EFFICIENT EXTERNAL SORTING ON FLASH MEMORY EMBEDDED DEVICES

Tyler Cossentine and Ramon Lawrence

Department of Computer Science, University of British Columbia Okanagan
Kelowna, BC, Canada
tcossentine@gmail.com ramon.lawrence@ubc.ca

ABSTRACT

Many embedded system applications involve storing and querying large datasets. Existing research in this area has focused on adapting and applying conventional database algorithms to embedded devices. Algorithms designed for processing queries on embedded devices must be able to execute given the small amount of available memory and energy constraints. Most embedded devices use flash memory to store large amounts of data. Flash memory has unique performance characteristics that can be exploited to improve algorithm performance. In this paper, we describe the Flash MinSort external sorting algorithm that uses an index, generated at runtime, to take advantage of fast random reads in flash memory. This algorithm adapts to the amount of memory available and performs best in applications where sort keys are clustered. Experimental results show that Flash MinSort is two to ten times faster than previous approaches for small memory sizes where external merge sort is not executable.

KEYWORDS

sorting, sensor node, flash memory, query processing

1. INTRODUCTION

Embedded systems are devices that perform a few simple functions. Most embedded systems, such as sensor networks, smart cards and certain hand-held devices, are computationally constrained. These devices typically have a low-power microprocessor, limited amount of memory, and flash-based data storage. In addition, some battery-powered devices, such as sensor networks, must be deployed for months at a time without being replaced. Many embedded systems manipulate datasets stored in flash memory.

The contribution of this paper is an external sorting algorithm, called *Flash MinSort,* designed specifically for embedded devices that contain flash memory storage. Depending on the properties of the dataset being sorted, *Flash MinSort* significantly reduces the amount of time and I/O operations when compared to existing algorithms designed for embedded devices. This performance improvement is achieved by generating and maintaining an index on the sort key as the dataset is being sorted. While performing a scan over the dataset, the index enables the algorithm to read only relevant pages of data using random reads. Unlike hard disk drives, flash memory can perform sequential and random page reads with roughly the same latency.

Flash MinSort performs exceptionally well when the sort key values in the input dataset are clustered. Temporally and spatially clustered data can be found in many embedded system applications, such as environmental monitoring with sensor networks. The algorithm also performs well when given sorted or near-sorted data.

An initial version of *Flash MinSort* was presented in [1]. This paper contains a generalized version of *Flash MinSort* that adapts to the amount of memory available and the size of the dataset. The previous version assumed that the size of the dataset was known exactly, which simplified partitioning when generating the index. The ability to adapt to the size of the dataset at

DOI: 10.5121/ijdms.2013.5101

runtime allows the algorithm to be used in a standard database query planner. If the algorithm is provided with more memory than the amount required for the index, it caches subsets of the dataset to reduce the number of reads. This paper contains a detailed analysis and experimental evaluation of *Flash MinSort* when applied to wireless sensor networks and new experimental results using data sets collected by Intel sensor networks [2].

The organization of this paper is as follows. In Section 2 is background on sensor networks and a summary of previous algorithms for flash-based sorting. *Flash MinSort* is presented in Section 3. Section 4 contains a theoretical analysis for *Flash MinSort* and other external sorting algorithms. Section 5 covers the use of *Flash MinSort* in query plans. The experimental evaluation is in Section 6, and the paper closes with conclusions and future work.

2. BACKGROUND

Sensor networks are used in military, environmental, and industrial applications [3, 4]. A wireless sensor node consists of an embedded processor, power management, sensing system, and a communications link [5, 6]. The memory consists of a small amount of random access memory (DRAM) in the range of 1 to 100KB and a larger amount of flash memory (100KB to 10 MB or more) for data storage. Accessing flash memory requires transferring data from the flash device over the bus to the processor and is significantly slower than accessing data in DRAM. Wireless sensor devices can have both internal and external sensor systems. Despite the continuing hardware performance improvement, there is a demand to limit component cost and power consumption which results in the limited node capabilities. Sensor networks used for data collection may have nodes process data locally, always send it back to a collection point called the sink, or do some combination of the two depending on the sensor network configuration. Data processing algorithms must be energy efficient (usually by avoiding transmissions) and consume a small amount of RAM.

A typical flash memory consists of pages (512 bytes to 4 KB) that are combined into blocks (4 KB to 128 KB). Reads and writes are done at the page level, although some devices [7] support direct byte reads. A page must be erased before it can be written. The erase unit is typically a block of pages. The Atmel AT45DB161D [7] flash chip with 16-megabits (2 MB) of storage is used in our experiments. This is a popular chip for sensor nodes and embedded applications due to its low cost, low energy usage, and performance. The chip supports both page and byte reads and supports erase at the unit size of a page (512 bytes), block (4 KB), sector (128 KB), and entire chip (2 MB). The chip has two internal page buffers that can be used for both reading and writing. Read operations read a page from flash to the buffer then transfer bytes from the buffer to the processor. It is also possible to directly stream a page or any sequential bytes from the chip to the processor, bypassing all buffering. Thus, it is possible to read from this device at the byte-level rather than the page-level. Pages are written to the device from an internal buffer.

Flash memory has unique performance properties. On a sensor node, flash memory is directly accessed from a flash chip. Unlike flash devices [8] that provide a block-level interface, a flash chip does not have layers of software for block mapping, wear levelling, and error correction. A flash memory chip has asymmetric read and write costs with writes being between 10 to 100 times more costly, and it supports random reads at approximately the same rate as sequential reads. This is considerably different than hard drives used in other data processing applications. Writes are more costly due to the requirement that flash blocks must be erased before they are written. Most devices have an internal buffer to store one or more pages. The processor will transfer data from an internal flash buffer to its RAM to process it. Although the transfer size between the flash memory and the buffer on the device is in the unit of pages, the transfer between a flash buffer and the processor can be done at the byte level.

For sorting a table of records, we use T to denote the number of tuples (records) in the table and P as the number of pages. L_P is the size of a page in bytes (depends on the flash device) and L_T is the tuple size in bytes (depends on the data). Assuming an unspanned record layout, the number of tuples per page is $N_T = \lfloor L_P / L_T \rfloor$, or for simplicity $N_T = T / P$. The amount of memory available to the sort operator in bytes is M. The size of the attribute(s) sorted, called the sort key size, is L_K. The number of distinct values for the sort key is D. *Flash MinSort* groups pages into regions. We denote the number of regions as R, and the number of pages in a region as N_P. A summary of these parameters is in Table 1.

Table 1. Sorting Parameters.

Notation	Definition
T	number of tuples in table to sort
P	number of pages in table
N_T	number of tuples per page
L_P	page size in bytes
L_T	tuple size in bytes
M	sort memory size in bytes
L_K	sort key size in bytes
D	number of distinct values for sort key
L_I	integer size in bytes
N_P	number of pages in a region
R	number of regions

Query processing on sensor nodes has been studied [9], including issues on local data processing and in-network aggregation and data processing. There have been several algorithms proposed for sorting on flash memory [10, 11, 12] which are designed to do more reads instead of writes due to the asymmetric costs. These algorithms are summarized below including performance formulas. For this discussion, we will assume sorting in ascending order. General external sorting algorithms were surveyed in [13].

The most memory efficient algorithm is the *one key scan* algorithm [10] that performs a read of the table for each distinct sort key value. The algorithm works by storing two key values, *current* and *split*: *current* is the key value that is being output in this scan and *split* tracks the next lowest key value after *current* and is updated while doing the scan. All records with the *current* value are output in order in the current scan. The algorithm needs an initial scan to determine the values of *current* and *split*. The algorithm performs $D + 1$ scans, regardless of the data size, with each pass performing P page I/Os. The major advantage is the memory consumed is only $2*L_K$.

The *heap sort* algorithm, called FAST(1) [12], uses a binary heap of size N tuples to store the smallest N tuples during each scan. Another value, *last*, is maintained which stores the largest sort key output so far. The number of scans is $\lceil T / N \rceil$ regardless of the data. One complication is handling duplicate sort key values. The solution [12] is to remember both the *last* value and the integer *record number* of the last tuple output. During a scan, a tuple is only considered if its key is greater than *last* or its key is equal to *last* and its record number is larger. Duplicates also complicate the heap structure as each record must store its record number as well as the sort key to allow for the heap to maintain the order of duplicates, which occupies $L_I * N$ space. This space overhead can be avoided by using a sorted array as a data structure, but insertions are then O(N) instead of O(log$N$). One page buffer is used as an input buffer. Despite using more memory, *heap sort* is slower than *one key scan* for a small number of distinct sort key values.

