Efficient Secondary Attribute Lookup in Key-Value Stores

Mohiuddin Abdul Qader, Shiwen Cheng, Abhinand Menon, Vagelis Hristidis
Department of Computer Science & Engineering
University of California, Riverside, California, USA
{mabdu002,schen064,ameno002,vagelis}@cs.ucr.edu

ABSTRACT

NoSQL databases like key-value stores achieve fast write throughput and fast lookups on the primary key. However, many applications also require queries on non-primary attributes. For that, several NoSQL databases have added support for secondary indexes. To our best knowledge, little work has studied how to support secondary indexing on pure key-value stores, which are a fundamental and popular category within the NoSQL databases range.

We propose a novel lightweight secondary indexing technique on log-structure merge-tree (LSM tree)-based key-value stores, which we refer as "embedded index". The embedded index, which utilizes Bloom filters, is very space-efficient, and achieves very high write-throughput rates, comparable to non-indexed key-value stores. It is embedded inside the data files, that is, no separate index table is maintained. To our best knowledge, this is the first work to use Bloom filters for secondary indexing. The embedded index also simplifies the transaction management, compared to alternative stand-alone secondary indexing schemes. For the cases when range query on secondary attributes in embedded index is required, we also propose to apply a modified version of interval tree to keep track of the attribute value range in each data block to support that.

As a second contribution, we have defined and implemented two "stand-alone indexing" techniques (i.e. separate data structures maintained for secondary indexes) on key-value stores that borrow ideas from the secondary indexing strategies used in column-based NoSQL and traditional relational (SQL) databases.

We implemented all indexing schemes on Google's popular open-source LevelDB key-value store. Our comprehensive experimental and theoretical evaluation reveals interesting trade-offs between the indexing strategies in terms of read and write throughputs. A key result is that the embedded index is superior for high write-throughput requirements. We created and published a realistic Twitter-style read/write workload generator to facilitate our experiments. We also published our index implementations on LevelDB as open source.

1. INTRODUCTION

In the age of big data, more and more services are required to ingest high volume, velocity and variety data, such as social networking data, smartphone Apps usage data and click through data in large search/recommendation systems. NoSQL databases were developed as a more scalable and flexible alternative to relational databases. NoSQL databases, such as BigTable [13], HBase [18], Cassandra [20], Voldemort [17], MongoDB [6] and LevelDB [4] to name a few, have attracted huge attention from industry and research communities and are widely used in products.

NoSQL systems like key-value stores are particularly good at supporting two capabilities: (a) fast write throughput and (b) fast lookups on the primary key of a data entry. However, many applications also require queries on non-key attributes which is a functionality commonly supported in RDBMs. For instance, if a tweet has attributes tweet id, user id and text, then it would be useful to be able to return all (or more commonly the most recent) tweets of a user. However, supporting secondary indexes in NoSQL databases is challenging because secondary indexing structures must be maintain during writes, while also managing the consistency between secondary indexes and data tables. This significantly slows down writes, and thus hurts the system's capability to handle high write throughput which is one of the most important reasons why NoSQL databases are used. Secondary indexes also complicate the transaction management.

Table 1 shows the operations that we want to support. The first three are already supported by existing key-value stores like LevelDB. Note that the secondary attributes and their values are stored inside the value of an entry, which may be in JSON format: $v = \{A_1 : val(A_1), \cdots, A_l : val(A_l)\}$, where $val(A_i)$ is the value for the secondary attribute A_i. For example, a data entry that consists of tweet id $t3$, user id $u1$ and text $text3$ can be written using *PUT(t3, {u1, text3})*. Then, entry (u1, {t3}) should be added to the secondary index table for the user id attribute. However, there could be an existing posting list of u1, e.g., it could be *(u1, {t2,t1})*. Thus the new update should be merged with the existing posting list at some point, so it eventually becomes *(u1, {t3,t2,t1})*.

Current NoSQL systems have adopted different strategies to support secondary indexes (e.g., on the user id of a tweet). For instance, MongoDB uses B-tree for secondary index, which follows the same way of maintaining secondary indexes as traditional RDBMs.

On the other hand, several log-structure merge-tree (LSM tree) [22] based NoSQL systems (e.g., Cassandra) store a secondary index as an LSM table (column family) similarly to how the data table is stored, and perform append-only updates (which we refer as "lazy" updates) instead of in-place updates. In particular, going back to the example operation *PUT(t3, {u1, text3})*, recent versions of Cassandra have adopted a lazy update strategy: it issues a *PUT(u1, {t1})* in the user id index table without retrieving the existing postings list for $u1$. The old postings list of u1 is merged with *(u1, {t1})* at a

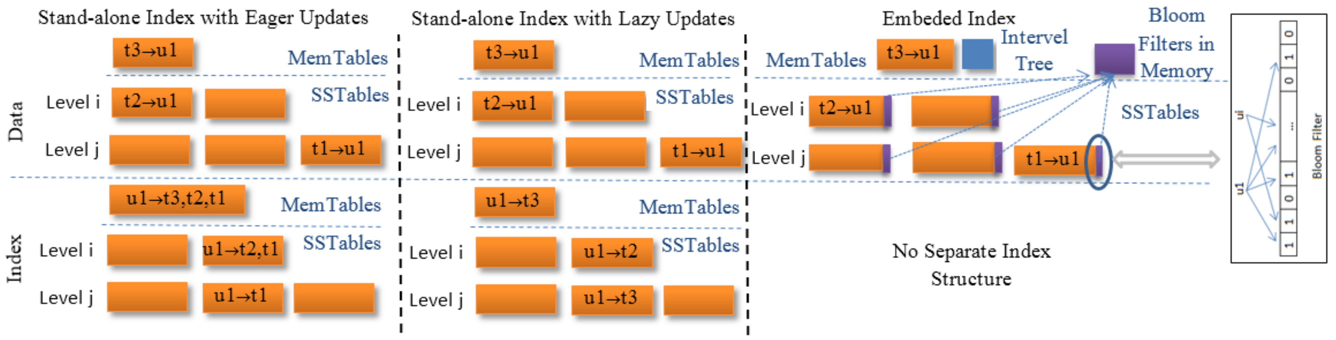


Figure 1: Comparison between eager and lazy updates in stand-alone index, and embedded index, right after *PUT(t3, {u1, text3})*. We use the notation *key* $\rightarrow$ *value*. For simplicity, we do not show the *text* attribute.

Table 1: Set of operations in a key-value store.

Operation	Description
GET(k)	Retrieve value identified by primary key k.
PUT(k, v)	Write a new entry $\langle k, v\rangle$ (or overwrite if k already exists), where k is the primary key.
DEL(k)	Delete the entry identified by primary key k if any.
LOOKUP(A, a)	Retrieve the entries with $val(A) = a$, where A is a secondary attribute and a is a value for this attribute
LOOKUP(A, a, K)	Retrieve the K most recent entries with $val(A) = a$.
RANGELOOKUP(A, a, b, K)	Retrieve the K most recent entries with $a \leq val(A) \leq b$.

later time, during the periodic compaction phase[1]. A drawback of the lazy update strategy is that reads on index tables become slower, because they have to perform this merging at query time. We have implemented both eager and lazy stand-alone indexing strategies for key-value stores that closely approximate the above described strategies.

