Sphinx: A Succinct Perfect Hash Index for x86

Sajad Faghfoor Maghrebi
University of Toronto
Canada
smaghrebi@cs.toronto.edu

Niv Dayan
University of Toronto
Canada
nivdayan@cs.toronto.edu

ABSTRACT

Many modern key-value stores rely on an in-memory index to map the location of each data entry in storage. The size of this index often becomes a memory bottleneck that makes it difficult to scale the system to large data sizes. To address this problem, the state-of-the-art approach is to structure this index as a succinct perfect hash table using only ≈ 4 bits per key. The downside is that the hash table encoding is computationally expensive to parse and may harm overall system performance.

We introduce Sphinx, a succinct perfect hash table reengineered for high performance on commodity CPUs. Sphinx is encoded in a manner that lends itself to efficient access using rank and select primitives, and it uses auxiliary metadata to decode common hash table slots instantaneously. Sphinx is also expandable and parallelizable. We compare Sphinx to the best alternatives and show that it leads to a 2x reduction in query latency, update latency, and memory footprint.

PVLDB Reference Format:
Sajad Faghfoor Maghrebi and Niv Dayan. Sphinx: A Succinct Perfect Hash Index for x86 . PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfmqrb/sphinx.

1 INTRODUCTION

Log-Structured Hash Tables. A Log-Structured Hash Table (LSH-Table) [2, 9, 15, 16, 49, 50] is a data structure that optimizes for storage bandwidth by making all write I/Os long and sequential. It performs all modification operations (insertions, updates, and deletes) out-of-place by first buffering them in memory. When the buffer is full, its contents are flushed into an append-only log in storage. To support queries, there is an in-memory index that maps each key to the location of the corresponding key-value entry in the log. There is also a garbage-collection mechanism to reclaim space occupied by entries that have been updated or deleted.

LSH-tables go by many colloquial names, including index+log and circular logs. The core idea traces back to the Log-Structured File System [39, 47, 48], though this idea has found myriad other applications (e.g., in cloud management [8, 38, 52] and flash translation layers [24, 28, 51]). In the database community, LSH-tables have become a popular data structure for key-value stores for SSDs [9, 25, 28], persistent memory [10, 53, 54, 57, 58], and disaggregated memory [31, 33, 59].

Fingerprint Indexes. The index of an LSH-table consumes a hefty memory footprint as it stores a mapping from every entry's key to its address in storage. To save space, many designs store a fingerprint (i.e., a short hash digest) for each key instead of the key itself [2, 9, 15, 50]. The downside of such fingerprint indexes is that they introduce false positives due to fingerprint collisions. These false positives lead to redundant I/Os that increase latency. While it is possible to limit false positives using longer fingerprints, doing so entails a higher memory footprint. As such, most existing LSH-table indexes exhibit a contention between latency and memory.

Delta Hash Table. The recent Delta Hash Table (DHT) [14] is an LSH-table index that eliminates false positives for queries to existing keys while at the same time taking less space than fingerprint indexes. DHT is a perfect hash table that resolves collisions for keys that coincide within the same hash slot. It does this by storing only the offset of the first bit that differentiates between their hash values. DHT encodes these offsets within a slot using a succinct trie. As a result, it can fully distinguish between all existing keys using ≈ 4 bits per entry.

However, DHT exhibits significant CPU overheads. The reason is that each trie is a succinct variable-length data structure that must be decoded bit by bit. These computational costs counteract the benefits of eliminating false positives, resulting in high latency. Thus, DHT was initially proposed and implemented for use by FPGAs. Yet, many applications cannot afford specialized hardware. This begs the following question: Is it possible for an LSH-table index to simultaneously achieve (1) no false positives, (2) a modest memory footprint, and (3) high computational efficiency on commodity CPUs?

Contributions. We introduce **Sphinx**: **S**uccinct **P**erfect **H**ash **In**dex for **x**86. Sphinx is a hash table that fully distinguishes between all keys in the dataset without storing the actual keys. Similarly to DHT, it uses a trie structure to differentiate between the hashes of keys that coincide within the same hash slot. However, it is engineered for speed on commodity x86 CPUs. It does so by encoding the tries in a manner that lends itself to hardware-optimized traversal using rank and select primitives. We also show that the trie structures follow a known and skewed distribution. This allows using an auxiliary data structure to decode common trie shapes instantaneously. In addition, we make the following contributions:

- We show how to expand Sphinx smoothly to scale its size in proportion to the data size.
- We support storing a variable-length number of reserve hash bits for each key in the index to reduce insertion

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

costs, save I/Os during expansions, and mitigate the cost of queries to non-existing keys.
- We resolve block overflows while maintaining a high load factor by hashing overflowing entries across several possible extension blocks.
- We compare Sphinx to a slew of other indexing data structures for LSH-tables and demonstrate its superior memory footprint and latency for queries, insertions and updates.
- We parallelize Sphinx and benchmark its performance in a multi-threaded environment.

While this paper focuses on LSH-table as its application, Sphinx can benefit any application of perfect hash tables including disaggregated key-value stores [33], OLAP engines [22], internal memory indexing [6], and packet processing [26, 35].

2 BACKGROUND

Log-Structured Hash-Table (LSH-Table). LSH-table is a data structure that stores key-value pairs and supports *put(key, value)*, *query(key)*, and *delete(key)* commands. A put command either inserts a new key-value pair or updates the value associated with an existing key. A query returns the most recent value associated with a given key. A deletion ensures that subsequent queries do not return any value associated with a given key.

Figure 1 illustrates the high-level design. All put and delete operations are handled out-of-place by first adding an entry to an in-memory buffer, typically structured as a hash table. In case of a delete, the value is a tombstone. To restrict memory footprint and recovery overheads, the buffer is assigned a limited capacity. Whenever it fills up, its entries are flushed to an append-only log in storage. This approach optimizes write throughput by ensuring that all storage writes are long and sequential.

To allow a query to find a matching data entry in the log, there is an in-memory index mapping each key to the storage location of the corresponding data entry [49]. A get command first searches the buffer for an entry with a matching key. If it does not find one, it searches the index. If it finds a matching key in the index, it follows a pointer to the location of the entry in storage.

As deletions and updates are performed out-of-place, they invalidate data entries in the log. Eventually, there is a need to reclaim space taken up by such obsolete data entries. To this end, LSH-table periodically migrates live entries from older regions of the log to the front so that these older regions can be rewritten. This process is known as garbage collection [9, 49].

Fingerprint Indexes. The choice of data structure for the index of LSH-table is crucial, affecting the system's memory footprint and overall performance. Many systems use hash tables as their index [2, 9, 15, 16, 49, 50]; however, when used to store full keys [32, 34, 49] or large key hashes [50], they consume a lot of memory.

To restrict memory overheads, most LSH-tables store a short fingerprint for each entry instead of the full key, along with a pointer to the location of the entry in storage [2, 9, 15, 50]. We refer to such indexes as fingerprint indexes. They can leverage compact data structures like Quotient [5, 18, 42, 43] or Cuckoo Filters [7, 21, 55] by storing an address alongside each fingerprint.

At the same time, fingerprint indexes introduce complications. A queried key may match a fingerprint of a different key. This is known as a false positive, and it results in redundant I/Os to storage. Such false positives degrade average and tail latency. Generally, a fingerprint index with fingerprints of length F bits entails an overhead of $1 + 2^{-F}$ storage I/Os - one to retrieve the target entry and 2^{-F} additional I/Os due to false positives.

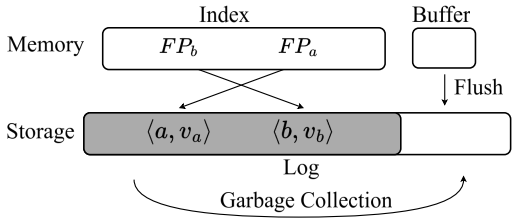

Figure 1: LSH-table streamlines data to storage and stores the location of data entries within an in-memory index.

Fingerprint matches can also occur during a put operation, either because there is already an existing entry with the same key or because of collisions with different keys (i.e., false positives). To check which of these cases occurred for a given fingerprint match, we must follow the pointer of the existing entry to storage to check its full key. If the full keys match, it means that the put operation is an update. We, therefore, modify the mapping entry in the index to point to the location of the updated key-value entry, thus invalidating the older entry. If no full key match is identified across all matching fingerprints, the put operation is an insertion, and so a new mapping entry is added to the index. We refer to this check as a read-before-write operation as it entails one storage I/O for each matching fingerprint on the write path.

Adaptive Fingerprint Indexes. Some fingerprint indexes eliminate false positives by disambiguating colliding fingerprints. They do so by enlarging [4, 56] or rehashing the conflicting fingerprint [30, 36]. These adaptations require additional metadata, which may be compressed at a cost of additional latency. Yet another approach is to add the full key of a colliding entry into a dictionary data structure rather than adding its fingerprint into the fingerprint index [33, 46]. Any operation (i.e., query, put or delete) first checks the dictionary for a full key match before checking the index for a fingerprint match. While such designs eliminate false positives and thus exhibit no redundant storage I/Os, they require using longer fingerprints to prevent too many fingerprint collisions in order to bound the dictionary's size.

Expanding Fingerprint Indexes. As more data is inserted into an LSH-table, its fingerprint index must expand to accommodate more mapping entries. However, a fingerprint index cannot rehash its mapping entries into a larger hash table because it does not store the original keys. As a result, many fingerprint indexes are pre-allocated with a large, fixed capacity from the get-go.

Recent expandable filters such as InfiniFilter [13] and Aleph Filter [12] address this issue. They expand by transferring one bit from each fingerprint to the entry's slot address to remap the entry into a 2x larger hash table. Crucially, they also support variable-length fingerprints to allow setting longer fingerprints to newer entries. This keeps the false-positive rate stable. We use them in our experimental evaluation in Section 4 to represent the most competitive fingerprint indexes.

