Fast Sorting on Flash Memory Sensor Nodes

Tyler Cossentine
Department of Computer Science
University of British Columbia Okanagan
Kelowna, BC, Canada
tcossentine@gmail.com

Ramon Lawrence
Department of Computer Science
University of British Columbia Okanagan
Kelowna, BC, Canada
ramon.lawrence@ubc.ca

ABSTRACT

Sensor nodes are being used in numerous domains for data collection and analysis. The ability to perform on device data processing increases the functionality and lifetime of a network as it avoids network transmission. Previous work has developed algorithms for sorting on sensor nodes with flash memory. These algorithms favour reads over writes due to the asymmetric costs. However, previous algorithms have not exploited the ability to perform random reads at the same cost as sequential reads. In this paper, we propose a new algorithm called *Flash MinSort* that uses random reads to rapidly sort in flash memory using a small amount of memory. The algorithm works especially well for sensor data which is often temporally clustered. Experimental results on random and real sensor data show that *Flash MinSort* is two to ten times faster than previous approaches for small memory sizes where external merge sort is not executable.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases;
H.2.2 [**Physical Design**]: Access Methods;
H.2.4 [**Systems**]: Query Processing

Keywords

sorting, sensor node, flash memory, query processing

1. INTRODUCTION

Sensor networks [6] are widely used for environmental monitoring, industrial automation, data collection, and event detection and tracking. Sensor nodes are primarily limited by battery life and component costs, so a sensor node has limited computational and memory resources. Most sensor nodes have a very small amount of main memory (1 to 40KB), most of which is used for networking and other tasks and not available for data processing. Data storage is achieved using flash memory. Sorting is widely used in query processing for ordering output, joins, grouping, and aggregation. The ability to sort allows for more powerful query processing on the node, which saves communication time and energy, as nodes can perform computations locally rather than sending data to a collection site. Reducing communication improves battery life and the overall lifetime and robustness of the sensor network.

The contribution of this paper is an efficient flash sorting algorithm that significantly reduces the amount of time and I/O operations performed over previous algorithms. The algorithm favours reads over writes due to the asymmetric costs of writing and reading in flash memory. To avoid writes, previous algorithms would scan the entire table multiple times even though most of the data scanned was of no value. The performance improvement is achieved by maintaining data statistics that allow the algorithm to efficiently scan only tuples of interest. Another major benefit is that the algorithm performs exceptionally well in the case of sorted or near-sorted data and exploits the temporal clustering common in sensor network data. When compared with existing algorithms under memory constrained conditions, *Flash MinSort* is twice as fast for random data and ten times faster for real sensor data. Another contribution is an analytical and experimental evaluation of existing algorithms and an analysis of the types of inputs and memory sizes where each algorithm is superior.

The organization of this paper is as follows. In Section 2 is background on sensor nodes, their components, and key issues with developing algorithms for use with flash memory. This section also summarizes the current algorithms for flash-based sorting. Our new *Flash MinSort* algorithm is presented in Section 3. Section 4 contains a theoretical analysis for *Flash MinSort* and other existing algorithms. The experimental evaluation is in Section 5, and the paper closes with conclusions and future work.

2. BACKGROUND

Sensor networks are used in military, environmental, agricultural and industrial applications [1, 6]. A wireless sensor node consists of an embedded processor, power management, sensing system, and a communications link [7, 11]. 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. Wireless sensor devices can have both internal and external sensor systems. Communications between devices is accomplished using radio frequency (*RF*) modulation techniques [6, 11]. Devices may 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.

Device energy usage is a major research challenge [11]. Nodes typically use an 8-bit processor [1, 11] with a small amount of local memory. Accessing flash memory requires transferring data from the flash device over the bus to the processor. Devices are chosen to be low in cost and energy efficient. For data processing applications, only a small amount of the RAM is usable as the rest is dedicated to sensing and network communication functions. Thus, it is critical to develop algorithms that function efficiently with minimal memory usage.

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 [4] support direct byte reads. A page must be erased before it can be written. The erase unit is typically a block of pages.

Flash memory has unique performance properties. On a sensor node, flash memory is directly accessed from a flash chip. Unlike flash devices [5] that provide a block-level interface and have layers of software for block mapping, wear levelling, and error correction, flash memory chips provide basic hardware functionality. Flash memory allows random reads at approximately the same rate as sequential reads. This is considerably different than hard drives used in other data processing applications. Further, there are asymmetric read and write costs with writes being between 1.5 to 100 times more costly depending on the flash memory chip. 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.

