Hash in a Flash: Hash Tables for Flash Devices

Tyler Clemons*, S M Faisal*, Shirish Tatikonda†, Charu Aggarwal‡ and Srinivasan Parthasarathy*
*The Ohio State University
2015 Neil Ave
, Columbus, OH, USA Email: {clemonst,faisal,srini}@cse.ohio-state.edu
†IBM Almaden Research Center
650 Harry Rd
San Jose, CA 95123
Email: statiko@us.ibm.com
‡IBM T. J. Watson Center
1101 Kitchawan Road
Yorktown Heights, NY, USA
Email: charu@us.ibm.com

***Abstract*—Conservative estimates place the amount of data expected to be created by mankind this year to exceed several thousand exabytes. Given the enormous data deluge, and in spite of recent advances in main memory capacities, there is a clear and present need to move beyond algorithms that assume in-core (main-memory) computation. One fundamental task in Information Retrieval and text analytics requires the maintenance of local and global term frequencies from within large enterprise document corpora. This can be done with a counting hash-table; they associate keys to frequencies. In this paper, we will study the design landscape for the development of such an out-of-core counting hash table targeted at flash storage devices. Flash devices have clear benefits over traditional hard drives in terms of latency of access and energy efficiency. However, due to intricacies in their design, random writes can be relatively expensive and can degrade the life of the flash device. Counting hash tables are a challenging case for the flash drive because this data structure is inherently dependent upon the *randomness* of the hash function; frequency updates are random and may incur random expensive random writes. We demonstrate how to overcome this challenge by designing a hash table with two related hash functions, one of which exhibits a data placement property with respect to the other. Specifically, we focus on three designs and evaluate the trade-offs among them along the axes of query performance, insert and update times, and I/O time using real-world data and an implementation of TF-IDF.**

I. Introduction

Advances in technology have enhanced our ability to produce, and store data at very large scales. The sheer volume of data being generated is increasingly hampering our ability to manage, retrieve, and subsequently analyze such data to derive actionable knowledge and information. McKinsey Global Institute estimated that companies from all sectors in United States have at least 100 TBs of stored data per company, and many have more than 1 PB [16]. A sizable fraction of this data is textual in nature. Examples abound ranging from data produced by financial services to data collected in administrative parts of government, from customer transaction histories maintained by retail and wholesale organizations to large private enterprise document collections, from large-scale Internet repositories like Wikipedia to online content generated by social networking and microblogging sites like Facebook and Twitter.

Given this data deluge it is becoming increasingly clear that traditional data management, information retrieval and mining algorithms must be enhanced via efficient disk aware data structures. Of particular interest in this context, are recent advances in storage technology that have led to the development of flash devices. These devices have several advantages over traditional hard drives (HDD) such as lower energy requirements, faster random and sequential seek times because of a lack of moving parts [2], [5]. Due to the superior access performance, flash devices have been utilized in enterprise database applications [22], as a write cache to improve latency [12], page differential logging [20], and also as an intermediate structure to improve the efficiency of migrate operations in the storage layer [21] .However, the writes to the drive can vary in speed depending upon the scenario. Sequential writes are quite fast, though random writes, and updates can be significantly slower. The reason for this is the level of granularity of erasing and updating data on such devices. An important property of the flash drive is that it supports only a finite number of erase-write cycles, after which the blocks on the drive may wear out.

The different trade-offs in read-write speed leads to a number of challenges for information retrieval applications, especially those in which there are frequent in-place updates to the underlying data. The hash table is a widely used data structure in modern IR systems [7]. A hash table relies on a hash function to map *keys* to their associated *values*. In a well designed table, the cost of insertion and lookup requires constant (amortized) time, and is independent of the number of entries in the table. Such hash tables are commonly used for lookup, duplicate detection, searching and indexing in a wide range of domains including information retrieval. A *counting hash table* is one in which in addition to the value associated with a key, a (reference) count is also kept up to date in order to keep track of the occurrences of a specific key-value pair.

Counting hash tables are utilized directly or as a preprocessing step phase of Latent Semantic Indexing [11], Probabilistic latent semantic analysis [14], association mining[24]), and Term Frequency-Inverse Document Frequency (TF-IDF) [17]. In the database context, such tables are used for indexing (e.g. XML indexing, and selectivity estimation[1]). As a specific example, consider TF-IDF, a technique commonly used in text mining and information retrieval [26]. TF-IDF measures the importance of a particular word to a document given a corpus of documents by tracking the frequency of keywords. This technique can be used for query processing, document ranking, and document similarity. Supporting hash tables are an enormous challenge for the flash drive because they are naturally based on random hash functions and exhibit poor access locality. In the IR context, one often requires frequent in-place updates to the counts, within individual records, leading to further complications. Such updates can be expensive, unless they can be carefully batched with the use of specialized update techniques. This paper will provide an effective method for updates to the tables in flash memory, by using a carefully designed scheme which uses two closely related hash functions in order to ensure locality of the update operations.