In addition to implementing and evaluating such stand-alone indexes for key-value stores, we propose a novel lightweight secondary indexing approach, which we refer as *"embedded index."* In contrast to stand-alone indexes, the embedded index does not create any specialized index structure. Instead, it attaches a memory-resident Bloom filter [12] signature on each data block for each indexed secondary attribute. Note that Bloom filters are already popular in NoSQL systems to accelerate the lookup on the primary key, but we propose for the first time also using them for secondary attributes. In order to support range query on secondary attribute, embedded index also maintains an in-memory data structure per indexed attribute: a modified version of interval tree storing the range of secondary attribute values in each data block in SSTable.

Figure 1 captures the differences between the three indexing strategies (eager and lazy updates on stand-alone index, and embedded index) on LSM-based key-value stores. In an LSM-style storage structure, data are normally organized as levels (three levels shown in Figure 1 where the top is in-memory and the bottom two on disk) with older data in lower levels, which are larger than recent (upper) levels. Each orange rectangle in Figure 1 represents a data file. The state shown is right after the following sequence of operations *PUT(t1, {u1, text1}),...,PUT(t2, {u1, text2}),...,PUT(t3, {u1, text3})*.

More details on the embedded index are provided in the rest paper, but an overview of answering LOOKUP(A, a, K) is as follows: A set of Bloom filters is attached to every data file that can help determine if a secondary attribute value exists in the file or not. Note that a copy of all Bloom filters (the violet blocks in the Embedded Index part) from all disk blocks (files contain blocks) is stored in memory for faster access. We can efficiently answer a query for the top-K most recent entries (e.g., tweet ids) based on a secondary attribute value (e.g., user id = u1) with the help of the embedded index as follows: We start from the in-memory blocks of LSM, and move down the hierarchy one level at a time, checking the blocks' Bloom filters, until K matched entries have been found.

We will show that the embedded index has much lower maintenance cost compared with stand-alone index techniques. In terms of implementation, the embedded index is also easy to be adopted as the original basic operations (GET, PUT and DEL) on the primary key remain unchanged. The embedded index also simplifies the transaction management as we do not have to synchronize separate data structures.

We implemented the proposed indexing techniques on top of LevelDB [4], which to date does not offer secondary indexing support. Over other candidates such as RocksDB [8] we chose LevelDB because it is a single-threaded pure single-node key value store, so we can more easily isolate and explain the performance differences of the various indexing methods.

We make following contributions in this paper:

- We propose a novel lightweight embedded indexing technique to support secondary attribute lookups in LSM-based key-value stores (Section 3).
- We propose a variation of interval tree data structure to support efficient secondary attribute range lookups in embedded index(Section 3).
- We propose and implement lazy and eager update variants of stand-alone indexing on LSM-based key-value stores, inspired by state-of-the-art NoSQL and relational databases (Section 4).
- We create and publish an open-source realistic Twitter workload generator of a mixture of reads (on primary key), lookups

[1]Earlier versions of Cassandra handled this by first accessing the index table to retrieve the existing postings list of u1, then writing back a merged posting list to the index table. Then the old posting list is obsolete. Such "eager" updates degrade the write performance.

(on secondary attributes) and writes. We also publish open-source versions of our indexes' implementation on LevelDB (Section 5).

- We conduct extensive experiments to study the trade-offs between the indexing techniques on various workloads. We also theoretically study these trade-offs (Section 5).

The rest of the paper is organized as follows. Section 2 reviews the related work and background. We discuss on the lessons learned and also on transactional management issues in Section 6. We conclude and present future directions in Section 7. The workload generator and the source code of our indexes are available at [7].

2. RELATED WORK AND BACKGROUND

2.1 Secondary Index in NoSQL databases

Cassandra [20] is a column store NoSQL database where attributes can be defined in the schema but the schema is dynamic (record may contain new attributes not predefined in the schema). Cassandra supports secondary index. For this, Cassandra populates a separate table (column family in Cassandra's terminology) to store the secondary index for each indexed attribute, where the row key of an index table is a secondary attribute value, and each row contains a set of columns with primary keys as column names. Upon deletes and updates in data table, instead of deleting the out-of-date column names from index table, Cassandra uses a "read-repair" strategy where the invalidation of these column names is postponed to when a read on them occurs. Compared with Cassandra, pure key-value stores do not have schema or column support and thus pose more challenges in supporting secondary index. In [23], the authors show how to support secondary indexes on HBase. Index structures are not their focus, instead they aim to offer different levels of consistency between data tables and indexes upon the writes in data tables. In contrast, in this paper we compare different secondary index structures.

Document-oriented NoSQL databases like CouchDB [2] and MongoDB [6] support storing JSON-style documents. Similarly to RDBMs, they employ B-tree variation to support secondary indexes. In this paper our focus is on LSM-style storage.

HyperDex [16] proposes a distributed key-value store supporting search on secondary attributes. They partition the data across the cluster by taking the secondary attribute values into consideration. Each attribute is viewed as dimension (e.g., a tweet may have two dimension tweet id and user id), and each server takes charge of a "subspace" of entries (e.g., any tweet with tweet id in [tid1, tid2] and user id in [uid1, uid2]). Innesto [21] applies a similar idea of partition and adds ACID features in a distributed key-value store. It claims to perform better on high dimension data than HyperDex. In this paper, our focus is on a single-machine key-value store. The distribution techniques of HyperDex and Innesto can be viewed as complementary if we want to move to a distributed setting.

AsterixDB [10] introduces an interesting framework to convert an in-place update index like B+ tree and R-tree to an LSM-style index, in order to efficiently handle high throughput workloads. Again, this can be viewed as complementary to our work as we work directly on LSM-friendly indexes (set of keys with flat postings lists).

2.2 Background

2.2.1 LSM tree

An LSM tree generally consists of an in-memory component (level in LevelDB terminology) and several immutable on-disk components (levels). Each component consists of several data files and each data file consists of several data blocks. As depicted in Figure 2, all writes go to in-memory component (C0) first and then flush into the first disk component once the in-memory data is over the size limit of C0, and so on. The on disk components normally increase in size as shown in Figure 2 from C1 to CL. A background process (compaction) will periodically merge the smaller components to larger components as the data grows. Delete on LSM is achieved by writing a tombstone of the deleted entry to C0, which will later propagate to the component that stores this entry.

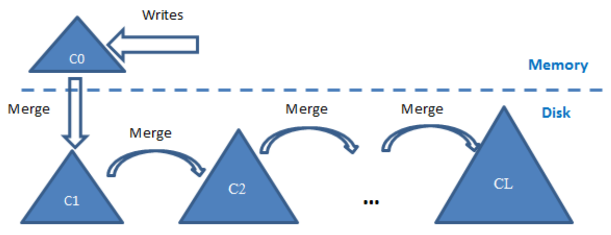


Figure 2: LSM tree components.

LSM is highly optimized for writes as a write only needs to update the in-memory component C0. The append-only style updates mean that an LSM tree could contain different versions of the same entry (different key-value pairs with the same key) in different components. A read (GET) on an LSM tree starts from C0 and then goes to C1, C2 and so on until the desired entry is found, which makes sure the newest (valid) version of an entry will be returned. Hence, reads are slower than writes. The older versions of an entry (either updated or deleted) are obsolete and will be discarded during the merge (compaction) process.

Because of LSM trees' good performance on handling high write throughput, they are commonly used in NoSQL databases such as BigTable [13], HBase [18], Cassandra [20] and LevelDB [4]. And conventionally, the in-memory component (C0) is referred as Memtable and the on disk components are referred as SSTable in these systems.

2.2.2 SSTable in LevelDB

As we will implement our indexing techniques on top of LevelDB, we present some storage details of LevelDB as background for ease of understanding. Level-$(i+1)$ is 10 times larger than level-i in LevelDB. Each level (component) consists of a set of disk files (SSTables), each having a size around 2 MBs. The SSTables in the same level may not contain the same key (except level-0[2]). Recent Cassandra versions support this level merging (compaction) option used in LevelDB.

