possible:

- When two input buffers contain less than $\frac{B}{2}$ values, these values are merged in main-memory using the mergesort algorithm to free one input buffer. This makes it possible to compact the content of main-memory working space and frees some buffers.
- When the first option is not possible, the buffer containing the maximum value in the tree is dropped from main-memory.

A block containing the next minimum value is then loaded from the storage device into the main-memory.

Example 4. Fig. 8 illustrates the run merge phase of MONTRES with four runs generated using three input buffers. Each run is composed of many blocks (2 are shown in the figure), and each block contains four values. The run index describes the distribution of values on the runs, it is sorted in ascending order according to minimum values. The run merge process starts by reading from the runs the three blocks with the lowest values in run index ($Block_{1-1}$, $Block_{3-1}$ and $Block_{2-1}$). These blocks are inserted into the tree. Then, all minimum values lower than or equal to the next minimum value in the run-index (value 14 in $Block_{4-1}$) are evicted into the final sorted file. Once done, MONTRES tries to load the next block of the run-index. As no input buffer is empty, the algorithm frees two input buffers by merging half the free input buffers.

4 I/O Analysis of MONTRES Algorithm

In this section we will discuss the I/O cost of MONTRES.

4.1 I/O Cost Analysis

The I/O cost of the MONTRES algorithm consists of two sub-costs: the run generation phase cost and the run merge phase cost, both in terms of read and write operations.

TABLE 3
Notations for the I/O Cost of MONTRES

Notation	Definition
P_W	Additional writes for merged on-the-fly data
P_R	Additional read operation during run generation phase
α	Additional read operation during run merge phase
G	Evicted data thanks to ascending block selection

4.1.1 Run Generation Phase I/O Cost

The run generation phase for MONTRES begins by creating the min-index data structure. This operation requires the whole input file to be scanned, which results in N read operations: $N \cdot R$.

Once the min-index has been created and sorted, the run creation begins. During this step, MONTRES performs N read operations to load chunks of data from the input file and N write operations to output the sorted data either to the intermediate runs or to the final sorted file in the case of minimum values (see example in Figs. 5b and 5c).

Performing merge on-the-fly during run generation induces P_R additional reads, which are related to reading minimum values from intermediate run blocks (Fig. 6f), and P_W additional writes for writing data already merged on-the-fly into the final sorted file (Fig. 6g), see Table 3 for notations. The run generation I/O cost is given by

$$RGRC + RGWC = (2 \cdot N + P_R) \cdot R + (N + P_W) \cdot W. \quad (6)$$

We will discuss P_R and P_W in what follows. For simplicity, we will not focus on the run expansion mechanism, so we fix the run size to M (the worst case).

In the example given previously (see Fig. 6), during the merge on-the-fly corresponding to sorting the second chunk ($Block_6$ and $Block_3$), MONTRES needed to read data from the first run. This was performed in order to merge with values of the current (second) chunk that were lower than the next minimum ($next\_min\_value = 17$). So values from the memory lower than or equal to 17 were merged with values 11 and 12 from the first block of the first run. Of course, if there were more values smaller than $next\_min\_value = 17$ in some other blocks of Run1, they would also have been merged. In the same way, if there were more runs already sorted that contain values smaller than $next\_min\_value = 17$, MONTRES would have read the blocks of those runs to proceed with the merge on-the-fly. For each new chunk to be sorted, the process is repeated.

One can note that P_R is highly dependent on the input data distribution of the file to sort (and chunks). So, if we have a random uniform distribution, there is a high probability that small values are scattered among different blocks of previously sorted runs. This means that many read operations will be performed. Indeed, for each chunk, previously sorted run blocks are read. So, the efficiency will be poor as only a small number of values (per chunk) will be effectively evicted (merged on-the-fly) to the sorted file.

On the other hand, if data are not randomly distributed, which may happen when data are partially sorted, then merge on-the-fly will perform less additional read operations but with higher efficiency, giving more merged data and thus a higher P_W. In fact, each block will either be read

a small number of times (each read will merge a large number of values) before being emptied or will not be read at all. In this case, P_R will be small and P_W will be high.

We can see that, in both previous cases, there is an additional I/O cost (compared with mergesort). However, in the second case, there are more merged values. This contributes to drastically decreasing the run merge phase I/O cost as it will be shown in the experimental section.

4.1.2 Run Merge Phase I/O Cost

During the run generation phase, a small set of data are directly evicted to the final sorted file by the ascending block selection, noted G (see Fig. 5b), while more data are evicted with the help of the merge on-the-fly mechanism (P_W). The reason why G does not appear in Equation (6) is that it does not imply additional I/Os because those values would either be evicted to the sorted file or to the run file (G is part of the written N data). So, the run generation phase outputs $P_W + G$ blocks of data into the final sorted file.

This amount of data is not candidate to the run merge phase. Then, the number of blocks in the run-index that are candidate to the merge is $N - P_W - G$.