This paper is organized as follows. The remainder of this section will discuss the properties of the flash drive which are relevant to the effective design of the hash table. We will then discuss related work and the contributions of this paper. In **Section II**, we will discuss our different techniques. **Section III** contains the experimental results. The conclusions and summary are contained in **Section IV**.

A. Properties of the Flash Drive

The solid state drive (SSD) is implemented with the use of Flash memory. The most basic unit of access is a page which contains between 512 and 4096 bytes, depending upon the manufacturer. Furthermore, pages are organized into *blocks* each of which may contain between 32 and 128 pages. The data is read and written at the level of a page, with the additional constraint that when any portion of data is overwritten at a given block, the entire block must be copied to memory, erased on the flash, and then copied back to the flash after modification. Moreover, flash drives can only support a limited number of erasures (between 10,000 and 100,000 erasures) after which the blocks may degrade and they may not be able to support further writes. Management of the flash device blocks is performed automatically by the software known as the Flash Translation Layer (FTL) on the flash drive. Thus, even a small random update of a single byte could lead to an erase-write of the entire block. Similarly, an erase, or clean, can only be performed at the block level rather than the byte level. Since random writes will eventually require erases once the flash device is full, it implies that such writes will require block level operations. On the other hand, sequential writes on the flash are quite fast; typically sequential writes are two orders of magnitude faster than random writes.

B. Related Work

Rosenblum and Ousterhout proposed the notion of log-structured disk storage management [25] and mechanisms similar to log-structured file systems are adopted in modern SSDs either at the level of FTL or at the level of file system to handle issues related to wear-leveling and erase-before-write [8], [15], [19], [18], [28]. As we discuss later, some of our buffering strategies are also inspired from log-structured file systems. There have been hash tables designed with SSDs including the work presented in [3], [4], [9], [29], [10]. Our designs differ from previous work because they are optimized for counting hash tables, and our primary hash table is completely resident on the SSD; other designs store the primary hash table on an HDD and utilize the SSD as a cache.

C. Contributions of this paper

In this paper, we design a *counting* hash table for SSDs that maintains frequencies using a combination of memory and disk buffering schemes. To our knowledge this has not been addressed thus far. In this work, we make the following specific contributions – (i) We propose a mechanism to support large *counting* hash tables on SSDs via a two-level hash function, which ensures that the random update property of flash devices is effectively handled, by localizing the updates to SSD; (ii) We devise a novel combination of memory- and disk- based buffering scheme that effectively addresses the problems posed by SSDs (random writes, write endurance). While the memory-resident buffer leverages the fast random accesses to RAM, the disk-resident buffer exploits fast read performance and fast sequential/semi-random write performance of SSDs; (iii) We perform a detailed empirical evaluation to illustrate the effectiveness of our approach by demonstrating the traditional IR algorithm TF-IDF using our hash table.

II. A Flash-Friendly Hash Table

The major property of a hash table is that its effectiveness is highly dependent upon updates which are distributed randomly across the table. On the other hand, in the context of a flash-device, it is precisely this randomness which causes random access to different blocks of the SSD. Furthermore, updates which are distributed randomly over the hash table are extremely degrading in terms of the wear properties of the underlying disk. This makes hashing particularly challenging for the case of flash devices.

Hash table addressing is of two types: *open* and *closed*, depending upon how the data is organized and collisions are resolved. These two kinds of tables are as follows:**(i) Open Hash Table:** In an open hash table, each slot of the hash table corresponds to multiple data entries. Each entry of the collection is a key-frequency pair. **(ii) Closed Hash Table:** Each slot contains a single key-frequency pair. However, since multiple pairs cannot be mapped onto the same entry, we need a collision resolution process i.e. when a hashed object maps onto an entry which has already been filled. A common strategy is to use *linear probing* in which we cycle through successive entries of the hash table until we either find an instance of the object itself (and increase its frequency), or we find an empty slot in which we insert the new entry. We note that a fraction of the hash table (typically at least a quarter) needs to to be empty in order to ensure that the probing process is not a bottleneck. The fraction of hash table which is full is denoted by the load factor f. It can be shown that $1/(1-f)$ entries of the hash table are accessed on the average in order to access or update an entry.

In this paper, we will use a combination of the open and closed hash tables in order to design our update structure. We will use a closed hash table as the *primary hash table* which is stored on the (Solid State) drive, along with a *secondary hash table* which is open and available in main memory. We assume that the primary hash table contains q entries, where q is dictated by the maximum capacity planned for the hash table for the application at hand. The secondary hash table contains $\lceil q/r \rceil$ entries where $r << q$. The hash function for the primary and secondary hash tables are denoted by $g(x)$ and $s(x)$, and are defined as follows:

$$g(x) = (a \cdot x + b)\text{mod}(q) \tag{1}$$

$$s(x) = ((a \cdot x + b)\text{mod}(q))\text{div}(r) \tag{2}$$

In general, the scheme will work with any pair of hash functions $g(x)$ and $s(x)$ which satisfy the following relationship:

