Design Patterns for Tunable and Efficient SSD-based Indexes

Ashok Anand†, Aaron Gember-Jacobson*, Collin Engstrom*, Aditya Akella*
†Instart Logic *University of Wisconsin-Madison
†ashok.anand@gmail.com *{agember,engstrom,akella}@cs.wisc.edu

ABSTRACT

A number of data-intensive systems require using random hash-based indexes of various forms, e.g., hash tables, Bloom filters, and locality sensitive hash tables. In this paper, we present general SSD optimization techniques that can be used to design a variety of such indexes while ensuring higher performance and easier tunability than specialized state-of-the-art approaches. We leverage two key SSD innovations: a) rearranging the data layout on the SSD to combine multiple read requests into one page read, and b) intelligently reordering requests to exploit inherent parallelism in the architecture of SSDs. We build three different indexes using these techniques, and we conduct extensive studies showing their superior performance, lower CPU/memory footprint, and tunability compared to state-of-the-art systems.

Categories and Subject Descriptors

C.2.m [**Computer Communication Networks**]: Miscellaneous; D.4.2 [**Operating Systems**]: Storage Management; E.2 [**Data**]: Data Storage Representations

Keywords

Solid state drives (SSDs), hashtables, bloom filters, memory efficiency, CPU efficiency, parallelism

1. INTRODUCTION

Data-intensive systems are being employed in a wide variety of application scenarios today. For example, key-value systems are employed in cloud-based applications as diverse as e-commerce and business analytics systems, and picture stores; and large object stores are used in a variety of content-based systems such as network deduplication, storage deduplication, logging systems, and content similarity detection engines. To ensure high application performance these systems often rely on random hash-based indexes whose specific design may depend on the system in question. For instance, WAN optimizers [5, 6], Web caches [4, 7], and video caches [2] employ large streaming hash tables. De-duplication systems [28, 30] employ Bloom filters. Content similarity engines and some video proxies [2, 11] employ locality sensitive hash (LSH) tables [24]. Given the volume of the underlying data, the indexes often span several 10s to 100s of GB, and they continue to grow in size.

Across these systems, the index is the most intricate in design. Heavy engineering is often devoted to ensure high index performance at low cost and low energy footprint. Most state-of-the-art systems [14, 15, 21, 25] advocate using SSDs to store the indexes, given flash-based media's superior density, 8X lower cost (vs. DRAM), 25X better energy efficiency (vs. DRAM or disk), and high random read performance (vs. disk) [25]. However, the commonality ends here. The conventional wisdom, which universally dictates index design, is that domain- and operations-specific SSD optimizations are necessary to meet appropriate cost-performance trade-offs. This poses two problems: (1) *Poor flexibility:* Index designs often target a specific point in the cost-performance spectrum, severely limiting the range of applications that can use them. It also makes indexes difficult to tune, e.g., use extra memory for improved performance. Finally, the indexes are designed to work best under specific workloads; minor deviations can make performance quite variable. (2) *Poor generality:* The design patterns employed apply only to the specific data structure on hand. In particular, it is difficult to employ different indexes in tandem (e.g., hash tables for cache lookup alongside LSH tables for content similarity detection over the same underlying content) as they may employ conflicting techniques that result in poor SSD I/O performance.

Our paper questions the conventional wisdom. We present different indexes that all leverage a common set of novel SSD optimizations, are easy to tune to achieve optimal performance under a given cost constraint, and support widely-varying workload patterns and applications with differing resource requirements; yet, they offer better IOPS, cost less, and consume lower energy than their counterparts with specialized designs.

We rely on two key innovations. (1) We leverage a unique feature of SSDs that has been overlooked by earlier proposals, namely, that the internal architecture of SSDs offers parallelism at multiple levels, e.g., channel-, package-, die-, and plane-level. Critically, the parallelism benefits are significant only under certain I/O workloads. Our key contribution lies in identifying these parallelism-friendly workloads and developing a set of design patterns for encapsulating the input workload for an index into SSD parallelism-friendly forms. (2) Based on the design patterns, we develop a new primitive called *slicing* which helps organize data on the SSD such that related entries are co-located. This allows us to combine multiple reads into a single "slice read" of related items, offering high read performance. We show how our design patterns inform slice size, the number of slices to co-locate at a particular SSD block, and the techniques to use for reading from and writing to slices. A

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2658260.2658270.

key feature of slicing is that slice size/composition (i.e., how many elements constitute a slice) offers simple knobs to trade off I/O performance for the memory overhead of any index data structure.

In §4, we conduct several experiments to profile the internal parallelism behavior on a desktop-grade SSD to identify parallelism-friendly I/O patterns, and derive the appropriate design patterns that guide the composition, configuration and use of slices. Then, we present the design of three random-hash based indexes that leverage our design patterns and slicing: a streaming hash table called SliceHash (§5), large Bloom filters called SliceBloom, and locality-sensitive hash tables called SliceLSH (§6).

Our index designs can be sketched as follows: We use small in-memory data structures (hash tables, Bloom filters, or LSH tables, as the case may be) as *buffers* for insert operations to deal with the well-known problem of slow random writes on SSDs. When full, these are flushed to the SSD; each of these flushed data structures is called an "incarnation". A similar approach has also been used in state-of-the-art techniques, e.g., [14, 25], to deal with slow random writes. However, they need to maintain complex metadata for lookups, which imposes high memory overhead or CPU cost. In contrast, we use a simple reorganization of data on the SSD such that all related entries of different incarnations are located together in a slice, thereby optimizing lookup and eliminating the need for maintaining complex metadata. We show that this frees memory and compute resources for use by higher layer applications. Further, based on an understanding of the SSD's writing policy, we appropriately reorder lookups, without violating application semantics, to distribute them uniformly across different channels and extract maximal parallelism benefits.

Our parallelism-centered design patterns and the slicing primitive together offer good performance at relatively low CPU or memory overhead in comparison to state-of-the art techniques. We show that our design techniques facilitate extending the indexes to use multiple SSDs on the same machine, offering linear scaling in performance while also *lowering* per-key memory overhead. State-of-the-art techniques cannot be scaled out in a similar fashion.

We build prototype indexes using a 128GB Crucial SSD and at most 4GB of DRAM. We conduct extensive experiments under a range of realistic workloads to show that our design patterns offer high performance, flexibility, and generality. Key findings from our evaluation are as follows: On a single SSD, SliceHash can provide 69K lookups/sec by intelligently exploiting parallelism which is 1.5X better than naively running multiple lookups in parallel. Lookup performance is preserved even with arbitrarily interleaved inserts, whereas state-of-the-art systems take up to a 30% performance hit. SliceHash has low memory footprint and low CPU overhead, yet it provides high lookup performance. Furthermore, SliceHash can be tuned to use progressively more memory (from 0.27B/entry to 1.1B/entry) to scale performance (from 70K to 110K ops/s) for mixed (50% lookup, 50% insert) workloads. When leveraging 3 SSDs in parallel, SliceHash's throughput improves to between 207K (lookup-only) and 279K (lookup/insert) ops/sec. SliceBloom performs 15K ops/sec with a mixed lookup/insert workload, whereas the state-of-the-art [22] achieves similar performance on a high-end SSD that costs 30X. SliceLSH performs 6.9K lookups/s.

2. DESIGN REQUIREMENTS AND EXISTING SYSTEMS

Our goal is to develop *generic* SSD design optimizations that can be applied *nearly universally* to a variety of random hash-based indexes that each have the following requirements:

Large scale: A number of data-intensive systems require large indexes. For example, WAN optimizer [5, 6] indexes are $\geq$32GB; data de-duplication indexes are $\geq$40 GB [3]. In keeping with the trend of growing data volumes, we target indexes that are an order-of-magnitude larger, i.e., a few hundred GB.

High performance and low cost: The index should provide high throughput, low per-operation latency, and low overall cost, memory, and energy footprint. To apply to a wide-variety of content-based systems, the index should provide good performance under both inserts/updates and reads. State-of-the art techniques for hash tables offer 46K IOPS [14, 25]; those for bloom filters offer 12-15K IOPS [22]. Our indexes should match or exceed this performance.

Flexibility: This covers various aspects of how easy the index is to use, as we discuss below.