MONTRES makes use of the run-index to select blocks to merge. It selects the blocks containing the lowest values first and merges them together and then continues in this way until all the blocks of the runs are processed.

As explained in Section 3.2, if the data are uniformly distributed among the run blocks, then some run blocks might be read more than once in order to be fully merged.

We denote by α the number of additional block read operations during the run merge phase. When the number of generated runs ($\frac{N}{M}$) is lower than M, then we have enough memory space to hold at least one block from each run during the merge, in this case $\alpha = 0$. Otherwise, when $\frac{N}{M} > M$, some blocks will be read more than once from the intermediate runs, depending on their data distribution.

Contrary to the traditional run merge algorithm which operates in $N \cdot \lceil \log_{M-1} \frac{N}{M} \rceil$ passes, MONTRES operates in a single pass. Each one of the $N - P_W - G$ blocks of the intermediate data that are candidate to the run merge phase is read at least once and written once.

The I/O cost of the run merge phase is then given by

$$RMRC + RMWC = (N - P_W - G + \alpha) \cdot R + (N - P_W - G) \cdot W. \quad (7)$$

One can note that in case of a uniform distribution, as discussed in previous section, P_W will be low. In addition, G will also be low because a small number of values will be directly evicted to the sorted file during the ascending block selection, M values (number of blocks per chunk) per chunk in the worst case. Concerning the value of α, the worst case happens when values are fully interleaved in the file to be sorted. In this case, during the run creation step, each run will contain B times M successive values. As a consequence, during the run merge phase, for each block read operation, at least M values will be evicted to the final sorted file (without considering buffer-related optimization). So α will be bounded by $N \cdot \frac{B}{M}$.

Overall, the cost of run merge will be higher for uniform distributions, but the cost remains reasonable as it will be discussed in the next section.

4.2 Discussion About the I/O Cost of MONTRES

Equation (6) shows that MONTRES performs $N + P_R$ additional read and P_W additional write operations during the run generation phase compared with the traditional external mergesort algorithm. These additional I/O operations make it possible to reduce the amount of data candidate to the run merge phase by $P_W + G$ blocks.

The total I/O cost of MONTRES is given by summing Equations (6) and (7)

$$IO\_cost = (3 \cdot N + P_R - P_W - G + \alpha) \cdot R + (2 \cdot N - G) \cdot W. \quad (8)$$

The I/O cost of MONTRES depends on the data distribution within the input run blocks. The worst case happens when the input file data is random with a uniform distribution. In this case, run generation only outputs the minimum of each block. So N (total number of blocks) values will be evicted to the sorted file during this first phase. As a consequence, G will be equal to $\frac{N}{B}$, which is very small compared with the overall number of blocks, and is thus negligible ($G \to 0$). P_W will also converge to zero as the merge on-the-fly will be inefficient. α will be equal to $N \cdot \frac{B}{M}$ as seen in the previous section. Concerning P_R, even though merge on-the-fly is inefficient, it performs many read operations. In the worst case, during the merge on-the-fly process, one block of each previous run will be read once for each chunk. However, since each read block will remain in the main memory for the next chunk (thanks to the page cache), P_R is approximated to $\frac{N}{M}$.

The resulting upper bound I/O cost is then given by

$$IO\_cost = \left(3 \cdot N + \frac{N}{M} \cdot (1 + B)\right) \cdot R + 2 \cdot N \cdot W. \quad (9)$$

On the other hand, the best scenario for MONTRES happens when minimum (and maximum) values are clustered within blocks. This situation is reached, for instance, when the input file data is partially sorted. In the best case, all values of a sorted chunk will be evicted directly from the main-memory into the final sorted file and no run merge phase will be required. So only Equation (6) applies. In this case, the ascending block selection will be sufficient for sorting all data, so merge on-the-fly will not be triggered; as a consequence: $P_W = 0$, $P_R = 0$, $\alpha = 0$ and $G = N$.

The resulting lower bound overall I/O cost is given by

$$IO\_cost = 2 \cdot N \cdot R + N \cdot W. \quad (10)$$

From Equations (9) and (10), the following inequality gives the lower and upper bound I/O cost for MONTRES

$$2 \cdot N \cdot R + N \cdot W \leqslant IO\_cost \leqslant \left(3 \cdot N + \frac{N}{M} \cdot (1 + B)\right) \cdot R + 2 \cdot N \cdot W. \quad (11)$$

We can see from the inequality (11) that the lower bound of the MONTRES I/O cost is better than the one of mergesort and FSort which is $2 \cdot N \cdot (R + W)$. Indeed, MONTRES theoretically saves at least one overall file write.

