

FlashDB: Dynamic Self-tuning Database for NAND Flash

Suman Nath
Microsoft Research
sumann@microsoft.com

Aman Kansal
Microsoft Research
kansal@microsoft.com

ABSTRACT

FlashDB is a self-tuning database optimized for sensor networks using NAND flash storage. In practical systems flash is used in different packages such as on-board flash chips, compact flash cards, secure digital cards and related formats. Our experiments reveal non-trivial differences in their access costs. Furthermore, databases may be subject to different types of workloads. We show that existing databases for flash are not optimized for *all* types of flash devices or for *all* workloads and their performance is thus suboptimal in many practical systems. FlashDB uses a novel self-tuning index that dynamically adapts its storage structure to workload and underlying storage device. We formalize the self-tuning nature of an index as a two-state task system and propose a 3-competitive online algorithm that achieves the theoretical optimum. We also provide a framework to determine the optimal size of an index node that minimizes energy and latency for a given device. Finally, we propose optimizations to further improve the performance of our index. We prototype and compare different indexing schemes on multiple flash devices and workloads, and show that our indexing scheme outperforms existing schemes under *all* workloads and flash devices we consider.

Categories and Subject Descriptors: H.2.4 [Database Management Systems]: Query processing H.3.1 [Content Analysis and Indexing]: Indexing methods

General Terms: Algorithms, Design, Measurement, Performance.

Keywords: B⁺-tree, NAND Flash, indexing, log-structured index, self-tuning index.

1. INTRODUCTION

This paper presents a database for sensor networks using flash based storage. There are many use cases where it is desirable to store data within the sensor network, rather than transmit it all to a central database. A first example are remote deployments where an economical communication infrastructure is not available and an expensive low rate satellite or cellular connection is used, such as in polar regions or remote seismic deployments. Another example is the large class of applications for which the entire raw data is seldom required. A third example includes mobile sensor nodes [10, 23], with sporadic and short lived connections. A fourth

example includes sensor networks of mobile devices which have significant local processing power [4, 12]. In these cases rather than uploading the entire raw data stream, one may save energy and bandwidth by processing queries locally at a cluster-head or a more capable node and uploading only the query response or the compressed or summary data. Storage centric networks have also been discussed in [6, 7].

In most cases where the storage is part of the sensor network, the storage device used is flash based rather than a hard disk due to shock resistance, node size, and energy considerations. Additionally, flash is also common in many mobile devices such as PDA's, cell-phones, music players, and personal exercise monitors. These devices can benefit from a having light weight database.

Our objective is to design storage and retrieval functionality for flash storage. A simple method is to archive data without an index, and that is in fact efficient in many scenarios. However, as we show in section 6, for scenarios where the number of queries is more than a small fraction ($\approx 1\%$) of the number of data items, having an index is useful. Hence, we focus on indexed storage. Prior work on flash storage provides file systems (e.g., ELF [5]) and other useful data structures such as stacks, queues and limited indexes (e.g., Capsule [14], MicroHash [22]). Our goal is to extend the functionality provided by those methods to B⁺-tree based indexing to support useful queries such as lookups, range-queries, multi-dimensional range-queries, and joins.

Existing database products are not well suited for sensor networks due to several reasons. Firstly, existing products, including ones that run on flash [15], are originally designed for hard disks and are not optimized for NAND flash characteristics. Secondly, existing products are not targeted to most sensor network workloads. Sensor network storage workload may be highly write intensive: data may be added to the database more frequently than it is read. This is different from many traditional database applications where data is read more frequently than it is written. A log-structured B-Tree indexing method was proposed in [21] to address these two problems.

However, all existing indexing schemes, including those in commercial products and research prototypes, suffer from the drawback that they are not optimized for *all* available flash devices or realistic workloads. They do not consider all factors in the design space and are suboptimal in many situations. For example, as we will show later, the log-structured design [21] performs well with a write-intensive workload on an on-board flash chip, but performs poorly when run with a read-write workload or with a compact flash card. Similarly, disk-based designs are suitable for certain types of flash and workloads, but are suboptimal for others. This limits the use of existing schemes in practical system design, especially when the system is being designed for multiple types of workloads and flexibility in flash devices.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IPSN'07, April 25-27, 2007, Cambridge, Massachusetts, USA.

Copyright 2007 ACM 978-1-59593-638-7/07/0004 ...\$5.00.

We address this in FlashDB with a self-tuning indexing method that can adapt itself to the dynamic behavior of multiple device and workload parameters that affect performance. Our current prototype can run on more capable sensor nodes such as iMotes [11], ENSBox [8], XYZ [13], NIMS [17], or CarTel [10]; however, its simplicity and small memory footprint (6kB for a database of 30,000 records, independent of record size) promises its use in more resource constrained devices (such as motes).

We make the following contributions in this paper:

- We discuss the design space of factors that influence flash based storage system design, through experiments with our flexible flash test bed.
- Indexing has an extra resource overhead compared to archiving the raw data. We evaluate when indexing is desirable.
- We present a self-tuning B⁺-tree design that can adapt to the dynamic behavior of the workload and the underlying device.
- We formalize the self-tuning nature of the index as a two state task system and propose an online algorithm that matches the theoretical lower bound of competitive ratio.
- We provide a framework to determine the optimal size of an index node that minimizes energy and latency for a given device. This framework is applicable not only to our storage system design but also to existing indexing methods.
- We prototype and test our proposals on multiple flash devices such as compact flash and secure digital. Our evaluation shows that our indexing scheme outperforms existing schemes under *all* workloads and flash devices we consider.

The next two sections discuss the design space and the need for a self-tuning database. Sections 4 and 5 discuss our system design. Evaluation is presented in section 6.

2. FLASH STORAGE DESIGN SPACE

2.1 Flash Characteristics

Flash devices are primarily of two types: NOR and NAND. While NOR devices have faster and simpler access procedures, their storage capacity is lower and these are thus preferred for program storage. NAND flash offers significantly higher storage capacity (currently 32Gb in a single chip) and is more suitable for storing large amounts of data. The key properties of NAND flash that directly influence storage design are related to the method in which the media can be read or written, and are discussed in [22]. In summary, all read and write operations happen at page granularity (or for some devices up to 1/8th of a page granularity), where a page is typically 512-2048 bytes. Pages are organized into blocks, typically of 32 or 64 pages. A page can only be written after erasing the entire block to which the page belongs. Page write cost is typically higher than read, and the block erase requirement makes writes even more expensive. A block wears out after 10,000 to 100,000 repeated writes, and so write load should be spread out evenly across the chip.

The above characteristics are those of basic flash chips. These chips are however available in various packages such as compact flash (CF) cards, secure digital (SD) cards, mini SD cards, micro SD cards, USB sticks, and even as composite chip modules that provide a disk like ATA bus interface to the host node using the flash module. These packages are often preferred over raw chips due to several advantages. For instance, in remote deployments accessed only infrequently, data may be retrieved simply by replacing the flash cards. Moreover, the flash may be upgraded as higher capacity flash becomes available without redesigning the node circuit board. Generic sensor node designs may already provide for a PCMCIA or CF interface to attach a flash card (e.g., Stargate).

Device	R (μJ)	W (μJ)	R (μs)	W (μs)
Samsung 128MB	0.74	9.9	15	200
Compact Flash 512MB	2970	6220	18000	29000
Mini SD 512MB	109	22292	1100	193000
128MB+Mica2 [14]	57.83	73.79	0.969	1081

Table 1: Page read and write costs for some flash packages (R: Read, W: Write). The CF card used is a Sandisk Ultra II and the mini SD card used is from Kingston.

