External Sorting on Flash Memory Via Natural Page Run Generation

Yang Liu, Zhen He, Yi-Ping Phoebe Chen and Thi Nguyen

Department of Computer Science and Computer Engineering, La Trobe University, VIC 3086, Australia
Email: y34liu@students.latrobe.edu.au, z.he@latrobe.edu.au, Phoebe.Chen@latrobe.edu.au,
nt2nguyen@students.latrobe.edu.au

The increasing popularity of flash memory means more database systems will run on flash memory in the future. One of the most important database operations is the external sort. Hence, this paper is focused on studying the problem of efficient external sorting on flash memory. In contrast to most previous work, we target the situation where previously sorted data has become progressively un-sorted due to data updates. Accordingly, we call this "partially" sorted data. We focus on re-sorting partially sorted data by taking advantage of the partial sorted nature of the data to speed up the run generation phase of the traditional external merge sort. We do this by finding "naturally occurring" page runs in the partially sorted data. Our algorithm can perform up to a factor of 1024 less write IO compared to a traditional external merge sort during the run generation phase. We map the problem of finding naturally occurring runs into the shortest distance problem in a directed acyclic graph (DAG). Accordingly, we propose an optimal solution to the problem using the well known DAG-Shortest-Paths algorithm. However, we found the optimal solution was too slow for even moderate sized data sets and accordingly propose a fast heuristic solution which we experimentally show finds a high percentage of page runs using a minimum of computational overhead. Experiments using both real and synthetic data sets show our heuristic algorithm can halve the external sorting time when compared to three likely competing external sorting algorithms.

Keywords: external sorting; flash memory; relational database; merge sort

1. INTRODUCTION

Flash memory is becoming more popular due to its rapidly decreasing price and increasing performance. The compact form of flash memory is used for small mobile devices while larger ones called solid state drives (SSD) are used to replace hard disk drives. Recent advances in flash memory technology has meant SSDs with the capacity of hundreds of GB are becoming ever more affordable. The study by Lee et. al. [1] shows that the use of flash memory can greatly increase the performance of database systems. The performance increase comes from flash memory's ability to perform fast random reads. However, writes are comparably slow due to flash memory's inability to write in-place. Therefore, any effective algorithm that uses flash memory needs to take advantage of fast random reads while performing writes sparingly.

External sorting is a fundamental operation in relational database systems, with applications such as: sorted merge join; facilitating fast range query lookups; result sorting; duplicate removal; uniqueness verifications etc.

In this paper, we focus on re-sorting data that was initially sorted but becomes progressively unsorted due to updates. We call this "partially" sorted data. One situation which prefers sorted data, but will still work (albeit sub-optimally) when the data is partially sorted is answering range queries using an unclustered index. In this situation, the sought

range would span less disk pages when the tuples are closer to being fully sorted. Another situation is a mostly read only database being updated in batch. In this case re-sorting can occur right after a batched update.

We do not control when the re-sort is needed, but instead assume some mechanism exists to tell us when re-sorting is necessary. In this paper, we extend the traditional external merge sort algorithm to take advantage of partially sorted data and the unique characteristics of flash memory. Our approach uses the partially sorted nature of the data to reduce the number of writes at the expense of increased random reads. This is particularly suitable for use on flash memory since flash memory is fast at random reads but slow at writing.

The traditional external merge sort algorithm has two phases. In the first phase, blocks of data are loaded into RAM and sorted to create sorted runs. In the next phase, the sorted runs are repeatedly merged until a single sorted run containing all the data is created. Our key insight is that the sorted run generation phase produces a lot of writes which can be minimized by finding disk pages that form a "naturally" occurring page run. We define a naturally occurring page run as a sequence of pages whose record value intervals do not overlap each other. Therefore all values in the second page of the naturally occurring page run are larger than all values in the first page and so on.
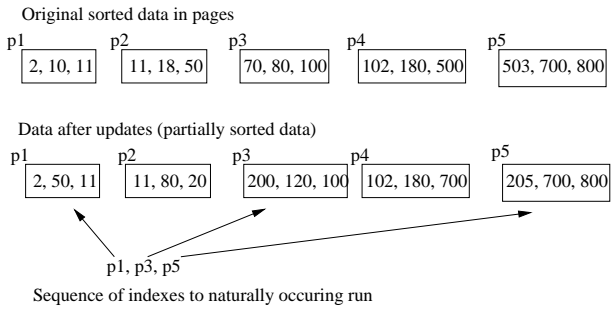
Original sorted data in pages

p1	p2	p3	p4	p5
2, 10, 11	11, 18, 50	70, 80, 100	102, 180, 500	503, 700, 800

Data after updates (partially sorted data)

p1	p2	p3	p4	p5
2, 50, 11	11, 80, 20	200, 120, 100	102, 180, 700	205, 700, 800

p1, p3, p5

Sequence of indexes to naturally occuring run

FIGURE 1. Example of a naturally occurring page run of 3 pages in partially sorted data.

Values within each page do not need to be internally sorted. We can then store indexes to these pages instead of creating a normal sorted run. Assuming an index entry occupies 4 bytes and a page occupies 4096 bytes, this results in a 1024-fold saving in write costs for sorted run generation. Figure 1 shows an example of a naturally occurring page run in partially sorted data. Note there is no overlap between the minimum and maximum value ranges of the pages of the naturally occurring page run.

From the above description, it can be seen that our technique aims to reduce the write IO costs during sorted run generation. In Section 4, we explain why it is a particularly good idea to optimize write IO costs in an external merge sort on flash memory.

The following two factors aid us in finding a high percentage of naturally occurring page runs: 1) the input data is partially sorted; and 2) we generate the smallest possible run size which does not increase the number of merge passes. Partially sorted data typically means the values in each page span a smaller range because the data is still mostly sorted. Smaller run sizes are easier to find since less pages with non-overlapping value ranges are required.

It should be noted that re-sorting partially sorted data has been studied extensively in the existing literature in the form of adaptive sorting [2, 3]. However, almost all existing work in this area is focused on internal sorting (where all data fit in RAM). The adaptive internal sorting algorithms are not designed to minimize the number of accesses to secondary storage by using a RAM buffer and therefore are not suitable for re-sorting data residing in secondary storage (the focus of this paper).

Estivill-Castro et al. [2], in their widely cited survey paper on adaptive sorting, identify the replacement selection sort [4, 5] as the only existing adaptive external sorting algorithm. A replacement selection sort can produce longer sorted runs when the data is partially sorted. However, unless the output of the run generation is a single sorted run, each data page still needs to be written out at least twice, once during run generation and once during the merge. In contrast, our approach of finding naturally occurring page runs can avoid writing a high percentage of the data pages during run generation. Experimental results show our approach can outperform replacement selection sort by up to a factor of 1.87.

We map the problem of finding naturally occurring page runs into the problem of finding the shortest path in a directed acyclic graph (DAG). Instead of finding the shortest path, we find the path whose length exactly equals the size of the page run. We adapt the well known DAG-Shortest-Paths algorithm [6] to solve our problem. Although the algorithm is guaranteed to find a page run of the required length if it exists, it is unfortunately too slow with a run time complexity of $O(NA + \frac{N}{T}(M + E))$, where N is the total number of pages to be sorted, T is the size of the sorted runs, M is the number of pages that can fit in RAM, A is the number of records per page and E is the number of edges in the DAG. In the worst case, E can equal $(M^2 + M)/2$. Accordingly, we propose a heuristic solution which comes close to the DAG-Shortest-Paths algorithm in terms of finding a high percentage of naturally occurring page runs but incurs much lower overhead. The run time complexity of our heuristic solution is $O(N(A + log_2 M))$. Experimental results show our heuristic algorithm can halve the sorting time compared to three likely competitors. In addition, our heuristic algorithm finds a high percentage of naturally occurring runs even when there is a significant percentage of random updates to previously sorted data.

This paper makes the following key contributions:

- We propose a new approach to speeding up sorted run generation for external merge sort. The idea is to cleverly exploit the characteristics of partially sorted data and the characteristics of flash memory by finding naturally occurring page runs.
- We show the problem of finding naturally occurring page runs can be mapped into the shortest path algorithm for a DAG.
- We propose two solutions to the problem. First is an optimal solution based on the well-known DAG-Shortest-Paths algorithm and the second is a fast heuristic solution which finds a high percentage of naturally occurring page runs at much lower overheads.
- Extensive experiments using both real and synthetic data sets against three likely competing algorithms demonstrates the superiority of our approach.

The remainder of the paper is organized as follows: Section 2 describes the characteristics of flash memory; Section 3 describes the traditional external merge sort algorithm; Section 4 describes why it is a good idea to minimize write IO costs during the run generation phase; Section 5 describes our approach of minimizing write IO costs during run generation using naturally occurring runs; Section 6 describes the experimental setup used to test our solution; Section 7 shows the experimental results and analysis for our empirical study on the effectiveness of our solutions; Section 8 describes the related work and finally in Section 9 we conclude the paper and outline directions for future work.

2. FLASH MEMORY CHARACTERISTICS

Our work is focused on the popular NAND flash memory. NAND flash memory has the following characteristics:

- **Asymmetric read versus write cost** Unlike the hard disk drive (HDD), the random write cost of flash memory is typically much higher than random read costs. This is due to flash memory's need to erase an entire block of data (a block consists of multiple pages) before writing to it. Therefore, if no pre-erased block exists, an expensive garbage collection process is started which cleans dirty blocks. Writing is done at the page grain (typically 4 KB).

- **Fast random read** Unlike HDD, flash memory does not have any rotational latency associated with data access. Therefore, it can perform random reads very fast. This combined with the high random write costs means replacing random writes with random reads is a good strategy for improving the performance of algorithms which use flash memory. The algorithm designed in this paper has this property.

- **Limited number of block erasures** Each block in flash memory has a limited lifespan in terms of the number of write/erase cycles before it becomes unstable and unusable. Newer flash drives have increasingly higher life spans, enduring up to 5 million write/erase cycles. Nevertheless, each erase operation slowly wears out the block. To maintain the functionality of the blocks for as long as possible, page updates should be spread out evenly across all available blocks. This is commonly called wear-leveling and is normally taken care of by a special type of firmware called the flash translation layer (FTL).