Applications leveraging a given index may require significant CPU and memory resources for their internal operations. For example, data de-duplication applications require CPU resources for computing SHA-1 hashes of fingerprints [12]. Various image and video search applications require CPU resources for computing similarity metrics after they find potential matches. Caching applications may want to use memory for caching frequently accessed content. To ensure that the applications can flexibly use CPU and memory and that their performance does not suffer, the index should impose low CPU and memory overhead. Unfortunately, many prior index designs ignore the high CPU overhead they impose in their singular quest for, e.g., low memory footprint, and high read performance (e.g., SILT [25]), which makes application design tricky. Equally importantly, application designers should be able to easily extend the index with evolving application requirements, e.g., add memory or CPU cores at a modest additional cost to obtain commensurately better performance. Finally, the index should work well under a variety of workload patterns.

In the rest of this section, we survey other related hash-based systems that employ flash storage. As stated earlier, none of these studies use techniques that are all generally applicable across different random hash-based indexes. Even ignoring this issue, all prior designs fall short on one or more of the above requirements.

2.1 SSD-Based Hash Tables

We start by reviewing a specific class of indexes, namely those based on hash tables. We review several prior systems each designed for a specific application domain. We highlight the design choices made in each case and the restrictions they impose.

Many recent works [14, 15, 20, 21, 25] have proposed SSD-based indexes for large key-value stores. As Table 1 shows, each design optimizes for a subset of metrics that matter in practice (i.e., high throughput, low latency, low memory footprint or low computation overhead). Unfortunately, these optimizations come at the expense of significantly impacting other metrics and they may impact the applications that use the indexes, as we argue below.

FlashStore[20] stores key-value pairs in a log-structured fashion on SSD storage, and uses an in-memory hash table to index them. It optimizes for lookup (on average, one SSD read per lookup), but imposes high memory overhead ($\sim$6 bytes/key). *SkimpyStash* [21] uses a low amount of memory—1 byte/key—to maintain a hash table with linear chaining on the SSD. However, it requires 5 page reads/lookup on average.

BufferHash [14] buffers all insertions in memory, and writes them in a batch to the SSD. It maintains in-memory Bloom filters [8] to avoid spurious lookups to any batch on the SSD. BufferHash requires $\sim$1 page read per lookup on average and works well across a range of workloads. However, it may need to read multiple pages in the worst case due to false positives of the Bloom filters. Buffer-

	FlashStore	SkimpyStash	BufferHash	SILT
Avg Lookup (#page read)	~1	**~5**	~1	~1
Worst Lookup (#page read)	1	**10**	**16**	**33**
Memory (# bytes/entry)	**~6**	~1	**~4**	~0.7
CPU overhead	Low	Low	Low	**High**

Table 1: Comparison of different SSD-based Hash tables under different metrics. The worst-case lookups are based on default prototype configurations of these systems. Existing SSD-based Hash tables are optimized for one set of metrics, but incur additional overhead or perform poor under other metrics (shown in bold red).

Hash also has a high memory overhead (~4 bytes/key) due to in-memory Bloom filters. Finally, BufferHash is difficult to tune: it requires a predetermined amount of memory (a function of SSD size) to ensure that the false positive rate is low and worst-case lookup cost is small.

SILT [25] offers a better balance across the different metrics than any of the above systems. SILT achieves a low memory footprint (0.7 bytes/entry) and requires a single page lookup on average. However, SILT uses a much more complex design than the systems discussed above. It employs three data structures: one of them is highly optimized for a low memory footprint, and the others are more write-optimized but require more memory. SILT continuously moves data from the write-optimized data structures to the memory-efficient one. In doing so, SILT has to continuously sort newly written data and merge it with old data. This increases the computation overhead, which may impact the applications that use SILT. Furthermore, these background operations affect the performance of SILT under a workload of continuous inserts and lookups as is common with, e.g., WAN optimizers. For example, the lookup performance drops by 21% for a 50% lookup-50% insert workload on 64B key-value pairs. While SILT is somewhat tunable—e.g., it is possible to tune the memory overhead between 0.7 and 2B per entry [25]—it doesn't permit configurations with arbitrarily low memory footprint contrary to our index designs.

Also, none of the above systems are designed for exploiting the intrinsic parallelism of SSDs. As we show in §4, lookup performance can improve by 5.2X if the underlying parallelism is optimally exploited.

2.2 Other Indexes

Other hashing-based data structures have received less attention than hash tables. But there has been growing interest in using SSDs to support them when the scale is large, especially for Bloom filters.

Buffered Bloom Filter [17] is an approach for SSD-resident Bloom filters that targets initial construction of Bloom filters to ensure a low memory footprint. However, this data structure cannot handle updates over time. BloomFlash [22] is an approach for SSD-resident Bloom filters that optimizes for writes. BloomFlash buffers bit updates in DRAM to avoid random writes to the SSD. It also uses a hierarchical organization to manage writes. Neither approach leverages parallelism intrinsic to SSDs. In particular, our experiments show that by adapting BloomFlash's design using our parallelism-centered patterns and techniques, we can achieve the same I/O performance using a commodity SSD that their design achieves with a high-end SSD costing 30X more.

The critical takeaways from the above discussion are that the individual designs are targeted to specific scenarios and workloads; they are often not easy to tune, e.g., to trade-off performance for memory; they are CPU intensive; and techniques used in one often don't extend to another.

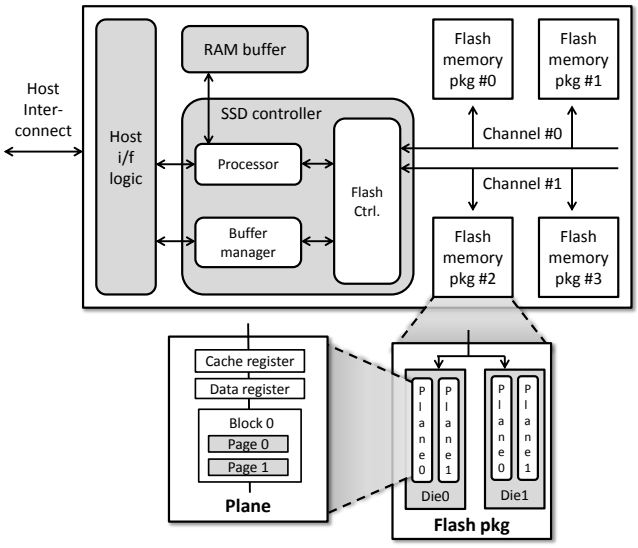


Figure 1: SSD internal architecture

Our goal is to develop guidelines to design indexes that offer high I/O performance and low memory overhead, are easy to tune, work well under a variety of workloads, and apply to a variety of indexes based on random hashing, including hash tables, locality-sensitive hash tables, and Bloom filters.

3. PARALLELISM IN SSD ARCHITECTURE

To meet our goal, we must first understand key properties of SSDs that influence the design and performance of random hash-based indexes. To this end, we describe the internal architecture of SSDs. We then describe the different forms of parallelism available within SSD architectures.

Figure 1 shows an illustration of a SATA-based SSD architecture. SSDs provide logical block addresses (LBAs) as an interface to the host. All I/O requests for LBAs are processed by an SSD controller. The controller receives I/O requests from the host via an interface connection (i.e., the SATA interface). The controller uses the flash translation layer (FTL) to translate logical pages of incoming requests to physical pages. It issues commands to *flash packages* via flash memory controllers. The flash memory controller connects to flash packages via *multiple channels* (generally 2-10).

Each package has two or more *dies* or chips. Each die is composed of two or more *planes*. On each plane, memory is organized into *blocks*; each block consists of many *pages*. Each plane has a data register to temporarily store the data page during reads or writes. For a write command, the controller first transfers data to a data register on a channel, and then the data is written from the data register to the corresponding physical page. For a read command, the data is first read from the physical page to the data register, and then transferred to the controller on a channel.

Different Forms of Parallelism. The internal architecture of SSDs incorporates varying degrees and levels of parallelism. Each of an SSD's channels can operate in parallel and independently of each other. Thus, SSDs inherently have *channel-level parallelism*. Typically, the data transfers from/to the multiple packages on the same channel get serialized. However, data transfers can be interleaved with other operations (e.g., reading data from a page to the data register) on other packages sharing the same channel [10, 29]. This

interleaving provides *package-level parallelism*. The FTL stripes consecutive logical pages across a gang of different packages on the same channel [10] to exploit package-level parallelism. Furthermore, the command issued to a die can be executed independently of the others on the same package. This provides *die-level parallelism*.