Further, LevelDB partitions its SSTables into data blocks (normally tens of KB in size). The format of an SSTable file in LevelDB is shown in Figure 3. The key-value pairs are stored in the *data blocks* sorted by their key. Meta blocks contain meta information of data blocks. For example, Stats Meta Blocks contain information such as data size, number of entries and number of data blocks. Filter Meta blocks store a Bloom filter for each block, computed based on the keys of the entries in that block, to speedup GET operations. Then, there are index blocks for both meta blocks and data blocks, which store the addresses of meta blocks and data blocks. In the end of the SSTable file is a fixed length footer that

[2]which is C1 in Figure 2 as LevelDB does not number the memory component

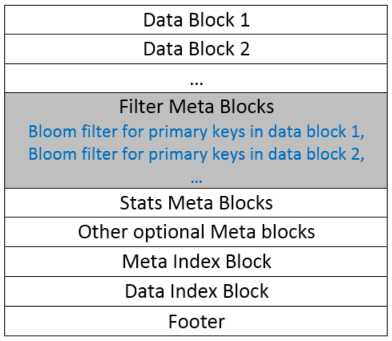

Data Block 1
Data Block 2
...
Filter Meta Blocks
Bloom filter for primary keys in data block 1,
Bloom filter for primary keys in data block 2,
...
Stats Meta Blocks
Other optional Meta blocks
Meta Index Block
Data Index Block
Footer

Figure 3: LevelDB SStable Structure.

contains the block address of the meta index and data index blocks [4].

2.2.3 Bloom Filter

Bloom filter [12] is a hashing technique to efficiently test if an item is included in a set of items. Each item in the set is mapped to several bit positions, which are set to 1, by a set of n hash functions. Then, the Bloom filter of a set is the OR-ing of all these bitstrings. To check the membership of an item in the set, we compute n bit positions using the same n hash functions and check if all corresponding bits of the Bloom filter are set to 1. If no, we return false, else we have to check if this items exists due to possible false positives. The false positive rate depends on the number of hash functions n, the number of items in the set S and the length of the bit arrays m, which can be approximately computed as Equation 1.

$$\left(1-\left[1-\frac{1}{m}\right]^{nS}\right)^{n} \approx \left(1-e^{-nS/m}\right)^{n} \tag{1}$$

Given S and m, the minimal false positive rate is $2^{-\frac{m}{S}ln2}$ by setting the $n = \frac{m}{S}ln2$.

3. LIGHTWEIGHT EMBEDDED INDEX IN LSM BASED KEY-VALUE STORE

Overview If there is no secondary index, then we perform a sequential scan to answer LOOKUP(A_i, a, K). The scanning would start from the in-memory component, and then move to disk components C1 to CL, until enough (K) matched entries are found. This is possible because the first (and smaller) components contain more recent data. Although this scan may avoid reading the lowest (and largest) components, it is still very costly because of the large amount of disk accesses. We use a B-tree for in-memory data (Memtable) and Bloom filters for on-disk data (SSTable) to dramatically speedup this scan. For the whole Memtable we create a B-tree for each indexed secondary attribute for fast match as it avoids to match toward every single key-value pair in Memtable. As Memtable is normally several MBs in total, thus the B-tree is small. As shown in Figure 1, in each SSTable file we build a set of Bloom filters on the secondary attributes, and append the bloom filters to the end of file. Specifically, a Bloom filter bit-string is stored for each secondary attribute for each block inside the SSTable file (recall that a SSTable file is partitioned into blocks). Note that these Bloom filters are loaded in memory (the same approach is used currently with the Bloom filters of the primary key in systems like Cassandra and LevelDB). Hence, the scan over the disk files in converted to a scan over in-memory Bloom filters. We only access the disk blocks which the Bloom filter returns as positive. We refer to this index as *Embedded Index* as no separate data files are created, but instead the Bloom filters are embedded inside the main data files.

LOOKUP using Embedded Index. The general LOOKUP procedure in a LSM based database is already explained above. However, we further clarify some important points here. Consider the sequence of operations *PUT(t1,{UserID: u1},..., PUT(t1,{UserID: u2}* followed by *LOOKUP(UserID,u1)*. When the LOOKUP is issued, it is possible that entry *(t1,u1)* exists in a lower component, say C2, while *(t1,u2)* exists in a higher component, say C1. Of course, eventually *(t1,u1)* will be deleted during a compaction process. Now, when the LOOKUP is executed, the Bloom filter of the block that contains *(t1,u1)* will be positive and hence $t1$ would be incorrectly returned. To avoid this, we issue a *GET(t1)* to check that $u1$ is returned, and only then output $t1$. Note that a similar strategy is needed as post-processing in the lazy stand-alone indexing strategy described in Section 4.2, as is also the case for state-of-the-art secondary stand-alone indexes on column-stores like Cassandra.

RANGELOOKUP using Embedded Index. To support the range queries on secondary attributes we maintain an interval tree based index structure. This index maintains an interval corresponding to the range of secondary attribute values for each data block in all the SSTables. It also stores a largest timestamp (sequence number) corresponding to each file block interval. Given a range query, this interval tree will return list of the blocks' pointer whose interval intersects with the given range and the list is sorted by the maximum timestamp value. The interval tree is explained with an example in Figure 4 where ranges (intervals) are constructed based on lexicographic ordering. If we issue an interval query $[n,o]$ toward the interval tree in Figure 4, it will return $[c,o]$, $[b,n]$, $[n,w]$, $[l,s]$ and $[g,n]$ in that order. Note that $[b,d]$ is not returned because it does not intersect with the range $[n,o]$.

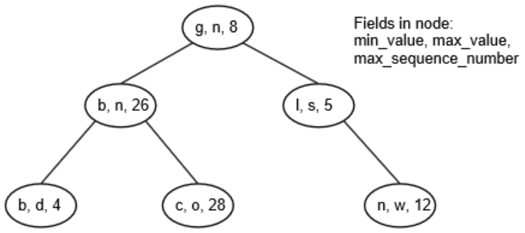


Figure 4: Interval tree example

We have also implemented a B-Tree which contains a inverted index list of secondary values for current memtable.

GET, PUT and DEL A key advantage of the embedded index is that only small changes are required for all three interfaces: GET(k), PUT(k,v), and DEL(k). Obviously, GET(k) needs no change as it is a read operation and we have only added Bloom filter and interval tree information to the data files. PUT and DEL don't have to worry about building the secondary attributes Bloom filters but just need to update the B-tree for the Memtable data.

Compaction However, we do need to change the flush process (in-memory component to first on disk component) and the merge (compaction) process (newer on disk component to older component). That is, every time an SSTable file is created, we compute the Bloom filter per block for each to-be-indexed attribute. Note that this operation is currently done for the primary key, but we extend it to the secondary attributes. We also added the intervals (range of the secondary attribute values) for the newly created SSTable blocks into the interval tree. Because SSTable is immutable, after its creation Bloom filter and intervals remain unchanged as long as the SSTable is valid (or else will be discarded).

To make the interval tree used for range queries consistent with the database, we modified the compaction process. Compaction merges a number of SSTable files to create a new sets of SSTable files. So for each compaction process, we delete all the blocks of the old SSTable files and add all the blocks of the newly created SSTable files. These operations in the compaction process can be summarized in Figure 5.

