Hindawi
Mobile Information Systems
Volume 2022, Article ID 3354935, 9 pages
https://doi.org/10.1155/2022/3354935

Research Article

ISort: SSD Internal Sorting Algorithm for Big Data

Yang Liu,[1] **Wenhan Chen**,[1] **Xuran Ge**,[1] **Zhiguang Chen**,[2] **Yang Ou**,[3] **and Nong Xiao**[1]

[1]*Institute for Quantum Information, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Hunan Province, Changsha City 410000, China*
[2]*Sun Yat-sen University, Guangzhou Province, Guangzhou City 510000, China*
[3]*Institute of Computing Technology, College of Computer, National University of Defense Technology, Hunan Province, Changsha City 410000, China*

Correspondence should be addressed to Yang Ou; michaelouyang@163.com

Received 20 May 2022; Revised 20 November 2022; Accepted 26 November 2022; Published 15 December 2022

Academic Editor: Yugen Yi

As a basic algorithm for big data processing, external sorting suffers from massive read and write operations in the external memory. Recent works separate part of the data processing work from the host side to the solid state drive (SSD) to reduce data transmission. However, the internal memory of the SSD is limited, and undesirable data retention could occur during the merge phase. Therefore, to improve the efficiency of memory, we propose an algorithm named ISort. Specifically, we build an index table between the memory and the address. The index table determines the order of pages being read in the merge phase according to their minimum values, which are read into memory sequentially to reduce the data residing in memory and improve memory efficiency. Since the merge phase is performed inside the SSD, ISort can take advantage of the high IO bandwidth within the SSD to speed up the execution of the merge phase. We search for the optimal ratio of read and write channels by comparing the "specialized channel" and the "hybrid channel" for data of read and write performance because the utilization of the channel will directly influence performance. Experimental results show that ISort can maintain better data processing speed when SSD memory is limited, outperforming other robust algorithms. In addition, the algorithm's performance using the crossover strategy is better than that using the specialization strategy.

1. Introduction

The development of storage technology and cloud computing has made it possible to process terabytes and petabytes of data [1]. However, the widespread use of data-intensive applications and personal mobile devices generates massive amounts of data, estimated to reach 185 zettabytes in 2025 [2]. Therefore, how to quickly mine valuable information from massive data has become an urgent problem to be solved in the era of rapid data growth.

External sorting is one of the most fundamental algorithms in data management systems, being used to deal with the situation that the main-memory capacity cannot hold all the data when data volume is too large. For example, in the MapReduce framework of Hadoop, a large number of external sorting is exploited to sort intermediate data and the final data in both operations of mapping and reducing [3]. Another instance is that external sorting plays a critical role in the database query procedure since a large amount of data is often involved in finding the desired results [4]. External sorting contains two phases: the run generation and the run merge. In the first phase, the input data are divided into blocks that can hold the memory capacity and then loaded into the memory for sorting. Sorted data blocks (Run) are then written back to the storage device. In the second phase, multiple runs are merged into a fully sorted chunk of data [5], which will require lots of I/O operations, resulting in I/O overhead. Therefore, the I/O time is critical in the external merge sorting elapsed time.

Traditional external sorting algorithms are mainly designed for hard disk drives (HDDs) characterized by slow speed, high power, and poor earthquake resistance. In contrast, SSDs have more obvious advantages such as no robotic arm, random access, high read and write bandwidth,

earthquake resistance, low energy consumption, high stability, long service life, and no noise. [6]. With the development of flash memory technology and the price reduction, SSDs are gradually replacing HDDs in the storage market [7]. However, external sorting is I/O-intensive because there are many read/write operations on storage devices in the execution process, which affects the performance of the algorithm and the service life of the SSD. To solve this problem, experts have tried to transfer computing to the SSD, called computing and storage fusion.

Many efforts have been made towards external sorting. Reference [8] makes use of the computing resources in SSD to accelerate deep learning. The blueDBM architecture [9] accelerates data queries in SSD computing. Reference [10] unloaded the external sorting work to SSD. In Reference [11], source data are divided into multiple blocks and sorted separately in memory, and the merge work begins when there is an access request. This approach uses the channel parallelism of SSD but does not consider the situation that the data are partially sorted. All the above methods suffer from the issue that when the pages of big and smaller data are read simultaneously, the big data will remain in memory for a long time, reducing the memory utilization. To tackle this issue, we build an index table to record the minimum value of each run for each block that is sequentially read to the input buffer and merged within the SSD. The channel congestion problem caused by the read/write rate is also discussed. In summary, our major contributions can be summarized as follows:

(i) We present a new external sorting algorithm named ISort that implements rapid sorting within the SSD. For partially sorted data, it records the minimum values of sorted blocks and indexes them to determine the order for merging. By avoiding the extended storage of large values in memory, ISort can enhance the internal memory utilization of SSD and significantly improve external sorting performance.