FTL is a software layer between the file system and flash memory hardware. FTL provides the file system with an interface to the flash memory that is identical to that of the common HDD. As a result, common file systems such as FAT, ATA or SATA can be used with flash memory. FTL performs updates out-of-place, garbage collection, wear leveling and error detection transparent to the file system. This is done by mapping virtual addresses to physical addresses. Although FTL addresses the need for wear leveling on flash memory, it does not address the issue of the skewed read/write speed ratio.

3. TRADITIONAL EXTERNAL MERGE SORT

Almost all external sorting algorithms are some variant of the traditional external merge sort. Therefore, we first provide a detailed description of the tradition external merge sort.

The overall idea behind the traditional external merge sort algorithm is to use the limited RAM buffer to sort small portions of the input data at a time and store them as "sorted runs". Then, the sorted runs are repeatedly merged into larger sorted runs until a single sorted run is produced. Accordingly the algorithm is divided into two phases, the run generation phase and the merge phase.
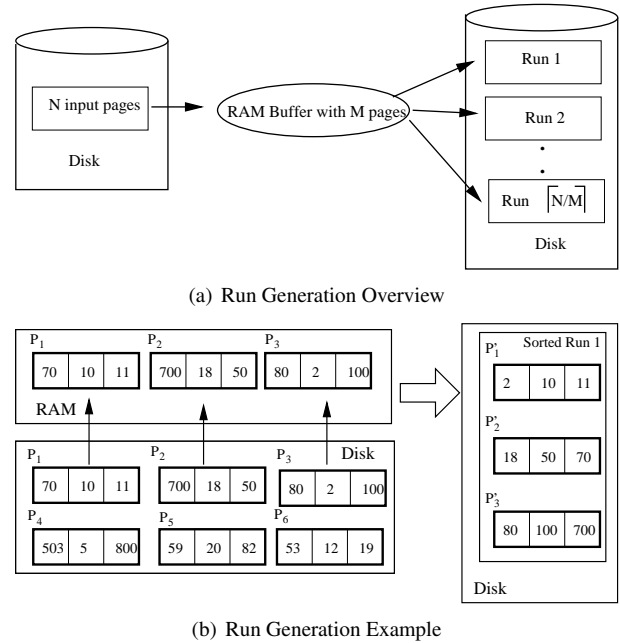

(a) Run Generation Overview

(b) Run Generation Example

FIGURE 2. Diagrams describing the run generation of Traditional External Merge Sort

Assume a RAM buffer of M pages and an input data of N pages. The run generation phase loads up M input data pages into the RAM buffer, sorts them and writes them back to disk as sorted runs. This is repeated until $\lceil N/M \rceil$ sorted runs are generated. Figure 2(a) shows this diagrammatically. Figure 2(b) shows an example which uses a three-page buffer to generate a three-page sorted run.

Figure 3(a) shows a diagram describing the overview of the merge phase. In the merge phase, K sorted runs are merged at a time, where K is a user-determined parameter. Each of the K sorted runs are allocated an input buffer which stores a cluster (unit of IO) of the sorted run's data. A single output buffer is allocated to store the current portion of the merged sorted run. Figure 3(b) shows an example of how sorted runs are merged. In the example, the data need to be sorted in ascending order. The first cluster (in this case just one page) of each sorted run is loaded into their respective input buffers. Next, the output buffer is filled with the three smallest values of all the input buffers (crossed out values). Next, the output buffer is flushed to disk and emptied. The same process is used repeatedly to fill the output buffer. Whenever an input buffer becomes empty, the next cluster of the corresponding sorted run is loaded into the input buffer. This entire procedure is repeated until the K sorted runs are merged into a single sorted run. Then, each of the merged sorted runs are combined repeatedly until there is only one resulting sorted run for the entire data set.

Section 8 reviews recent research into improving the traditional external merge sort algorithm and other approaches to achieving fast external merge sort.
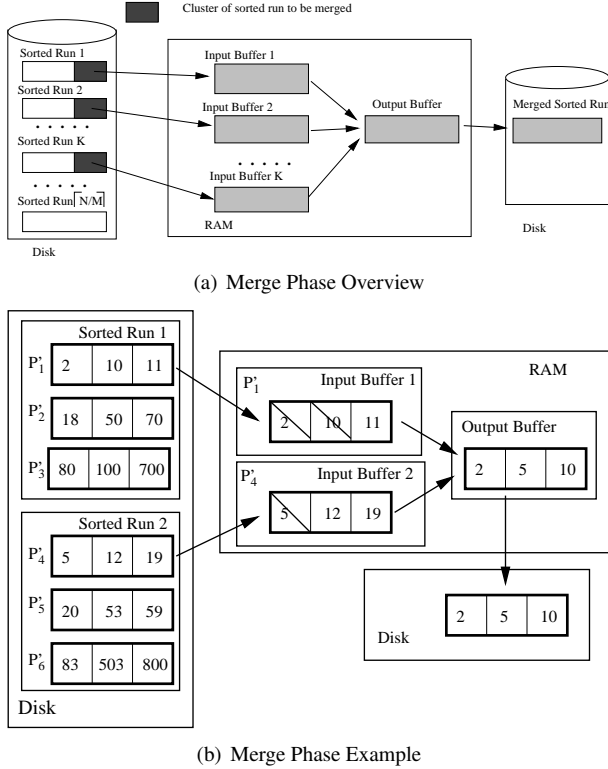
(a) Merge Phase Overview

(b) Merge Phase Example

FIGURE 3. Diagrams describing the merge phase of the Traditional External Merge Sort

4. WHY OPTIMIZE WRITE IO COSTS OF RUN GENERATION?

In this section, we describe the rationale for focusing on optimizing the write IO costs of run generation instead of the other IO costs associated with an external merge sort. The merge cluster size is an important determinant of the performance of the external merge sort. According to the analysis in Appendix A.2, the best cluster size for flash memory is one page. As shown in Table A.1 of Appendix A.2, this translates to a one pass merge in almost all practical situations. Therefore, in the rest of this section, we assume the external merge sort uses a single merge pass. The total IO cost for one pass merge (TIO_{OP}) for performing the external merge sort is given by the equation below:

$$
\begin{aligned}
TIO_{OP}(N) &= RGRC + RGWC + MRC + MWC \quad (1) \\
&= N(FM_{RC} + FM_{WC}) + N(FM_{RC} + FM_{WC})
\end{aligned}
$$

where $RGRC$, $RGWC$, MRG, MWC refer to the following costs respectively: sorted run generation read IO cost; sorted run generation write IO cost; merge phase read cost; and merge phase write cost. The remaining terms have the same definition as those defined for Equation A.2.

Equation 1 states that for a one pass merge, we only need to load every page once and write them out once for both the sorted run generation and merge phases. We now analyze the possibility of avoiding some of each of the following

costs in Equation 1: $RGRC$; $RGWC$; MRC; and MWC. Avoiding some $RGRC$ would mean we can create the sorted runs without ever loading some of the pages into RAM. This seems impossible unless we already have some kind of index on the data. However, we assume no such index exists. Although some existing techniques try to avoid some $RGWC$ and MRC by keeping the last generated sorted run in RAM and use it directly during merging, this approach does not result in much savings if the data set size N is much larger than the memory size M. Finally, it seems impossible to avoid MWC since the aim is to generate a sorted sequence onto flash memory. In contrast to the previous cost analysis, it is possible to avoid close to all $RGWC$ by finding naturally occurring sorted runs using our technique described in Section 5 and only write indexes to them rather than the actual runs themselves. In addition, avoiding write IO is much more profitable than read IO for flash memory due to the asymmetrical read versus write costs of flash memory.

5. SORTED RUN GENERATION BASED ON NATURALLY OCCURRING PAGE RUNS

Section 4 explained the rationale behind reducing write IO costs during the run generation phase of an external merge sort instead of other IO costs. In this section, we present our approach for achieving this, using naturally occurring runs. The idea is to create naturally occurring page runs by finding pages whose value ranges do not overlap each other. We call these naturally occurring page runs. An example is shown in Figure 1. In the example, the naturally occurring page run consists of 3 pages, $p1$, $p3$ and $p5$. Notice the value interval created from their minimum and maximum values do not overlap. We then densely pack these indexes into pages and write them out instead of naturally occurring runs themselves. Assuming an index entry occupies 4 bytes and a page occupies 4 KB, this translates to a 1024 factor saving in write costs during run generation.

The remainder of this section is presented as follows. First, we formally define the problem of finding naturally occurring runs in Section 5.1. Second, we show the problem of finding naturally occurring page runs can be mapped to finding the shortest path problem in a directed acyclic graph in Section 5.2. Third, as it is not clear we should look for large page runs or small ones, since large ones will result in less merge passes but are harder to find, in Section 5.3 we describe the analysis used to arrive at the optimal size of naturally occurring page runs. Finally in Section 5.4, we describe a fast heuristic algorithm for finding naturally occurring runs.

5.1. Problem of finding naturally occurring runs

In this section, we first formally define the problem of finding naturally occurring page runs and then show how it can be mapped into the well known problem of finding the shortest path in a directed acyclic graph.

The problem of finding naturally occurring page runs is defined as follows:
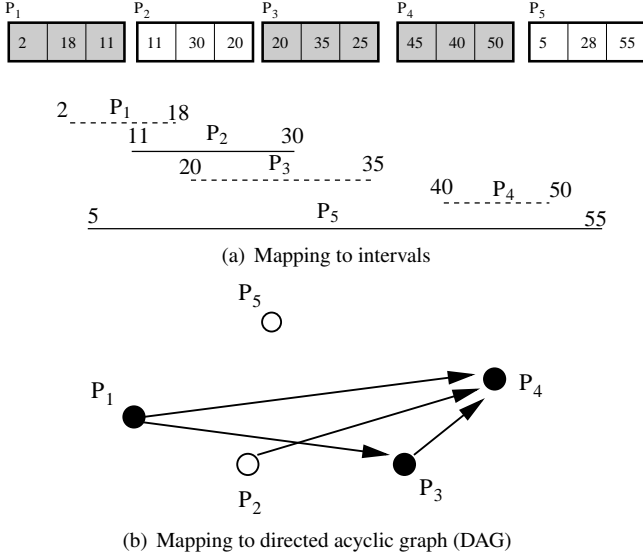
(a) Mapping to intervals

(b) Mapping to directed acyclic graph (DAG)

FIGURE 4. Example showing intervals being mapped into a DAG

DEFINITION 5.1 (Find naturally occurring runs problem). *Given a set of pages S, where each page $s \in S$ has an associated value interval $< s_{min}, s_{max} >$, find a naturally occurring page run which consists of a set of T pages R such that $R \subseteq S$ and no pair of pages $r_i \in R$, $r_j \in R$, $i \neq j$ have overlapping value intervals.*

In the above problem definition, T is an input parameter to the problem. Figure 4(a) shows an example in which we are looking for $T = 3$ non-overlapping pages among the 5 pages available to create a naturally occurring page run. The example also shows the value intervals corresponding to the pages. In the example, the pages P_1, P_3 and P_4 form one possible naturally occurring page run since none of their value intervals overlap.