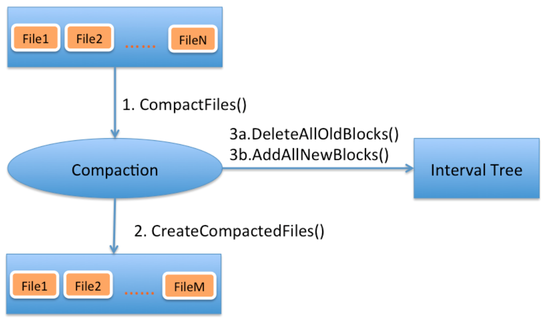


Figure 5: Interval Tree maintained in the compaction

To summarize, the embedded index dramatically accelerates LOOKUP queries, while incurring minimal cost on writes and reads on the primary key. Our experiments in Section 5 confirm the effectiveness of this lightweight indexing technique.

3.1 Embedded Index Implementation in LevelDB

LevelDB, among NoSQL databases like Cassandra, already employs Bloom filters to accelerate reads on primary keys. Several types of *meta blocks* are already supported in LevelDB and we highlight the *filter meta block*, which stores a Bloom filter on the primary keys for each data block. As LevelDB already natively supports filter meta blocks, we just have to modify it to add secondary attribute Bloom filters as shown in Figure 6.

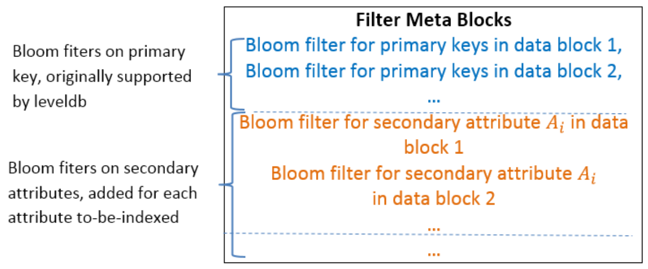

Filter Meta Blocks
Bloom fiters on primary key, originally supported by leveldb
Bloom filter for primary keys in data block 1,
Bloom filter for primary keys in data block 2,
...
Bloom fiters on secondary attributes, added for each attribute to-be-indexed
Bloom filter for secondary attribute A_i in data block 1
Bloom filter for secondary attribute A_i in data block 2
...
...

Figure 6: Filter Meta Block change on LevelDB to support secondary attribute index.

Support top-K retrieval: LOOKUP(A_i, a, K)

We mentioned above that when performing LOOKUP, we scan one level at a time until K results have been found. A key challenge is that within the same level there may be several entries with the queried secondary attribute value $val(A_i) = a$, and these entries are not stored ordered by time (but by primary key). Fortunately, LevelDB, as most other NoSQL systems, assigns an auto-increment sequence number to each entry at insertion, which we use to perform time ordering within a level. Hence, we must always scan until the end of level, before termination.

To efficiently compute the top-K entries, we maintain a min-heap ordered by the sequence number. If we find a match (i.e. we find a record that has secondary attribute value $val(A_i) = a$, then if the heap size is equal to K, we first check whether it is an older record than the root of the min heap. If it is a newer match or the heap size is less than K, then we check whether it is a valid record (i.e. whether it is deleted or updated by a new record in data table). We check this by calling a $GET(k)$ for this record in the data table. If it is not a deleted record, we check the secondary attribute value whether it matches with the entry that we found for our result.

We also maintain a hashset of K keys representing the already fetched results to make sure no duplicate entry is present in the results. If a duplicate entry comes with a greater sequence number, we update the min-heap with this newer entry and delete the old duplicate. Once the scan finishes a whole level with the heap containing K entries, it can safely stop, perform heapsort and return the results in the heap as decreasing order of the sequence number of each entry as the LOOKUP query results. We present the pseudocode of LOOKUP in LevelDB implementation in the Appendix.

Support top-K retrieval: RANGELOOKUP(A_i, a,b, K)

Similar to LOOKUP, to efficiently compute the top-K entries, we also maintain a min-heap ordered by the sequence number. So we first search in the in-memory B-Tree for all the records for the given secondary attribute value range $[a, b]$. If we find the top-K here, we stop and return the results. If not, then we search in the interval tree to find all the blocks that have ranges of secondary attribute values intersected with given range. We iterate through the blocks one by one in sorted order of maximum timestamp, and for each block we linearly search for any records which have a secondary attribute value in the range $[a, b]$. We keep pushing any successful matched records in our min-heap. We stops when next block in the iterator has a smaller maximum timestamp value than the timestamp value of the min record in the min-heap.

3.2 Analysis of Cost for Operations with Embedded Index

Overhead on GET/PUT/DEL queries. As discussed above, the embedded index does not incur much overhead on these operations. It only adds very small overhead on the compaction process to build a Bloom filter for each indexed attribute per block (if RANGELOOKUP is supported then there is also small overhead on maintaining the interval tree) and small overhead on updating the B-tree for Memtable.

LOOKUP query. A Bloom filter is a probabilistic data structure, i.e., it returns false positives with a probability. This probability is related to the bits array size and the number of entries a bloom filter already mapped. We study more details in experiments. Here we give a general theoretical analysis on the number of SSTable IO during answering LOOKUP query using embedded index.

In LevelDB a level has 10 times the number of SSTable files (and thus blocks) than its upper level. As a consequence, the number of IOs grows exponentially when scanning lower levels. Thus a smaller K in LOOKUP query is faster, because of the higher chance that the LOOKUP query can be satisfied in the first few levels.

Assume the number of blocks in level-0 is b (and thus level-i is approximately $b \cdot 10^i$). As discussed in Section 2.2 (see Equation 1), the minimal false positive rate of the bloom filter is $2^{-\frac{m}{s} ln2}$, denoted as fp. Thus the approximate number of block accesses on level-i for LOOKUP due to false positives is $fp \cdot b \cdot 10^i$. Hence, if a LOOKUP query needs to search on d levels, the expected block accesses are $\sum_{i=0}^{d}(fp \cdot b \cdot 10^i) = \frac{fp \cdot b \cdot (10^{d+1}-1)}{9}$. If matched entries are found in H blocks, the total number of block accesses is $H +$

$\frac{fp\cdot b\cdot(10^{d+1}-1)}{9}$. Note that H may be greater than K as we have to go to the end of the level for finding most recent K records.

The number of table accesses for different operations in Embedded Index are presented in Table 2.

Table 2: Table accesses for operations in Embedded Index. Assume the LOOKUP query needs to search in d levels of SSTables.

Action	Read Data	Write Data
GET(k)	1	0
DEL(k)	0	1
PUT(k, v)	0	1
LOOKUP(A_i, a, K)	$H + \frac{fp\cdot b\cdot(10^{d+1}-1)}{9}$	0

The embedded index will achieve better LOOKUP query performance on low cardinality data (i.e., an attribute has small number of unique attribute values), because fewer levels must be accessed to retrieve K matches.

4. STAND-ALONE INDEX IN LSM BASED KEY-VALUE STORE

A stand-alone secondary index in a key-value store can be logically created as follows. For each indexed attribute A_i, we build an index table T_i, which stores the mapping from each attribute value to a list of primary keys, similarly to an inverted index in Information Retrieval. That is, for each key-value pair $\langle k, v\rangle$ in data table with $val(A_i)$ not null, k is added to the postings list of $val(A_i)$ in T_i. LOOKUPs can then be supported straightforwardly by this stand-alone index.

Compared with embedded index, stand-alone index could have overall better LOOKUP performance, because it can get all candidate primary keys with one read in index table in eager updates (or up to L reads when there are L levels in the case of lazy updates as we discuss below). However, stand-alone index is more costly on the maintenance of index table upon writes (PUT(k, v) and DEL(k)) in data table in order to keep the consistency between data and index tables.