(ii) The specific proportion adjustment of SSD hardware equipment is carried out during the operation of ISort algorithm. We find the best ratio of parallel channel read-write numbers by comparing the effects of different ratios of the read/write channel on external sorting.

(iii) The experimental results show that ISort has better read and write performance than previous works. For example, ISort improves the read and write performance when the total amount of data increases. ISort also improves the performance when the data size remains the same and the memory size increases.

The rest of this paper is organized as follows. The background and motivation are introduced in Section 2. Section 3 describes the detailed implementation of ISort and different channel strategies. Simulation experiments are presented in Section 4. Section 5 provides an overview of the related work. The conclusion is presented in Section 6.

2. Background and Motivation

In this section, we first describe the basic external sorting algorithm and general architecture of a typical SSD, then we discuss the motivation of this work.

2.1. External Sorting. Traditional external sorting is generally divided into two phases, as illustrated in Figure 1. Source data are initially in the storage device. The first phase divides the data into accommodating blocks according to the input buffer size. Then, each block is loaded into host memory for sorting, and the sorted data blocks are written back to the storage device. The second phase merges the sorted data blocks generated in the first phase into a sorted output through several iterations [5]. After these iterations, the merge operation will produce multiple read and write operations for the storage devices, resulting in high I/O overhead. Because of the large performance gap between DRAM and storage devices, the I/O times are decisive in the elapsed time of external merge sorting. A critical evaluation factor is for data-intensive applications, whether the data can be processed quickly and allow a response to the results.

2.2. Solid State Drives (SSDs). With the development of flash memory technology and price reduction, SSDs have gradually become the mainstream type of high-performance storage media. SSDs based on flash memory have been widely studied by industry and academia because they provide random access, high speed, high throughput, and low energy consumption.

Considering the architecture, flash memory can be divided into single-level cell flash memory (SLC) and multi-level cell flash memory (MLC) [12]. MLC allows a single storage unit to hold twice as much data, making it cheaper to manufacture. MLC has a slower writing speed, higher power consumption, shorter life, and higher error rate than SLC. SSD can be divided into NOR flash memory and NAND flash memory according to the type. The random read speed of NOR flash memory is fast, but the erase and the programming operation is slow, and its flash capacity is relatively small. NOR flash memory allows random storage, is suitable for frequent read and write situations, and is usually used to store program code. Compared with NOR flash memory, NAND flash memory has a lower cost, higher density, higher capacity, and faster erase and write speed, being suitable for data storage. This article only discusses NAND flash memory, which provides three basic operations:

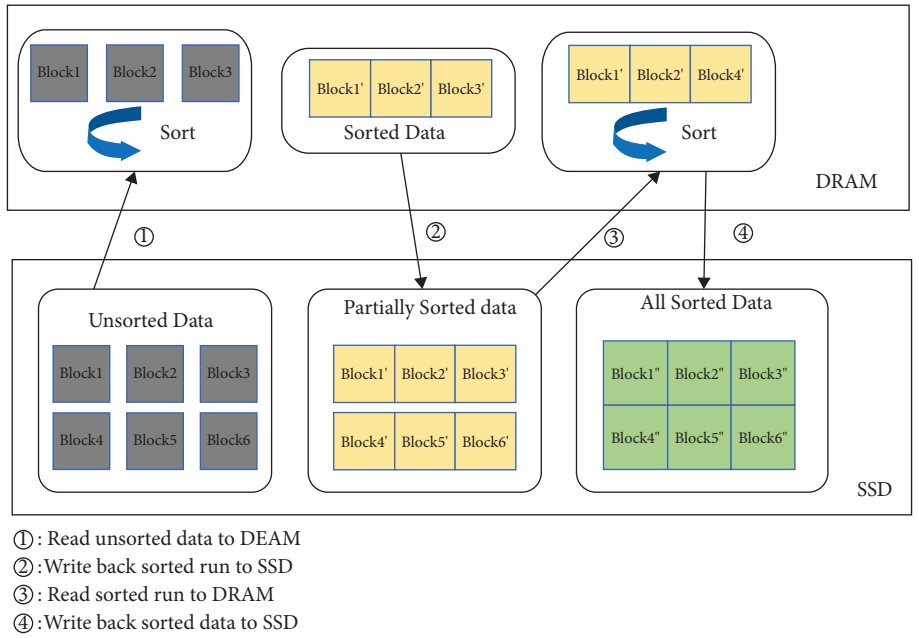

Figure 1: External sorting phases.

(i) Read/write operations: The basic unit of read/write operations is the page, but the erase operation's basic unit is the block. The write operation of flash memory is generally 200–700 μs, approximately ten times that of the read operation.

(ii) Erase operation: The erase operation sets all values on the target block to 1. However, if a flash page has been written, and we want to write the block again, we need to erase it first. This process is called erasing before writing [13]. The delay of the erase operation is about 2–3 ms longer than that of the I/O operation. Therefore, frequent erase operations will affect the overall performance.

