NEXT: A New Secondary Index Framework for LSM-based Data Storage

JIACHEN SHI, Nanyang Technological University, Singapore
JINGYI YANG, Nanyang Technological University, Singapore
GAO CONG, Nanyang Technological University, Singapore
XIAOLI LI, Institute for Infocomm Research, A*STAR & Nanyang Technological University, Singapore

Key-value databases with Log Structured Merge tree are increasingly favored by modern applications. Apart from supporting fast lookup on primary key, efficient queries on non-key attributes are also highly demanded by many of these applications. To enhance query performance, many auxiliary structures like secondary indexing and filters have been developed. However, existing auxiliary structures suffer from three limitations. First, creating filter for every disk component has low lookup efficiency as all components need to be searched during query processing. Second, current secondary index design requires primary table access to fetch the data entries for each output primary key from the index. This indirect entries fetching process involves significant point lookup overhead in the primary table and hence hinders the query performance. Last, maintaining the consistency between the secondary index and the primary table is challenging due to the out-of-place update mechanism of the LSM-tree. To overcome the limitations in existing auxiliary structures for non-key attributes queries, this paper proposes a novel secondary index framework, NEXT, for LSM-based key-value storage system. NEXT utilizes a two-level structure which is integrated with the primary table. In particular, NEXT proposes to create secondary index blocks on each LSM disk component to map the secondary attributes to their corresponding data blocks. In addition, NEXT introduces a global index component which is created on top of all secondary index blocks to direct the secondary index operation to the target secondary index blocks. Finally, NEXT adopts two optimization strategies to further improve the query performance. We implement NEXT on RocksDB and experimentally evaluate its performance against existing methods. Experiments on both static and mixed workloads demonstrate that NEXT outperforms existing methods for different types of non-key attributes.

CCS Concepts: • **Information systems → Data access methods**.

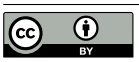
Additional Key Words and Phrases: LSM-based KV Storage, Secondary Indexing, Query Optimization

ACM Reference Format:
Jiachen Shi, Jingyi Yang, Gao Cong, and Xiaoli Li. 2025. NEXT: A New Secondary Index Framework for LSM-based Data Storage. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 193 (June 2025), 25 pages. https://doi.org/10.1145/3725330

1 Introduction

Many modern applications such as trajectory planning, real-time recommendations and auto decision-making, require the support of query processing on continuous data stream from various sources like smart devices, IoT sensors and social media. Data management systems today face the challenges of high volume data ingestion and effective real-time queries execution. Key-value

Authors' Contact Information: Jiachen Shi, jiachen001@e.ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Jingyi Yang, Nanyang Technological University, Singapore, Singapore, jyang028@e.ntu.edu.sg; Gao Cong, Nanyang Technological University, Singapore, Singapore, gaocong@ntu.edu.sg; Xiaoli Li, Institute for Infocomm Research, A*STAR & Nanyang Technological University, Singapore, Singapore, xlli@i2r.a-star.edu.sg.



ACM 2836-6573/2025/6-ART193
https://doi.org/10.1145/3725330

database, characterized by its straightforward design and rapid data access, is developed as a more scalable and high-speed alternative data storage solution to relational databases. To support write-intensive workloads, many key-value databases today adopt the Log-Structured Merge tree (LSM-tree) [31] as the storage structure. Different from traditional storage structure that updates data in-place, LSM-tree buffers writes (i.e., inserts, updates and deletes) in memory and then flush them to storage devices in the form of immutable sort runs using sequential I/Os. This out-of-place design allows database systems to avoid expensive random I/Os and hence achieve fast ingestion. Furthermore, LSM-tree periodically sort-merges runs in persistent storage and utilizing auxiliary in-memory structures to facilitate effective query performance. Because of these advantages, LSM-trees are not only adopted by many industrial key-value databases such as Apache AsterixDB [1], Bigtable [6], LevelDB [20] and Amazon DynamoDB [14], but also used as underlying storage engines for relational databases (e.g., RocksDB [18] in MyRocks [28] and Pebble in CockroachDB [24]).

Plenty of academic studies have been proposed to optimize different aspects of the LSM-tree to better support high write throughout and efficient primary key lookup, examples like Monkey [11], Dostoevsky [12], bLSM [38], Lethe [36] and LSbM-tree [42]. Beside primary key search, to support flexible workloads and real-time analytics, many applications today also demand high performance queries on non-key attributes. Taking Facebook's social graph database service as an example, queries such as finding user IDs who liked a specific post involves extensive search on secondary key [4]. Another example is Twitter, which allows user to search posts with custom filters on the geographical location or the user group. Given the short response time application users would expect in these scenarios, it is important to design an efficient secondary index for LSM-KVs.. Existing auxiliary structures for non-key attribute lookup can be classified into two classes [33]. One is the standalone secondary index [2, 22, 27, 41] which maintains a separated data structure (e.g., another LSM-tree) to map the non-key attribute to the corresponding primary keys. Another class is per-segment structure like filter or zone map [21, 23, 33] which is created for each in-disk LSM component (i.e., SSTable) to skip redundant I/Os. Figure 1 demonstrates the overview of these two classes.

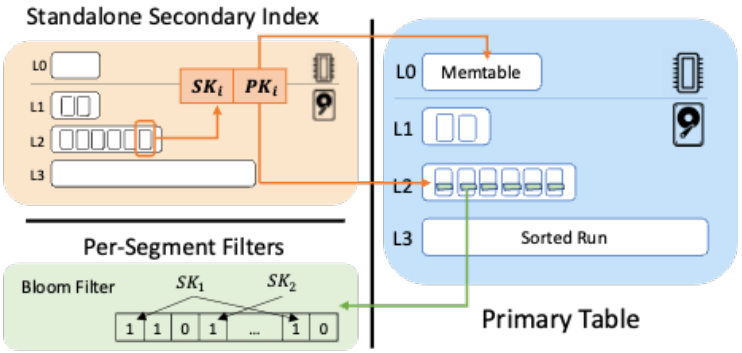


Fig. 1. Overview of Existing Secondary Indexing

Standalone Secondary Index. Similar to the secondary index design in traditional page-oriented systems, majority of existing LSM-based systems maintain a separate secondary index structure outside the primary table which maps the target secondary attributes to the associated data record. However, unlike page oriented systems where secondary indexes directly point to the actual physical location (i.e., page id) of the record, the stand-alone secondary index in a LSM-based system only stores the primary keys associated with the secondary attribute. This is because in an

LSM with out-of-place updates, the actual physical locations of the records are constantly changing with incoming flushing or compaction operations.

As a result, during query processing of LSM-based systems, index navigation[27] is needed to obtain the actual record given the primary keys output from the secondary index. Specifically, the system needs to first traverse the secondary index to find a list of matching primary keys, and then issue multiple point lookup to the primary table to fetch value for each target primary key. Figure 2a shows an example of the index navigation. The index navigation process incurs significant read amplifications during query processing [25, 27, 44] because point lookup operation is far less efficient compared to reading a given page as it requires iterating over the different levels of the LSM-tree, checking the Bloom filters, reading all versions of the record with the given key, and reconcile across the different versions. This process is exacerbated by long range query, which produces a long list of candidate primary keys, each requiring a separate point lookup operation over the primary table. Apart from the overhead of performing a large number of point lookup, the index navigation process may conduct redundant reads on the same block for different primary keys, further increasing read amplification. In addition, as modifications upon entry in primary table follow out-of-place method (i.e., based on primary key only, without reading the obsolete value), the maintenance overhead to keep the secondary index consistent with primary table is also significant [2, 27].

Per-segment Filters. Per-segment filters are also adopted to support secondary attributes lookup. Through constructing Bloom filter and zone map for each SST file, secondary query process can skip file I/O that does not contains the matching attribute and locate the data blocks with overlapping ranges effectively. One advantage of per-segment filter over standalone secondary index is its lower maintenance cost in terms of time and space. As SST files are immutable and the per-segment filters are naturally computed during SST files creation, no additional updates process nor synchronization of a separate data structure is needed. However, per-segment filter has non-negligible overheads of CPU and memory during query processing [10, 13]. This is because per-segment filters only store the membership and value ranges of the entries in a SST file, all SST files' filters need to be searched during query processing. Furthermore, as the entries in LSM-tree are sorted based on primary key, when the secondary attributes are not correlated to the primary key, the pruning effectiveness of zone map which stores the value range of the SST file will be negligible [33].

To address the limitations of existing auxiliary index structures for non-key attributes in LSM-based storage systems, this paper introduces a novel secondary index framework called NEXT, designed for LSM-based systems. Unlike traditional auxiliary structures, NEXT employs a two-level index structure that is tightly integrated with the primary table to enhance query efficiency. There are four key features in NEXT: (1) **Direct Data Block Indexing.** Instead of storing primary keys in the secondary index, NEXT proposes to index the data block location (i.e., the offset within each SST file) for each secondary attribute value. This design allows query operations to directly locate matching entries in the primary table, eliminating the need for costly index navigation seen in traditional standalone secondary indexes. (2) **Per-segment Secondary Index Blocks.** Inspired by the per-segment structure's lower maintenance overhead, NEXT proposes to create *secondary index blocks* within each SST file. These blocks store secondary indexes for each file independently, functioning similarly to the leaf nodes of conventional secondary indexes (e.g., B+-trees or R-trees). Within each secondary index block, secondary attribute values are sorted and mapped to corresponding data blocks. This design, which naturally aligns with SST file creation, simplifies the consistency maintenance. (3) **Global Index Component.** To address the read amplification issue seen in per-segment filters, NEXT introduces a *global index component* stored in RAM. This component directs query operations to the appropriate secondary index blocks, acting like internal nodes in traditional index structures which narrow down the search space. By traversing this global

index, redundant reads on unmatched secondary index blocks are avoided. (4) **Optimization in Query Performance.** To further improve query efficiency, this paper introduces two optimization strategies for index construction and query processing. These strategies focus on minimizing the false positive rate of the global index component and read amplifications during query execution.