2.1 Delta Hash Table

The Delta Hash Table (DHT) is a recent index for LSH-tables designed to eliminate false positives while taking up less memory than fingerprint indexes. DHT is a dynamic perfect hash table that resolves all collisions using succinct tries.

DHT is an array of fixed-sized blocks. Each block contains a fixed number of adjacent variable-length slots (64 by default). DHT stores an average of at most one entry per slot. However, each slot can contain zero or more entries to accommodate variability in the number of entries that get hashed to different slots. We refer to DHT as being *at capacity* when the overall number of entries equals the overall number of slots.

Keys to be inserted to DHT are hashed into a 16-byte digest consisting of three parts: the block ID, slot ID, and fingerprint. The block ID and slot ID map the entry to a given slot within a given block. The fingerprint is used for collision resolution when multiple keys reside within the same slot and spans the rest of the hash. Overflowing blocks are managed using chaining. Since slots are variable-length while blocks are fixed-length, a block must be scanned to locate the slot corresponding to a given entry.

Delta Trie. To resolve hash collisions, any slot containing two or more entries employs a data structure called delta trie. A delta trie distinguishes the fingerprints within a slot without storing the full fingerprints. Each internal node in this trie represents the most significant fingerprint bit that distinguishes between the two sets of fingerprints in its left and right sub-trees. Figure 2 Part (A) gives an example of a delta trie containing five fingerprints FP_1 to FP_5. The root node directs FP_1 and FP_3 to the left sub-tree and FP_2, FP_4, and FP_5 to the right sub-tree due to the difference in their most significant bit. Moving down, FP_1 and FP_3 diverge at the third bit (i.e., at Node *b*). Similarly, FP_4 and FP_5 diverge from FP_2 at Node *c* due to the difference in their second bit. Finally, FP_4 and FP_5 are different along their fourth bit at Node *d*. The resulting order of leaf nodes in the trie reflects sorting based on the keys' fingerprints.

Physical Slot Encoding. A slot and its delta trie are succinctly encoded to conserve space. Figure 2 Part (B) shows the encoding of the delta trie from Figure 2 Part (A). The encoding is composed of three self-delimiting fields: size, structure, and index.

Size Field. The size field counts the number of entries in the slot using 0-terminated unary encoding. It uses unary encoding since each slot contains an average of at most one entry. This keeps the field small (i.e., at most 2 bits on average).

Structure Field. This field encodes the topology of the delta trie. Each internal node is represented using two bits: one for the left child and one for the right. A bit is set to `1` if the corresponding child is an internal node and `0` if it is a leaf. These bits are arranged in depth-first order, starting from the root. In Figure 2 Part (B), for example, the initial two bits of the structure field (i.e., `11`) correspond to the root node and indicate that both of its children are also internal nodes. The following two bits `00` for Node b indicate that both of its children are leaves. Node c's bits are `01` as only its right child is an internal node. For the last internal node d, the bits should be `00` as both of its children are leaf nodes, yet we do not need to explicitly encode these bits. The reason is that the last internal node in a depth-first order always has two leaf children, and so we avoid encoding these bits to save space. Thus, the structure field is materialized only for slots with three or more entries.

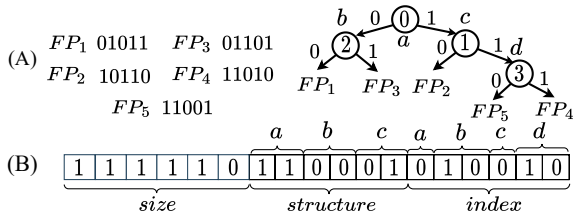

Figure 2: (A) depicts five keys sorted by the first differentiating bit of their fingerprints (FP_i), with the fingerprints on the left and the corresponding delta trie on the right. (B) illustrates the physical encoding of the delta trie.

Index Field. For each internal node, the index field encodes the position of the most significant (leftmost) bit that differentiates the fingerprints in its two sub-trees. Similarly to the structure field, it is arranged in depth-first order. For the trie in Figure 2, for example, the values to be encoded by this field are 0, 2, 1 and 3 in that order.

Note that the index field for any internal node other than the root is greater than the index field of its parent by at least 1. Thus, for each non-root internal node, DHT only encodes the difference between that node's index and its parent. This is done using unary encoding where `0` denotes a difference of 1, `10` denotes 2, `110` denotes 3, etc. In Figure 2 Part (B), for example, the encoding is `0` for Node a, `10` for Node b, `0` for Node c, and `10` for Node d.

Full Block Example. Figure 3 Part (B) depicts a block with four slots storing the six entries in Part (A). The entries get mapped to each of the four slots based on the Slot IDs of their hashes. As shown, Slot 0 contains one entry, Slot 1 contains two entries, Slot 2 is empty, and Slot 3 contains three entries. Only Slots 1 and 3 require a delta trie (Part (D) of the figure) as they each contain two or more entries. This means that for Slots 0 and 2, there are no structure or index fields. For Slot 1, there is no need to store a structure field either because it only contains two entries. The slots are stored contiguously within the block. For each slot, the size, structure, and index fields are adjacent. While these fields are variable-length, each of them is self-delimiting, thus uniquely decodable.

Trie Store and Payload Store. The size, structure and index fields for the different slots are stored within a part of the block that we refer to as the Trie Store. In Figure 3 Part (B), the Trie Store consists of 18 bits, and its last two bits are unused. The Trie Store effectively orders all entries within the block based on their hashes. It is followed by a Payload Store, which contains pointers to the location of the corresponding data entries in storage. The payloads are ordered based on the hashes of their corresponding keys. Thus, to obtain the payload associated with a given key, we can infer the offset of the entry's hash within the sorted order from the Trie Store and fetch the matching payload from the Payload Store.

Guarantees. When DHT is at capacity, most slots only contain zero or one entry. Hence, most slot encodings are small, allowing DHT's Trie Store to achieve 3-4 bits/entry memory footprint at capacity [14]. In addition, DHT exhibits no false positives when querying for existing keys since it fully distinguishes between all existing keys' hashes. In contrast, fingerprint indexes require more memory and incur false positives when querying for existing keys.

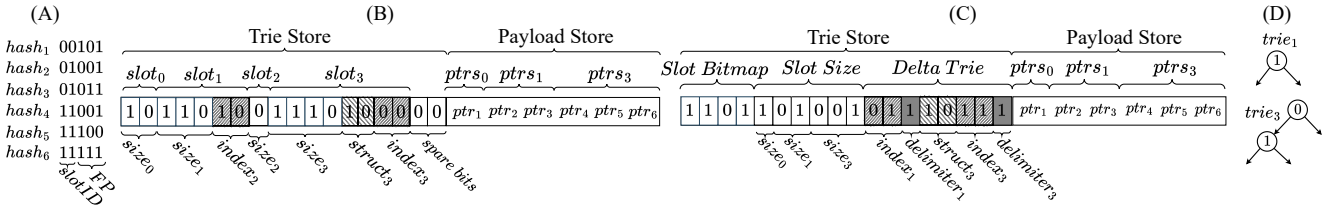

Figure 3: (A) shows the hashes of six keys, where the first two bits encode the slot ID and the rest form the fingerprint. (B) depicts the DHT block design with 4 slots, encoding these keys in the Trie Store and storing sorted payloads in the Payload Store. (C) presents the Sphinx block design with 4 slots, storing the same keys in the Trie Store and sorted payloads in the Payload Store. It consists of a 4-bit *Slot Bitmap Segment*, indicating whether the trie is empty, followed by a *Slot Size Segment* for slot sizes and a *Delta Trie Segment* encoding the topology of each trie. (D) shows the corresponding tries for slot$_1$ and slot$_3$.

Payload Memory Overhead. While the Trie Store is highly memory-efficient, the Payload Store's pointers to the log could significantly increase the memory overhead, overshadowing index's low memory cost by pointer storage. To address this, the same work that introduces DHT [14] also proposes a pointer compression scheme.

The pointer compression scheme relies on sorting the contents of the LSH-table's buffer by the hash of each key. Consequently, the storage-based log comprises multiple *chunks* of sorted entries, each created by a single buffer flush. Each key is indexed to the ID of the chunk containing it and the index that does this is called the Global Index. Since chunks are larger than a single page, each chunk can be uniquely identified using fewer bits than a standard pointer. Within each chunk, another index (a DHT instance) maps each key to the specific page containing it, known as the Local Index. Because entries in the log are sorted by hash, they follow the same order as the local DHT instance, allowing pointers within it to be delta encoded. Specifically, for each entry, the page offset is stored in unary encoding relative to the preceding entry. Given that each entry is typically smaller than a disk page, the average size of this unary encoding is only 1-2 bits per entry.

This compression scheme reduces the average pointer size from 4 bytes to under 2 bytes. With smaller pointers, index metadata becomes a significant contributor to the overall memory footprint. Although orthogonal to our focus on computational challenges, we implement this scheme on top of Sphinx, motivating our total memory footprint. We detail the memory footprints in Section 4.

Sizing Blocks. As with any hash table design, the number of entries mapped to each block is variable and approximately follows a Poisson distribution [29, 45]. This variability poses a challenge in achieving high memory utilization, as some blocks may overflow while others remain underutilized. The DHT design further exacerbates this issue by making the slots variable in length. To address this challenge, DHT employs large blocks, each containing dozens of slots (64 slots per block in practice). This approach reduces the variability in the cumulative size of the slots and enables high block utilization while limiting the probability and length of overflows.

Core Challenge: Computational Overheads. DHT's many memory optimizations come at the cost of computational efficiency. To access a specific slot, all preceding slots within the block must be traversed first. To make things worse, each slot and its constituent fields are of variable length, requiring a bit-by-bit traversal to decode. This process entails many branch mispredictions, which further increase CPU costs. Due to these overheads, DHT was originally designed for FPGAs [14]. However, many applications cannot afford customized hardware. This begs the following research question: is it possible for an LSH-table index to achieve a modest memory footprint (3-4 bits per entry), no false positives, and high computational efficiency, all at the same time?