Flash memory can only be subjected to a limited number of erasures. If the data block is erased frequently, it can no longer be used. The SSD controller adds a transformation layer named the flash translation layer (FTL) to avoid writing after erasure. Flash memory does not support overwriting. FTL writes the new data to other free pages when updating data, and the original data are marked invalid. FTL has three main functions: address mapping, load balancing, and garbage collection [14]. Address mapping can be divided into page mapping, block mapping, and hybrid mapping [15]. The mapping table for block mapping is tiny, being able to reduce memory overhead and offer an excellent response to read requests. Load balancing can improve the performance of the SSD and prolong its service life. Garbage collection [16] periodically reclaims space occupied by invalid data and erases appropriate blocks to recycle free pages.

Figure 2 illustrates the general architecture of a NAND SSD, which is composed of a master controller chip, a set of DRAM, multiple interfaces, and an array of NAND flash memory chips connected to flash controllers by multiple channels.

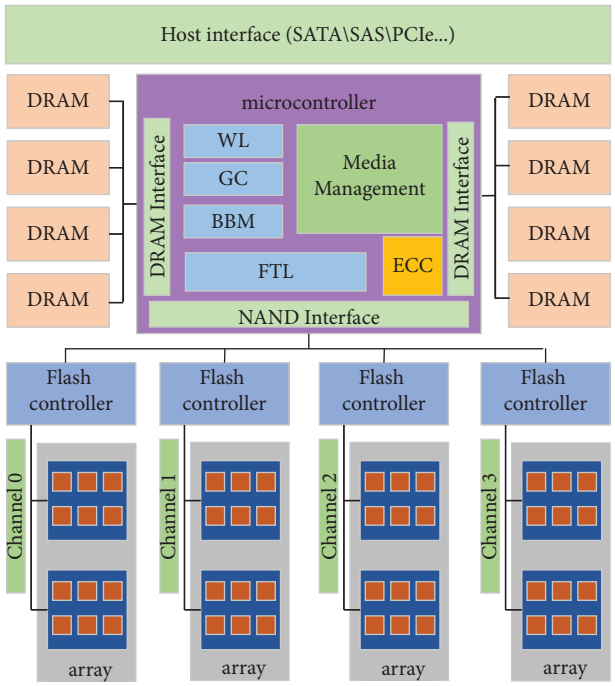

Figure 2: The internal architecture of SSD.

2.3. Motivation. In our practical application, source data usually have data locality. The most recent research algorithms mainly focus on reducing the amount of data transferred between memory and out-of-memory devices. Reference [10] takes advantage of SSDs' internal computing power but does not consider their limited internal memory resources. Completely ignoring the data's characteristics will lead to a large amount of data being stuck in the memory for

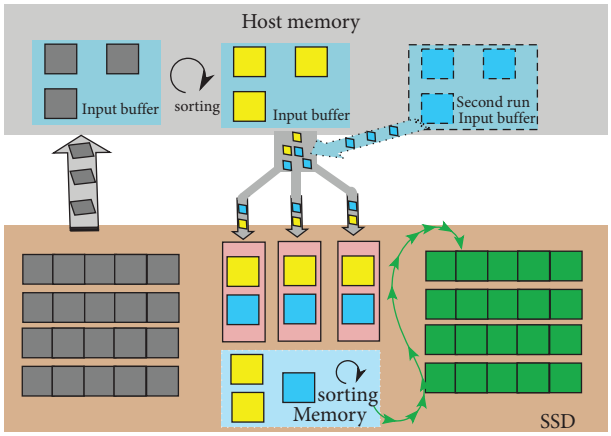

Figure 3: The process of ISort.

Table 1: Algorithm parameters.

Notation	Definition
Q	Size of the unsorted keys in the record
P	Number of pages
B	Number of blocks
p_i^j	Page with identifier i in run j
b_i	Block with identifier $0 \le i \le B-1$
C	Number of channels
M	Available main memory (blocks)
D	Sorted dataset
R	Number of runs
CK_1	The smallest C pages
r_i	The ith run, $1 \le i \le R$
CK_2	The second smallest C pages

a long time. Therefore, we hope to make full use of the internal resources of the SSD and the characteristics of the data itself to achieve the acceleration of the external sorting algorithm.

3. ISort Design

We present ISort, an external sorting, that performs data merging by exploiting the internal hardware infrastructure of SSDs. We introduce its architecture and elaborate on its design techniques.