Write Challenges. Recall that PUT($k, \{A_i : a_i\}$) may be an insert or update. We assume a single secondary attribute A_i for presentation simplicity. Given this PUT, to maintain the consistency between T_i (index table for attribute A_i) and data table, two steps need to be taken:

1. If the PUT is an update, that is, there was previously a PUT($k, \{A_i : a'_i\}$), then we must remove k from the postings list of a'_i.

2. Add k to the postings list for a_i in T_i.

Note that for DEL(k) operations only the first step must be taken.

The first step requires issuing GET(k) on data table to get a'_i if it exists. Since the reads on LSM trees are much slower than writes (if not considering cache), this could significantly slow down writes. To avoid this, we can apply three options to postpone the removal of k from a'_i's list in index table T_i. One simple option is that we do not handle this step at all. This one should be used for workloads with very low ratio updates/deletes. For an append-only workload (i.e., no delete or update on existing entries), this strategy performs best. The second one is to do the cleanup of invalid keys during compaction on index table. That is, when a compaction merge an old file to a new file, it checks the validation of each key in the old file and decides whether writes a key to the new file. This process could make the compaction process more IO intensive and thus affects the overall performance of the database. Another option is to do periodic cleanup on SSTables when the system is not busy. In our experiments we implement the first option.

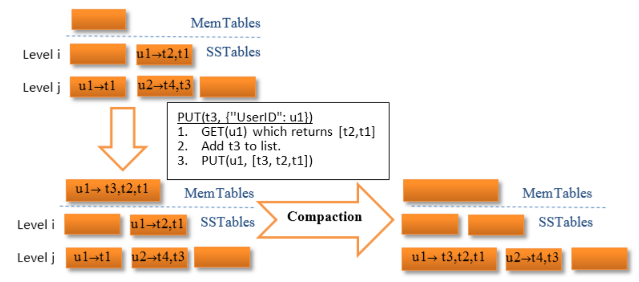


Figure 7: Example of stand-alone index with eager updates and its compaction.

For the second step, we present two options to maintain the consistency between data and index tables, eager updates and lazy updates, discussed in Sections 4.1 and 4.2 respectively. We have implemented both options on LevelDB.

4.1 Stand-alone Index with Eager Updates

Eager Updates (PUT) Upon PUT($k, \{A_i : a_i\}$), a stand-alone index with eager updates first reads the current postings list of a_i from index table, adds k to the list and writes back the updated list. *This means that, in contrast to the lazy update, only the highest-level posting list of a_i needs to be retrieved, as all the lower ones are obsolete.*

Example 1. *Assume the current status of a database shown in Table 3. Now suppose PUT(t3, {"UserID": u1, "Text": "t1 text"}) is issued, which updates the UserID attribute of t3 from u2 to u1. Figure 7 depicts a possible initial and final instance of the LSM tree.*

Table 3: TweetsData table storing tweets data.

Key	**Value**
t1	{"UserID": u1, "text": "t1 text"}
t2	{"UserID": u1, "text": "t2 text"}
t3	{"UserID": u2, "text": "t3 text"}
t4	{"UserID": u2, "text": "t4 text"}

Note that the compaction on the index tables works the same as the compaction on the data table.

LOOKUP A key advantage of eager updates is that LOOKUP(A_i, a) only needs to read the postings list of a in the highest level in T_i. However, there could be invalid keys in the postings list of a. Thus, similarly to the embedded index LOOKUP post-processing, for each returned key k, we need to call a GET(k) on data table and make sure $val(A_i) = a$. To support top-K LOOKUPs, we maintain the postings lists ordered by time (sequence number) and then only read a K prefix of them. The pseudocode of LOOKUP(A_i, a, K) is shown in the appendix.

4.2 Stand-alone Index with Lazy Updates

Lazy updates (PUT) The eager update still requires a read operation to update the index table for each PUT, which significantly slows down PUTs. In contrast, the lazy update works as follows. Upon PUT($k, \{A_i : a_i\}$) (on the data table), it just issues a PUT($a_i, [k]$) to the index table T_i but nothing else. Thus the postings list for a_i will be scattered in different levels. Figure 8 depicts the update of index table by the lazy updates strategy upon the PUT in Example 1.

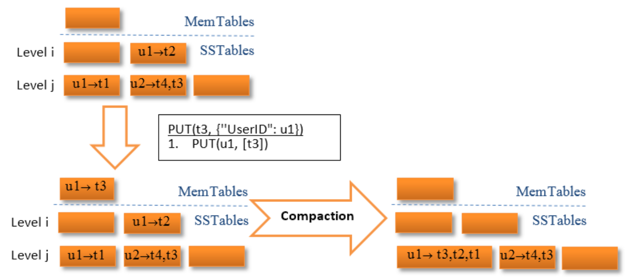


Figure 8: Example of stand-alone index with lazy updates and its compaction.

Support top-K query: LOOKUP(A_i, a, K) Since the postings list for a could be scattered in different levels, a query LOOK(A_i, a, K) needs to merge them to get a complete list. For this, it checks the Memtable and then the SSTables, and moves down in the storage hierarchy one level at a time. Note that at most one postings list exists per level. The pseudocode of LOOKUP is shown in the appendix.

Support top-K range query: RANGELOOKUP(A_i, a, b, K) To support range query RANGELOOKUP(A_i, a, b, K) on secondary attribute in Stand-alone index, we used range query API in LevelDB on primary key. We issue this range query on our index table for given range $[a,b]$. For each match x, we issue a LOOKUP(A_i, x, K) on index table. Each LOOKUP will return a list of results with size no larger than K for x which is sorted by time. To return a top-K among all the records in range $[a,b]$, we need to merge all the results and sorted them by timestamps. For that purpose, we again changed in the compaction and stored the sequence number of each entry in the postings list. To maintain the top-k efficiently we used a min-heap of size K similar to our LOOKUP implementation in Embedded Index. So we push each record in the postings list to this min-heap and we keep continue until we find all the matches in the index table in this range.

4.3 Analysis of Cost for Operations on Stand-alone Index

Overhead on GET/PUT/DEL queries Assume we have an append-only workload, and hence no maintenance on index table is needed and thus incurs no overhead for GET and DEL query on data table. However for a PUT query on data table, a Read-Update-Write process is issued on the index table by the eager update variant, in contrast to only one write on the index table in the lazy updates variant. Table 4 shows the number of table accesses for these three queries.

LOOKUP For eager updates, only one read[3] on the index table is needed to retrieve the postings list, as all lower level lists are invalid. Then, for each key in the top-K list, it issues a GET query on the data table. Hence, it takes a total of $K+1$ disk accesses for LOOKUP(A_i, a, K) if there exist K matched entries. In contrast for lazy updates, a read on the index table may involves up to L (number of levels) disk accesses (because in worst case the postings list is scattered across all levels). Thus the total cost of LOOKUP(A_i, a, K) is $(K+L)$ disk accesses. Note that this cost would be higher if we also had updates in the workload since we would have to check for invalid entries.

5. EXPERIMENTS

[3]assuming there is no false positive by the Bloom filter on the primary key in the index table

5.1 Experimental Setting

We run our experiments on machines with Quad Core Intel(R) Xeon(R) E3-1230 v2@3.30GHz CPU and 16GB RAM. The operating system used in these machines is CentOS version 6.4.

We collected 18 millions of tweets with overall size 9.88 GB (in JSON format) which have been posted and geotagged within New York State. Then we ran experiment using different benchmark files having different read write ratio and different read lookup ratio. We selected TweetID as the primary key attribute and UserID as the secondary attribute for indexing.