Concerning the upper bound I/O costs of mergesort and FSort, they are equal to $N \cdot (1 + \lceil \log_{M-1} \frac{N}{M} \rceil)(R + W)$ and $N \cdot (1 + \lceil \log_{M-1} \frac{N}{2M} \rceil)(R + W)$, respectively. We can

TABLE 4
SSD Performance According to Data Sheets

SSDs	PC SSD	server SSD	DC SSD
Random Reads (IOPS)	100 K	92 K	85 K
Sequential Reads (MB/s)	550	530	550
Random writes (IOPS)	36 K	50 K	43 K
Sequential writes (MB/s)	520	460	300

TABLE 5
Experimented Datasets Statistics

Dataset	Random	A	B	C	D	E
D	1,021	993	7	1,001	1,007	1,013
Entropy	23.1	18.9	2.8	16.4	19.1	20.5
Sorted	0%	0%	0%	80%	60%	40%

distinguish two cases: (1) if $\frac{N}{M} < M$: in this case mergesort and FSort do only one pass and $\alpha = 0$. This gives MONTRES an overall cost of $(3 \cdot N + \frac{N}{M}) \cdot R + 2 \cdot N \cdot W$ while both mergesort and FSort perform in $2 \cdot N \cdot R + 2 \cdot N \cdot W$ meaning at least one file scan less. Thus for small values of $\frac{N}{M}$, and in case of highly random and uniformly distributed data, mergesort and FSort may outperform MONTRES. (2) In the case where $\frac{N}{M} > M$, FSort and mergesort will operate in more than one pass, let us say at least two passes. In this case the cost will be $3 \cdot N \cdot R + 3 \cdot N \cdot W$. Since N is much higher than $\frac{N}{M} \cdot (1 + B)$, and W is at least as costly as R, so MONTRES will always outperform mergesort and FSort. This will be confirmed through the experimental validation as described in the next section.

5 Experimental Validation

This section describes the experiments conducted to validate the efficiency of the MONTRES.

5.1 Experimental Methodology

We measured the performance of MONTRES by varying four main parameters: (1) the data benchmarks used, (2) the main-memory working space allocated for sorting to have different values of $\frac{N}{M}$, (3) the use of the page cache (direct I/Os or not), and (4) the SSD model.

The performance of MONTRES was compared with four main algorithms: MinSort, natural page run, the traditional external mergesort and FSort. However, we will focus on mergesort and FSort. Indeed, as shown in the comparative study described in Section 2.3, these two algorithms were the most efficient among state-of-the-art algorithms when it came to narrowing the memory workspace while increasing the volume of data to sort.

5.1.1 Measured Metrics

We evaluated the total execution time of the sorting process for each algorithm and considered I/O costs of both the run generation phase and the run merge phase.

5.1.2 Experimental Datasets

We used the TPC-H benchmark [13] to generate datasets of integer values (4 bytes). We experimented with two different datasets: unsorted and partially sorted. Indeed, efficient external sorting algorithms should provide good performance for any initial distribution of input data.

Unsorted datasets were obtained from the TPC-H benchmark. The first dataset (A) was obtained from the column Quantity of the Lineitem table. It contains a large value interval and few redundant values as one can see in Table 5. This translates into a high entropy, but which is still smaller than the fully random case. The value of D is also high (993 for a maximum of 1,024 values per block). The second dataset (B) was obtained with the year information from the Date column of the Orders table. This dataset contains a smaller value interval, and thus many equal values, which leads to small D value and a smaller entropy.

Partially sorted datasets were obtained by updating initially sorted data from dataset (A). We experimented with three configurations, namely dataset (C), dataset (D) and dataset (E) with different update rates: 20, 40 and 60 percent, respectively, as in [10]. As one can observe, the higher the sorted dataset proportion, the lower the entropy. However, the entropy stays high because the values used for the update are random.

5.1.3 Allocation of Main-Memory Working Space

Scalability is one of the key issues that motivated the design of MONTRES. Indeed, external sorting algorithms should be efficient even with a growing $\frac{N}{M}$ ratio.

Sorting operators in DBMS usually compete for main-memory with other operators and other queries. Therefore, the main-memory working space allocated for sorting is limited while the amount of input data is growing larger.

In order to evaluate MONTRES with different values of $R_i = \frac{N}{M}$, we can either tune the N or the M parameter. We chose to experiment with 8 GB data size ($N = 2{,}097{,}152$) for different amounts of main-memory working space (M).

Block size was set to system page size, 4,096 Bytes. Therefore, each block held $B = 1{,}024$ integer values.

In the experiments, we used seven different values of $R_i = \frac{N}{M}$: $R_1 = 8$, $R_2 = 32$, $R_3 = 64$, $R_4 = 128$, $R_5 = 512$, $R_6 = 2{,}048$ and $R_7 = 8{,}192$. While most state-of-the-art studies, such as [12], experimented with $\frac{N}{M}$ lower than 512, we chose to test with higher values (2,048 and 8,192) to prove the scalability of MONTRES.