3.1. Overall Architecture. We propose a new external sorting mechanism called ISort that performs data merging by exploiting the internal hardware infrastructure of SSDs. The traditional external sorting algorithm cannot be transferred to SSD because it uses FTL to process the host-side data request [17], as described in Figure 2. Therefore, we need to change the SSD standard software architecture layer. However, the direct use of the merge sort algorithm will result in the high consumption of memory resources in the SSD, which will significantly impact SSD performance. To solve this problem, we built a page min index to record the minimum values of all pages. The minimum values determine the order of pages entering the input buffer. The whole process of ISort is shown in Figure 3, where the gray block

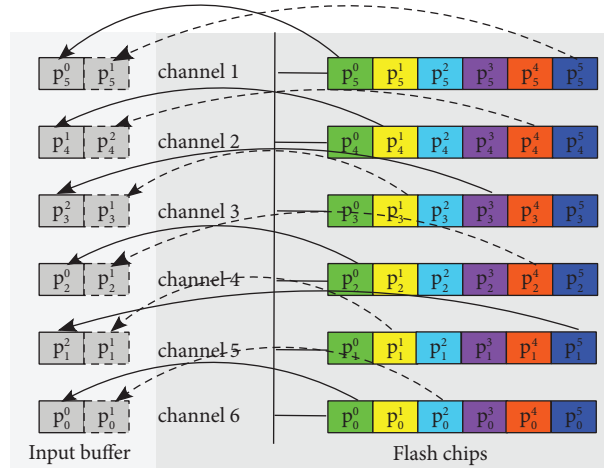

Figure 4: The status of the input buffer and the layout of the flash chips.

represents unsorted data, the yellow and blue block represent the internal ordered data, and the green block represents fully sorted data. We divide ISort into two phases. The first is the run generation phase, which differs from the traditional external sorting in that the data are written back to the storage device. The run merge phase performs operations inside the SSD.

Table 1 defines notations for ISort. The size of the key in the record is Q. Keys are allocated in B blocks, each represented as b_i with $0 \le i \le B-1$. We use M to denote the host memory size assigned to perform the sort. The record is divided into $R = T/M$ parts, indicating the number of runs. P indicates the number of pages in a block; C represents the number of channels; D represents fully sorted datasets; CK_1 represents the minor C pages; and CK_2 represents the second minor C pages.

3.2. Run Generation Phase. Algorithm 1 aims to generate intermediate sorted files called runs. We discard the value in the record because we only need to sort the key. We split Q into S_i, each of which is the size of an input buffer (see lines 2 and 3 in Algorithm 1). To accelerate the speed of writing storage, ISort activates multiple flash channels simultaneously, making full use of the parallelism of the SSD. However, when the key value is skewed in the run, the channel will be blocked, resulting in slower read operations. We slice each run into the page using interlaced write between channels. During the write-back process, we recorded the page minimum index table in the SSD memory, recorded the page's minimum value in each run, and built an index called page index.

3.3. Run Merge Phase. Algorithm 2 describes the run merge phase of ISort. The min index order recorded by Algorithm 1 reads CK_1 into the SSD's internal memory input buffer. Unlike traditional sorting methods, we do not look for a

minor page every run. It is also possible that the parallel page read simultaneously is from the same run. In ISort, the order in which pages are loaded into the SSD's internal memory only follows the min page index. Because of the partial ordering of the data, a page we would like to see may be in high-value runs that will not be read for a long time. These data will not be output after they are read into memory but instead will be output when a more extensive page is encountered. Next, we read CK_2 to the input buffer in order as the buffers for CK_1. By doing so, the data transmission capacity can be better matched with computing power. When a page is consumed, we can supplement the data without affecting the sorting of CK_1. When ISort is satisfied such that there are C pages in memory, the merging process starts synchronizing with the buffer data transfer process. The minor key in the input buffer is copied to the output buffer in each iteration. We used a qsort in memory, and the computational complexity is $O(n)$. We flush the output buffer to a flash chip if the output buffer becomes full. The same run is interlaced on a different channel. Therefore, this process will not occur when a channel does not have a page, except in the final phase. However, parallel read/write operations may cause channel congestion, which is discussed in the experiment section.

Figure 4 illustrates an example. For the convenience of demonstration, we draw six channels and six input buffers to illustrate the merge phase of the ISort algorithm in more detail. Suppose there are six runs interleaved across six channels. We represent them in different colors. The CK_1 in the flash chips is transferred to the input buffer in parallel, as shown in process 1. CK_2 is transferred to the input buffer in parallel, as shown in process 2. When a page in CK_1 is exhausted, the CK_2 page of the same channel in CK_2 is immediately converted to CK_1. At the same time, the reading of the next page is triggered. CK_2 will be continuously converted into CK_1 as it is consumed. At best, CK_1 is distributed on a different channel, and we can implement the concurrent reading of the channel, as shown in process 1. In the worst case, CK_1 is distributed on the same channel, and we can only read it serially, as in the traditional method. Because our data are partially ordered, and the data of each run are cross-placed on a different channel, and the worst-case probability is negligible.