The Atmel AT45DB161D [4] 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 one of the internal buffers.

For sorting a table of records, we will 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 $N_T = \lfloor \frac{L_P}{L_T} \rfloor$, or for simplicity $N_T = \frac{T}{P}$. 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. The algorithm presented in this paper will group 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 Figure 1.

Notation	**Definition**
T	number of tuples in table to sort
P	number of pages in table
N_T	number of tuples per page = T/P
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

Figure 1: Sorting Parameters

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 [2, 3, 10] 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 [12].

The most memory efficient algorithm is the *one key scan* algorithm [2] 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*. It is straightforward to see that this 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 $2L_K$.

The *heap sort* algorithm, called FAST(1) [10], 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 $\lceil \frac{T}{N} \rceil$ regardless of the data. One complication is handling duplicate sort key values. The solution [10] 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 $O(logN)$. One page buffer is used as an input buffer. Despite using more memory, this algorithm may be slower than *one key scan* if the number of distinct sort key values is small.

The standard sort-merge algorithm 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, $\lfloor \frac{M}{L_P} \rfloor - 1$ 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 [10] is a *generalized external merge sort* algorithm that builds on FAST(1) to allow multiple passes and larger data files. 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 $\lfloor \frac{M}{L_P} \rfloor$ pages in a single pass like *external merge sort*, the sublist size can be up to Q pages, where $\lfloor \frac{M}{L_P} \rfloor \leq Q \leq P$. 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 allow it to merge Q sublists in each pass instead of $\lfloor \frac{M}{L_P} \rfloor - 1$. The optimal selection of Q requires searching all possible values.

FSort [3] 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.

An algorithm performance summary is in Figure 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. Note that 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 5.

3. FLASH MINSORT ALGORITHM

The core idea of the *Flash MinSort* algorithm is that random reads can replace sequential scans for retrieving only the tuples required. All previous algorithms assumed the input was scanned sequentially and often read records that were not necessary. Since random I/Os have the same cost as sequential I/Os, this can be exploited to avoid reading unnecessary data. The algorithm works by building a simple dynamic index over the table that stores the minimum value in each region. This minimum value is updated as the sort progresses. Instead of scanning the entire table, only the region with the minimum value is searched. The minimum value in the region is updated and the process repeats.

The term *region* is used to refer to a sequence of one or more pages. In the ideal case, a region consists of only one page. The amount of space required to store the minimum value per page is $L_K * P$ which may be larger than M. Thus, we group adjacent pages into regions and compute the maximum number of regions that there is sufficient memory to be able to store one key value for. The algorithm is adaptable to the amount of memory available, and the minimum amount of memory is $4L_K + L_I$ for two regions. With two regions, only two minimum values must be stored. For each sort key value, we only search the regions where it occurs.

The algorithm maintains a *current* minimum value and *next* minimum value. It also uses an integer (or perhaps smaller) counter to represent the location of the next minimum tuple in a region (*nextIdx*). This variable is used to improve efficiency and handle duplicates. If the algorithm encounters another tuple in the region with a key value the same as *current*, it stops and sets *nextIdx* to that location. That tuple will be the next one output. This guarantees that duplicates are output in the order they appear.

As the algorithm is scanning a region for records with a key value equal to *current*, it is simultaneously updating the minimum value associated with that region. The algorithm continues scanning the region from the position of the current tuple (*loc*). It will stop the scan immediately if it finds a tuple with *current* value. Otherwise, it continues to the end of the region updating the value of *next* as smaller values are seen. The algorithm does not need to scan the start of the region up to *loc* as either this was done during the search for *current* (starting from record 0 in the region) or the records before *loc* were scanned and *next* updated when searching for a previous value of *current* in the region. Since a region is always scanned from the beginning, all tuples are considered when determining the next minimum.

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. The complete algorithm is given in Figure 3.

The performance of Flash MinSort is very good, especially 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

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

Figure 2: Existing Sort Algorithm Performance Formulas

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.