5.1.4 Hardware and Operating System Configurations

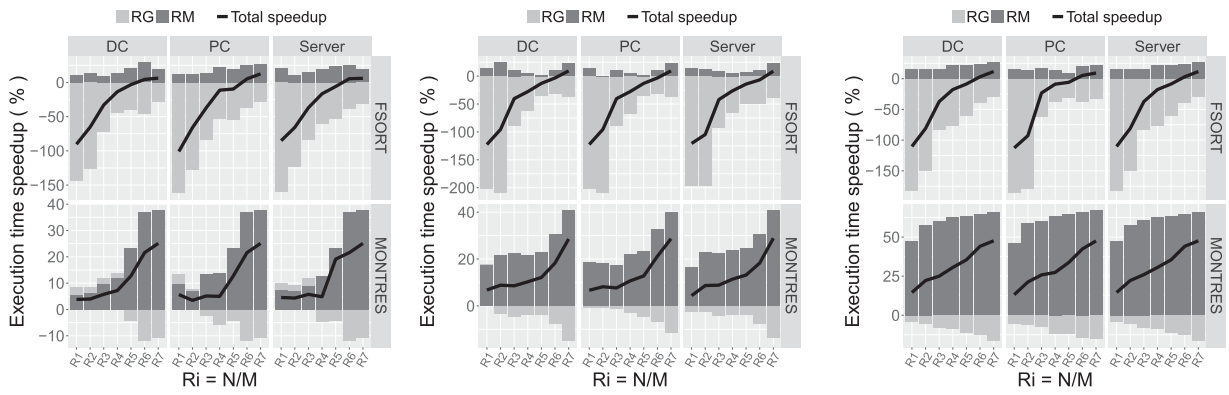
As some traditional SGBDs use direct I/Os, we decided to experiment with and without this option to better evaluate the performance of MONTRES.

We experimented with three different SSDs from different categories: *PC SSD*: entry-level *SSD 850 PRO series* designed for personal computers [18], *Server SSD*: *SSD 845 DC PRO MZ-7WD400EW* designed for server machines [19], and *Data Center (DC) SSD*: *intel DC serie S3710* designed for data-centers [20]. Table 4 summarizes SSD performances.

We used a GNU Linux running on DELL precision T7910 server machine with 32 GB of main-memory and 2 Intel Xeon E5-2630v3 processors clocked at 2.1 GHz.

5.2 Experimental Results

In this section, we will first present the results of our experiments on MinSort and Natural page run. These results confirm that for many workloads, these algorithms cannot be



(a) Dataset A (b) Dataset B (c) Dataset C

Fig. 9. Execution time speedup for MONTRES and FSort compared with mergesort for Datasets (A), (B), and (C) for DC, PC and server SSD.

used. In the rest of the experimental part of this paper, we will focus on mergesort and FSort. Then, we will discuss results obtained with datasets (A) and (B), dataset (A) being close to the worst case for our algorithm because its data are unsorted and uniformly distributed over the blocks compared with dataset (B). After this, we will present results obtained with partially sorted data (datasets (C), (D), and (E)). Finally, we will investigate the contribution of each optimization of MONTRES with regards to the dataset.

5.2.1 Comparison with MinSort and Natural Page Run

MinSort. MinSort relies on value redundancy to optimize the sorting algorithm by saving the run merge phase. For dataset (B), MinSort performed well (comparable to MONTRES and mergesort). Indeed, as already discussed, dataset (B) is characterized by value redundancy as it contains the year information from the Date column of the Orders Table. However, when run with dataset (A), which has very few redundant values, performance dropped drastically. In fact, while MONTRES sorted 8 GB of data in less than 2,400 seconds, MinSort only sorted 8 MB in the same duration. Thus, we decided not to consider it in the rest of this section.

Natural Page Run. Natural page run relies on finding blocks containing data values that do not overlap. It uses some heuristics to find those blocks, which requires a large amount of computing power. We tested Natural page run and observed that it gave very bad results compared with the other algorithms (FSort, mergesort and MONTRES) in most cases. For instance, in the case of dataset (A), the sorting time more than doubled compared with MONTRES. For this reason, we did not consider this algorithm in the rest of the experimental work.

5.2.2 Results on Unsorted Datasets

Results with Dataset (A). Fig. 9a shows MONTRES and FSort execution time speedups for dataset (A) with different ratios of $\frac{N}{M}$ using the experimented SSDs.

The figure shows that in the best case, MONTRES algorithm speeds up the external mergesort by 25 percent while FSort enhances it by a ratio of 8 percent for $\frac{N}{M} = 8{,}192$ (R_7). The worst case of MONTRES occurs when the ratio $\frac{N}{M}$ is very low, in case of $\frac{N}{M} = 8$ (R_1), the MONTRES algorithm only accelerates the external mergesort by 5 percent.

One can observe that MONTRES increases the run generation phase I/O cost by 4 percent on average, considering all SSDs, when $\frac{N}{M} > 64$ (R_4 and more) and speeds up the run merge phase by 20 percent, while FSort overloads the run generation phase by 71 percent on average and enhances the run merge phase by 19 percent. When $\frac{N}{M} = 8, 32$ and 64, MONTRES speeds up the run generation by 2 percent on average in addition to the speedup of run merge which is about 7 percent on average.