$$s(x) = g(x)\text{div}(r) \tag{3}$$

It is easy to see that the entries which are pointed to by a single slot of the memory-resident table are located approximately contiguously on the drive-resident (closed) table, because of the way in which the linear probing process works. This is an important observation, and will be used at several places in ensuring the efficiency of the approach. Linear probing essentially assumes that items that collide onto the same hash function value will be *contiguously located in a hash table with no empty slots between them*. Specifically, the mth slot on the secondary table, corresponds to entries starting from $r \cdot (m-1) + 1$ up to entry $r \cdot m$ in the primary table. We note that *most* entries which would be pointed to by the mth slot of the secondary table would also map onto the afore-mentioned entries in the primary table, though this would not always be true because of the overflow behavior of the linear probing process beyond these boundaries.

A. Desirable Update Properties of an SSD-based Hash-Table

A naive implementation of a hash table will immediately issue update requests to the hash table as the data points are received. The vast majority of the write operations will be random page level writes due to the lack of locality, which is inherent in hash function design. As mentioned before, the cost of such operations will also increase the cost of cleans and random writes. A desirable property for a hash table would be *block-level updates* and *semi-random writes*. The *block-level update* refers to the case when there are multiple updates written to a block, and they are all accomplished at one time. If there are k updates written to a block, we should combine them into one block-level write operation. This can reduce the number of cleans from k to one. The semi-random writes refer to the fact that the updates to a particular block are in the same order as they are arranged on the block, even though updates to different blocks may be interleaved with one another.

B. Hash table designs

Recall that we combine an open hash table in main memory with a closed-hash table on the SSD. This open (or secondary) hash table is typically implemented in the form of a RAM buffer denoted as H_R. The RAM buffer will contain updates for each block of the SSD and execute batch updates to the primary hash table on disk, or *data segment* (denoted by H_D), at the block level. This approach can reduce block level cleaning operations.

C. Memory Bounded Buffering

The overall structure of the common characteristics of the hash table architectures presented in this paper is illustrated in Figure 1. We refer to this scheme as *Memory Bounded Buffering* or *MB*. The RAM buffer, H_R , in the diagram is an open (or secondary) hash table and the data segment is a closed (or primary) hash table. There are s slots, each of which corresponds to a *block* in the data segment. The maximum capacity of the data segment is q pages, r pages per block and g entries per page. Thus, the number of slots in the secondary hash table, s, must be equal to q/r. Updates are flushed onto the SSD one block at a time. Because of the relationships between the hash functions of the primary and secondary table, the *merge* process of a given list requires access to only a particular set of SSD blocks which can be maintained in main memory during the merging process.

D. Memory and Disk Bounded Buffering

Since H_R is main-memory resident, it is typically restricted in size. Therefore, a second buffer can be implemented on the SSD itself. This new segment is referred to as the *change segment* or S_E. The change segment acts as a second level buffer. When H_R exceeds its size limitations, the contents are sequentially written to the change segment at the page level starting from the first available page in an operation known as *staging*. When full, the change segment *merges* with the data segment and begins from the top of the change segment. A page in the change segment may contain updates from multiple blocks because pages are packed with up to g entries irrespective of their slot origin. Thus, the change segment is organized as a log structure that contains the flushed updates of the RAM buffer. This takes advantage of the semi-sequential write performance of the SSD and increases the lifetime of the SSD. The space allocated to the change segment is in addition to the space allocated to the data segment. This hash table (with change-segment included) is illustrated in Figure 2. It is important to note that a *stage()* operation differs from a *merge()* operation in two ways, specifically, stages write at the page level while merges operate at the block level. Furthermore, stages involve updates to the change segment while merges involve updates to the data segment.

There are two types of architectures for the change segment. In the first design, the change segment S_E is viewed as a collection of blocks where each block holds updates from multiple lists from H_R. In other words, multiple blocks in the data segment are mapped to a single block in the change segment. We arrange the change segment in a way such that each change segment block holds the updates for k data segment blocks. The value of k is constant for a particular instantiation of the hash-table, and can be determined in an application-specific way. For an update-intensive application, it is advisable to set k to a smaller value at the expense of SSD space. When a particular change segment block is full, we merge the information in the change segment to the data segment blocks. By adding the change segment, we are providing a more efficient buffering mechanism. Staging a segment is more efficient than merging it because the change segment is written onto the SSD with a straightforward sequential write, which is known to be efficient for the SSD. This approach is called *Memory Disk Buffering* or *MDB*. In this variation of the

MDB scheme, (which we henceforth will refer to as *MDB-L* for *MDB-Linear*) the space allocated for the change segment is viewed as a single large monolithic chunk of memory without any subdivisions. This view resembles a large log file. Thus, the change segment blocks are not assigned to k data segment blocks. The writes to the change segment are executed in FCFS fashion. This type of structure mimics a log-structured file system and fully takes advantage of the SSD strength in sequential writes. We maintain a collection of pointers to identify the ranges, measured in pages, that a particular slot in the RAM buffer has been staged. These pointers are similar to the indexing information [25] maintained in log-structured file systems that helps in reading the files from the log efficiently. A merge operation is triggered when the change segment is full. The collection of pointers can be used to identify the pages a particular block was staged to. This process produces random reads on the change segment because the ranges span multiple stage points. During a merge, each page will be requested by each data segment block that has entries staged onto it. After all of the pages for a particular data segment block are read from the change segment, the entries are merged with the corresponding data segment block.