One key optimization is that the algorithm does not need to operate on entire tuples and pages to perform the sort. Only when the tuple is output does an entire tuple need to be read. Otherwise, only the sort key attribute must be read to update the minimum. This has the potential to dramatically reduce the amount of I/O performed and the amount of data sent over the bus from the flash memory to the processor. If the flash chip supports direct byte addressable reads, searching for the minimum key in the region does not require reading entire pages assuming the sort key offsets can be calculated. This calculation is trivial as fixed-size records are often used for sensor nodes. 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 but on average the number of region scans is $R * D_R$.

In Figure 4 is 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. This allows MinSort to store a minimum value for each page. In the diagram, a value with a rectangle around it is output as a minimum value in that iteration, and a value with a circle around it was scanned in order to determine the next minimum in the page. The first six iterations are shown. On each iteration a tuple is output in the correct sorted order. The algorithm first scans the entire table and calculates the minimum value for each page. These values are stored in an array in memory. The first iteration searches the array for the minimum value and finds the value 1 in page 1 is the minimum. It then loads the page and searches for a 1 which it finds in the first tuple. This tuple is output. Next, it updates the minimum value by continuing to search the page. The minimum in the page is initially 9, but then it finds another 1 at record 4. It sets *nextIdx* to location 4 in the page and leaves the block minimum unchanged. On iteration 2, it is not necessary to search the minimum array as $nextIdx = 4$. Page 1 is currently in memory and the algorithm jumps directly to record 4 to output it. The minimum in page 1 is set to 9 without scanning the rest of the page as $next = 9$ from the scans searching for the two 1 values. Iteration 3 requires searching the minimum array to find the minimum is 1 in page 7. The search for this record finds it in location 2. While updating the minimum in the block, it finds another 1 and sets $nextIdx = 4$. This allows it to directly output this record in iteration 4 and update the minimum in the block to 2. Iteration 5 requires searching the minimum array and finding 1 is the minimum in page 8. A record with key 1 is found as the first record in page 8 and output. Updating the minimum in the page stops immediately at the next record which is also a 1. The second record in the page is output in iteration 6. The algorithm continues until all records are output.

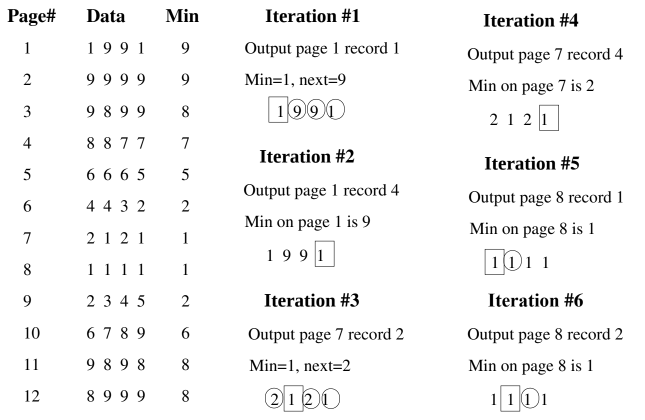
Page#	Data	Min
1	1 9 9 1	9
2	9 9 9 9	9
3	9 8 9 9	8
4	8 8 7 7	7
5	6 6 6 5	5
6	4 4 3 2	2
7	2 1 2 1	1
8	1 1 1 1	1
9	2 3 4 5	2
10	6 7 8 9	6
11	9 8 9 8	8
12	8 9 9 9	8

Figure 4: MinSort Example

In this example, the number of page reads is 39, tuple reads is 148, and bytes read is 1444. In comparison, the one key sort would perform 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* would be able to only store 3 records in the heap (ignoring all other overheads of the algorithm) and perform 16 passes for a total of 192 page I/Os, 768 tuple I/Os, and 15,360 bytes. One key sort reads 3 times more pages and over 6 times more bytes than *Flash MinSort* and heap sort reads almost 5 times more pages and over 10 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.

```
procedure FlashMinScan()
    $numPagesPerRegion = \lceil \frac{numPage * L_K}{M - 2 * L_K - L_I} \rceil$
    $numRegions = \lceil \frac{numPages}{numPagesPerRegion} \rceil$
    Scan input and update minInRegion array with minimum value in each region
    $nextIdx = 0$;
    while (data to sort)
        if (nextIdx == 0)
            $i$ = location of minimum value in minInRegion array
            $current = minInRegion[i]$;
            $next$ = maxvalue;
        end if

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

        // Update minimum in region
        if (sorted region)
            $current = r.key$ of next record or maxvalue if none
            nextIdx is 0 if next key does not equal current, or next index otherwise
        else
            $nextIdx = 0$;
            for each record $r$ in region $i$ after loc
                if ($r.key == current$)
                    nextIdx = location of record in region
                    break;
                end if
                if ($r.key > current$ AND $r.key < next$)
                    $next = r.key$;
                end if
            end for
            if ($nextIdx == 0$)
                $minInRegion[i] = next$;
            end if
        end if
    end while
end procedure
```