When used in one of these packages, the interface to the flash chip is through an abstraction layer called Flash Translation Layer (FTL). The FTL provides a disk like interface, which includes the capability to read and write a page directly without worrying about the erase-before-write constraint. The FTL also provides wear levelling by distributing writes uniformly across the media. However, FTL internally needs to deal with the characteristics of the underlying flash device. Thus, even if a storage system uses a flash package with a built-in FTL, it is prudent to consider the flash characteristics in the storage design.

2.1.1 Experimental Observations

We measure the behavior of different flash packages to guide our design decisions in FlashDB. We set up a testbed that allows us to read and write pages directly to flash packages without using the file system. Testbed details can be found in [16].

Read/Write Costs. Table 1 shows the energy consumed and time taken for page read and write for various flash cards, as derived from our experiments. In addition, it also shows the costs for a flash chip interfaced to a mote [14] (row 4). As a comparison, we also include the same data for a flash chip [19] from its data sheet (row 1), if it were to be interfaced in a manner that the chip itself were to be the bottleneck in read-write operations.

Our measurements show that *read write costs and their ratios differ significantly across flash packages*. For example, a write is ≈ 200 times more expensive than a read for the mini SD card, while it is only ≈ 2 times more expensive for the CF card.

Moreover, in some instances, the data transfer bus used to interface the flash chip may not be fast enough to take full advantage of the chip speed, thus making the energy and time costs dependent on the bus speed rather than chip limitations [22].

Access Pattern. Our measurements show that the energy and time to read and write a page on different flash packages follow a linear model consisting of a fixed access cost and an incremental (per page) access cost. The energy E_r and E_w spent for reading and writing N_p consecutive CF card pages can be approximated by $E_r = 2884.53 + 92.41N_p(\mu J)$ and $E_w = 6144.57 + 121.93N_p(\mu J)$. This model is based on extensive measurements for $N_p \in [1, 10000]$ and averaging over 10 random runs for each measurement. For the SD card, the models are $E_r = 96.79 + 20.98N_p(\mu J)$ and $E_w = 22350.40 + 21.41N_p(\mu J)$.

A similar model for a flash chip interfaced to a mote was shown in [14]. Thus, accessing a large number of pages at once may be beneficial since that would amortize the fixed cost over a larger number of pages.

Another observation in our experiments is that re-writing to the same (logical) page address is slower than writing to a new page address in sequential order. The variation is small for read. The average page write time, for the mini SD card, for re-writing to the same page address was observed to be four times that for writing to a subsequent page in sequential order. The reason for this comes from the page write constraint for flash devices mentioned earlier, and the method used to cope with it in the card's firmware. The

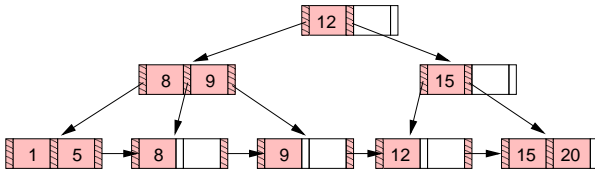


Figure 1: A B⁺-tree.

exact numbers observed are specific to the cards used and may vary across manufacturers.

2.2 Workload Properties

Two important workload properties can affect the performance of a storage system.

Read-write Ratio. While most sensor applications are write-intensive, they do experience varying read-write ratio over time. For example, in an acoustic sensing application [8], a sensor can store data frequently when birds chirp around it and can read data frequently when scientists query sensors for different chirping instances. The data access pattern may even change dynamically or vary for different segments of the stored data. For instance, certain data points below an interesting threshold value may be rarely retrieved but other data values may be frequently accessed.

Even when the application workload is mostly write-only, the workload experienced by the index itself may not be so. As we show in Sections 2.3 and 3, updating an index requires reading a part of it to locate where to apply the update. Therefore, different parts of the index can experience different read-write ratio; i.e., some part of the index can be read-intensive while some other part can be write-intensive.

Data Pattern. The variation of data values to be indexed can affect the reads/writes seen by different parts of an index. If the data to be indexed are random in nature, different parts of the index structure are likely to be updated with similar frequency. However, if the data is correlated in nature, such as if data values measured over a short time period lie within a short range, some part of the index may become more write-intensive than others at a given time. In many situations, workload pattern may not be known a priori or may change over time.

2.3 Indexing Methods

To support database-style queries on stored data, we consider B⁺-tree, a popular indexing data structure used in various incarnations in database systems. Powerful queries such as lookup, range queries, multi-dimensional range-queries, and joins can be efficiently supported by a B⁺-tree. It supports efficient operations to find, delete, insert, and browse the data. Typically, it is used as an external (outside of RAM) index to maintain large data sets. A B⁺-tree is a balanced tree in which every path from the root node to a leaf node has the same length. In each B⁺-tree node, keys (value field being indexed) and pointers are interspersed as a list $\langle p_0, k_0, p_1, k_1, \dots, p_d \rangle$, where $\lceil n/2 \rceil \leq d \leq n$ (except the root node) and n is fixed for a particular tree. The elements k_i in the list represent keys to be indexed, in sorted order while the elements p_i represent pointers to child nodes or to data records. The leaf nodes store all the keys to be indexed. In a leaf node, a pointer p_i points to the actual data record with key k_i . The last pointer p_d points to its next sibling, which helps scanning a large range of data in sorted order. In a non-leaf node, a pointer p_i points to the subtree whose leaf nodes contain keys k in the range $k_i \leq k < k_{i+1}$. Figure 1 shows an example B⁺-tree index.

To search for a key k in a B⁺-tree index, the tree is traversed top (root) to bottom, choosing the appropriate child pointer at every

non-leaf node. This requires reading all the nodes on the path from the root node to the target leaf node. To search for keys within a range $[l, h]$, the leaf node containing l is first located. Then sibling pointers are used to read the next leaf node until the node contains keys $> h$. To insert a key k , the leaf node L that should contain k is first located. If L contains less than n items, k is inserted into L . Otherwise, half of L 's items are put into a new node L' and k is inserted into L or L' . Creation of the new node L' requires recursively adding a new entry into L 's parent node. Finally, to delete a key k , the leaf node L containing k is first located (via top-down traversing). If after deletion of k , L contains less than $n/2$ items, it is merged with the previous leaf node and pointers to it are recursively deleted from L 's parent node. See [20] for details.

Two different B⁺-tree designs exist for flash based indexes. The first design, here referred to as B⁺-tree(Disk), assumes that the storage is a disk-like block-device. The second design, called B⁺-tree(Log), uses a log-structured index.

B⁺-tree(Disk). This design is built upon a disk-like abstraction (such as the one provided by FTL) over flash. A B⁺-tree node, depending on its size, is stored over multiple consecutive flash pages. To read the node, corresponding pages need to be read. To update a node, corresponding pages are read into RAM, modified, and then written back. Microsoft SQL Everywhere [15] uses this design.

The advantage of B⁺-tree(Disk) design is code portability: the existing implementation for hard disks can be run on flash. The disadvantage is that updates are expensive. Even if only a small part of a B⁺-tree(Disk) node is updated (e.g., a pointer is changed), the whole node needs to be read into RAM and written back. If the node is written to a new physical page, the old page needs to be garbage collected. Therefore, B⁺-tree(Disk) is inappropriate with write-intensive workload.

B⁺-tree(Log). This design, inspired by log-structured file systems [5, 18] and proposed in [21], avoids the high update cost of B⁺-tree(Disk). The basic idea behind B⁺-tree(Log) is to organize the index as transaction logs. A write operation on a B⁺-tree node is encoded as a log entry and is placed in an in-memory buffer. When the buffer contains enough data to fill a page, it is written to flash. Another in-memory data structure maintains, for each node, a linked list of page addresses where log entries for the node are stored on the flash.

The advantage of B⁺-tree(Log) is its small update cost since the page write cost is amortized over multiple updates. However, reading a node is expensive because many log entries, which may spread over multiple pages, need to be read to construct the node.

