An Efficient External Sorting Algorithm for Flash Memory Embedded Devices

by

Tyler Andrew Cossentine

B.Sc., The University of British Columbia, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The College of Graduate Studies

(Interdisciplinary Studies)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

January, 2012

Abstract

Many embedded systems 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. Sorting is a fundamental algorithm used frequently in databases. Flash memory has unique performance characteristics. Page writes to flash memory are much more expensive than reads. In addition, random reads from flash memory can be performed at nearly the same speed as sequential reads. External sorting can be optimized for flash memory embedded devices by favoring page reads over writes. This thesis describes the *Flash MinSort* algorithm that, given little memory, takes advantage of fast random reads and generates an index at runtime to sort a dataset. The algorithm adapts its performance to the memory available and performs best for data that is temporally 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.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded systems are devices that perform a few simple functions. Most embedded systems, such as sensor networks, smart cards and certain handheld devices, are severely 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 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. The contribution of this thesis is an efficient external sorting algorithm designed specifically for embedded systems with flash memory storage, called *Flash MinSort*.

Depending on the properties of the dataset being sorted, *Flash MinSort* has the potential to significantly reduce the amount of time and disk access operations when compared to existing algorithms designed for low-memory devices. The performance improvement is achieved by generating and maintaining an index on the sort key attribute 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 traditional 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 temporally clustered. Temporally clustered data can be found in many embedded systems applications, such as environmental monitoring with sensor networks. The algorithm also performs well when given sorted or near-sorted data. The *Flash MinSort* algorithm is capable of adapting to the amount of memory available and the size of the dataset. It gracefully degrades from a one-pass sort as the size of the dataset grows larger than main memory.

An initial version of *Flash MinSort* was presented in [12]. This thesis 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 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 from disk. This thesis contains a detailed analysis and experimental evaluation of *Flash MinSort* when applied to the area of sensor networks. Applications of *Flash MinSort* to traditional database management systems with solid state drives are also explored.

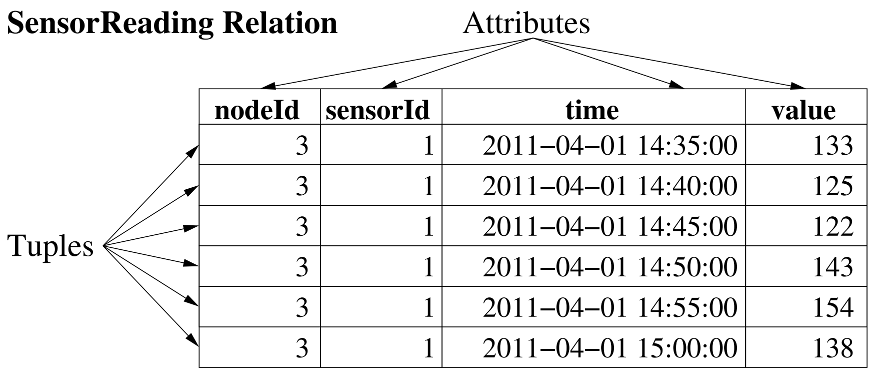
The organization of this thesis is as follows. Chapter 2 focuses on background information on relational databases, embedded devices, flash memory, sensor networks, and sorting algorithms. Chapter 3 contains a detailed description of the *Flash MinSort* algorithm and Chapter 4 provides an analysis of performance. Chapter 5 shows experimental results of *Flash MinSort* with comparison to existing algorithms. Chapter 6 discusses the potential applications of the algorithm and future work.

Chapter 2

Background

2.1 Relational Databases

A database management system (DBMS) is a piece of software that manages and provides an interface to a relational database [18]. A relational database consists of a collection of relations that store data. A relation is a table that contains rows, known as tuples. Each column in a relation is called an attribute. In a given tuple, each attribute contains either a value or NULL. The cardinality of a relation is the number of rows it contains. Figure 2.1 shows the different parts of a relation.



nodeId	sensorId	time	value
3	1	2011–04–01 14:35:00	133
3	1	2011–04–01 14:40:00	125
3	1	2011–04–01 14:45:00	122
3	1	2011–04–01 14:50:00	143
3	1	2011–04–01 14:55:00	154
3	1	2011–04–01 15:00:00	138

Figure 2.1: Parts of a Relation

2.1.1 Structured Query Language

Structured query language (SQL) is a language commonly used to manage relational databases [18]. A DBMS executes SQL queries on relations and returns the resulting tuples. For example, a query such as **SELECT DISTINCT nodeId FROM SensorReading WHERE value > 130** executed against the SensorReading relation will return tuples containing just a unique *nodeId* attribute where the *value* attribute is greater than 130.

2.1.2 Query Plan

A DBMS parses an SQL query and generates a query plan [18]. A query plan consists of a tree of database operators necessary to generate the requested output. Operators are implemented as iterators that return a tuple at a time and can include scan, select, project, sort, and join. The relation generated by any branch of the query tree can be materialized by maintaining a copy in memory or writing it to external storage. Materialization can be used to reduce memory usage or avoid regenerating the same input relation multiple times.

The scan operator iterates over a dataset on disk, returning a tuple at a time. The selection operator, represented by σ_c, returns an input tuple as output if the boolean expression c evaluates to true. Projection, represented by Π, returns a new tuple that contains only the specified attributes of an input tuple. The sort operator sorts a relation in ascending or descending order on one or more attributes. If the input relation does not fit into memory, the sort operator requires it to be materialized on disk. The join operator, represented by $\bowtie$, returns an output tuple that is a combination of one tuple from each of the input relations if the join predicate evaluates to true. Typical join algorithms include block-based nested loop join, sort-merge join, and hash join.

A database operator that is implemented as an iterator returns a tuple at a time to its parent node in the query tree. A query tree can be right-deep, left-deep or bushy. A right-deep query tree always has the base relation as the left join argument. Iterator functionality typically includes the init, next, close and rescan methods. After the query plan has been constructed, these functions are called on the root node of the tree to retrieve results. An example of a query plan executed on the SensorReading relation can be found in Figure 2.2.

2.1.3 Indexing

An index is a data structure that allows for fast lookup of tuples in a relation [18]. Indexes are important for databases since many queries only examine a small subset of a relation. An entry in an index contains a search key value for an attribute of the relation and a pointer to the location of the tuple that has that value. Indexes take up less space on disk because each entry contains a single attribute of the relation. Depending on the implementation, an index might allow tuples to be retrieved in sorted order.

An index can be dense or sparse. A dense index contains an entry for

```
SELECT value, time
FROM SensorReading
WHERE time > '2011-03-01'
ORDER BY value
```

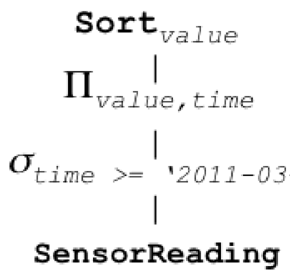


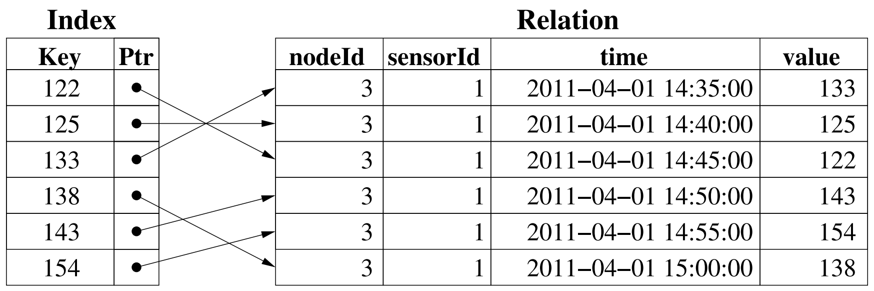
Figure 2.2: Query Plan

every tuple in the relation. Figure 2.3 contains an example of a dense, single-level index on the SensorReading relation. A sparse index contains entries for only some of the tuples in a relation. An entry in a sparse index typically points to a single page of tuples on disk. Figure 2.4 contains an example of a sparse, single-level index on the SensorReading relation. A sparse index is a more space efficient choice for a relation that is stored in an ordered format on disk because it only stores one entry per page of tuples. A dense index has a search advantage over a sparse index by allowing a query to determine if a search key exists without accessing the entire relation on disk.

A multi-level index has more than one index level on a relation. With the exception of the first level, each level of a multi-level index points to the index below it. Multi-level indexes have a performance advantage because fewer pages of lower-level indexes need to be read during a search. The first level of an index can be dense or sparse, but all higher levels of an index must be sparse. The B+-tree data structure is commonly used for indexes in a modern DBMS.

2.1.4 Sorting

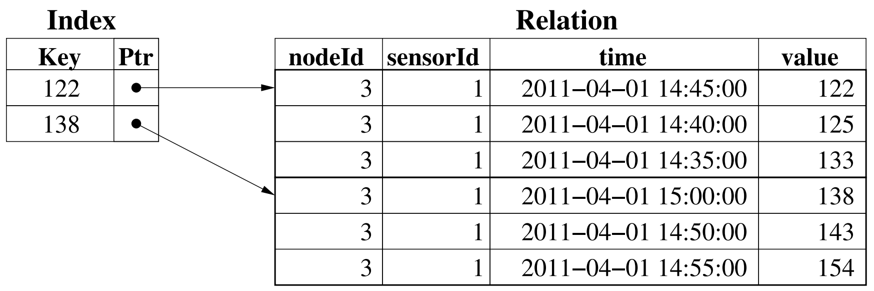
Sorting is a fundamental algorithm used in databases for ordering results, joins, grouping, and aggregation. A detailed analysis of sorting algorithms can be found in [23]. An internal sorting algorithm sorts a dataset using main memory only. Quicksort, heapsort, selection sort, and insertion sort



Key	Ptr
122	●
125	●
133	●
138	●
143	●
154	●

nodeId	sensorId	time	value
3	1	2011–04–01 14:35:00	133
3	1	2011–04–01 14:40:00	125
3	1	2011–04–01 14:45:00	122
3	1	2011–04–01 14:50:00	143
3	1	2011–04–01 14:55:00	154
3	1	2011–04–01 15:00:00	138

Figure 2.3: Dense Index



Key	Ptr
122	●
138	●

nodeId	sensorId	time	value
3	1	2011–04–01 14:45:00	122
3	1	2011–04–01 14:40:00	125
3	1	2011–04–01 14:35:00	133
3	1	2011–04–01 15:00:00	138
3	1	2011–04–01 14:50:00	143
3	1	2011–04–01 14:55:00	154

Figure 2.4: Sparse Index

are examples of internal sorting algorithms. When there is not a sufficient amount of main memory to sort an entire dataset, it is necessary to make use of external memory, such as a hard drive. General external sorting algorithms were surveyed in [37].

A common sorting algorithm used by a DBMS to sort large relations is external merge-sort [18]. External merge-sort loads chunks of a relation R into memory M blocks at a time, where M is the size of available memory in disk blocks. Each chunk of R is sorted in-place using a sort algorithm such as quicksort and the sorted runs are written out to disk. If the number of sorted runs is less than or equal to M, they can be merged in one read pass by loading them into memory a block at a time. More than one read pass will be necessary if the number of sorted sublists on disk is greater than M. Figure 2.5 shows an example of sorting the first column of the relation using external merge-sort. In this example, there is enough available memory to hold three disk blocks (M = 3) and each tuple in the initial relation represents an entire block. After the first pass, there are four sorted sublists stored on disk. After two merge passes, there is a single sorted list stored on disk.

External distribution sorts have the potential to perform well when sorting data in external memory and the range of values is known. External bucket sort partitions the dataset into buckets on disk. Each bucket contains a subset of the range of values in the input data. A page of memory is required as a cache for each bucket. Once the page becomes full, the values are written to the bucket on disk. This is much more efficient than writing a tuple at a time to each bucket. After each pass, the buckets are recursively sorted with further partitioning. This process continues until the relation is in sorted order on disk. External radix sort is also a distribution sort that is commonly used to sort data located in external memory.