Figure 3: MinSort Algorithm

3.1 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 [2] 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 [13]. 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 *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, so if the original number of distinct sort keys was D then the number actually sorted is $\sigma * D$. Let α represent the reduction in input size from both operators, so if the original table was of size $T * L_T$ the sorted relation size is $\alpha * T * L_T$. The cost formulas in the following section can be modified by multiplying by σ or 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.

4. THEORETICAL ANALYSIS

In this section, we compare the theoretical performance of *Flash MinSort* with existing algorithms to determine classes of inputs where each algorithm dominates.

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 $D+1 < \frac{T}{\lfloor \frac{M-L_P}{L_T+L_I} \rfloor}$ or for simplicity $D < \frac{T*L_T}{M}$. 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.

Flash MinSort dominates *heap sort* when $1+D_R < \frac{T*L_T}{M}$. *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.

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 most likely will not use a large amount of the available memory. 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 1.5 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.

This analysis considered only complete page I/Os, if the flash chip allows direct memory reads, the performance of *Flash MinSort* is even better.

5. EXPERIMENTAL EVALUATION

The experimental evaluation compares *Flash MinSort* with *one key sort*, *heap sort*, and the standard *external merge sort* algorithm. The sensor device used for the experiment has an Atmel Mega644p processor with speed of 8 MHz, a local memory size of 4KB, and a Atmel AT45DB161D [4] chip with capacity of 2 MB. The maximum amount of memory available for data processing is 2000 bytes with the rest of the memory used for node functions and program stack space. This node design was used for field measurement of soil moisture for use with an automated irrigation controller [8]. The system was designed to collect sensor readings every minute, store them locally, and periodically send them back to the controller. For this evaluation, we took continuous portions of some of the 3 months of the live data and loaded it onto the device for testing. We also generated ordered and random data sets. The record size is 16 bytes, and the sort key is a 2 byte integer which represents the soil moisture reading computed from a 10-bit analog-to-digital converter. The flash page size is 512 bytes. All numbers are the average of 3 runs.

5.1 Raw Device Performance

The raw device performance was tested to benchmark the read and write times. 50,000 records were used for a file size. The Atmel chip provides three different read mechanisms: direct byte array reads to RAM, direct page reads to RAM, and read page to buffer then transfer at the byte-level 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 31, 23, and 33 seconds respectively. Thus, buffering has no 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 (records). We tested three types of key scans. The first reads only the keys directly from flash, the second reads a page into a flash buffer then reads the keys from the buffer, and the third reads an entire page into RAM and only copies out the necessary key values. For 16 byte records (32 records per page), the time to perform a key scan using the three methods was 21, 22, and 30 seconds respectively. We use the first method that reads keys directly from flash as 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.

In terms of write performance, the flash memory requires an on-chip buffer to be filled then wrote out as a page. You can either fill the on-chip buffer a tuple at a time or a page at time. For writing 50,000 records, buffering a tuple at a time takes 58 seconds and buffering a page at a time takes 37 seconds. Compared to the read performance, when transferring tuples from on-chip buffers the write-to-read ratio is 1.8, and the ratio is 1.6 when transferring pages from on-chip buffers. This flash memory is especially designed for fast write performance, so the ratios are extremely low which benefits *external merge sort*. The raw write performance of the flash chip is masked by the slow processor and bus speeds. A doubled bus speed increases the ratios to 2.8 and 2.2 respectively.

5.2 Real Data

The real data set consists of 10,000 records (160KB) collected by the sensor network system during Summer 2009 [8]. Since the data is collecting soil moisture, the data variation is in a confined range, and the data follows a fairly regular pattern. There are 43 distinct data values. The performance of the algorithms by time and I/Os is shown in Figures 5 and 6. Each chart shows the algorithm performance as the memory available increases. Note that we do not display the data for *heap sort* as its times are an order of magnitude larger. For a memory size of 30 bytes, it buffers only one tuple and performs 10,000 passes for a time of 24,403 seconds. For memory size of 100 bytes (5 tuples), the time is 9800 seconds and for 1200 bytes (74 tuples), the time is 1671 seconds. *Heap sort* would never be competitive on this data set as the maximum memory available is 2KB.