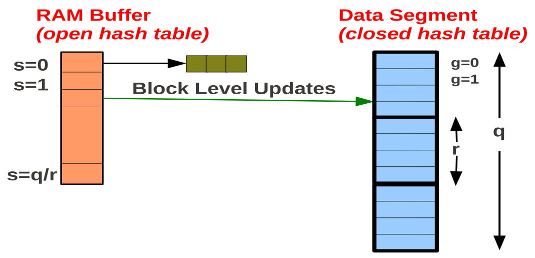


Fig. 1. Hash Table with RAM buffer

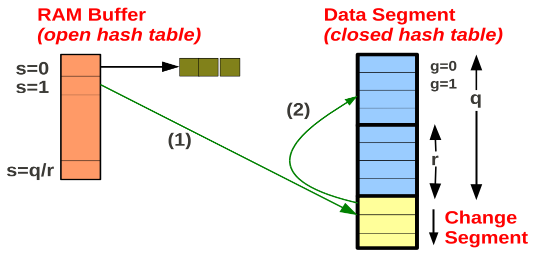


Fig. 2. Hash Table with RAM buffer and Change Segment

E. Element Insertion and Update Process

The element insertion process is designed to perform individual updates on the memory-resident table only, since this can be done in an efficient way. Such changes are later rolled on to the RAM buffer (which is in turn rolled on to the change segment for some of the schemes). An update or insertion in the TF-IDF context corresponds to increasing the occurrence count for a term. For each incoming record x, we first apply the hash function $s(x)$ in order to determine the slot to which the corresponding entry belongs. We then determine if the key x is present inside the corresponding slot $s(x)$. If the element is found, then we increase its frequency. The second case is when the key x is not contained inside the buffer which is pointed to by the slot $s(x)$. In such a case, we add the key x as a new element to the H_R. The size of the H_R increases in this case. If H_R has grown too large, it is flushed either directly onto the change segment or the SSD itself, depending upon whether or not the change segment is implemented in the corresponding scheme. Because of the relationship between the hash functions of H_R and the SSD based hash table, such an update process tends to preserve the locality of the update process, and if desired, can also be made to preserve semi-random write properties.

During the insertion process of new items, linear probing may occur because H_D is a closed hash table. If the linear probing process reaches the end of the current SSD block, then we *do not* move the probe onto the next block. Rather, an overflow region is allocated within the SSD table which takes care of probing overflows beyond block boundaries. The last index of the last page of an SSD block is a pointer referring to the overflow region. The entry that was resident at this position now resides in the conflict region alongside the newly inserted entry. Thus, the *data segment* is a collection of blocks with logical extensions. The overflow region, a collection of SSD blocks, is allocated when the hash table is created and it's size is a user parameter. If an item is deleted from the data segment, it can either be removed or its frequency can be set to zero. This feature is handled during the *merge()* operation. Deleted items are ignored and removed during the rewriting of the data segment block.

F. Query Processing

In the context of TF-IDF, a query can be thought of as a request for a word's occurrence count. In the simple hash table, queries are fulfilled by an I/O request to the data segment. However, in our proposed designs the corresponding entry may be found either in the change segment or the RAM buffer. Therefore, the query processing approach must search the change segment and the RAM buffer in addition to the data segment. Thus, the frequency of a queried item is the total frequency found in the change segment, RAM buffer, and data segment. The search of the RAM buffer may be inexpensive because it is in main memory. On the other hand, access to the change segment requires access to the SSD.

For the case of the data segment, the query processing approach is quite similar to that of standard hash tables. A hash function is applied to the queried entry in order to determine its page level location inside of the data segment. If the entry is found, the frequency is returned. If the item is not found, linear probing begins because the disk hash table is a closed hash. Linear probing halts if the entry is found or an empty entry was discovered. The query processing of the change segment requires locating the entry. The location of the entry may reside in multiple segments due to repeated flushing of the RAM buffer. Recall that *MDB* partitions the change segment. When a RAM bucket is staged, it is always written to the same change segment block. We locate the appropriate change segment block and bring it into memory to be searched. In *MDB-L*, RAM buckets can reside on multiple pages, and thus we must issue random page reads. We expect *MDB-L* to be faster because of page level access.

III. Experiments

In this section we present an empirical analysis of the hash table designs discussed in the previous sections. We evaluate the performance of the three main schemes discussed in this article, namely *MB*, *MDB* and *MDB-L*. Broadly, our objectives are to understand the I/O overheads of various schemes and their query performance while executing TF-IDF. Additionally, since SSD disks permit a limited number of clean operations, it is also important to quantify the wear rate of the devices. We begin with a discussion of the experimental setup.