MONTRES run generation overhead is due to additional read/write operations performed by the ascending block selection and merge on-the-fly mechanisms. FSort run generation overhead is mainly due to additional CPU operations performed to create larger sorted runs.

We can also note that the narrower the main-memory working space is (higher $\frac{N}{M}$), the lower the run generation execution time and the higher the run merge execution time become. When the main memory working space is large, the MONTRES merge on-the-fly mechanism processes then produces more data to the final sorted file. This increases the run generation overhead and enhances the run merge phase. However, with larger working space, FSort sorts more data during the run generation, which results in larger runs. This increases the run generation phase I/O cost but accelerates the run merge phase.

MONTRES outperforms FSort during the run merge phase because it reduces the amount of intermediate data to merge and performs the run merge phase in a single pass while FSort does this in $\log_{M-1}\frac{N}{2M}$ passes.

Table 6 gives the measures obtained on additional reads performed by MONTRES, P_R, on the intermediate runs, and the amount of data written to the final sorted file at the end of the run generation phase, $P_W + G$. The table shows that, at the end of the run generation phase, MONTRES outputs 6,7 percent of the input file size into the final sorted file by performing only 2.0 percent of additional reads on the intermediate run blocks.

As one can see in Fig. 9a, when $\frac{N}{M} = 8$, the speedup of MONTRES compared with mergesort for the run merge phase is 7.7 percent. When $\frac{N}{M} = 8{,}192$, this speedup increases

TABLE 6
I/O Cost Measure for MONTRES with Dataset (A)

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + G(\%N)$	7.3	7.1	7.0	6.9	6.7	6.7	6.6
$P_R(\%N)$	1.8	1.9	1.9	2.0	2.0	2.0	2.3

TABLE 7
I/O Cost Measure for MONTRES with Dataset (B)

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + G(\%N)$	24.2	24.2	24.1	24.3	24.3	24.3	24.1
$P_R(\%N)$	8.2	8.3	8.0	8.0	8.1	8.1	8.1

and reaches 37 percent. On the other hand, the amount of data written into the final sorted file, $P_W + G$, is stable whatever the value of $\frac{N}{M}$. This is because the higher $\frac{N}{M}$, the better the performance of the MONTRES run merge phase compared with mergesort and FSort. In effect, MONTRES operates in one pass while the number of passes increases according to $\frac{N}{M}$ for the two other algorithms (see Section 5.2.5).

It might seem counterintuitive that MONTRES outperforms both mergesort and FSort when $\frac{N}{M}$ is low for dataset (A) with respect to the I/O cost analysis in Section 4.2. In fact, dataset (A) is not fully random or uniform (see Section 5.1.2). We experimented with a synthetically generated fully random dataset and, in this case, it turned out that MONTRES increased the execution time by around 8 percent for small $\frac{N}{M}$ ratios (using direct I/Os so as not to rely on the page cache). In addition, system profiling identified three other improvements: (1) CPU time of MONTRES is better as it performs less comparisons (more than 10 percent difference), (2) MONTRES performs less in-memory operations, (3) MONTRES takes better advantage of the system page cache. For the latter, we will see in Section 5.2.4 that, with Direct I/Os, the performance of MONTRES can be lower.

Results with Dataset (B). Fig. 9b presents execution time speedup of tested algorithms when sorting dataset (B) with different ratios of $\frac{N}{M}$. One can observe that MONTRES speeds up mergesort by 28 percent while FSort decreases the execution time by 8 percent for $\frac{N}{M} = 8{,}192$ (R_7).

Similar to the results observed with dataset (A), both MONTRES and FSort increase the run generation phase I/O cost and speed up the run merge phase.

We also noticed that MONTRES execution times were lower when sorting dataset (B) compared with dataset (A). In fact, as dataset (B) contains more redundant values, the merge on-the-fly mechanism output more data into the final sorted file and reduced the amount of intermediate data candidate to the run merge phase.

Table 7 gives the measures on additional reads, P_R, performed by MONTRES on the intermediate runs, and the amount of data written on the final sorted file at the end of the run generation phase, $P_W + G$. The table shows that at the end of the run generation phase, MONTRES outputs 24.3 percent of the input file size into the final sorted file by performing 8 percent of additional reads on the intermediate run blocks, which is a very good result.

The amount of data written into the final sorted file at the end of the run generation phase is higher with dataset (B) than with dataset (A). Both datasets have a random distribution, but dataset (B) has a small interval of values, resulting in higher density of neighboring values within a block. This allows to merge on-the-fly more data values into the final sorted file (see Section 5.2.5). In a similar way to dataset (A), the higher the ratio $\frac{N}{M}$, the better the performance of MONTRES compared with mergesort.