One key sort is good due to the limited number of distinct values, which is common in sensor applications due to the use of 10-bit analog-to-digital converters. *One key sort* performance does not change as more memory is added.

There are two implementations of *Flash MinSort*: basic *MinSort* transfers a complete page from the flash to RAM while *MinSortDR* performs direct byte reads from the flash. All algorithms except *MinSortDR* buffer one page in memory which is not counted in the memory cost. *MinSortDR* performs fewer I/Os than regular *MinSort* and is faster for small memory sizes. Further, it did not need to buffer a whole page for scanning to improve its performance, so its effective memory usage is even smaller. The advantage decreases as more memory is available as *MinSort* will typically use most of the records on every page it retrieves as the regions are smaller. *MinSortDR* would be even better if the record sizes were larger. With 32 records per page, the 32 separate calls to memory for 2 bytes at a time are not greatly faster than one call to read a whole 512 byte page.

The smallest amount of memory that *external merge sort* can run with is 1536 bytes (3 pages). Its runtime is 143 seconds, and it performs 7 write passes and 8 read passes. *Flash MinSort* runs faster than *external merge sort* with considerably less memory and no writes. As memory increases, *external merge sort* would become more competitive since *Flash MinSort* no longer benefits from additional memory once a region equals one page. However, for small memory sizes, *external merge sort* is not possible for on-node data processing due to its large minimal memory requirement.

5.3 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 (duplicates allowed). The number of distinct values was 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 Figures 7 and 8. 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 155 seconds for the random data set as the sorting during initial run generation took slightly more time.

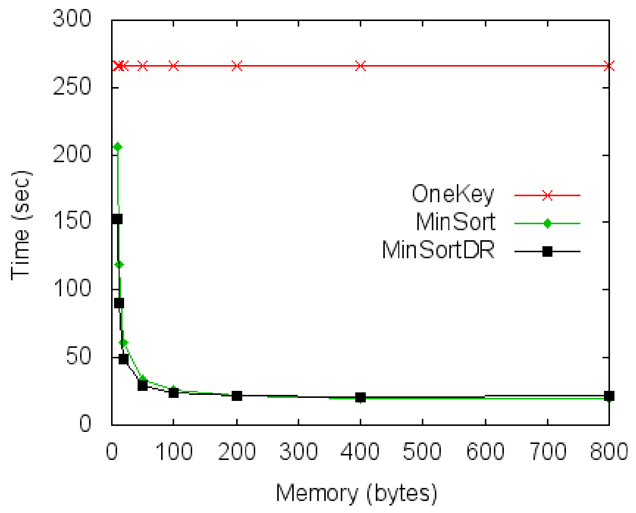

Figure 5: Sorting 10,000 Real Data Records (Time)

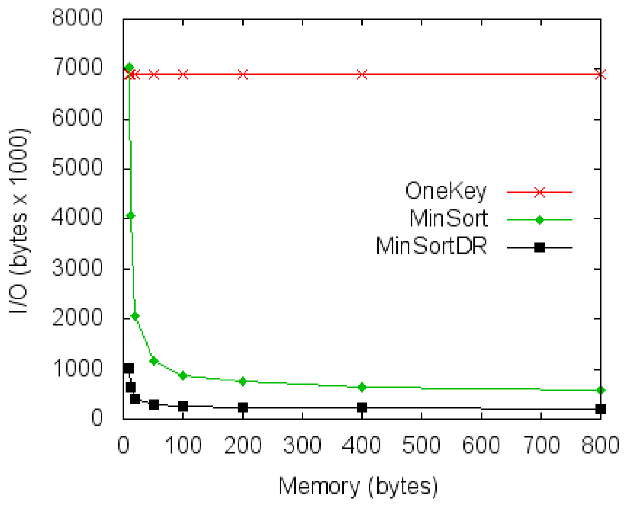

Figure 6: Sorting 10,000 Real Data Records (Bytes)

5.4 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 Figures 9 and 10. As expected, *Flash MinSort* dominates based on its ability to adapt to sorted inputs. The *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*