A. Experimental Setup

To evaluate our hash table configurations, we used the DiskSim simulation environment [6], managed by Carnegie Mellon University; and the SSD Extension for this environment created by Microsoft Research [23]. We operated Disksim in slave mode. Slave mode allows programmers to incorporate Disksim into another program for increased timing accuracy. In a simulated environment as opposed to using a true SSD, we are able to quantify the number of clean operations and the exact I/O cost of our methods executing TF-IDF. We conducted our experiments on three different configurations of the latest representative NAND flash SSDs from Intel (see Table I for details). Among these, two SSDs are MLC (Multi-Level Cell) and the other is SLC (Single-Level Cell) based SSDs. We have chosen from both MLC and SLC because of their differing characteristics. While MLC provides much higher data density and lower cost (which makes it more popular), it has a shorter lifespan and slow read/write performances. SLC, on the other hand, has faster read/write performances and a significantly longer lifespan. SLCs also entail lower internal error rate making them preferable for higher performance, high-reliability devices [13]. All hash table experiments involve inserting, deleting, and updating key value pairs. The size of the RAM buffer is parameterized on the size of the data segment and expressed as a percentage. The rationale here is that we believe that an end application may need to create multiple hash tables on the same SSD. Moreover, the characteristics of access may vary across applications (i.e. one may want different RAM buffer sizes for each hash table). The change segment is likewise parameterized and the overflow segment for all experiments was set to a minimal value (one block) since this was found to be sufficient. Key-value pairs are integer pairs. We conducted our experiments on a DELL Precision T1500 with an Intel ® Core ™ i7 CPU 860@2.8GHz with 8GBs of memory running Ubuntu 10.04. Our code was implemented in C++. The H_R data structure utilized the C++ Standard Template Library [27] for its implementation. The RAM buffer buckets that correspond to data segment blocks are arranged inside a C++ vector and their indexes correspond to their placement on the data segment. For example, the first block inside the data segment corresponds to the first block in the RAM buffer. The data segment can be viewed as an array logically divided into blocks and further divided into pages.

B. TF-IDF

To demonstrate the efficacy of our methods, we implemented the TF-IDF algorithm, see **Section I**, using our hash table designs and noted the I/O times for frequency updates and queries.

	SSD Configurations		
	MLC-1	**MLC-2**	**SLC**
Capacity	40GB	80GB	32GB
Flash Memory	MLC	MLC	SLC
Page Size (KB)	4	4	4
Sustained Sequential Read (MB/s)	Upto 170	Upto 250	Upto 250
Sustained Sequential Write (MB/s)	Upto 35	Upto 70	Upto 170
Read Latency (μs)	65	65	75
Write Latency (μs)	110	85	85
Cost (USD)	109.99	224.99	377.79

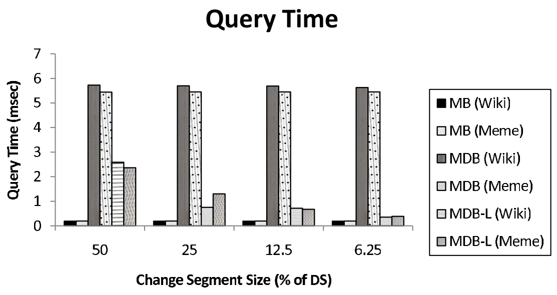
TABLE I. SSD Configurations

C. Data Sets

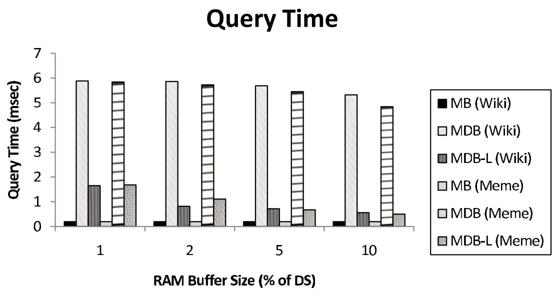
We use two datasets: a Wikipedia and MemeTracker dump, which are essentially large *text* files. **Wiki:** The first data set we use is a collection of randomly collected Wikipedia articles. We chose $100,000$ random wikipedia articles collected from Wikipedia's publicly available dump[1]. Our $100,000$ random articles were approximately $1GB$ in size. This dataset contains $136,749,203$ tokens (keywords) with $9,773,143$ unique entries. For the testing of this data set, our hash table was set to $100MB$. On a 128 page per block SSD this amounts to approximately 205 SSD blocks allocated to the data segment. To evaluate I/O performance during inserts or updates[2] we simply insert (or update) tokens (corresponding counts) into the hash table. Statistics and times for various operations (cleans, merges, stages etc.) are computed and discussed shortly. To evaluate query performance we first processed 35 million tokens. Subsequently, roughly 100 million words were inserted. Simultaneously with inserts, we also issued a million queries interleaved randomly across inserts. A query is a hash table lookup. In the TF-IDF context, this corresponds to *"how frequent is a keyword"* which allows us to compute the TF-IDF score of a keyword. Of these queried items, on average (spread across 10 different random workloads) $933,139$ of them were present inside the hash table at query time. **Meme:** The second data set we report is the MemeTracker dataset discussed in **Section 1**. We downloaded the *August 2008* dataset. We found $17,005,975$ unique entries and $402,005,270$ total entries. Since this dataset is slightly larger our hash table size was $200MB$. On a 128 page per block SSD this translates to approximately 410 SSD blocks allocated to the data segment. I/O performance is evaluated by inserting (or updating) tokens just as with the Wiki dataset. For query performance, the first 130 million words were inserted into the hash table. Subsequently, the remaining 270 million words were interleaved with about one million queries. Of these queried items, (spread across 10 random workloads) $959,731$ of them were found inside the hash table.