External merge sort performs one read pass to construct sorted sublists of size M, which are output to secondary storage. The merge phase then buffers one page from each of the sublists and merges them to produce an output. On each merge pass, $\lfloor M / L_P \rfloor$ - 1 sublists are merged, so multiple merge passes may be required. *External merge sort* and its extensions have two basic issues. First, writing is more expensive than reading, so multiple read scans are often faster than read/write passes. The larger issue is that to execute a sort-merge efficiently requires numerous page buffers in RAM. At minimum, three pages must be available where two would be used to buffer one page of each of the two sublists being merged and another is used to buffer the output. With so few pages, it is common for the algorithm to require many passes which reduces its efficiency. Even three pages may be too much memory for some applications. Three 512 byte pages occupy 1536 bytes, which is a significant amount of the 4K available for small sensor nodes. *External merge sort* becomes more useful as M and P increase.

FAST [12] is a *generalized external merge sort* algorithm that uses FAST(1) to allow multiple passes. FAST uses FAST(1) to perform multiple scans of a subset of the input rather than building more sublists. Thus, instead of sorting up to $\lfloor M / L_P \rfloor$ pages in a single pass like *external merge sort*, the sublist size can be up to Q pages, where $\lfloor M / L_P \rfloor <= Q <= P$. The number of pages Q is configurable. This has the advantage as it avoids writing by using reads instead. The algorithm uses a heap to merge Q sublists in each pass instead of $\lfloor M / L_P \rfloor$ - 1. The optimal selection of Q requires searching all possible values.

FSort [11] is a variation of *external merge sort* with the same merge step but uses replacement selection for run generation. Replacement selection generates runs of approximate size $2M$. The rest of the *external merge sort* algorithm performance is unchanged.

A summary of algorithm performance is in Table 2. The algorithm costs assume that the sort output is not counted. *All of the merge sort variants (external merge sort, FAST, and FSort) also perform writes as well as reads.* None of the algorithms explicitly adapt to the data distribution in the table. The cost to sort a sorted table is the same as the cost to sort a random table. It is common in sensor networks that the sensor data exhibits spatial and temporal clustering that can be exploited. None of the algorithms dominates the others as performance depends on the relative sizes of the sort parameters. An analytical comparison of the ranges of algorithm dominance is in Section 4.

Table 2. Existing Sort Algorithm Performance Formulas.

Algorithm	Memory	Scans	Read Scans	Write Scans
one key	$2*L_K$	$S = D+1$	S	0
FAST(1)	M	$S = \frac{T}{\frac{M - L_P}{L_T + L_I}}$	S	0
merge sort	M	$S = \left\lceil \log_{\lfloor \frac{M}{L_P} \rfloor - 1} \left\lceil \frac{P * L_P}{M} \right\rceil \right\rceil$	S + 1	S
FAST	M	$S = \left\lceil \log_Q \left\lceil \frac{P}{Q} \right\rceil \right\rceil$	S + 1	S
FSort	M	$S = \left\lceil \log_{\lfloor \frac{M}{L_P} \rfloor - 1} \left\lceil \frac{P * L_P}{2 * M} \right\rceil \right\rceil$	S + 1	S

3. FLASH MINSORT ALGORITHM

The core idea of the *Flash MinSort* algorithm is that random page reads can replace sequential scans to reduce the amount of I/O needed to sort a relation. This algorithm is specifically designed for data stored in flash memory, where random reads have roughly the same cost as sequential reads. All previous algorithms perform sequential scans of the input relation. *Flash MinSort* takes advantage of low-cost random reads by building a simple dynamic index (*minimum index*) that stores the smallest sort key value for each region of the input relation. The index is implemented as an array and it is searched by performing a linear scan. A *region* represents one or more adjacent pages of data in flash memory and the *sort key* is the attribute that the relation is being sorted on. Instead of reading the entire relation during each pass, the algorithm only reads the pages of regions that contain the current sort key value being sent to output. Once all pages in a region have been read, its index value is updated with the next smallest sort key value that was encountered. This process repeats until the entire relation has been output in sorted order.

In the ideal case, each region consists of a single page of data. The amount of space required to store an index value for each page is $L_K * P$, which may be larger than the amount of available memory (M). Thus, we group adjacent pages into regions by computing the maximum number of sort key values that can be stored in memory. The algorithm is adaptable to the amount of memory available and the minimum amount of memory required is $4 * L_K + L_I$ for two regions. With two regions, only two sort key values must be stored by the index.

Flash MinSort keeps track of the *current* sort key value being output and the *next* smallest sort key value encountered while searching a region. It also records the location of the next tuple to be accessed in a region (*nextIdx*). After finding a tuple that has a sort key value equal to *current* and sending it to output, the algorithm continues its scan through the pages and tuples of the region. If it encounters another tuple with a sort key value equal to *current*, it stops and sets *nextIdx* to that location. When the next tuple is requested, the search continues with the tuple located at *nextIdx*. This algorithm guarantees that tuples with the same sort key value are output in the same order that they appear in the input relation.

As the algorithm is scanning a region for tuples with a sort key value equal to *current*, it is simultaneously keeping track of the *next* smallest sort key value encountered. Once the end of the region has been reached, the minimum index value of the region is set to the value of *next*. Since a region is always scanned from the beginning, all tuples are considered when determining the next minimum index value.

Figure 1 contains an example with T=48, P=12, N_T=4, L_K=L_I=4, L_T=20, D=9, D_R=2.3 and M=60 bytes. The first two passes over the minimum index are shown. Each region represents a single page of tuples and a tuple represented by a sort key value with a rectangle around it is sent to output. A sort key value inside of a circle is used to determine the next smallest value in the region being scanned.

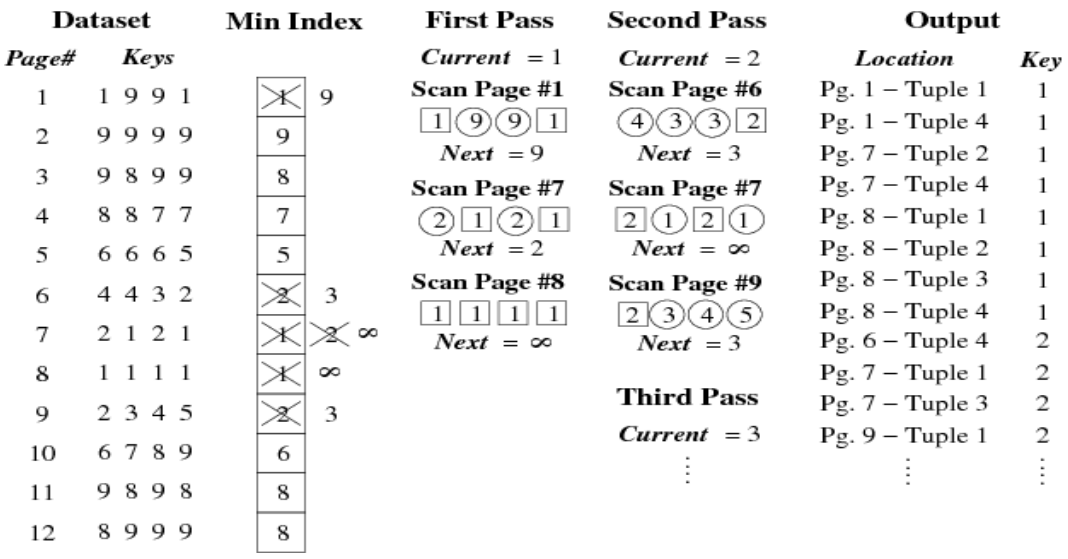

Figure 1. MinSort Example

To initialize the minimum index, *Flash MinSort* reads the entire relation to determine the smallest sort key value in each region. The first pass begins by performing a linear scan on the index. It encounters region 1, which has a minimum value equal to *current*. Page 1 is loaded into memory and the algorithm searches for a tuple with a sort key value equal to 1. The first tuple in the page is sent to output. The algorithm continues searching the tuples in the page, updating the next minimum value of the region as it encounters sort key values greater than *current*. At the second tuple, the minimum value of the region is updated to 9. When the algorithm encounters another tuple with a sort key value equal to *current* at tuple 4, it sets *nextIdx* to 4. When the next tuple is requested, page 1 is currently in memory and the algorithm jumps directly to tuple 4 to send it to output. The minimum index value of region 1 is set to 9.