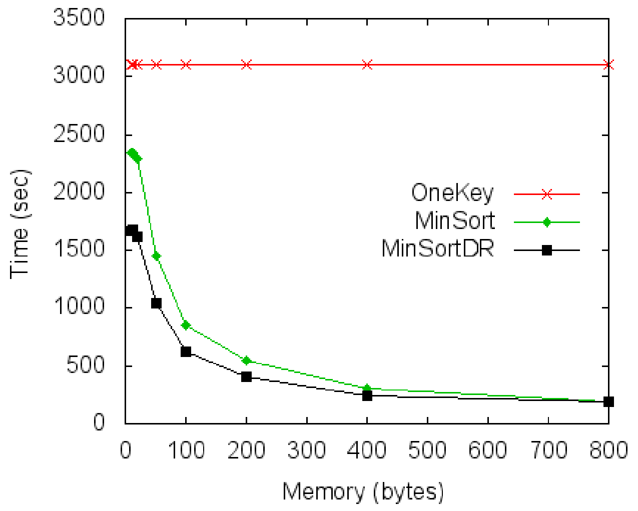

Figure 7: Sorting 10,000 Random Records (Time)

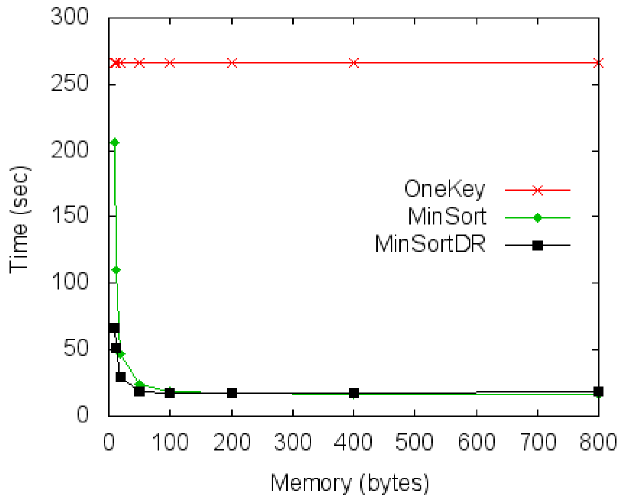

Figure 9: Sorting 10,000 Ordered Records (Time)

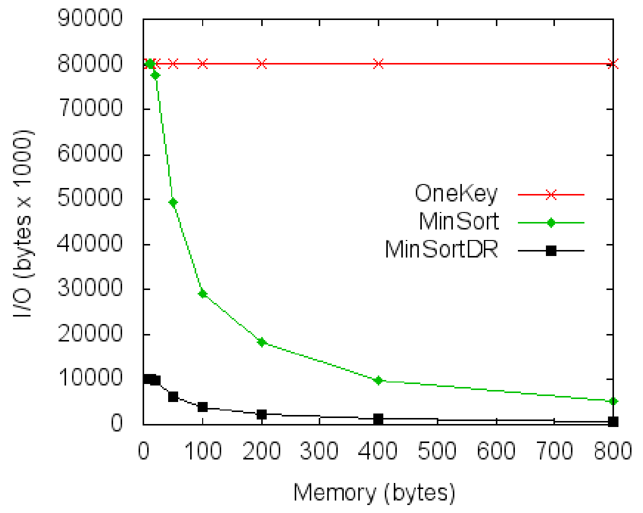

Figure 8: Sorting 10,000 Random Records (Bytes)

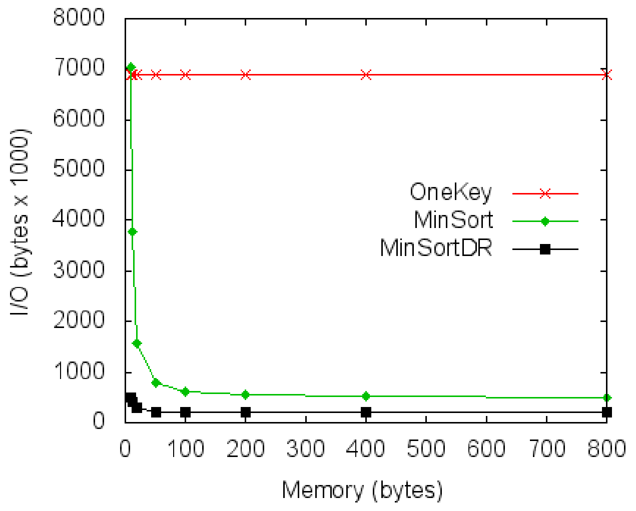

Figure 10: Sorting 10,000 Ordered Records (Bytes)

key sort is still competitive while *heap sort* (not shown) is not for these memory sizes. *External merge sort* took 137 seconds. *Heap sort* had the same times as the other two experiments.