3. THE NEED TO SELF-TUNE

We argue that existing indexing schemes are not optimized for *all* flash devices or *all* workloads discussed in the last section, limiting their performance, especially when the system is designed for multiple types of workloads and flexibility in flash devices. To address this shortcoming, an indexing scheme must be able to dynamically adapt itself to underlying storage and workload properties. The following discussion uses some evaluation results given in Section 6.

Which of the above two B⁺-tree designs should one use? The decision depends on the workload and device characteristics. B⁺-tree(Log) is optimized for write-intensive workload (e.g., where data is queried rarely) and for devices where writes are significantly more expensive than reads (e.g., SD card), because B⁺-tree(Log) reduces flash writes at the cost of increased flash reads. Our evaluation shows that B⁺-tree(Log) is 80% more efficient than B⁺-tree(Disk) with SD card and write-only workload. However, changing either device property or workload property can make B⁺-

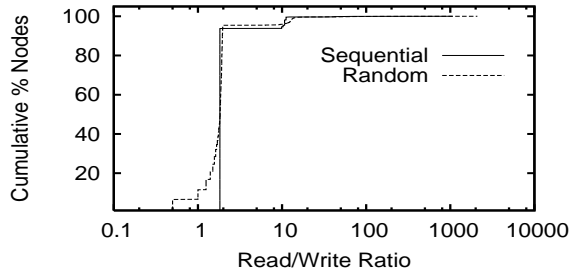


Figure 2: Cumulative distribution of read/write ratio of different nodes in a B⁺-tree. The workload consists of 30K writes of random and sequential values.

tree(Log) perform worse than B⁺-tree(Disk). For example, with a CF card, whose read and write costs are comparable, B⁺-tree(Log) performs 32% worse than B⁺-tree(Disk) (even with a write-only workload), and with a read-intensive workload, B⁺-tree(Log) can be 40% more expensive than B⁺-tree(Disk). This happens because the increased cost of additional reads in B⁺-tree(Log) does not get compensated by the reduced cost of writes.

What if the workload or flash device used is unknown or can change over time? Multiple applications with different workload properties may use the same database. The same application may experience varying read/write ratio at different times (e.g., a write-intensive sensing application may become read-intensive when several queries follow an interesting event). The flash device may be upgraded after the database has been designed. The database designer may not know which flash card the application user will plug in. As discussed above, choosing an inappropriate design can incur a high penalty in performance. This motivates a self-tuning indexing scheme that dynamically adapts its storage structure to the workload and underlying storage device, thus optimizing energy and latency for *any* workload or storage.

The self-tuning aspect of an indexing scheme can further increase the performance beyond the best achievable by either B⁺-tree(Log) or B⁺-tree(Disk), even if the workload and device are fixed. Assume that the workload is write-intensive and the flash used has significantly higher write cost than read. We know that B⁺-tree(Log) is better than B⁺-tree(Disk), but *is B⁺-tree(Log) optimal?* Consider the most favorable workload for B⁺-tree(Log)—insert-only workload. Note that even though the workload only inserts new keys to the B⁺-tree(Log), individual B⁺-tree nodes see both read and write operations (many B⁺-tree(Log) nodes are read in order to locate the right nodes to insert keys). Figure 2 shows the cumulative distribution of read/write ratio of different nodes of a B⁺-tree constructed by a write-only workload. It shows that although most of the nodes of the tree have very small read/write ratio, some nodes have very high read/write ratio. Generally, these nodes are near the root node and are read to access leaf nodes. Clearly, B⁺-tree(Log) design is not suitable for these nodes; rather they should be structured as B⁺-tree(Disk) nodes. Thus, the index should be able to tune its structure at a granularity finer than the entire index; each node should be organized independently depending on the workload it experiences.

4. FLASHDB

FlashDB is a database optimized for flash devices. It is self-tuning; after it is initially configured with the page read and write costs of the underlying storage device, it automatically adapts its storage structure in a way that optimizes energy consumption and latency for the workload it experiences. Thus, FlashDB instances

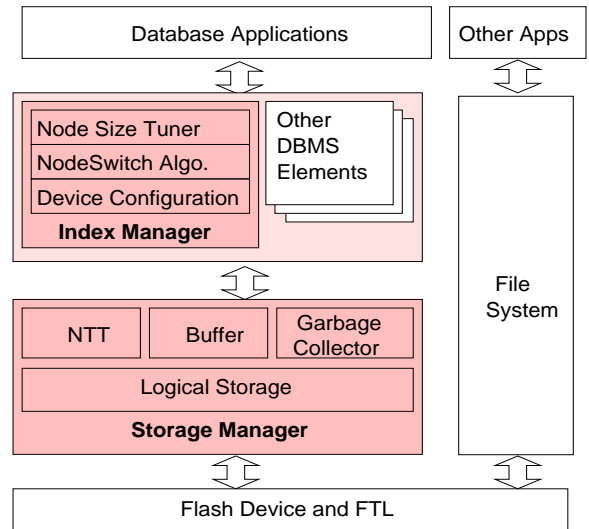


Figure 3: FlashDB Architecture.

running on different flash devices or having different workloads (e.g., with different read/write ratio or different data correlation) will choose different organization of data on the underlying physical device. FlashDB consists of two main components (Figure 4): a Database Management System that implements the database functions including index management and query compilation, and a Storage Manager that implements efficient storage functionalities such as data buffering and garbage collection.

The primary focus of this paper is the self-tuning Index Manager (that uses a B⁺-tree data structure) of FlashDB’s database management system and related functionalities of the Storage Manager. In the rest of this section, we briefly describe the Logical Storage component of Storage Manager. The Index Manager and the rest of the Storage Manager will be described in next two sections.

Logical Storage (LS). LS provides a logical sector address abstraction on top of physical flash pages. Components over LS access sectors through two APIs, `ReadSector` and `WriteSector`, in the granularity of a *sector*, typically of the same size as a physical flash page. Also, available addresses for writing may be obtained via `Alloc` and unused sectors freed using `Free`. LS hides flash-specific complexities using:

Out-of-place Update: As in-place update is expensive in all flash devices, when `WriteSector(addr, data)` is called, LS finds the next unused physical page p , writes `data` to it, and maintains a mapping from logical address `addr` to p . The page previously mapped by `addr` is marked dirty.

Garbage Collection: It cleans dirty pages produced by `Free` and `WriteSector` operations. Since a page can not be erased independently, first, a flash block containing dirty pages is chosen. Then, valid pages of the block are copied to another block. Finally, the block is erased.

The Storage Manager (SM) is configured with a partition of the physical storage space; other applications bypassing SM, such as file systems, operate outside this partition. The SM partition can be grown or shrunk dynamically. Growing the SM storage partition does not affect existing data; subsequent `Alloc` and `WriteSector` operations take this additional physical space into account. Shrinking the partition requires remapping used sectors and copying their data to pages within the new partition.

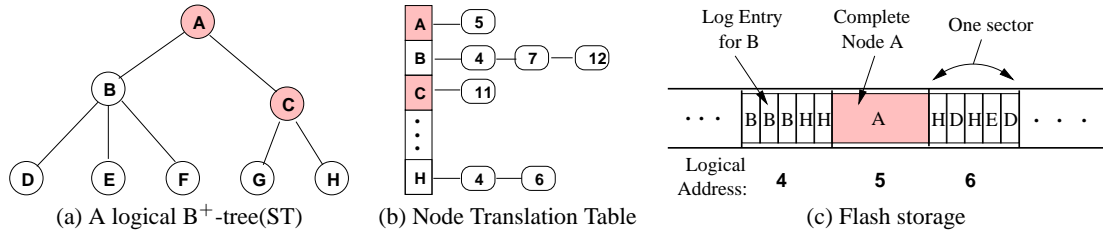


Figure 4: A logical B⁺-tree(ST), part of the Node Translation Table, and corresponding layout of data on flash storage. In (a), shaded nodes are in Disk mode while the others are in Log mode. In (c), sector 5 contains the whole node A while sectors 4 and 6 contains log entries for different nodes in Log mode.