5.2 Benchmark

We are aware of several public workloads for key-value store databases available online such as YCSB [14]. However, as far as we know there are no workload generator which allows fine-grained control of the ratio of queries on primary to secondary attributes. Thus, to evaluate our secondary indexing performance, we created a realistic Twitter-based workload generator. The generator works on a sample of tweets e.g., 1% sample collected through Twitter Streaming API. Each tweet is a PUT on the database. A user can specify a *read-to-write ratio* to generate benchmarks with desired Reads/Writes ratio. The *primary-to-secondary-reads ratio* parameter allows the user to adjust the ratio of primary key reads (GET) to secondary index reads (LOOKUP). In our experiments, we set this parameter to 100. When generating the benchmark file, one read (primary key read and secondary attribute lookup) is generated based on the probability of each candidates in the most recent writes. We have one another integer parameter *read buffer size* for this: we maintain the frequency of each primary key / secondary attribute values in the most recent *read buffer size* number of writes in order to probabilistically choose when candidates to read. In our experiments we set it to 50K.

5.3 Experimental Evaluation

In the section we evaluate our secondary indexing techniques. Because the support of RANGELOOKUP requires to maintain extra data structures which incurs overhead, thus we report the performance of versions with and without RANGELOOKUP supported separately. We first report the performance the indexing variants (Stand-alone Index with Eager Updates, Stand-alone Index with Lazy Updates, Embedded Index) without RANGELOOKUP supported in Section 5.3.1. We study factors that could affect the performance in Sections 5.3.2, 5.3.4 and 5.3.3. We show the space efficiency for the indexing variants in Section 5.3.5. We present the performance of variants with RANGELOOKUP supported in Section 5.3.6.

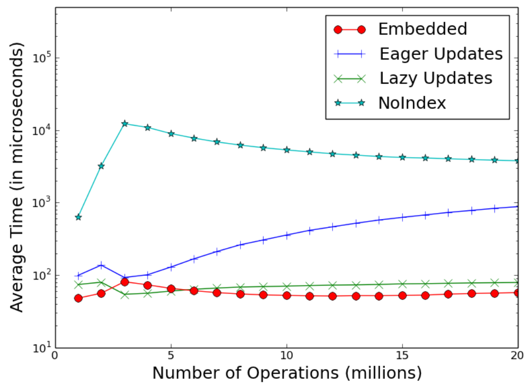
5.3.1 Performance of indexing methods without the support of RANGELOOKUP

We conduct our experiments on workloads with different reads-to-writes ratios which represent different use cases. Figures 9 and 10 show the overall, GET, PUT, and LOOKUP performance of these indexing methods on write heavy ($\frac{Reads}{Writes} = \frac{1}{9}$) and read heavy workloads ($\frac{Reads}{Writes} = 9$), respectively. Note here reads include GET and LOOKUP queries. We also measure the performance when no secondary index is built, in which we scan the database sequentially to find the matched entries to for answering LOOKUP queries.

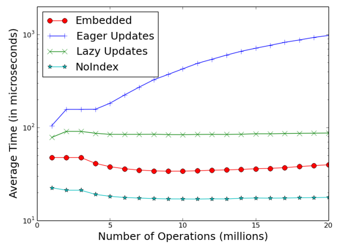
On both workloads, we set primary-to-secondary-ratio to 100 and used a fixed *bits per key* $= 100$ for Bloom filter. Note that both benchmarks both contain 20 millions operations. Shown in Figures 9 and 10, we record the performance once per 1 millions operation top-$K = 5$ in LOOKUP queries.

Table 4: Table accesses for operations. Assume l attributes are indexed on the primary table.

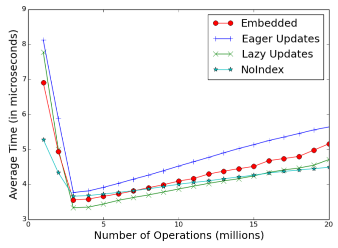
Action	Read Data	Write Data	Read Index	Write Index
GET(k)	1	0	0	0
DEL(k)	0	1	0	l
PUT(k, v)	0	1	l (eager), 0 (lazy)	l
LOOKUP(A_i, a, K)	K	0	1 (eager), L (lazy)	0



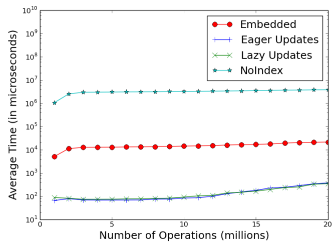
(a) Overall performance



(b) PUT performance

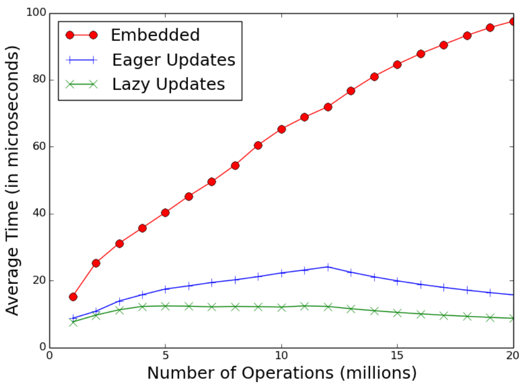


(c) GET performance

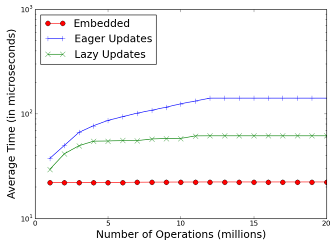


(d) LOOKUP performance (Top-$K = 5$)

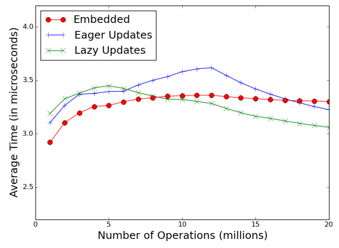
Figure 9: Performance of different indexing variants on write heavy benchmark.



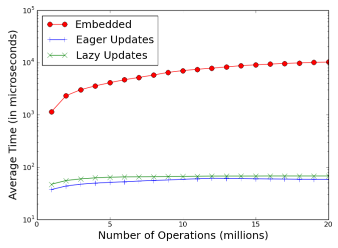
(a) Overall performance



(b) PUT performance



(c) GET performance



(d) LOOKUP performance (Top-$K = 5$)

Figure 10: Performance of different indexing variants on read heavy benchmark.

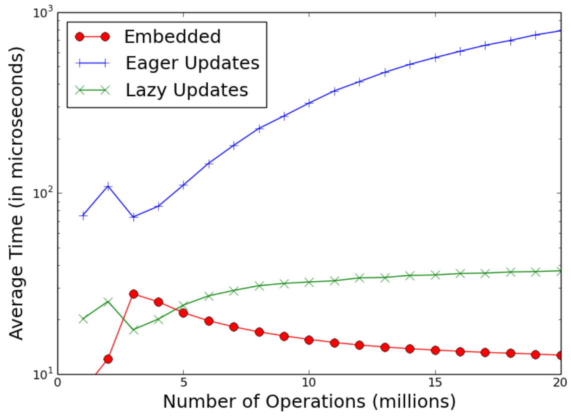
It shows in Figure 9 that Stand-alone Index with Eager Updates cannot scale for large dataset. Embedded Index achieves very similar GET performance as Stand-alone Index with Lazy Updates but give much better PUT performance. Thus on the Write Heavy benchmark, Embedded Index gives better overall performance than Stand-alone indexes (around 40% better than Lazy Updates)..