Through implementing NEXT on RocksDB and evaluating it against other secondary indexing methods, our experimental results demonstrate that NEXT outperforms existing secondary indexing techniques for both static and dynamic workload. Our major contributions can be summarized as:

- We propose a new secondary index framework, NEXT, for LSM-based data storage, which overcomes the costly index navigation limitations of standalone secondary index and addresses the read-amplifications issue of per-segment filters.
- We introduce two optimization strategies on index constructions and query processing which enhance the performance of NEXT.
- We implement the proposed index framework into RocksDB and extensive experiments are conducted to evaluate the performance of NEXT on both static and dynamic workload. Experimental results demonstrate that NEXT outperforms existing secondary index techniques in LSM-based storage.

2 Background and Related Work

In this section, we start by describing the general idea of the LSM-tree. We then provide a concise review of related work on secondary index techniques in LSM-based storage.

2.1 Log-Structured Merge Trees

The Log-Structured Merge tree (LSM-tree) is an ordered, disk-based storage architecture proposed by [31] to sustain high data ingestion. By adopting sequential writes with out-of-place updates design, LSM-tree delivers superior write performance compared to in-place-update storage structure.

LSM-tree stores data in the form of key-value pairs, where an entry's unique identifier is referred as *key*, and other attributes associated with the entry are referred as *value*. In LSM-tree, entries are typically sorted and accessed based on their keys.

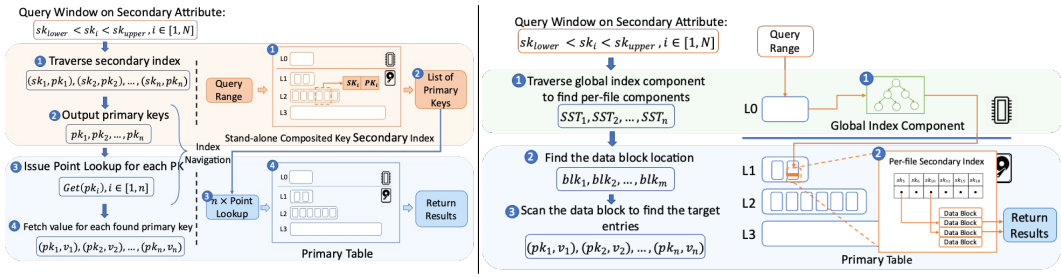
The LSM-tree has a multi-level structure on storage which contains an in-memory component and disk components. Typically, for an LSM-tree with L levels, the first level (*Level 0*[1]), also known as *MemTable*, resides in main memory while other levels are disk-resident [12, 16]. As is the case in [18], a *MemTable* is usually a Skip Lists or other index structures support operations that are semantically equivalent (e.g., B^+-tree). For disk components, each level contains one or more sorted runs and the capacity of level increases exponentially as the level increases (i.e., there is a constant size ratio between adjacent levels). Each sorted run is further partitioned into smaller components named *SSTables*[2] which store the sorted KV pairs as byte arrays. In this paper, both *SSTables* and *SST files* are used interchangeably to refer the storage unit of each sorted run. SSTables are immutable and usually have block-based format where data is chunked into fix-sized blocks.

When new KV pairs are inserted, they are firstly buffered in MemTable. Once the buffer reaches capacity, the MemTable is converted into immutable MemTable and flushed to disk storage as a sorted run. As SSTables are immutable, modifications (i.e., Updates and Delete) of existing entries are handled as new inserts. To delete an entry in LSM-tree, an "anti-matter" entry (or tombstone) is inserted as new entry. The "anti-matter" entry contains the deleted key and a byte-long flag in value field to mark the deletion. To query entries in LSM-tree, sorted runs and blocks where target entries

[1]Some documents may refer level 0 as the first level in disk component. In this paper, we refer level 0 as the memory component.

[2]SSTables can be referred as Sorting String Table in LevelDB or Static Sorted Table in RocksDB

may reside will first be located and read into memory. Then the target entries will be searched within the block. After entries are fetched from the components, entries with the same key will be *reconciled* to find the most recent valid value. To limit the number of sorted runs and optimize query performance, LSM-tree periodically undergoes *compaction* to merge disk components and to garbage collect logically invalidated entries.



(a) Standalone Secondary Index (b) NEXT secondary index framework

Fig. 2. LSM-based Secondary Index Query Processing

2.2 Secondary Indexes in LSM-based Data Storage

As discussed above, LSM-based key-value systems are designed to support high write throughput and efficient primary key lookup. However, many applications also require queries on attributes in the value field other than the primary key. As entries in LSM-tree are sorted based on primary key and scattered throughout the disk components, without the support of secondary index on value attributes, the database systems need to perform expensive full table scan to locate the target entries. LSM-based systems have adopted different indexing approaches to support query on value attributes. Qader et al. [33] conduct a comparative study on secondary indexing techniques in LSM-based systems and broadly split them into two classes: Per-segment filters and standalone secondary index.

2.2.1 ***Per-segment Filters.*** Per-segment filters are light-weight data structures associated with each LSM segment (e.g., SST file or sorted run) that efficiently checks for membership of a given attribute value, and help to prune LSM segments during query execution. Qader et al. [33] propose to utilize Bloom filters for secondary attribute lookup. They construct Bloom filter for each data block inside the SST file to store the membership of each secondary attribute value. In addition to Bloom filter, SlimDB [34] utilizes Cuckoo filter [19] to map key fingerprint to the most recent matching level. On top of Cuckoo filter, Kipf et al. [23] introduce an index structure named Cuckoo Index (CI) which extend the application of Cuckoo filter on secondary attribute indexing. However, a drawback of these filter-based techniques is that they cannot support range queries. To accelerate range queries on secondary attribute, Qader et al. [33] also propose to add zone maps in each SST file. Zone map is used to store the maximum and minimum secondary attribute values in each data block so that query operation can skip blocks that have non-overlapping ranges. However, the pruning effectiveness of zone map is trival when the correlations between secondary attribute and primary key is weak. Different from the zone map, the per-segment secondary index blocks in NEXT index each entry individually and organize them based on the secondary attribute values, and thus the performance is independent on the primary key correlation. Recently, embedded R-tree index [21] , which builds a R-tree index for each SSTable file, was proposed to enhance spatial queries performance on LSM-based systems. However, all embedded R-trees need to be searched

during query processing which incurs significant read amplifications. In contrast to the embedded spatial index, NEXT introduces a global index component which narrow down the search space and hence reduces the read amplifications.

2.2.2 ***Standalone Secondary Index.*** Apart from per-segment filter, majority of LSM-based systems adopt standalone secondary index, where a separated index structure is maintained outside the primary table storage. For instance, MongoDB [22] adopts conventional B^+-tree as secondary index which follows the secondary index approach of traditional relational database systems. Besides B^+-tree, some LSM-based systems store the secondary index as another LSM table (e.g., different column family[3]). LSM-based secondary indexes have two common index types including *posting list* and *composite index*. The posting list stores the mapping from each attribute value to a list of associated primary keys. It is adopted by systems like BigTable [6] and Cassandra [3]. For composite index, the attribute value and the primary key is concatenated together to form the index key and the index value is set to null. The search on entries' attribute value is turned into prefix range search on the composited index key. Composite index is easy to implement and widely adopted by many applications [1, 9, 28]. However, as the standalone secondary index only stores the mapping to primary keys, the query execution needs to go through a transition named as *Index Navigation* [27]. To fetch the entries from the primary storage, massive amount of point lookup to the primary table is needed. This will introduce significant read overhead [25, 27, 44]. Our evaluation using a uniformly distributed micro-benchmark demonstrates that directly accessing the primary table, bypassing index navigation, can reduce workload latency by over 50% for range queries with an average cardinality of 65. To mitigate the impact of index navigation, Luo et al. [27] introduce methods such as batched point lookup and blocked Bloom filter [32]. Nevertheless, as long as the index navigation is needed, the read amplifications issue will remain. Futhermore, Li et al. [25] propose a decoupled secondary index which points to record value independely. However, their method is based on key-value separation storage [26]. Different from existing standalone secondary index design, NEXT proposes to index the data block location for each attribute value and to create the secondary index blocks for each SST file. Through integrating the secondary index with the primary table storage, the costly index navigation can be eliminated.

3 NEXT Design

In this section, we first present the overview of NEXT. Then we discuss the details of NEXT's structure, and describe how NEXT support basic operations of the secondary index in LSM-based data storage. Lastly, we evaluate the complexity of NEXT as compared to existing secondary index designs.

3.1 Overview

Figure 2b shows the overview of the index structure in NEXT. NEXT proposes a novel two-level secondary index framework. For the bottom level, NEXT creates secondary index blocks for each SST file to store the data block location for each secondary attribute value, leveraging the immutable nature of the data block location in the SST file. The secondary index blocks are stored inside each SST file and naturally maintained with the file to utilize the advantage of the per-segment filters in index maintenance. On top of all secondary index blocks, NEXT introduces a global index component which resides in RAM. The global index has a multi-level tree structure. Each tree node contains entries that map a secondary value range to lower level tree nodes or secondary index blocks on disk. During query processing, the global index component first directs the search

[3]Column Families provide a way to logically partition the database, different column families share the same write-ahead log but have their own memtables and table files.

operation to the matching secondary index blocks to reduce read amplifications. Matching data block is then located from the seondary index block entries and scanned to fetch the results.