The algorithm continues to perform a linear scan through the index until it encounters region 7. Page 7 is loaded into memory and it is searched in the same manner as page 1. The process of scanning the index and searching pages continues until all tuples with a sort key value equal to *current* have been sent to output. A pass over the index is performed for each distinct sort key value.

Pseudocode for *Flash MinSort* is shown in Figure 2. The first three lines calculate the number of regions and the number of pages per region. These values depend on the amount of memory (*M*) and the number of pages that store the relation. Each iteration of the while loop proceeds in three stages. In stage one, the next region to be searched is determined by scanning the index until a region is found that has a minimum value equal to *current*. This stage is skipped if the algorithm is in the middle of scanning a region (*nextIdx* > 0).

The second stage searches the region for a tuple with a sort key value equal to *current*. If *nextIdx* is 0, the search begins from the start of the region; otherwise, the search left off at the next tuple to be sent to output. While the search proceeds, the *next* smallest value for the region is updated. At the end of this stage, the next smallest tuple in the relation has been sent to output. The final stage updates the minimum index value of the region. This value is either the next tuple (if any) for sorted regions, or it requires all remaining tuples in the region to be read. This search terminates early if another tuple with sort key value equal to *current* is found in the region. In that case, *nextIdx* is set to that tuple's location.

```
procedure FlashMinSort()
    numPagesPerRegion = ⌈(numPages*L_K)/(M−2*L_K−L_I)⌉
    numRegions = ⌈numPages/numPagesPerRegion⌉
    Scan input and update min array with smallest value in each region
    nextIdx = 0;
    while (data to sort)
        // Find region with smallest value
        if (nextIdx == 0)
            i = location of smallest value in min array
            current = min[i];
            next = maxvalue;
        end if

        // Find current minimum in region
        startIndex = nextIdx;
        Scan region i starting at startIndex looking for current
        During scan update next if (key > current AND key < next)
        Output tuple with key current at location loc to sorted output

        // Update minimum in region
        if (sorted region)
            current = r.key of next tuple or maxvalue if none
            nextIdx is 0 if next key ≠ current, or next index otherwise
        else
            nextIdx = 0;
            for each tuple r in region i after loc
                if (r.key == current)
                    nextIdx = location of tuple in region
                    break;
                end if
                if (r.key > current AND r.key < next)
                    next = r.key;
            end for
            if (nextIdx == 0)
                min[i] = next;
        end if
    end while
end procedure
```

Figure 2. Flash MinSort Algorithm

3.1. Adaptive to Memory Limits

The base version of the algorithm in Figure 2 does not adapt to the input relation size. The number of regions was calculated statically based on a known relation size. In a real system, the

size of the input relation is an estimate, and the operator must adapt to poor estimates. Further, if the input was smaller than expected, perhaps small enough to perform a one-pass sort, *Flash MinSort* would perform needless I/Os as it would allocate more regions than required. To resolve this issue, we demonstrate in this section how the number of regions can be dynamically adjusted as the input relation is processed. This allows *Flash MinSort* to gracefully degrade from one-pass sort by increasing the number of regions as required.

First, consider the case where the amount of memory available (M) is less than a full page. The algorithm will use this memory to store the minimum value in each region. The challenge is that without knowing the exact size of the input relation, we do not know how many pages are in a region during initialization. The solution to this problem is to dynamically build the minimum index by merging regions once the memory limit has been reached. The basic idea is that we start with the assumption that the index will consist of one page per region. We perform a linear scan through the relation and fill the index with the minimum value for each page encountered. Once the index is full, we merge adjacent regions. We continue to scan through the relation, but now each region represents two pages of data. This process repeats until the entire relation has been scanned. If the iterator does not see page boundaries, it treats each tuple as a region and has the issue of potentially crossing page boundaries when traversing a given region.

Consider the example in Figure 3 In this example, M=32 bytes and the number of pages (P=12) is unknown to the algorithm. With this amount of memory, a maximum of five regions (20B) can be stored in the index since twelve bytes are used by other variables in the algorithm. Figure 3 shows how the algorithm builds the index by merging adjacent regions. The first five page pages are read and their minimum sort key values are inserted into the index. Once the sixth page is read, there is no space to store that region in the index, so adjacent regions in the index are merged. Each entry in the minimum index represents two adjacent pages. The next entry in the index (1) represents pages seven and eight. The following entry (2) represents pages nine and ten. Once page eleven is read, adjacent regions are merged and each region now represents four adjacent pages. After reading all input, the number of regions is three and each region represents four pages.

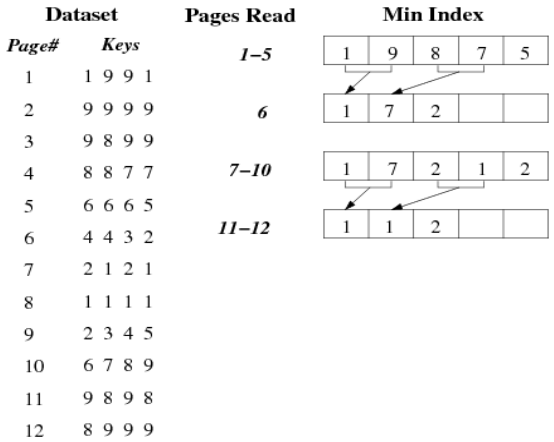

Figure 3. Dynamic Region Size Example

If the amount of memory available is larger than a page, there is potential for further optimization. More specifically, it may be possible to perform a one-pass sort if the entire input relation fits into memory. The goal is to gracefully degrade from one-pass sort to building the index and increasing the number of pages per region as required. The memory provided to the algorithm is treated as a large byte array. It is shared by the minimum index, allocated from the

start of the array, and a cache of input pages/tuples. If the minimum index becomes full during the initial scan, memory is returned to the operator by discarding cached pages. Once all cache pages have been discarded, the minimum index is merged when it becomes full.

The algorithm begins by caching pages, or tuples (depending on the operator below), in memory. If the entire relation fits into available memory, an in-place sort is performed. If the buffer becomes full while performing the initialization pass, the first page is released. This page is assigned to the minimum index and the first entry in the index stores the smallest sort key value of the page that was just released. The minimum sort key value of each page located in the cache is added to the index. At this point, the buffer consists of the minimum index, containing the smallest sort key value of each page encountered so far, and one or more cached data pages. As each page is read, the minimum sort key value is determined. A newly read page is cached by overwriting the page with the largest minimum sort key value. Potentially, all of the buffer will contain the minimum index and there is still is not enough memory to store an index value for each page. At this point, the previous algorithm that produces regions representing multiple adjacent pages is used.

Pages 1–5

Page#	Data
1	1 9 9 1
2	9 9 9 9
3	9 8 9 9
4	8 8 7 7
5	6 6 6 5

Page 6

Page#	Data
min	1 9 8 7
min	5 2 _ _
6	4 4 3 2
4	8 8 7 7
5	6 6 6 5

Page 7

Page#	Data
min	1 9 8 7
min	5 2 1 _
6	4 4 3 2
7	2 1 2 1
5	6 6 6 5

Page 8

Page#	Data
min	1 9 8 7
min	5 2 1 1
6	4 4 3 2
7	2 1 2 1
8	1 1 1 1

Page 9

Page#	Data
min	1 9 8 7
min	5 2 1 1
min	2 _ _ _
7	2 1 2 1
9	2 3 4 5

Page 10

Page#	Data
min	1 9 8 7
min	5 2 1 1
min	2 6 _ _
7	2 1 2 1
10	6 7 8 9

Page 11

Page#	Data
min	1 9 8 7
min	5 2 1 1
min	2 6 8 _
7	2 1 2 1
11	9 8 9 8

Page 12

Page#	Data
min	1 9 8 7
min	5 2 1 1
min	2 6 8 8
7	2 1 2 1
12	8 9 9 9

Figure 4. Dynamic Input Buffering Example

Figure 4 shows an example of the algorithm using our running example. There are five pages of memory are available. In the diagram, the *Page#* column shows what page is in that buffer slot (either an input page number or *min* to indicate the page is used by the minimum index), and the *Data* column shows the actual data in that page.