5.2. Shortest path based solution to finding naturally occurring runs

We can map the value intervals of Definition 5.1 to a directed acyclic graph (DAG) as follows. Each vertex represents a page P_i and there is a directed edge from vertex P_i to P_j if and only if P_i and P_j are non-overlapping and P_i's range is less than P_j's range. There can not be any cycles in this graph because of the transitivity of less than. Then finding a naturally occurring run of size T is equivalent to finding a path of length T. This is a simple variant of the longest path problem in a DAG, where instead of finding the longest path, we stop as soon as we find a path of length T. The longest path problem is NP for a general graph, however, for a DAG, it can be computed in polynomial time by mapping it into the shortest path problem where the edge weights are negated.

We use the well known DAG-Shortest-Paths algorithm [6] as the basis for our solution. Since we only have a limited amount of RAM, we can only search among the pages that can fit into the memory. Algorithm 1 details the algorithm we use. The algorithm works by first filling the RAM buffer

with unprocessed pages (Line 2) from the flash memory. We load the pages in a particular order to maximize the chances of finding page runs (please see separation distance loading in Section 5.4 for more details). Next, we find the minimum and maximum values from the loaded pages to construct the value intervals. We then insert these value intervals into the DAG. In Line 5, we use a simple variant of the shortest path algorithm to find a path of length T. Instead of finding the shortest path, we stop the search as soon as we have found a path of length T. If no path is found, we create a normal sorted run by sorting T pages with the longest intervals (Line 7) and writing them into the flash memory. Removing the longest intervals is desirable since these intervals generally are less connected with other pages because of their tendency to overlap with other pages (e.g. page P_5 in Figure 4). If a naturally occurring page run is found we do not write the pages out but instead write index entries to the locations of the pages into an index buffer page.

Algorithm 1 FindNaturalRuns(Input P:Sequence of pages to be sorted, T: optimal naturally occurring run size; M: RAM buffer size; N input data size)

1: **while** more pages in P need to be processed **do**
2:    **fill** RAM Buffer with next set of unprocessed pages from P using separation distance loading.
3:    **extract** value intervals from loaded pages
4:    **insert** newly extracted intervals into DAG
5:    find the first naturally occurring page run of size T in DAG using DAG-Shortest-Paths algorithm [6]
6:    **if** no naturally occurring page run of size T can be found in DAG **then**
7:       **sort** T pages with longest intervals into one run and write it to flash memory as a normal sorted run
8:    **else**
9:       Write index entries for pages of naturally occurring page run into index page
10:       Flush index page if it becomes full
11:    **end if**
12:    Remove processed pages from RAM buffer
13: **end while**
14: If index page is not empty, flush it to flash memory

THEOREM 5.1 (Run time complexity of Algorithm 1). *The computational complexity of Algorithm 1 is $O(NA + \frac{N}{T}(M + E))$, where N is the total number of data pages to be sorted, T is the size of the page runs, M is the size of the RAM buffer in pages, A is the number of records per page and E is the number of edges in the DAG.*

Proof. The minimum and maximum values for each of the N pages containing A records per page need to be found to create the value intervals (Line 3), hence the cost is NA. We run the code between Lines 2 and 12 $\frac{N}{T}$ times because during each iteration of the while loop we process T of the N pages either into a naturally occurring page run or normal sorted run. The shortest path algorithm used in Line 5 incurs the dominant run time cost and it has a run time complexity

of $O(M + E)$ [6]. Therefore, the run time complexity of Algorithm 1 is $O(NA + \frac{N}{T}(M + E))$. $\square$

The run time complexity of Algorithm 1 can be very high since in the worst case E equals $(M^2 + M)/2$. Our experiments show the execution time of Algorithm 1 increases very fast with increasing data size, making it unusable even for moderate sized data. Accordingly, we developed a fast heuristic algorithm to solve the problem of finding naturally occurring page runs without the need to create a graph. This algorithm is presented in Section 5.4.

5.3.   Optimal naturally occurring run size

In this section, we will define the optimal size of naturally occurring page runs. This is non-trivial since larger naturally occurring page runs are hard to find, but smaller ones may cause more than one merge pass. The reason larger naturally occurring runs are hard to find is that they require more non-overlapping pages. Smaller naturally occurring page runs may increase the number of merge passes because it might exceed the maximum number of runs that can be merged in RAM in one pass. Therefore, we define the optimal naturally occurring run size as follows:

DEFINITION 5.2 (optimal naturally occurring page run size). *The optimal naturally occurring page run size is defined as the minimum run size subject to the constraint the number of merge passes equals the minimum achievable for any run size.*

We now define the equation we use to compute the optimal page run size given the following: a data size of N pages; a RAM size of M pages; one-page used to accumulate index entries during run generation; a one page output buffer during the merge; and an index buffer of E pages used during the merge. We need a small index buffer of E pages during the merge to prevent repeated disk loads for naturally occurring page run index lookups. Since the index pages are densely packed, we found in our experiments we only need a very small index buffer of 20 pages to prevent almost any repeated index loads. The optimal naturally occurring page run size (ONRS) can be computed by the equation below:

$$ONRS(N, M, E) = \left\lceil \frac{N}{(M - E - 1)^{\lceil log_{M-E-1} \lceil \frac{N}{M-1} \rceil \rceil}} \right\rceil$$
(2)

THEOREM 5.2. *Equation 2 satisfies the optimal naturally occurring page run size Definition 5.2.*

Proof. For Equation 2 to be correct, its denominator must specify the maximum number of runs subject to the constraint of Definition 5.2. This is because $\left\lceil \frac{N}{number of runs} \right\rceil$ equals the run size. So, we turn our attention to proving the denominator of 2 equals the maximum number of runs subject to the constraint in Definition 5.2. The $\left\lceil log_{M-E-1} \lceil \frac{N}{M-1} \rceil \right\rceil$ expression, which we will call $MinPasses$, specifies the minimum number of merge passes. This is because the $\left\lceil \frac{N}{M-1} \right\rceil$ expression

inside the $MinPasses$ expression specifies the maximum run size possible, given a RAM of M pages with one reserved for accumulating indexes to naturally occurring runs. The minimum number of merge passes occurs when the maximum number of runs $(M - E - 1)$ are merged in one pass. This is because of the M pages available RAM, E pages are reserved for the index buffer and another one page is reserved for the output buffer. Hence $MinPasses$ is the minimum number of merge passes for any given run size. $(M - E - 1)^{MinPasses}$ is the maximum number of runs that can be merged in $MinPasses$. $\square$

Having specified a way of computing the optimal naturally occurring page run size and proven it is correct, we describe our algorithm for finding naturally occurring runs of the optimal size. In the remainder of this paper, whenever we mention naturally occurring page runs, we mean page runs of optimal size as specified in this section.
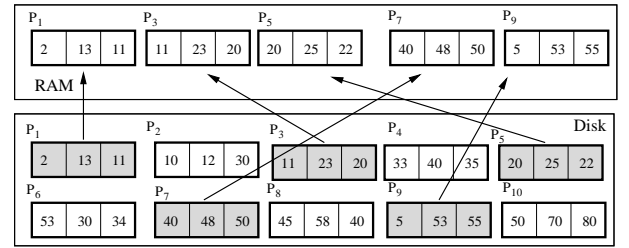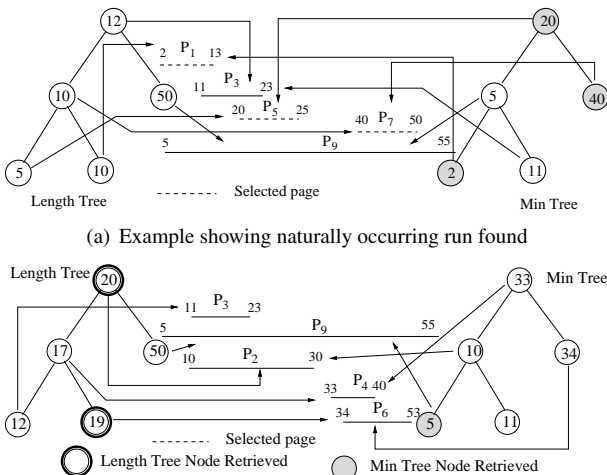
5.4.   Fast heuristic algorithm

FIGURE 5. Example illustrating loading pages with one page separating them.

Our heuristic approach to finding naturally occurring page runs of optimal size starts by loading as many pages as possible into the RAM buffer and then looking for non-overlapping pages among them. To maximize our chances of finding naturally occurring page runs, we use the following three techniques:

- **Separation distance loading of pages.** Loading pages which are far apart from each other into the RAM buffer gives us a high chance of finding naturally occurring page runs. This is because the input data is partially sorted and therefore pages further apart are less likely to overlap. Accordingly, we load pages $\lfloor \frac{N}{M-1} \rfloor - 1$ pages apart from each other, where N is the data size and M is the RAM buffer size in pages. The $M - 1$ term comes from the fact one page is reserved for accumulating index entries to naturally occurring runs. This allows the loaded pages to have the maximum distance apart from each other. When loading pages, we wrap back to the beginning of the input page sequence once we reach the end. Figure 5 shows an example where $N = 10$, $M = 5$ and therefore pages should be loaded with one page separating them.