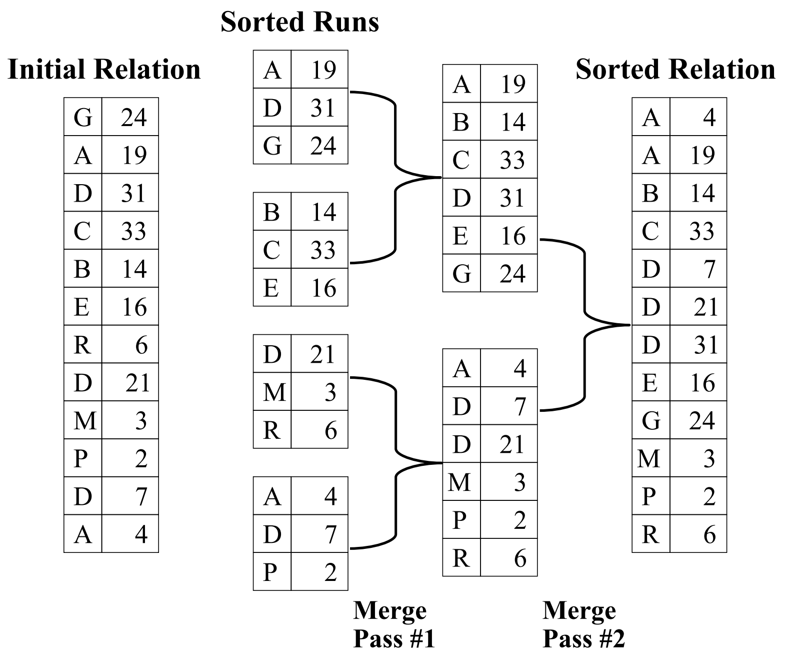


Figure 2.5: External Merge Sort

2.2 Flash Memory

Electrically erasable programmable read-only memory (EEPROM) is a type of non-volatile, byte-erasable memory used in a wide range of devices. Flash memory is a specific type of EEPROM that is available in higher capacities and is erasable in large blocks. There are two types of flash memory in common use, NOR and NAND, that differ in their underlying memory cell technology [31]. NOR flash is not as dense or energy-efficient as NAND, but provides high-speed, random byte-level reads of data [28]. Flash memory is used in a wide range of areas, such as solid state drives (SSD) and embedded systems. Flash memory is available in either surface mount or removable formats, such as Secure Digital (SD) and MultiMediaCard (MMC). Removable flash devices have internal controllers and have higher idle current and energy consumption [28]. Surface mount flash chips can have a parallel or serial interface. Serial flash chips are required for sensor nodes that do not have a large number of general purpose I/O pins available. Current flash memory chips enable embedded systems to store several gigabytes of data while using little energy.

Flash memory has unique performance properties. After many erase cycles, a block of pages will not be able to reliably store the values written to it. This is known as wear. Many flash devices, such as the controller on a SSD, implement a wear leveling strategy to ensure that individual pages do not wear out prematurely. This strategy often includes error detection and a logical to physical page mapping that ensures that erase cycles are spread evenly across all blocks. Unlike devices [8] that contain a controller with a block-level interface, a flash memory chip by itself does not provide block mapping, wear leveling, and error correction. A flash memory chip has asymmetric read and write costs, with writes being between 10 to 100 times more costly. Random page reads can be performed at rougly the same speed as sequential reads. This is considerably different than the hard disk drives (HDD) used in most modern computers. A HDD contains rotating magnetic media and has a track seek time and rotational latency penalty associated with random data access [18].

In flash, writes are more costly due to the requirement that a block must be erased before one of the pages it contains is written. Depending on the type of flash memory, a block can consist of one or more pages of data. Many serial flash devices have one or more internal static random access memory (SRAM) buffers to store pages. Although the transfer size between the flash memory and the buffer on a device is in the unit of pages, the transfer between a SRAM buffer and the processor can be done at the byte level.

A typical storage solution used in embedded devices is the surface-mount Atmel DataFlash AT45DB family of serial NOR flash [5]. This is a popular family of devices due to their simple set of commands, low pin-count serial peripheral interface (SPI) bus, and SRAM buffers. These devices have a page-erase architecture and support page-level reads and writes. The internal SRAM buffers are used to read a page of data or support self-contained read-modify-write operations to flash memory without copying a page of data from flash to main memory. This family of devices also supports random byte-level reads to directly stream a page or *any sequential bytes* from the chip to the processor, bypassing all buffering. One chip in this family is the Atmel AT45DB161D [4], which contains 16-megabits (2MB) of flash storage. The chip supports erase at the unit size of a page (512B), block (4KB), sector (128KB), and entire chip (2MB). It has two internal SRAM page buffers (512B).

2.3 Embedded Devices

2.3.1 Smart Cards

Smart cards are used in a wide range of areas such as banking, television, and healthcare. Like most embedded devices, they have very little computational power, memory (4KB SRAM), or data storage (128KB EEPROM). One study covers the application of database techniques to multi-application smart cards [7]. The study shows that common database operators can run on smart cards with very little memory consumption by using extreme right-deep trees to pipeline execution and avoid recomputing operations in the query plan.

2.3.2 Wireless Sensor Networks

Sensor networks are used in military, environmental, agricultural and industrial applications [1, 9]. A typical wireless sensor node contains a low-power embedded processor, memory/storage, radio transceiver, sensors, power management and a power source [13, 24, 36]. Sensor memory consists of a small amount of random access memory (SRAM) and a larger amount of non-volatile memory (EEPROM or flash) for data storage. Sensor nodes can use both internal and external sensor systems to collect data. Sensors measure environmental properties and typical sensing hardware includes temperature sensors, light sensors, humidity sensors, accelerometers and cameras. Communication between sensor nodes is accomplished using radio frequency modulation techniques [9, 36].

Over the past decade, sensor networks have been successfully deployed in a wide range of areas. In [22], a 64 node network designed for structural health monitoring was deployed on the Golden Gate Bridge. Sensor networks have potential in military applications for battlefield monitoring and intrusion detection. In [11], a 144 node network was deployed to track

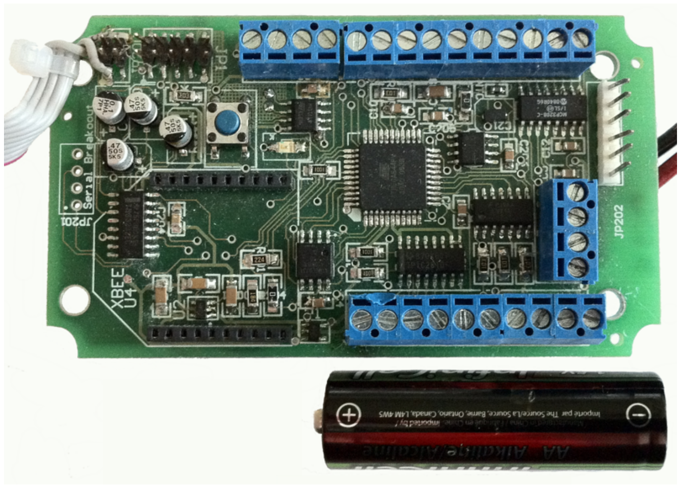


Figure 2.6: Wireless Sensor Node

targets and perform surveillance in a large outdoor field. Sensor networks have also been deployed in environmental and agricultural areas. An early application of sensor networks was habitat monitoring [27]. In [16], a sensor network and adaptive irrigation controller was shown to reduce turfgrass water consumption. In [6], a network of 65 sensor nodes was deployed over a period of six months in a vineyard. Depending on the application, sensor nodes can be statically positioned or mobile. In [39], sensor nodes with ECH_2O capacitance-based sensors and cameras were statically positioned to measure soil moisture and grass coverage in a field. This research also involved attaching sensor nodes with GPS and accelerometer sensors to cattle to collect data on herd behavior.

In basic applications, sensor nodes request data from sensors and transmit it across the network to a common collection point known as the sink [1, 24]. Sensor nodes can act as a source of information or they can help route data towards the sink in multihop networks. Depending on the application, there are several common network topologies and routing protocols that can be implemented [24]. In order to analyze sensor data, track objects, perform spatial querying or perform geographic routing, the absolute or relative position of individual sensor nodes must be known. Sensor node locations can be pre-configured or the network can use one of the many localization algorithms designed for wireless sensor networks [24].

Energy Consumption

In most applications, sensor networks must be able to survive long-term deployments without user intervention. Sensor node replacement is usually not a cost-effective option since “the cost of physically deploying the sensor nodes often outweighs the cost of the nodes themselves” [34]. Labor costs for sensor network deployment are relatively high when the network is composed of thousands of nodes or deployed over rugged and remote terrain. Many applications require sensor networks to remain operational for several months or years [24]. Since sensor nodes are battery powered, they are extremely energy constrained and require low-power components combined with energy efficient algorithms. In [35], the authors provide a power-analysis detailing the energy usage of the individual components of common sensor node platforms. Their research shows that the wireless radio uses a large share of the power used by all components and communication must be managed effectively to increase network lifetime.

There are several areas of research that focus on improvements to hardware and software with the goal of increasing sensor network lifetime. Sensor nodes use low-power components and can employ dynamic voltage scaling (DVS) or dynamic power management (DPM) to reduce energy usage [29]. Another hardware-based area of research involves harvesting environmental energy, specifically by developing sensor platforms that can take advantage of solar cells [32, 34, 39]. In [39], the Fleck-3 sensor node platform was combined with small solar panels for a network of statically positioned sensor nodes. Initial experimental results using the Fleck-3 platform show that solar panels can generate more than enough energy to sustain sensor nodes indefinitely under ideal conditions.

In addition to continuing improvements to the energy efficiency of sensor node hardware, advanced routing and data collection algorithms are necessary to increase sensor network lifetime. To increase lifetime, “most protocol designs in wireless sensor networks are designed explicitly with energy efficiency as the primary goal” [24]. Research on multihop routing protocols shows that transmitting data over many small hops can potentially use less energy than a single long-distance broadcast [1, 36]. Another key area of research is in energy-aware routing protocols [21]. Sensor networks used for data collection may have nodes store and process data locally, or immediately send it back to the sink node. One recent study examined the tradeoff between local data storage and communication [28]. More specifically, the research compared the energy costs of low-power parallel NAND flash memory to radio communication. The study concluded that the energy costs of storage are two orders of magnitude less than for communication. Another study showed that the energy required to transmit 1Kb a distance of 100 meters equals 3 million instruction executions on a processor with 100MIPS/W power [33]. Since the cost of communication is high in wireless sensor networks, local data storage and in-network query processing has the potential to significantly reduce traffic volume and power consumption [1, 33].

Query Processing

Query processing on sensor nodes has been studied, including the issues of local data processing and in-network aggregation [1]. Query processing systems provide a generic, high-level query interface that is not application specific and greatly simplifies the task of retrieving data from the sensor network. Recent research has focused on modeling sensor networks as a distributed database system supporting SQL-like query languages [14, 17, 19]. In a distributed database system, a query is issued to the sensor network at the sink node and propagates throughout the network. Each node processes the query locally and only relevant tuples are returned. A sensor network database reduces data transmission and increases network lifetime.

One sensor network query processing system, called TinyDB, implements a distributed stream database on a sensor network [26]. A stream database enables the user to query a real-time stream of data rather than data located in persistent storage. TinyDB provides the user with a declarative, SQL-like query language and supports selection, projection, aggregation and grouping. Queries are executed on a virtual *sensors* table and can include information such as query duration and sample intervals. The *sensors* table is considered to be virtual because sensor data is not materialized in persistent storage. A typical query would be **SELECT nodeId, temp FROM sensors, SAMPLE PERIOD 1s FOR 10s**. When a sensor node receives a query, it generates an optimal query plan and starts collecting data from its attached sensors. Sensor data is collected once per sample interval and each tuple passes through the query plan. If a tuple of sensor data is an answer to the query, it is transmitted to the sink node over the multihop sensor network using a tree-based routing protocol. TinyDB supports partial aggregation, which further reduces the amount of data that is transmitted [19]. Partial aggregation pushes aggregation computations down the tree and uses values at intermediate nodes to compute the final result. Complex queries that involve the sort and join operators are only possible by materializing a data stream on one of the nodes in the network. Queries allow the user to set the sample rate and time period, desired network lifetime, and can include a stopping event or trigger.