The first five pages are directly read into the buffer. Before reading page 6, a cached page must be released. Since the algorithm cannot do a one-pass sort, it determines the smallest sort key value of each page and builds the minimum index. The minimum index occupies the entire first page and the first element of the second page. Since page 3 has the largest index value, it is released and page 6 is loaded into that location. An entry for page 6 is added to the index. Now that page 4 has the largest index value, it is released from the cache and page 7 is loaded into that location. An entry for page 7 is added to the index. This process continues and page 8 replaces page 5 in the cache. Loading page 9 requires a new page to be allocated to the minimum index, and page 7 is released. Pages 10 to 12 are read and their minimum sort key values are added to the index.

After reading the entire relation, three pages are used to store the minimum index and two pages are used to cache input. As the sort proceeds, those two pages will store the most recently read pages. A cached page is selected for release if it has the largest index value of all cached pages.

Note that for most datasets the minimum index will consume a small amount of memory relative to the data page size. In the example, the assumption is that each page can only store four integers and that tuples in the input page consist only of the sort key. In practice, the sort key is often a small number of bytes relative to the tuple size, and tens of tuples could be stored on a page. Hence, a minimum index page would be used to index more than four regions.

3.2. Exploiting Direct Byte Reads

An optimization for the *Flash MinSort* algorithm is to read only sort keys instead of entire tuples. An entire tuple only needs to be read when it is being sent to output. Otherwise, only the sort key is read to determine the next smallest value in the region. This has the potential to dramatically reduce the amount of I/O performed and the amount of data sent over the bus from flash memory to the processor. If a flash storage device supports direct byte addressable reads and the sort key offsets can be easily calculated, searching for the minimum key in the region does not require reading entire pages.

Figure 5 gives an example of direct byte reads. In this example, a page contains 512 bytes and each tuple is 16 bytes wide. Each page contains exactly 32 tuples. The highlighted *value* attribute is 4 bytes wide and it is used as the sort key. Since tuples are fixed in size and do not span multiple pages, the offset of the value attribute in every tuple can be calculated. If the storage device is directly byte addressable, only 128 bytes need to be read to examine all sort keys in a page. If it is not byte addressable, all 512 bytes must be read.

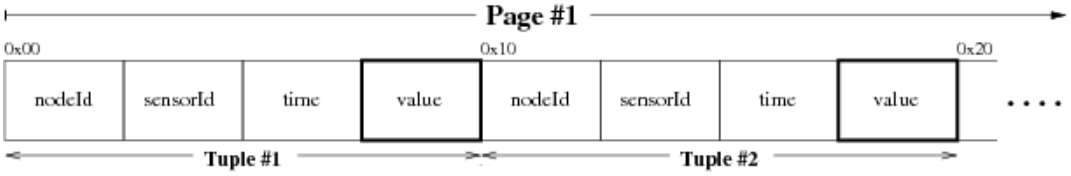

Figure 5. Direct Byte Reads

3.3. Detecting Sorted Regions

An optimization for sorted regions allows the algorithm to avoid scanning the entire block for the next minimum. Detecting sorted regions is an optimization that can be done during the initial scan that determines the minimum values in the regions and requires at least one bit of space per region.

4. THEORETICAL ANALYSIS

In this section, we examine the theoretical performance of the external sorting algorithms based on the properties of the input dataset.

4.1. Flash MinSort Performance

The performance of Flash MinSort is especially good for data sets that are ordered, partially ordered, or exhibit data clustering. If a region consists of only one page, then in the worst case a page I/O must be performed for each tuple for a total of $P+T$ page I/Os. It is possible that the entire page must be scanned to find the next minimum value resulting in $T+T * N_T$ tuple I/Os. If a region consists of multiple pages, then in the worst case a whole region must be read for every tuple output (and a minimum calculated). Then the number of page I/Os is $P+T*N_P$ and the number of tuple I/Os is $T+T*N_P*N_T$.

In the best case, which occurs when the relation is sorted, the number of page I/Os is $2*P$ (first pass to determine if each page is sorted and to calculate minimums and a second pass that reads pages and tuples sequentially). The number of tuple I/Os is $2*T$. If the relation is reverse sorted,

the page I/Os are $P+T*N_P$ as it reads each page once and the tuple I/Os are $T+T*N_P*N_T$ as it must search the entire region for the next minimum every time.

On average for random, unsorted data the performance depends on the average number of distinct values per region, D_R. The algorithm scans a region for each distinct value it contains. Each scan reads all tuples and pages in a region. Average page I/Os is: $P + R * D_R * N_P = P * (1+D_R)$ and average tuple I/Os is: $T+ R * D_R * N_P * N_T = T * (1 + D_R)$. With a sorted region, the algorithm does not scan the region for each distinct value as long as it does not leave the region and return. If the algorithm leaves a region, it must start the scan from the beginning again since it does not remember its last location. A binary search can be used instead of a linear search from the beginning for a sorted region. We have also investigated the performance of storing both the minimum value and the offset in the region to avoid scanning the region, but the results did not show an improvement as additional memory is consumed that is better used to reduce the region size.

Considering only byte I/Os, the amount transferred in the worst case is $T * L_K + T * L_T + T * N_P * N_T * L_K$, the average case is $T * L_K + T * L_T + R * D_R * N_P * N_T * L_K$, and the best case is $T * L_K + T * L_T$. The term $T * L_K$ is the cost to perform the initial scan and compute the minimums for each region. This scan does not need to read the entire tuple (or pages), but only the key values. The second term, $T * L_T$, is the cost to read and output each tuple in the file in its correct sorted order. The last term varies depending on the number of region scans. Each region scan costs $N_P * N_T * L_K$ as the key for each tuple in the region is read. In the best case, a region is scanned only once and tuples are never revisited. In the worst case, each tuple will trigger a region scan, and on average the number of region scans is $R * D_R$.

In the example in Figure 1, the number of page reads is 39, tuple reads is 148, and bytes read is 1444. In comparison, *one key sort* performs 10 passes reading all pages for a total of 120 page I/Os, 480 tuple I/Os, and 9600 bytes. The *FAST(1) heap sort* is able to only store 3 records in the heap (ignoring all other overheads of the algorithm) and performs 16 passes for a total of 192 page I/Os, 768 tuple I/Os, and 15,360 bytes. *One key sort* reads three times more pages and over six times more bytes than *Flash MinSort*, and *heap sort* reads almost five times more pages and over ten times more bytes. This data exhibits a typical continuous function common for sensor readings.

In the worst case with a random data set with all distinct sort key values, *Flash MinSort* has costs of 60 page I/Os, 240 tuple I/Os, and 4800 bytes which is still considerably better than the other two algorithms. The direct read version of *Flash MinSort* would only read 1920 bytes.

4.2. Algorithm Comparison

The performance of *one key scan* depends directly on the number of distinct sort keys D. The performance of *heap sort* depends on the sort memory size M and tuple size L_T. *One key scan* will be superior if $D + 1 < \frac{T}{\frac{M-L_P}{L_T+L_I}}$ or for simplicity $D < T * L_T / M$. If the number of distinct values is small or the number of tuples or their size is large, *one key scan* will dominate. Since M is small, *one key scan* dominates for sensor applications until D approaches T.

Flash MinSort always dominates *one key scan* in both page I/Os: $P(1+D_R) < P(1+D)$ and tuple I/Os: $T(1+D_R) < T(1+D)$ as D_R the average number of distinct values per region is always less than the number of distinct values for the whole table D.

Basic *Flash MinSort* dominates *heap sort* when $1+D_R < T * L_T / M$. *Flash MinSort* is superior unless the size of the table being sorted $T * L_T$ is a small fraction of the available memory (e.g. input table is only twice the size of available memory). In the worst case, $D_R=N_T$ (each tuple in a page is distinct), *Flash MinSort* will dominate unless the ratio of the input size to the memory size is less than the number of tuples per page. Given the amount of memory available, this is very rare except for sorting only a few pages.

The adaptive version of *Flash MinSort* changes the analysis slightly. First, both algorithms will perform a one-pass sort if the input fits in memory with identical performance. When the input is slightly larger than memory, *Flash MinSort* builds its minimum index and uses the rest of the space available for buffering input pages. The pages are buffered according to their minimum values (keep pages with smallest minimum in cache). In general, *Flash MinSort* will perform $1 + D_R * (1\text{-}hitRate)$ I/Os per page in this case, with *hitRate* being the cache hit rate on each request. A rough estimate of cache hit rate can be calculated by determining the percentage of the input relation buffered in memory which is $(M - P * L_K) / (T * L_T)$ or approximately $M / (T * L_T)$ since the space used for the minimum index, $P * L_K$, will typically be small. Thus, adaptive *Flash MinSort* will dominate *heap sort* when $1 + D_R * M / (T * L_T) < T * L_T / M$.