Figure 5 shows six sorted runs. Each run consists of three pages, and each page contains three keys. Let us assume that the input buffer can drop 12 pages, as shown in 4. The middle of Figure 5 shows the traditional method of reading the minimum page of each run into the input buffer. When a page with large values and a page with decimal values appear in the input buffer simultaneously, it will cause long-term retention in memory, thereby reducing memory utilization. The lower part of Figure 5 shows our method. Based on the page-min-index, ISort reads sequentially to avoid the occurrence of pages in the input buffer and improve input buffer space utilization.

4. Experimental Results

4.1. Evaluation Design. This section describes the experimental platform setup and the methodology to evaluate ISort.

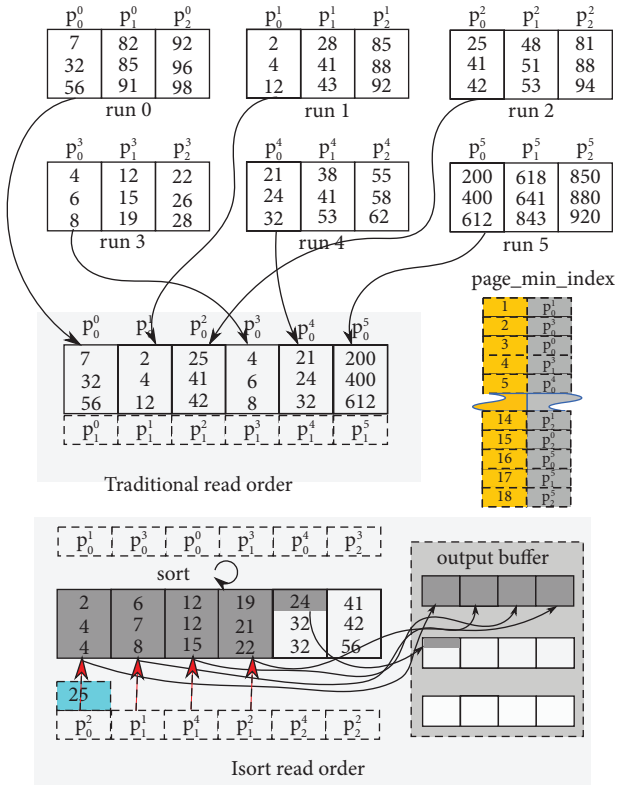

Figure 5: ISort algorithm running diagram.

In the following experiments, we used SSDsim [18], an open-source solid state simulation system that follows the ONFI protocol, having high accuracy and modularization advantages. The hardware configuration parameters of the SSDsim simulator used in this paper are shown in Table 2.

We take ActiveSort as the baseline that includes an additional write-back operation than ISort. The comparison is conducted from the perspective of dataset size and memory. We also evaluate the impact of SSD memory and I/O trace on performance. Also, we use different channel ratios to test the performance of a specialized channels and hybrid channels.

4.2. Experimental Results. Since external sorting is an IO-intensive algorithm, read and write requests are initiated frequently and alternately in the merging phase, as shown in Figure 6. When more channels are used for writing, both the read time (RT) and write time (WT) of ActiveSort increase evidently, while ISort decreases, indicating the superior performance of ISort.

If the read-write request separation processing is carried out and the number of reading channels increases, it will lead to writing request processing congestion. Similarly, reducing the number of reading channels can reduce read request processing congestion. To avoid idle channels and improve channels' resource utilization, we can make all channels read and write requests during the merge phase.

(1) **Input**: Unsorted data Q
(2) **Output**: Sorted runs $r_0, \ldots, r_{R-1}$
(3) $S \leftarrow \{S_0, \ldots, S_{R-1} | S_i = \text{split}(Q)\}$
(4) **for** i from 0 to $R-1$ **do**
(5) $\quad R_i \leftarrow \text{SortInHostMemory}(S_i)$
(6) $p_i^j \leftarrow \{p_0^0, \ldots, p_i^j | p_i^j = \text{split}(R_i)\}$
(7) **end for**
(8) $W \leftarrow P/C$
(9) **for** i from 0 to $R-1$ **do**
(10) $\quad$ **for** k from 0 to $W-1$ **do**
(11) $\quad\quad$ Open Channels
(12) $\quad\quad$ write from p_i^j to p_{i+C}^j
(13) $\quad\quad$ InsertIndexToMinIndex (minimum (p_i^j), page.id)
(14) $\quad\quad$ SortMinIndex()
(15) $\quad$ **end for**
(16) **end for** = 0

Algorithm 1: Pseudo-code for the run generation phase.

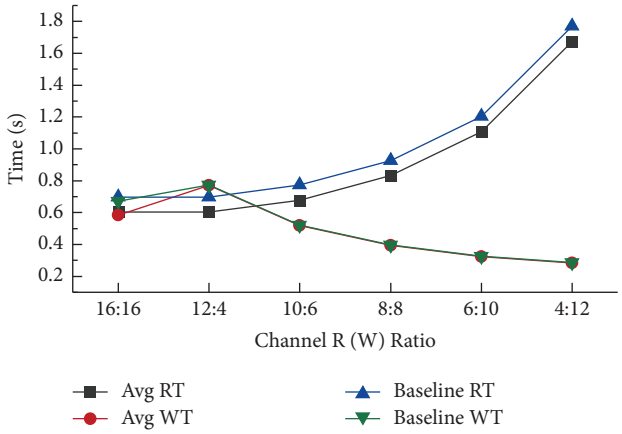

Figure 6: The average response time varying channel R/W ratio.