Additional Challenges. In addition to the above core challenge, DHT exhibits several other limitations relative to fingerprint indexes. (1) DHT is statically sized and cannot expand in proportion to the data size. As a result, it is allocated with a fixed capacity from the system's get-go and consumes a lot of space until it fills up. This can defeat the point of the many memory optimizations it proposes. (2) As only the least number of necessary fingerprint bits are preserved to distinguish between existing keys, a query targeting a *non-existing key* has a higher chance of matching some existing entry, leading to a false positive. (3) Similarly, as fewer fingerprint bits are maintained for existing entries, an insertion is more likely to match some existing key's hash and thereby result in a read-before-write operation. Sphinx systematically tackles each of the above problems.

3 SPHINX

Sphinx is a succinct dynamic perfect hash table that improves on DHT along several fronts. It is encoded succinctly yet in a manner that lends itself to CPU-friendly rank/select commands (Section 3.1). It introduces the notion of a Decoder, a compact, cache-resident data structure containing precomputed metadata that swiftly decodes the bitwise representation of common delta tries (Section 3.2). It maintains a high load factor by grouping blocks into block groups, each with its own extension blocks, into which overflowing entries are hashed and stored (Section 3.3). It limits I/Os for insertions and zero-result queries by storing extra fingerprint bits, referred to as Reserve Bits, alongside each entry to quickly rule out the existence of a key in question (Section 3.4). It uses a fine-grained expansion algorithm that limits I/O by leveraging fingerprint metadata within the index (Section 3.5). It supports multi-threading (Section 3.6).

3.1 Block Structure and Efficient Parsing

Similarly to DHT, Sphinx consists of fixed-sized blocks of variable-length slots. Unlike DHT, Sphinx's block design lends itself to efficient rank/select commands. The *rank(b, i)* command counts the

Algorithm 1 Find the Payload corresponding to a Key

```
    Inputs:
        hash: hash of the queried key
    Output: payload index corresponding to the queried key, −1 if no match is found
 1: blockID, slotID, FP ← extract(hash)
 2: trieStore, payloadStore ← Blocks[blockID]
 3: if not trieStore[slotID] then
 4:     Return −1
 5: end if
 6: occupiedSlotsBefore ← rank(trieStore[0 : SLOTCNT − 1], slotID)
 7: numEntriesBefore ← select(trieStore[SLOTCNT: ], occupiedSlotsBefore) + 1
 8: numEntriesEnd ← select(trieStore[SLOTCNT: ], occupiedSlotsBefore + 1) + 1
 9: slotSize ← numEntriesEnd − numEntriesBefore
10: if slotSize = 1 then
11:     Return numEntriesBefore
12: end if
13: occupiedSlots ← rank(trieStore[0 : SLOTCNT − 1], SLOTCNT)
14: onesSlotBitmapSeg ← occupiedSlots
15: onesSlotSizeSeg ← occupiedSlots
16: onesDeltaTrieSeg ← 2 × (numEntriesBefore − occupiedSlotsBefore)
17: onesBefore ← onesSlotBitmapSeg + onesSlotSizeSeg + onesDeltaTrieSeg
18: numOnesDeltaTrie ← 2 × (slotSize − 1)
19: startDeltaTrie ← select(trieStore, onesBefore) + 1
20: endDeltaTrie ← select(trieStore, onesBefore + numOnesDeltaTrie)
21: index ← decodeDeltaTrie(trieStore, FP, startDeltaTrie, endDeltaTrie, slotSize)
22: Return payloadStore[numEntriesBefore + index]
```

number of set bits (1s) from the start of Bitmap b up to and excluding Index i. The *select(b, i)* command finds the offset of the i^{th} set bit in Bitmap b. These operations can be efficiently implemented using specialized CPU instructions, as shown later in this section. We show how to use these commands to traverse a block one word at a time rather than one bit at a time. The main challenge is encoding the block in a way that allows us to quickly skip to the correct slot without having to traverse every preceding bit. Our new encoding uses slightly more space—roughly 0.3 bits per entry—but improves parsing speed by an order of magnitude on commodity CPUs.

Structure Overview. Sphinx is organized into 2^x blocks, each containing 2^y slots. To map a key, the first x bits of its hash are used to identify the block, followed by the next y bits to identify a slot within that block. Each block consists of a Trie Store and a Payload Store. The Trie Store succinctly maps a key's hash to its offset within the sorted order of hashes in a block. The Payload Store organizes the payloads of all keys in the block contiguously based on the order of the keys' hashes. A payload is a pointer to the corresponding data entry in storage.

A block consists of 64 slots, each storing an average of at most one entry. The large number of entries per block restricts occupancy variability and keeps overflows small on average. Each Trie Store consists of 256 bits to be both word and cache aligned. A Trie Store occupies ≈ 4 bits/entry.

Query Algorithm & Running Example. Algorithm 1 takes the hash of a key as input. Its output is the corresponding payload in the Payload Store. To find this offset, the algorithm traverses the Trie Store to find this payload's offset. At a high level, Lines 3-9 obtain the number of entries in the target slot, while Lines 13-20 find the starting and ending position of the slot's delta trie if it exists. The notation $b[x : y]$ represents a substring of Bitmap b beginning at the x^{th} bit and including all bits up to and including the y^{th} bit. SLOTCNT denotes the number of slots in a block (i.e., 64 slots).

We explain the query algorithm and block structure with the running example using Figure 3 Part (C). The figure illustrates a small block consisting of 18 bits and four slots, i.e., 4.5 bits per slot on average. These slots contain one, two, zero, and three entries, respectively. Suppose the user queries for a key corresponding to $hash_5$ = `11100` in Figure 3 Part (A). The first two bits, `11` = 3, map the key to Slot 3. The remaining three bits, `100`, form the fingerprint. The hashes $hash_4$, $hash_5$, and $hash_6$ are all mapped to Slot 3. $Hash_5$ has an offset of 1 within the target slot and an offset of 4 within the block as a whole.

Slot Bitmap Segment. Each Trie Store begins with a Slot Bitmap, which contains one bit per slot to indicate if it is non-empty. In Figure 3 Part (C), the bitmap is set to `1101`, indicating that Slot 2 is empty and the rest are non-empty. Lines 3–5 of Algorithm 1 check the bitmap and terminate the algorithm if the target slot is empty. Otherwise, we obtain the number of occupied (i.e., non-empty) slots preceding the target slot using a rank command (Line 6). In our running example, Line 6 returns 2, indicating that there are two non-empty slots before Slot 3, our target.

Slot Size Segment. After the Slot Bitmap, Sphinx stores the number of entries in each non-empty slot using a 1-delimited unary code. These codes are laid out contiguously, with the length of each unary code matching the number of entries in the corresponding slot. We refer to each code as the size field of the corresponding slot. In Figure 3 Part (C), $size_3$ represents the number of entries in Slot 3. It is encoded as `001`, indicating that this slot contains three entries.

Lines 7 to 9 of Algorithm 1 employ select commands to find the delimiters of the size fields of the target slot and the slot immediately preceding it. By adding one to each of these delimiter positions, we obtain the number of entries before the target slot and the number of entries up to the end of the target slot, respectively (i.e., 3 and 6 in the running example). By subtracting these values in Line 9, we obtain the target slot size (i.e., 3 in our example).

For a slot with only one entry, there is no delta trie. Thus, there is no need for further traversal of the Trie Store; the payload index can be inferred directly based on the starting offset of the target slot. Lines 10–12 verify if the current slot has exactly one entry, returning the count of preceding entries as the payload index. In our running example, Line 11 is skipped because Slot 3 contains multiple entries.

Delta Trie Segment. For slots with two or more entries, Sphinx stores a delta trie, which consists of a structure field and an index field. As shown in Figure 3 Part (C), these fields are laid out adjacently for each slot and contiguously within the Trie Store. The challenge is to find the starting and ending bit offsets of these fields for a given slot. Our solution is to infer the number of 1s within the Trie Store prior to the target slot's delta trie. We then employ the select command to skip directly to the target delta trie based on the number of prior 1s.

Skipping the Slot Bitmap and Slot Size Segments. The number of 1s within the Slot Bitmap Segment and Slot Size Segment is each equivalent to the number of occupied slots in the block. In Line 13 of Algorithm 1, we count the number of occupied slots within the Trie Store using a rank command on the Slot Bitmap Segment. We record that this is the number of 1s to be skipped within the Slot Bitmap Segment and Slot Size Segment in Lines 14 and 15.

In our running example, the rank command in Line 13 indicates that there are three occupied slots within the block. We indeed

Algorithm 2 Rank Implementation over 256 bits

```
inputs: bitmap B, bit position pos
1: numSetBits ← 0, offset ← 0
2: for each 64-bit word W in B:
3:     pop ← POPCNT(W)
4:     if offset + 64 ≥ pos then Return rank64(W, pos − offset) + numSetBits
5:     numSetBits ← numSetBits + pop
6:     offset ← offset + 64
```

Algorithm 3 Select Implementation over 256 bits

```
inputs: bitmap B, target bit i
1: numSetBits ← 0, offset ← 0
2: for each 64-bit word W in B:
3:     pop ← POPCNT(W)
4:     if numSetBits + pop ≥ i then Return select64(W, i − numSetBits) + offset
5:     numSetBits ← numSetBits + pop
6:     offset ← offset + 64
```

observe that the total number of 1s within the Slot Bitmap and Slot Size Segments is $3 \cdot 2 = 6$.

Number of 1s in a Delta Trie. Next, to reach the beginning of the target delta trie, our parsing algorithm must infer how many 1s there exist across all delta tries of slots preceding the delta trie of the target slot. As a building block for this process, we must infer the number of 1s in a delta trie based on the number of entries within the corresponding slot.

Unlike DHT, each unary counter within the index field in Sphinx is 1-delimited, so it contains exactly one bit set to 1. For a delta trie with n entries, there are $n-1$ internal nodes and, thus, $n-1$ unary codes within the index field. Hence, for a slot with n entries, the index field contains $n-1$ bits set to 1s.