5.2.3 Results with Partially Sorted Datasets

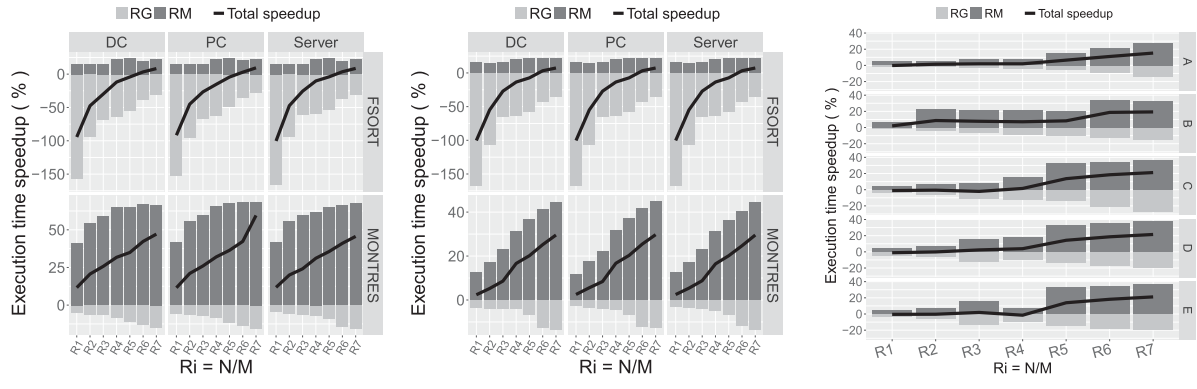
Figs. 9c, 10a and 10b illustrate the execution time speedup of the tested algorithms when sorting dataset (C) with 20 percent updates, dataset (D) with 40 percent updates and dataset (E) with 60 percent updates, respectively, using the server SSD.

The results show that the MONTRES algorithm enhances the external mergesort by 45 percent while FSort does so by 11 percent when the ratio $\frac{N}{M} = 8{,}192$ (R_7, averaged over the three datasets).

We observe that MONTRES and FSort increase the run generation phase execution time and reduce the run merge phase. In Fig. 9c the MONTRES algorithm increases the run generation phase by 9.5 percent (8.5 percent in Fig. 10a and 6.7 percent in Fig. 10b) and speeds up the run merge phase by 60 percent (59 percent in Fig. 10a and 29 percent in Fig. 10b) on average, while FSort increases the run generation phase cost by 88 percent (71 percent in Fig. 10a and 66 percent in Fig. 10b) and speeds up the run merge phase by 20 percent on average (19 percent in Fig. 10a and 17 percent in Fig. 10b).

The run generation overhead is higher with dataset (C) than with the others as more data are partially sorted, MONTRES performs a better merge on-the-fly that outputs more data to the final file. FSort performs additional CPU operations to expand the run size.

We observe that MONTRES reduces the run merge I/O cost better than FSort. This is because MONTRES performs only one pass, while FSort decreases the number of generated runs. Unfortunately, decreasing the number of runs



(a) Dataset D (b) Dataset E (c) Direct IO on server SSD

Fig. 10. Execution time speedup for MONTRES and FSort compared with mergesort for Datasets (D), (E) for DC, PC and server SSD and for Direct I/Os.

TABLE 8
Average I/O Cost Measure for MONTRES with Dataset C, D and E

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + G(\%N)$	45.7	45.6	45.4	45.3	44.2	44	43.8
$P_R(\%N)$	11.0	11.1	11.2	11.2	11.8	12.1	12.3

does not always reduce the number of merge passes. For example, when the ratio $\frac{N}{M}$ equals 128 or 256, FSort and mergesort perform the run merge with the same number of passes resulting in the same I/O cost.

Table 8 gives the average measures P_R and $P_W + G$, for datasets (C), (D), and (E). As expected, results show that the amount of data written to the sorted file is much larger with partially sorted data than with unsorted data.

Contrary to datasets (A) and (B), where data are partially sorted, the speed up is due to the three optimizations of MONTRES: merge on-the-fly mechanism, run merge algorithm, and run expansion policy. As the data is partially sorted, the minimum values will be clustered within some blocks and maximum values grouped in others. Then, the run expansion policy can easily create longer runs, contributing to the overall speedup.

5.2.4 Direct I/O

Fig. 10c illustrates the speedup of MONTRES compared with mergesort for datasets A, B, C, D and E, bypassing Linux page cache (using server SSD)

We can observe that the worst case for MONTRES occurs with dataset (A) for $\frac{N}{M} = 8$ (R_1). Here, MONTRES gives an execution time 1.13 percent higher than mergesort. In the best case, MONTRES enhances mergesort by 21 percent for dataset (B) with $\frac{N}{M} = 8{,}192$. In fact, when sorting dataset (A) with a low $\frac{N}{M}$, the merge on-the-fly mechanism performs many additional I/O operations to output a small number of values (see Section 4.2).