5. SELF-TUNING INDEXING

In this section, we present our self-tuning B⁺-tree (hereafter referred to as B⁺-tree(ST)) designed for NAND flash.

5.1 B⁺-tree(ST) Design

The fundamental new feature of B⁺-tree(ST) is the flexibility to store an index node in one of two modes: *Log* or *Disk*. When a node is in *Log* mode, each node update operation (e.g., adding or deleting keys) is written as a separate log entry, similar to B⁺-tree(Log). Thus, to read a node in *Log* mode, all its log entries (which may be spread over multiple pages) need to be read and parsed. When a node is in *Disk* mode, the whole node is written together on consecutive pages (number of pages depends on the node size), and hence reading a node requires reading the corresponding sectors. At any point of time, some logical nodes of a B⁺-tree(ST) may be in *Log* mode while the others in *Disk* mode. Moreover, nodes may change their modes dynamically as workload or storage device change. Figure 4(a) shows a snapshot of a logical B⁺-tree(ST).

5.1.1 Storage Manager Components

B⁺-tree(ST) introduces two components in the Storage Manager of FlashDB: a *Log Buffer* and a *Node Translation Table (NTT)*. The Log Buffer, which can hold upto one sector worth of data, is used only by the nodes currently in *Log* mode. When a node in *Log* mode is modified, the corresponding log entries are temporarily held in the Log Buffer. When a page worth of entries are collected, they are written to flash together, to avoid expensive small writes.

For simplicity, FlashDB currently supports ACID semantics of individual read/write operations only; more complex transactions will be explored in our future work. To ensure this, all or none of the log entries for an individual write operation are flushed from the Log Buffer to flash. This is done by flushing the Log Buffer to flash *before* a write operation if the Log Buffer does not have enough space to hold *all* the log entries for the new operation.

The NTT maps logical B⁺-tree(ST) nodes to their current modes and physical representations. Figure 4(b) shows a part of the NTT for the logical tree in Figure 4(a). For a node in *Disk* mode (e.g., node A), the NTT records the addresses of the sectors (e.g., 5) where the node is written on flash. For a node in *Log* mode, the NTT maintains a linked list of addresses of all the sectors that contain at least one valid log entry for that node. For example, in Figure 4(b), node B has at least one log entry in sectors 4, 7, and 12. Note that a sector containing node B’s log entries can contain log entries for other nodes as well (e.g., in sector 4 in Figure 4(c)), as the Log Buffer may have log entries for many nodes when flushed.

5.1.2 Operations

As described in Section 2.3, operations such as key search, addition, and deletion on a B⁺-tree translate to create, read, and update of tree nodes. Given the NTT, we perform these node-level operations as follows. To create a node with id x , we create an entry

with id x with *Log* mode in the NTT. To read or update a node x , we first read its current mode from $NTT[x]$. If x is in *Disk* mode, we read the node from or update to the sectors given by $NTT[x]$. Operations on x in *Log* mode are more involved. To update x , we construct a log entry for the update operation and put it into the Log Buffer. Later, when the Log Buffer has one sector worth of data, all the log entries in the Log Buffer are written to an available sector provided by the `Alloc()` API of Logical Storage and the address of the sector is added at the beginning of the linked list at $NTT[x]$. To read x , we read the Log Buffer and all the sectors in the linked list at $NTT[x]$ to collect log entries for x and parse the logs to construct the logical node.

5.1.3 Structure of Sector, NTT and Log Entries

B⁺-tree(ST) data, before being written to a flash sector, is encapsulated with a small header that contains the following fields: 1) *Checksum* to check for errors during read, and 2) *NodeMode* which can be *Log* or *Disk*. *NodeMode* enables identifying sectors before applying optimizations for a specific mode (e.g., compaction and garbage collection described in Section 5.3.1).

Each NTT entry contains the following fields: 1) *SectorList* which points to sectors containing the node or its log entries, 2) *IsLeaf* which is true if the logical node is a leaf node, and 3) *LogVersion* which is the latest version of log entries of the node.

Each log entry of a node in *Log* mode contains the following fields. 1) *NodeID*: it is the logical node id which distinguishes log entries of different nodes. 2) *LogType*: it describes the operation on the logical node and can be of three types: `ADD_KEY`, `DELETE_KEY`, and `UPDATE_POINTER`. The first two types are used to add and delete a (key, pointer) tuple in B⁺-tree nodes, while the last type is required for updating the last pointer of a nonleaf node¹ or the sibling pointer of a leaf node. It is relatively easy to convert a node into, or construct a node from, log entries of these three types. 3) *SequenceNumber*: it is incremented by one after each log entry generated for the logical node. It helps in applying the log entries in the order they are generated. 4) *LogVersion*: it is the latest version of log entries, as logs can become stale due to our compaction mechanism described in Section 5.3.1.

The log entries contain enough information such that even if the application crashes and loses its in-memory NTT, the NTT can be reconstructed by scanning the entire flash. This is an expensive operation and can be avoided by periodically checkpointing the NTT into flash (Section 5.3.3).

5.2 Self-tuning Issues

To make B⁺-tree(ST) efficient and self-tuning, the mode of a node must be decided and updated carefully. Further, the size of an index node must be chosen optimally. The second issue can help improve B⁺-tree(Disk) and B⁺-tree(Log) designs also.

¹Each node has one more pointers than keys.

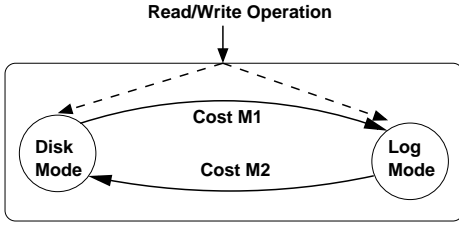


Figure 5: A B⁺-tree(ST) node switching between two modes.

5.2.1 Mode Switching Algorithm

At the heart of B⁺-tree(ST), there is an algorithm that decides when a node should switch between *Disk* and *Log* modes. Switching between modes incurs costs. To switch a node x from *Log* to *Disk* mode, the node is constructed by reading the log entries for the node, and is written back in *Disk* mode. To switch it from *Disk* to *Log* mode, first, the node is read in *Disk* mode, then, it is encoded into a set of log entries representing the node, and finally, the logs are placed in Log Buffer. $NTT[x]$ is modified accordingly.

We now address the switching algorithm. Since a node in *Disk* mode is optimized for reads and a node in *Log* mode is optimized for write, intuitively, a node should be in *Disk* mode (or in *Log* mode) when it expects to see a lot of read operations (or, write operations respectively). Switching between modes incurs costs, and so we need to ensure that nodes do not switch modes unnecessarily. Moreover, since switching decisions must be made dynamically, we need an online algorithm for switching.

We can abstract the switching problem as a *Two-state Task System* shown in Figure 5. A node can be in two modes: *Log* and *Disk*. A read R or a write W can be served by the node in either mode, but the costs are different in different modes. The node can switch from one mode to the other by paying a certain cost. The goal is to find an online algorithm for the node to dynamically choose modes to minimize the total cost of serving requests and switching modes.

The above problem is a generalization of file migration on two states [2] and a special case of metrical task system [3]. This problem has a known lower bound of 3 on the competitive ratio² [3].

Our online algorithm to make switching decisions in B⁺-tree(ST) is shown in Algorithm 5.2.1. The algorithm is simple and practical: it only needs to be configured with the costs of reading and writing a page and can be implemented with a single counter per node. The algorithm is run independently for each node of the tree. Moreover, its competitive ratio matches the theoretical lower bound, as stated in Theorem 1.