- **Find non-overlapping intervals using a minimum interval tree (Min Tree).** We index pages loaded into RAM using their corresponding minimum value inside

the Min Tree. The Min Tree can be implemented by using any main memory binary search tree such as the red and black tree. Note the tree must allow for duplicate keys. We use the Min tree to scan for non-overlapping intervals from left to right. Therefore, we start with the interval with the smallest minimum value L and then use L.maximum to find the next interval U which has the property U.minimum is the closest to L.maximum. We perform all these searches using the Min Tree. Figure 6(a) shows an example of using the Min Tree to find a naturally occurring page run of size three pages. The first page found is P_1 which has the smallest minimum value. Then, the next two pages found are P_3 and P_7 in that order.

- **Remove long intervals using an interval length tree (Length Tree).** When it is determined that we can not find enough non-overlapping pages to form a naturally occurring page run, we create a "normal" sorted run. A normal sorted run is created by sorting the values inside its set of pages and writing into the disk sorted like the traditional external merge sort. When creating a normal sorted run, we try to pick the remaining pages that have the longest interval length. This is because the intervals with the longest interval length are more likely to overlap with other pages. The Length tree can also be implemented by any binary search tree which allows duplicate keys. Figure 6(b) shows an example where the page P_9 was the first picked, using the Min Tree since it is the one with the smallest minimum value. However, after that, no other RAM buffer resident pages can be picked since they all have minimum values larger than the maximum value of P_9. Therefore, the Length Tree is used to pick two pages (P_2 and P_6) with the longest remaining interval length to be sorted to create a normal sorted run.

(a) Example showing naturally occurring run found

(b) Example showing no naturally occurring page run found, so need to sort data in pages

FIGURE 6. Example showing our heuristic algorithm finding naturally occurring page runs.

Our experiments show the above techniques used together

are very effective in finding a high percentage of naturally occurring page runs in partially sorted data. Algorithm 2 shows the pseudo code for finding naturally occurring page runs. Lines 2 to 5 populate the RAM buffer with pages loaded from the disk and inserts them into the Min Tree and Length Tree. Lines 6 to 17 look for naturally occurring runs among the pages in the RAM buffer. If no naturally occurring run can be found, then Lines 8 to 12 create a normal sorted run using the pages picked so far and the pages with the longest remaining value intervals. Finally, if a naturally occurring page run is found, Lines 19 to 21 are used to write the index entries for the pages of the run.

Algorithm 2 FindNaturalRuns(Input P:Sequence of pages to be sorted, T: optimal naturally occurring page run size; M: RAM buffer size; N input data size)

1: **while** more pages in P need to be processed **do**
2:   **Fill** RAM Buffer with next set of unprocessed pages from P using separation distance loading.
3:   **extract** value intervals from loaded pages.
4:   **insert** extracted value intervals into Min Tree and Length Tree.
5:   **initialize** current run list CL to empty.
6:   **while** $|CL| < I$ and RAM buffer is not empty **do**
7:     **if** no interval in Min Tree has minimum value greater than maximum value of last element in CL **then**
8:       // no naturally occurring run found
9:       **insert** T - $|CL|$ of the longest intervals in Length Tree into CL
10:      **remove** inserted intervals from Min Tree and Length Tree
11:      **sort** pages in CL in RAM and **write** into flash memory.
12:      **remove** all pages in CL from the RAM buffer
13:    **else**
14:      **insert** interval in Min Tree with smallest minimum value into end of CL
15:      **remove** inserted interval from Min Tree and Length Tree
16:    **end if**
17:   **end while**
18:   **if** naturally occurring run was found **then**
19:     Write index entries for pages of naturally occurring run found from CL into index page
20:     Flush index page if it becomes full
21:     Remove pages in CL from RAM buffer
22:   **end if**
23: **end while**
24: If index page is not empty flush it to flash memory

The benefits of finding naturally occurring runs would be outweighed by its computational costs if Algorithm 2 is too slow. Therefore, we use the theorem below to show the computation costs of Algorithm 2 is modest.

THEOREM 5.3 (Run time complexity of Algorithm 2). *The computational complexity of Algorithm 2 is* $O(N(A +$

$log_2M)$), *where N is the total number of data pages to be sorted, M is the size of the RAM buffer in pages and A is the number of records per page.*

Proof. The minimum and maximum values for each of the N pages (containing A records per page) need to be found to create the value intervals (Line 3), hence the cost NA. Each of the N pages needs to be inserted (Line 4) and removed (Lines 10 and 15) from the Min Tree and Length Tree. Each insertion and deletion takes $O(log_2M)$ since each of these trees indexes M intervals and they are both binary search trees. The Min Tree is searched in Line 7 and the Length tree is searched in Line 9 but both of these searches are only done at most once per page. Therefore, all operations on the Min Tree and Length Tree costs $O(Nlog_2M)$.   □

5.5. Merge Phase

The merge phase of our approach is the same as the external merge sort with two exceptions. First, pages belonging to naturally occurring runs are first internally sorted when loaded into RAM. Second, we reserve a small portion of the RAM buffer to buffer index pages to naturally occurring page runs. We use the least recently used buffer replacement policy for the index page buffer. The size of the index page buffer can be very small, just 20 pages in our experiments. This is because each index page can contain indexes to 1024 pages of naturally occurring runs.

A small optimization that can improve the performance of the index page buffer is to rearrange index page entries so that the same i^{th} page of page runs are grouped into the same page. For example, an index page would contain only the 2nd pages of page runs. This improves the temporal and spatial locality of references to index pages during the merge. The reason is the merge phase tends to progress through the pages of the page runs close to parallel. For example, when the 2nd page of one page run is being processed, it is common to be processing the 2nd page of the other page runs at the same time.

6. EXPERIMENTAL SETUP

The experiments were conducted using a 256GB Super Talent Ultra Drive GX FTM56GX25H SSD. As mentioned in the introduction, SSDs are high capacity flash devices, designed to replace hard disk drives. The read and write characteristics of the flash drive compared to common HDD are shown in Table 1. From the table, we can see the random read performance of the SSD is around 2.5 times faster than random write. However, the HDD random read and write performances are much more similar. The random read performance of the SSD is much higher than the HDD.

The CPU we used had the following specifications: Intel Core 2 Duo E8500; 3.2 GHz; and 6 MB L2 cache. The machine had 3 GB of total available RAM. However, we further restricted the amount of RAM available to the tested algorithms. The experiments were conducted on the linux operating system. Linux automatically caches all IO requests. This would invalidate our experiment results since

	Real data default value	Synthetic data default value
RAM size (pages)	2, 000	20, 000
Data size (pages)	19, 018	150, 000
Record size (bytes)	200	200
$UpdatePercentage$	20	20
$MaxUpdateRange$	20	20
Page size (bytes)	4096	4096

TABLE 2.   Default parameter settings for both the real and synthetic data sets.

it would mean pages loaded during run generation will be available for reuse in the merge phase without the need to reload from the SSD. Therefore, we **disabled the operating system's caching functionality**.

We have used both real and synthetically generated data sets. Our real data set is the United States Census 2000 data set which can be downloaded from [7]. The census data consisted of more than 200 columns. We chose, by default, column 7 which contains the population of each county of the USA, including urban and rural. However, we tested the first 100 columns and found the performance difference between our algorithm and the others stayed about the same across all the different columns. Section 7.4 shows the results from columns 6 to 20. The synthetic data set was generated by generating random integers between 0 and 1, 000, 000 with uniform random distribution.

For both real and synthetic data, we did the following to convert them into partially sorted data. The data was first sorted. Next, updates were introduced by randomly picking a certain percentage of values to be updated (we call this $UpdatePercentage$). For each picked value x, a random new value was assigned in the range $x - x\frac{\delta}{100}$ and $x + x\frac{\delta}{100}$ using uniform random distribution, where δ is a parameter we call $MaxUpdateRange$. Note $MaxUpdateRange$ is a percentage value.

Table 2 shows the default parameter settings used for the real and synthetic data sets.

We compared the performance of our approach against three rival external sorting algorithms. They are described below, along with any parameter settings we have used for them.

- **NaturalDAGMSort** This is shortest path on a DAG algorithm described in Section 5.2, with the index buffer used in the merge phase (Section 5.5) set to 20 pages and run size determined by Equation 2.
- **NaturalMSort** This is our fast heuristic algorithm described in Section 5.4, with the index buffer used in the merge phase set to 20 pages and run size determined by Equation 2.
- **TradMSort** This is the traditional external merge sort algorithm described in Section 3, with run size set to the total size of RAM.
- **ReplaceMSort** This is the replacement selection merge sort algorithm [8] described in Section 8.1, with run size set to the total size of RAM. This algorithm is selected because it is known to perform well for partially sorted data, the reason being that this

	Sequential Read (MB/s)	Sequential Write (MB/s)	Random Read (IO/s)	Random Writes (IO/s)
Super Talent UltraDrive GX FTM56GX25H 256 GB SSD	230	180	4630	1866
Seagate Barracuda 7200 RPM ST3500418AS 500GB SATA HDD	73	64	159	111

TABLE 1. Read and write characteristics for the SSD and HDD used in our experiments.

algorithm can take advantage of partially sorted data to create longer sorted runs.

- **FAST** This is the external sorting algorithm designed by Park et. al. [9] especially for use with flash memory. The algorithm was described in Section 8.3.

For all of the above algorithms, we used a merge cluster size of 1 page following the analysis in Section Appendix A.2.

7. EXPERIMENTAL RESULTS

We conducted six experiments. The first four experiments used the real data set and the last two used the synthetic data set. In the first experiment, we varied the data size of the real data. In the second experiment, we varied RAM size. In the third experiment, we varied the amount of update. In the fourth experiment, we reported the results from sorting different columns of the US Census 2000 data. In the fifth experiment, we tested the scalability of the different algorithms by varying the size of the synthetic data set. Finally, in the sixth experiment, we varied the record size of the synthetic data set.

We only report the results for NaturalDAGMSort for experiment 1 since NaturalDAGMSort runs too slowly for all other experiments due to the larger data size.