3.2 Structure

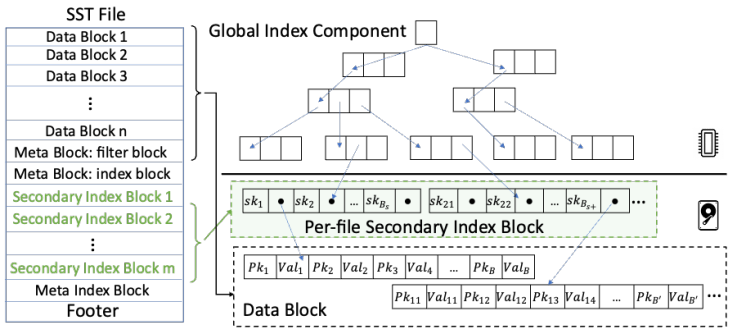


Fig. 3. Structure of NEXT

Per-segment Component. Persistent data in LSM-based system are stored in a collection of SST files. Due to the continued merging and compactions applied upon all SST files, the physical locations of the data entries keep changing and this makes a separated secondary index structure to directly access the data entry via physical location challenging. However, based on the immutable nature of the SST file, once it is created, the relative location of the data entry in the file is fixed. This inspire us to access the data entry using the relative location instead of the physical location. For each SST file, it usually adopts block-based format and the data is chucked into fix-sized blocks. Figure 3 shows an example of the SST file in LSM-based system on the left. As shown, in addition to data blocks, a SST file also contains meta blocks such as filter block for filter information and index blocks for primary index storage. During query processing, block that may contain the target entries will be read into memory and full block scan will be conducted to find the entries. To support fast entry retrieval via secondary attribute, different from existing per-segment filters that only store the membership of the secondary value, NEXT proposes to store the block location (i.e., relative location in SST file) for each secondary value. In each SST file, additional meta blocks are created as secondary index blocks. These blocks contains one index entry for each KV pair in the file. The index key is the indexed attribute value while the index value is the block location (i.e., file offset). Similar to the the traditional secondary index for page-oriented systems, NEXT sorts the index entries based on the index key and partitions them into multiple secondary index blocks. The secondary index blocks, therefore form the first level of the index structure of NEXT. As shown in Figure 3, each secondary index block (highlighted in green) contains sorted index entries that consist of the attribute values and the associated pointer to the data block.

Global Index. As the per-segment component only stores relative position within each SST file, when performing a secondary attribute query on an LSM-based storage, each SST files must be searched using the same predicate. Although the secondary index blocks can direct the query to the matching data blocks effectively at block level, the reading overhead of searching all SST files' secondary index blocks is non-trivial. To narrow down the searching space and improve the query performance, NEXT introduces a global index structure which functions like the internal nodes in traditional secondary index. Through constructing a multi-level index structure on top of the per-file secondary index blocks, NEXT's global index can efficiently locate the matching SST files and secondary index blocks. The global index of NEXT stores an index entry for each secondary

index block in the SST file. The index key is the range of attributes values in the index block and the index value is the secondary index block location (i.e., the SST file location and the relative location of the secondary index block in the file). For an LSM-based storage with N data entries, the number of index entries in the global index will be N/B_s where B_s is the number of index entries per secondary index block. As the number of entries in global index is B_s times smaller than the total number of the entries, the size of the global index will be light and hence it is designed to be memory based. For each global index in NEXT, an additional $O(N/B_s)$ memory overhead will be added to the system. However, this memory overhead is manageable. For example, the default block size in RocksDB is $4KB$ and the size of each numerical secondary index entry in NEXT is $32B$ (i.e., 16 Bytes for numerical ranges and 16 Bytes for the block pointer (aka. BlockHandle in RocksDB)). Hence, the size of the secondary index is less than 1% of the size of the data. To support different value types, the global index can be implemented with different data structures. For numerical attribute, as the secondary index blocks from different SST files will have overlapping ranges, traditional secondary index's (e.g., B^+-tree) structure is not applicable. To index entries with overlapping ranges, the global index of NEXT can be implemented with main-memory index structure for intervals (e.g., [5, 7]). For multi-dimensional value such as spatial data, the global index can be implemented using index structure like R-tree [39] which is good at indexing multi-dimensional information.

3.3 Basic Operations

In this section, we discuss how NEXT supports the basic operations of secondary index in LSM-based data storage.

Index Creation. Algorithm 1 describes the process of the index creation of NEXT. First, during SST file creation, we initialize a temporary auxiliary vector to store the secondary index information (line 1). When a data block is full and flushed to the SST file, we extract the secondary attribute value and the data block location for each KV-pair in the data block and add them to V as a secondary index entry (line 2-5). Second, similar to other meta blocks (e.g., primary index block, filter block and stats block), the secondary index blocks will be created after the flushing of data blocks. The vector V will be sorted based on the secondary attribute values and written into multiple blocks sequentially (line 6-9). For each secondary index block, we record the value range of the block (line 10). When the size of the secondary index block exceeds the capacity, the value range of the secondary index block together with the block location form a tuple and is inserted into the global index component as a global index entry (line 11-14). With insertions of the global index entries, the global index structure will self-maintain the internal nodes with node splits and merges. The creation process will be conducted for each SST creation and hence the global index component will contain entries across all SST files in the LSM storage. As Algorithm 1 involves sorting the attribute values in the new SST file, an additional computation cost of $O(N_{SST}log(N_{SST}))$ where N_{SST} is the number of entries in a SST file will be incurred during file creation. Although this cost is larger than the creation cost of the embedded Bloom filter ($O(N_{SST}k)$, where k is the number of hash functions), the additional cost is logarithm of the number of entries in a SST file which is much smaller than the total number of entrie in the database. Furthermore, as multiple threads will be used to conduct compactions in the background in parallel, this optimization strategy of LSM-tree will minimize the impact of the additional cost to the system.

Search. The search operation in NEXT starts with searching the secondary attribute in the *MemTable*. As NEXT only indexes the secondary attributes of data in disk, for data buffered in memory, a full *MemTable* scan is needed to find the target entries. After the *MemTable* scan, the query operation will start searching the global index component. From the root node of the global index, the query process will traverse down the index structure and locate the matching SST files and data block (i.e., leaf nodes). Then all the target data blocks in the primary LSM storage will be

Algorithm 1 NEXT Creation Algorithm

Input: SST file, S; the block size, δ
Output: Per-SST Secondary index blocks, $SB(s)$; Global index component, GL

```
1: V ← ∅ // V: auxiliary vector
2: while S.creation() not complete do
3:     if datablock is flushed then
4:         for KVpair ∈ datablock do
5:             V.add(< sec_val, datablock_loc >);
6: Sort V based on sec_val;
7: while V.size() > 0 do
8:     sec_entry ← V.pop_front();
9:     SB.add(sec_entry);
10:    SB.val_range.update(sec_entry);
11:    if SB.size() > δ then
12:        S ← SB; //flush secondary index block to SST
13:        GL ←< SB.val_range, SB.location >;
14:        SB ← ∅; //create new secondary index block
15: return SBs and GL;
```

read into memory and the block scan will be conducted to find the matching data entries. NEXT adopts the exiting validation strategy to maintain the consistency with the primary LSM storage. Similar to [40, 41, 44], an in-memory update memo is maintained to store the latest timestamp and number of obsolete entries for each updated/deleted primary key. After finding all matching entries from the secondary index, the results will be validated through checking the existence of the found primary keys in the update memo and comparing the latest timestamp with the timestamps of the entries which are generated when the data is inserted to the primary LSM storage. Apart from update-memo-based validation strategy, other "lazy" update validation strategies can be easily integrated with NEXT (e.g., the primary index LSM [27]). As the global index component of NEXT will direct query operation to data blocks with matching value ranges, the search operation for point look in NEXT follows the same procedures as discussed above. Figure 4 demonstrates the searching process with NEXT.

Update and Delete. One advantage of NEXT's per-segment component is that it reduces the overhead of the consistency management as the synchronization with the primary LSM storage is simplified. Specifically, the per-segment component is created naturally when a SST file is created and it is immutable, as a result, update and delete operations for NEXT's per-segment component are not required. For the global index component, as it indexes the secondary index blocks across SST files, update and delete operations are necessary to keep track of live SST files in the primary LSM storage. The primary LSM-based storage supports Multi-version concurrency control (MVCC). A version (aka snapshot) is used to refer to a list of valid SST files and blob files at a certain point in time [18, 20]. When a flush or compaction finishes, a new version will be generated to reflect the changes in the SST files state. During the creation of new version, the global index component will also be updated (i.e., inserting new index entries for new generated SST file and deleting old index entries for compacted files) so that it is consistent with the latest version information. When new data update or delete arrives, it will firstly be buffered in memory. During compaction or flushing, obsolete entries due to updates or deletes on primary keys will be cleaned and are not included in the new generated per-segment component. For obsolete entries that reside in existing SST file and

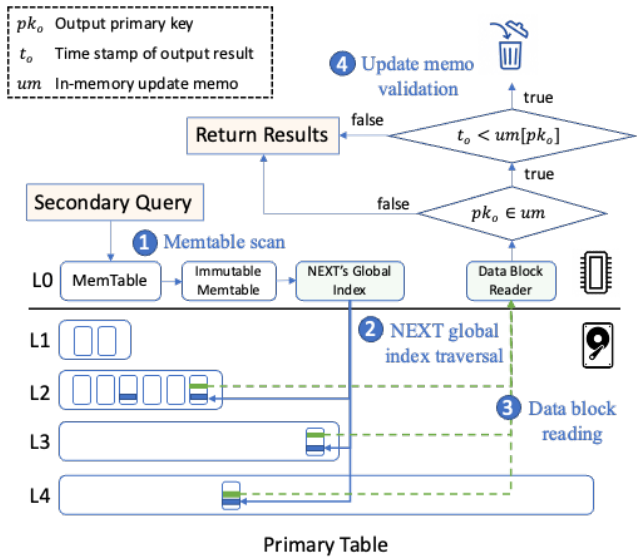


Fig. 4. NEXT Secondary Index Query Flow

are not cleaned during query process, the validation strategies can help NEXT filter out invalid results. Different from standalone secondary index, data deletes or updates only propagate to NEXT after compactions or flushing which create new SST files and per-segment components.