5.5 Discussion

MinSort dominates *one key sort* and *heap sort* with or without using direct byte reads from the device. This is due to its ability to use random I/Os to jump to the relevant tuples. With sensor data that exhibits temporal clustering, its performance is especially good. Even the worst case of random data still shows a significant benefit. Larger record sizes would benefit *MinSortDR* as it would perform fewer small requests to retrieve keys as there are fewer tuples on a page. *MinSortDR* was faster despite not having the scan page buffer overhead used by all other algorithms.

MinSort is a generalization of *one key sort* as both function the same if there is only one region. The difference is that *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 our sensor node architecture, *heap sort* would only be superior for sorting data less than 10 pages in size (since we can at most buffer 4 pages in memory (2K)). Otherwise, the number of passes and execution time increases significantly.

External merge sort has good performance but only with a very large minimum memory consumption of 1536 bytes which makes it not applicable for many small sensor applications. *Flash MinSort* is faster for the practical memory sizes that are executable on the test device. Interestingly, for the real data set, *external merge sort* will never be faster regardless of the amount of memory provided (assuming at least two passes are required). The effective number of distinct values per region D_R is about 2 and the effective write-to-read ratio is 1.6 resulting in an effective number of passes of 3 for *Flash MinSort* and 2 read passes and 1 write pass = 3.6 for *external merge sort*. For the random data set, the effective D_R is just under 32, so the effective number of passes is 33 for *Flash MinSort*. If *external merge sort* uses the minimum 3 pages of memory, the weighted number of passes for *external merge sort* is 7 write + 8 read = 19.2 which explains why it outperforms for the random data set. *External merge sort* benefits significantly from the very competitive write times on the device compared to reads.

Another issue with *external merge sort* and any algorithm that performs writes in passes is that the amount of flash memory consumed is three times the size of the input table: the original source input table (the raw data), the sorted runs in the previous pass, 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 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 and performance.

6. CONCLUSIONS

This work has presented *Flash MinSort*, a sorting algorithm especially designed for sorting in flash memory on small sensor nodes. Its performance dramatically improves over previous algorithms by exploiting random I/Os to only retrieve tuples of interest. Future work includes implementing an entire tiny database specifically for these sensor nodes which will use this sorting algorithm, and modifying the existing irrigation sensor application to use the database.

7. REFERENCES

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications*, 40(8):102–114, Aug 2002.

[2] N. Anciaux, L. Bouganim, and P. Pucheral. Memory Requirements for Query Execution in Highly Constrained Devices. In *VLDB*, pages 694–705, 2003.

[3] P. Andreou, O. Spanos, D. Zeinalipour-Yazti, G. Samaras, and P. K. Chrysanthis. FSort: external sorting on flash-based sensor devices. In *DMSN'09: Data Management for Sensor Networks*, pages 1–6, 2009.

[4] Atmel. Atmel Flash AT45DB161D Data Sheet, 2010.

[5] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[6] C. Buratti, A. Conti, D. Dardari, and R. Verdone. An Overview on Wireless Sensor Networks Technology and Evolution. *Sensors*, 9(9):6869–6896, 2009.

[7] D. Culler, D. Estrin, and M. Srivastava. Guest Editors' Introduction: Overview of Sensor Networks. *Computer*, 37:41–49, 2004.

[8] S. Fazackerley and R. Lawrence. Reducing turfgrass water consumption using sensor nodes and an adaptive irrigation controller. In *IEEE Sensors Applications Symposium (SAS)*, pages 90 –94, 2010.

[9] M. J. Franklin, J. M. Hellerstein, and S. Madden. Thinking Big About Tiny Databases. *IEEE Data Eng. Bull.*, 30(3):37–48, 2007.

[10] H. Park and K. Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8):1298 – 1312, 2009.

[11] M. Vieira, J. Coelho, C.N., J. da Silva, D.C., and J. da Mata. Survey on Wireless Sensor Network Devices. In *Emerging Technologies and Factory Automation*, pages 537–544, 2003.

[12] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[13] X. Wang and M. Cherniack. Avoiding ordering and grouping in query processing. In *VLDB*, pages 826–837, 2003.