[1]See http://dumps.wikimedia.org/

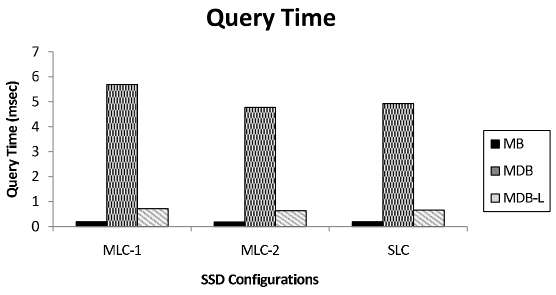
[2]As noted earlier deletes are handled as inserts with a negative count.



(a) Query times for varying change segment sizes

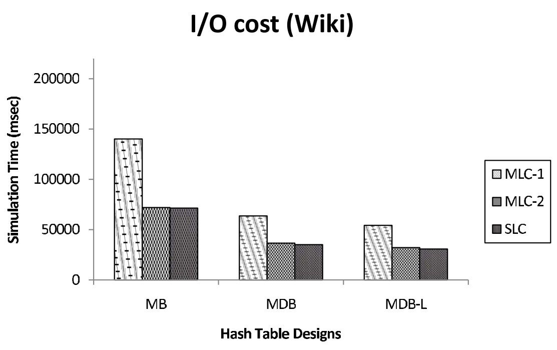


(b) Query times for varying RAM buffer

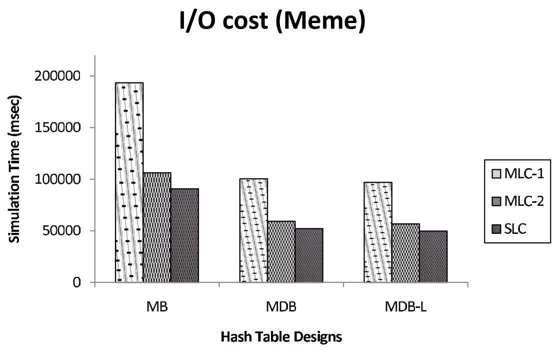


(c) Query times for all SSD configurations

Fig. 3. Query Times

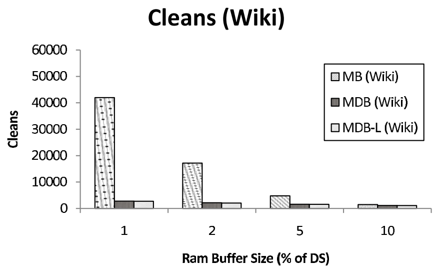


(a) I/O cost for *Wiki* dataset

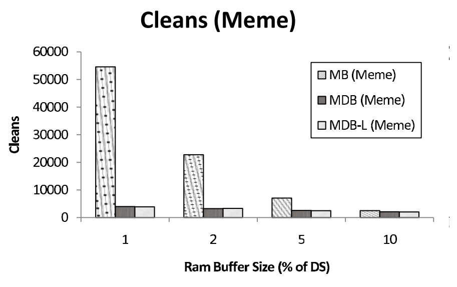


(b) I/O for *Meme* dataset

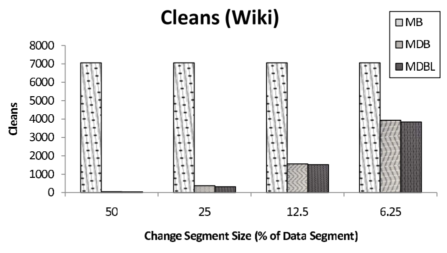
Fig. 4. I/O Costs



(a) Cleans for *Wiki* dataset



(b) Cleans for *Meme* dataset



(c) Cleans for varying change segment sizes

Fig. 5. Cleans

D. Query Time Performance

In each graph, the Y-axis is the average time per query in milliseconds. Results on both Wiki and Meme are provided in Figure 3. The main trends we observe include: i) the query time for *MB* are quite low (does not have a change segment); ii) the query time for *MDB* is quite high and does not drop significantly with reduction in change segment size; iii) the query time for *MDB-L* improves dramatically with a reduction in the change segment size; and iv) query times for Meme are marginally lower than the query times for Wiki for both *MDB* and *MDB-L* . These trends can be explained as follows. Query costs for MB are essentially fixed since they essentially have to combine the counts from the memory buffer (negligible) and require typically a page read to access the requisite information from the data segment. Query costs for *MDB* require consolidation of information from the memory buffer (negligible) and from the change segment (expensive – dominated by block level reads) and the data segment (usually a single page read). Query costs for *MDB-L* require consolidation across the memory buffer (negligible), the change segment (typically requiring a few page reads which are significantly reduced as the size of the change segment is reduced) and the data segment (usually a single page read). This is reflected in our first experiment, shown in Figure 3(a) for MLC-1, in which we varied the change segment while fixing the RAM buffer to 5%. With regards to the difference between Wiki and Meme query times, upon drilling down into the data, we find that on average there are 11.5% more page reads for Wiki. This may be an artifact of the linear probing costs within both datasets, given the fact that the ratio of number of unique tokens to hash table size is slightly higher for Wiki. For the second experiment, again on the MLC-1 configuration, we fixed the change segment to 12.5% and varied the the RAM buffer for both datasets(see Figure 3(b)). We observe that with an increase in RAM buffer size that : i) *MB* shows a negligible change in average query time; ii) *MDB* shows decrease in average query time; iii) *MDB-L* shows a significant decrease in average query time performance; and iv) query times on Meme are typically faster than those on Wiki. To explain these trends we should first note that increasing RAM buffer size has the general effect of reducing the number of stage operations, and thus the average size of the amount of useful information within the change segment. Thus the time it takes to consolidate the information within the change segment in order to answer the query, is on average lower, for both *MDB* and *MDB-L*. For *MDB-L* the improvement is more marked because fewer page reads are required. The explanation for why query times are lower for Meme are similar to what we observed for the previous experiment. The third experiment we performed on query time performance was to evaluate the performance of the three SSD configurations on the Wiki dataset shown in Figure 3(c). Here the RAM buffer was set to 5% and the Change Segment was set to 12.5%. The results are along expected lines in that average query times are slightly better on MLC-2 and SLC over MLC-1 for the *MDB* method. The superior read performance for both page level and block level operations are the primary reason. This difference is marked in the case of *MDB* but for both *MDB-L* and *MB* the difference is negligible. *MDB* requires a block level read for