Crash Recovery. In the event of the LSM-based system failure or shutdown abnormally, the global index component of NEXT needs to be recovered to the latest valid version. During version creations, a transactional log (e.g., similar to the Manifest in [18]) is utilized to keep track of the state changes (i.e., updates in the global index). On system (re)start, the latest saved global index is loaded back to memory and the latest log contains the consistent state of the global index. Any subsequent update to the global index is logged to the log file. When the transactional log exceeds a certain size, a new log will be created. The latest log file pointer is updated and the synced global index is saved to disk. Upon successful update to new log file, the redundant logs and outdated global index will be purged. The size limit of the log introduces a trade-off between background operations and startup overhead. When the limit is too small, new logs are heavily created which introduces extra overhead to save the global index to disk. On the other hand, when the limit is too large, the log replay process will increases the (re)open time of the system. To balance the trade-off, the system should periodically performs log rollovers during non-peak hours.

3.4 Complexity Analysis

Next we evaluate the complexity of secondary attribute queries on the LSM-based system integrated with NEXT. Given an LSM-based primary data storage with L_p levels and *1-leveling*[18, 37] data layout [4], we assume there are total N data entries, the number of data entries fit into a data block is B and the number of entries fit into a secondary index block is B_s ($B_s \gg B$ as the size of a secondary index entry is much smaller than the k-v pair).

The secondary attribute query cost of NEXT consists of the CPU costs on global index traversal and the I/O costs on matching data blocks fetching. For the global index component, as there are N/B_s index entries (i.e., total number of secondary index blocks), the CPU cost of index traversal

[4]Multiple sorted runs in first disk level and one sorted run for other levels

is $O(\log(N/B_s))$. After the global index outputs the data block locations, data block fetching is performed to find the matching data entries. As the locations outputed from the global index component may includes false positive (i.e., the query window overlaps with the secondary index block value range but not the entry attribute value), additional block fetching is expected. For the ease of analysis, we assume that there are ε false positive block fetching for each secondary query. Table 1 shows the I/O costs of both point query and range query (with selectivity of s) on LSM-based system with NEXT.

In addition, we also evaluate the effectiveness of existing secondary index methods with index navigation for comparisons. The secondary query cost of standalone secondary index method consists of two parts, the cost of searching the separated secondary index structure and the point lookup costs on the primary storage. Firstly, the I/O cost of fetching the target k-v pair for a specific primary key is $O(\phi L_P)$ [35], where ϕ is the false positive rate of the Bloom filter on primary keys. This is because during point lookup, only levels whose Bloom filter indicates the presence of the target key will be read and searched. Secondly, for an LSM-based standalone secondary index in the form of composite key, the I/O cost of point look is $O(\phi L_s)$ [5] where L_s is the number of levels in secondary index. For a range query, as matching entries distribute across SST files, each file or sort run needs to be searched and the data blocks contain the matching entries need to be read. Hence, the I/O cost of a range query on the secondary index is $O(s \cdot N/B'_s)$, where B'_s is number of index entries in the data block of the secondary index and s is the selectivity of range query. As secondary query on LSM-based system with standalone secondary index invloves the navigtion from secondary index to the primary storage, the total costs of the queries will be the sum of both processes and they are shown in Table 1. As both point query and range query involve the checkings of Bloom filters, the CPU cost of standalone secondary index will be $O(k)$ where k is the number of hash functions used in the Bloom filter.

As shown, for range queries, the index navigation costs ($sN \cdot O(\phi L_p)$) of existing standalone secondary index on the primary storage scale linearly with the output primary keys (sN). These point lookup operations on primary storage may incur significant overhead [25, 27] especially for long range queries. According to our evaluations, for an LSM-based system with 100 million entries , the point lookup operations take up to 78.6% of total cost on average for range queries with average selectivity of 6.076×10^{-6}. Different from standalone secondary index, the I/O cost of NEXT on range queries scale linearly with the number of matching blocks ($s \cdot N/B$). Through eliminating the index navigation and avoiding redundant reads on the same data block for different keys, the I/O cost can be reduced by B times. On the other hand, the point lookup cost of both methods have same order of magnitude.

Table 1. Query Costs Analysis

Methods	Query Types	CPU cost	I/O cost
NEXT	Point	$O(\log(N/B_s))$	$O(1+\varepsilon)$
	Range	$O(\log(N/B_s))$	$O(s \cdot N/B + \varepsilon)$
Standalone	Point	$O(k)$	$O(\phi L_s) + O(\phi L_p)$
	Range	$O(k)$	$O(s \cdot N/B'_s) + sN \cdot O(\phi L_p)$

4 Optimizations

To overcome limitations in index creation and query processing which will lead to extra query overhead, NEXT introduces two optimization strategies.

[5]Bloom filter and fence pointer are created at each level.

Index Creation. The naive per-file secondary index blocks creation method may result in oversize value range of the block, and hence large false positive rate of global index traversal. During the per-SST file secondary index block creation, a secondary index block is flushed when it exceeds the capacity limit. Then a tuple which contains the value range of the block and the block location is inserted to the global index component as a global index entry. As the creation of the secondary index blocks is not value-aware, the resulting value range of the block may be significant larger than what the data entries cover. For example, assuming the majority of the entries in the secondary index block fall within a range between a and $a + k$, and the last entry inserted to the block is $a + m$ where $m \gg k$. As a result, the value range of the secondary index block will be a to $a + m$ which is significant larger than the majority entries cover. During query processing, as the value range of the secondary index block is oversize, the false positive rate of global index traversal will significantly increase. As discussed in previous section, the high false positive rate of the global index will introduces extra I/O cost and hence hinder the query performance. To overcome this limitation, NEXT introduces a novel strategy for global index entries creation. Instead of only using one global index entry for each secondary index block, NEXT proposes to represent the secondary index block through multiple global index entries so that the attribute values within each index range are more condensed and hence the false positive rate can be reduced. Intuitively, we can partition the secondary index block value range equally and generate a global index entry for each range. However, this method is not effective to overcome the limitation. This is because when the values in the secondary index block are close to each other, the equally partition method will not improve the query performance, instead it will incur extra overhead as the global index component will contain more entries. Furthermore, when the values distribution in the secondary index block is skew, the equally partition method may not be able to index the outlier effectively as the oversize value range will still remain for the part that contains the outlier. To solve the value range oversize problem, NEXT proposes a value-aware strategy. Specifically, during the secondary index block creation, NEXT computes the increases in the value range after the flushing of each entry. When the increase in value range is larger than a pre-define threshold δ, we split the secondary index block value range and add an extra global index entry for the secondary index block. This strategy allows NEXT to adaptively add global index entries for secondary index block with oversize value range. As a result, large gaps between entries in the secondary index block will no longer be indexed and the entries within each indexed value range are more condensed. Hence, the false positive rate of global index traversal can be significantly reduced.

Query Processing. As the secondary attribute values and the data blocks have many-to-one relationship, same data block may be read into memory multiple times which result in redundant I/Os. To prevent redundant scan of the same data block, NEXT proposes to maintain a data block read history for each query. Different from block cache of the system which will check the data block again, the read history is used to skip data block retrieval if the data block is already scan during a query processing. As a result, each matching data block will only be scanned once for each query.

5 Online Index Creation

Online secondary index creation in an LSM-based database involves constructing and maintaining new secondary index structures on existing data store with minimal service interruptions. This is an important problem as it directly impacts the performance and scalability of the system in real-world workloads. In this section, we firstly discuss the online index creation problem. Then we propose a potential solution for creation of NEXT on an existing data store.

In an LSM-based database, online index creation generally consists of two steps. First, the system scans existing SST files to generate index entries. Second, the index entries are merged into the index

structure. For a standalone secondary index, this process creates a new entry for every existing key–value pair, which is then inserted into a separate index structure. By contrast, embedded secondary indexes require generating a filter or zone map for each SST file based on the attribute values. For NEXT, the per-segment indexes are created in a similar way to the embedded index, but additionally the global index is updated based on the newly created per-segment indexes. Since the creations of the embedded index and NEXT involve scanning and modifying the existing SST files, they tend to incur higher creation costs than standalone index.