Multiple operations of the same type (read/write/erase) can happen simultaneously on different planes in the same die. Currently, a *two plane command* is widely used for executing two operations of the same type on two different planes simultaneously. This provides *plane-level parallelism*. Furthermore, the data transfers to/from the physical page can be *pipelined* for consecutive commands of the same type.

4. PARALLELISM-FRIENDLY DESIGN PATTERNS

At the heart of our work lies a generic set of techniques for carefully extracting the above intrinsic parallelism of SSDs to ensure high performance without sacrificing generality and tunability. In what follows, we first outline known properties of SSD I/O, and techniques for accommodating them (§4.1). We then describe design patterns that help account for both the known properties as well as the available forms of parallelism (§4.2).

4.1 Reads and Writes

The read/write properties of SSDs are well known. In particular, a page is the smallest unit of read or write operations, meaning that reading a 16B entry (such as a key-value pair in a hash table) is as costly as reading an entire page. Also, the performance of random page reads is comparable to that of sequential page reads. Thus, we arrive at design pattern **DP1**: *Organize data on the SSD in such a way that multiple entries to be read reside on the same page.*

SSDs show poor performance under a heavy random write workload [27]. Even the random read performance is affected in a mixed workload of continuous reads and writes [14]. A common design pattern, which we call **DP2**, used to accommodate this property is: *Leverage a small amount of memory to buffer writes and flush data out to the SSD at a granularity lower bounded by the size of a block (typically 128K)*; we adopt this in our design.

We now describe the benefits of, and techniques for, applying these insights along with leveraging SSD parallelism.

4.2 Extracting Parallelism

Channel-level Parallelism. The throughput of page reads can be significantly improved by leveraging channel-level parallelism. However, a simple way of using multiple threads to issue requests in parallel does not work in the general case: when a sudden skew in input keys forces all requests to go to the same channel, naive parallel lookups will obviously not provide any benefits. To extract the benefits of parallelism under a wide-range of workloads and workload variations, we need to ensure that the requests issued to the SSD are spread uniformly across the channels. This becomes possible if we know the mapping between pages and channels. Armed with this knowledge, we can then reorder lookup requests to ensure that those issued concurrently to the SSD are uniformly spread across channels.

However, the mapping is often internal to SSDs and not exposed by vendors. Recent work [18] has shown that this mapping can be reverse engineered. As mentioned earlier, the FTL stripes a group of consecutive logical pages across different packages on the same channel. The authors in [18] discuss a technique to determine the size of the group that gets contiguously allocated within a channel; they call this logical unit of data a *chunk*. They show how to determine the chunk size and the number of channels. Using this, they also show how to derive the two common mapping policies: (1) *write-order mapping*, where the i^{th} chunk write is assigned the channel $i \% N$, assuming N is the number of channels, and (2) *LBA-based mapping*, where the logical block address (LBA) is mapped to channel number $LBA \% N$.

As an example, we employed the technique in [18] with a Crucial SSD. We estimated the chunk size and number of channels to be 8KB and 32, respectively. We further found that the Crucial SSD follows write-order mapping. Figure 2a shows the lookup performance of the our channel-aware technique that uses the above estimates of the SSD channel count and mapping policies, for different numbers of threads (labeled "Best"). We also show the worst case (labeled "Worst"), where we force requests to go to the same channel. We find that the gap between the two is quite substantial—nearly 5.2X. As a point of comparison, we also show the performance of simply issuing multiple requests using multiple threads without paying attention to channel-awareness (labeled "Rand"): we see that this is up to 1.5X worse for this workload.

Thus, we arrive at the following design pattern **DP3**: *when performing lookups, rearrange them such that the requests are evenly spread across channels.*

We further investigate if issuing concurrent writes leads to similar benefits as concurrent reads; as stated above, each write should be at least the block size (DP2). Figure 2b shows results for the Crucial SSD. We see that parallelism offers marginal improvement at best. The reason is that the Crucial SSD's write-order-based mapping assigns consecutive chunks (8KB) to different channels, and so, by default, any write larger than the chunk size is distributed over multiple channels. Thus, we arrive at **DP4**: *simply issuing large bulk writes suffices - issuing writes concurrently is not essential to improving write throughput.*

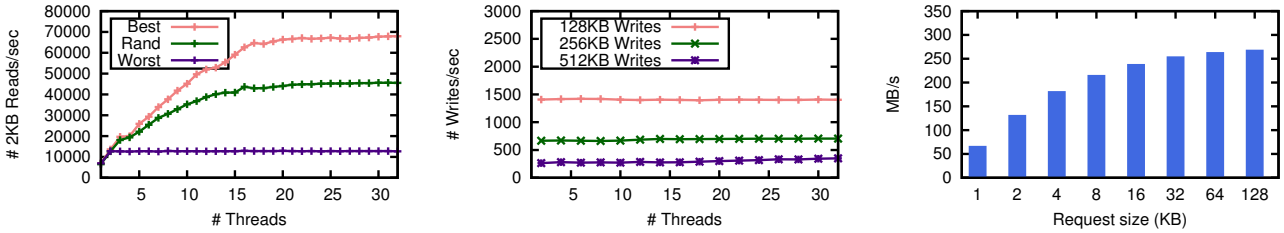
Package-level parallelism: Figure 2c shows the random read performance for different read sizes. We observe high read bandwidth when large reads are issued. This is because reads up to the chunk size (8KB) can exploit package-level parallelism. Reads larger than the chunk size can exploit both channel-level and package-level parallelism. Thus, we have **DP5**: *when possible, it helps to issue large reads.*

Note that this design pattern cannot be used for regular lookups into an index data structure, as issuing large reads may retrieve useless data resulting in low system goodput. However, as we will show later, this design pattern aids in instrumenting patterns 1–4.

Plane-level parallelism: Earlier works [18, 29] have shown that intermingling small reads and small writes affects plane-level parallelism, leading to a performance drop of up to $1.3X$ in throughput compared to issuing consecutive small reads followed by consecutive small writes. However, the above design patterns already dictate that we issue large writes (DP2) and small reads (small page reads for lookup requests; DP1), which already ensure that small reads and small writes are not intermingled by default. Thus, there are no further undesirable interactions with plane-level parallelism.

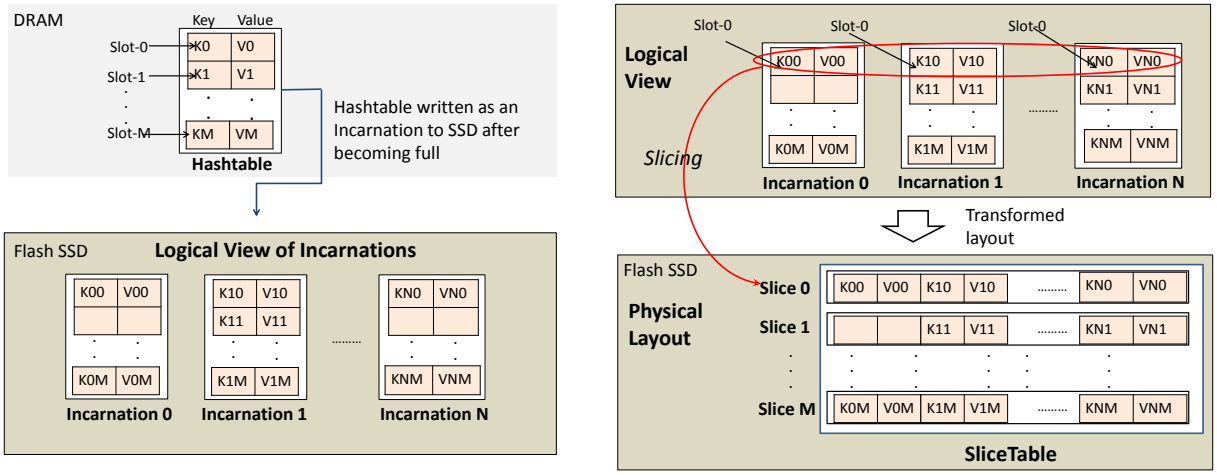
5. STREAMING HASH TABLES: SLICEHASH

In this section, we discuss how, using the design patterns (DP1–5), we can develop techniques for building high-performance large streaming hash tables where <key, value> pairs can be looked up, inserted, updated and evicted over time. We call our index SliceHash. We will describe how to build other index data structures in §6.