As a best case example for *heap sort*, assume a 2 to 1 input to memory size ratio with M=2000 bytes, $T*L_T$=4000 bytes, L_P=500 bytes, and P=8 pages. The number of passes for *heap sort* is 2, so each page is read twice. The number of times each page is read by *Flash MinSort* is $1 + D_R$, and the cache hit rate is approximated by $(M - P * L_K) / (T * L_T) = (2000 – 16) / 4000 \approx 0.50$. The actual number of reads per page is $1+0.5 * D_R$. The value of D_R will determine the superior algorithm, but the addition of the input page cache makes *Flash MinSort* much more competitive in this memory ratio range.

In comparison to *external merge sort*, the relative performance depends on two critical factors: the number of distinct sort keys and the write-to-read time ratio. The number of distinct sort keys affects only *Flash MinSort*. The write-to-read time ratio is how long a write takes compared to a read. As each pass in the sort merge algorithm both reads and writes the input, a write ratio of 5:1 would effectively cost the equivalent of 6 read passes. To simplify the discussion, we will assume that *external merge sort* is given sufficient memory to only require two passes. In practice, this is highly unlikely due to the device memory constraints. With this amount of memory, *Flash MinSort* is able to have a region be one page, and the minimum index consumes a small amount of memory leaving a lot of memory for input buffering. If the write-to-read ratio is X, then *Flash MinSort* dominates if $P * (1+D_R) < (2+X)*P$ or $D_R < X+1$. Since the common ranges of the write-to-read ratio are from 5 to 100, and D_R is bounded by the number of records that can fit in a page (N_T), *Flash MinSort* will dominate *external merge sort* for a large spectrum of the possible configurations even while using considerably less memory and performing no writes. Similar to the previous analysis, the adaptive version of *Flash MinSort* reduces the number of actually I/Os performed based on the cache hit rate which has a significant affect for input to memory ratios in the range of 1 to 10.

The previous analysis considered only complete page I/Os, if the flash chip allows direct memory reads, the performance of *Flash MinSort* is even better. As discussed in Section 3.2, *Flash MinSort* will only read the keys when scanning a page to update its minimum index and only retrieve the tuple required from a page rather than the whole page when outputting individual tuples. The result is considerable savings in bytes transferred from the device and bytes transferred from device buffers to the processor over the bus.

5. SORTING IN DATA PROCESSING

Sorting is used extensively in data processing for ordering output, joins, grouping, and aggregation. For sorted output, the sort operator is typically applied at the end of the query plan. Sorting used for joins, grouping, and aggregation requires the algorithm to be implemented in an iterator form. This section discusses some of the issues in using *Flash MinSort* in iterator-based query plans.

Sorting a base table can be done with or without using an iterator implementation as the algorithm has direct access to the table stored in flash. *Flash MinSort* requires the ability to perform random I/Os within the input relation. At first glance, *Flash MinSort* does not work well in the

iterator model as it requires the input relation to be materialized to allow for random reads that it uses to continually get the next smallest tuple. One simple solution would be to materialize the input relation before the operator. Materialization is typically used [10] as an alternative to rescanning the input many times which is often more costly than materialization depending on the complexity of the subplan. However, in many cases avoiding materialization is preferable due to the large write cost and the temporary space that is required.

A better alternative is to exploit the well-known idea of interesting orders for sorting [14]. Instead of executing the sort as the top iterator in the tree, the sort can be executed first during the table scan and ordering preserved throughout the query plan. This allows *Flash MinSort* to execute without materialization. Depending on the query plan, early sorting with *Flash MinSort* may still be more efficient than performing sort as the last operation using other algorithms.

Consider a query plan consisting of a base table scan, selection, projection, and sort to order the output. The plan with the sort on top is only executable with *Flash MinSort* if the input from the projection is materialized first. However, if the sorting is done first the plan is executable and may still be more efficient than the original plan using another sort algorithm. The selection potentially reduces the number of distinct values to be sorted, and both operators reduce the size of the input relation in terms of bytes and pages. Let σ represent the selectivity of the selection operator, and α represent the reduction in input size from projection. Thus, if the original table was of size $T * L_T$ the sorted relation size is $\sigma * \alpha * T*L_T$. The cost formulas in the following section can be modified by multiplying by σ and α to compare the performance of *Flash MinSort* with the other operators. A similar analysis holds for plans with joins, so the query optimizer can easily cost out all options to select the best plan.

As an example, consider a query plan involving a sort, projection, selection, and base table scan. The base table has P=20 pages with L_P=500 bytes, so the input size is 10000 bytes. Assume M=1000 bytes, the selectivity σ = 0.5, and the size reduction due to projection α =0.4. The effective input size for the sort if performed as the top operator of the query plan is 10,000 bytes * 0.5 * 0.4 = 2000 bytes. Since M=1000 bytes, two passes would be required for *external merge sort* or *heap sort*. For *external merge sort*, this involves writing 2000 bytes to flash as sorted runs and merging. For *heap sort*, unless the input is materialized, this requires executing the subplan twice. Thus the total I/Os is 20,000 bytes (as the input relation needs to be scanned, filtered, and projected twice). If *Flash MinSort* was executed above the base table scan to allow random I/Os, its effective input size is 10,000 bytes. The number of input table scans is $1 + D_R$. Depending on the value of D_R, *Flash MinSort* may have as good performance as *heap sort* despite sorting a larger input. Clearly, the best choice depends on the ratio of the input size to memory size for both algorithms and the selectivity of the plan. Note that subplans with non-materialized joins would be especially costly to re-execute if performing *heap sort*.

6. EXPERIMENTAL EVALUATION

The experimental evaluation compares *Flash MinSort* with *one key sort*, *heap sort*, and the standard *external merge sort* algorithm. The sensor node used for evaluating these algorithms has an Atmel Mega644p processor clocked at 8 MHz, 4KB of SRAM, and a 2MB Atmel AT45DB161D [7] serial flash chip. The maximum amount of memory available to an operator is 2KB, with the rest of system memory used for critical node functionality. The serial flash has a page size of 512 bytes. This sensor node design was used for field measurement of soil moisture and integrated with an automated irrigation controller [15]. The system was designed to take sensor readings at fixed intervals and periodically send data back to the controller.

6.1. Raw Device Performance

The performance of the Atmel DataFlash chip was tested by benchmarking the read and write bandwidth. The data used for benchmarking contained 50,000 records. The Atmel chip provides

three different read mechanisms: direct byte array reads to RAM, direct page reads to RAM, and page reads to an internal buffer and then to RAM. We constructed three types of file scans: one that reads individual tuples using a direct byte array read, a second that reads a whole page to RAM, and a third that reads a page into an on-chip buffer then access the tuples on the page one at a time. The time to scan the file with each of these methods was 5.31, 3.68, and 5.76 seconds respectively. Thus, buffering has limited performance difference compared to direct to RAM reads. However, there is a performance difference in transferring large amounts to RAM from flash memory (buffered or not) as there are fewer requests to be sent over the bus with each request having a certain setup time. Although there is a full page memory cost of doing the direct page read, we use it for *one key sort*, *heap sort*, and *Flash MinSort* to improve their performance and do not count this memory usage for the algorithm. The first two algorithms especially benefit because they perform numerous sequential scans of the data.

The direct byte array read feature allows *Flash MinSort* to read only the sort keys instead of the whole page. We tested two types of key scans. The first reads only the keys directly from flash and the second reads keys from a page stored in a flash buffer. For 16 byte records (32 records per page), the time to perform a key scan using these methods was 2.13 and 2.64 seconds respectively. We use the first method that reads keys directly from flash since it has the best performance and does not require buffering a page in RAM for good scan performance. The performance of this direct read increases further as the record size increases relative to the key size. The ability to only read bytes of interest has a significant performance improvement, primarily due to the time to transfer the data over the bus to the CPU from the device.