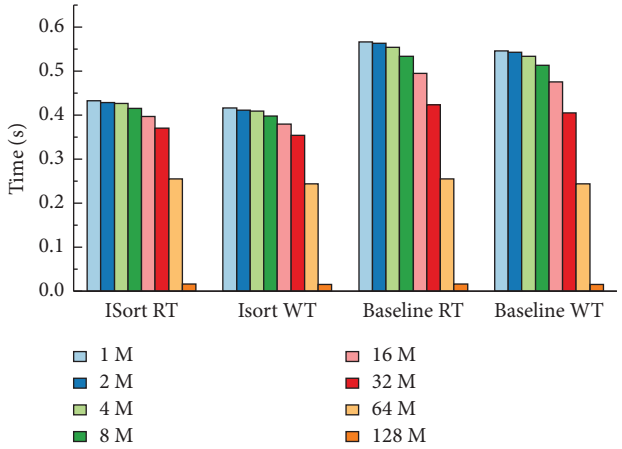

Figure 7: The average response time varying DRAM size within SSD.

As shown in Figure 7, DRAM within SSD can cache read and write requests. With the increase of DRAM capacity, the hit rate of read and write requests can be improved.

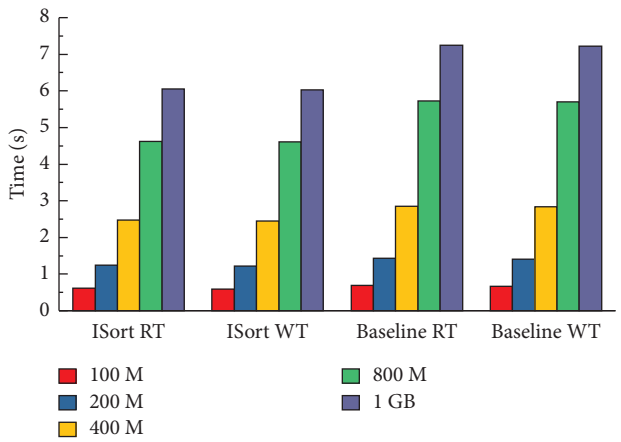

Figure 8: The average response time when varying datasets.

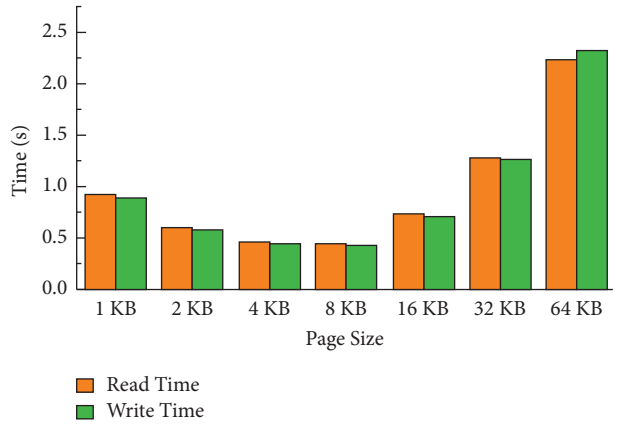

Figure 9: The average response time varying page size.

Figure 8 shows the results of different data sets. We can find that ISort has a relatively more stable performance improvement than ActiveSort.

```
(1) Input: Partial sorted runs r_0, ..., r_{R-1}
(2) Output: Sorted data
(3) Read CK_1 and CK_2
(4) while has not yet processed all pages do
(5)   if there are C pages in the memory then
(6)     Sort (CK_1)
(7)     Output minimum key into buffer
(8)     if the output buffer is full then
(9)       Flush the output buffer to flash chip
(10)    end if
(11)  end if
(12) end while = 0
```

Algorithm 2: Pseudo-code for run merge phase.

Table 2: Experimental set-up.

Parameter	Value
DRAM size	8 MB
Channel number	16
Chip number	32
Die number	2
Plane number	2
Block number	64
Page number	64
Subpage page	4
Page capacity	4096
Subpage capacity	1024

Figure 9 shows the results of different page sizes. When page size is 4 kB or 8 kB, ISort has better performance. However, the performance will degrade as the page size increases or decreases. When the page size is relatively large, it will cause channel congestion and increase the request processing time.

5. Related Work

When source data are too large, it is necessary to use external sorting when it is impossible to load all the data into limited memory for sorting at one time. External sorting can be divided into HDD-based external sorting, embedded flash memory-based external sorting, SSD-based external sorting, and NVM-based external sorting.