(a) Concurrent 2K reads (b) Concurrent large writes (c) Concurrent large reads (32 threads)

Figure 2: Concurrent I/O performance



(a) Hash table in DRAM and its incarnations on SSD (b) Physical layout of incarnations on SSD

Figure 3: Basic SliceHash structure

Our key innovation, which applies across all the data structures, is the use of a *slicing* primitive for storing multiple related entries on the same page (DP1) thereby helping combine multiple index lookups into one page read. We use known techniques for dealing with random writes (DP2) but adapt them to work with slicing (based on DP4 and DP5). Finally, we discuss how we implement support for concurrent I/O in SliceHash (based on DP3). We show how the design patterns influence key aspects of the configuration of the data structure as well as the techniques we use to read from and write to the SSD.

5.1 Basic SliceHash

Figure 3 shows the basic overview of SliceHash. SliceHash hierarchically organizes the hash table across DRAM and SSD. We maintain an in-memory hashtable, and inserts *only* happen in this in-memory table. After the in-memory table is full, it is written as an *incarnation* to the SSD in a batch. Over time, multiple incarnations are written to the SSD. This aspect of the design is motivated by DP2 for avoiding slow writes/updates to random SSD locations during insertion of keys. BufferHash [14] also uses a similar design principle, but SliceHash differs in how the data in the incarnations is laid out on the SSD.

SliceHash uses the idea of slicing to lay out the data. Figure 3b shows the physical layout of the data in incarnations in the form of a slicetable. Before describing construction of the slicetable, we define a few key terms:

- A *slot* is an index into the in-memory hash table or an on-SSD incarnation, where an entry (i.e., a key-value pair) is or can be stored.
- For a given slot, a *slice* is a list of all entries located at the slot within all on-SSD incarnations. In Figure 3b, the slice for slot-0, i.e., *slice-0*, contains entries from slot-0 from each incarnation, e.g., <K00, V00> from incarnation-0, <K10, V10> from incarnation-1, and <KN0 , VN0> from incarnation-N.
- A *slicetable* then refers to a sequential arrangement of slices on the SSD, each slice corresponding to a given slot. A slicetable can span multiple SSD blocks. In Figure 3b, the slicetable contains slice-0, slice-1, ..., slice-M.
- A *SliceHash* is comprised of both the in-memory hash table and the on-SSD slicetable.

Slicing improves lookups. The main advantage of using slicing is that lookup is vastly simplified and more efficient compared to storing incarnations directly on the SSD.

When incarnations are stored directly on the SSD, we may have to examine all incarnations since the key may be present in any of them. Since each incarnation occupies a different set of SSD pages, a key lookup may incur multiple SSD page reads.

In contrast, using slicing simplifies lookups: we hash a key to obtain the slot, and simply read the corresponding slice. We then compare the input key against the entries in the slice to obtain the relevant value. For example, in Figure 3b, look ups for keys in the slot-0 of all incarnations only require reading the corresponding

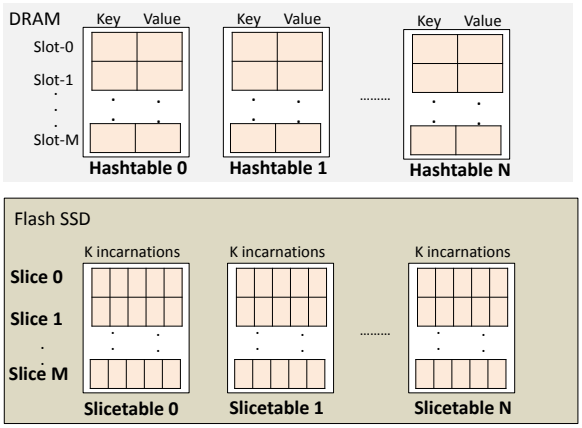


Figure 4: Partitioned SliceHash

slice-0. By limiting the size of a slice to be one or a few pages (DP1), we correspondingly limit the cost of lookup.

Impact on inserts. While slicing positively impacts lookups, it makes inserts complex. In particular, when flushing a full in memory hash table to the SSD, we need to maintain the structure of the slicetable on the SSD. Because a slice has entries from all incarnations, we would need to modify each slice to include entries from the new incarnation. However, the cost of this operation gets amortized over multiple inserts.

We first read as many blocks of a slicetable as possible from the SSD to memory (since, the entire slicetable might not fit in memory). This amounts to a "large" read and hence can be performed at high throughput (DP5). We then modify these blocks at the appropriate positions for each slice, and write back to the SSD; as DP4 indicates, such large writes help leverage channel-level parallelism. We continue reading, modifying and writing back the subsequent remaining blocks of the slicetable, till the whole slicetable is modified on SSD.

While this imposes a high overhead, it is only incurred when the in-memory hash table is full. Since the vast majority of insert operations still happen in memory, the impact of this flush operation on an average insert is small. We further discuss in §5.1.1 how we reduce these overheads.

To summarize, the basic operations in SliceHash are as follows:

Inserts and updates: Keys are inserted only into the in-memory hash table. When this becomes full, we flush it to the SSD while maintaining the slicetable structure. When the on-SSD slicetable becomes full, we employ the simple "eviction policy" of overwriting the oldest incarnation. For updates, we simply insert the new key-value pair in the in-memory hash table.

Lookups: We first look up the key within the in-memory hash table. If the key is not found, we read the corresponding slice from the SSD, scan the entries for all incarnations *from the latest to the oldest*. This ensures that the lookup does not return stale values in the face of updates.

5.1.1 Partitioning SliceHash

Maintaining a single large slicetable spanning the entire SSD is not scalable: in particular, this can cause the flush of the in-memory hash table to take an undue amount of time during which lookup operations can also be blocked (note that SSD I/Os are blocking). Further, it requires multiple SSD I/Os for reading, modifying and writing back the single large slicetable to SSD. To mitigate this and control the worst case insertion cost, we adopt a strategy similar to BufferHash: We partition the in-memory hash table to multiple small in-memory hash tables based on the first few bits of the keys' address space. We then maintain a separate slicetable for each in-memory hash table (shown in Figure 4). If an in-memory partition becomes full, we only need to update the corresponding (smaller) slicetable on the SSD. Thus, we can read the smaller slicetable entirely in memory, modify it at appropriate positions, and write it back to SSD.

Henceforth, we assume a partitioned SliceHash is in use. Furthermore, we use the term "in-memory hash table" to refer to one of the partitions in memory (Figure 4).

5.1.2 Some Optimizations

Note that, for ease of explanation, we present the case of a simple hash table with a single entry per slot. However, we can trivially support hash tables with a fixed-size bucket of entries for every slot; in this case, each slice will have buckets of entries from all incarnations for a given slot. We can also support an N-function Cuckoo hash table [23]; in this case, a key lookup may need to read up to N slots in the worst case (when the key is not found in the first $N-1$ slots). Lookup cost is bounded by N page reads.

SliceHash may require an SSD page read for a key lookup even if the key is not present in the entire data structure. Additional memory, if available, can be used to reduce such spurious lookups through the use of a summary data structure, such as a Bloom filter, for every slicetable. All lookups are first checked against Bloom filters. SSD operations are issued only if the Bloom filters indicate that the key is present. Crucially, our design can use memory opportunistically: e.g., we maintain Bloom filters only for some partitions, e.g., those that are accessed frequently. This gives SliceHash the ability to adapt to memory needs, while ensuring that in the absence of such additional memory, application performance targets are still met.

5.2 Adding concurrency to SliceHash

In order to leverage the parallelism inherent to SSDs, I/O requests should be issued in such a manner that they are spread uniformly across channels (DP3). We use two components to achieve this: (1) a *scheduler* for request selection, and (2) a *worker* for SSD reads/writes.

The scheduler processes requests in batches. It first process all requests that can be instantly served in memory. Then, it processes lookup requests which require reading from the SSD. We have developed a channel-estimator (described later) to estimate the mapping between read requests and channels. Using these estimates, the scheduler finds a set of K requests (we choose K as the size of the SSD's NCQ (native command queue)).

The request selection algorithm works as follows. The goal here is to ensure that requests get uniformly distributed across channels to optimally exploit channel parallelism offered by NCQ. To meet this objective, we maintain a "depth" for each channel, which estimates the number of selected requests for a channel. We take multiple passes over the request queue until we have selected K requests (size of SSD's NCQ). In each pass, we select requests that would increase the depth of any channel by at most 1. In this manner, we first find the set of read requests to be issued.