Next, recall that the structure field exists only for a slot that contains three or more entries. When there are exactly three entries, this field is encoded as `10` or `01`, meaning there is one set bit. Each additional entry in the slot adds one internal node and thus one additional set bit. For a slot with n entries, the structure field contains $n-2$ set bits.

Overall, for a slot with n entries, we have a total of $2n-3$ set bits when $n \geq 2$, and 0 otherwise. For example, the delta trie for Slot 3 in Figure 3 Part (C) is represented as `1011`, where the first two bits correspond to the structure field and the latter two represent the index field. In this example, we have $n = 3$. Our formula correctly indicates that there are $2n-3 = 3$ set bits within Slot 3's delta trie.

Delimiter Field. For a slot with two or more entries, the expression $2n-3$ accurately provides the number of 1s that the slot's delta trie contributes to the Delta Trie Segment. For a slot with only one entry, however, there is no delta trie. Such slots do not contribute any 1s to the Delta Trie Segment. The issue is that for a slot with 1 entry, the expression $2n-3$ yields -1 instead of 0. Hence, the expression $2n-3$ cannot be used out-of-the-box to predict how many bits within the Delta Trie Segment to skip, as it would underestimate the number of 1s before the target slot on account of slots containing one entry.

To fix this problem, we introduce one additional bit to delimit each delta trie within the Delta Trie Segment. In Figure 3 Part (C), there are two delta tries within the Trie Store, each of which is delimited by a 1. Accounting for this delimiter, the number of bits set to 1 within a delta trie becomes $2n-2$ for $n \geq 1$. Crucially, this expression correctly returns 0 when $n = 1$, capturing the fact that a slot with a single entry does not have a delta trie. The delimiter field keeps each slot uniquely decodable as it consists of a single bit in a predictable location.

Using this new expression, we can compute the number of 1s preceding the target delta trie within the Delta Trie Segment on the basis of (1) the total number of entries preceding the target slot from Line 6, and (2) the number of occupied slots preceding the target slot from Line 7. We do so in Line 16 of Algorithm 1. In Line 17, we add up the total number of 1s in the Trie Store as a whole before the target delta trie.

In our running example, there are three preceding entries and two occupied slots before the target slot. Plugging these numbers into Line 16, we infer that there are $2 \cdot (3-2) = 2$ bits to skip within the Delta Trie Segment prior to the target slot's delta trie. Including the 1s from the Slot Bitmap and Slot Size Segments, there are exactly 8 bits set to 1 preceding the target delta trie (Line 17).

Finding The Delta Trie Boundary. Line 19 of Algorithm 1 employs the select command to derive the offset of the target slot's delta trie. We also locate the end of the target delta trie in Line 20 by employing a select command on the sum of the number of preceding 1s and the number of 1s within the target delta trie from Line 18. In our example, Line 19 indicates that the target delta trie begins at bit offset 13 within the Trie Store. From Line 18, we know that the number of bits within the target delta trie is 4. We use this to infer that the end of the target run is at offset 17 in Line 20.

Inserts/Updates. When writing an entry, Sphinx reuses much of the logic and code from Algorithm 1. It calculates the target slot's size (Lines 6–9) using constant-time rank/select commands. If the slot is empty, it flips its bit in the Slot Bitmap Segment to 1, inserts a 1 into the Slot Size Segment at the offset found in Line 7, and inserts the new payload into the Payload Store at the same offset.

If the slot is non-empty, Sphinx again uses hardware-optimized rank/select commands to locate the start of the trie (Lines 13–19). Since tries only disambiguate already-inserted keys and cannot tell whether the incoming key is new or an update, Sphinx performs a single read-before-write, incurring one I/O. It then either overwrites the existing payload or, for an insertion, identifies the first differentiating bit between hashes, updates the trie, and inserts the payload at the offset in Line 22. Note that the trie is shallow and not too costly to update. To make room for the revised trie and slot size, Sphinx performs fast shifts over the 256-bit confined block, operating at word granularity with bitwise masking to preserve unchanged bits. It also uses compiler-optimized AVX-512 `memmove` operations to move the payloads to clear space for an insertion.

Since insertion or update follows a fixed sequence of instructions over a bounded number of words, it runs in constant time. Expansions do occur only after exponentially many inserts, yielding amortized constant complexity (see Section 3.5).

Deletions. Deleting a key acts similarly to the insertion algorithm. Sphinx issues exactly one read-before-write to confirm the presence of the key in question in the log. On a match, it decrements slot's size in the Slot Size Segment, clears the slot bit in the Slot Bitmap Segment if it becomes empty, and trims the corresponding trie. Finally, it reclaims space by shifting the block's content left using the same hardware-optimized shift explained before. Like an insertion, a deletion runs in constant time.

Fast Rank and Select. Rank and select on a 64-bit word can be implemented efficiently using x86 bit manipulation instructions [23, 41, 42]. The implementations are as follows:

$$\text{rank64}(word, pos) = \text{POPCNT}(word \,\&\, (2^{pos} - 1))^1$$

$$\text{select64}(word, i) = \text{TZCNT}(\text{PDEP}(2^{i-1}, word))^2$$

For Sphinx, we extend these commands to operate over a 256-bit Trie Store in Algorithms 2 and 3. For the rank command, given a position *pos*, the algorithm iterates over each 64-bit word of B using the POPCNT instruction to count the number of set bits until it locates the word containing the pos^{th} bit. We then apply the rank64 command on this word and return the number of 1s seen so far.

For the select command, Algorithm 3 also first iterates over each 64-bit word. It counts the number of bits within them using POPCNT until it finds the word containing the i^{th} set bit based on the number of 1s seen so far. We then apply the select64 command on this word and return the i^{th} bit's overall offset.

To further optimize these commands, we consolidate multiple calls into a single traversal of each block, eliminating redundant instruction execution. In other words, rather than executing separate rank/select commands, like those in Lines 19 and 20 of Algorithm 1, we merge them in a way that considers previously calculated intermediate values. As a result, we enable a single-pass iteration per block to retrieve all required information.

We also attempted to accelerate these commands using SIMD vectorization. However, since Algorithm 2 and Algorithm 3 operate on at most four words with a few parallelizable instructions, the added overhead resulted in higher latency. We also experimented with loop unrolling for Algorithms 2 yet found that the compiler already does that automatically.

Memory Analysis. The only space overhead that Sphinx adds to the Trie Store compared with DHT is the delimiter field. To analyze this overhead, note that the number of entries per slot follows a Poisson distribution with a mean $\lambda \leq 1$ (with $\lambda = 1$ at capacity). Hence, the probability that a slot requires a Delimiter Bit is $1 - \Pr[X \leq 1] \leq 0.28$. Rounding up, each entry contributes an average of at most 0.3 bits/entry to the Trie Store. In theory, this slightly increases the propensity of a Trie Store to overflow, though the impact is practically negligible as shown in Section 4.

Summary. By adopting the new block structure and parsing the Trie Store word by word rather than bit by bit, we can find a slot's delta trie far more efficiently. Next, we focus on how to decode the contents of a delta trie efficiently once we have found it.

3.2 Decoding Delta Tries

Once a query arrives at a slot with two or more entries, it must decode the slot's trie to infer the correct offset of the entry's payload in the Payload Store. This section shows how to decode most tries without a bit-bit-bit traversal or incurring branch mispredictions. We exploit the fact that the slots in Sphinx follow a Poisson distribution, similar to any hash table. Hence, most tries only have 2-4 entries while larger tries are vanishingly rare ($\approx$ 1%). Thus, we design a data structure called a Decoder that allows to quickly decode the most common delta tries while resorting to a bit-by-bit traversal only for uncommon tries. The Decoder is a small hash table wherein each entry consists of three fields.

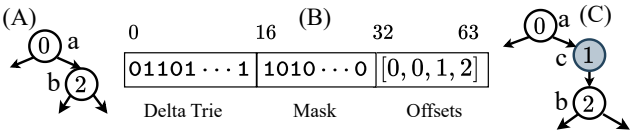

Figure 4: The trie with three entries in Part (A) has the corresponding Decoder entry in Part (B). Part (C) depicts the same trie as in Part (A) with a non-differentiating node c.

Delta Trie Field. The first field is the 16-bit key of the hash table. It contains delta trie encodings as they appear in the Trie Store. For example, consider the trie in Figure 4 Part (A). The structure field of this trie is `01` while the indexes field is `101`. Appending these, the overall encoding is `01101`. We pad this encoding with eleven 1s to fill up the length of the field, as shown in Part (B) of the figure.

Mask Field. The Mask Field is a bitmap consisting of 16 bits. The bit at offset i is set to 1 for every differentiating bit offset within the trie. For example, in Figure 4 Part (A), the bits at offsets 0 and 2 differentiate between the three entries in this trie. Hence, the bits at these two offsets are set to 1s in the corresponding mask field. We use this bitmap to extract the relevant bits from the fingerprint of a queried key using the Parallel Extract Command (PEXT), which extracts bits from its first operand (i.e., the fingerprint) using the Mask given as the second operand.

Offsets Field. The Offsets Field contains an entry for every possible permutation of the differentiating bits of the trie. Each permutation corresponds to one possible setting of these bits, which maps a fingerprint with this permutation of differentiating bits to an offset within the trie's sorted order. In Figure 4 Part (B), for example, the bits at offsets 0 and 2 of a queried key's fingerprint may be 00, 01, 10 or 11. The Offsets Field indicates that for each of these permutations, the corresponding offset is 0, 0, 1 and 2, respectively.

Query Workflow. Suppose a query searches the Trie Store as explained in Section 3.1 and arrives at a delta trie such as the one in Figure 4 Part (A). We first pad the trie encoding with 1s and search the Decoder for a matching entry. If we find one, we apply the PEXT command on the key's fingerprint and the decoder entry's mask. For example, if the key's fingerprint begins with `100`, applying the PEXT command on it with the mask in Part (A) yields `10`, or two, as the result. We then access the entry at index 2 of the Offsets field to obtain 1, meaning that the queried key maps to offset 1 within the slot. We add this to the number of preceding entries in the block to obtain the matching offset in the Payload Store.