THEOREM 1. *Algorithm SWITCHMODE is 3-competitive.*

Proof: See [16].

Note that the competitive ratio assumes worst case scenarios; our algorithm performs much better in practice. Our evaluation shows a performance within $1.3\times$ of the optimal solutions with a real workload (Section 6).³

Implementation. SWITCHMODE implementation requires maintaining one counter for every B-tree node, representing the accumulated difference of costs of the two modes since the last mode

²Competitive ratio of an online algorithm is the worst case of the ratio between the cost incurred by the algorithm and the best-case cost (possibly found by an offline optimal algorithm).

³We recently developed another 3-competitive algorithm that performs within $1.05\times$ of the optimal algorithm with real workloads [1].

ALGORITHM 1. SWITCHMODE

(The following algorithm runs for each B-tree node n)

1. Initialize $S \leftarrow 0$ when migrate to the current mode.
 2. For every read-write operation O
 - Suppose c_1 is the cost of serving O in current mode and c_2 is the cost of serving it in the other mode.
 - $S \leftarrow S + (c_1 - c_2)$
 3. Suppose, M_1 and M_2 are the costs for transition between two modes. Then, switch to the other mode if $S \geq M_1 + M_2$
-

switch. We store the counter for node x in $NTT[x]$. Suppose c_r and c_w represent the costs of reading and writing one sector (i.e., one page in the underlying flash). Also assume that a B⁺-tree(ST) node in *Disk* mode takes k_s sectors. Then, computing costs in *Disk* mode is simple: each read and write operation on the node costs $k_s \cdot c_r$ and $k_s \cdot c_w$ respectively. In *Log* mode, if a write operation generates l log entries, and a flash page can contain maximum k_e entries, the cost of the write operation is $l \cdot c_w / k_e$. Similarly, if the log entries for a node are spread over p flash pages, then reading the node costs $p \cdot c_r$. If the node is currently in *Log* mode, values for p and l can be accurately determined. However, if the node is currently in *Disk* mode, these values can only be estimated. Our evaluation shows that $p = 3$ and $l = 2$ are reasonable estimates.

5.2.2 Optimal Node Size

Theoretically, the size of a B⁺-tree node can be of any number of pages, such as $k_s = 1, 2, \dots$ or even a block⁴. Gray et al. [9], use a utility-cost analysis to suggested that for disk-based systems, a B⁺-tree node size of around 16KB provides the maximum utility-cost ratio, based on disk access costs. *What B⁺-tree node size is the optimal for flash?*

Intuitively, bigger nodes have the benefit of shortening the height of the tree and thus reducing the number of nodes need to be read to reach from the root to the target leaf node. However, a bigger size also increases the cost of retrieving an individual node: the bigger the node, the more data needs to be read from the storage. These two competing factors lead to the following utility-cost analysis.

Consider a B⁺-tree indexing N items. The size of a B⁺-tree node is $NodeSize$ and a node can contain $EntriesPerNode$ entries. Then, the height (in nodes) of the tree is given by:

$$Height \sim \log_2(N) / \log_2(EntriesPerNode) \quad (1)$$

The *utility* on a B⁺-tree node measures how much closer the node brings an associative search to the destination data record leaf node, and is defined by the divisor of Equation 1.

$$NodeUtility = \log_2(EntriesPerNode) \quad (2)$$

For example, if each index entry is 16 bytes, then a 1 KB index page that is 70% full will contain about 44 entries. Such a node with have a utility of 5.5, about half the utility of a 48K node. Intuitively, the bigger a node, the smaller the height of the index, and the fewer the number of nodes required to touch before accessing a key at a leaf node.

Now consider the cost of accessing a node. First, assume that all B⁺-tree operations are writes and all nodes are in *Disk* mode. A write operation requires reading all the nodes in the path from the

⁴Storing a node in a flash block simplifies the design of a B⁺-tree, since we can do in-place update at block-level. However, it may hurt performance.

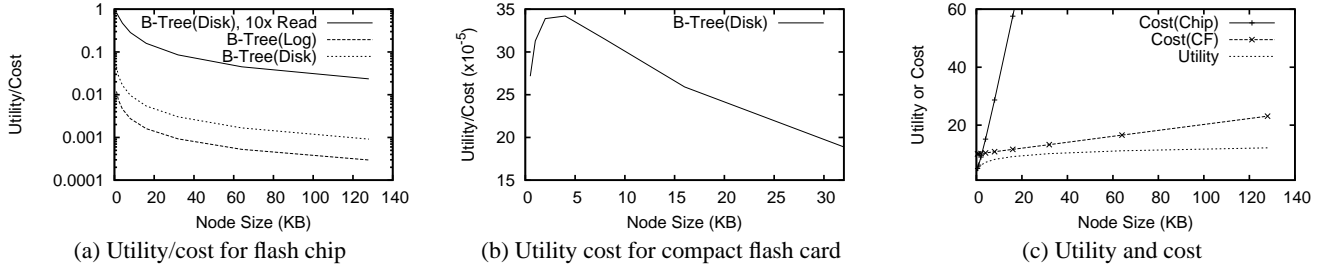


Figure 6: Utility/cost for B^+ -tree nodes of different sizes. The B^+ -tree has 30K 16 bytes index entries and each node is 70% full.

root to a leaf node and then writing at least the leaf node. Thus, the amortized access cost for a single node is:

$$Cost = \frac{Height}{Height + 1} ReadCost + \frac{1}{Height + 1} WriteCost \quad (3)$$

We can now plug experimentally observed costs from section 2.1.1 into the above equation. Finally, the utility/cost ratio of a certain node size is the ratio of Equation 2 and Equation 3.

We now relax some of the assumptions above. For B^+ -tree(Log), each individual node read operation requires reading multiple pages while each individual node write operation requires writing a fraction of a page. For read-intensive workload, nodes will be read more often than they are written. The above analysis can easily be extended for these two cases by adjusting the weights of $ReadCost$ and $WriteCost$ in Equation 3. The results of such analysis are shown in Figure 6(a).

Figure 6(a) plots the utility/cost ratios for Samsung K9K1G08R0B (128MB) flash chip under various node sizes and for real workloads (details in Section 6). It shows that, for all modes of a B-Tree node (*Disk* and *Log*) and for different workload (read-intensive and write-intensive), the utility/cost ratio is maximized when the node size is as small as possible. However, in practice, the smallest granularity of read/write operations in a flash is a page; therefore, the utility/cost ratio is maximized when a node can be fit in exactly one flash page (typically 512 bytes).

Figure 6(b) shows the utility/cost ratio for the Sandisk CF card (512MB), and surprisingly, ratio is maximized when the node size is approximately 4KB.

The difference can be explained by Figure 6(c). Suppose r denotes the ratio between fixed and incremental costs of accessing a storage device, i.e., if the cost of accessing x bytes is approximated as $a+bx$, $r = b/a$. For the compact flash card (CF), $r \approx 10^4$. Due to this high value of r , the access cost is highly dominated by the fixed cost and the cost curve for CF is relatively flat in Figure 6(c); therefore, it converges to the logarithmic utility curve until the node size is around 4K. However, flash chips have a relatively smaller value of $r \approx 10^3$, and therefore the cost curve diverges from the utility curve from the very beginning. Intuitively, the higher the value of r , the higher the optimal node size.

FlashDB uses the above framework to compute the optimal node size for the flash it operates on. The above analytical results match FlashDB experiments [16].