To overcome the limitations of NEXT in the online index creation, we propose a "lazy" construction mechanism. In particular, as new writes arrive, SST files on lower level are more likely to be involved in subsequent compactions, we "delay" the per-segment index creations for these files. Starting from the largest level in the primary LSM-tree, NEXT constructs the index incrementally from high level to low level. During the index creation, when compactions are triggered for SST files on lower level, the index creations for the new files can then be conducted in parallel. This "lazy" construction mechanism allows some of the index construction overhead to be amortized to compactions and reduces redundant modifications of the compacted SST files. As a result, it can reduce the construction cost and allow the database to maintain high throughput while the new index is being built.

6 Experiments

In this section, we evaluate the proposed secondary index framework, NEXT, against existing LSM-based secondary index techniques. We first describe the experimental setup and implementations (Section 6.1). Then we conduct extensive experiments to evaluate NEXT's performance on basic operations with static workloads (Section 6.2) and overall performance with mixed workloads (Section 6.3). Lastly, we investigate the impact of the proposed optimizations (Section 6.4), the impact of data skew (Section 6.5), the case when multiple secondary indexes exist (Section 6.6) and the memory usage (Section 6.7).

6.1 Experimental Setup

Dataset and Workloads. Following previous works [25, 27, 33, 44], we evaluate the proposed secondary index framework with two types of workloads: *static* and *dynamic*. For static workload, we first conduct all the insertions with the creation of the secondary index. Then we issue queries on the LSM-based storage with static data. For static workload, the operations such as INSERT, GET and PUT are isolated. In this paper, we adopt two datasets for static workload experiments. One is the Tweet dataset. Similar to the benchmarks used in [33, 44], we collect 33 millions of geo-tagged tweets from Twitter Streaming API with a size of 13.7GB. The average size of each tweet is about 420 Bytes (each contains a unique Tweet ID and 10 other attributes) and the average number of tweets per user is 4.7. For the Tweet dataset, we select *UserID* and *Coordinates* of each tweet as the numerical and spatial secondary attributes for evaluation respectively. In addition, we also include one extra dataset, the OpenStreetMap Buildings (OSM Buildings), from SpatialHadoop [17]. It contains 115 millions of building objects with a size of 26GB. To evaluate the effectiveness of NEXT on different types of attributes, we selected the bounding rectangle (spatial) and perimeter of the boundary (numerical) of each building object as two secondary attributes. For static query workloads, we generate queries through randomly sampling attributes from the dataset as the query centres and apply query windows with different sizes to control the selectivity.

Different from static workload, data insertion, modifications and queries executions are interleaved in dynamic workload. To simulate realistic twitter-based operations, we adopt a open-sourced twitter-like workload generator, *chirp* [43], to generate three types of mixed (dynamic) workloads for secondary index evaluations. Table 2 shows the distribution of different operations for each

Table 2. Operations Frequency Ratios of Mixed Workloads

Workload Type	Operations Ratios			Total Number
	INSERT	QUERY	UPDATE	
Write Heavy	80%	20%	0%	4 million
Read Heavy	20%	80%	0 %	
Update Heavy	40%	20%	40%	

mixed workload. For insertions, the insert key-value pairs are selected from a seed set of tweets collected from Twitter Streaming API and the insert sequence is based on the respective *Time* attribute. For queries, we only generate queries on secondary attributes (*UserID & Coordinates*) with a fixed size query window. The query center is randomly selected from the inserted data. To generate *UPDATE* operations, we first randomly select primary key from existing data then insert a different tweet with the selected primary key.

Implementations. We implemented NEXT on top of RocksDB [18], an LSM-based key-value storage system that is widely adopted by both industry and academical studies [4, 12, 15]. RocksDB only supports primary index and queries on the key of each data entry (KV-pair). We enable secondary index creation in RocksDB through adding new block based table options and secondary index builder which allow the system (i.e., block_based_table_builder.cc) to create secondary index meta block for each SST file during block based table construction. We implement Algorithm 1 (in Section 3.3) to the code of the index builder (index_builder.cc) and maintain the global index component together with the version storage information (i.e., version_builder.cc, version_edit.cc and version_set.cc in RocksDB) which resides in memory. For spatial secondary index global component, we adopt an open-source in-memory R-tree [30] as the container. For numerical index, we utilize the 1-Dimensional version of the R-tree to act as a interval tree. The secondary attribute queries are supported through adding a new type of iterator based on the secondary attribute and the created secondary index. All algorithms are implemented in C++.

Compared Systems. We compare NEXT to three classes of state-of-the-art secondary index techniques in LSM-based storage systems. Based on the findings of [33], none of these techniques dominates the others. We implement a representation from each class on the same system as NEXT, *i.e.,* RocksDB, and optimize it for a fair comparison. The details of each compared techniques are as follow:

- **Eager.** We compare NEXT to a secondary index with eager update strategy which follows the same way of secondary index maintenance in RDBMS [22]. This method is adopted by the popular MongoDB. Following the tuning manual [29], the conventional secondary index is fitted entirely in RAM for fast query processing. We adopt in-memory index containers (i.e., B-tree [8] and R-ree [30]) for the secondary index implementations.
- **Embedded.** As the embedded index (Bloom filters and zone map) proposed in [33] heavily depends on the index attribute's time correlations, we compared NEXT to a more recent embedded secondary index technique proposed by [21]. We implement the technique through creating a secondary index for each SST file. During query processing, an iterator on all SST files' secondary indexes is utilized to searched for the target entries.
- **Standalone.** We compare NEXT to the the standalone secondary index with composite keys and lazy update which is widely adopted by systems like AsterixDB [1] and Spanner [9]. We implement the standalone secondary index through utilizing another LSM-tree to store the composite keys and adopt an in-memory update memo [41] for consistency maintenance.

Table 3. OSM Buildings Workload Latency (in seconds)

Secondary Index	Spatial			Numerical		
	Small	Medium	Large	Small	Medium	Large
Eager	1.07	40.49	812.63	7.80	133.29	657.80
Embedded	19.79	87.13	302.77	6.94	36.90	**170.35**
Standalone	**1.00**	58.63	785.48	5.73	52.07	512.52
NEXT	1.63	**34.22**	**263.12**	**2.89**	**20.15**	202.51

Table 4. Tweet Workload Latency (in seconds)

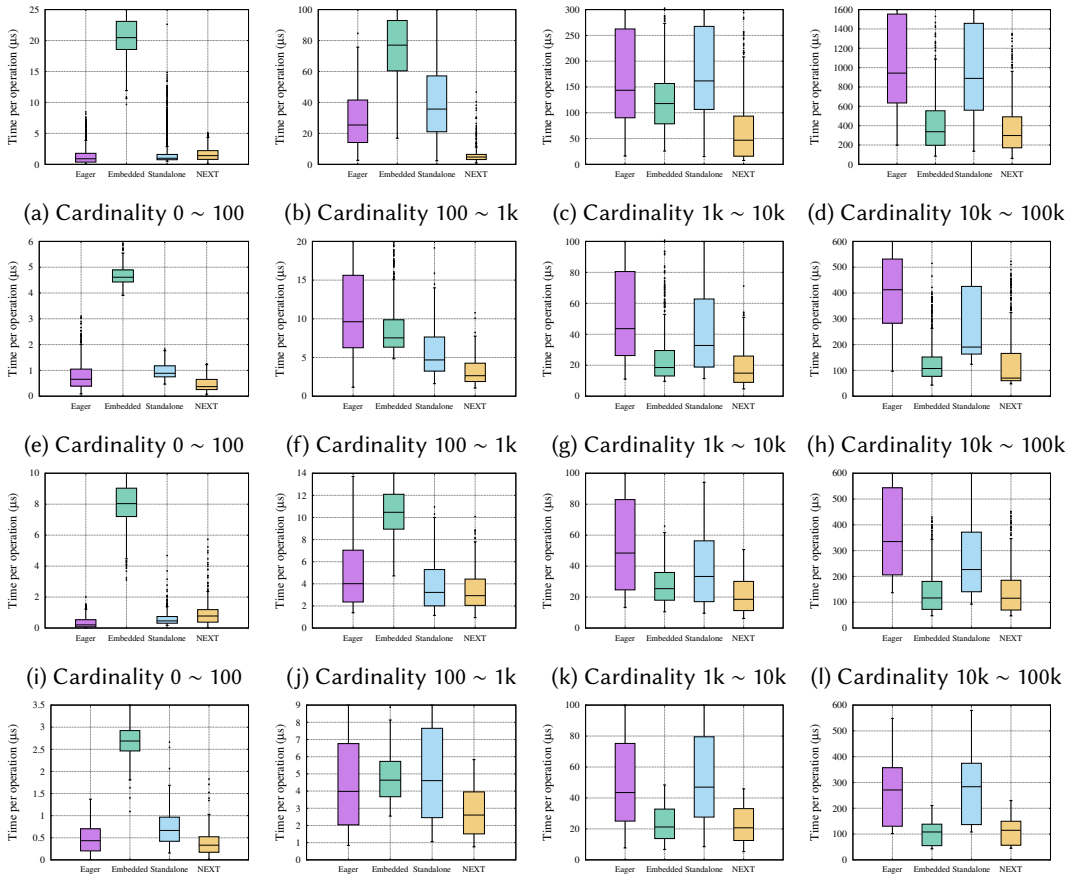
Secondary Index	Spatial			Numerical		
	Small	Medium	Large	Small	Medium	Large
Eager	**0.82**	4.77	203.67	1.19	11.52	118.27
Embedded	8.49	10.07	73.30	3.13	7.47	**49.28**
Standalone	0.92	3.80	138.37	1.52	12.70	124.49
NEXT	1.06	**2.86**	**69.83**	**0.79**	**5.75**	52.29

LSM Store Configurations. For all LSM-based storages used in this paper, according to the configuration tuning guide [18], we set both memory buffer size and block cache to be 64 MB. We adopte a block size of 4 KB and data compression is turned off.
Platform. All experiments in this paper are conducted on a server with a 10-core Intel i9-10900X CPU, which runs Ubuntu 20.04 LTS. The server is equipped with 64 GB DRAM and 1 TB SSD.