As we can see in Fig. 10c, except for dataset (B), the speedup of MONTRES is rather small, especially when the number of merge passes of mergesort equals $\log_{M-1}\frac{N}{M} = 1$. This is true for $\frac{N}{M}$ ratios equal to $8, 32, 64, 128, 512$. In this case, MONTRES only enhances mergesort execution time by 12 percent on average when sorting datasets A, C, D and E, while the speedup for dataset B is 19 percent on average.

We also observe that the run generation overhead is doubled when $\frac{N}{M} = 2{,}048$ and 8192 compared with the case using the page cache. This is because the additional read operations during the run generation phase are performed without benefiting from the page cache.

Disabling the page cache slows down MONTRES execution time by $11, 25, 40, 32, 17$ percent on average, for datasets A, B, C, D and E, respectively compared with the case that relies on the page cache. However, Montres still enhances the performance of mergesort in nearly all cases.

5.2.5 MONTRES Optimization Analysis

Table 9 shows the proportion of speedup brought about by each optimization of MONTRES on the different datasets according to $\frac{N}{M}$. We performed experiments by removing the optimizations one by one (as each experiment was conducted independently, the sum of the optimizations only converges toward 100 percent).

As one can observe, the run expansion only contributes to the overall speedup for partially sorted data, here dataset (C), while the overall speedup for datasets (A) and (B) is close to zero. This is because most blocks of datasets (A) and (B) have overlapping values between blocks.

On the other hand, the merge on-the-fly mechanism brings around 100 percent of optimization for low values of $\frac{N}{M}$ for all datasets and this proportion decreases for the last columns thanks to the run merge phase optimization.

The proposed run merge optimization contributes to the overall speedup when ratios of $\frac{N}{M}$ are high, meaning that the run merge phase operates with several passes for mergesort and FSort while it does in one pass with MONTRES.

6 Conclusion

Due to the exponentially growing volume of data being generated, sorting algorithms need to adapt to become more scalable when dealing with large datasets under memory pressure. In this paper, we analyzed state-of-the-art algorithms and concluded that their scalability was an issue. This was the motivation behind the design of MONTRES.

MONTRES relies mainly on two optimizations for the run generation phase: merge on-the-fly and run expansion, and one optimization for the run merge phase, which is a one-pass algorithm regardless of the ratio $\frac{N}{M}$.

We evaluated the I/O cost of MONTRES with different $\frac{N}{M}$ ratios, data distributions, and SSDs. MONTRES outperformed traditional algorithms by reducing execution times by more than 40 percent when $\frac{N}{M}$ is high compared with mergesort and FSort. MONTRES will be soon integrated into a commercial DBMS.

TABLE 9
MONTRES Speedup Measured Separately for Each Optimization on the Datasets A,B and C

	Optimisations	R_1	R_2	R_3	R_4	R_5	R_6	R_7
Dataset A	Run expansion	0%	0%	0%	0%	0.8%	1.6%	1.45%
	Merge on-the-fly	98.9%	98.5%	100%	99.2%	98.8%	37%	32%
	Run Merge	0%	0%	0%	0%	0%	62.12%	63.12%
Dataset B	Run expansion	0%	0%	0%	0%	0%	0.01%	0.02%
	Merge on-the-fly	100%	100%	100%	100%	100%	49.63%	54.65%
	Run Merge	0%	0%	0%	0%	0%	45.81%	49.16%
Dataset C	Run expansion	1.36%	1.78%	2.44%	2%	3.22%	2.8%	3.13%
	Merge on-the-fly	97.9%	96.1%	98.1%	98.3%	98.9%	49.94%	38.07%
	Run Merge	1.20%	1.41%	1.74%	2.17%	2.30%	36.87%	59.20%

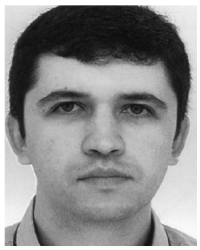
As a perspective for future work, it would be interesting to study the impact of page cache mechanisms on different sorting algorithms to see how to benefit from prefetching and write back algorithms tailored for SSDs (such as [21]) to further reduce the I/O cost. We would also like to investigate the effects of such sorting algorithms on NVMs, which have yet another performance model.

Acknowledgments

This work is supported by the French National Association of Research and Technology (ANRT).