The external sorting based on HDD generally reduces the search time and rotation delay by optimizing the algorithm and reducing the random access to the external memory device. Reference [19] staggered placement and a new reading strategy are proposed, speeding up the execution of the external sorting algorithm based on HDD and improving the performance. Reference [20] proposed an external sorting algorithm based on HDD. This external sorting algorithm does not require additional disk space and does not generate intermediate data. The main idea is to use quick sorting and a particular merging strategy to reduce the number of comparisons in the sorting process to improve execution performance.

Compared with HDD, flash-based SSD has no disk head and a mechanical arm, so there is no seek time and rotation delay [21]. Reference [22] designed an FTL (FTL-SS) based on a single channel and single way and extended the FTL to the case of multichannel and multiways, thus verifying the versatility and effectiveness of this method. References [23–26] make full use of the internal parallelism of SSD through two phases and request rescheduling and dynamic write request mapping to improve the performance of SSD. Reference [27] proposes a channel striping technology to improve the resource utilization of the channel. FMsort makes full use of SSD's fast access delay and high random. The I/O bandwidth is able to speed up the execution of external merge sorting [28]. Montres [29] takes advantage of the performance of SSD to speed up external sorting processing. ActiveSort implements the merging operation of external sorting inside SSD by using Active SSD [10]. Active SSD is a special kind of SSD [30]. Kang et al. proposed a multichannel storage system based on NAND flash memory. The storage system has a plurality of independent channels, with each channel having a plurality of NAND flash memory chips [31].

With the development of new storage technology, new storage technology such as PCM, STT-RAM, and ReRAM have been widely used. PCM [32, 33] is a new nonvolatile storage medium with byte-addressable, high density, and high persistence. NVM is a nonvolatile storage device with byte-addressable, nonvolatile, random access, high density, low energy consumption, and high access speed [34]. However, NVM also has some limitations. The service life of NVM is limited, and the reading and writing performance is asymmetric [35–37]. Ahmed Khernache et al. proposed MONTRES-NVM, which is an external sorting algorithm based on the PCM and DRAM hybrid storage system [38].

6. Conclusion

The amount of data has increased exponentially in recent years, and our demand for data processing speed has gradually increased. The emergence of ActiveSSD provides a new possibility for us to process data in the near data segment. Traditional sorting algorithms need to be adjusted to better adapt to changes in memory size. This paper

analyzes the latest algorithms and concludes that the large numerical data generated in memory will remain in memory for a long time, affecting memory utilization. The main idea of ISort is to use the computing resources within SSD to deal with the merging phase. We use each page's minimum order to read the data to solve the problem of limited internal memory in SSD. To further improve the speed, we adopt the interleaving strategy in the write back part of the run generation phase. Many IO operations will produce varying degrees of read and write congestion; we have carried out different channel read-write ratio tests. We evaluated the performance of different read-write channel ratios, data size, page size, and SSD memory size. Compared with active sort + write, the performance of ISort reduces execution time by more than 36%. As a perspective for future work, it is significant to work to study the influence of different storage devices on various algorithms. For benefits from data access according to the characteristics of other storage devices and to further reduce the time overhead, this paper only discusses channel-level parallelism. In future in-depth research, we can continue to explore the deeper level of parallelism. At the same time, in future research, we will continue to study that data placement leads to increased garbage collection load. During the merge phase of the outer sort, writing the ordered data back to SSD can continue to explore where the output structure is written back when compared with the effect of opening up new space and allocating piecemeal space, which is better.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by National Key R&D Program of China under Grant no. 2021YFB0300103, National Natural Science Foundation of China (no. 61872392, U1911401), and the Major Program of Guangdong Basic and Applied Research (No. 2019B030302002).

References

[1] A. Thusoo, S. Zheng, S. Anthony et al., "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*, A. K. Elmagarmid and D. Agrawal, Eds., ACM, Indianapolis, IA, USA, pp. 1013–1020, June 2010.

[2] J. Boukhobza and P. Olivier, *Flash Memory Integration: Performance and Energy Issues*, ISTE Press - Elsevier, London, UK, 1st edition, 2017.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys*, vol. 38, no. 3, p. 10, 2006.

[5] W. Dobosiewicz, "Replacement selection in 3-level memories," *The Computer Journal*, vol. 27, no. 4, pp. 334–339, 1984.

[6] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A Case for Flash Memory Ssd in enterprise Database Applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*, pp. 1075–1086, Vancouver, BC, Canada, June 2008.

[7] A. M. Caulfield, J. Coburn, T. Mollov et al., "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, New Orleans, LA, USA, November 2010.

[8] S. M. Vikram, Z. Qureshi, W. Liang et al., "Deepstore: in-storage acceleration for intelligent queries," in *Proceedings of the The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, pp. 224–238, Columbus, OH, USA, August 2019.