Stream database systems for sensor networks have a limited ability to execute complex queries locally. Another limitation of stream databases is that storing data locally rather than streaming it allows for a much higher sensor sample rate without transmitting any tuples across the network while using the same amount of energy. Stream databases are not appropriate for applications that involve periodic connectivity where the sink is not always reachable. The ability to query data stored locally is very important for applications where the entire dataset is needed for analysis. In these applications, it might be necessary to query more detailed historical data around an event of interest.

In [15], a database system, called StonesDB, was designed to query historical sensor data stored locally at each node. StonesDB is a two-tier database system consisting of sensor nodes at the lower tier and more powerful proxy nodes at the higher tier. Sensor nodes store data locally in flash and send metadata to their associated proxy node. Proxy nodes cache metadata and query results locally. A query is sent to all proxy nodes and each proxy node attempts to answer the query using cached data. If the proxy node cannot answer the query, it is forwarded to the individual sensor nodes. Each sensor node implements a low-power DBMS that processes queries and returns relevant tuples. This system uses several strategies to reduce energy usage by minimizing the number of erase and write operations in flash. One strategy is to physically partition each data stream in flash and generate a separate index for each partition. Another strategy is to write an index once by holding it in main memory until a partition has been filled.

Energy-efficient algorithms for database operators will increase the life of a sensor network. In addition to reducing execution time, energy efficient algorithms typically focus on reducing the number of I/O operations in persistent storage. These algorithms must be able to execute with very limited amounts of memory. There have been customized algorithms developed to improve operator performance on flash embedded devices for sorting (next section), joins and grouping [2].

2.4 External Sorting for Flash Memory Devices

There have been several algorithms proposed for sorting on flash memory [2, 3, 30] 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.

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). We will assume that tuples are stored in an unspanned layout, meaning that they do not cross page boundaries on disk. 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. 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 2.1.

Table 2.1: 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 $= T/P$
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

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*. 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 $2L_K$. One key scan takes advantage of additional memory by buffering tuples of the relation into a *current array*. Each scan of the relation fills the current array with tuples that have the next smallest sort key values. When the current array overflows, tuples with the largest sort key value are discarded to eliminate the need to track the highest tuple number output for a given sort key value. The value of next is set to the largest sort key value of the tuples in current.

The *heap sort* algorithm, called FAST(1) [30], 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 [30] 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 the record number has not already been output. 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.

External merge sort performs one read pass to construct sorted sublists of size M, which are written to external storage. The merge phase buffers one page from each of the sublists and merges the tuples to produce a new sorted sublist. 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 executing sort-merge efficiently requires numerous page buffers in memory. At a minimum, three pages (1,536B) of memory must be available to the operator. Two pages are used to buffer one page from both of the 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 on small sensor nodes. *External merge sort* becomes more practical as M and P increase.

FAST [30] is a *generalized external merge sort* algorithm that uses 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 value of Q is determined at runtime based on the amount of available main memory, input file size and read-to-write ratio of the flash storage device. The algorithm uses a heap data structure to allow it to merge Q sublists in each pass instead of $\lfloor \frac{M}{L_P} \rfloor - 1$.

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.

A summary of algorithm performance is in Table 2.2. 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.

Table 2.2: Existing Sorting Algorithm Performance

Algorithm	**Memory**	**Scans**	**Read Scans**	**Write Scans**	**Stable**
one key	$2 * L_K$	$S = D + 1$	S	0	Yes
FAST(1)	M	$S = \frac{T}{\lfloor \frac{M - L_P}{L_T + L_I} \rfloor}$	S	0	No
merge sort	M	$S = \lceil log_{\lfloor \frac{M}{L_P} \rfloor - 1}(\lceil \frac{P * L_P}{M} \rceil) \rceil$	$S + 1$	S	Yes
FAST	M	$S = \lceil log_Q \lceil \frac{P}{Q} \rceil \rceil$	$S + 1$	S	No
FSort	M	$S = \lceil log_{\lfloor \frac{M}{L_P} \rfloor - 1}(\lceil \frac{P * L_P}{2 * M} \rceil) \rceil$	$S + 1$	S	Yes

2.5 Summary

Embedded devices are used in a wide range of areas and are often resource constrained in terms of energy and processing power. These devices contain a low-power microcontroller, small amount of RAM, and flash memory for persistent data storage. Many applications require data to be stored and processed locally. Some devices, such as smart cards, must provide advanced querying and security functionality. Other battery-powered devices, such as wireless sensor networks, require energy-efficient data collection and processing strategies to be considered cost-effective.

Existing research on data management and query processing strategies for embedded devices tends to focus on the application of traditional database algorithms and techniques. Sorting is a key operation used in databases for ordering results, joins, grouping, and aggregation. Current approaches to sorting relations on embedded devices, even those designed specifically for flash memory, do not adapt to data distributions, clustering or handle very small memory efficiently.

Chapter 3

Flash MinSort

3.1 Algorithm Description

The core idea of the *Flash MinSort* algorithm is that random page reads can replace sequential scans to reduce the amount of disk 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. In a given pass, these algorithms read pages and tuples that are not needed.

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 $4L_K + L_I$ 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 3.1 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.

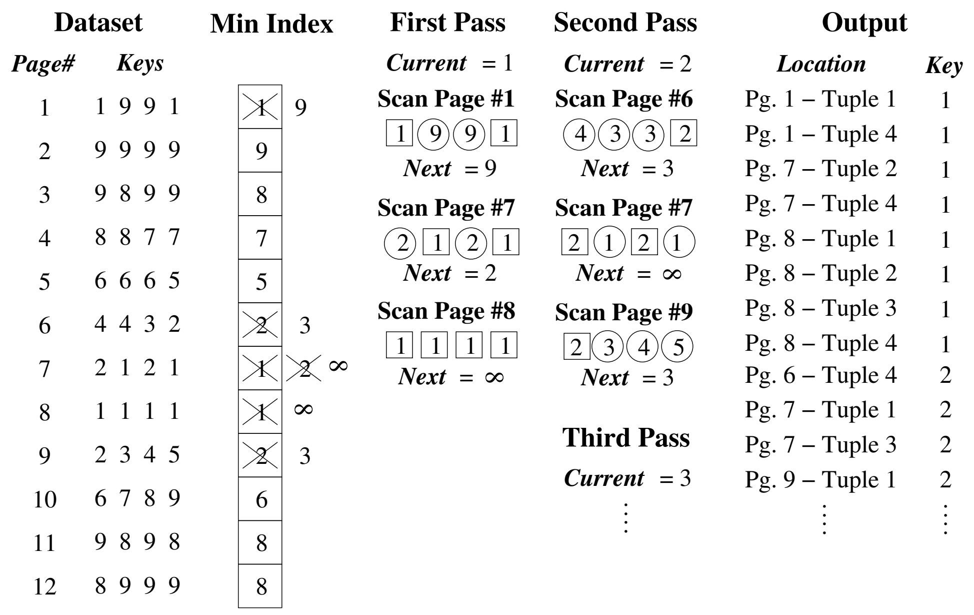


Figure 3.1: Flash MinSort Example

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 3.2. 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()
    $numPagesPerRegion = \lceil \frac{numPages*L_K}{M-2*L_K-L_I} \rceil$
    $numRegions = \lceil \frac{numPages}{numPagesPerRegion} \rceil$
    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 $\neq$ $current$, 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 3.2: Flash MinSort Algorithm

3.2 Adapting to Memory Limits

The base version of the algorithm in Figure 3.2 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 and size of the 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 3.1. 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 3.3 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.

Dataset

Page#	***Keys***
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
6	4 4 3 2
7	2 1 2 1
8	1 1 1 1
9	2 3 4 5
10	6 7 8 9
11	9 8 9 8
12	8 9 9 9

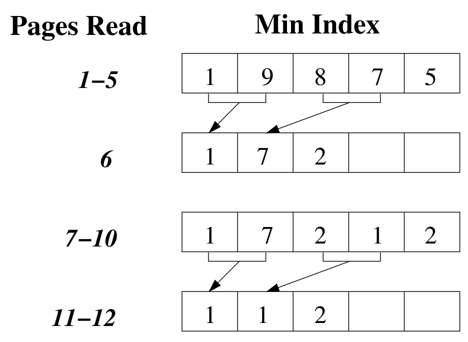


Figure 3.3: Dynamic Region Size Example

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 pages by merging adjacent regions 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 3.4: Dynamic Input Buffering Example

Figure 3.4 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 all of the 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 (its index value is the same as page 8 and it is first in the minimum index). 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 the pages in the cache.

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.

3.3 Exploiting Direct Byte Reads

One possible optimization to the *Flash MinSort* algorithm is that it does not need to read entire pages when scanning a region. 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 3.5 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.

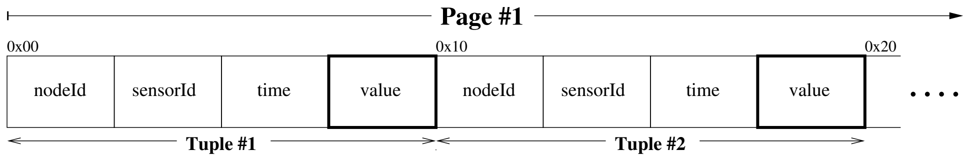


Figure 3.5: Direct Byte Reads

3.4 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.

Chapter 4

Algorithm Analysis

This chapter compares the theoretical performance of *Flash MinSort* with existing algorithms to determine classes of inputs where each algorithm dominates.

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 3.1, 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 $D + 1 < \frac{T}{\left\lfloor \frac{M - L_P}{L_T + L_I} \right\rfloor}$ 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.

Basic *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.

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 $1 + D_R * (1 - \mathit{hitRate})$ 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 $\frac{M-P*L_K}{T*L_T}$ which is approximately $\frac{M}{T*L_T}$ since the space used for the minimum index, $P * L_K$, will typically be small. Thus, adaptive *Flash MinSort* will dominate *heap sort* when $1 + D_R * (\frac{M}{T*L_T}) < \frac{T*L_T}{M}$.

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 $\frac{M-P*L_K}{T*L_T} = \frac{2000-16}{4000} \approx 0.50$. 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 4.1, *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. This results in considerable savings in bytes transferred from the device and bytes transferred from device buffers to the processor over the bus.

4.3 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 [38]. 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, such as Figure 2.2, 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 and projectivity of the plan. Note that subplans with non-materialized joins would be especially costly to re-execute if performing *heap sort*.

Chapter 5

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 [4] 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 [16]. The system was designed to take sensor readings at fixed intervals and periodically send data back to the controller.

The data used for evaluation was collected at one minute intervals over a period of three months. Continuous portions of three months of the live sensor data were loaded onto the device for testing. The algorithms were also run on pre-generated ordered and random data sets. The tuple size is 16 bytes and the sort key attribute is a 2 byte integer. In the real sensor data, this attribute is the soil moisture reading computed from a 10-bit analog-to-digital converter. All results are an average of three runs.

5.1 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.

In terms of write performance, the flash memory requires an on-chip buffer to be filled and then written out to 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 25.03 seconds and buffering a page at a time takes 23.02 seconds. In the case where a singe tuple is transferred at a time, the write-to-read ratio is 4.72. When transferring entire pages, the write-to-read ratio is 6.26. The raw read and write performance of the flash chip is masked by the slow processor and limited bus bandwidth on the sensor node.

5.2 Real Data

The real dataset consists of 100,000 records (1.6MB) collected by a sensor network during Summer 2009 [16]. The schema can be found in Table 5.1. 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 disk I/Os is shown in Figures 5.1 and 5.2. Each chart shows the algorithm performance as the amount of memory increases. 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.

One key sort has better 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 disk 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.

Table 5.1: Soil Moisture Table Schema

Attribute	**Type**	**Width**
year	unsigned byte	1
month	unsigned byte	1
day	unsigned byte	1
hour	unsigned byte	1
minute	unsigned byte	1
second	unsigned byte	1
sensorid	unsigned short	2
nodeid	unsigned short	2
value	unsigned short	2
flow	unsigned short	2
zone	unsigned byte	1
status	unsigned byte	1

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.