6.2. Real Data

The real dataset consists of 100,000 records (1.6MB) collected by a sensor network during Summer 2009 [15]. The schema consists of 12 fields with total size of 16 bytes. The data used for testing is a 10,000 record (160KB) subset of the sensor network data. Since the individual sensor nodes collected soil moisture, the data has few distinct values and they are temporally clustered. There are 42 distinct sort key values and the average number of distinct sort key values per page is 1.79. The performance of the algorithms by time and I/Os is in Figure 6 with varying memory sizes. Note that the charts do not display the data for *heap sort* as its times are an order of magnitude larger. For a memory size of 100 bytes (4 tuples), the time is 3,377 seconds and for 1200 bytes (60 tuples), the time is 302 seconds. *Heap sort* is not competitive on this device since the maximum amount of memory available is 2KB.

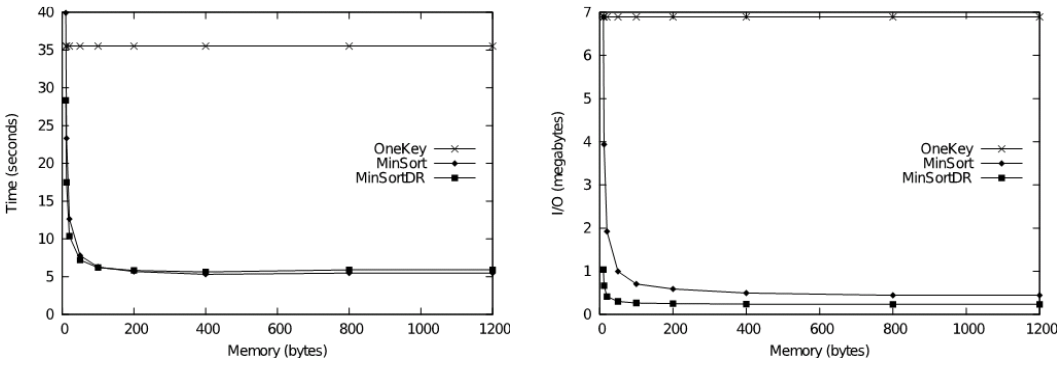

Figure 6. MinSort – Real Data

One key sort has good performance due to the small number of distinct sort key values. This type of data is common in sensor applications due to the use of 10-bit analog-to-digital converters. The performance of *one key sort* does not improve with additional memory.

There are two implementations of *Flash MinSort*: basic *Flash MinSort* transfers a complete page from the flash to RAM and *MinSortDR* performs direct reads of the sort keys from flash. All

algorithms, with the exception of *MinSortDR*, require an I/O buffer in memory. This buffer is common to all algorithms that perform I/O and it is not included in the cost. *MinSortDR* performs fewer I/Os than regular *Flash MinSort* and is faster for small memory sizes. For clustered data, this performance advantage decreases as more memory becomes available since *Flash MinSort* will output a greater number of records on each page it retrieves. The relative performance of *MinSortDR* would be even better if the dataset had a larger record size. With 32 records per page, there are 32 separate I/O operations to retrieve 2 bytes at a time. Since there is an overhead to each I/O operation, direct reads of the sort keys is not much faster than reading the entire page of data in a single call.

External merge sort requires a minimum of three pages (1,536B) of memory to sort a dataset. With three pages of memory, seven write passes (1.12MB) and eight read passes (1.28MB) are performed with a run time of 76 seconds. Given little memory, *Flash MinSort* is faster than *external merge sort* and it does not require any writes to flash. As memory increases, *external merge sort* becomes more competitive. However, for small memory sizes typically found on wireless sensor nodes, *external merge sort* is not executable.

6.3. Random Data

The random data set consists of the 10,000 records, but each original data value was replaced with a randomly generated integer in the range from 1 to 500. This number of records was selected as the performance of *one key sort* becomes too long for larger relations, and it is realistic given that sensor values are commonly in a narrow range. The performance of the algorithms by time and I/Os is shown in Figure 7. Both *heap sort* and *one key sort* have the same execution times regardless of the data set (random, real, or ordered). *External merge sort* took 78 seconds for the random data set.

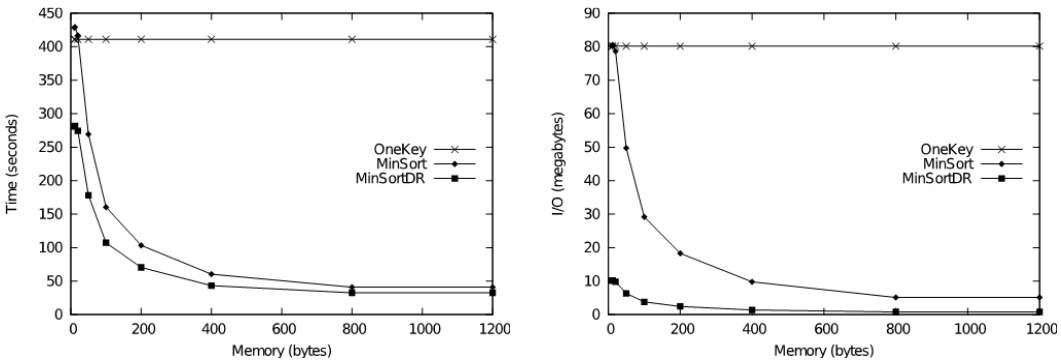

Figure 7. MinSort – Random Data

6.4. Ordered Data

The ordered data set consists of the same 10,000 records as the real data set except pre-sorted in ascending order. The results are in Figure 8. *Flash MinSort* dominates based on its ability to adapt to sorted inputs. The basic *Flash MinSort* implementation does not explicitly detect sorted regions but still gets a benefit by detecting duplicates of the same value in a region. *MinSortDR* stores a bit vector to detect sorted regions as a special case. This along with only retrieving the bytes required gives a major advantage. *One key sort* is still competitive while *heap sort* is not for these memory sizes. *Heap sort* has the same execution time as the previous two experiments. *External merge sort* took 75 seconds.

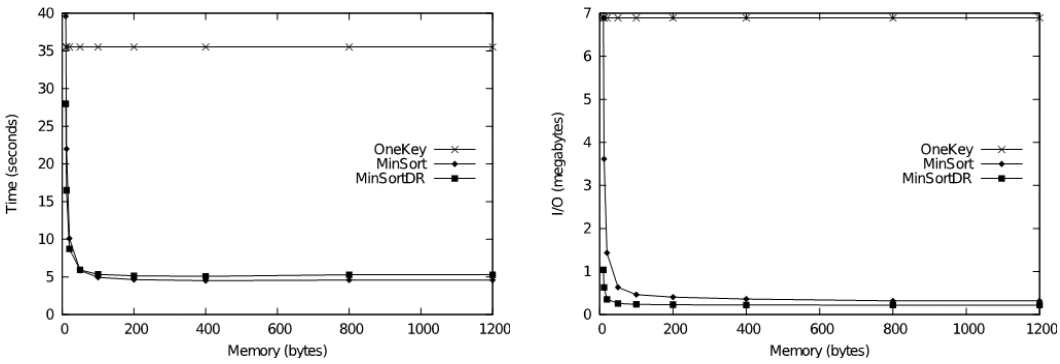

Figure 8. MinSort – Ordered Data

6.5. Berkeley Dataset

The Berkeley dataset was from a sensor network at the Intel Berkeley Research Lab [2]. This dataset contains 2.3 million records collected by 54 sensor nodes with on-board temperature, light, humidity and voltage sensors. Each record is 32 bytes in size, and each page of flash contains 16 records. A 5,000 record subset was used to evaluate the sorting algorithms. The number of distinct sort keys and the average number of distinct sort keys per page is in Table 3.

Table 3. Berkeley Data Sets and Distinct Values

Sorted Attribute	Total Distinct	Average Distinct Per Page
Temperature	847	9.03
Humidity	467	8.14
Light	62	2.90

Figure 9 shows the execution time and I/O of the algorithms when sorting the dataset on the light attribute. Figure 10 shows data for the humidity attribute. Figure 11 shows the results of sorting on the temperature attribute. External merge sort took 60 seconds to complete for all three experiments with three pages of memory. The results for all algorithms display the same relative performance as the previous experiments.