6.2 Results on Static Workloads

In this section, we evaluate the basic operation costs of the compared secondary index mechanisms on static workloads.

6.2.1 ***Query Performance.*** To investigate the effect of different secondary index techniques on non-index-only secondary attribute query performance, we synthesise three query workloads with different query windows to represent workloads with different selectivity for each attribute of each dataset. Each workload contains 1000 queries with randomly selected query centres. For OSMB dataset, as both secondary attributes (bounding rectangle and perimeter) are based on geographic coordinates, we set three query windows with 0.001, 0.01 and 0.1 decimal degrees which results in average result cardinalities of 17, 608 and 18753 (403, 3929 and 40064) for spatial (numberical) attribute. For Tweet query workloads, we use the same decimal degrees as OSMB for *Coordinates* attribute and numerical ranges of 10k, 100k and 1000k for *UserID* attribute. The average result cardinalities are 56, 350 and 14887 (108, 1107 and 11349) for spatial (numerical) attribute. Table 3 and Table 4 show the workload latency for OSMB and Tweet datasets. We use "Small", "Medium" and "Large" to represent workloads with three ranges of selectivity respectively. As shown, both datasets share the common patterns. **For workload with small selectivity, standalone secondary index (both Eager and LSM-based standalone) and NEXT significantly outperform embedded secondary index.** This is because embedded index in each SST file needs to be queried to locate the target entries. In contrast, NEXT utilizes the global index to locate the target data block directly which enjoys a smaller query overhead. Similarly, standalone secondary index can quickly locate matching primary keys without searching all SST files. For spatial workload with small selectivity, standalone and eager outperform NEXT. One possible reason can be the false positive rate for



(a) Cardinality 0 ~ 100 (b) Cardinality 100 ~ 1k (c) Cardinality 1k ~ 10k (d) Cardinality 10k ~ 100k

(e) Cardinality 0 ~ 100 (f) Cardinality 100 ~ 1k (g) Cardinality 1k ~ 10k (h) Cardinality 10k ~ 100k

(i) Cardinality 0 ~ 100 (j) Cardinality 100 ~ 1k (k) Cardinality 1k ~ 10k (l) Cardinality 10k ~ 100k

(m) Cardinality 0 ~ 100 (n) Cardinality 100 ~ 1k (o) Cardinality 1k ~ 10k (p) Cardinality 10k ~ 100k

Fig. 5. Query latency with different result cardinalities. (a) to (d) for OSMB Spatial; (e) to (h) for OSMB Numerical; (i) to (l) for Tweet Spatial; (m) to (p) for Tweet Numerical.

spatial index is larger which results in redundant I/O cost. For numerical workload with small selectivity, NEXT outperforms all existing secondary index up to 75%. **For workload with medium selectivity, NEXT outperforms all compared secondary index techniques. NEXT can reduce the work latency from 15.5% to 84.9%.** When the workload selectivity increases, as discussed in previous sections, standalone secondary index techniques will suffer from the significant overhead from index navigation. As a result, the query performances of both eager and standalone secondary index are worse than embedded index and NEXT. **For spatial workload with large selectivity, NEXT achieves the best query performance. For numerical attribute workloads, embedded slightly outperforms NEXT.**

In addition to workload latency, we also evaluate the effect of different secondary index techniques on single query execution time with different results cardinalities. Figure 5 demonstrate the query latency in the form of box-and-whisker plots. These figures show that **NEXT has top performance for queries across all results cardinalities**. When the cardinality is small, the point lookup overhead on primary table for target primary keys are negligible. As a result, both eager and standalone secondary index performs better. Nevertheless, when the results cardinality increases,

the point lookup overhead becomes significant and the query execution times with eager and standalone secondary index will be larger than those of embedded index and NEXT. For embedded secondary index, as all embedded indexes need to be searched during query process, it does not perform well for queries with small cardinality. This is because there are significant amount of redundant reads. For queries with larger cardinality, as the target entries scatter across all SST files, embedded index will have better performance.

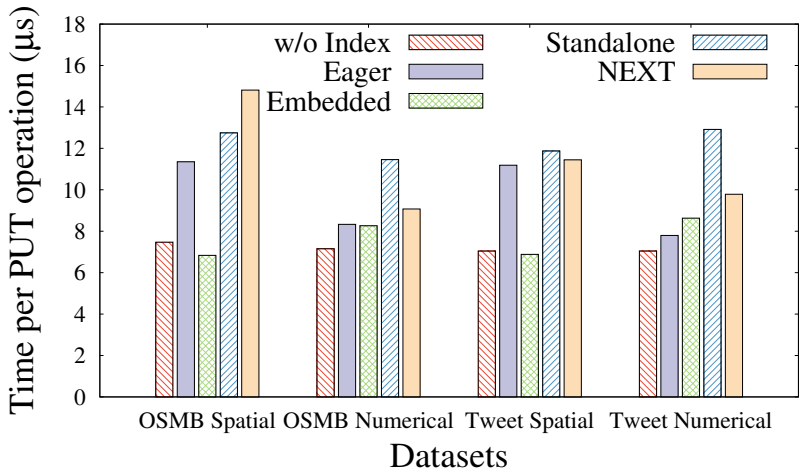
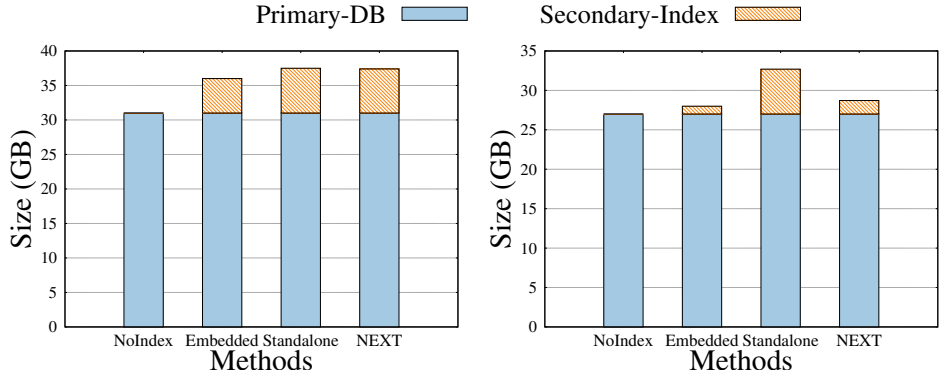


Fig. 6. Put Performance

6.2.2 ***Insert Performance.*** In this experiment, we evaluate how the insert operation is affected by the various secondary index techniques in term of average operation execution time and database size. For the static datasets, the insert workloads (without any update) for OSM Buildings and Tweet contain 100m and 30m entries respectively. Figure 6 shows the average execution times of each insert operation on different datasets with different secondary indexes (i.e., secondary indexes on different attributes). As shown, the embedded index has relatively better insert performance. This is because the embedded index does not maintain a separated structure and the embedded index creations are performed together with the SST file creations in the background thread pool. For OSM Buildings (OSMB) spatial and Tweet Spatial datasets, the average insertion times of the embedded secondary index are similar to those of system w/o index, this is because when the merge and compaction tasks can be executed smoothly, the background process will not introduce extra overhead on data insertion nor write stall. For eager secondary index, as the index is maintained in RAM, the insert operation is better than standalone and NEXT. **Compared to standalone which needs to maintain another LSM-tree to store the composite keys, NEXT has better performance except for OSMB with spatial attribute.** This is because the per-segment component of NEXT is similar to the embedded index which is created through background process and the global component is maintained in RAM. For OSMB spatial, the insertion performance of NEXT is slower than standalone, one reason can be the creation of both per-file component and global index for complicate spatial objects involve frequent non-trivial data sorting and node splitting which introduce extra cost to data insertion.

Figure 7 shows the **database sizes** with different secondary index techniques. For NEXT, the secondary index consists of a per-segment embedded component and a standalone global index component. Compared to Embedded secondary index, NEXT will introduce extra storage overhead for the global index component. Compared to standalone secondary index, the storage overhead



(a) OSM Buildings Spatial (b) OSM Buildings Numerical

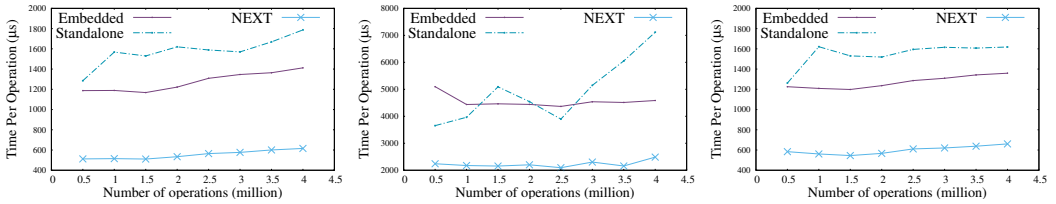
Fig. 7. Database & Secondary Index Size

of NEXT is comparable. As shown, the size of the standalone secondary index is 20.97% of the primary table while the sizes of embedded index is 16.13%. For NEXT, the total size of secondary index takes 20.65% of the primary table which is slightly smaller than that of standalone secondary index. For dataset Tweet, similar trends are observed.