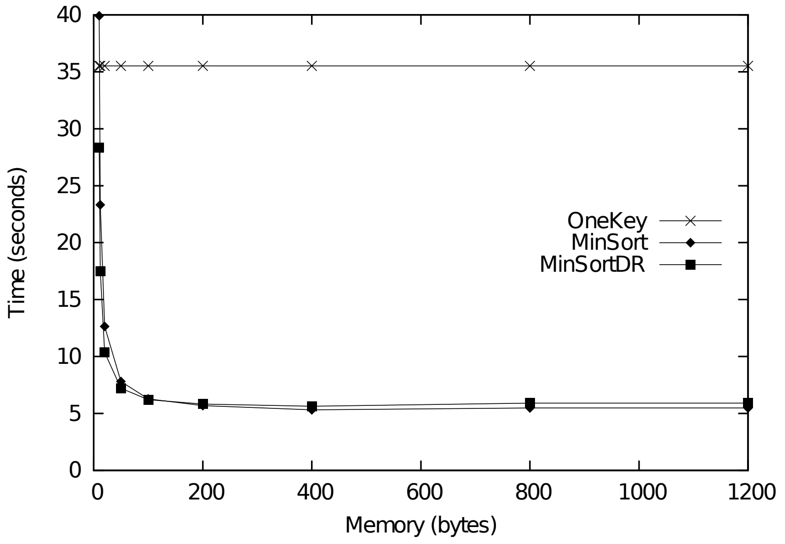


Figure 5.1: Sorting 10,000 Real Data Records (Time)

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. 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 5.3 and 5.4. 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 as the sorting during initial run generation took slightly more time.

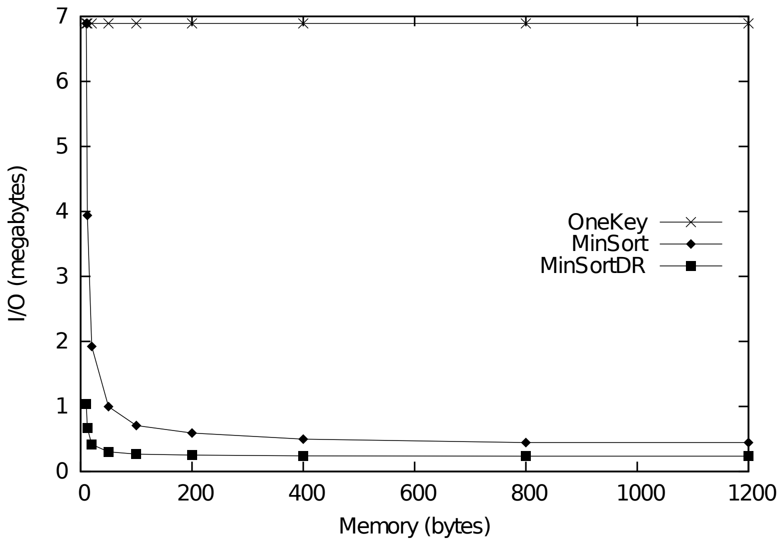


Figure 5.2: Sorting 10,000 Real Data Records (Disk I/O)

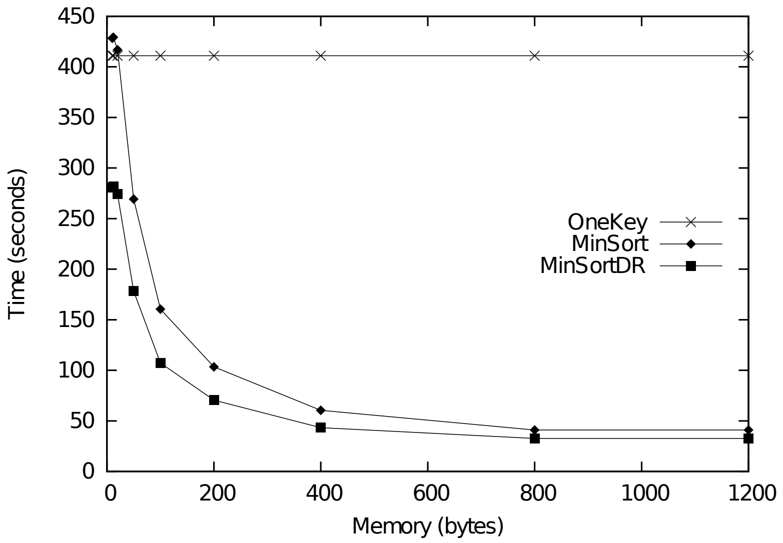


Figure 5.3: Sorting 10,000 Random Records (Time)

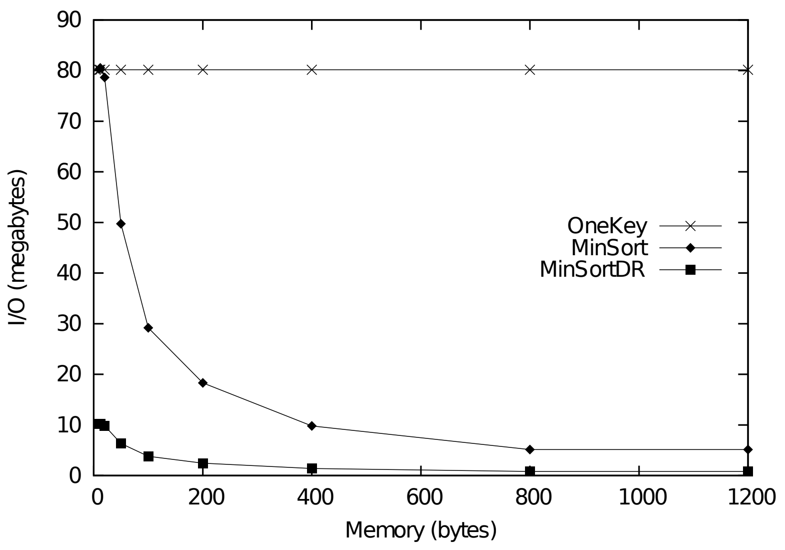


Figure 5.4: Sorting 10,000 Random Records (Disk I/O)

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 5.5 and 5.6. As expected, *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* (not shown) is not for these memory sizes. *Heap sort* has the same execution time as the previous two experiments. *External merge sort* took 75 seconds.

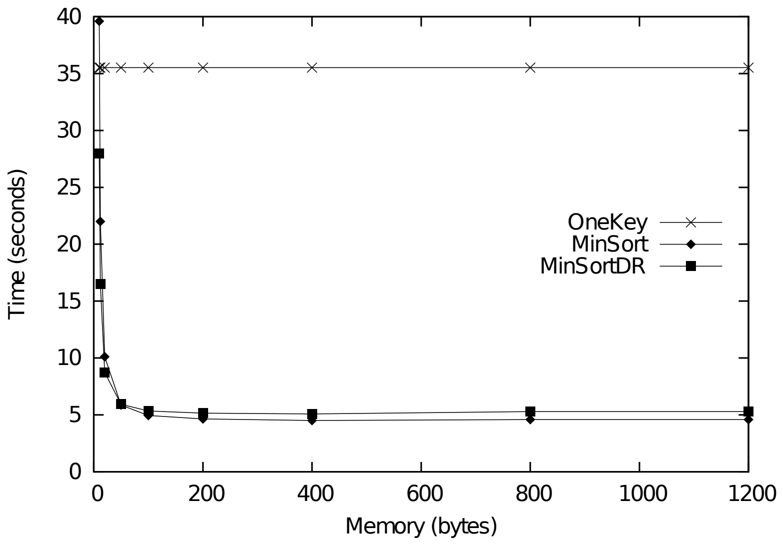


Figure 5.5: Sorting 10,000 Ordered Records (Time)

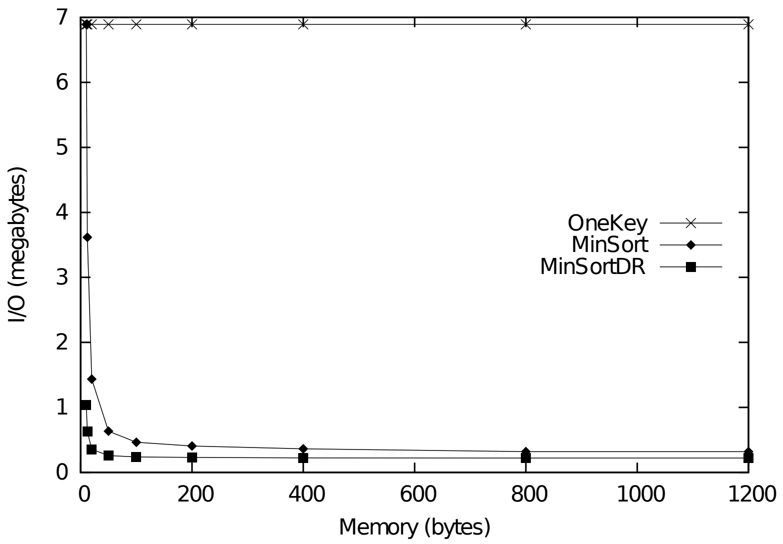


Figure 5.6: Sorting 10,000 Ordered Records (Disk I/O)

5.5 Berkeley Dataset

The Berkeley dataset was collected from a sensor network at the Intel Berkeley Research Lab [25]. This dataset contains 2.3 million records collected by 54 sensor nodes with on-board temperature, light, humidity and voltage sensors. The schema can be found in Table 5.2. Each record is 32 bytes in size and each page of flash contains 16 records.

Table 5.2: Berkeley Table Schema

Attribute	**Type**	**Width**
year	unsigned short	2
month	unsigned short	2
day	unsigned short	2
hour	unsigned short	2
minute	unsigned short	2
second	unsigned short	2
epoch	unsigned short	2
moteid	unsigned short	2
temperature	float	4
humidity	float	4
light	float	4
voltage	float	4

Table 5.3: Berkeley Data Distinct Values

Attribute	**Total Distinct**	**Average Distinct Per Page**
temperature	847	9.03
humidity	467	8.14
light	62	2.90

A 5,000 record subset of this dataset was used to evaluate the sorting algorithms. Information on the number of distinct sort keys and the average number of distinct sort keys per page can be found in Table 5.3. Figures 5.7 and 5.8 show the execution time and disk I/O of the algorithms when sorting the dataset on the light attribute. Given three pages of memory, *external merge sort* took 60.02 seconds to complete. Figures 5.9 and 5.10 repeat these experiments for the humidity attribute. In this case, *external merge sort* took 59.87 seconds to complete. Finally, Figures 5.11 and 5.12 show the results of sorting on the temperature attribute. *External merge sort* took 59.83 seconds to complete.

With the exception of external merge sort, these figures are consistent with the results in previous sections. It is possible that external merge sort is faster for this dataset because there is half the number of records and the initial sort phase doesn't copy as much data in memory.