The scheduler then instructs the worker to process the chosen read requests in parallel. The worker simply employs multiple threads to issue requests to the SSD. Each thread is "associated" with a channel and is assigned requests that correspond to this channel. When a thread completes a request, it accepts new requests for the channel. As the SSD page reads complete, the worker searches the entries of all incarnations on the pages for the input key. After pro-

Symbol	Meaning	Symbol	Meaning	Symbol	Meaning
n	Number of partitions	F	Total SSD size	r_p	Page read latency
s	Size taken by a hash entry	M	Total memory size	r_b	Block read latency
u	Utilization of the hash table	N	Number of SSDs	w_b	Block write latency
s_{eff}	Effective size of the hash entry ($= s/u$)	P	Size of an SSD page/sector		
k	Number of incarnations ($= F/M$)	B	Size of an SSD block		
r_{write}	Ratio of insertions to block writes	H	Size of a single hash table ($= M/n$)		
R	Insert rate	S	Size of slicetable ($= H \times k$)		

Table 2: Notations used in cost analysis

cessing lookups, the scheduler assigns SSD insert requests to the worker soon after an in-memory hash table fills up and needs to be flushed to the SSD. The worker accordingly reads/writes slicetables from/to the SSD.

Channel Estimation. We now describe a simple technique to estimate the channels corresponding to the read requests issued to the SSD, which is a crucial component in performing concurrent I/O on the SSD (DP3). We focus on SSDs that use write-order mapping (the mapping strategy can be inferred using the techniques in [18] as mentioned in §4). Similar approaches can be employed for SSDs that use other write policies.

As discussed in §4, chunk writes in write-order mapping are striped across channels, i.e., the first write goes to the first channel, the second write goes to the second channel, and so on. We leverage this property and restrict the size of a slicetable to be a multiple of $N \times ChunkSize$, where N is the number of channels. Thus, whenever a slicetable is written to the SSD, there will be N chunk writes, with the i^{th} chunk write going to the i^{th} channel. In other words, once we determine the relative chunk identifier (first, or second, or N^{th}) for an offset in the slicetable, we can determine the channel. The relative chunk identifier can be determined as the offset modulo chunk size. Although this is a heuristic, experiments show that it is remarkably effective at helping the scheduler schedule requests across channels (§7).

5.3 Leveraging multiple SSDs

Due to its simple design and low resource footprint, SliceHash can be easily extended to run across multiple SSDs attached to a single machine. We elaborate below on two possibilities: one that offers high throughput and the other that offers low memory footprint.

Throughput-oriented design. We can exploit multiple SSDs to increase parallelism and obtain high throughput. To do this, we partition the key-space across multiple SSDs so that incoming requests are distributed across SSDs and can be processed by SSDs in parallel.

Memory-oriented design. We can also exploit multiple SSDs to lower the memory footprint of SliceHash. In this design, a slicetable for an in-memory hash table expands across multiple SSDs, i.e., each slice has its entries stored across multiple SSDs. So the slicetable can contain a larger number of incarnations compared to the number of incarnations when using a single SSD. As the number of incarnations increases, the memory footprint is reduced (§7.4). Although more incarnations must be read when reading a slice, lookups can be issued to multiple SSDs in parallel, avoiding any loss in performance.

5.4 Analysis

In this section, we analyze the I/O latency and the memory overhead of SliceHash. We also estimate the number of writes to the SSD per unit time, and its impact on SSD lifetime. Alongside, we illustrate the knobs SliceHash offers to easily control cost-performance trade-offs; such tunability is missing from almost all prior designs. Table 2 summarizes the notation used.

Memory overhead per entry. We estimate the memory overhead per entry. The total number of entries in an in-memory hash table is H/s_{eff}, where H is the size of a single hash table and s_{eff} is the effective average space taken by a hash entry (actual size (s)/utilization (u)). The total number of entries overall in SliceHash for a given size F of the SSD is: $(\frac{F+M}{H}) \times \frac{H}{s_{eff}} = \frac{F+M}{s_{eff}}$

Here, M is the total memory size. Hence, the memory overhead per entry is, $\frac{M}{\#entries}$, i.e., $\frac{M}{F+M} \times s_{eff}$, or $\frac{1}{k+1} \times s_{eff}$, where k is the number of incarnations.

For $s = 16$B (key 8 bytes, value 8 bytes), $u = 80\%$, $M = 1$GB, and $F = 32$GB, the memory overhead per entry is *0.6 bytes/entry*.

In contrast, state-of-the-art approaches for SSD-based hash tables, e.g., SILT [25] and BufferHash [14] have memory overheads of *0.7 bytes/entry* and *4 bytes/entry*, respectively. The use of Bloom filters (used to prevent lookups from incurring multiple SSD reads across incarnations) in BufferHash imposes high memory overhead.

Insertion cost. We estimate the average time taken for insert operations. We first calculate the time taken to read a slicetable and then write it back. This is given by: $(\frac{S}{B} \times r_b + \frac{S}{B} \times w_b)$, where S is the size of the slicetable, B is the size of an SSD block, and r_b and w_b are the read and write latencies per block, respectively. This flushing happens after H/s_{eff} entries are inserted to the hash table; all insertions up to this point are made in memory. Hence, the average insertion cost is $\left(\frac{S}{B} \times r_b + \frac{S}{B} \times w_b\right) \times \frac{s_{eff}}{H}$

Replacing S by $H * k$, we get $\frac{(r_b+w_b) \times s_{eff} \times k}{B}$, which is independent of the size of the hash table.

For a typical block read latency of 0.31ms [13], a block write latency of 0.83ms [13], $s = 16$B, $M = 1$GB, $F = 32$GB, and $u = 80\%$, the average insertion cost is $\sim 5.7\mu$s. Similarly, the worst case insertion cost of SliceHash is $(0.31 + 0.83) \times \frac{S}{B}$ms. By configuring S to be same size as B, we can control the worse case insertion cost to $(0.31 + 0.83) = 1.14ms$.

In contrast, BufferHash has average and worst case insertion latencies of $\sim 0.2\mu$s and $0.83ms$, both of which are better than SliceHash. We believe that the somewhat higher I/O costs are an acceptable trade-off for the much lower memory footprint in SliceHash.

Lookup cost. We consider a Cuckoo hashing based hash table implementation with 2 hash functions. Suppose that the probability of success for the first lookup is p. For each lookup, a corresponding slice is read. Configuring H, the size of an in-memory hash table, to match that of a page, the average lookup cost becomes $r_p + (1-p) \times r_p$ or $(2-p) \times r_p$, assuming that all of the lookups go to the SSD. For $p = 0.9, r_p = 0.15$ ms, the average lookup cost is 0.16 ms. SILT and BufferHash have a similar average lookup cost.

The worst case happens when we have to read both pages corresponding to the two hash functions. Thus, the worst case lookup latency is $2 \times r_p$. For $r_p = 0.15$ ms, this cost is 0.3 ms. In contrast,

BufferHash may have very high worst case lookup latency because it may have to scan all incarnations due to the false positives of Bloom filters. For $k = 32$, this cost would be as high as 4.8 ms.

Frequency of SSD writes, and knobs for tunability. We estimate the ratio of the number of insertions to the number of block writes to the SSD; we denote this as r_{write}. A hash table becomes full after every H/s_{eff} inserts, after which the corresponding slicetable on the SSD is modified. The number of blocks occupied by a slicetable is S/B or $k \times H/B$. Thus, $r_{write} = \frac{H}{s_{eff}} \times \frac{B}{k \times H} = \frac{B}{k \times s_{eff}}$

Thus, by increasing the number of incarnations k, the frequency of writes to the SSD (which is inversely proportional to r_{write}) also increases. This in turn affects the overall performance.

Note, however, that increasing the number of incarnations also decreases the memory overhead as shown earlier. We investigate this dependency in more detail in §7.4 and find that our design provides a smooth trade-off between memory overhead and performance, allowing designers the flexibility to pick a point in the design space that best fits their specific cost-performance profile.