6.2.3 ***Summary of Results.*** We next summarize the static workload experimental results. For query on static data, NEXT has the best overall performance. For small and large results cardinalities, NEXT has comparable or even better performance than the best of existing secondary indexing techniques. For queries with medium size of cardinality, NEXT outperforms all existing techniques as it eliminates the costly index navigation and avoids redundant reads. For insert performance and space overhead, NEXT is comparable to the standalone secondary index. Although for OSMB Spatial dataset, the PUT performance of NEXT is slightly slower ($\sim 2\mu s$), as we will show in the mixed workload experiment, even for the write-heavy workload, the advantages on query performance will outweigh the difference in PUT performance. Overall, the experimental results demonstrate that NEXT can enhance query performance without introducing significant overhead on insertion and storage.

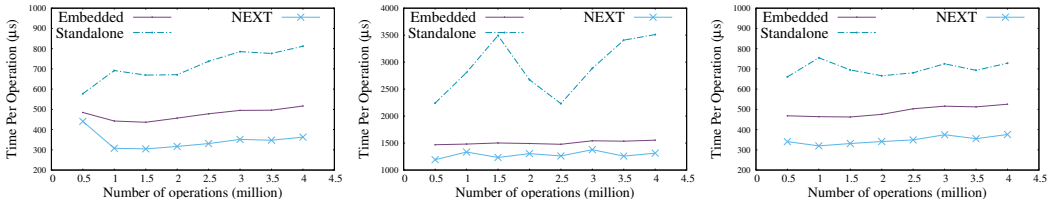
6.3 Results on Mixed Workloads

In this section, we evaluate NEXT, the LSM-based standalone and embedded index under dynamic workloads. We do not consider Eager in this experiment as it is shown to be unusable in previous work [33]. Different from the static workload experiment that we can bulk load and fit the eager index entirely in memory, dynamic workload experiment on eager index involves expensive random accesses. The dynamic workloads consists of different types of operations (Insert, Update and Query) which are more representative of real-world applications. Each workload contains 4 million operations and only one secondary attribute is indexed and queried at one experiment. The query windows adopted in the mixed workload experiment are 0.01 decimal degree and 100k value range for spatial and numerical secondary attributes respectively. Before executing the mixed workload, we pre-load the system with 20 millions of records. As it is challenging to isolate the performance of each type of operations, we record the average operation latency every 500k operations. Figure 8 and Figure 9 show the overall performance of different secondary index techniques on Tweet datasets with different type of value attributes being indexed.



(a) Write Heavy Workload (b) Read Heavy Workload (c) Update Heavy Workload

Fig. 8. Overall performance comparison for Spatial Mixed Workloads

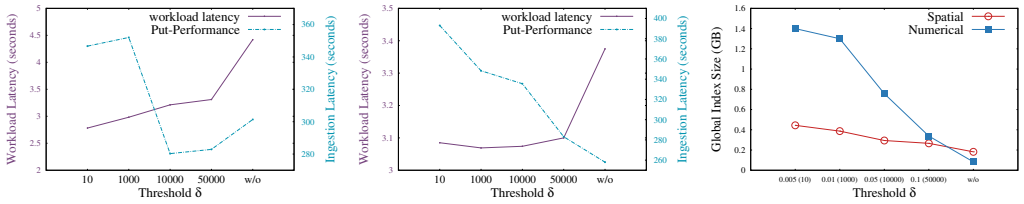


(a) Write Heavy Workload (b) Read Heavy Workload (c) Update Heavy Workload

Fig. 9. Overall performance comparison for Numerical Mixed Workloads

Figure 8a and 9a show the average operations latency for write heavy workloads. As shown, as more data being inserted, the average operations latency for all systems increases. However, for embedded secondary index and NEXT, there are uncommon drops in operation latency at the beginning of workload execution. One possible explanation can be at the start of workload execution, data insertions are buffered in main memory which does not affect the other operations significantly. At the same time, the embedded indexes and target data blocks are cached as query operation being performed. As a result, the query operation latency is reduced due to the cached data. Compared to the LSM-based standalone secondary index, the increases in operations latency are lesser for embedded index and NEXT. As shown in the static workload experiment, embedded secondary index and the secondary index blocks in NEXT are created naturally with the SST file construction which will not introduce significant overhead to insert opertion. In addition, for query operation, the new arrival data will introduce additional I/O cost only when the data is flushed to disk as new SST file. As the global index component directs the query operation to target data blocks directly, NEXT is able to mitigate the impact from new flushed data blocks. For update heavy workload, similar trends as write heavy workload are observed. This shows that update memo can mitigate the impact of data update on the query performance.

For read heavy workload, as shown in Figure 8b and 9b, there are fluctuations on operations performance of standalone secondary index and NEXT. For standalone secondary index, the fluctuations may be due to the *1-leveling* data layout. As the standalone LSM-tree (secondary index) maintains multiple sorted runs in the first disk level, it needs to search all sorted runs together with the Memtable for each query operations. When data is inserted to the system, the number of sorted runs in the first disk level will increase, resulting in higher query operation cost. With more data being inserted, compactions will be conducted to merge the sorted runs in first disk level to higher level. Hence, the query operation performance will be improved. For NEXT, the degree of fluctuations is smaller compared to that of standalone secondary index. This is because the fluctuations on NEXT is solely due to the Memtable. As NEXT only index data entries reside in disk, it needs to scan the Memtable for every query operation. When new insertions are buffered in Memtable, the query operation cost may be increased. When the Memtable is flushed to disk as



(a) Tweet Spatial (b) Tweet Numerical (c) Global Index Size

Fig. 10. Effect of the Proposing Optimization on Global Index

new SSTable, the query operation cost will then be reduced. For embedded secondary index, as it search all embedded index for each query operation and the number of embedded index changes slowly, its performance is relatively stable for read heavy workload.

Summary: NEXT is shown to be able to handle different mixed workloads and achieve better performance than existing standalone secondary index and embedded secondary index techniques.

6.4 Effect of the Proposed Optimization

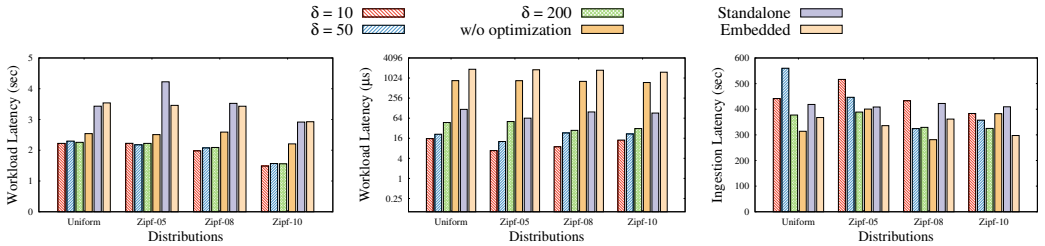
To overcome the problem of the oversize index range (discussed in Section 4), we propose a novel packing strategy which adding more index entries for the same indexed block when the changes in index range for a new entry exceed a threshold δ. To investigate how the proposed strategy affects the basic operation performance and the global index size, we conduct experiments on systems with various values of δ. Figure 10a and 10b shows the impact of δ on ingestion performance and query workload latency for different secondary attributes on Tweet dataset. **As shown, compared to system w/o the optimization, adopting our proposed packing strategy can reduce the query workload latency up to 25%.** In addition, when the value of δ decreases, the index entries within the index range will be more condensed and hence the workload latency will be lower. However, a smaller value of δ does not guarantee a better query performance. As shown in 10b, when the value of δ decreases from 1000 to 10, the workload latency slightly increases. This is because when the index range is sufficiently condensed, adding more entries for the index block will only increase the query overhead without reducing the false positive rate. Hence, to ensure the best query performance, we need to adjust the δ value carefully for different datasets. We will explore the problem of finding the appropriate value of δ for different datasets in future work.

For ingestion performance, as smaller δ will result in more secondary index entries in global index component, the ingestion latency will be higher. This is shown in Figure 10b. However, for spatial attribute, a different trend is observed. When compared to systems w/o the optimization and $\delta = 0.1$, the system with $\delta = 0.05$ has a lower ingestion latency. One possible reason can be that when the index range is smaller, the global index creation algorithm can better pack the entries with lesser node splitting. As a result, the ingestion performance is better given that the increase in number of index entries is not significant. When the number of index entries increases significantly (i.e., when δ decreases from 0.05 to 0.01), the ingestion performance will degraded.

Lastly, Figure 10c shows how the global index size changes with δ. For numerical attribute, the global index size will be 1.4 GB when $\delta = 10$. Although it is significantly larger than the global index of system w/o the optimization, the total secondary index size is only 3.56% larger than the standalone secondary index size. This optimization strategy introduces a trade-off between index size and query performance. To meet different application requirements, user can adjust the value of δ accordingly.