5.3 Optimizations to B^+ -tree(ST)

5.3.1 Log Compaction and Garbage Collection

In building an index, a B^+ -tree(ST) node x can get updated many times, resulting in a large number of log entries potentially spread over a large number of sectors on flash. This has two disadvantages. First, it makes $NTT[x].SectorList$ very long and increases the memory footprint of the NTT. Second, it becomes expensive

to read x , since a large number of sectors are read. To overcome these problems, we incorporate two types of log compaction. First, as done in [21], all the log entries for x are read into memory and then written back to a small number of new sectors. This is helpful, since log entries for x may share sectors with log entries of other nodes, and hence provides the opportunity to be clustered into fewer sectors. However, it still cannot guarantee an upper bound of the number of sectors required for a node, since the number of log entries for a node can grow indefinitely over time.

To address this concern, we propose a *semantic compaction* mechanism, where log entries having opposite semantics are discarded during compaction. For example, if the data item k is added to node x and then deleted from it later (e.g., during a node split operation after x becomes full), x will have log entries `ADD_KEY k` and `DELETE_KEY k`. These two log entries cancel each other and are hence discarded. Similarly, multiple `UPDATE_POINTER` log entries for x can be replaced by the last entry. For such compaction, we must consider the sequence number of the log entries such that we apply the logs in order. It is easy to show that if a node can contain at most n data items, it will have at most $n + 1$ log entries, bounding the size of the linked list in $NTT[x]$ to be $(n + 1)/EntriesPerSector$. Semantic compaction requires log entries to have a version number which is incremented after each semantic compaction. After compaction, $NTT[x]$ is updated with the current sector address list. During subsequent reads, log entries of order versions are ignored.

Semantic compaction introduces stale log entries (having older version numbers) and we use a Log Garbage Collection (LGC) component to reclaim the space. Note that LGC is different from the Garbage Collection (GC) in Storage Manager; GC reclaims spaces from dirty pages, while LGC reclaims spaces from dirty log entries. LGC is activated when the flash is low in available space (i.e., when Storage Manager fails to allocate a new sector.) It starts by scanning the whole flash. For each sector, LGC first looks at its header information to determine if the sector contains log entries. We call such sectors *Log* sectors. For each *Log* sector, LGC counts the fraction of stale log entries in that sector. If it is above a threshold, the sector is selected for garbage collection. LGC then writes the fresh log entries to the Log Buffer, removes the sector address from the NTT, and returns the sector to Storage Manager. The Log Buffer eventually flushes the log entries to flash and adds the new addresses to the NTT.

5.3.2 Bigger Log Buffer

If available, FlashDB can use a large Log Buffer as writing a large amount of data at a time has smaller per byte cost than writing smaller amounts at a time. In addition, when writing to flash, log entries can be reorganized such that entries of the same node stay in as few sectors as possible. This reduces NTT memory footprint and read cost, since fewer pages need to be read to collect all the log entries for a node.

5.3.3 Checkpoint and Rollback

FlashDB also supports checkpointing and rollback of indices. Checkpointing allows a device to capture the state of an index, while rollback allows it to go back to a previously checkpointed state. This helps deal with software bugs, hardware glitches, energy depletion, and other faults possible in sensor nodes.

Checkpointing requires making both in-memory states and in-flash data persistent. The NTT is less than $6KB$ as shown in Section 6.3, and storing it into flash is a negligible storage overhead. However, simply storing the NTT is not sufficient due to Logical Storage and Garbage Collection functions in the Storage Manager. First, the NTT keeps track of logical addresses of sectors and the Logical Storage may change the mapping between logical to physical addresses over time. So, if the rollback operation loads a previously checkpointed NTT, physical pages currently mapped by sector addresses in the NTT may not be the same ones mapped during the checkpoint time. To address this, we replace the sector addresses in a checkpointed NTT with their physical addresses. Second, garbage collection may copy the content of a page p to a new location p' and erase p . If p is part of a checkpointed version, future rollback operation will fail to find the data for p (which is now in p'). To address this, during garbage collection, we update the checkpointed NTT with p' . Note that, we do not need to update the whole NTT, only the page containing p needs update. Moreover, garbage collection is an infrequent operation, so the amortized cost is small. Since updating in-flash NTT is expensive we prefer blocks with no checkpointed data over the ones having checkpointed data for garbage collection.

Rollback requires loading the NTT into memory, creating new logical addresses in Logical Storage that map to the physical addresses in in-flash NTT, and placing the logical addresses in the restored NTT in memory.

6. EXPERIMENTAL EVALUATION

Our current FlashDB prototype is written in .NET compact framework, a common language run time environment for embedded devices such as Stargates. In this section we evaluate performance of this prototype. We enable the log compaction optimization (Section 5.3.1); but use a small Log Buffer to hold logs worth one flash page—a bigger buffer will provide better performance of FlashDB. In the rest of the section, we first try to understand under what situation indexing provides more benefit than archiving data in an append-only list. Second, we investigate B^+ -tree(ST)'s performance under different flash types and workloads. Third, we measure the memory footprint of B^+ -tree(ST). Finally, we investigate how SWITCH-MODE performs with real workload.

Flash Devices. We use the following types of flash in our evaluation: (1) FLASHCHIP: Samsung K9K1G08R0B (128MB) flash chip, (2) CAPSULE: a Toshiba flash chip interfaced to a mote [14] (we used the cost numbers reported in [14] in our flash emulator), (3) COMPACT FLASH (CF): the Sandisk compact flash card (512MB), and (4) SECURE DIGITAL (SD): Kingston mini SD card (512 MB). We use FLASHCHIP as the default flash type. The properties of the CF and mini SD cards were obtained from experiments with our testbed (Section 2.1.1).

Workloads. We use three different workloads. (1) LABDATA: a stream of temperature data collected from 35 sensors in our office building. It consists of 30,000 data points. (2) RANDOM: a sequence of 30,000 random data points. (3) SEQUENTIAL: a sequence of 30,000 unique data points in increasing order. We use LABDATA as the default workload. We use RANDOM and SEQUENTIAL for sensitivity study; the cost with real workloads

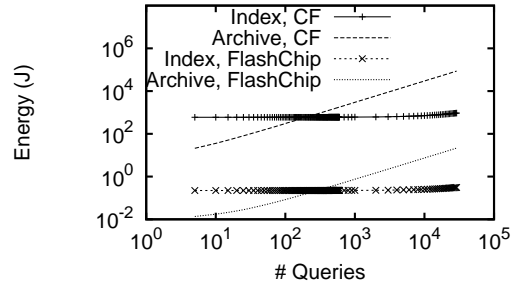


Figure 7: Comparison of energy costs for List archiving and indexing. Both the axes are in Log scale.

should be between that of RANDOM and SEQUENTIAL.

In all figures in this section, the labels SEQ, RND, and LAB denote SEQUENTIAL, RANDOM, and LABDATA data respectively. The labels Disk, Log, and ST refer to an index using B^+ -tree(Disk), B^+ -tree(Log), or B^+ -tree(ST) respectively.

6.1 When is Indexing Useful?

Different data management policies may be adopted for archiving a data stream. In one extreme, data can be stored as an append-only list (call it a List) with cheaper ($O(1)$) data insertion and more expensive query operation (each query would require a sequentially scan of cost $O(n)$). On the other extreme, data can be indexed using a B^+ -tree, with slightly more expensive ($O(\text{Log}(n))$) data insertion and cheap ($O(\text{Log}(n))$) query operation. Here we evaluate these two policies to understand when building a B^+ -tree index is preferable over a List.

Figure 7 compares energy consumption of B^+ -tree(ST) and List under different query rates. The x-axis shows the number of queries made, and the y-axis shows the total energy consumed by first archiving the LABDATA data (i.e., building B^+ -tree(ST) or List) and then querying the data. Each query asks for data within a randomly chosen small range.