If a query does not find a matching Decoder entry (i.e., in $\approx$ 1% of the cases), it resorts to a bit-by-bit traversal of the delta trie as described in the DHT paper [14].

Implementation & Size. As the decoder is static and only needs to be constructed once prior to the system's deployment, we do a brute-force search for a hash function that causes each decoder entry to map to a unique bucket. To allow this brute-force search to succeed in tractable time, we over-provision the hash table by 75% (i.e., 512 buckets and 128 valid entries). Despite this over-provisioning, the decoder only takes up to 4KB and thus comfortably fits in the L1 cache. At the same time, the decoder is collision-free, meaning it does not incur the typical overheads and branch mispredictions of collision resolution methods such as chaining or linear probing.

We devised a hash function using only XOR, bit shifts, and AND operations, which are cheaper than the multiplication, addition, or division used in standard hash functions.

Selecting Most Frequent Tries. Given a delta trie, we aim to determine its probability to decide whether caching its decoder is beneficial. This probability is given by the following formula, which we break down and illustrate step-by-step:

$$P = \left(\frac{e^{-1}}{n!}\right) \cdot \left(\prod_{j=1}^{m} 2^{-k^{(j)}+1}\right) \cdot \left(\prod_{i=1}^{n-1} \binom{k_l^{(i)} + k_r^{(i)}}{k_l^{(i)}} \cdot 2^{-(k_l^{(i)}+k_r^{(i)})}\right). \quad (1)$$

Firstly, the number of entries (n) follows a Poisson distribution with a mean of 1 at capacity [37], giving us the leftmost term in Equation 1. For instance, Figure 4 Part (C) contains 3 entries, yielding $\frac{e^{-1}}{3!} \approx 0.06$.

Next, we consider the trie topology, composed of nodes that either differentiate between fingerprints (differentiating nodes) or pass them along to deeper differentiating nodes. Non-differentiating nodes, through which $k > 1$ fingerprints pass, require all fingerprints to share the same bit at the node's index. Thus, the probability for each non-differentiating node is $2^{-k} \cdot 2$, giving us the second term in Equation 1 after accounting for all m of them. In Figure 4 Part (C), Node c is the only non-differentiating node with two entries ($k = 2$), resulting in probability $2^{-2+1} = \frac{1}{2}$.

For a differentiating node, let k_l and k_r represent the number of fingerprints going left (bit 0) and right (bit 1), respectively. We select k_l fingerprints from a total of $k_l + k_r$, yielding $\binom{k_l+k_r}{k_l}$ possibilities, each with a probability of $2^{-(k_l+k_r)}$. The product of all $n-1$ differentiating nodes gives us the final component of Equation 1. For instance, at Node a, one fingerprint goes left and two right, resulting in probability $\binom{3}{1} \cdot 2^{-3} = \frac{3}{8}$. Similarly, Node b, with one fingerprint on each side, has probability $\binom{2}{1} \cdot 2^{-2} = \frac{1}{2}$.

Therefore, the total probability of the trie in the example is the product of all these calculated probabilities: $0.06 \cdot \frac{1}{2} \cdot \frac{3}{8} \cdot \frac{1}{2} = 0.0056$. We store within the decoder an entry for each of the 128 most frequent tries that occur.

3.3 Overflow Management

When blocks reach their size limit, block overflows occur. DHT uses chaining, which increases cache misses. In contrast, Sphinx groups several blocks (i.e., a *block group*) and collectively manages their overflows chain-free. Overflowing slots are redirected to a few blocks (i.e., *extension blocks*), keeping them near their original blocks, eliminating pointers and improving memory utilization.

By default, a block group comprises 64 adjacent blocks and serves as the unit of expansion: when an extension block overflows, its block group expands (see Section 3.5). A directory array [20, 34] tracks these groups by their IDs (i.e., *group IDs*). During an overflow, Sphinx relocates overflowing slots one by one into an extension block until the original block can contain the remainder. The count of overflowing slots is encoded in unary at the end of the original Trie Store: Bit 255 starts as 1 to signal no overflow; each overflow shifts the set bit left (`10`, `100`, etc.). TZCNT [1, 27] instantly counts overflows by counting trailing zeros.

Extension Blocks. By default, each group contains four extension blocks. Each extension block consists of an *ExBlock* (i.e., a regular block that stores overflowing slots) and a 128-bit *Metadata* bitset

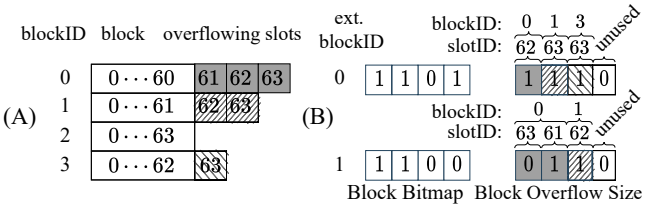

Figure 5: (A) shows 4 blocks, where blocks 0, 1, and 3 have three, two, and one overflowing slots. (B) depicts how the two extension blocks organize their Metadata for these six slots.

divided into two 64-bit fields: the *Block Bitmap*, which marks which of the group's 64 blocks overflow into this extension block, and the *Block Overflow Size*, which records how many slots each block overflows here. The ExBlock stores overflowing slots in the same order indicated by the Metadata, allowing their new slot IDs to map directly to Metadata positions. Metadata decoding relies on the same rank/select commands as Lines 6-9 of Algorithm 1.

The number of extension blocks per group is tunable: too few cause premature expansion, while too many waste space. To find the sweat spot, we estimate the average number of overflowing entries per Trie Store and Payload Store, separately. In the Trie Store, each block holds 64 entries, each using under ≈3.3 bits; an additional 0.7 bits per entry is reserved to absorb variation, so the expected overflow is under 0.2 entries, which is effectively negligible. In the Payload Store, each block can hold up to 64 payloads. Under a Poisson distribution with $\lambda = 64$, the expected overflow is $E[(X - 64)_{X>64}] \approx 3.19$, where X is the number of payloads in the Payload Store. With 64 blocks per group and 64 payload slots per extension block, $\lceil \frac{3.19 \times 64}{64} \rceil = 4$ extension blocks effectively cover overflows.

When Slot j in Block i (with $0 \le i, j < 64$) overflows, we assign it to Extension Block $(i + j) \bmod 4$ and insert it immediately after any prior overflows from smaller block IDs (or, if in the same block, larger slot IDs). This deterministic scheme evenly spreads overflows across the four extension blocks per block group, maintaining a 95% per-group load factor before expansion.

As a simplified example, Figure 5 shows four blocks and two extension blocks. Each overflowing slot spills to Extension Block $(i+j) \bmod 2$. Blocks 0, 1, and 3 have three, two, and one overflowing slots, respectively. In Block 0, Slots 63, 62, and 61 are redirected to ExBlock 1, 0, and 1; in Block 1, Slots 62 and 63 go to ExBlock 1 and 0; and Block 3's single overflow is assigned to ExBlock 0.

3.4 Reserve Bits

The design of Sphinx as we have seen it so far only encodes the bits that differentiate among the hashes of existing keys. This approach requires ≈ 4 bits / entry while ensuring at most 1 I/O for updates/deletes/queries to existing keys. This is a significant improvement over Fingerprint filters, which require at least F bits / entry and thus entail $1+2^{-F}$ read I/Os per each queries/updates/delete. However, Sphinx so far also has a disadvantage relative to fingerprint filters. A query to a non-existing key or an insertion of a new key will always match some existing key and lead to one read I/O, similarly to DHT [14]. In contrast, fingerprint filters only perform a read I/O with a probability of 2^{-F} in case of a false positive for these operations. To bridge this gap, we augment Sphinx with a

Table 1: Average number of read I/Os per operation. F denotes the number of fingerprint bits stored.

Operation	fingerprint filters	DHT	Sphinx
Query (existing key)	$1 + 2^{-F}$	1	1
Query (non-existing key)	2^{-F}	1	2^{-F}
Insertion	2^{-F}	1	2^{-F}
Update	$1 + 2^{-F}$	1	1
Deletion	$1 + 2^{-F}$	1	1

few extra hash bits for each key, i.e., Reserve Bits, to improve the performance of insertions and zero-result queries for applications that require lower overheads for these cost metrics.

The Reserve Bits are the highest-order bits of the fingerprints, i.e., the bits of the key's hash that come right after the slot address. Checking these Reserve Bits during an insertion and query allows ruling out the existence of the key and thus avoid a storage access. Sphinx allows storing a configurable number of Reserve Bits. Using F Reserve Bits reduces the read I/O cost per insertion and zero-result query from 1 to 2^{-F} I/Os on average. Thus, the Reserve Bits allow Sphinx to navigate the same trade-offs as fingerprint filters with respect to memory vs. the I/O cost of insertions and zero-result queries. At the same time, Sphinx is still different from fingerprint filters in that the performance of updates/delete/queries to existing keys is still strictly one read I/O as these operations use the Trie Store rather than the Reserve Bits. Table 1 summarizes these costs for fingerprint filters, DHT, and Sphinx. As shown, Sphinx enjoys the best of both worlds from DHT and fingerprint filters.

Sphinx stores the Reserve Bits for each entry adjacently to its payload within the Payload Store. Thus, accessing the Reserve Bits does not entail an additional cache miss. In the next subsection, we describe the particular encoding for the Reserve Bits and how Sphinx uses it to improve expansion efficiency.

3.5 Expansion

As more entries are inserted into Sphinx, it eventually expands to accommodate more entries. Sphinx expands at the fine granularity of a block group (see Section 3.3) to prevent performance slumps, similarly to extendible hashing [20]. When a block group is full, we remap its entries into two separate block groups, and we keep track of all block groups using an expandable directory. However, Sphinx faces a challenge for efficient expansion relative to plain extendible hashing. As it does not store the full data keys, it cannot simply rehash them across a larger address space. A naive solution is to reread every key from storage and rehash it, though this would be expensive. DHT dealt with this problem by allocating a very large statically sized hash table, yet this approach requires a lot of memory from the getgo and restricts the maximum data size [14].