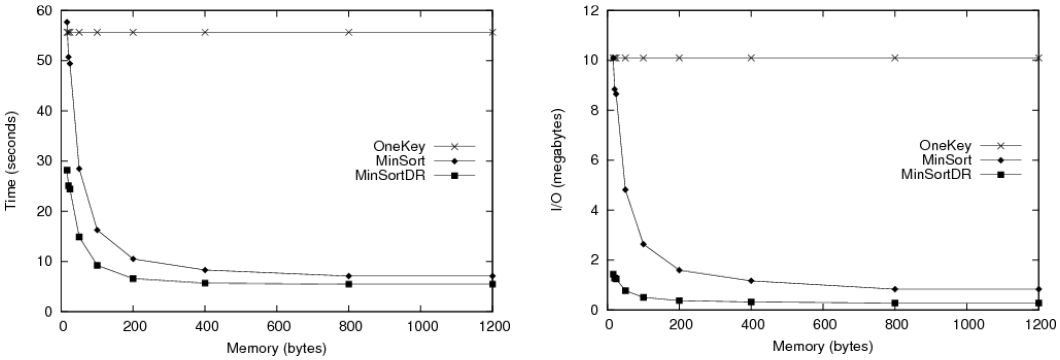

Figure 9. Sorting on Light Attribute

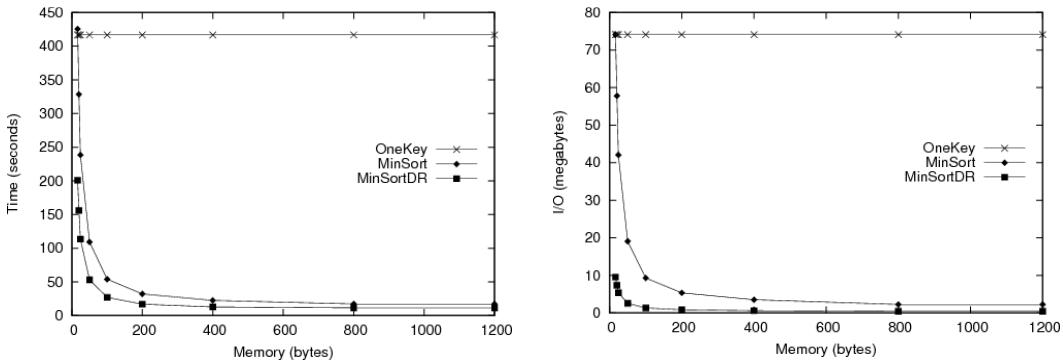

Figure 10. Sorting on Humidity Attribute

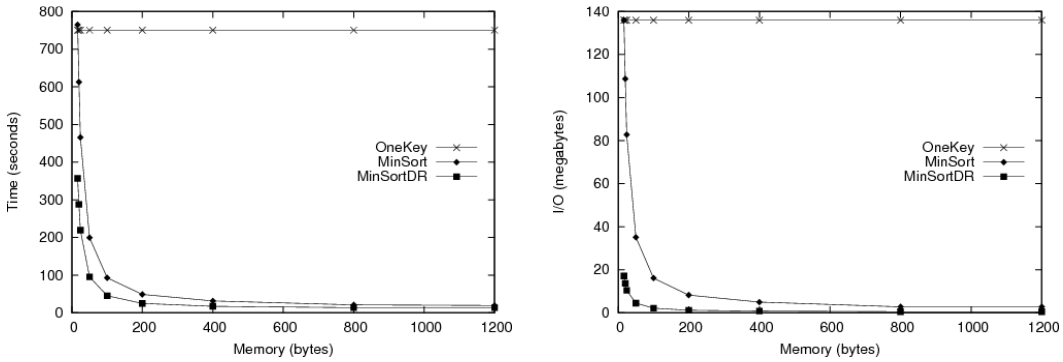

Figure 11. Sorting on Temperature Attribute

6.6. Adapting to Memory Limits

Adapting to memory limits imposes almost no performance penalty to the algorithm and may even increase performance. The left-side of Figure 12 shows the real dataset and the right-side shows the random dataset. The algorithms are provided with a fixed 2KB of memory and the size of the dataset is gradually increased from 800B (50 tuples) to 9600B (600 tuples). The adaptive version of *Flash MinSort* outperforms the base version when the dataset can be sorted in-place. At 200 tuples, the dataset cannot be sorted in-place and the adaptive version has a slight advantage because it caches pages in memory instead of reading from flash. As the dataset grows larger, the caching performance advantage disappears since the number of cache hits is small relative to the total number of pages read.

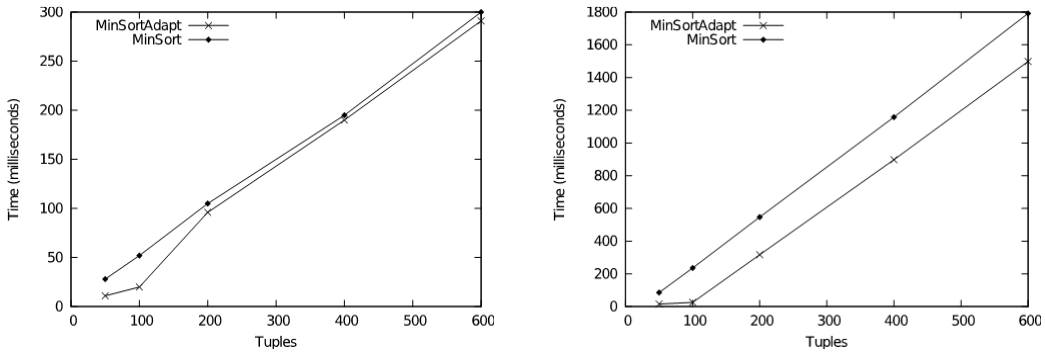

Figure 12. In-Place Optimization (Time). Right-side: Random Data, Left-side: Real Data

Figure 13 demonstrates the performance of the two versions of the algorithm with small memory sizes. The dataset is 160KB (10,000 tuples). The base version of the algorithm has a performance advantage because the size of the dataset is known when determining the number of pages represented by a region. The adaptive version of the algorithm determines the region size as it performs the initial scan of the relation. Given 50 bytes of memory, both versions of the *Flash MinSort* algorithm have 20 regions in the index, with each region representing 16 pages of tuples. Increasing the amount of memory to 75 bytes, the base version of the algorithm has 33 regions, with each representing 10 pages of tuples. The adaptive version does not take advantage of the additional memory and still has 20 regions in the index. The performance difference between the two versions at these points can clearly be seen in the figures.

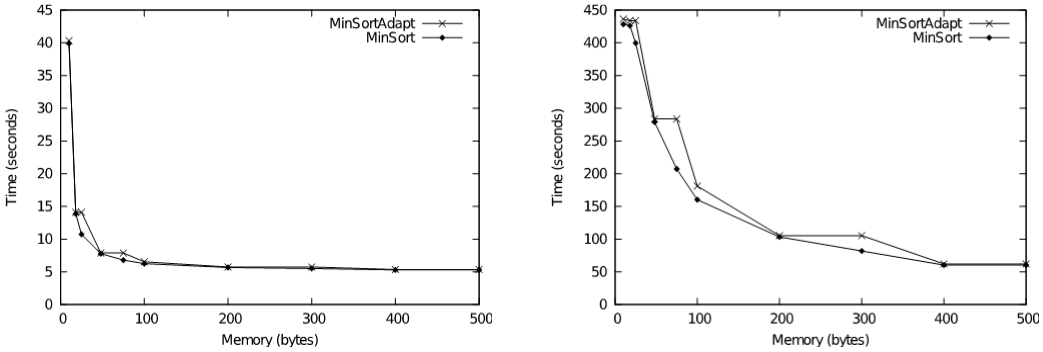

Figure 13. Adapting to Memory Limits (Time). Right-side: Random Data, Left-side: Real Data

7. DISTINCT SORT KEY VALUES

This section examines the effect of increasing the average number of distinct sort key values per region (D_R) on the sorting algorithms discussed in previous sections. Two datasets were generated with different D_R values. The record size is 16 bytes and the sort key is a 2 byte integer. All experiments have 1600 bytes of available memory. Figure 14 shows the execution time and I/O of the algorithms on a dataset with D_R = 8. Figure 15 shows results with D_R = 32. The runtime of *Flash MinSort* is longer as D_R increases because more I/O is performed. The runtime on a dataset containing 2000 tuples increases from 2.15 seconds with D_R = 8 to 6.6 seconds with D_R = 32. *Flash MinSort* is faster than *heap sort* even for dataset sizes where it performs more I/O. This is likely due to the cost of maintaining the heap by copying a large amount of data in memory.