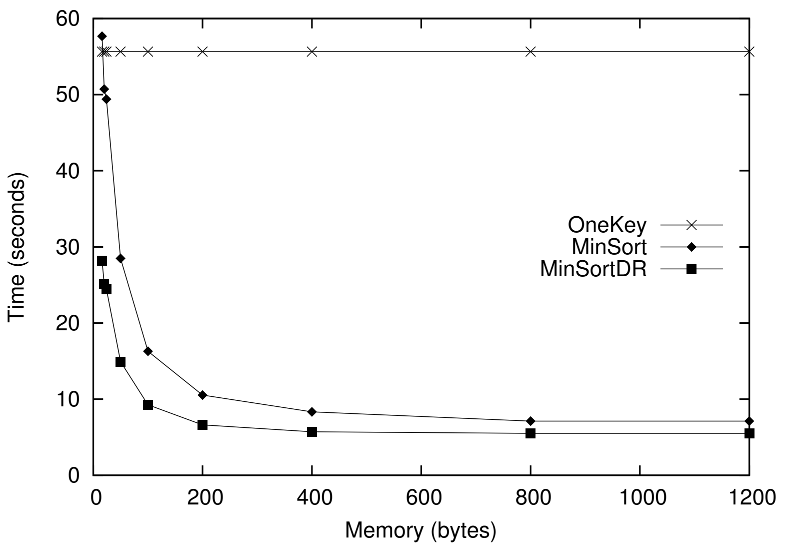


Figure 5.7: Sorting on Light Attribute (Time)

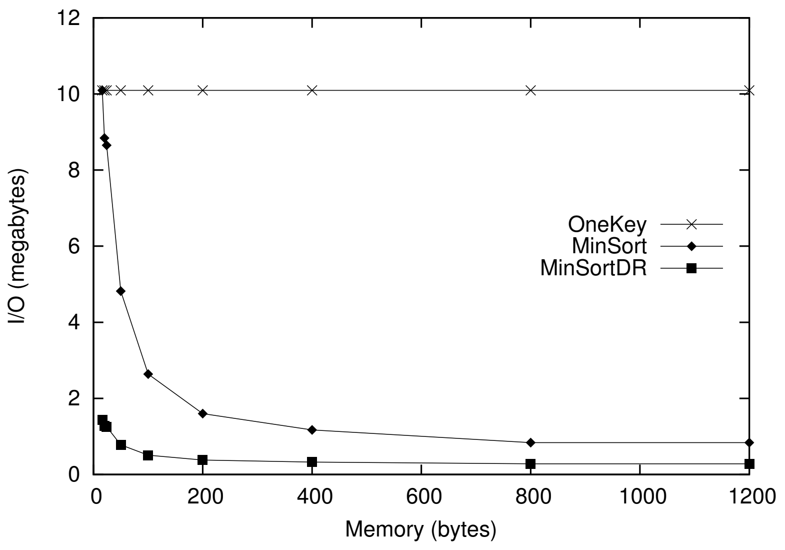


Figure 5.8: Sorting on Light Attribute (Disk I/O)

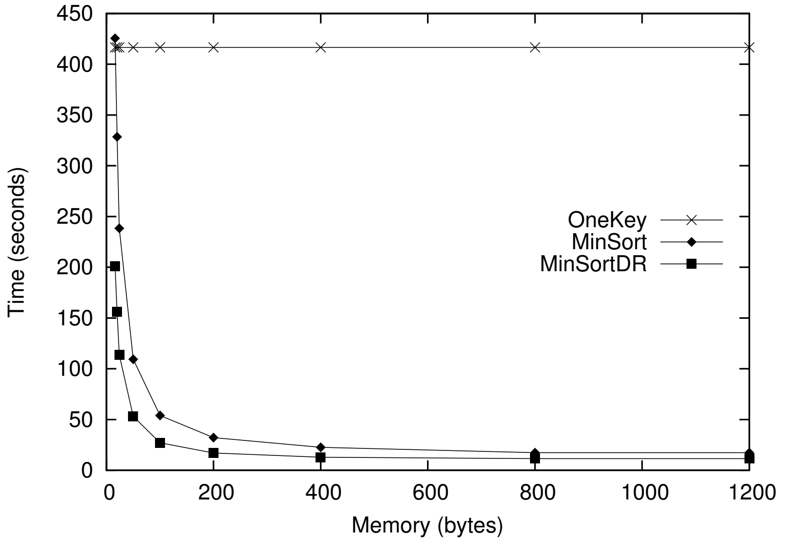


Figure 5.9: Sorting on Humidity Attribute (Time)

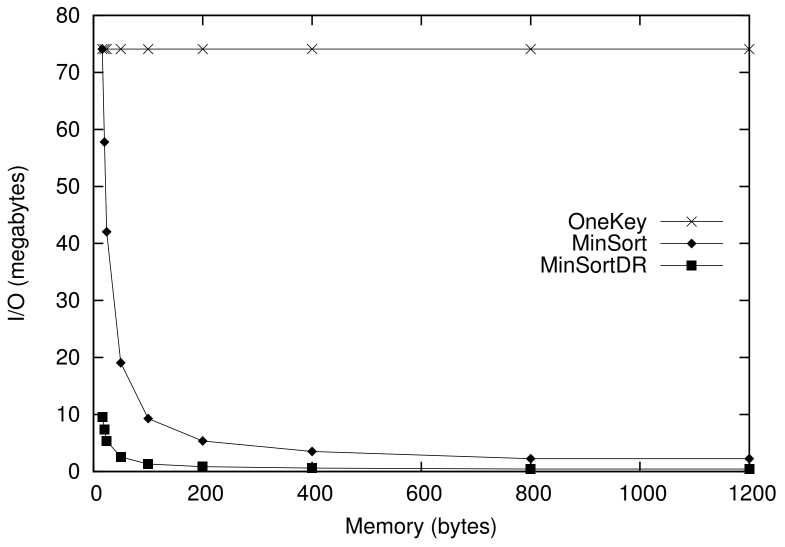


Figure 5.10: Sorting on Humidity Attribute (Disk I/O)

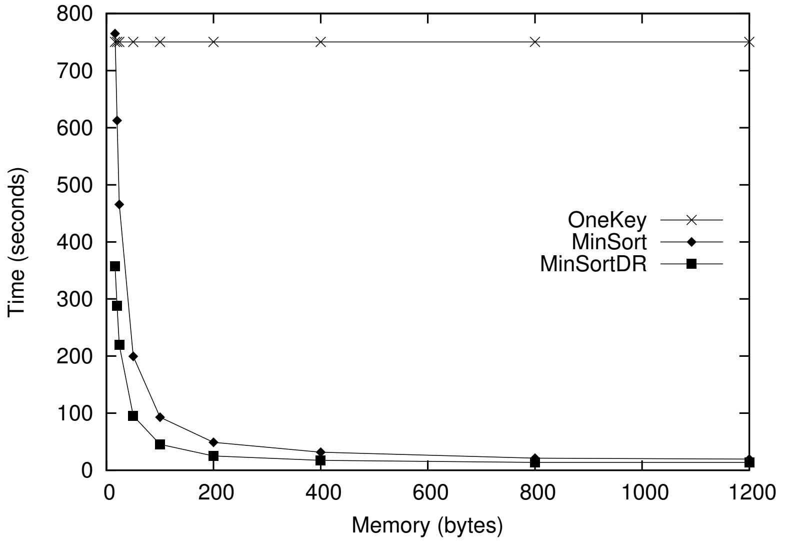


Figure 5.11: Sorting on Temperature Attribute (Time)

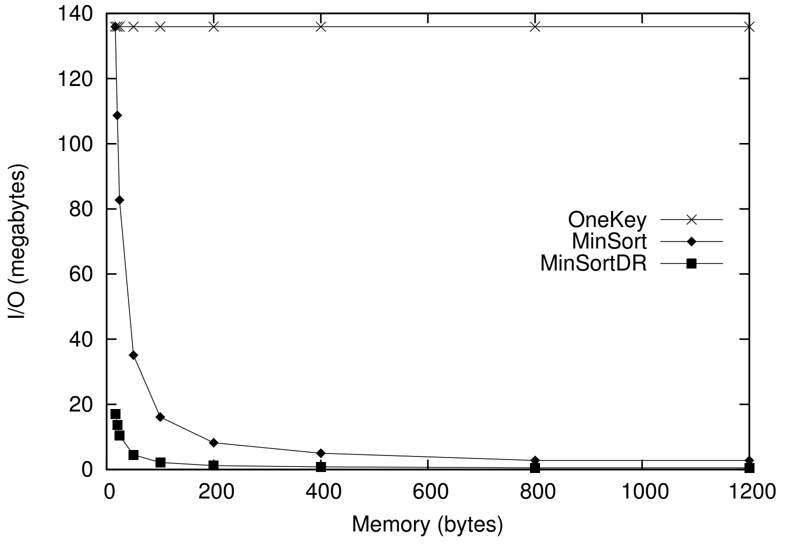


Figure 5.12: Sorting on Temperature Attribute (Disk I/O)

5.6 Adapting to Memory Limits

Figures 5.13 and 5.14 contain a comparison between the adaptive and base versions of the algorithm. It demonstrates sorting a dataset where there is either enough memory to sort a dataset in-place or cache one or more pages. Figure 5.13 shows the real dataset and Figure 5.14 shows the random dataset. In both figures, 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).

In both figures, 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. In the real dataset, the base version of the algorithm performs 18 page reads and the adaptive version performs 14 page reads with 4 cache hits. In the random dataset, the base version of the algorithm performs 177 page reads and the adaptive version performs 71 page reads with 106 cache hits. As the dataset grows larger, the caching performance advantage disappears since the number of cache hits is very small relative to the total number of pages read.

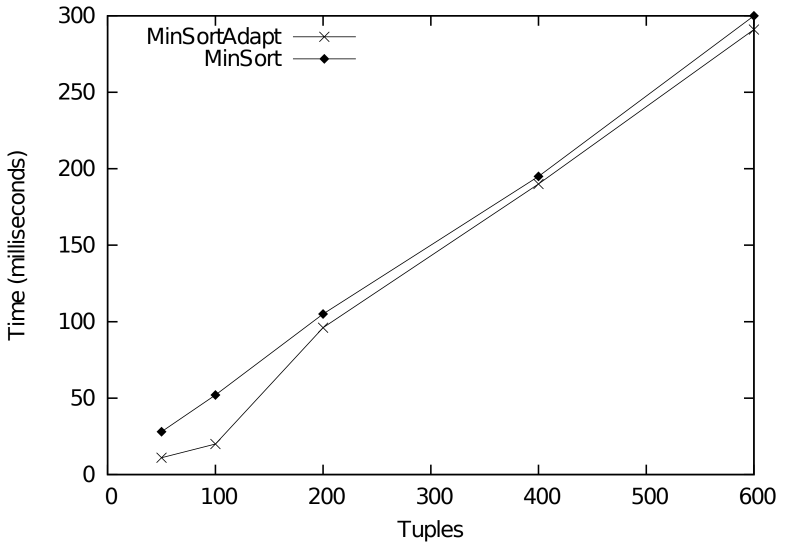


Figure 5.13: In-place Optimization (Real)

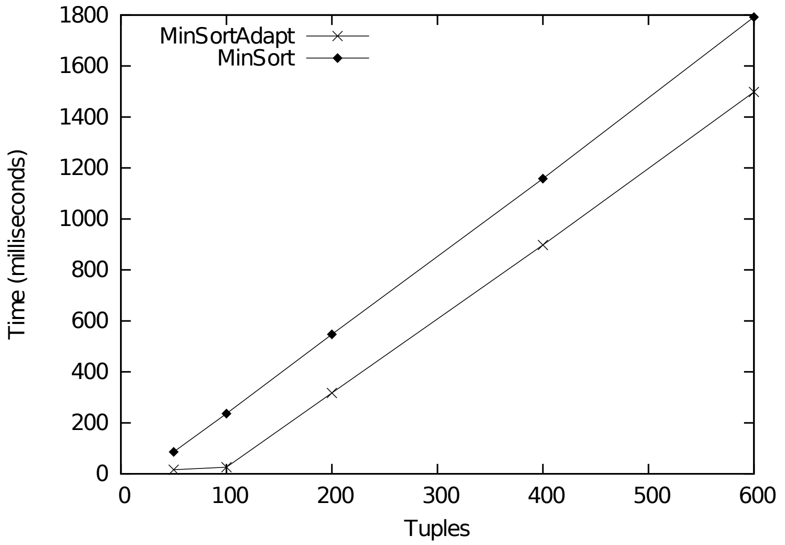


Figure 5.14: In-place Optimization (Random)

Figures 5.15 and 5.16 demonstrate the performance of the two versions of the algorithm with small memory sizes. The dataset in both figures is 160KB (10,000 tuples). Figure 5.15 shows the real dataset and Figure 5.16 shows the random dataset. In both figures, 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.