[9] S. Jun, M. Liu, S. Lee et al., "Bluedbm: distributed flash storage for big data analytics," *ACM Transactions on Computer Systems*, vol. 34, no. 3, pp. 1–31, 2016.

[10] Y. Lee, L. C. Quero, S. Kim, J. Kim, and S. Maeng, "Activesort: efficient external sorting using active ssds in the mapreduce framework," *Future Generation Computer Systems*, vol. 65, pp. 76–89, 2016.

[11] Y. Liu, Z. He, Y. P. P. Chen, and T. Nguyen, "External sorting on flash memory via natural page run generation," *The Computer Journal*, vol. 54, no. 11, pp. 1882–1990, 2011.

[12] J. Seol, H. Shim, J. Kim, and S. Maeng, "A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2009*, J. Henkel and S. Parameswaran, Eds., ACM, Grenoble, France, pp. 137–146, October 2009.

[13] J. Lee, S. Kim, H. Kwon et al., "Block recycling schemes and their cost-based optimization in nand flash memory based storage system," in *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007*, pp. 174–182, Salzburg, Austria, September 2007.

[14] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," *ACM SIGARCH - Computer Architecture News*, vol. 37, no. 3, pp. 279–289, 2009.

[15] H. Kim and S. Lee, "A new flash memory management for flash storage system," in *Proceedings of the 23 Annual International 1999 COMPSAC'99*, pp. 284–289, Phoenix, AZ, USA, October 1999.

[16] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS - Performance Evaluation Review*, vol. 37, no. 1, pp. 181–192, 2009.

[17] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 229–240, 2009.

[18] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the 25th International Conference on Supercomputing, 2011*, pp. 96–107, Tucson, AZ, USA, May 2011.

[19] L. Zheng and P. A. Larson, "Speeding up external mergesort," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 2, pp. 322–332, 1996.

[20] R. Islam, N. Adnan, N. Islam, and S. Hossen, "A new external sorting algorithm with no additional disk space," *Information Processing Letters*, vol. 86, no. 5, pp. 229–233, 2003.

[21] C.-H. Wu and K.-Y. Huang, "Data sorting in flash memory," *ACM Transactions on Storage*, vol. 11, no. 2, pp. 1–25, 2015, 25.

[22] S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung, "Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 3, pp. 1392–1400, 2009.

[23] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based ssds," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 9–12, 2010.

[24] Y. Chen, J. Li, and H. Gao, "Fssort: external sort for solid state drives," in *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 71–80, Melbourne, Australia, May 2021.

[25] K. Myung, S. Kim, H. Y. Yeom, and J. Park, "Efficient and scalable external sort framework for nvme ssd," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2211–2217, 2020.

[26] Y. Chen, J. Li, and H. Gao, "Finding the optimal execution scheme of external mergesort on solid state drives," *World Wide Web*, vol. 24, no. 3, pp. 781–804, 2021.

[27] L. Chang and T. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 187–196, San Jose, CA, USA, September 2002.

[28] J. Lee, H. Roh, and S. Park, "External mergesort for flash-based solid state drives," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1518–1527, 2016.

[29] A. Laga, J. Boukhobza, F. Singhoff, and M. Koskas, "Montres: merge on-the-run external sorting algorithm for large data volumes on ssd based storage systems," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1689–1702, 2017.

[30] D. Tiwari, S. Boboila, S. Vazhkudai et al., "Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines," pp. 119–132, 2013.

[31] J.-Uk Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance nand flash-based storage system," *Journal of Systems Architecture*, vol. 53, no. 9, pp. 644–658, 2007.

[32] B. C. Lee, P. Zhou, J. Yang et al., "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, no. 1, p. 143, 2010.

[33] Y. Fu, "Caram: A Content-Aware Hybrid Pcm/dram Main Memory System Framework," 2020, https://arxiv.org/abs/2007.13661.

[34] H. Li and Y. Chen, "Emerging non-volatile memory technologies: from materials, to device, circuit, and architecture," in *Proceedings of the 2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, pp. 1–4, Seattle, WA, USA, August 2010.

[35] J. S. Meena, S. M. Sze, U. Chand, and T. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale Research Letters*, vol. 9, no. 1, p. 526, 2014.

[36] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: a study of caching and tiering approaches," *ACM Transactions on Storage*, vol. 10, no. 4, pp. 1–21, 2014.

[37] T. Roy and K. Kant, "Enhancing endurance of ssd based high-performance storage systems using emerging nvm technologies," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1070–1079, IEEE, New Orleans, LA, USA, May 2020.

[38] M. B. Ahmed Khernache, A. Laga, and J. Boukhobza, "MONTRES-NVM: an external sorting algorithm for hybrid memory," in *Proceedings of the IEEE 7th Non-Volatile Memory Systems and Applications Symposium, NVMSA 2018*, pp. 49–54, IEEE, Hakodate, Sapporo, Japan, August 2018.