a query and the performance difference for this type of operation is more pronounced for MLC-2 and SLC, over when compared with MLC-1. To conclude we should reiterate that the query performance times we observe here are for our update-intensive query workload where we interleaved queries with inserts (averaged over multiple runs). In this environment the query time performance of MB is always the fastest. For *reasonable parametric settings MDB-L typically approaches MB in performance while MDB is always an order of magnitude worse in terms of performance*. We should note that we also evaluated query times for all three methods in more stable settings (few updates/inserts) In such a *stable setting we found that the query times for all three methods was identical*. The query cost essentially boils down to a page read or two on the data segment (since the change segment is empty and does not factor). Furthermore, MDB is bounded by a single block read while the query time of MDB-L may vary. However, as our results indicate, the pointer guided page level accesses of MDB-L provide efficient read access that outperforms MDB.

E. I/O Performance

In this section we examine the I/O performance of the three strategies. To ignore the impact of queries in this section, our workloads for both datasets simply insert all the tokens or words into their respective hash tables. In our first experiment, we report overall I/O cost from the perspective of the SSD for the three SSD configurations for both Wiki (see Figure 4(a)) and Meme (see Figure 4(b)). The RAM buffer is set to 5% and the change segment is set to 12.5% in this experiment. The main trend we observe are that both *MDB-L* and *MDB* require comparable yet significantly lower I/O costs than *MB*. This is primarily attributable to the presence of the change segment which enables *sequential (MDB-L) or semi-random writes (MDB)*. We trade block level operations for page level operations. In our analysis of block and page level operations, *MDB-L* had more page level operations and fewer block level operations than *MDB* due to the linear change segment. This gives *MDB-L* an edge of over *MDB*. in our deeper analysis we verified some of the intuitive ideas of our designs: as we increase the size of the change segment, we decrease the number of block level *merge()* operations and as we increase the RAM buffer, we reduce the number of *stage()* operations. Additionally, as we shall see shortly, *MB* requires a large number of erasures which also contribute to the overall I/O cost. Another trend we observe is that among the SSD configurations SLC and MLC-2 offer comparable performance with a slight edge to SLC. MLC-1 is quite a bit slower. This is primarily attributable to the superior write bandwidth of SLC and MLC-2. Finally we observe that the overall I/O times are higher for Meme over Wiki (larger dataset and larger hash table). Not shown in our reports are the performance measures for a hash table without the use of a buffer. The advantage of this scheme is fast query times because queries are only page level reads on the data segment. However, results show that such a hash table would induce 1,680,323 cleans for the Wiki dataset and 6,669,932 cleans for the Meme dataset. The I/O performance are on the order of 615 times slower on the Wiki dataset and 1500 times slower for the Meme dataset for reported times for the results in Figure 4. This increase is caused by cleaning time and random page writes. It is clear that there is a benefit from our designs. Summing up the I/O performance it is fair to say that for most reasonable parameter settings *MDB* and *MDB-L* significantly outperform *MB* in terms of the cost of I/O from the perspective of the flash device. Additionally it should be noted that the merging operation within both *MDB* and *MDB-L* will happen completely within the SSD (allowing for an overlap of CPU operations – not reflected in any of the experiments) whereas a merge for *MB* and staging for the other two methods will require some CPU intervention. Also note that an *MB* merge operation is significantly more expensive than an *MDB* or *MDB-L* stage operation (random writes versus semi-random/sequential writes).