Sphinx addresses this challenge by *re-purposing* more hash bits from the Trie Store and Reserve Bits whenever possible. It also encodes the Reserve Bits field in a manner that allows opportunistically obtaining more hash bits during queries to save I/Os for the next few expansions. These techniques are inspired by recent expandable filter data structures [3, 12, 13, 40, 44].

Re-Purposing Bits. In order to remap each entry during expansion to one of two new block groups, one additional hash bit is needed. Sphinx handles this by lending the most significant bit (MSB) from the block ID to the group ID, the MSB from the slot ID to the block ID, and the MSB from the fingerprint to the Slot ID. Whenever possible, Sphinx infers the most significant fingerprint bit from the delta trie or from the Reserve bits, as described below. Only if the most significant fingerprint bit is not encoded, Sphinx resorts to reading an entry from storage to extract it.

Reserve Bits Encoding. While the Reserve Bits field in Sphinx is fixed-length, it consists of two variable-length sub-fields. The first is a 1-delimited unary encoded age counter, which counts how many expansions ago a given entry was inserted. The remaining bits are the most significant fingerprint bits of the hash. For example, in the following 3-bit encoding *1*10, the leftmost one indicates the entry was just inserted and the remaining two bits 10 are the most significant fingerprint bits. After the first expansion, the encoding becomes *01*0 as the unary counter is incremented and the most significant fingerprint bit becomes a part of the slot address. After one additional expansion, the encoding becomes *001*. At this point, we have no more fingerprint bits left. The advantage of this variable-length encoding is that while fingerprints are repurposed during expansion, newer entries after the expansion can be set the maximum number Reserve Bits to save I/Os in future expansions.

Expansion Algorithm. The expansion algorithm traverses the slots in a block group one by one and deals with each slot according to three cases. (Case 1) If a slot is empty, no action is required. (Case 2) For a single-entry slot, one bit is repurposed from the Reserve Bits if it exists. If there are no more fingerprint bits available for this entry, though, the original key is retrieved from the log.

For a slot containing multiple entries, the algorithm distinguishes two cases: (Case 3-1) If the first differentiating bit is at Index 0, the delta trie is effectively partitioned into two groups, one with fingerprints beginning with `0` and the other with 1. This inherent grouping directly determines the new slot IDs for all entries within the slot. (Case 3-2) If the first differentiating bit occurs at an index greater than 0, all fingerprints share the same initial bit (i.e., identical MSB). In this scenario, an entry is randomly selected, and its MSB is evaluated through its Reserve Bits, if available, or by a single log access to resolve the first bit of the fingerprint.

Rejuvenation Operations. Sphinx may fetch an older entry from storage, whether during a regular query, a read-before-write in an insertion, or the expansion process itself. We treat this as an opportunity to rehash the key while extracting it from the log, replenishing its Reserve Bits. This reduces the cost of future expansions by avoiding additional storage accesses. We find that under standard workloads, rejuvenation operations will replenish most fingerprint bits, leading to largely I/O-free expansions.

3.6 Concurrency

Sphinx's directory layout naturally enables parallelism because operations within one group never interfere with others. We partition groups among worker threads according to their group IDs, ensuring entries are evenly distributed. A fast, concurrent, lock-free queue [17] hands off tasks from the main thread to each worker, so every thread can process its own queue independently, and for operations confined within a group no locks are needed.

After a block group expands or contracts, we briefly acquire a lock to update the directory's entries or, if necessary, resize it. Other threads continue using the original directory throughout this

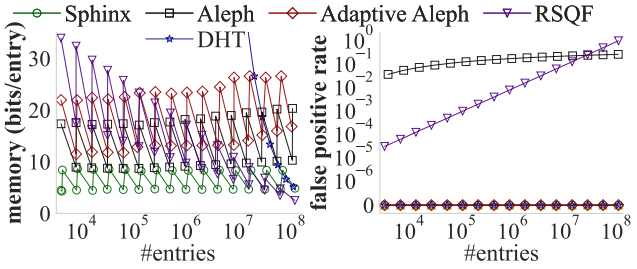

Figure 6: Sphinx eliminates false positives, uses at most half the memory of other baselines, and maintains a stable memory footprint while scaling.

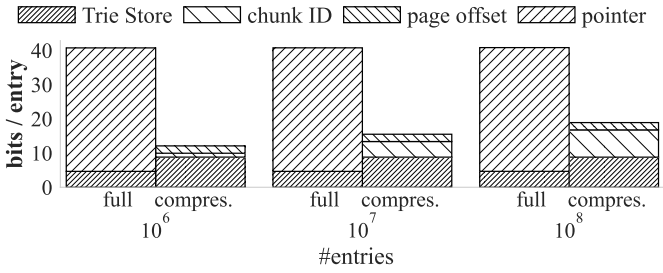

Figure 7: Compressing pointers reduces the pointer overhead and signifies the total memory efficiency of Sphinx.

process, since updates to the entries occur in separate slots and the switch to the resized directory happens via a single atomic pointer swap. This design keeps the critical section brief.

4 EVALUATION

We evaluate Sphinx against several baselines representative of state-of-the-art indexes for LSH-tables. We show the benefits of Sphinx across different platforms and use cases by conducting experiments on an NVMe SSD, on Optane persistent memory, and in-memory.

Platform 1. We run the purely in-memory and SSD benchmarks on a Precision 5860 Tower running Ubuntu 22.04. It features an Intel Xeon W7-2495X (24 cores/48 threads, 2.5 GHz) with CPU caches of 1.9 MB L1, 48 MB L2, and 45 MB L3, along with 64 GB of RAM. Storage includes a 512 GB SK Hynix PC801 NVMe SSD (OS drive) and a 2 TB Samsung 980 PRO SSD (benchmark drive).

Platform 2. We run Optane benchmarks on a SYS-620U-TNR system running Ubuntu 22.04, which has a 960 GB Samsung MZ1L2960 SSD as its OS drive. It is powered by dual Intel Xeon Silver 4314 CPUs (32 cores/64 threads, 2.4 GHz) with CPU caches of 2.5 MB L1, 40 MB L2, and 48 MB L3. There are 256 GB of RAM and 496 GB of Intel Optane Persistent Memory 200 Series module.

Baselines. We compare Sphinx against six baselines representative of the state-of-the-art LSH-table indexes. The Rank-and-Select Quotient Filter (RSQF) [42] represents mainstream fingerprint indexes that suffer from a rapidly increasing false positive rate (FPR) as the index expands. Aleph Filter [12] is built on top of RSQF and supports variable-length fingerprints. This allows assigning longer fingerprints to newer entries, giving it a more scalable FPR. We use the C++ implementation from [19] for RSQF and Aleph Filter, which supports storing a payload alongside each fingerprint. In the Adaptive Aleph Filter baseline, we combine Aleph Filter with a dictionary of full keys to resolve fingerprint collisions and eliminate false positives when querying for existing keys, as described in Section 2. Next, we implemented DHT [14] in C++ since the original implementation is closed-source and designed for FPGAs. Because DHT is statically sized, we add DHT-expandable, a variant of DHT that expands similarly to Sphinx to allow an apples-to-apples comparison with other expandable baselines.

Across all experiments, RSQF is initialized with 15-bit fingerprints. It loses one fingerprint bit in each expansion and by the end of each experiment, its fingerprints run out of bits. Aleph Filter is assigned 5-bit fingerprints to keep it as competitive as possible with Sphinx's memory budget. The Adaptive Aleph Filter is assigned 7-bit fingerprints to limit fingerprint collisions, preventing the dictionary from blowing up the memory footprint. Several experiments benchmark some Sphinx variants with Reserve Bits. All indexes expand once their expansion granularity reaches ≈ 95% load factor.

Experiment 1: Memory Usage vs. FPR. Figure 6 measures memory footprint and false positive rate as the data grows. We employ uniformly distributed insertions, though other workloads lead to similar results. The query workload targets uniformly random existing entries. The fluctuations in the number of bits per entry for most baselines are due to being nearly full before expanding and dropping to half full after.

Sphinx dominates the other baselines in this experiment. Its memory footprint is the lowest, and it exhibits a zero FPR. While DHT also exhibits a zero FPR, it has the highest initial memory footprint because it is statically sized. RSQF's memory footprint decreases throughout the experiment due to its shrinking fingerprints, yet this leads to its FPR skyrocketing. Aleph Filter maintains a stable memory footprint and scalable FPR, though it is dominated by Sphinx. The Adaptive Aleph Filter also has a zero FPR through a significantly higher memory footprint than Sphinx.

Experiment 2: Total Memory Usage. Figure 7 details the total memory overhead of Sphinx before and after applying the pointer compression scheme introduced in [14] as the dataset grows. In full mode, Sphinx uses 32-bit absolute pointers and requires ≈ 40 bits/entry, with pointers consuming over 35 bits.

In compressed mode, however, Sphinx compacts these pointers and configures each chunk to approximately 8 MB (tunable in our implementation), reducing the total memory overhead to about 20 bits/key. Of this, roughly 9 bits come from the Trie Stores, split evenly between the global and local index. The remaining overhead stems from the Payload Stores: up to 9 bits for chunk IDs in the global index and ≈ 2 bits for delta-encoded page offsets in the local index. Applying the compression scheme halves total memory use and restores metadata as a significant part of the index footprint, rather than letting pointers dominate. Buffer is excluded from this figure as its memory becomes negligible as the dataset grows.

Experiment 3: In-Memory Performance Evaluation. Figure 8 reports average and 99th-percentile latency for queries, updates, and insertions, with insertion cost including expansion. Our goal is to capture the CPU overhead for these operations by isolating index manipulation from I/O latency, so both the index and the dataset reside in memory.