However, on the Read Heavy workload, the Lazy Updates version gives the best overall performance among these indexing techniques as shown in Figure 10.

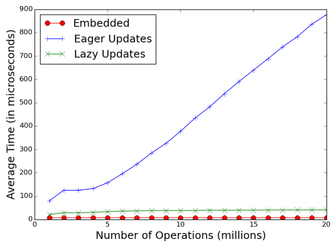
5.3.2 *Performance of indexing methods varying key-value pair size*

We study the effect of the key-value pair size on the performance of these indexing techniques. For that, we only keep two attributes (user id and timestamp) of each tweet in the value. The data contains the same number of tweets but the total size is reduced to 1.24GB in JSON format (vs. 9.88 GB without removing attributes). We generate a new workload using the same parameters as we used in former experiments. We present the performance results on the write heavy workload ($\frac{Reads}{Writes} = \frac{1}{9}$) in Figure 11.

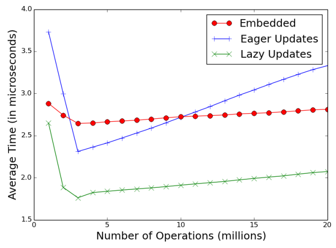
We observe that the performance of all variants improve with smaller key-value pairs compared with the results shown in Figure 9. However, we see the advantage of Embedded Index over Lazy Updates in terms of overall performance is larger. Because each data block contains relatively more key-value pairs, Embedded Index has higher chance to check smaller number of SSTables files and levels to satisfy a LOOKUP query.



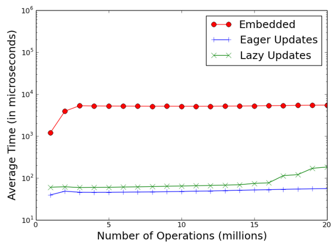
(a) Overall performance



(b) PUT performance



(c) GET performance



(d) LOOKUP performance (Top-$K = 5$)

Figure 11: Performance for different indexing variants on write heavy benchmark with smaller key-value pairs.

5.3.3 LOOKUP Performance of Embedded Index varying top-Ks

In this section, we specifically study the LOOKUP performance with different Ks on Embedded Index with all other database settings unchanged. Figure 12 shows the average running time of LOOKUP queries among the 20 millions of operations in the Write Heavy workload. We can see that as K increases, the LOOKUP performance decreases as expected.

5.3.4 LOOKUP Performance of Embedded Index varying Bloom filter lengths

The length of each Bloom filter used in Embedded Index also has affect on LOOKUP performance in two folds: (1) with larger Bloom filter, the false positive rate will drop thus the number of blocks to be loaded into memory is smaller, and (2) the cost of checking each Bloom filter is higher because more hash functions are applied for each key to check its existence as the Bloom filter length increases.

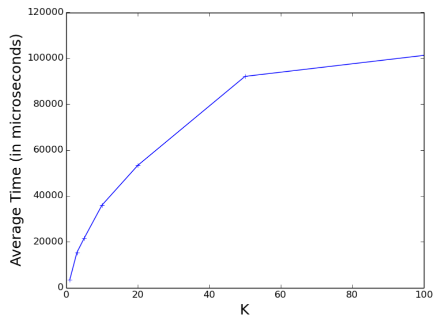


Figure 12: LOOKUP performance with different top-K on the write heavy benchmark.

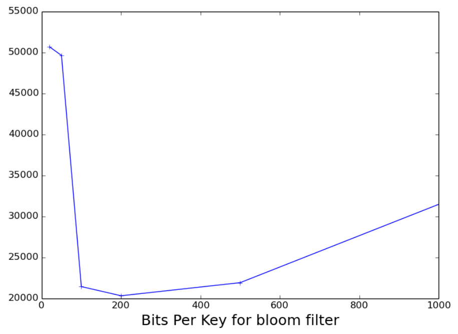


Figure 13: LOOKUP performance with different bloom filter setting (bits per key) on the write heavy benchmark.

We conducted the experiment by varying the value of *bits per key* from 20 to 1000. Figure 13 shows the average performance of LOOKUPs in the Write Heavy workload. Here we can see that the LOOKUP performance starts to increase with increasing bits per key setting of Bloom filter (20 to 200) as the false positive rate decreases. But then the performance decreases with larger value of bits per key where the false positive rate is low enough but the cost of checking Bloom filters increases.

Table 5: Averaged false positive rates of LOOKUP queries and final database sizes running Write Heavy workload under different settings of bits per key for Bloom filter.

Bits Per Key	Database Size	False Positive Rates
20	10,127 MB	0.64%
50	10,256 MB	0.44%
100	10,470 MB	0.08%
200	10,899 MB	0.08%
500	12,186 MB	0.08%
1000	14,325 MB	0.08%

We also show the space tradeoffs in Table 5. As the bits per key increases, the bloom filter takes more space. In our experiment we set *bits per key* as 100, because the LOOKUP performance is very close to setting it as 200 to 300 but with reasonable space overhead.

5.3.5 Space efficiency of indexing methods

Table 6: Database size of different indexing techniques on the two workloads. Small Write Heavy denotes the workload in which each value only contains two attributes. The numbers in parenthesis are the overhead over LevelDB with no secondary index built.

Implementation	Read Heavy	Write Heavy	Small Write Heavy
LevelDB (no secondary index)	1,569 MB	10,249 MB	1,445 MB
Eager Updates	1,615 MB (2.9%)	10,886 MB (6.2%)	2,102 MB (45.4%)
Lazy Updates	1,608 MB (2.5%)	10,610 MB (3.5%)	1,825 MB (26.3%)
Embedded	1,590 MB (1.3%)	10,470 MB (2.2%)	1,659 MB (14.8%)

Table 6 shows the database sizes of different indexing techniques on the workloads. It shows that Embedded Index is more space efficient than Stand-alone indexing techniques. Note that for each indexing variant, its absolute values of the overhead on Small Write Heavy and the overhead on Write Heavy are very close. For instance, the overhead of space in Embedded Index is mainly from Bloom filters which is decided by the number of entries and bits per key setting. Thus as Small Write Heavy and Write Heavy have same number of key-value pairs thus the total space of Bloom filter is unchanged.

5.3.6 *Performance of indexing methods with the support of RANGELOOKUP*

Through former experiments, we see Eager Updates will not scale for large dataset. Thus, we only implement RANGELOOKUP support on Stand-alone Index with Lazy Updates and Embedded Index.

Because LSM components are time correlated, thus Embedded Index combined with interval tree is efficient for RANGELOOKUPs only when secondary attribute is correlated with time. To test the performance with RANGELOOKUP, we build the secondary index on the CreationTime attribute instead of UserID.

Figure 14 compares the performance of Lazy Updates and Embedded Index on the write heavy workload. It shows that the Embedded Index achieve very close overall performance than Lazy Updates in this workload. In general, Embedded Index gives better RANGELOOKUP and PUT performance while Lazy Updates advantages in GET performance.