7.1. Real data: vary data size results

In this experiment, we compare the performance of the different algorithms when the data size is varied from 1000 pages to 5000 pages for the real US Census 2000 data. We set the RAM size to 500 pages. We left the other parameters to the default settings specified in Table 2.

Figure 7 shows the results for this experiment. The results show our heuristic NaturalMSort algorithm is the best performer for total execution time (Figure 7 (b)). This can be explained by the similarity between the shape of total execution time and the number of write IO curves (Figure 7 (b)). This suggests that the write IO costs dominate all other costs and that NaturalMSort produces the second least number of write IO. Although NaturalDAGMSort produces a lower number of write IO than NaturalMSort, it is not practical to use it when the data size is above 3000 pages (it takes more than 2 hours for 4000 pages) because of its high run time complexity.

Although the NaturalMSort algorithm executes much faster than NaturalDAGMSort, it still manages to find a high percentage of naturally occurring page runs as shown in Table 3. The table shows NaturalDAGMSort is able to find a higher percentage of naturally occurring page runs than NaturalMSort, but this comes at a very high CPU cost.

7.2. Real data: vary RAM size results

In this experiment, we compare the performance of the different algorithms when the RAM size is varied from 200 pages (around 1% of total data size) to 5000 pages (around 26% of total data size) for the real US Census 2000 data. We left the other parameters to the default settings specified in Table 2. We do not include the results for NaturalDAGMSort in this or any subsequent experiments because it too slow for any data size larger than 3000 pages, as shown in Section 7.1.

Figure 8(a) shows the total execution time result when the RAM size is varied. Table 4 shows the ratio of total execution time of the competing external sort algorithm against the total execution time of NaturalMSort. Therefore, a ratio of 2 means NaturalMSort outperforms the competing algorithm by a factor of 2. The results show that when the RAM size is 800 pages (4% of total data size) or larger, NaturalMSort significantly outperforms the other algorithms. This can be explained by looking at Table 5, which shows that when the RAM size gets to 800 pages, 72% of the runs used by NaturalMSort are naturally occurring. Furthermore, the table also shows at 800 pages, runs need to be 25 pages long. This shows that even in this difficult situation, our heuristic algorithm can find naturally occurring page runs a large percentage of the time.

Figure 8(a) also shows NaturalMSort does not perform as well as the other algorithms when RAM size is very small (below 800 pages or 4% of total data size). This is because when the RAM size is that small, the size of the naturally occurring page runs is large (see Table 5), which makes it hard to find many non-overlapping pages. At this point, the small benefit from finding the small percentage of naturally occurring page runs is outweighed by the high cost of looking for them.

The results in Figures 8(b) and 8(c) show that NaturalMSort outperforms TradMSort and ReplaceMSort at larger RAM sizes because of its ability to reduce write IO costs rather than read IO costs. This matches our aim explained in Section 4. The reduced write IO costs is due to the high percentage of naturally occurring runs found, as shown in Table 5.

The reason the FAST algorithm is generally the worst performer is that it tries to create longer sorted runs by taking too many read passes through the data. This can be seen from Figure 8(b).

7.3. Real data: vary update

In this section, we answer the question of how robust NaturalMSort is to varying amounts of updates in the partially sorted data. We answer this question by injecting
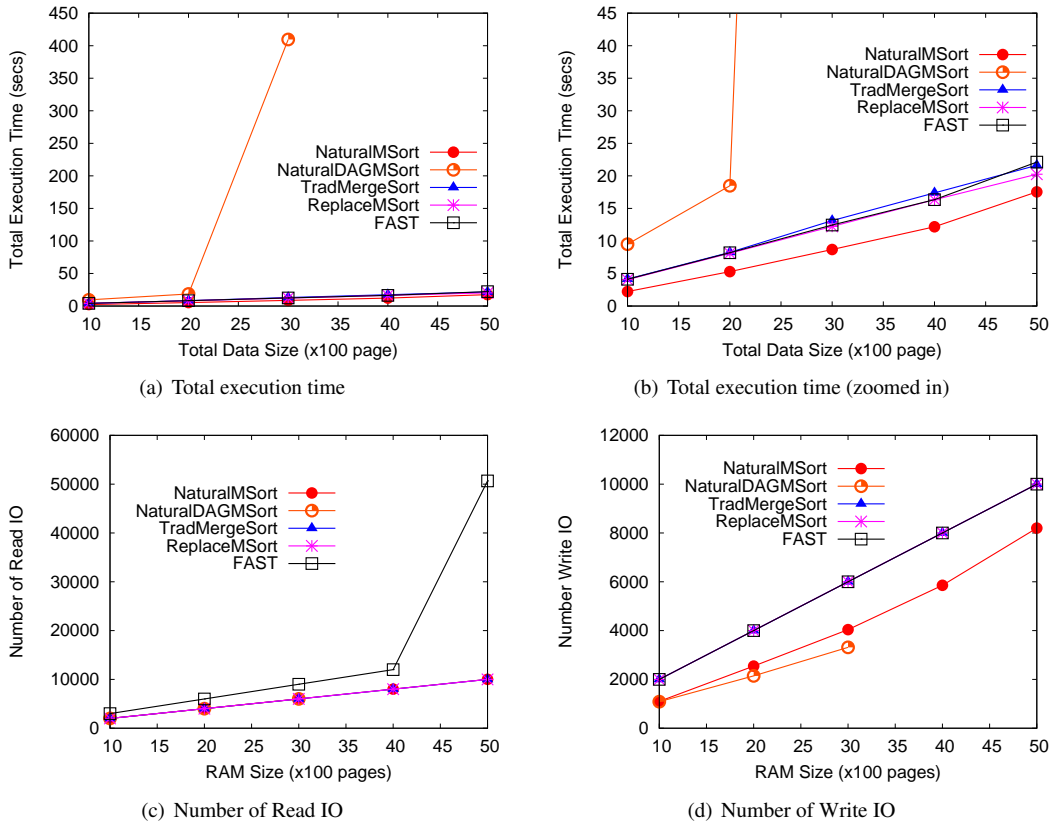
(a) Total execution time

(b) Total execution time (zoomed in)

(c) Number of Read IO

(d) Number of Write IO

FIGURE 7. Vary total data size for the real data

Data Size (pages)	1000	2000	3000	4000	5000
Percentage of runs being naturally occurring(heuristic design)	90.4	73.00	65.60	53.80	36.20
Percentage of runs being naturally occurring(DAG based design)	91.60	93.00	89.80		
Size of naturally occurring page runs (pages)	3	4	6	8	10

TABLE 3. Vary data size results for the real data

(a) Total execution time

(b) Number of Read IO

(c) Number of Write IO
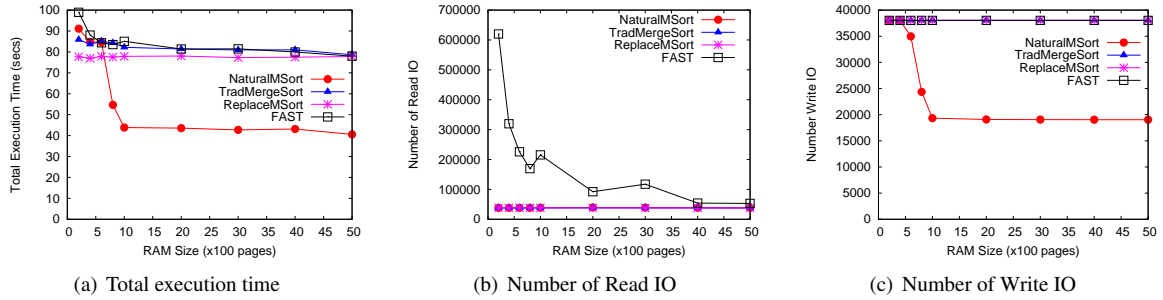
FIGURE 8. Vary RAM size for the real data

RAM Size (x 100 pages)	2	4	6	8	10	20	30	40	50
TradMSort/NaturalMSort	0.94	0.98	1.0	1.54	1.87	1.86	1.89	1.87	1.93
ReplaceMSort/NaturalMSort	0.85	0.91	0.92	1.42	1.78	1.79	1.81	1.80	1.92
FAST/NaturalMSort	1.09	1.04	1.0	1.53	1.94	1.87	1.91	1.85	1.92

TABLE 4. Total execution time of $\frac{\text{competing algorithms}}{\text{NaturalMSort}}$ for varying RAM size for the real data

RAM Size (x 100 pages)	2	4	6	8	10	20	30	40	50
Percentage of runs being naturally occurring	0	0.27	16.31	72.04	98.43	99.93	99.98	99.99	99.99
Size of naturally occurring page runs (pages)	107	51	33	25	20	10	7	5	4

TABLE 5. Vary RAM size results for the real data.

MaxUpdateRange	UpdatePercentage										
		5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
5%	2.18	2.31	2.31	2.29	2.27	2.28	2.25	2.26	1.43	2.27	1.86
10%	2.20	2.24	2.32	2.12	2.26	1.67	2.17	1.32	2.24	2.10	2.17
20%	2.28	2.29	2.26	1.72	2.08	1.61	1.29	2.15	2.11	2.15	2.09
30%	2.29	2.28	2.15	2.18	2.15	1.36	2.04	2.08	2.12	2.06	2.09
40%	2.28	2.32	2.11	2.18	1.75	1.44	1.59	1.56	1.65	1.51	1.45
50%	2.30	2.26	2.16	1.95	1.40	1.22	1.19	1.14	1.11	1.13	1.19
60%	2.34	2.21	2.11	2.07	1.53	1.28	1.01	0.99	0.98	0.97	0.97
70%	2.28	2.58	2.17	2.08	1.54	1.14	1.04	0.98	0.98	0.96	0.43
80%	2.18	2.22	2.15	2.06	1.51	1.15	1.00	1.05	1.05	0.97	0.96
90%	2.22	2.22	2.15	2.13	1.58	1.15	1.01	1.05	0.96	1.04	0.96
100%	2.23	2.19	2.14	2.09	1.37	1.14	1.03	0.98	0.97	0.96	0.96

TABLE 6. Vary update results of $\frac{\text{TradMSort total execution time}}{\text{NaturalMSort total execution time}}$ for the real data.