References

[1] A. Thusoo, et al., "Data warehousing and analytics infrastructure at Facebook," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1013–1020.
[2] O. Mutlu, "Main memory scaling: Challenges and solution directions" in *More than Moore Technologies for Next Generation Computer Design*. New York, NY, USA: Springer, 2015, pp. 127–153.
[3] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Upper Saddle River, NJ, USA: Pearson Education, 1998.
[4] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surveys*, vol. 38, no. 3, Sep. 2006, Art. no. 10.
[5] W. Zhang and P.-A. Larson, "Dynamic memory adjustment for external mergesort," in *Proc. 23rd Int. Conf. Very Large Data Bases*, 1997, pp. 376–385.
[6] V. Estivill-Castro and D. Wood, "Foundations for faster external sorting," in *Foundation of Software Technology and Theoretical Computer Science*. Berlin, Germany: Springer, 1994, pp. 414–425.
[7] J. Boukhobza and P. Olivier, *Flash Memory Integration: Performance and Energy Issues*. Amsterdam, The Netherlands: Elsevier, 2017.
[8] J. Boukhobza, "Flashing in the cloud: Shedding some light on NAND flash memory storage systems," in *Data Intensive Storage Services for Cloud Environments*. Hershey, PA, USA: IGI Global, Apr. 2013, pp. 241–266.
[9] H. Park and K. Shim, "FAST: Flash-aware external sorting for mobile database systems," *J. Syst. Softw.*, vol. 82, no. 8, pp. 1298–1312, 2009.
[10] Y. Liu, Z. He, Y.-P. P. Chen, and T. Nguyen, "External sorting on flash memory via natural page run generation," *Comput. J.*, vol. 54, no. 11, pp. 1882–1990, 2011.
[11] T. Cossentine and R. Lawrence, "Efficient external sorting on flash memory embedded devices," *Int. J. Database Manage. Syst.*, vol. 5, no. 1, pp. 1–20, 2013.
[12] J. Lee, H. Roh, and S. Park, "External mergesort for flash-based solid state drives," *IEEE Trans. Comput.*, vol. 65, no. 5, pp. 1518–1527, May 2016.
[13] T. P. P. Council, "TPC-H benchmark specification," 2008. [Online]. Available: http://www.tpc.org/tpch/
[14] P. Andreou, O. Spanos, D. Zeinalipour-Yazti, G. Samaras, and P. K. Chrysanthis, "FSort: External sorting on flash-based sensor devices," in *Proc. 6th Int. Workshop Data Manage. Sensor Netw.*, 2009, pp. 10:1–10:6.
[15] G. Graefe, "Sorting in a memory hierarchy with flash memory," *Datenbank-Spektrum*, vol. 11, no. 2, pp. 83–90, 2011.
[16] L. M. Grupp, et al., "Characterizing flash memory: Anomalies, observations, and applications," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 24–33.
[17] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Commun. ACM*, vol. 29, no. 10, pp. 996–1000, 1986.
[18] SSD 850 Pro. [Online]. Available: http://www.samsung.com/fr/business/business-products/ssd/850-pro-series/MZ-7KE256BW, Accessed on: Mar. 07, 2017.
[19] SSD 845 DC Pro - MZ-7WD800. [Online]. Available: http://www.samsung.com/fr/business/business-products/ssd/845-dc-pro/MZ-7WD800EW, Accessed on: Mar. 07, 2017.
[20] Intel DC S3710. [Online]. Available: http://www.intel.fr/content/www/fr/fr/solid-state-drives/solid-state-drives-dc-s3710-series.html, accessed on: Mar. 07, 2017.
[21] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: A learning linux prefetching mechanism for SSD performance model," in *Proc. 5th Non-Volatile Memory Syst. Appl. Symp.*, 2016, pp. 1–6.



Arezki Laga received the engineering (with Hons.) degree in computer science from Ecole nationale Supérieure d'Informatique (E.S.I), Algiers, Algeria, in 2012 and the MSc degree in computer science from the University Bretagne Occidentale, France, 2014. He is now working toward the PhD degree in computer science with the University Bretagne Occidentale. His current research interests include data processing algorithms, database systems, operating system design, flash memory, and SSD.

Jalil Boukhobza received the electrical engineering (with Hons.) degree from the Institut Nationale d'Electricité et d'électronique (I.N.E.L.E.C), Boumerdès, Algeria, in 1999, and the MSc and PhD degrees in computer science from the University of Versailles, France, in 2000 and 2004, respectively. He was a research fellow at the PRiSM Laboratory (University of Versailles) from 2004 to 2006. He has been an associate professor with the University Bretagne Occidentale, Brest, France, since 2006 and is a member of Lab-STICC. His main research interests include flash-based storage system design, performance evaluation and energy optimization, and operating system design. He works on different application domains such as embedded systems, cloud computing, and database systems.

Frank Singhoff received the engineering degree in computer science from the CNAM, Paris, in 1996 and the PhD degree from Télécom-ParisTech, in 1999. He is a full professor of computer science in the Lab-STICC, University Bretagne Occidentale, France. His current research focuses on real-time scheduling theory and architecture description languages. In 2002, he started Cheddar, a toolset designed to perform real-time scheduling analysis. He is also a member of the SAE AS-2C committee working on the AADL. He received an ACM SIGAda Outstanding Ada Community Contributions Award in 2010.

Michel Koskas received the graduate degree in mathematics from Ecole Normale Supérieure and the PhD degree in mathematics. He is a researcher in the Institut National de la Recherche Agronomique (INRA), Paris, France. He is specialized in algorithmics. His current research include DNA sequencing algorithms and data management. He also designs statistical algorithms for clustering, scoring, classification, etc. He is chief science officer for a software company.