Effect on SSD lifetime. SliceHash increases the number of writes to the SSD which may impact its overall lifetime. We now estimate the lifetime of an SSD as follows. For a given insert rate of R, the number of block writes to the SSD per second is $\frac{R}{r_{writes}}$ or the average time interval between block writes is $\frac{r_{writes}}{R}$. Say the SSD supports E erase cycles. Also, assume that the wear leveling scheme for the SSD is perfect. Then, the lifetime (T) of the SSD could be approximately estimated as number of blocks, $\frac{F}{B}$, times erase cycles, E, times the average time interval between block writes, $\frac{r_{writes}}{R}$, i.e., $T = \frac{F \times E \times r_{writes}}{R \times B}$

Consider a 256GB MLC SSD drive that supports 10000 erase cycles [16]. We use SliceHash on this SSD with $M = 4$GB of DRAM, i.e., $k = 64$. With a 16B entry size and utilization of 80%, the ratio r_{write} would be 102.4. Even with $R = 10K$ inserts/sec (required, e.g., for a WAN optimizer connected to 500 Mbps link), the SSD would last 6.8 years. Thus, despite an increase in the writes to SSD, its lifetime would still be reasonably long.

In sum, our analysis shows that our design patterns help SliceHash to reduce the memory overhead to 0.6 bytes/entry and limit the lookup cost to 1 page read on average, without significantly affecting the average insert performance or SSD lifetime. A simple knob—the number of incarnations—helps control the performance-cost trade-off in a fine-grained fashion. We empirically study the performance and flexibility benefits of SliceHash in §7.

Next, we discuss how our key techniques can also be applied to other (hashing-based) data structures.

6. GENERALITY

In this section, we discuss how the five design patterns discussed in §3 and the slicing primitive discussed in §5, can be used to design other hashing-based data structures, particularly, Bloom filters and locality sensitive hashing (LSH)-based indexes. Many of the supporting design techniques we used in SliceHash—the use of incarnations, slices, slicetables, and optimizations for multiple SSDs—are derived directly from the design patterns and hence, as argued below, they also apply directly to other data structures.

Bloom Filters. Bloom filters have traditionally been used as in-memory data structures [8]. As some recent studies have observed [17, 22], with storage costs falling and data volumes growing into the peta- and exa-bytes, space requirements for Bloom filters constructed over such datasets are also growing commensurately. In limited memory environments, there is a need to maintain large Bloom filters on secondary storage. We show how we can apply our techniques for supporting Bloom filters on SSD storage effectively.

Similar to SliceHash, we maintain several in-memory Bloom filters and corresponding slicefilters on the SSD; the in-memory Bloom filters are written to the SSD as incarnations. Each slice in a slicefilter contains the bits from all incarnations taken together for a given slot.

In traditional Bloom filters, a key lookup requires computing multiple hash functions and reading entries corresponding to the bit positions computed by the hash functions. In our case, for each hash function we first look up the corresponding in-memory Bloom filter and then the corresponding slicefilter on the SSD.

The number of hash functions would determine the number of page lookups, which could limit the throughput. We now argue how this cost can be controlled.

Since SSD storage is much cheaper than DRAM, we can use more space per entry on the SSD – i.e., use a large m/n where m and n are the Bloom filter size and the number of unique elements, respectively; this allows us to use fewer hash functions (smaller h) while maintaining similar overall false positive rate [1]. For example, for a target false positive rate of 0.0008, instead of using $m/n = 15$ and $h = 8$, we can use $m/n = 32$ and $h = 3$. By reducing h, we can reduce the number of page lookups and improve performance.

Our design patterns and the techniques we derive from them enable us to reduce the effective memory footprint per key (where a "key" refers to a unique element inserted into the Bloom filter) while achieving high performance, similar to the trade-offs we were able to achieve with SliceHash. For example, choosing $m/n = 32$, we can use a combination of a 256MB DRAM and a 64GB SSD (leading to 256 incarnations per Bloom filter) to store Bloom filters. This results in an effective memory overhead of 0.125 bits per entry and causes block writes to the SSD every 128 key insertions. Our evaluation in §7.5 shows that we achieve good throughput with this configuration.

LSH-based index. Locality sensitive hashing [24] is a technique used in the multimedia community [26, 9] for finding duplicate videos and images at large scale. LSH systems use multiple hash tables. For each key, the corresponding bucket in each hash table is looked up. Then, all entries in the buckets are compared with the key to find the nearest neighbor based on a certain distance metric.

In SliceLSH, each LSH hash table is designed using SliceHash. When a query arrives, it is distributed to all SliceHash instances. Leveraging the design patterns, we can subtly tweak the data structure to more closely align with how LSH works and ensure improved I/O performance. Specifically, when we write in-memory LSH hash tables to the SSD, we arrange them such that: (1) all chunks (group of consecutive logical pages assigned to same channel, as defined in §4.2) of each slicetable get mapped to the same channel (this is in contrast with SliceHash where each chunk in a slicetable may go to a different channel), and (2) the chunks corresponding to different LSH hash tables map to different channels. We write in-memory LSH hashtables to the SSD together; while writing to the SSD, we rearrange the chunks of LSH slicetables to satisfy the above properties. The benefit of this approach is that multiple LSH hash table lookups for a given key will be uniformly distributed over multiple channels. This helps us maximally leverage the intrinsic parallelism of SSDs resulting in high lookup throughput (§7).

7. EVALUATION

In this section, we measure the effectiveness of our design patterns as applied to the three different indexes described above, and we show the flexibility offered and the generality of our design choices. For simplicity, a majority of our evaluation focuses on SliceHash.

7.1 Implementation and Configuration

We have implemented SliceHash in C++ using ∼3K lines of code. I/O concurrency is implemented using the pthread library. We use direct I/O for access to the SSD. We use the simple "noop" scheduler in the Linux kernel (which implements basic FIFO scheduling of I/O requests) for leveraging the intrinsic parallelism of SSDs.

Each hash table is implemented using Cuckoo hashing [23] with 2 hash functions and 3 entries per bucket, which corresponds to 86% space utilization. As mentioned in §5, we have multiple in-memory hash tables. The size of each of these is 128KB, so each can hold ∼7K key-value entries of size 16B each. Slicetables corresponding to different in-memory hash tables are arranged across continuous logical block addresses on the SSD.

We evaluate SliceHash on a 128GB Crucial M4 SSD attached to a desktop with dual 2.26 GHz quad-core Intel Xeon processor. We use 32 threads for issuing concurrent I/O requests, corresponding to the number of channels in the Crucial SSD. The size of the NCQ is also 32.

Unless otherwise specified, the size of each slicetable is 4096 KB and the slicetable contains 32 incarnations of an in-memory hash table. This amounts to using 4GB DRAM in total toward the SliceHash data structure.

7.2 SliceHash Performance

We evaluate the lookup and insert performance of SliceHash, examining its throughput, memory footprint, and CPU overhead under different mixes of read and write workloads. We compare SliceHash with BufferHash and SILT.

Methodology. BufferHash [14] does not consider concurrent I/O access. For a fair comparison against SliceHash, we added concurrency to BufferHash using the pthread library. We also added locking mechanisms to ensure that no two threads access the same in-memory hash table of BufferHash at the same time. We use a similar configuration as in [14], i.e., 16 incarnations and 128 KB in-memory hashtable partition with maximum of 4096 entries. We use 8GB DRAM for in-memory hashtables, 8GB DRAM for Bloom filters and 128 GB for flash SSD. The memory footprint of this configuration is ∼ 4 byte/entry.

SILT [25] considers concurrent access by default. We use 4 SILT instances with 16 client threads concurrently issuing requests. A merge operation is triggered when a partition has one or more HashStores; we do not limit the convert or merge rates. At the beginning of each experiment, we insert 1 billion random key-value pairs to "warm-up" SILT's stores.

We use YCSB [19] to generate uniformly random key-value workloads with varying lookup and insert ratios.[1] Each workload consists of 1 billion operations, unless otherwise noted.

Lookup performance. Figure 5a shows the performance of the three systems—SliceHash (*SH*), multi-threaded BufferHash (*BH+MT*), and SILT (*SILT*)—for a lookup-only workload. We observe that SliceHash achieves 69K lookups/sec while SILT and BH+MT achieve only 62K lookups/sec (10% lower) and 57K lookups/sec (12% lower), respectively. *SliceHash achieves higher lookup performance because it exploits channel-level parallelism by running multiple threads accessing different channels in parallel*. In contrast, neither SILT nor BH+MT are designed to exploit such channel-level parallelism.