MaxUpdateRange	UpdatePercentage										
		5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
5%	1.84	1.87	1.86	1.84	1.85	1.86	1.84	1.83	1.18	1.87	1.52
10%	1.88	1.86	1.87	1.82	1.85	1.38	1.78	1.08	1.84	1.73	1.82
20%	1.87	1.86	1.82	1.38	1.75	1.32	1.05	1.79	1.81	1.82	1.79
30%	1.85	1.80	1.77	1.75	1.15	1.86	1.77	1.79	1.82	1.78	1.84
40%	1.85	1.86	1.80	1.80	1.47	1.26	1.38	1.39	1.38	1.36	1.31
50%	1.83	1.85	1.79	1.73	1.25	1.09	1.06	1.03	1.02	1.02	1.01
60%	1.86	1.83	1.76	1.36	1.07	0.91	0.90	0.88	0.88	0.88	0.87
70%	1.86	1.82	1.80	1.76	1.34	1.04	0.91	0.89	0.88	0.88	0.88
80%	1.83	1.81	1.78	1.35	1.03	0.92	0.89	0.89	0.88	0.88	1.82
90%	1.87	1.83	1.80	1.77	1.32	1.04	0.91	0.88	0.87	0.88	0.87
100%	1.86	1.83	1.80	1.75	1.24	1.03	0.93	0.88	0.89	0.87	0.87

TABLE 7. Vary update results of $\frac{\text{ReplaceMSort total execution time}}{\text{NaturalMSort total execution time}}$ for the real data.

MaxUpdateRange	UpdatePercentage										
		5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
5%	1.85	1.89	2.03	2.06	1.96	2.06	1.92	1.87	1.26	1.96	1.57
10%	1.90	1.88	1.89	2.02	1.93	1.42	2.00	1.11	1.88	1.79	1.85
20%	2.05	1.95	2.03	1.51	1.82	1.36	1.16	1.86	1.88	1.89	1.92
30%	1.88	1.95	2.01	2.01	1.96	1.19	1.83	1.88	1.90	1.87	1.89
40%	1.90	2.00	2.00	2.01	1.51	1.30	1.43	1.45	1.42	1.39	1.38
50%	2.07	1.97	1.91	1.41	1.87	1.12	1.09	1.06	1.05	1.05	1.05
60%	1.93	1.98	1.96	1.94	1.38	1.10	0.94	0.91	0.91	0.91	0.91
70%	1.93	2.00	1.97	1.97	1.50	1.07	0.94	0.98	0.98	0.91	0.98
80%	1.89	1.96	1.99	2.00	1.50	1.06	0.93	0.91	0.91	0.99	0.99
90%	1.92	2.01	2.00	1.99	1.47	1.17	0.93	0.91	0.90	0.90	0.97
100%	1.90	2.00	1.97	1.96	1.39	1.17	0.96	0.90	0.99	0.98	0.97

TABLE 8. Vary update results of $\frac{\text{FAST total execution time}}{\text{NaturalMSort total execution time}}$ for the real data.

varying amount of updates to the sorted real data by varying the values of $UpdatePercentage$ (as described in Section 6) and $MaxUpdateRange$ from 5% to 100%. We left the other parameters to the default settings specified in Table 2.

Table 6, 7, and 8 show the performance of NaturalMSort against TradMSort, ReplaceMSort and FAST, respectively. The tables show the ratio of total execution time of the competing external sort algorithm against the total execution time of NaturalMSort. Therefore, a value of two means NaturalMSort outperforms the competing algorithm by a factor of 2.

The results show that NaturalMSort outperforms the other external sorting algorithms for a wide range of update amounts. When an update amount is low, NaturalMSort can outperform TradMSort by more than a factor of 2. This can be explained by two facts. First, NaturalMSort can potentially save half of the write IO costs and it also uses less CPU time. Second, as we saw in the previous experiments, write IO costs are the dominant costs on total execution time. Due to the design of NaturalMSort, it can virtually avoid any write IO during run generation and thereby effectively halve the write IO costs. In addition, we found TradMSort can spend a factor of 13.7 more CPU time for run generation compared to NaturalMSort. This is because TradMSort needs to sort large runs in RAM which takes a lot more time. In contrast, NaturalMSort does not need to sort naturally occurring runs at all during run generation. Although NaturalMSort incurs slightly more CPU overhead during the merge phase (internal sorting of pages belonging to naturally occurring page runs) than NaturalMSort, this overhead is much lower than the CPU overhead savings made during run generation.

As expected, when the amount of updates increases, NaturalMSort loses its performance advantage against the competing algorithms. However, it is encouraging to see that NaturalMSort can tolerate around 50% update in both $UpdatePercentage$ and $MaxUpdateRange$ before it starts to be slightly worse than the competing algorithms. These results show NaturalMSort can tolerate a large amount of updates to partially sorted data when finding naturally occurring page runs.

7.4. Real data: vary column number results

In previous experiments, we used the default real data column 7 of the US census data. In this section, we report the results for columns 6 to 20 of the US Census 2000 data. This allows us to show that the use of column 7 in the previous experiments was representative of the different columns of the US census data set. We actually tested the first 100 columns of the real data but due to space constraints, we only show the results for columns 6 to 20. The performance difference between the algorithms did not vary much across all 100 columns tested. The columns below 6 were not tested since they contained categorical data instead of numerical data.

The results for this experiment are reported in Table 9. The results show the performance of NaturalMSort did not vary much across the different columns tested. We analyzed the data distribution of the different columns and found that they all conform to a very skewed Zipfian distribution. Although this means we essentially tested our algorithm on the same data distribution, nevertheless it shows our results are robust across the different columns of the US Census 2000 data. The synthetic data used in subsequent sections use uniform distribution. This means we have tested our algorithms across both a very skewed Zipfian distributed data set and an uniform distributed data set.

7.5. Synthetic data: vary data size

In this section, we compare the performance of the algorithms when the total data size is varied for the synthetic data set. The remaining parameters are set to the default

Column Number	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
NaturalMSort	42.78	43.04	43	42.93	43	44.48	43.65	49.57	49.89	42.66	42.87	54.2	43.74	50.92	48.33
TradlMSort	83.58	80.66	81.03	80.91	84.06	83.21	83.49	86.27	86.28	83.45	84.2	85.47	86.87	84.06	85.65
ReplaceMSort	77.13	78.03	77.96	77.58	77.59	77.9	78.03	77.63	77.44	77.15	77.68	77.49	77.27	77.39	81.4
FAST	79.32	81.40	80.53	80.94	79.57	80.36	79.74	79.42	79.31	79.9	79.69	79.38	79.93	79.35	79.77

TABLE 9. Total execution time (secs) results for varying column number of US Census 2000 Data

(a) Total execution time SSD

(b) Number of Read IO

(c) Number of Write IO
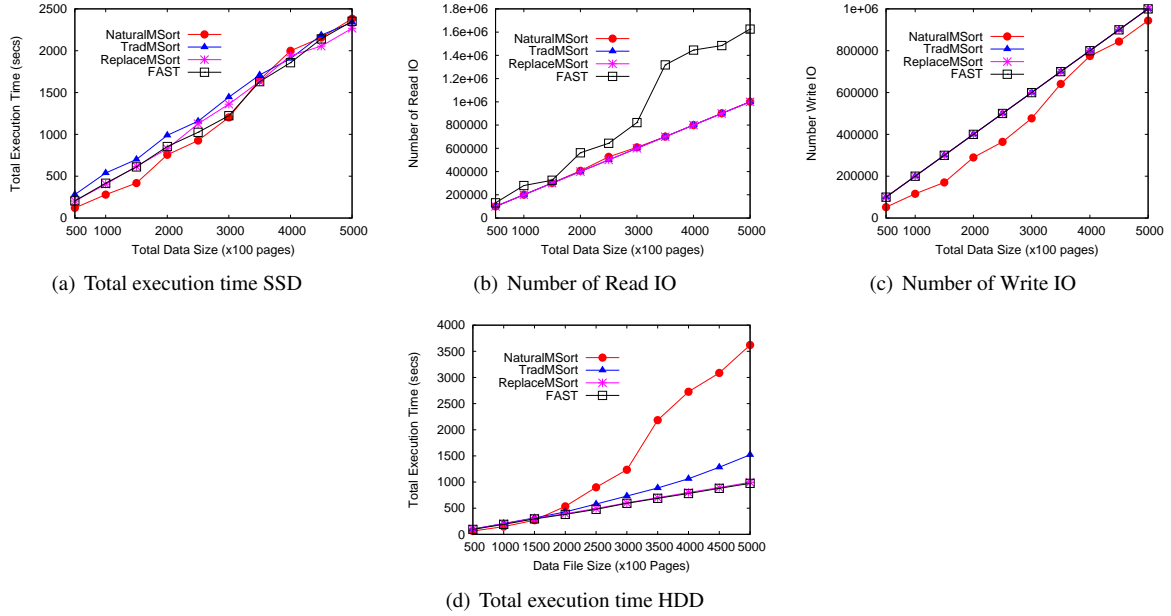

(d) Total execution time HDD

FIGURE 9. Vary synthetic data size

values described in Section 2. We show both SSD and HDD total execution time results because we want to verify our claim that NaturalMSort is customized for the unique characteristics of the SSD and therefore not suitable for the HDD.

We will first discuss the results for the SSD. Figure 9(a) shows the total execution time results for the different algorithms when using the SSD. Table 10 presents the same results in terms of the ratio of total execution time of each of the sorting algorithms against NaturalMSort. Therefore, a value of two means NaturalMSort outperforms the competing algorithm by a factor of 2. The results show NaturalMSort significantly outperforms the competing algorithms when the total data size is smaller than 300, 000 pages. This is because NaturalMSort finds a high percentage of naturally occurring page runs at this lower total data size range, as can be seen in Table 11

NaturalMSort loses its advantage over the competing algorithms when the total data size increases. This is because, in this experiment, the RAM size does not change and therefore, as the data size increases, the RAM to data size ratio drops rapidly. When the total data size increases beyond 300, 000 pages, the corresponding RAM size / data size ratio drops below 0.0067. Lower RAM size / data size ratio results in increases in the size of naturally occurring runs which make them harder to find (see Table 11).