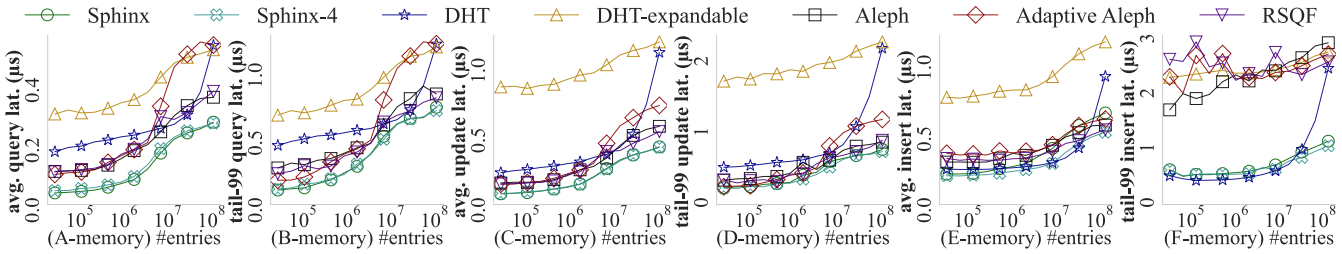

Figure 8: In memory, Sphinx variants achieve the lowest tail and average query/update latency, and stay competitive in insertion.

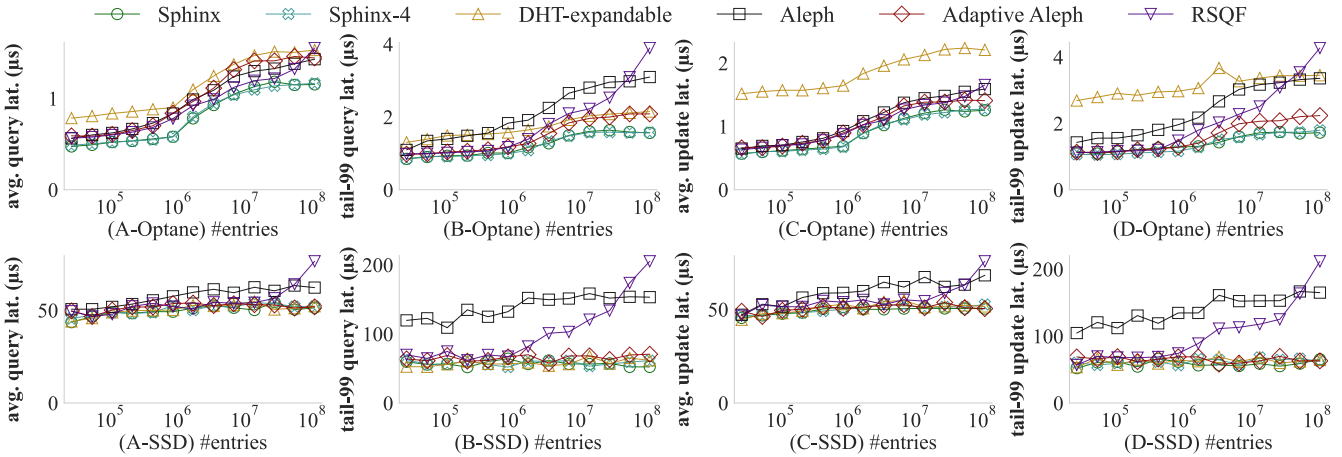

Figure 9: On Optane, Sphinx variants are fastest; on SSD, they match or beat all baselines by avoiding false positives.

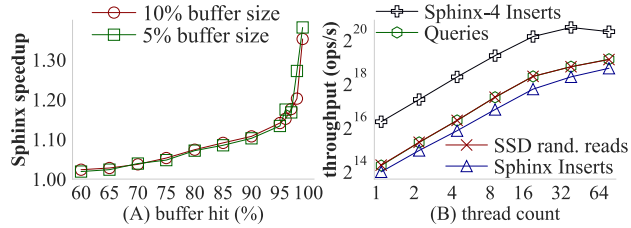

Figure 10: (A) Sphinx leads to substantial benefits with SSD under workload skew. (B) Sphinx scales well with threads.

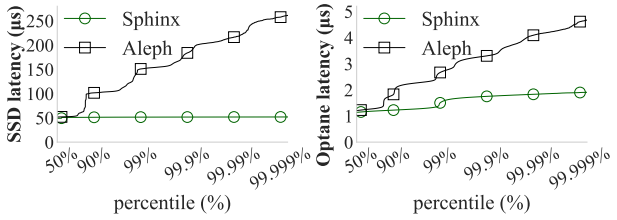

Figure 11: Sphinx maintains stable tail latency even in extreme percentiles.

Across all metrics, latency increases as the index outgrows CPU caches. DHT and DHT-expandable show the highest latency due to their bit-by-bit block traversal. Initially, DHT is faster than DHT-expandable, as it stays mostly empty until the latter half of the insertions. Once filled, their performance converges as expected, given their similar inefficient parsing strategies. RSQF and Aleph Filter show rising average and tail latency due to false positives triggering extra log accesses. While Adaptive Aleph Filter avoids false positives, it still incurs overhead from accessing both dictionary and filter. Furthermore, the use of linear probing during insertions in RSQF, Aleph Filter, and Adaptive Aleph Filter can lead to multiple entry shifts, significantly inflating their tail latencies.

In contrast, Sphinx variants achieve the lowest average and tail latencies for both queries and updates, thanks to their false-positive-free design, CPU-efficient parsing, and cache alignment. Sphinx also remains competitive in average insertion latency and delivers the best tail insertion latency due to its hardware-optimized traversal and tightly confined Trie Store. This holds even though it incurs an additional read-before-write, an overhead avoided by fingerprint-based methods. Sphinx-4 sidesteps most of these read-before-writes and outperforms all baselines across the board.

Experiment 4: Performance Evaluation over Optane/SSD. We now move the dataset to Optane and SSD and evaluate query and update latency. Later, we evaluate insertion latency in Experiment 8. DHT's static size makes it qualitatively different from the other baselines, so we replace it with DHT-expandable.

In the Optane setting, shown in the first row of Figure 9, all baselines exhibit similar trends to the in-memory benchmarks. Sphinx variants continue to outperform the others, although average latency increases due to the slower storage medium.

On SSD, shown in the second row of Figure 9, RSQF, and Aleph Filter are further penalized because each false positive triggers an additional expensive I/O operation. In this experiment, Adaptive Aleph Filter, DHT-expandable, and Sphinx variants perform best,

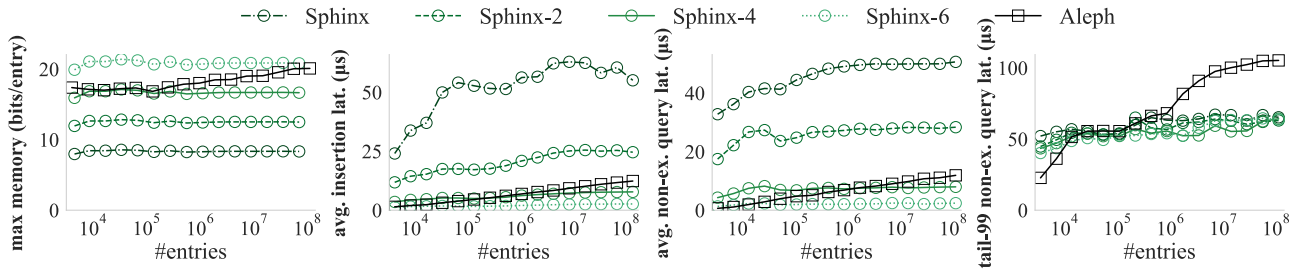

Figure 12: Sphinx-4 consistently outperforms Aleph across all metrics as data scales, benefiting from Reserve Bits' efficiency gains, while Sphinx variants remain stable and Aleph degrades.

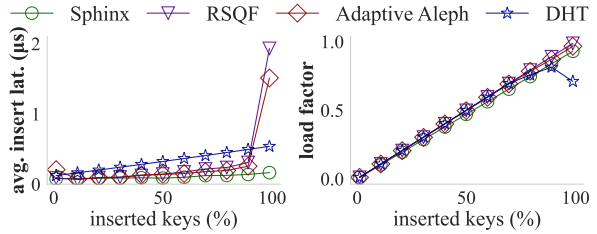

Figure 13: Sphinx exceeds 95% load factor while keeping the most stable and lowest insertion latency.

with only a slight difference between Sphinx and DHT-expandable under a uniform workload. As we show next, however, Sphinx still leads to substantial benefits with SSD under workload skew, which is the more common case for real-world workloads.

We also note that Experiments 3 and 4 using YCSB's 50%/50% update-query mix under Zipfian skew [11] yielded similar results.

Experiment 5: Under Workload Skew. We evaluate Sphinx against DHT-expandable on an SSD with a block cache in memory, varying the fraction of queries hitting the cache. Each query still traverses the index to locate the target page before accessing it from the cache. We report the speedup as DHT-expandable's latency divided by Sphinx's.

Figure 10 Part (A) shows the results for buffer pool sizes equal to 5% and 10% of the data size. As the hit rate (proportion of queries targeted cached entries) increases from 60% to 99% on the x-axis, more of the overall system overhead shorts from I/Os to CPU. From a hit rate of 90% and onwards, which is common for real-world skewed workloads, Sphinx provides a speedup improvement between 10% and 35%. This demonstrates that even in systems where I/O cost is very expensive, the CPU optimizations for Sphinx are still worthwhile, given the skew in real-world workloads.

Experiment 6: Tail Latency. In this experiment, we measure tail latency more comprehensively on both Optane and SSD settings, comparing Sphinx to Aleph Filter, the state-of-the-art fingerprint index. As shown in Figure 11, Aleph Filter exhibits mounting tail latency as several false positives can take place for each query in rare cases. The tail latency of Aleph Filter worsens in both settings for more extreme tail percentiles. In contrast, Sphinx avoids false positives and thus exhibits predictable and almost flat query latency across the board.