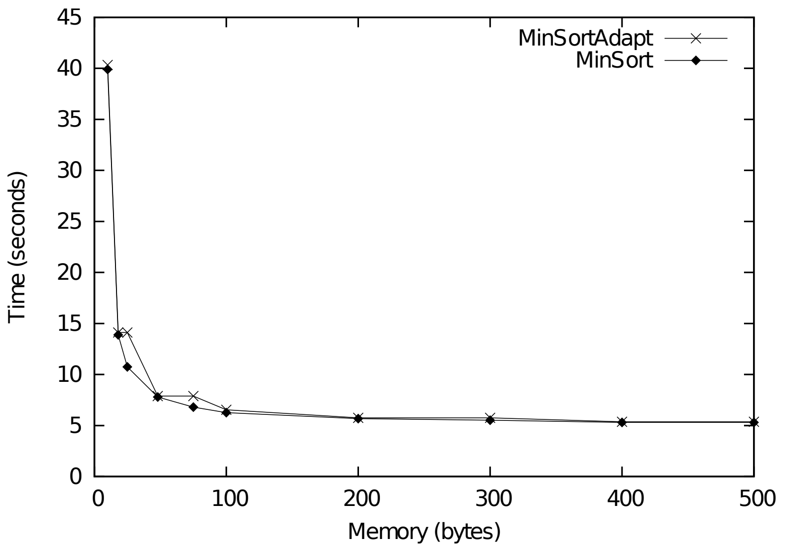


Figure 5.15: Adaptive Algorithm (Real)

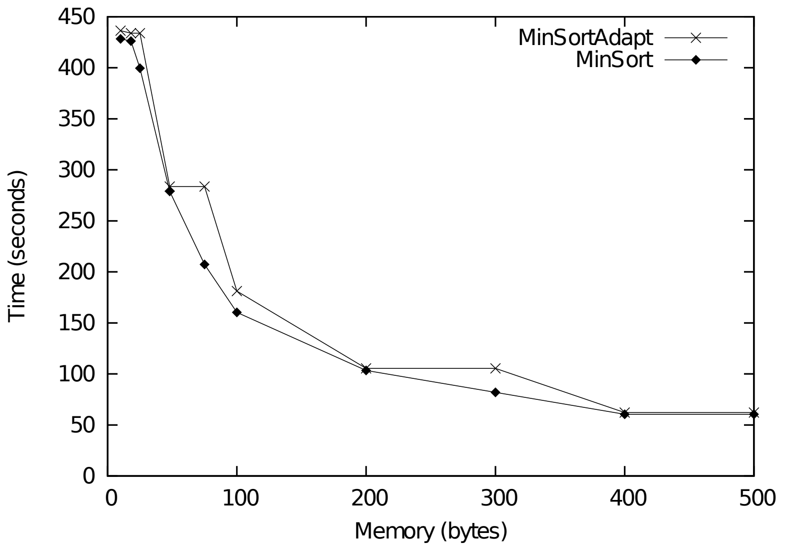


Figure 5.16: Adaptive Algorithm (Random)

5.7 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. Three 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, which is more than enough for *Flash MinSort* to represent each page as a region in the index.

Figures 5.17 and 5.18 show the execution time and disk I/O of the algorithms on a dataset with $D_R = 8$. Figures 5.19 and 5.20 repeat these experiments with $D_R = 16$. Finally, Figures 5.21 and 5.22 show the results with $D_R = 32$. *Flash MinSort* is the only algorithm effected by properties of the dataset other than record size and the number of records. The runtime of *Flash MinSort* is longer as D_R increases because more disk 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 $D_R = 32$.

In the final two sets of figures, Flash MinSort is faster than heap sort even for dataset sizes where it performs more disk I/O. This is likely due to the cost of maintaining the heap by copying a large amount of data in memory.

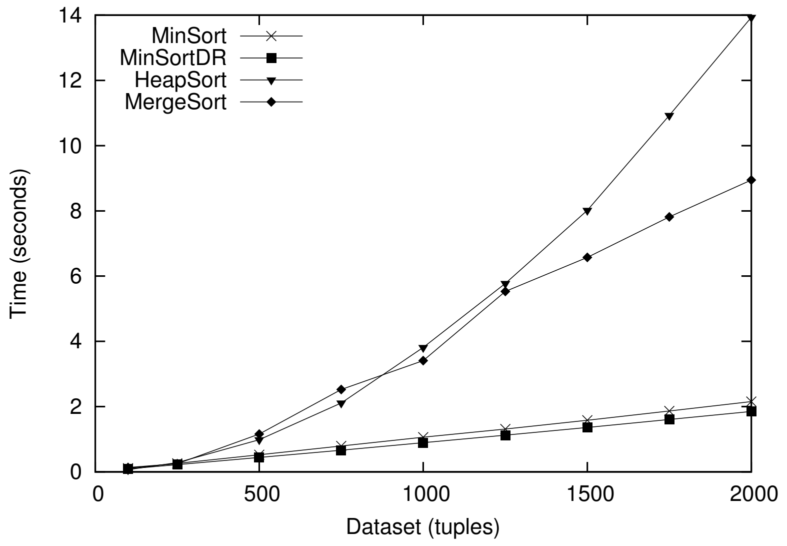


Figure 5.17: Time ($D_R = 8$)

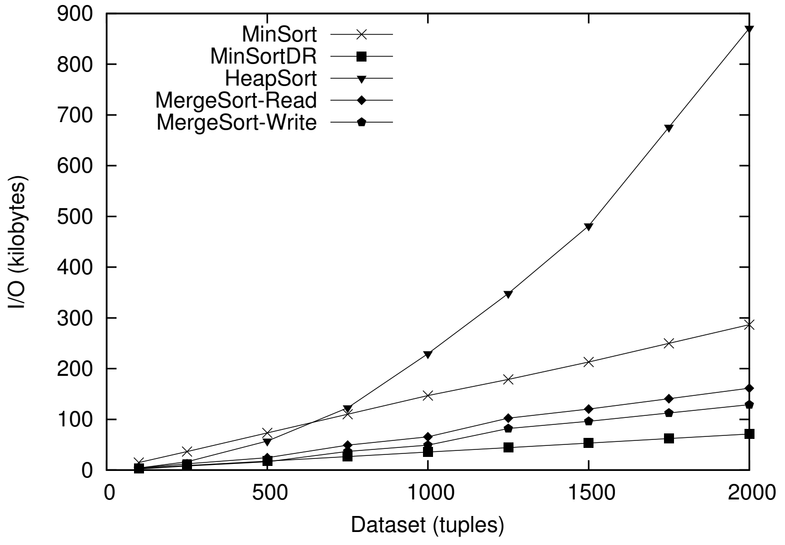


Figure 5.18: Disk I/O ($D_R = 8$)

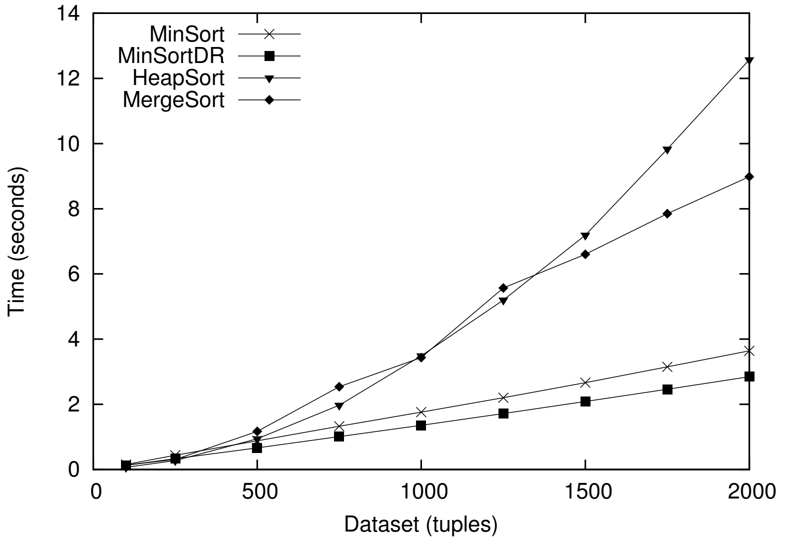


Figure 5.19: Time ($D_R = 16$)

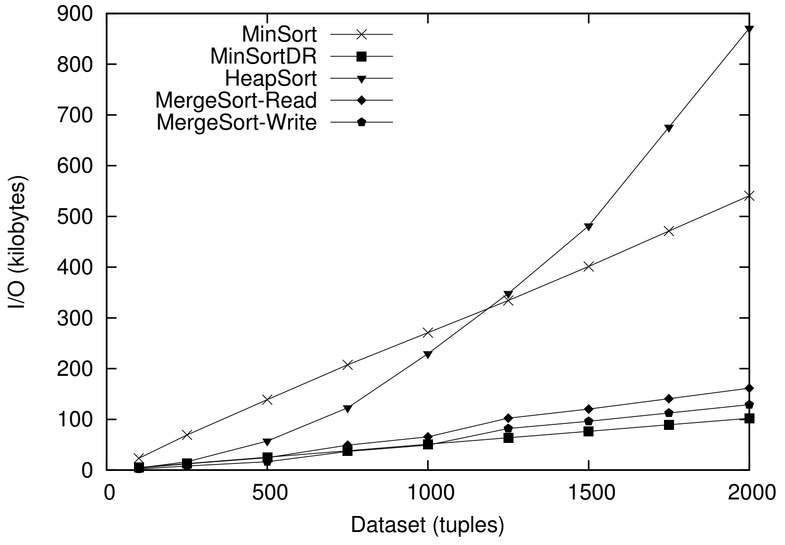


Figure 5.20: Disk I/O ($D_R = 16$)

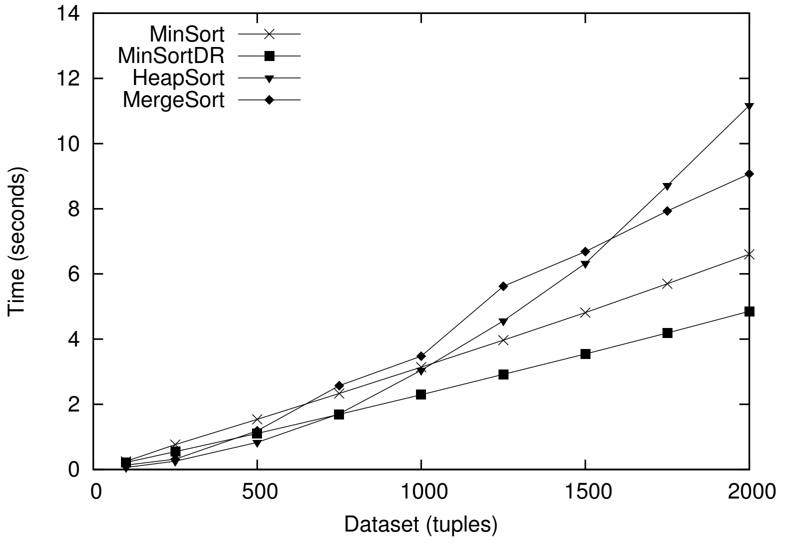


Figure 5.21: Time ($D_R = 32$)

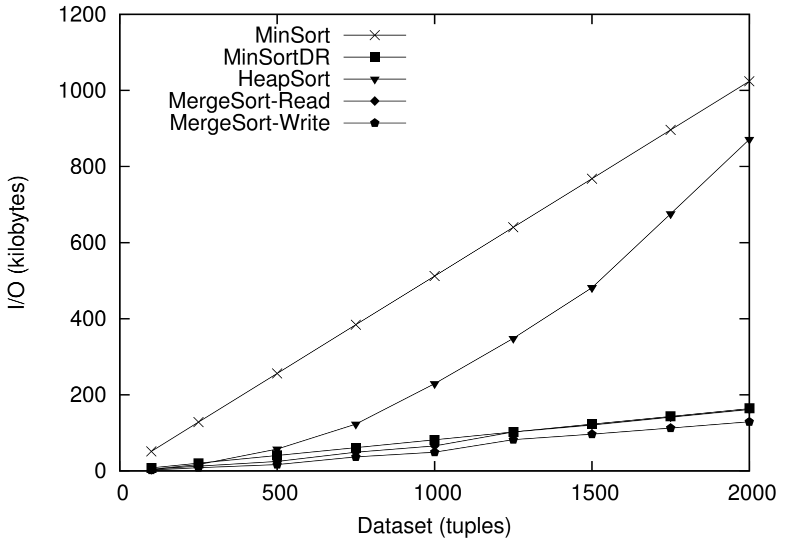


Figure 5.22: Disk I/O ($D_R = 32$)

5.8 Solid State Drives

Accessing data stored on a SSD is considerably different from accessing data stored on an individual flash chip. Although SSDs use arrays of flash memory chips as their underlying storage technology, they contain a sophisticated controller that provides block-level access to the operating system. This controller typically implements on-device buffering, logical-to-physical address translation, error detection and wear leveling. Unlike earlier experiments with flash chips, all I/O operations on a SSD must be performed at the page level. Some SSDs achieve nearly symmetric read/write throughput by using large on-board buffers and real-time data compression to reduce write amplification. The maximum performance of these drives can only be reached when they are used to store highly compressible data.