F. Cleans

In our next experiment we take a closer look at the number of clean operations required by these methods for both datasets (see Figure 5). Our graphs display the variation of RAM buffer size for both datasets and the variation of change segment size for the Wiki dataset along the Wiki dataset for the X-axis. The Y-axis is the amount of erasures. The main trends and explanations for these trends are: i) the number of cleans goes down with increasing RAM buffer sizes since there are fewer stages and merges as shown in Figure 5(a); ii) the number of cleans is significantly higher for *MB* compared to that of the other two methods because the change segment provides an extra level of buffering for *MDB* and *MDB-L* as shown in Figure 5(b); iii) the number of cleans increases for *MDB* and *MDB-L* as we decrease the size of the change segment because the change segment fills more often and thus there are more merges. *MB* does not use the change segment so it stays a constant value; and iv) the number of cleans for *MDB* and *MDB-L* are very similar with *MDB-L* being slightly better. The reduction for *MDB-L* is clearly attributable to the linear change segment design.

IV. Conclusions

Hash tables pose a challenge for SSDs because of their random write access patterns. Traditional IR algorithms, such as TF-IDF rely on hash tables and may suffer under a poor hash table design. TF-IDF requires extensive updating of term frequencies. Updating an entry inside of a disk hash table may trigger an entire erasure of an SSD block. Repeatedly updating a hash table can be detrimental to the limited lifetime of the underlying SSD. A simple hash table *without* buffering can be implemented. It has superior query time but it induces a substantial amount of cleans and I/O cost. From our experiments, we believe that an SSD friendly hash table will have a RAM buffer and a disk based buffer that supports semi-random writes. These features will increase the locality of updates and reduce the I/O cost of the hash table for both low and high end SSDs. Overall our results reveal that when one accounts for both I/O performance and query performance, *MDB-L* seems to offer the best of both worlds on the workloads we evaluated for reasonable parameter settings of change segment size and RAM buffer size. Using MDB-L, we observe 1500 times speedup in execution time and over 6 million fewer erasures when compared to a naive implementation. In the future, we would also like to extend our design to hash functions that do not rely on the *mod* operator (e.g. extendible hashing) and examine various checkpointing methods.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX*, 2008.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. *USENIX*, 2010.

[4] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[5] A. Birrell, M. Isard, T. C., and T. Wobber. A Design for High Performance Flash Disks. *SIGOPS*, 2007.

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, G. R. Ganger, and et al. The disksim simulation environment version 4.0 reference manual, May 2008.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms 2nd edition*. The MIT Press, 2001.

[8] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys*, 2004.

[9] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory . In *USENIX*, 2010.

[10] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. In *VLDB*, 2010.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 1990.

[12] G. Graefe. The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. *DaMoN*, 2007.

[13] L. Hewlett-Packard Development Company. Solid state drive technology for ProLiant servers: Technology brief, October 2008.

[14] T. Hofmann. Probabilistic latent semantic indexing. SIGIR '99, 1999.

[15] C. Hyun, Y. Oh, E. Kim, J. Choi, D. Lee, and S. Noh. A Performance Model and File System Space Allocation Scheme for SSD. *MSST*, 2010.

[16] M. G. Institute. Big data: The next frontier for innovation, competition, and productivity.

[17] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[18] H. Kim and U. Ramachandran. FlashLite: A User-Level Library to Enhance Durability of SSD for P2P File Sharing. In *ICDCS*, 2009.

[19] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *ICCE*, 2002.

[20] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *SIGMOD*, 2010.

[21] I. Koltsidas and S. D. Viglas. Flashing Up the Storage Layer. *VLDB*, 2008.

[22] S. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. *SIGMOD*, 2008.

[23] Microsoft. SSD Extension for DiskSim Simulation Environment, March 2009.

[24] R. Relue and X. Wu. Rule Generation With the Pattern Repository. In *ICAIS*, 2002.

[25] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *TOCS*, 1992.

[26] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *INFORMATION PROCESSING AND MANAGEMENT*, pages 513–523, 1988.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition, June 1997.

[28] D. Woodhouse. JFFS: The journalling flash file system. Red Hat, 2001.

[29] D. Zeinalipour-Yatzi, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *In FAST*, 2005.