The results show ReplaceMSort and FAST consistently outperform TradMSort. This is because both ReplaceMSort

and FAST create longer sorted runs than TradMSort. Longer run size means less runs are merged together which, in turn, reduces CPU usage.

Figure 9(d) shows the total execution time results for the HDD. As expected, the results show that the NaturalMSort is consistently the worst performer. This is due to the extra seeks incurred by NaturalMSort, outweighing the benefits gained from a reduced number of write IO. NaturalMSort incurs extra seeks in two places: when loading pages far apart from each other during run generation; and loading the scattered pages of naturally occurring page runs during the merge phase. These extra seeks do not incur much overhead when NaturalMSort is operating on the SSD but on the HDD, it dominates the execution time.

7.6. Synthetic data: vary record size

The record size is an important parameter for determining the success of our approach of finding naturally occurring run, the reason being that more records in one page makes it harder to find naturally occurring runs, since it increases the chances of having more extreme maximum and minimum values in each page. Accordingly, in this section, we compare the performance of the algorithms when the record size is varied for the synthetic data set. The remaining parameters are set to the default values described in Section 2.

Figure 10(a) and Table 12 show NaturalMSort consis-

Total Data Size (x100 pages)	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
TradMSort / NaturalMSort	2.32	1.92	1.68	1.30	1.24	1.20	1.03	0.95	1.01	0.98
ReplaceMSort / NaturalMSort	1.67	1.44	1.52	1.09	1.21	1.13	0.98	0.97	0.95	0.95
FAST / NaturalMSort	1.69	1.48	1.56	1.13	1.10	1.01	0.98	0.92	0.99	0.98

TABLE 10. Total execution time of $\frac{\text{competing algorithms}}{\text{NaturalMSort}}$ for varying synthetic total data size when using the SSD.

Data file size(x100 pages)	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Percentage of runs being naturally occurring	96.11	84.22	86.56	55.23	54.32	41.14	16.89	6.35	12.46	11.21
Size of naturally occurring page runs (pages)	3	6	8	11	13	16	18	21	23	26

TABLE 11. Vary record size results for the synthetic data set.

tently outperforms the other algorithms for the entire range of record sizes tested. Furthermore, it shows NaturalMSort can outperform competing algorithms by more than 30%, even when the record size is just 50 bytes (translates to 81 records in one page). This is hard to achieve since finding naturally occurring page runs becomes difficult when so many records in the same page can be updated. Table 13 shows even when records are only 50 bytes, NaturalMSort can find 63.58% of naturally occurring page runs. These results show the robustness of NaturalMSort to changing record sizes.

8. RELATED WORK

Most research in the area of external sorting have focused on speeding up the run generation or merge phases of the external merge sort, hence we start our discussion on these two sub-areas of research.

8.1. Speeding up the run generation phase

Speeding up sorted run generation has been studied by the research community [10, 3, 11]. There are basically two general ways for creating sorted runs in an external merge sort. The first is load-sort-store and the other is replacement selection [8]. Load-sort-store is the approach described in Section 3 which repeats the following: it loads as many pages as possible into the RAM buffer and sorts them into a sorted run and stores them back into the disk. Replacement selection typically creates runs twice as long as load-sort-store by only including records in the current run which are higher than the highest key output so far, and the non-qualifying records are used for the next run. This approach is particularly suitable for situations where the data is almost sorted, since in this case, very long sorted runs can be created. In addition, replacement selection allows computation and IO to be overlapped during run generation. Larson and Graefe [10] proposed efficient memory management algorithms for run generation which support variable length records. A replacement selection algorithm is proposed that handles variable length records. The proposed algorithms offer performance which gracefully adapts to varying RAM buffer sizes. Larson [3] proposed a batched version of replacement selection which is also CPU cache conscious and can handle variable sized records. All of the above work either attempts to speed up CPU

processing in run generation or create longer runs. Either way, they do not reduce the amount of write IO. In contrast, our work is focused on reducing the amount of write IO during run generation by finding naturally occurring page runs.

8.2. Speeding up the merge phase

Most proposed improvements on the traditional external merge sort algorithm are focused on speeding up the merge phase [12, 13, 14]. Double buffering and forecasting (see [8]) are well-known techniques for overlapping computation and disk IO during the merge phase. The idea in double buffering is to assign two buffers to each run during the merge phase. One buffer stores the current cluster of the run being processed and the other is used as a prefetch buffer for the next cluster of the same run. This allows the processing of the current cluster to be overlapped with the prefetching of the next cluster. Forecasting assigns one buffer to each run and one extra prefetch buffer to the particular run which is forecasted to be the next one that needs to be loaded. Salzberg [15] analyzed the effectiveness of double buffering under different conditions for situations where memory size is large. Zhang and Larson [13] proposed three buffering and read ahead strategies that improve on traditional double buffering and forecasting.

Zheng and Larson [12] proposed using extra buffer space during merging to order the retrieval of data blocks from the HDD in order to reduce total I/O time. Estivill-Castro and Wood [14] extended this approach by grouping adjacent run blocks together to reduce disk seeks. In contrast to the above techniques, we are focused on improving the run generation phase of the external merge sort instead of the merge phase.

8.3. External sorting using flash memory

Andreou et. al. [5] proposed a variant of the replacement selection external merge sort algorithm for sorting in Flash Memory. Park et. al. [9] proposed the Flash-Aware external Sorting algorithm (FAST) which replaces writes with multiple read passes. Their approach can be conceptualized as extending replacement selection sort to create longer sorted runs. Instead of creating the longest run you can using a one pass replacement selection sort, they take multiple read passes to create longer sorted runs. Our work contrasts from theirs in that we can significantly reduce
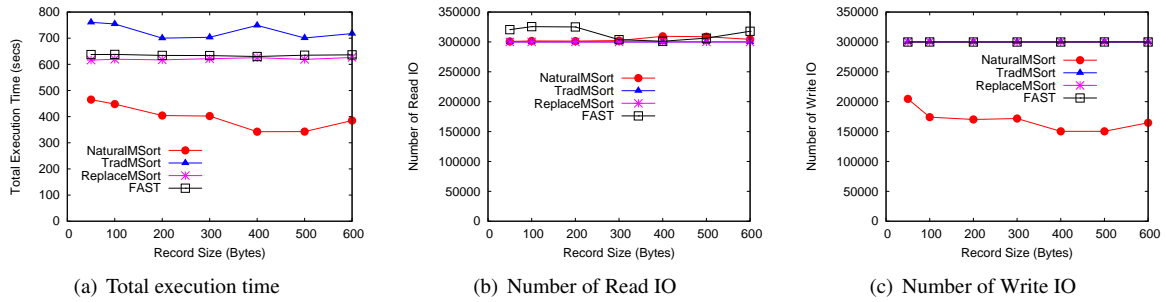
(a) Total execution time   (b) Number of Read IO   (c) Number of Write IO

FIGURE 10. Vary record size for the synthetic data

Record Size (Bytes)	50	100	200	300	400	500	600
TradMSort / NaturalMSort	1.6354	1.6836	1.7323	1.7489	2.1876	2.0450	1.8644
ReplaceMSort / NaturalMSort	1.3251	1.3829	1.5270	1.5452	1.8267	1.8056	1.6268
FAST / NaturalMSort	1.3704	1.4239	1.5679	1.5751	1.8397	1.8529	1.6530

TABLE 12. Total execution time of $\frac{\text{competing algorithms}}{\text{NaturalMSort}}$ for varying synthetic record size

the number of write IO without needing to significantly increase the total number of read IO. We only introduce a small amount of read IO during the merge phase to read indexes to naturally occurring page runs.

8.4. Virtual sorted runs

We now describe the existing work that is closest to ours. Haerder [16] first proposed the idea of combining multiple non-overlapping runs into one longer run during the merge phase of external sorting. The non-overlapping runs can be virtually concatenated by declaring they belong to one run instead of physically joining them in the disk. Graefe [17] called these "virtual" runs and stated that this approach is particularly suitable for almost sorted input. Graefe goes a step further to suggest virtual runs can be created by concatenating "partial" virtual runs. For example, if a large part of the one sorted run does not overlap with a large portion of another sorted run, then these can be joined to form one longer virtual sorted run. However, no experimental work was done to verify the effectiveness of the approach.

Although this work appears similar to ours, there are a number of key differences. First, we create naturally occurring page runs by combining non-overlapping *pages* of tuples to speed up the *run generation*. In contrast, their work is focused on virtually concatenating non-overlapping *sorted runs* to speed up the *merge phase*. Section Appendix A.2 explains our analysis which concludes optimal external merge sort for flash memory typically has only *one* merge pass. However, their approach is only useful for multi-pass merge. In addition, our problem is more challenging since it has a much larger search space. This is because we search for non-overlapping *pages* instead of *sorted runs*. Second, we develop a method to determine the optimal size for naturally occurring runs based on the characteristics of flash memory (see Section 5.3), whereas, no such work has been done in virtual run generation. Third, we show the problem

of finding naturally occurring page runs can be mapped to the shortest path algorithm for a DAG. No such work exists for finding virtual runs. Fourth, we propose a heuristic solution to the problem of finding naturally occurring runs which we empirically show is computationally efficient and can find a high percentage of naturally occurring runs. No existing empirical or theoretical work has been conducted on the effectiveness of using virtual runs to speed up external merge sort.

8.5. Other work on external sorting

Nyberg et. al. [11] propose the AlphaSort algorithm which performs CPU cache sensitive run generation. In addition, they use multiple processors to break the sort into subsort chores which are then merged by a root process. Some researchers focus on the problem of adjusting external merge sort to adapt to the changing amount of memory available [18, 19]. In contrast, we assume the memory available for sorting is pre-determined and fixed. Yiannis et. al. [20] propose a compression-based external sorting technique. Govindaraju et. al. [21] propose the use of the graphics processing unit to sort large databases. Some researchers work on the problem of performing external sorting in-place [22, 23], namely no temporary disk space is required during sorting. In contrast, we assume there is adequate temporary space available on the flash memory to perform out of place external sorting. We think this is a reasonable assumption, given the fast pace at which SSDs are increasing in size and decreasing in cost.