Percentage Inserts	Memory Footprint (bytes/entry)		
	SliceHash	BH+MT	SILT
0%	0.6	4	0.21
50%	0.6	4	0.21–0.57
100%	0.6	4	0.21–1.46

Table 3: Memory footprint under various workloads

Percentage Inserts	CPU Utilization (%)		
	SliceHash	BH+MT	SILT
0%	16	18	27
50%	12	24	67
100%	8	92	72

Table 4: CPU utilization under various workloads

Insert performance: We now study the insert throughput of SliceHash. Figure 5b shows the performance of the three systems for a continuous insert-only workload. We observe that SliceHash can achieve 125K inserts/sec. In contrast, BufferHash can achieve almost 1100K inserts/sec for the same configuration (i.e., 128 KB in-memory hash table), and SILT can achieve 254K inserts/sec.

BufferHash achieves much better insert performance, since it is write-optimized structure. However, BufferHash imposes very high memory footprint (∼4 bytes/entry). SILT achieves better insert performance but at the expense of an /increase in memory footprint (0.21 - 1.46 bytes/entry) due to a backlog of HashStores, as shown in Table 3. In contrast, SliceHash keeps the memory footprint small (0.6 bytes/entry) while achieving reasonably good insert performance. In fact, by bounding SILT's memory footprint to 0.6 bytes/entry, the insert rate of SILT is significantly impacted and reduced to only 46K inserts/s (as shown by "SILT-cap" in Figure 5b. Thus, *under same memory footprint, SliceHash is ∼3X better than SILT*.

We believe that the reduced insert performance of SliceHash is an acceptable trade-off for the significantly low memory overhead (Table 3) and better/more consistent lookup performance offered by SliceHash.

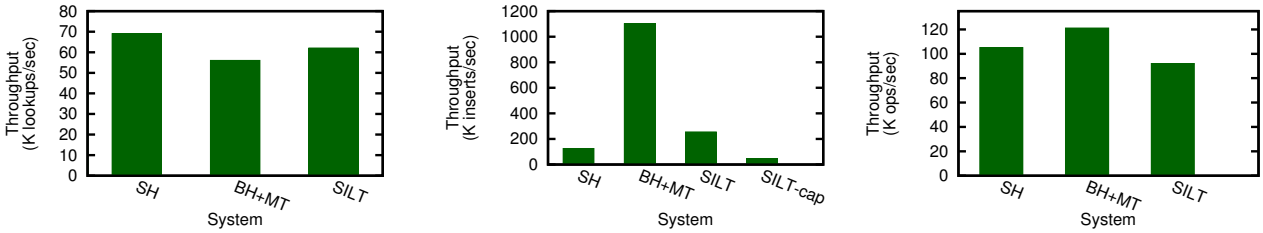
Moreover, SliceHash can be augmented with a small write-optimized table (using a BufferHash-like data structure) for handling bursts of writes; this table can be written back to SliceHash during a low I/O activity period. SILT uses a similar idea; it uses a write-optimized data structure for handling writes, which is later merged into SILT's read-optimized data structures. However, merging in SILT is far more compute-intensive (needs sorting) than writing a hash table back to a slicetable with SliceHash, which just requires copying entries to appropriate positions. As shown in Table 4, the average CPU utilization[2] during an insert-only workload is 72% when running SILT and 8% when running SliceHash.

Mixed workload. Finally, we investigate how SliceHash performs under a continuous workload of 50% lookups and 50% inserts. Figure 5c shows the performance of the three systems in this mixed workload setting. We observe that SliceHash provides 105K ops/sec, versus 121K ops/sec for BH+MT and 92K ops/sec for SILT.

BH+MT only has to write the buffer to the SSD when the buffer becomes full, while SILT and SliceHash have to perform extra operations, which affect their performance. SliceHash performs 14%

[1] We use the upper 8 bytes of the SHA1 hash of each YCSB-generated key as our 8 byte key.

[2] Utilization is the sum of %user, %nice, and %system as reported by iostat at 1 second intervals.



(a) Lookup-only (b) Insert-only (c) Mixed (50% lookup / 50% insert)

Figure 5: Performance under varying workloads and systems

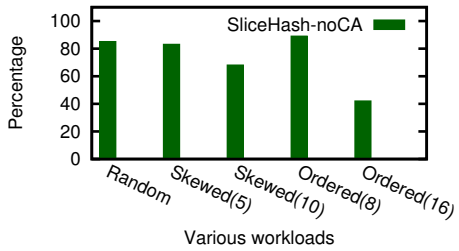


Figure 6: SliceHash-noCA relative to SliceHash

# Incarnations	Insert-only (ops/sec)	Mixed (ops/sec)	Memory footprint (B/entry)
16	207K	110K	1.1
32	139K	93K	0.6
48	85K	79K	0.38
64	66K	70K	0.27

Table 5: Memory vs. performance trade-off

better than SILT, and imposes very little CPU overhead (12%). In contrast, SILT imposes high CPU overhead (67%) due to its background converting and merging operations (Table 4).

7.3 Contribution of Optimizations

We now study how the two main parallelism-centered optimizations—request-reordering and slice-based data layout—contribute to SliceHash's performance.

Request-reordering. We study the extent to which reordering can be beneficial compared to a naive scheme of issuing requests in FIFO order. SliceHash-noCA (i.e., SliceHash with no channel-awareness) does not consider the request-to-channel mapping when assigning requests to a thread; requests are simply assigned to threads in the order the requests are made.

We consider three types of workloads to study the impact: (1) *Random:* the keys are generated randomly, so the distribution of requests among channels is also random; (2) *Skewed:* the channel distribution is skewed, i.e., a certain number of requests (configured by the skew parameter S) go to the same channel, while the remaining requests are evenly distributed across channels; and (3) *Ordered:* the requests are uniformly distributed across channels, however their ordering is such that the first K requests go to first channel, the second K requests go to second channel and the i^{th} set of K requests go to channel i *mod* N (where N is the number of channels; $N = 32$ for Crucial SSD). Essentially, if $K = 1$, even a FIFO scheme would have all 32 requests going to different channels (the best case), while if $K = 32$, it would result in all flash page read requests going to the same channel (the worst case).

Figure 6 shows the performance of SliceHash-noCA relative to the performance of SliceHash. In the worst case (Ordered (K=16)), SliceHash-noCA can only achieve 42% of SliceHash performance. Under a small skew of 5 requests (S=5), the performance drops by 17%; larger skew (S=10) deteriorates performance by almost 30%. Even with a random workload, where keys are likely to be evenly distributed across channels, we see a performance drop of 15%. These results indicate that channel-awareness is crucial to high performance in SliceHash.

Slicing. Slicing helps in reducing SliceHash's memory footprint compared to BufferHash's use of Bloom filters. In principle, BufferHash could avoid using Bloom filters and maintain the same memory footprint as SliceHash while leveraging concurrency to obtain good performance. However, we show that doing so has a severe performance impact.

We use the lookup-only workload from §7.2 to measure the throughput. We observe that BufferHash without Bloom filters achieves very low performance, only 8K lookups/sec. In contrast, SliceHash-noCA achieved 57K lookups/sec. Since the central difference between SliceHash-noCA and Bufferhash without Bloom filters is the use of slicing, this result shows that slicing is crucial for collectively achieving high performance and a low memory footprint.

7.4 Tuneability in SliceHash

SliceHash is highly flexible and can be tuned to match application requirements. SliceHash has a very small memory footprint (∼0.6 bytes/entry), and it can leverage additional memory to improve lookup performance, e.g., by using Bloom filters (§5.1.2). It also has a small CPU footprint, so it can easily be used with other applications requiring compute-intensive tasks. In contrast, BufferHash has a high memory footprint (Table 3), and SILT imposes high CPU overhead due to continuous sorting (Table 4); these aspects limit their suitability to a range of important applications.

In addition, SliceHash provides the flexibility to tune the memory footprint at the cost of performance, and it can scale to multiple SSDs without usurping memory/CPU, as we show below.