When the number of queries made is small (≈ 0), List is more energy efficient than B^+ -tree(ST). This is expected, since the additional cost of building a B^+ -tree(ST) is not compensated by a small number of queries processed. However, the total cost for List grows fast as the number of queries increases. In contrast, additional queries incur much smaller cost with B^+ -tree(ST), demonstrated by relatively flat curves for B^+ -tree(ST). Interestingly, the curves for B^+ -tree(ST) and List cross at a very small number of queries: around 300 queries, for both types of flash in Figure 7. This tells us that *building an index is useful when we expect that the number of queries on the archived data will be more than 1% of the total number of data items.*

6.2 Tunability of B^+ -tree(ST)

In this section we evaluate how well B^+ -tree(ST) adapts with different devices and workloads. First, we use different types of flash with LABDATA as the default workload. Then, we use different workloads with FLASHCHIP as the default flash device. Finally, we vary both flash devices and workloads.

Varying Flash Devices. Figure 8(a) shows the energy consumed for indexing the LABDATA data with different indexing schemes over different types of flash. We see that B^+ -tree(Log) is 40% and 80% more efficient than B^+ -tree(Disk) when used over FLASHCHIP and SECURE DIGITAL respectively. On the other hand, B^+ -tree(Disk) is 38% and 32% more efficient than B^+ -tree(Log) for CAPSULE and COMPACT FLASH respectively. This supports our claim that

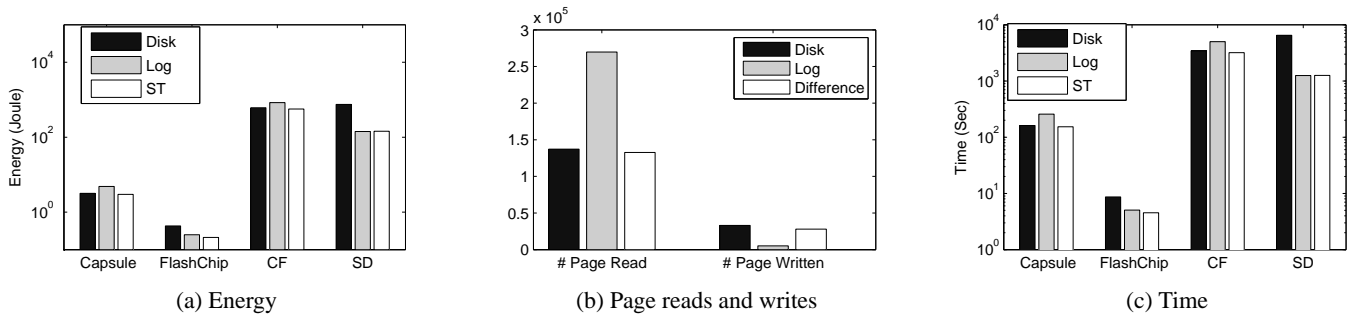


Figure 8: Energy and time with different storage devices.

none of these two schemes is flexible in terms of flash types. This can be explained by Figure 8(b), that shows the number of pages read and written by B^+ -tree(Disk) and B^+ -tree(Log). As shown, B^+ -tree(Log) writes $\approx 28K$ fewer pages and reads $\approx 132K$ more pages than B^+ -tree(Disk). This increased number of reads is justified over the reduced number of writes if a page read is at least $\approx 132/28 = 4.7$ times cheaper than a page write. For FLASHCHIP and SECURE DIGITAL, reads are $> 10\times$ cheaper than writes, making B^+ -tree(Log) more efficient than B^+ -tree(Disk) over these devices. In contrast, for CAPSULE and COMPACT FLASH, read and write costs are comparable, implying that B^+ -tree(Disk) is better for these devices. Similar conclusions holds for the time required to build the index, as shown in Figure 8(c).

Further, B^+ -tree(ST) performs better than or as good as the best of B^+ -tree(Log) and B^+ -tree(Disk) for all flash packages. The reason is that in B^+ -tree(ST), individual nodes stay in the mode which is optimal in respect to the device and workload seen by the node. With FLASHCHIP and SECURE DIGITAL, most of the index nodes stay in the *Log* mode, giving a performance comparable to B^+ -tree(Log). In fact, B^+ -tree(ST) consumes less energy than B^+ -tree(Log), since some of the read-intensive nodes near the root of the index stay in the *Disk* mode. Due to this finer granularity control of node mode, B^+ -tree(ST) consumes, for example, 10% less energy than B^+ -tree(Log) for FLASHCHIP.

Varying workload. We now investigate whether B^+ -tree(ST) can tune itself to different workloads. We use SEQUENTIAL, RANDOM, and LABDATA data under two different scenarios. First, we consider a write-intensive scenario where the index is built with no queries made. Second, we consider a read-intensive scenario where each data item is queried 10 times after the index is built. We use FLASHCHIP as the storage device for this set of experiments.

Figure 9 shows the sum of energy consumed to build the index and query it with different workloads. As discussed previously, B^+ -tree(Log) better than B^+ -tree(Disk) in the write intensive case but worse with read-intensive workloads. This is explained by the fact that retrieving a B^+ -tree(Log) node typically requires reading multiple flash pages where log entries for the node is stored. With increasing query workload, this additional cost dominates. The figures also show that B^+ -tree(ST) successfully tunes itself to perform better than both B^+ -tree(Disk) and B^+ -tree(Log), under all workloads. The results for latency, omitted for brevity, are similar [16].

Sensitivity Analysis. We now investigate how sensitive different indexing schemes are to varying workload and device properties. We vary two properties in our experiments: (1) R/W frequency: this is the ratio of read counts to write counts in the LABDATA workload, with the number of writes being fixed to 30,000, and (2) W/R cost: this is the ratio of write cost to read cost of the device, with the read cost being fixed to $1\mu J$. Figure 12 shows

the energy consumed by different indexing schemes under different workload and device properties. As shown, with increasing R/W frequency, the cost for B^+ -tree(Log) *increases slightly* while that for B^+ -tree(Disk) *decreases significantly*. In contrast, with increasing W/R cost, the cost for B^+ -tree(Log) *decreases significantly* while that for B^+ -tree(Disk) *decreases slightly*. As a result, the graphs for B^+ -tree(Disk) and B^+ -tree(Log) orient in different directions and intersect, implying the benefits of different schemes in different regions of the working space. Note that the line of intersection between two planes would have been different if we used RANDOM or SEQUENTIAL, instead of LABDATA.

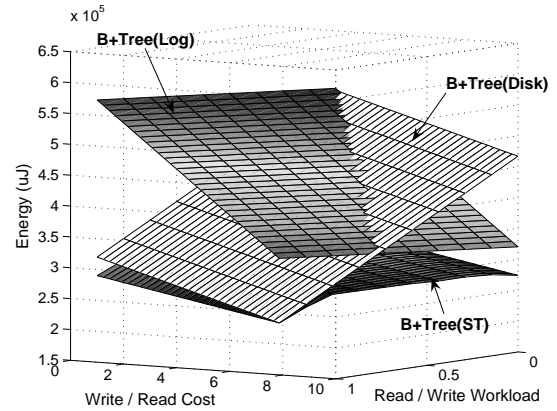


Figure 12: Sensitivity of indexing schemes to read/write ratio of workload and read/write cost of flash.

Again, note that, B^+ -tree(ST) always performs better than both B^+ -tree(Disk) and B^+ -tree(Log).

6.3 Memory Footprint

The NTT used by B^+ -tree(ST) has one of the biggest footprints of FlashDB implementation. The NTT needs to maintain information for each index node, and therefore its size increases with the number of index nodes. Figure 10 shows the size of the NTT as a function of the number of data items being indexed. Increasing the number of data items increases the number of index nodes, and hence increases the size of the NTT. However, the footprint is reasonably small even with large number of data items, showing the feasibility of using our indexing mechanism in a cluster-head, gateway, or more capable sensor node. Moreover, B^+ -tree(ST) has a smaller footprint than B^+ -tree(Log), explained by the fact that the *Disk* mode nodes in B^+ -tree(ST) do not require to maintain the addresses of pages containing log entries of the nodes. The graph

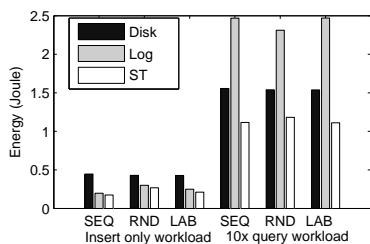


Figure 9: Energy and time with different workloads.