9. CONCLUSION

In this paper, we proposed a novel approach for speeding up the sorted run generation phase of an external merge sort. The approach is designed to sort partially sorted data. The idea is to replace writing sorted runs to disk with writing indexes to naturally occurring page runs which can potentially save up to a factor of 1024 on write IO costs. The

Record Size(Bytes)	50	100	200	300	400	500	600
Percentage of runs being naturally occurring	63.58	72.03	86.56	85.55	99.86	99.86	90.26
Size of naturally occurring page runs (pages)	8	8	8	8	8	8	8

TABLE 13. Vary record size results for the synthetic data set.

approach is particularly suitable for the characteristics of flash memory, where write costs are higher than read costs.

In this paper, we proposed a formula for determining the optimal size of naturally occurring page runs. We also map the problem of finding naturally occurring runs into the shortest path problem on a DAG. Accordingly, we use a shortest path algorithm to solve the problem. However, the algorithm was found to have high run time complexity and experimentally found to be impractical to use for even moderately sized data. Hence, we proposed a fast heuristic solution which executes much faster but still finds a high percentage of naturally occurring page runs.

A detailed experimental study was conducted into the effectiveness of our proposed algorithms against three likely competitors, using both real and synthetic data sets. The results show our fast heuristic algorithm can find a high percentage of naturally occurring runs, even when there was a high percentage of updates. Further, finding naturally occurring runs had a significant impact on reducing total execution time. The results show our approach prefers a larger RAM to total data size ratio, since in these conditions, the required size of the naturally occurring runs are smaller. However, a RAM to total data size ratio of 0.067 is all that is needed for our approach to outperform its competitors. Our approach can halve the sorting time compared to its competitors under favorable situations but is only slightly worse under unfavorable situations.

In the future, we plan to propose even faster and more effective heuristics for finding naturally occurring page runs. We will investigate how our approach can be integrated into the sorted merge join operation.

ACKNOWLEDGMENT

APPENDIX A. IMPORTANCE OF CLUSTER SIZE FOR TRADITIONAL EXTERNAL MERGE SORT

In this section, we explain why determining the best cluster size (IO unit) for an external merge sort is particularly important for performance. We highlight the difference between the best cluster size for HDD and flash memory.

Appendix A.1. The best merge cluster size for HDD

As mentioned in Section 3, a traditional external merge sort loads sorted runs in clusters into the RAM buffer during the
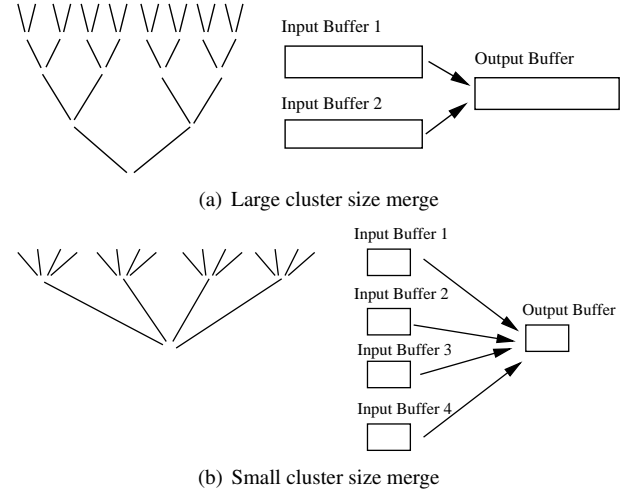

(a) Large cluster size merge

(b) Small cluster size merge

FIGURE A.1. Example showing merging at different cluster sizes

merging of sorted runs. Figure 1(a) shows that when the cluster size is large, only a very few runs can be merged in each step, resulting in more merge passes. During each merge pass, all data pages need to be loaded from the disk and written back to the disk in a sorted order. Therefore, less merge passes result in less IO. However, using smaller cluster sizes as shown in Figure 1(b) results in more random seeks (which is very costly on the HDD) because the same amount of data is loaded into RAM in smaller clusters. The positive effect of smaller clusters is less merge passes. Therefore, the best cluster size is non-trivial to determine for the HDD. This has been demonstrated experimentally in the paper by Lee et. al. [1]. The IO cost of the merge phase ($MCost$) can be described using the equation below (assuming the output buffer size is one cluster):

$$MCostHDD(N,M,C) = \lceil log_{\lfloor M/C \rfloor - C}(\lceil N/M \rceil) \rceil \times \quad \text{(A.1)}$$
$$(HDD_{seek}\lceil N/C \rceil + N(HDD_{RC} + HDD_{WC}))$$

where N, M and C are the data size in the pages, the RAM buffer size in the pages and the cluster size in the pages, respectively. HDD_{seek} is HDD seek cost, HDD_{RC} the HDD transfer cost for reading one page and HDD_{WC} the HDD transfer cost for writing one page.

As we can see from Equation A.2, the number of merge passes ($\lceil log_{\lfloor M/C \rfloor - 1}(\lceil N/M \rceil) \rceil$) decreases with decreasing cluster size C, but the number of HDD seeks per pass ($\lceil N/C \rceil$) increases with decreasing C. HDD_{seek} is typically much higher than HDD_{RC} and HDD_{WC}.

$M \downarrow N \rightarrow$	1 GB	10 GB	100 GB	1000 GB
50 MB	1	1	1	2
500 MB	1	1	1	1
1 GB	0	1	1	1
4 GB	0	1	1	1

TABLE A.1. Table showing the optimal number of merge passes for sorting on the flash memory for various memory (M) and data (N) sizes.

Appendix A.2.   The best merge cluster size for flash memory

The best merge cluster size for flash memory is much more trivial to find since flash memory does not have rotational latency and therefore, its random read and sequential read performance is much more similar. Hence, we can approximate the merge IO costs for flash memory ($MCostFlash$) as follows (assuming the output buffer size is one cluster):

$$MCostFlash(N,M,C) = \lceil log_{\lfloor M/C \rfloor - C}(\lceil N/M \rceil) \rceil N (FM_{RC} + FM_{WC})$$
(A.2)

where the terms above have the same definition as for Equation A.2 and FM_{RC} is the cost of reading a page from the flash memory and FM_{WC} is the cost of writing a page into the flash memory.

Equation A.2 shows smaller cluster size C will result in a lower cost of merge on the flash memory since it results in less merge passes with no corresponding increase in IO cost during each pass. This is confirmed by Lee et. al. [1] whose experiments show the optimal cluster size for flash memory is in the range of 2 to 4 KB. Most modern flash memory drives have a page size of 4 KB. Therefore, the optimal cluster size for flash memory is just one page in size.

Having established that the optimal cluster size is one page for flash memory, we show in Table A.1 the number of merge passes ($\lceil log_{\lfloor M \rfloor - 1}(\lceil N/M \rceil) \rceil$) for various M and N sizes. We assume the page size of 4 KB. From the table, we can conclude for most typical memory and data sizes, the optimal number of merge passes is 1. The only case where more than 1 merge pass is needed is when 50 MB of memory is used to sort 1 TB of data.

REFERENCES

[1] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., and Kim, S.-W. (2008) A case for flash memory SSD in enterprise database applications. *SIGMOD Conference*, Vancouver, Canada, 9-12 June, pp. 1075–1086. ACM.

[2] Estivill-Castro, V. and Wood, D. (1992) A survey of adaptive sorting algorithms. *ACM Computing Survey*, **24**, 441–476.

[3] Larson, P.-A. (2003) External sorting: Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, **15**, 961–972.

[4] Dobosiewicz, W. (1984) Replacement Selection in 3-level Memories. *The Computer Journal*, **27**, 334–339.

[5] Andreou, P., Spanos, O., Zeinalipour-Yazti, D., Samaras, G., and Chrysanthis, P. K. (2009) Fsort: external sorting on flash-based sensor devices. *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, 24 August.

[6] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990) *Introduction to Algorithms*. The MIT Press.

[7] Bureau, U. C. (2000). United States Census 2000. http://www.census.gov/main/www/cen2000.html.

[8] Knuth, D. E. (1973) *Sorting and Searching volume 3 of the The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts.

[9] Park, H. and Shim, K. (2009) FAST: Flash-aware external sorting for mobile database systems. *J. Syst. Softw.*, **82**, 1298–1312.

[10] Larson, P.-A. and Graefe, G. (1998) Memory management during run generation in external sorting. *SIGMOD Conference*, New York, NY, USA, pp. 472–483. ACM.

[11] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. (1994) AlphaSort: a RISC machine sort. *SIGMOD Record*, **23**, 233–242.

[12] Zheng, L. and Larson, P.-A. (1996) Speeding up external mergesort. *IEEE Transaction on Knowledge and Data Engineering*, **8**, 322–332.

[13] Zhang, W. and Larson, P.-A. (1998) Buffering and read-ahead strategies for external mergesort. *VLDB Conference*, San Francisco, CA, USA, pp. 523–533. Morgan Kaufmann Publishers Inc.

[14] Estivill-Castro, V. and Wood, D. (1994) Foundations for faster external sorting (extended abstract). *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, London, UK, pp. 414–425. Springer-Verlag.

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

[16] Härder, T. (1977) A scan-driven sort facility for a relational database system. *VLDB Conference*, pp. 236–244. VLDB Endowment.

[17] Graefe, G. (2006) Implementing sorting in database systems. *ACM Computing Survey*, **38**, 10.

[18] Pang, H., Carey, M. J., and Livny, M. (1993) Memory-adaptive external sorting. *VLDB Conference*, San Francisco, CA, USA, pp. 618–629. Morgan Kaufmann Publishers Inc.

[19] Zhang, W. and Larson, P.-A. (1997) Dynamic memory adjustment for external mergesort. *VLDB Conference*, San Francisco, CA, USA, pp. 376–385. Morgan Kaufmann Publishers Inc.

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

[21] Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006) Gputerasort: high performance graphics co-processor sorting for large database management. *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 325–336. ACM.

[22] Six, H. W. and Wegner, L. (1984) Sorting a Random Access File in situ. *The Computer Journal*, **27**, 270–275.

[23] Dufrene, W. R. and Lin, F. C. (1992) An Efficient External Sort Algorithm with no Additional Space. *The Computer Journal*, **35**, 308–310.