Experiments were run on a workstation containing an Intel Core i7-950 3.06GHz processor, 16GB DDR3 and a 120GB OCZ Vertex 2 SATA II SSD[20]. This workstation ran CentOS 5.6 and the SSD was formatted with the ext3 file system. The performance of most SSDs depends on the amount of data being transferred to or from the disk. Larger transfer sizes are split into page-sized chunks and transferred in parallel across an array of on-device flash chips. Figure 5.23 shows the read and write throughput of the OCZ Vertex 2. Throughput was calculated by averaging five runs of the IOzone benchmarking utility [10]. Optional flags were set to bypass the operating system's page cache and perform direct reads and synchronous writes on a 1GB file. A queue depth of one was chosen so that this test accurately reflects the workload created when sorting a single dataset. This drive has a maximum sequential read bandwidth of 238MB/s at a transfer size of 2MB. The maximum sequential write bandwidth was 205MB/s at a transfer size of 4MB. The SandForce 1222 controller in the Vertex 2 uses on-the-fly data compression to provide nearly symmetrical read and write bandwidth.

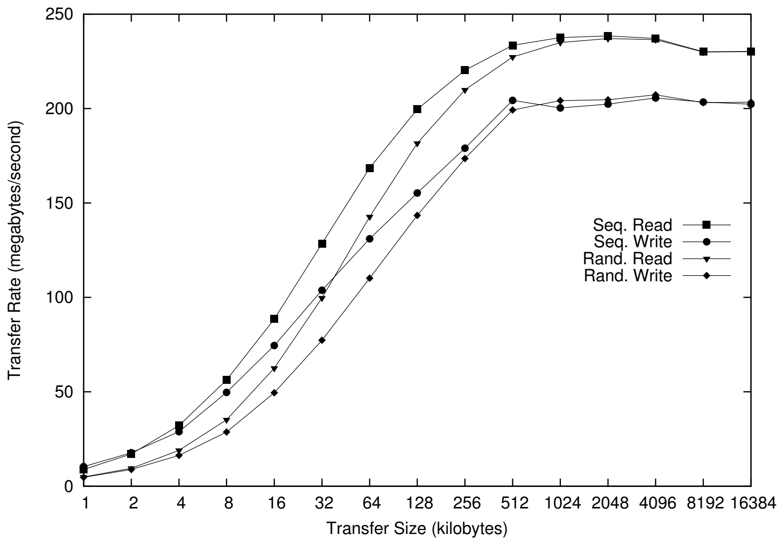


Figure 5.23: SSD Throughput Benchmark

The following experiments compare the Flash MinSort algorithm with external merge sort. These experiments use real and random datasets. Each dataset contains 10,000,000 tuples and has a total size of 16MB. The real dataset contains 135 distinct sort key values. This data is the result of copying the full soil moisture sensor dataset used in previous experiments 100 times. The random dataset contains 500 distinct sort key values that are uniformly distributed. All experiments use direct I/O to bypass the kernel's page cache.

These datasets are much larger than previous experiments on the sensor node. The size of index used by Flash MinSort can become very large when given enough memory to store a value for each page. To reduce the overhead of performing a linear scan over the entire index for each distinct sort key value, a binary min heap is used to store the index. A node in the heap consists of a region number and the smallest sort key value in that region.

Initially, the entire dataset is scanned to find the smallest sort key value in each region. Once a region has been scanned, its node is inserted into the heap. During each iteration of the algorithm, the root node is removed from the heap to determine the region to be scanned and the current sort key value. The pages of that region are searched for records that have the current sort key value. After all pages in the region have been read, the node is updated with the next smallest sort key value and inserted into the heap. If all tuples in a region have been sent to output, the node is not inserted into the heap. This process repeats until the heap is empty. There is additional overhead when using a heap to store the index since each region requires $8 + L_K$ bytes of memory on a 32-bit system. *Flash MinSort* is no longer a stable sorting algorithm when a heap is used for the index.

Unlike earlier experiments, data does not have to be transferred in chunks that are exactly equal to the physical page size of the SSD. The idea behind the *Flash MinSort* algorithm is to use a dynamic index to sort a dataset stored in external memory. Since I/O bandwidth is a bottleneck for external sorting, the use of a dynamic index can significantly reduce the amount of data transferred and the sort time. Given sufficient memory to store an index entry for each page, a smaller logical page size has the potential to reduce the amount of data transferred between external memory and the processor. A smaller logical page size will never result in more data being transferred; however, as seen in Figure 5.23, the transfer rate of a SSD is low at small transfer sizes.

Figures 5.24 and 5.25 show the execution time and disk I/O of the *Flash MinSort* algorithm as the logical page size is increased. These tests were run on the real dataset and the algorithm was provided with enough memory to create an index entry for each page of data. The optimal logical page size depends on the average number of distinct sort keys per page. When sorting the real dataset, the optimal logical page size is 32KB. If a larger logical page size is selected, the increase in disk I/O will outpace the increase in transfer rate. Figures 5.26 and 5.27 repeat this experiment using the random dataset. In this case, the optimal logical page size is 1MB.

Figure 5.28 shows the performance of the *Flash MinSort* algorithm when sorting the real dataset. The results are shown for page sizes of 32KB and 1024KB. Given less than 5KB of memory, *Flash MinSort* performs roughly the same amount of I/O for both page sizes and the larger transfer size is much faster. If the algorithm is given more memory, the smaller page size results in less disk I/O and better performance. At 30KB of available memory, *Flash MinSort* takes 41.23 seconds to sort the entire dataset. Figure 5.29 shows the performance of the *Flash MinSort* algorithm when sorting

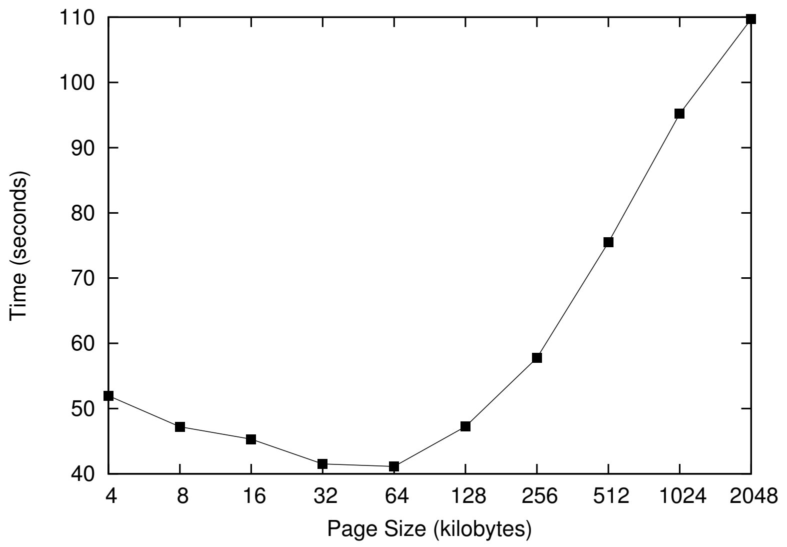
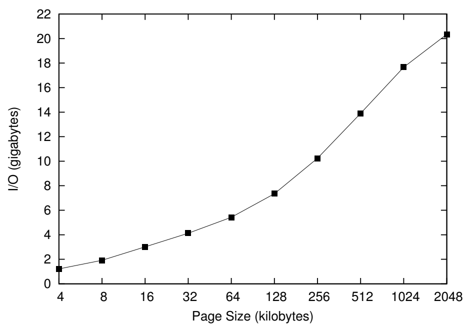
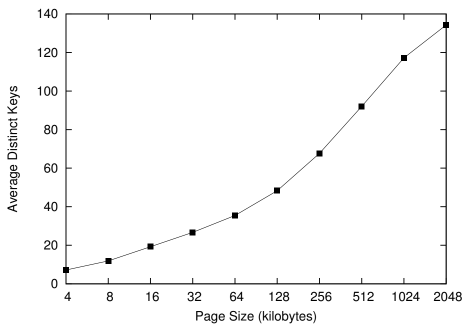


Figure 5.24: Real Data - Increasing the Logical Page Size (Time)



(a) Disk I/O



(b) Avg. Distinct Sort Keys Per Page

Figure 5.25: Real Data - Increasing the Logical Page Size (I/O)

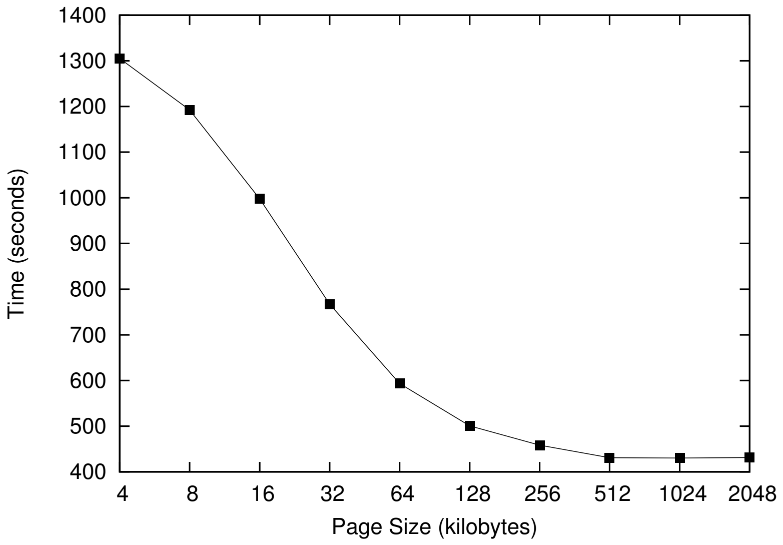
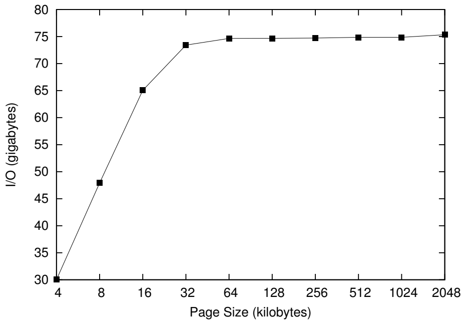
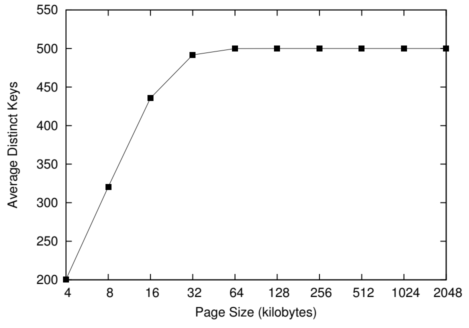


Figure 5.26: Random Data - Increasing the Logical Page Size (Time)



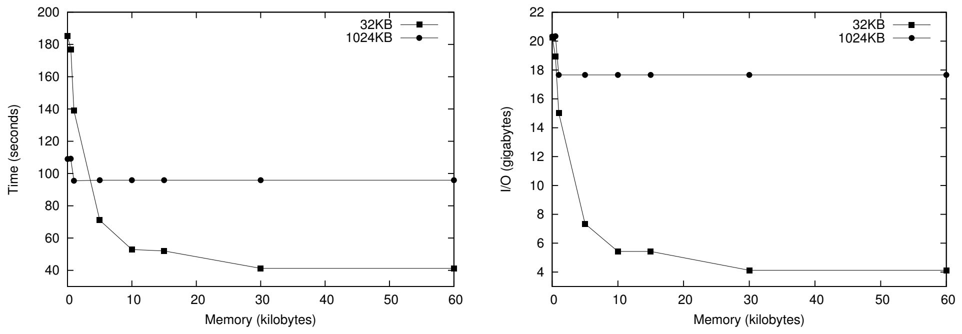
(a) Disk I/O



(b) Avg. Distinct Sort Keys Per Page

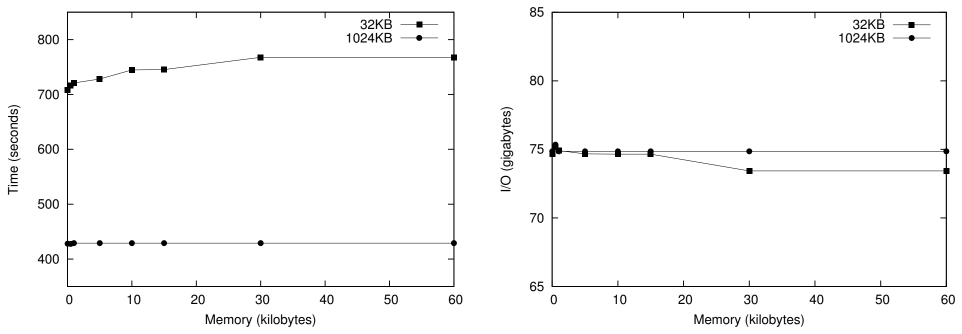
Figure 5.27: Random Data - Increasing the Logical Page Size (I/O)

the random dataset. In this case, the amount of disk I/O is roughly the same for both page sizes and the larger transfer size is always faster.