Memory footprint vs. Performance. By increasing the number of incarnations for a given SSD size and hence, using larger slicetable, we can reduce the memory footprint of SliceHash (memory footprint depends on the ratio of size of the in-memory hashtable and size of the slicetable). The side effect is that the number of block writes to flash SSDs is higher (since larger slicetable gets written to SSD when hashtable becomes full), which can affect the performance. Table 5 shows this trade-off for mixed (50% lookup/50% insert) and insert-only workloads. SliceHash provides a throughput between 110K-70K operations/sec for a mixed workload and 207K-66K operations/sec for an insert-only workload; SliceHash's memory footprint ranges from 1.1 bytes/entry to 0.27 bytes/entry. The lookup-only workload is not shown here, as performance re-

# SSDs	Lookup-only (ops/sec)	Mixed (ops/sec)	Memory footprint (B/entry)
1	69K	93K	0.6
2	138K	186K	0.3
3	207K	279K	0.2

Table 6: Leveraging multiple SSDs

mains close to 69K lookups/sec regardless of the number of incarnations.

Scaling using multiple SSDs. We evaluate SliceHash on our Intel Xeon machine using up to 3 SSDs for both the high-throughput and low memory footprint configurations outlined in §5.3.

We find that SliceHash can provide linear scaling in performance with the throughput-oriented configuration (Table 6). With 3 SSDs, SliceHash offers 207K ops/sec for a lookup-only workload, and 279K ops/sec for a mixed workload. Because of its low CPU and memory footprint, SliceHash can easily leverage multiple SSDs on a single physical machine to match higher data volumes and provide higher overall throughput without usurping the machine's resources. Neither SILT nor BufferHash can scale in this fashion: the former due to high CPU overhead (67% CPU utilization when one SSD is used; 3 SSDs exceed the CPU budget for a single machine) and the latter due to high memory overhead (48 GB for 3 SSDs).

In the memory-oriented configuration, SliceHash's memory overhead falls as the number of SSDs is increased, to 0.2 B/entry when 3 SSDs are used. But the throughput stays the same as using a single SSD. Neither SILT or BufferHash can offer similar scale down of memory.

7.5 Generality: SliceBloom and SliceLSH

We now show how our general design patterns improve the performance of other indexes.

We evaluate SliceBloom on the 128GB Crucial SSD using 512 MB DRAM. We use $m/n = 32$ and $k = 3$ hash functions with a memory overhead of 0.1 bits/entry. Under a continuous mixed workload, our system can perform 15K ops/sec. With naive parallelism, the system performance can drop to 5K ops/sec, especially when all requests go to the same channel. In contrast, BloomFlash [22] achieves similar performance for a mixed workload, but on a high-end Fusion-io SSD (100,000 4KB I/Os per sec) that costs 30X more ($6K vs. $200). Furthermore, on a low-end Samsung drive, BloomFlash only provides 4-5K lookups/sec.

We also evaluate SliceLSH on the Crucial SSD. We use 10 hash tables, where each hash table uses 256MB in memory, and the corresponding slicetable occupies 8GB on flash. SliceLSH can perform 6.9K lookups/sec, as it has to look up each hash table. By design, SliceLSH can intrinsically exploit channel parallelism. Hence, our system consistently offers similar performance under various workload patterns (results omitted for brevity).

7.6 Summary of key results

Our evaluation results show that our general design patterns improve the performance as well as tunability and flexibility of various indexes. Specifically,

- On a single SSD, SliceHash can achieve 69K lookups/sec, 10-12% higher than SILT and BufferHash. SliceHash can retain high lookup performance under different workloads by exploiting the internal parallelism of an SSD, while SILT's and BufferHash's performance can drop by 30%.
- SliceHash has much lower memory overhead (0.6 bytes/entry) compared to BufferHash (4 bytes/entry) and SILT (0.7 bytes/entry). Further, it has much lower CPU utilization (12%) in comparison to SILT (67%) and BufferHash (24%) under mixed lookups and inserts.
- SliceHash is highly flexible and tunable. SliceHash can be easily tuned to vary the memory footprint from 0.6 bytes to 0.27 bytes/entry, with slight degradation in performance from 93K to 70K operations/sec under mixed lookups and inserts. In addition, due to its low memory footprint and CPU overhead, SliceHash can be easily scaled using multiple SSDs, unlike other index designs.
- We also show that our design patterns are applicable to other indexes (SliceBloom and SliceLSH), and can improve their effectiveness.

8. DISCUSSION

Generality of SSD mapping policy. Using knowledge of the mapping policy for a Crucial SSD, we have shown how we can exploit its internal parallelism to get high performance for our index designs. Other SSDs may have different mapping policies, and similar techniques can be used to exploit their intrinsic parallelism. Even if the mapping policies are not completely known, certain patterns can be learned to understand how to exploit intrinsic parallelism. In addition, we can also consider designing new interfaces for SSDs, which could help applications leverage the underlying parallelism without revealing its internal mapping policies. We keep these problems open for future research.

Tolerating failures. Our designs maintain in-memory data structures which are vulnerable to system failures or crashes. This can be addressed using standard techniques, such as, appending in-memory inserts to a log file on an additional small SSD (similar to SILT [25]). In the event of crashes, this log file can be used to reconstruct the in-memory hash tables.

Large key-value pairs. Our designs are suited for small key-value pairs so that the size of a slice is limited to one or a few pages for a given number of incarnations. To accommodate large key-value pairs (e.g., few KBs or more), our designs can be extended by having additional indirection. Instead of storing the large value in the slice, we can only store the location information of the large value stored elsewhere on the SSD. This requires additional lookup, however, it keeps the insertion cost small.

9. CONCLUSION

A key impediment in the design of emerging high-performance data-intensive systems is the design of large hash-based indexes that offer good throughput and latency under specific workloads and at specific cost points. Prior works have explored point solutions using SSDs that are each suited to a narrow setting and crucially lack flexibility and generalizability.

In this paper, we develop a set of general techniques for building large, efficient and flexible hash-based systems by carefully leveraging unique properties of SSDs. Using these techniques, we first build a large streaming hash table, called SliceHash, that provides higher performance, while imposing low computation overhead and low memory overhead, compared to the state-of-the-art. Developers can easily tune SliceHash to meet performance goals under tight memory constraints and satisfy the diverse requirements of various data-intensive applications. The indexes also perform well under a range of workloads. We illustrate the generality of our ideas by showing that they can be applied to building other efficient and flexible hash-based indexes.

Additionally, our work shows the promise of adopting the design patterns and primitives we advocate to develop other general SSD-based indexes.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful feedback. This work is supported in part by National Science Foundation grants CNS-1302041, CNS-1314363, CNS-1330308, CNS-1040757, and CNS-1065134. Aaron Gember-Jacobson is supported by an IBM PhD Fellowship.

11. REFERENCES

[1] Bloom filters – the math. http://cs.wisc.edu/~cao/papers/summary-cache/node8.html.

[2] BlueCoat video caching appliance. http://www.bluecoat.com/company/press-releases/blue-coat-introduces-carrier-caching-appliance-large-scale-bandwidth-savings.

[3] Disk backup and deduplication with DataDomain. http://datadomain.com.

[4] Memcached: A distributed memory object caching system. http://memchached.org.

[5] Peribit Networks (acquired by Juniper in 2005): WAN optimization solution. http://www.juniper.net.

[6] Riverbed: WAN optimization. http://riverbed.com/solutions/wan_optimization.

[7] A. Badam, K. Park, V. S. Pai and L. Peterson. HashCache: Cache storage for the next billion. In *NSDI*, 2009.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics, 2005*, 1(4):485–509.

[9] A. Torralba, R. Fergus, and Y. Weiss. Small code and large image databases for recognition. In *CVPR*, 2008.

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

[11] A. Anand, A. Akella, V. Sekar, and S. Seshan. A case for information-bound referencing. In *HotNets*, 2010.

[12] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: The implications of universal redundant traffic elimination. In *SIGCOMM 2008*.

[13] A. Anand, S. Kappes, A. Akella, and S. Nath. Building cheap and large CAMs using BufferHash. Technical Report 1651, University of Wisconsin, 2009.

[14] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *NSDI*, 2010.

[15] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[16] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential RAID: rethinking RAID for SSD reliability. In *EuroSys*, 2010.

[17] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered bloom filters on solid state storage. In *ADMS*, 2010.

[18] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, 2011.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[20] B. K. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[21] B. K. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *SIGMOD*, 2011.

[22] B. K. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H.-C. Du. BloomFlash: Bloom filter on flash-based storage. In *ICDCS*, 2011.

[23] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Workshop on Distributed Data and Structures (WDAS)*, 2006.

[24] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[25] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[26] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *CIKM*, 2004.

[27] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. In *VLDB*, 2008.

[28] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, 2002.

[29] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *PVLDB*, 5(4):286–297, 2011.

[30] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.