6.5 Impact of data skew.

Since the false positive rate of NEXT's global index is closely related to the data distributions, it is important to understand the impacts of data skew on our proposed optimization. In this section, we firstly synthesise four additional numerical attributes in Tweet dataset which follow uniform distribution and 3 levels of Zipf distribution (i.e., shape factor 0.5, 0.8 and 1.0) respectively. Each attribute contains integer values range from 1 to 100 millions. For range query workload, it contains 1000 queries with randomly selected query centres and a fixed query range. Since it is common for a database system to store categorical variables in numerical form, we synthesise point query workload on the synthetic attributes to evaluate the performance on categorical attribute. Figure 11 show the performances of NEXT with different δs and existing secondary indexing techniques (i.e., standalone and embedded) on different data distributions. As shown in Figure 11a, for *Zipf-1.0* and *Zipf-0.8* which have more skew data distributions, the effectiveness of our proposed optimization is more significant. The differences between $\delta = 200$ and *w/o optimization* are 19% and 29% respectively while those for *Zipf-0.5* and *Uniform* are 11%. For *Zipf-1.0*, as tuples are skew toward smaller values, the range query results are smaller and hence it has the lowest query latency. For range workload, NEXT with and without optimization outperform both existing methods. For point query on categorical data, the false positive rate problem will be more significant. As shown in Figure 11b, the reductions in workload latency through applying our proposed optimization are more remarkable. This implies that our proposed optimization is able to greatly reduce the false positive rate of the global index and hence improve the query performance for different data distributions. Without our proposed optimization, the point workload latency of NEXT is higher than that of standalone secondary index as the index navigation cost is negligible for point query. However, with our proposed optimization, NEXT outperforms both existing methods for all distributions. Figure 11c shows the ingestion performances under different data distributions. One interesting observation is that the ingestion performance of *Zipf-1.0* without optimization is slower than $\delta = 200$. The reason for this is because the data is skew toward smaller values and there are more node splittings during the global index creation which hinder the data ingestion performance.



(a) Range Workload Performance (b) Point Workload Performance (c) Ingestion Performance

Fig. 11. Effect of Data Skew on the Proposing Optimization

6.6 Existence of Multiple Secondary Indexes

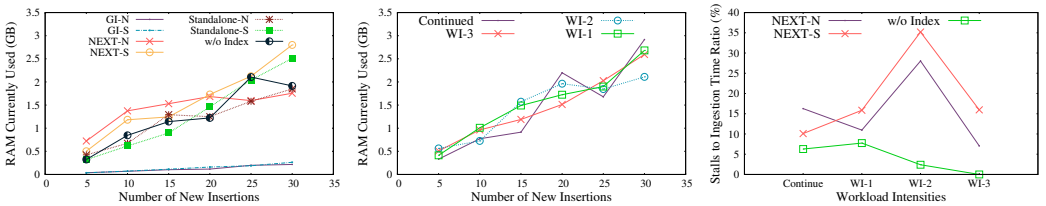
Since it is common for database system to build multiple secondary indexes on the same table, in this section, we evaluate NEXT's performance when multiple secondary indexes are constructed. Table 5 shows the ingestion performance, storage overhead and query performance of NEXT and the standalone secondary index when two types of secondary indexes are created together. As shown, the ingestion performance and storage overhead of NEXT are similar to those of the standalone index. However, compared to the results in Section 6.2, existence of multiple secondary indexes

Table 5. Multiple Secondary Index Performance

Dataset	Tweet				OSM Buildings			
Sec. Index	Standalone		NEXT		Standalone		NEXT	
Avg. Put Time (μs)	15.16		17.07		20.13		18.74	
Pri. DB Size (GB)	13		16		27		35	
Sec. Index Size (GB)	1.90		0.78		5.70		0.25	
Sec. Index Size 2 (GB)	1.90		0.44		5.70		1.23	
Total DB Size (GB)	16.80		17.23		38.40		36.48	
Workload Latnecy								
Attributes	2D	1D	2D	1D	2D	1D	2D	1D
Small (secs)	0.86	1.49	1.03	0.79	0.51	5.36	1.48	4.18
Medium (secs)	4.23	12.66	2.67	6.08	7.27	48.25	4.70	20.38
Large (secs)	165.85	129.03	69.20	51.30	210.38	494.35	83.41	183.49

will incur larger *PUT* time and higher space consumption. Similar to Section 6.2, we evaluate the query performance using three workloads with different query ranges. The spatial workload and numerical workload are represented as 2D and 1D respectively. As shown, the query performance is similar to the single index case.

6.7 Memory Overhead Analysis



(a) Memory Usage of Different Methods (b) Memory Usage vs Workload Intensities (c) Write Stalls vs Workload Intensities

Fig. 12. Memory Usage and Write Stalls

In this section, we evaluate the memory overhead of NEXT and its impacts on the database system. Figure 12a demonstrates the memory usage of NEXTs (with spatial and numerical secondary index), standalone secondary indexes and RocksDB without secondary index during the Tweet dataset insertions. We measure the total physical memory (RAM) in used and the size of the global index (GI) after every 5 millions of insertions. As shown, the global index only takes a small portions of the used RAM. Furthermore, the memory usage of NEXT is similar to the standalone secondary index and RocksDB W/O seconary index. In addition, we evaluate the memory usage and write stalls of NEXT under four different workload intensities (i.e., continued, WI-1 = 2 $inserts/\mu s$, WI-2 = 1 $inserts/\mu s$ and WI-3 = 0.5 $inserts/\mu s$). As shown in Figure 12b, for workload with higher intensity, the fluctuations of the memory usage will be greater. This implies that more RAM is used for compactions and MemTables maintenance. Figure 12c shows the write stalls incured by NEXT under different workload intensities. As the workload intensities decreases, there will be higher write stalls for NEXT. One reason for this observation is that internally RocksDB will batch write requests from different thread to improve the performance, but the lower workload intensity may not fully utilize this optimization which results in higher write stalls. For NEXT, the creation of secondary index will slow down the compactions and introduces additional write stalls for high

workload intensity. This limitation can be mitigated through increasing the number of threads for background operations or increase the size of the MemTable to reduce the write amplification.

7 Conclusion

In this paper, we propose NEXT, a new secondary index framework for general-purpose LSM-based key-value storage system. Different from existing secondary index for key-value databases, NEXT adopts a two-level index structure which aiming to avoid the expensive index navigation during seondary queries processing. Particularly, NEXT proposes to create secondary index blocks for each SST file to map the secondary attribute to the associated data block location directly. In addition, NEXT introduces a global index component which narrow down searching space through directing the query operation to target secondary blocks. To optimize query perfromance, this paper also introduces two strategies to improve NEXT's effectiveness. Extensive experiments on different workloads and types of secondary attributes demonstrate that NEXT is able to improve query performance on secondary attributes without introducing extra overhead on data ingestion nor space consumption.

Acknowledgments

This research is supported in part by Singapore MOE AcRF Tier-2 grant MOE-T2EP20223-0004, and MOE AcRF Tier-1 grant RT6/23.

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916.

[2] Sattam Alsubaiee, Alexander Behm, Vinayak R. Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (2014), 841–852.

[3] Apache. [n. d.]. Cassandra. http://cassandra.apache.org/.

[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1185–1196.

[5] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*. ACM, 100–109.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26.

[7] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1257–1270.

[8] Google Code. [n. d.]. Google Code CPP-Btree. https://code.google.com/archive/p/cpp-btree/source.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8.

[10] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 155–171.

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 79–94.

[12] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 505–520.

[13] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 365–378.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, 205–220.

[15] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org.

[16] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4 (2021), 26:1–26:32.

[17] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 1352–1363.

[18] Facebook. [n. d.]. RocksDB. https://github.com/facebook/rocksdb/.

[19] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*. ACM, 75–88.

[20] Google. [n. d.]. LevelDB. https://github.com/google/leveldb/.

[21] Junjun He and Huahui Chen. 2022. An LSM-Tree Index for Spatial Data. *Algorithms* 15, 4 (2022), 113.

[22] MongoDB Inc. [n. d.]. MongoDB. https://www.mongodb.com/.

[23] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. 2020. Cuckoo Index: A Lightweight Secondary Index Structure. *Proc. VLDB Endow.* 13, 13 (2020), 3559–3572.

[24] Cockroach Labs. [n. d.]. CockroachDB. https://www.cockroachlabs.com/.

[25] Fei Li, Youyou Lu, Zhe Yang, and Jiwu Shu. 2020. SineKV: Decoupled Secondary Indexing for LSM-based Key-Value Stores. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*. IEEE, 1112–1122.

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28.

[27] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proc. VLDB Endow.* 12, 5 (2019), 531–543.

[28] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[29] MongoDB. [n. d.]. MongoDB Manual. https://www.mongodb.com/docs/manual/.

[30] Gero Mueller. [n. d.]. R-Trees: A Dynamic Index Structure for Spatial Searching. http://www.superliminal.com/sources/sources.htm.

[31] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[32] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14 (2009).

[33] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 551–566.

[34] 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 (2017), 2037–2048.

[35] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2489–2497.

[36] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference*

2020, online conference [Portland, OR, USA], June 14-19, 2020. ACM, 893–908.

[37] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and Analyzing the LSM Compaction Design Space (Updated Version). *CoRR* abs/2202.04522 (2022).

[38] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 217–228.

[39] Timos K. Sellis. 2000. Review - The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD Digit. Rev.* 2 (2000).

[40] Jaewoo Shin, Jianguo Wang, and Walid G. Aref. 2021. The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2285–2290.

[41] Jaewoo Shin, Jianguo Wang, and Walid G. Aref. 2023. An Update-intensive LSM-based R-tree Index. *CoRR* abs/2305.01087 (2023).

[42] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. IEEE Computer Society, 68–79.

[43] UC-Reverside-DatabaseLab. [n. d.]. Chirp: A Twitter-like Workload Generator. http://www.cs.ucr.edu/~ameno002/benchmark/.

[44] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2023. Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 817–832.

Received October 2024; revised January 2025; accepted February 2025