(a) Time (b) Disk I/O

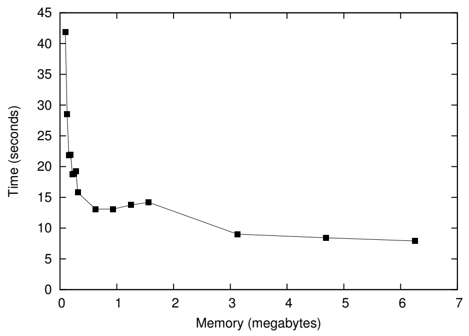
Figure 5.28: MinSort - Real Data



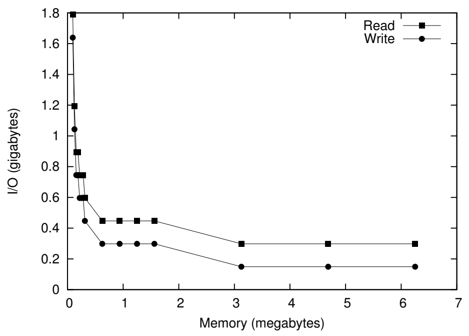
(a) Time (b) Disk I/O

Figure 5.29: MinSort - Random Data

Figure 5.30 shows the performance of *external merge sort* on the real dataset with a 32KB page size. *External merge sort* requires a minimum of three pages of memory. Figure 5.31 repeats this experiment with a 1024KB page size. The performance of *external merge sort* is not shown for the random dataset. The increase in execution time when sorting random data is very small since the amount of disk I/O is unchanged. The small increase in execution time is due to the time required to perform an in-place sort on random data during the initial run generation. A comparison of the two algorithms using the real dataset can be found in Figure 5.32. *External merge sort* outperforms *Flash MinSort* as the amount of memory increases.

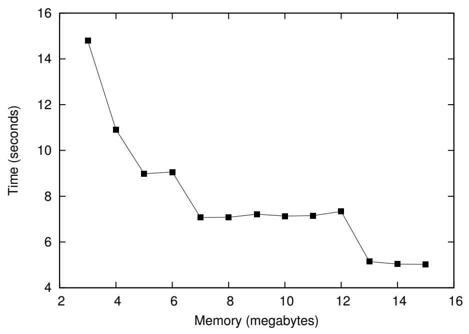


(a) Time

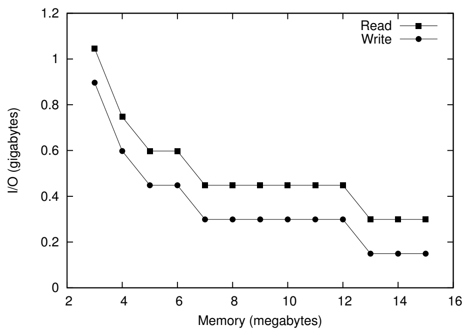


(b) Disk I/O

Figure 5.30: MergeSort - Real Data (32KB Page Size)

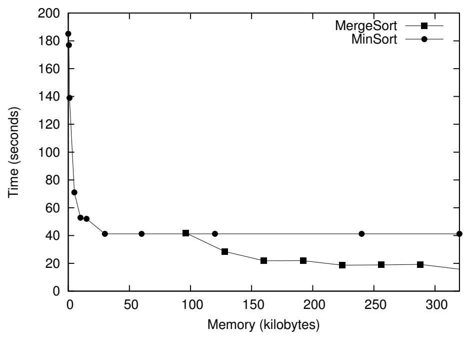


(a) Time

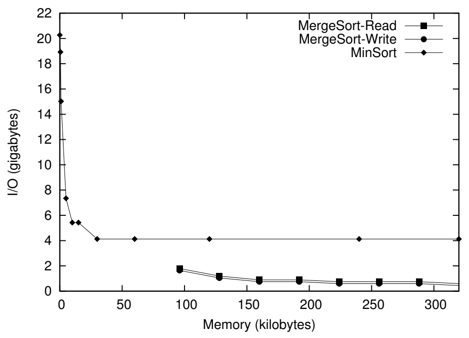


(b) Disk I/O

Figure 5.31: MergeSort - Real Data (1024KB Page Size)



(a) Time



(b) Disk I/O

Figure 5.32: Real Data Comparison (32KB Page Size)

Chapter 6

Conclusion

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 reasonable performance, but it will only be 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 systems applications. When given the smallest amount 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.

Another issue with *external merge sort*, and any other algorithm that performs writes of the entire relation in passes, is that 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 leveling. For applications whose primary function is environmental monitoring and data collection, dealing with the additional space required and wear leveling significantly complicates the design and performance.

The ability to adapt to the size of the dataset at runtime allows *Flash MinSort* to be used in a standard database query planner. The adaptive version sorts the dataset in-place when it is small enough to fit into memory. When the ratio of cache hits to page reads is large, the adaptive version of the algorithm provides a slight performance advantage. While *Flash MinSort* is executable when there is very little memory available, it is not able to outperform *external merge sort* running on a typical workstation with a SSD.

An efficient sort operator greatly increases the local data processing capability of energy-constrained 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 disk I/O operations required to sort a dataset. *Flash MinSort* can be combined with other energy efficient operators in an embedded database system to increase the functionality and lifetime of low-cost embedded systems.

Bibliography

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications*, 40(8):102–114, Aug 2002. → pages 9, 10, 11, 12

[2] N. Anciaux, L. Bouganim, and P. Pucheral. Memory Requirements for Query Execution in Highly Constrained Devices. In *VLDB*, pages 694–705, 2003. → pages 13, 14, 30

[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. → pages 13, 16

[4] Atmel. Atmel Flash AT45DB161D Data Sheet, 2010. → pages 9, 33

[5] Atmel. Atmel Corporation - Serial Flash, (*http://www.atmel.com/products/SFlash/*), 2011. → pages 8

[6] R. Beckwith, D. Teibel, and P. Bowen. Report from the Field: Results from an Agricultural Wireless Sensor Network. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 471–478, Washington, DC, USA, 2004. IEEE Computer Society. → pages 10

[7] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling Down Database Techniques for the Smartcard . In *IN VLDB*, pages 11–20, 2000. → pages 9

[8] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009. → pages 8

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

[10] D. Capps. Iozone Filesystem Benchmark (*http://www.iozone.org/*), 2006. → pages 50

[11] P. Chen, S. Oh, M. Manzo, B. Sinopoli, C. Sharp, K. Whitehouse, G. Tolle, J. Jeong, P. Dutta, J. Hui, S. Shaffert, S. Kim, J. Taneja, B. Zhu, T. Roosta, M. Howard, D. Culler, and S. Sastry. Experiments in Instrumenting Wireless Sensor Networks for Real-Time Surveillance. In *Proceedings of the International Conference on Robotics and Automation*, 2006. → pages 9

[12] T. Cossentine and R. Lawrence. Fast Sorting on Flash Memory Sensor Nodes. In *IDEAS 2010*, pages 105–113, 2010. → pages 2

[13] D. Culler, D. Estrin, and M. Srivastava. Guest Editors' Introduction: Overview of Sensor Networks. *Computer*, 37:41–49, 2004. → pages 9

[14] T. E. Daniel, R. M. Newman, E. I. Gaura, and S. N. Mount. Complex Query Processing in Wireless Sensor Networks. In *Proceedings of the 2nd ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM2HW2N '07, pages 53–60, New York, NY, USA, 2007. ACM. → pages 12

[15] Y. Diao, D. Ganesan, G. Mathur, and P. J. Shenoy. Rethinking Data Management for Storage-Centric Sensor Networks. In *CIDR'07*, pages 22–31, 2007. → pages 13

[16] 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. → pages 10, 33, 34

[17] M. J. Franklin, J. M. Hellerstein, and S. Madden. Thinking Big About Tiny Databases. *IEEE Data Eng. Bull.*, 30(3):37–48, 2007. → pages 12

[18] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1 edition, 2002. → pages 3, 4, 6, 8

[19] J. Gehrke and S. Madden. Query Processing in Sensor Networks. *IEEE Pervasive Computing*, 3:46–55, 2004. → pages 12

[20] O. T. Group. OCZ Vertex 2 Sata II 2.5 SSD - OCZ (*http://www.ocztechnology.com/ocz-vertex-2-sata-ii-2-5-ssd.html*), 2010. → pages 50

[21] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences - Volume 8*, HICSS '00, page 8020, Washington, DC, USA, 2000. IEEE Computer Society. → pages 11

[22] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures using Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 254–263, New York, NY, USA, 2007. ACM. → pages 9

[23] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Redwood City, CA, USA, 2 edition, 1998. → pages 5

[24] B. Krishnamachari. *Networking Wireless Sensors*. Cambridge University Press, 2005. → pages 9, 10, 11

[25] S. Madden. Intel Lab Data, (*http://db.csail.mit.edu/labdata/labdata.html*), 2004. → pages 40

[26] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30:122–173, March 2005. → pages 12

[27] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM. → pages 10

[28] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-Low Power Data Storage for Sensor Networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 374–381, New York, NY, USA, 2006. ACM. → pages 8, 11

[29] R. Min, T. Furrer, and A. Chandrakasan. Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 43 –46, 2000. → pages 11

[30] H. Park and K. Shim. FAST: Flash-Aware External Sorting for Mobile Database Systems. *Journal of Systems and Software*, 82(8):1298 – 1312, 2009. → pages 13, 15

[31] P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash Memory Cells - An Overview. *Proceedings of the IEEE*, 85(8):1248 –1271, aug 1997. → pages 8

[32] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Information Processing in Sensor Networks*, pages 364–369, 2005. → pages 11

[33] G. J. Pottie and W. J. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43:51–58, May 2000. → pages 12

[34] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava. Design Considerations for Solar Energy Harvesting Wireless Embedded Systems. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, pages 457–462. IEEE Press, 2005. → pages 11

[35] V. Raghunathan, C. Schurgers, S. Park, M. Srivastava, and B. Shaw. Energy-Aware Wireless Microsensor Networks. In *IEEE Signal Processing Magazine*, pages 40–50, 2002. → pages 11

[36] M. Vieira, C. J. Coelho, D. J. da Silva, and J. da Mata. Survey on Wireless Sensor Network Devices. In *Emerging Technologies and Factory Automation*, pages 537–544, 2003. → pages 9, 11

[37] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, 2001. → pages 6

[38] X. Wang and M. Cherniack. Avoiding Ordering and Grouping In Query Processing. In *VLDB*, pages 826–837, 2003. → pages 31

[39] T. Wark, P. I. Corke, P. Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-hurley. Transforming Agriculture through Pervasive Wireless Sensor Networks. *IEEE Pervasive Computing*, 6:50–57, 2007. → pages 10, 11