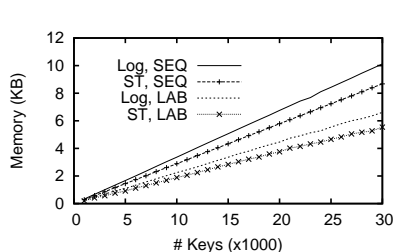


Figure 10: Memory footprint of NTT in B^+ -tree(Log) and B^+ -tree(ST) built over different lengths of LABDATA and SEQUENTIAL data.

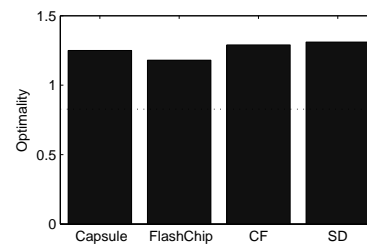


Figure 11: Node switching algorithm is within a factor of 1.3 of the optimal algorithm under real workload.

also shows that, interestingly, SEQUENTIAL data requires a larger NTT. This is due to the fact that a B^+ -tree has more index nodes if the data is inserted in increasing (or decreasing) order, instead of random order. With SEQUENTIAL, items are always inserted into the latest leaf node and leaf nodes remain unused after they are split into half. Since leaf nodes have the smallest possible size, the total number of nodes increases. The other in-memory data structure used is the Log Buffer, which has a very small overhead (one page, $\approx 512B$, in our implementation).

6.4 Performance of SWITCHMODE

As proved, our SWITCHMODE algorithm is 3-competitive, i.e., it incurs at most 3 times the cost of an optimal algorithm in the worst case scenario. To understand SWITCHMODE's behavior with a real workload, we build a B^+ -tree(ST) over the LABDATA data and capture the read/write sequences, W_{rw} , on 10 random logical B^+ -tree nodes, their transition sequence $T_{SwitchMode}$ between Disk and Log modes, and the average energy $E_{SwitchMode}$ consumed by reading/writing only these nodes. We then use the traces W_{rw} off-line to compute the optimal transition sequence T_{OPT} of the same 10 nodes. Finally, we compute the average energy E_{OPT} consumed by these nodes had they followed T_{OPT} . Figure 11 shows the optimality ratio $E_{SwitchMode}/E_{OPT}$. Clearly, the optimality ratio is far better than 3 in practice. For all types of flash, SWITCHMODE's cost is within $1.25 \times$ the optimal cost. We observed the same bound in experiments with other workloads.

7. CONCLUSION

We discussed the design space of database design for flash based storage in sensor networks. In addition to the flash characteristics, we also considered the influence of storage and retrieval workload that affects design choices such as whether to use indexing and which data structures to use for indexing. We showed that while existing log based designs proposed to address flash characteristics help improve performance, they are neither optimal, nor universally applicable across all workloads and flash devices. Our proposed self-tuning design can adapt itself to various combinations of system parameters to not only achieve the best of the performance of existing methods that are applicable in different regions of the design space but in fact improve the performance over and above the specialized methods for most regions. An analysis of the algorithm used for self-tuning was presented to evaluate its optimality. Experiments with real world data and our embedded implementation demonstrate the multiple advantages of our design.

Acknowledgement. We thank Goetz Graefe, Jim Gray and David Lomet for their valuable inputs on FlashDB design. Yossi Azar and Uri Feige helped in competitive analysis of the SWITCHMODE algorithm.

8. REFERENCES

- [1] AZAR, Y., FEIGE, U., AND NATH, S. On the work function algorithm for two state task systems. Tech. Rep. MSR-TR-2007-20, Microsoft Corporation, February 2007.
- [2] BLACK, D. L., AND SLEATOR, D. D. Competitive algorithms for replication and migration problems. Tech. Rep. CMU-CS-89-201, Carnegie Mellon University, 1989.
- [3] BORODIN, A., LINIAL, N., AND SAKS, M. An optimal online algorithm for metrical task systems. In *ACM STOC* (1987).
- [4] BURKE, J., ESTRIN, D., HANSEN, M., PARKER, A., RAMANATHAN, N., REDDY, S., AND SRIVASTAVA, M. B. Participatory sensing. In *ACM Sensys Workshop on World-Sensor-Web* (2006).
- [5] DAI, H., NEUFELD, M., AND HAN, R. ELF: an efficient log-structured flash file system for micro sensor nodes. In *ACM SenSys* (2004).
- [6] DESNOYERS, P., GANESAN, D., AND SHENOY, P. TSAR: A two tier sensor storage architecture using interval skip graphs. In *ACM Sensys* (2005).
- [7] DIAO, Y., GANESAN, D., MATHUR, G., AND SHENOY, P. Re-thinking data management for storage-centric sensor networks. In *Third Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar* (January 2007).
- [8] GIROD, L., LUKAC, M., TRIFA, V., AND ESTRIN, D. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *ACM SenSys* (2006).
- [9] GRAY, J., AND GRAEFE, G. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.* 26, 4 (1997), 63–68.
- [10] HULL, B., BYCHKOVSKY, V., ZHANG, Y., CHEN, K., GORACZKO, M., MIU, A., SHIH, E., BALAKRISHNAN, H., AND MADDEN, S. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys* (2006).
- [11] INTEL. Intel mote 2. http://www.intel.com/research/downloads/imote_overview.pdf.
- [12] KANSAL, A., XIAO, L., AND ZHAO, F. Relevance metrics for coverage extension using community collected cell-phone camera imagery. In *ACM Sensys Workshop on World-Sensor-Web: Mobile Device Centric Sensor Networks and Applications* (October 2006), pp. 12–16.
- [13] LYMBEROPOULOS, D., AND SAVVIDES, A. Xyz: A motion-enabled, power aware sensor node platform for distributed sensor network applications. In *IPSN SPOTS* (April 2005).
- [14] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys* (2006).
- [15] MICROSOFT. "sql server 2005 everywhere edition". <http://www.microsoft.com/sql/ctp.sqlserver2005everywhereedition.aspx>.
- [16] NATH, S., AND KANSAL, A. Flashdb: Dynamic self-tuning database for nand flash. Tech. Rep. MSR-TR-2006-168, Microsoft Corporation, 2006.
- [17] PON, R., BATALIN, M., GORDON, J., KANSAL, A., LIU, D., SHIRACHI, L., KAISER, W., SUKHATME, G., AND SRIVASTAVA, M. Networked infomechanical systems: A mobile wireless sensor network platform. In *IEEE/ACM IPSN-SPOTS* (April 2005).
- [18] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (1992).
- [19] SAMSUNG. Samsung K9K1G08R0B 128M x 8 bit NAND Flash Memory.
- [20] SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. *Database Systems Concepts*. McGraw Hill, 2002.
- [21] WU, C.-H., CHANG, L.-P., AND KUO, T.-W. An efficient b-tree layer for flash-memory storage systems. In *RTCSA* (2003).
- [22] ZEINALIPOUR-YAZTI, D., LIN, S., KALOGERAKI, V., GUNOPULOS, D., AND NAJJAR, V. MicroHash: An efficient index structure for flash-based sensor devices. In *USENIX FAST* (2005).
- [23] ZHANG, P., SADLER, C. M., LYON, S. A., AND MARTONOSI, M. Hardware design experiences in zebnet. In *ACM SenSys* (2004).