Experiment 7: Concurrency. Figure 10 Part (B) shows Sphinx-4 and Sphinx under increasing numbers of worker threads. Each point is the median of three runs inserting and querying 4 million uniformly random 16-byte key-value pairs. Sphinx-4 achieves higher single-threaded insertion throughput due to its Reserve Bits, which reduce read-before-writes and I/O during expansions. Both systems scale similarly with thread count. Query throughput grows with more threads until saturating SSD random read bandwidth. Insertions scale too but remain 10–30% slower than queries for Sphinx, as they require read-before-writes and occasionally trigger expansions and buffer flushes. Since all queries target existing keys, Sphinx-4 and Sphinx perform identically in query throughput.

Experiment 8: Using Reserve Bits. Figure 12 evaluates Sphinx variants on an SSD under uniformly distributed insertions and queries as the data size grows. We test four versions, each using a different number of Reserve Bits (0, 2, 4, and 6), and compare them to Aleph Filter, the most memory-efficient baseline. For each variant, we measure maximum memory per entry, insertion latency (including expansion), and zero-result query latency—reporting both average and tail latency for non-existing keys.

Reserve Bits significantly improve Sphinx's insert latency by reducing read-before-writes and I/O during expansions. Similarly, they improve performance for queries targeting non-existing keys by, again, ruling out the existence of the key and avoiding the need to search for it in the log. We observe that Aleph Filter's performance across the different metrics gradually deteriorates with increasing data size and FPR, and that Sphinx-4 outperforms Aleph Filter across all metrics, including insertion.

Experiment 9: Load Factor. Figure 13 shows the load factor during insertions until the index reaches capacity. Load factor is defined as the ratio of stored data to total physical space. We measure this over an index with size of a block group, the expansion unit in Sphinx, for fair comparison. RSQF and Adaptive Aleph Filter reach over 95% load factor but suffer sharp spikes in insertion latency beyond that point. DHT maintains flatter latency but starts higher and worsens as it expands faster than it fills, ending below 70% load factor. In contrast, Sphinx exceeds 95% load factor while maintaining the lowest and most stable insertion latency.

5 CONCLUSION

We introduced Sphinx, a succinct, perfect hash table optimized for high performance on commodity CPUs. Future work could explore designing an ARM-compatible version of Sphinx and further investigate the trade-offs introduced by Reserve Bits.

REFERENCES

[1] AMD, Inc. 2024. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions.* https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 1–14.

[3] Jim Apple. 2022. Stretching your data with taffy filters. *Software: Practice and Experience* 52 (08 2022), 2349–2367. https://doi.org/10.1002/spe.3129

[4] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. arXiv:1711.01616 [cs.DS] https://arxiv.org/abs/1711.01616

[5] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (July 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[6] Fabiano C. Botelho, Anísio Lacerda, Guilherme Vale Menezes, and Nivio Ziviani. 2011. Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences* 181, 13 (2011), 2608–2625. https://doi.org/10.1016/j.ins.2009.12.003 Including Special Section on Databases and Software Engineering.

[7] Alex D. Breslow and Nuwan S. Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.* 11, 9 (May 2018), 1041–1055. https://doi.org/10.14778/3213880.3213884

[8] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2464–2476. https://doi.org/10.1145/3448016.3457565

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, Jordan Hunter, and Michael F. Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 275–290.

[10] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[12] Niv Dayan, Ioana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *arXiv preprint arXiv:2404.04703* (2024).

[13] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. Infinifilter: Expanding filters to infinity and beyond. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[14] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, Avraham (Poza) Meir, Mark Mokryn, Iddo Naiss, and Noam Rabinovich. 2021. The end of Moore's law and the rise of the data processor. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2932–2944. https://doi.org/10.14778/3476311.3476373

[15] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2, 1414–1425.

[16] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 25–36.

[17] Cameron Desrochers. [n.d.]. ConcurrentQueue: An industrial-strength lock-free queue for C++. https://github.com/cameron314/concurrentqueue. GitHub repository. Accessed: 2025-02-19.

[18] Peter C Dillinger and Panagiotis Manolios. 2009. Fast, all-purpose state storage. In *Model Checking Software: 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings 16*. Springer, 12–31.

[19] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6, Article 244 (Dec. 2024), 27 pages. https://doi.org/10.1145/3698820

[20] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.* 4, 3 (Sept. 1979), 315–344. https://doi.org/10.1145/320083.320092

[21] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[22] Kevin P Gaffney and Jignesh M Patel. 2024. Is Perfect Hashing Practical for OLAP Systems?. In *CIDR*.

[23] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. CTI Press and Ellinika Grammata Greece, 27–38.

[24] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *SIGPLAN Not.* 44, 3 (March 2009), 229–240. https://doi.org/10.1145/1508284.1508271

[25] Longzhe Han, Yeonseung Ryu, and Keunsoo Yim. 2006. CATA: a garbage collection scheme for flash memory file systems. In *Proceedings of the Third International Conference on Ubiquitous Intelligence and Computing* (Wuhan, China) *(UIC'06)*. Springer-Verlag, Berlin, Heidelberg, 103–112. https://doi.org/10.1007/11833529_11

[26] Zhuo Huang, Jih-Kwon Peir, and Shigang Chen. 2011. Approximately-perfect hashing: Improving network throughput through efficient off-chip routing table lookup. *Proceedings - IEEE INFOCOM*, 311 – 315. https://doi.org/10.1109/INFCOM.2011.5935158

[27] Intel Corporation. 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* https://cdrdv2.intel.com/v1/dl/getContent/671200

[28] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. 2022. IPLFS: Log-Structured File System without Garbage Collection. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 739–754. https://www.usenix.org/conference/atc22/presentation/kim-juwon

[29] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-optimal filtering: Bloom overtakes Cuckoo at high throughput. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 502–515. https://doi.org/10.14778/3303753.3303757

[30] David J. Lee, Samuel McCauley, Shikha Singh, and Max Stein. 2021. Telescoping Filter: A Practical Adaptive Filter. In *29th Annual European Symposium on Algorithms (ESA 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Petra Mutzel, Rasmus Pagh, and Grzegorz Herman (Eds.), Vol. 204. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 60:1–60:18. https://doi.org/10.4230/LIPIcs.ESA.2021.60

[31] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[32] Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A Performant Extendible Adaptive Hashing Index. *Proc. ACM Manag. Data* 1, 1, Article 108 (May 2023), 25 pages. https://doi.org/10.1145/3588962

[33] Zirui Liu, Xian Niu, Wei Zhou, Yisen Hong, Zhouran Shi, Tong Yang, Yuchao Zhang, Yuhan Wu, Yikai Zhao, Zhuochen Fan, and Bin Cui. 2025. CuckooDuo: Extensible Dynamic Perfect Hashing for RDMA-based Remote Memory KV Store. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE)*. IEEE. https://pkuzhao.net/publication/cuckooduo/CuckooDuo.pdf

[34] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[35] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2006. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*. IEEE, 2774–2778.

[36] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2020. Adaptive Cuckoo Filters. *ACM J. Exp. Algorithmics* 25, Article 1.1 (March 2020), 20 pages. https://doi.org/10.1145/3339504

[37] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press.

[38] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and A. El Abbadi. 2015. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:16457773

[39] John Ousterhout and Fred Douglis. 1989. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.* 23, 1 (Jan. 1989), 11–28. https://doi.org/10.1145/65762.65765

[40] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate A Set Without Knowing Its Size In Advance. https://doi.org/10.48550/arXiv.1304.1188

[41] Prashant Pandey, Michael Bender, and Rob Johnson. 2017. A Fast x86 Implementation of Select. (06 2017). https://doi.org/10.48550/arXiv.1706.00990

[42] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3035918.3035963

[43] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1386–1399. https://doi.org/10.1145/3448016.3452841

[44] Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 636–644. https://doi.org/10.1145/3626246.3654681

[45] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14, Article 4 (Jan. 2010), 18 pages. https://doi.org/10.1145/1498698.1594230

[46] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: a space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.* 10, 13 (Sept. 2017), 2037–2048. https://doi.org/10.14778/3151106.3151108

[47] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[48] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. 1993. An Implementation of a Log-Structured File System for UNIX.. In *USENIX Winter*. 307–326.

[49] Justin Sheehy and Dave Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. Basho White Paper.

[50] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a real-time operational dbms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1389–1400.

[51] Radu Stoica and Anastasia Ailamaki. 2013. Improving flash write performance by using update frequency. *Proc. VLDB Endow.* 6, 9 (July 2013), 733–744. https://doi.org/10.14778/2536360.2536372

[52] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: a scalable log-structured database system in the cloud. *Proc. VLDB Endow.* 5, 10 (June 2012), 1004–1015. https://doi.org/10.14778/2336664.2336673

[53] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: a write-optimized persistent log-structured hash-table. *Proc. VLDB Endow.* 15, 11 (July 2022), 2895–2907. https://doi.org/10.14778/3551793.3551839

[54] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 773–788. https://www.usenix.org/conference/atc22/presentation/wang-jing

[55] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 197–210. https://doi.org/10.14778/3364324.3364333

[56] Richard Wen, Hunter McCoy, David Tench, Guido Tagliavini, Michael A. Bender, Alex Conway, Martin Farach-Colton, Rob Johnson, and Prashant Pandey. 2024. Adaptive Quotient Filters. *Proc. ACM Manag. Data* 2, 4, Article 192 (Sept. 2024), 28 pages. https://doi.org/10.1145/3677128

[57] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[58] Yitian Yang and Youyou Lu. 2023. NICFS: a file system based on persistent memory and SmartNIC. *Frontiers of Information Technology & Electronic Engineering* 24, 5 (2023), 675–687.

[59] Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. 2022. RACE: One-sided RDMA-conscious Extendible Hashing. *ACM Trans. Storage* 18, 2, Article 11 (April 2022), 29 pages. https://doi.org/10.1145/3511895