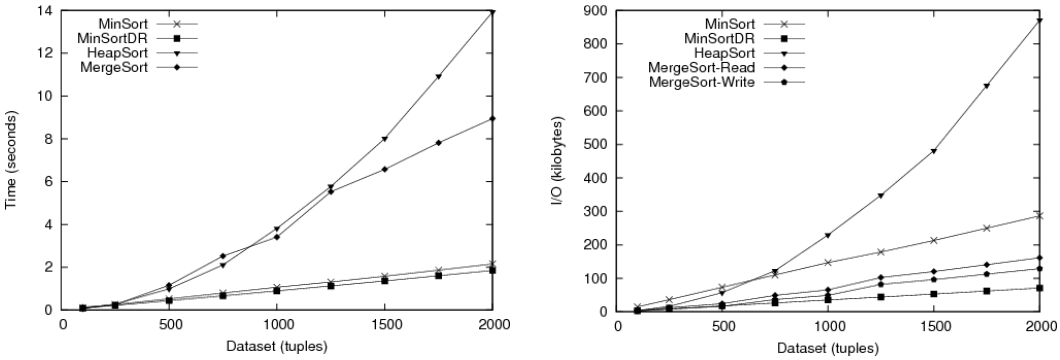

Figure 14. Sorting with Eight Distinct Values per Page (D_R = 8)

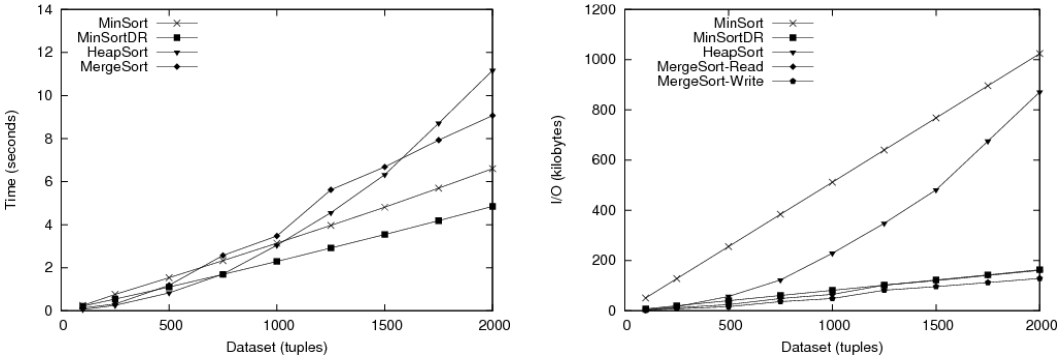

Figure 15. Sorting with 32 Distinct Values per Page (D_R = 32)

7.1 Discussion

Flash MinSort outperforms *one key sort* and *heap sort* on memory constrained embedded devices. This performance advantage is due to its ability to use low-cost random I/Os in flash memory to examine only relevant pages. The performance of the algorithm is especially good when sorting datasets that have few distinct values and exhibit clustering. Even when sorting a random dataset, there is still a performance advantage because no expensive write passes are performed. For flash storage that allows direct reads, *MinSortDR* is even faster and does not have the in-memory page buffer overhead of the other algorithms. *MinSortDR* would see even larger performance gains if the sort key was smaller relative to the size of a tuple. Given a larger tuple size, it would perform fewer small reads relative to the overall size of the dataset.

Flash MinSort is a generalization of *one key sort* as both function the same if there is only one region. The difference is that *Flash MinSort* is able to use additional memory to divide the table into smaller regions and reduce the amount of I/O performed. The primary factor in the performance of both algorithms is the number of distinct values sorted. A smaller number of distinct values results in better performance.

As the memory available increases, *heap sort* becomes more competitive. It is not the absolute memory size that is important, but the ratio of memory available versus sort data size. For small sensor nodes, both the absolute memory and relative amount of memory is very limited. In the sensor node architecture used for testing, *heap sort* can potentially outperform *Flash MinSort* when the input dataset is less than ten pages in size. The reason for this limitation is that we can buffer at most four pages (2KB) of data in memory. If the dataset is larger than ten pages, the number of sequential read passes and execution time increases significantly.

External merge sort has good performance, but it is only competitive with *Flash MinSort* when it is supplied with additional memory to generate larger initial sorted runs. Since *external merge sort* requires a minimum of three pages (1,536B) of memory, it is unsuitable for many low-cost embedded applications. When three pages of memory, *external merge sort* takes up to 4.5 times longer than *Flash MinSort* to sort a typical dataset collected by wireless sensor nodes. *External merge sort* performs fewer byte I/Os from flash, but the write-to-read ratio of a typical flash memory chip contributes to the performance difference. Further, the amount of flash memory consumed is three times the size of the relation. This extra storage requirement includes the original relation, the sorted runs being merged, and the sorted runs being produced in the current pass. If *external merge sort* is used on the table storing sensor readings, the maximum input table is 1/3 of the maximum flash memory size and only one sort algorithm can run at a time. Further, whenever writes are introduced the system must deal with wear levelling. For applications whose primary function is environmental monitoring and data collection, dealing with the additional space required and wear levelling significantly complicates the design.

The ability to adapt to the size of the dataset at runtime allows *Flash MinSort* to be used in applications when the size of the input to sort is not known in advance. The adaptive version has minimal overhead, can sort the dataset in-place when it is small enough to fit into memory, and even has a performance advantage in certain situations.

8. CONCLUSIONS

An efficient sort operator greatly increases the local data processing capability of embedded devices. *Flash MinSort* uses very little memory and is executable on the most computationally-constrained devices. Compared to existing external sort algorithms for these devices, it reduces the runtime and number of I/O operations required to sort a dataset. Future work includes implementing an entire tiny database specifically for these sensor nodes and modifying the existing irrigation sensor application to use the database.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of NSERC that helped fund this research.

REFERENCES

[1] Cossentine, T. & Lawrence, R., (2010) "Fast sorting on flash memory sensor nodes", *IDEAS 2010*, pp. 105-113.

[2] Madden, S., (2004) "Intel Lab Data", Retrieved at: http://db.csail.mit.edu/labdata/labdata.html.

[3] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E., (2002) "A Survey on Sensor Networks", *IEEE Communications,* Vol. 40, No. 8, pp. 102-114.

[4] Buratti, C., Conti, A., Dardari, D., Verdone, R., (2009) "An Overview on Wireless Sensor Networks Technology and Evolution", *Sensors,* Vol. 9, No. 9, pp. 6869-6896.

[5] Culler, D., Estrin, D., Srivastava, M., (2004), "Guest Editors' Introduction: Overview of Sensor Networks", *Computer,* Vol. 37, pp. 41-49.

[6] Vieira, M., Coelho, C., Da Silva, D., Da Mata, J., (2003) "Survey on Wireless Sensor Network Devices", *Emerging Technologies and Factory Automation*, pp. 537-544.

[7] Atmel (2010), Atmel Flash AT45DB161D Flash Data Sheet, Retrieved at: http://www.atmel.com/dyn/resources/prod_documents/doc3500.pdf.

[8] Bouganim, L., Jónsson, B., Bonnet, P., (2009) "uFLIP: Understanding Flash IO Patterns", *CIDR*.

[9] Franklin, M. J., Hellerstein, J. M., Madden, S., (2007) "Thinking Big About Tiny Databases", *IEEE Data Engineering Bulletin,* Vol. 30, No. 3, pp. 37-48.

[10] Anciaux, N., Bouganim, L., Pucheral, P., (2003) "Memory requirements for Query Execution in Highly Constrained Devices", *VLDB*, pp. 694-705.

[11] Andreou, P., Spanos, O., Zeinalipou-Yazti, D., Samaras, G., Chrysanthis, P., (2009) "FSort: External Sorting on Flash-based Sensor Devices", *DMSN 2009 Data Management for Sensor Networks,* pp. 1-6.

[12] Park, H. & Shim, K. (2009) "FAST: Flash-aware external sorting for mobile database systems", *Journal of Systems and Software,* Vol. 82, No. 8, pp. 1298-1312.

[13] Vitter, J. S. (2001) "External memory algorithms and data structures: Dealing with massive data", *ACM Computing Surveys,* Vol. 33, No. 2, pp. 209-271.

[14] Wang, X. & Cherniack, M. (2003) "Avoiding ordering and grouping in query processing", *VLDB*, pp. 826-837.

[15] Fazazckerley, S. & Lawrence, R. (2010) "Reducing turfgrass water consumption using sensor nodes and an adaptive irrigation controller", *IEEE Sensors Applications Symposium*, pp. 90-94.