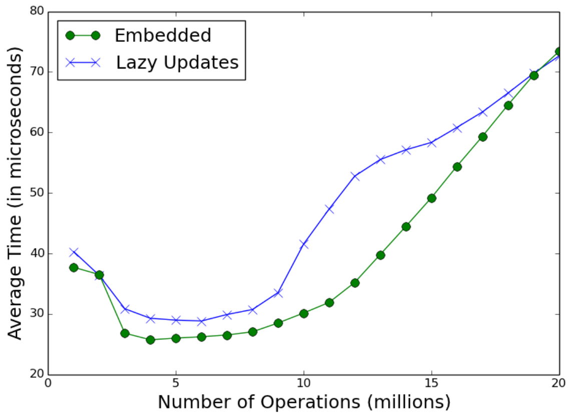
5.3.7 *Effect for multiple secondary attribute indexes*

We have shown experiments on one secondary index. From these results, we can intuitively discuss the performance if we have more indexes on other secondary attributes. Overall performance of Embedded Index will remain similar as we have seen there is very small overhead on PUT and GET by adding bloom filters for secondary attributes. But for Stand-alone indexes, PUT latency will increase linearly with the number of secondary indexes, because we have to write to different index tables every time we issue a PUT in the primary data table. The database size will also increase with the number of indexes for all of them, but we have already seen that Embedded Index is more space efficient than the Stand-alone indexes.

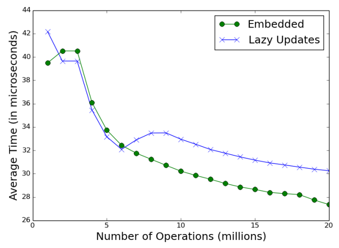
6. DISCUSSION

Lessons learned. The experimental results show the trade-offs between the three indexing strategies. As expected, no solution is best for all workloads.

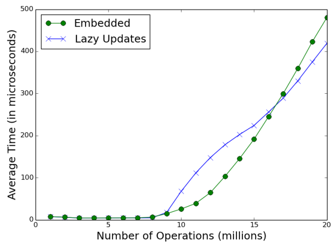
Since the Embedded Index only incurs very small space overhead compared to the other two, it is a better choice in the cases when space is big concern, e.g., to create a local key-value store on a mobile device. When space consumption is not a major concern, the Embedded Index is still favorable when the query workload contains relatively small ($< 5\%$) ratio of LOOKUP queries over GETs but very heavy writes throughput ($> 50\%$ of all operations).



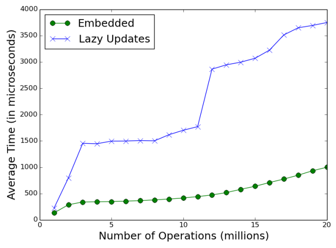
(a) Overall performance



(b) PUT performance



(c) GET performance



(d) RANGELOOKUP performance (Top-$K = 5$)

Figure 14: Performance for the two RANGELOOKUP supported indexing variants on write heavy benchmark.

For instance, in wireless sensor network where a sensor generates data of form (measurement id, temperature, humidity), the database must support some basic querying capabilities in order for the network to support queries [19, 15]. E.g., A monitor dashboard can be used to query the data from certain sensor or lookup the record with temperature in a range (if the sensor is recording temperature). Hence, in these applications reads and lookups are rare but writes have very high rate.

A limitation of Embedded is that it does not support RANGELOOKUP. We facilitate this by extending RANGELOOKUP with interval tree.

Table 7: Use cases for different indexing strategies.

Example Applications	*Primary Key*	*Secondary Keys*	*Suggested Index*
Wireless Sensor Network	Record ID	Sensor ID	Embedded Index
Facebook or Twitter	Post ID	User ID	Stand-alone Index with Lazy Updates
Snapchat or Whatsapp	Message ID	User ID	Stand-alone Index with Eager Updates

This is efficient for RANGELOOKUP operations when the secondary key is somewhat correlated with the primary key (or by time if records were stored by time in SSTables), but inefficient for uncorrelated attributes.

In other cases, a Stand-alone index is a better choice. The Lazy Updates variant adds more complexity on the cleanup of updated keys but achieves overall better write performance than Eager Updates. Thus, the Lazy Updates variant is preferable over Eager Updates when the workload involves very few updates. For instances, it is reported that there are much more reads than writes in Facebook and Twitter [5, 1].

Otherwise when the workload involves high ratio of updates in value or deletes, the Eager Updates version is a better choice. In the big data era, due to the cheap disk lots of applications do not remove data. However, we can see popular services with high ratio of deletes in database storage. For instance, applications such as Snapchat and WhatsApp have read heavy workloads like Facebook and Twitter. However they involves more deletes. E.g., messages in Snapchat and WhatsApp will be deleted from the server when messages expire [3, 9]. Thus in their databases they should have high ratio of deletes, which makes Eager Updates more favorable than Lazy Updates.

Example use cases for the three index methods are summarized in Table 7. In the work of [11], the authors collected traces from Facebook's Memcached (a popular in-memory key-value store) deployment which give more examples of practical workloads with high ratio of reads, writes or deletes.

These conclusions are summarized in Figure 15.

Transaction Support. Transaction are commonly supported in all kinds of RDBMS. However, NoSQL systems contain limited support for it. For instance, a recent Cassandra release started to support a "light-weight" transaction which supports the get-update-write process of a single data entry as an atomic operation. Similar functionality is also offered by MongoDB, while systems like HyperDex and AsterixDB offer stonger supports on transactions.

LevelDB implements GET, PUT and DEL as atomic operations, and also supports transactional operation of several writes on a single table, i.e., a write batch consists of several PUT and DEL. As we already explained, the embedded index just writes more filter blocks during PUTs to LevelDB, and hence does not break the atomicity of these basic operations. However, in stand-alone index implementations, PUT and DEL operations introduces multiple reads/writes across index and data tables. To keep them atomic, special transaction management is needed, which we leave as future work.

7. CONCLUSIONS

In this paper we study three secondary indexing methods for pure key-value stores, namely Stand-alone Index with Eager Updates, Stand-alone Index with Lazy Updates and Embedded Index. We propose a new version of interval tree combined with Embedded Index to support range queries on secondary attribute.

We implement these indexing methods over LevelDB and conduct extensive experiments to examine their performance. The experimental results show the tradeoffs between the three indexing techniques. We argue the appropriate choice of indexing techniques for different workloads and applications.

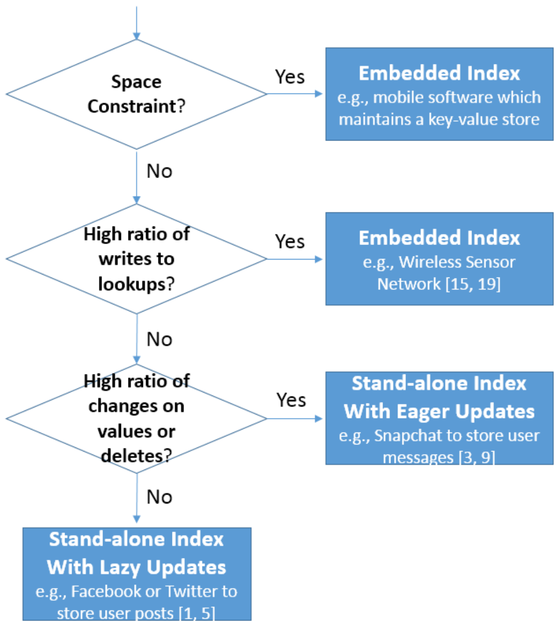

Space Constraint?
Yes
Embedded Index
e.g., mobile software which maintains a key-value store
No
High ratio of writes to lookups?
Yes
Embedded Index
e.g., Wireless Sensor Network [15, 19]
No
High ratio of changes on values or deletes?
Yes
Stand-alone Index With Eager Updates
e.g., Snapchat to store user messages [3, 9]
No
Stand-alone Index With Lazy Updates
e.g., Facebook or Twitter to store user posts [1, 5]

Figure 15: Overall process to decide which secondary index strategy to use according to query and data workload.

8. ACKNOWLEDGMENTS

This project was partially supported by a Samsung GRO grant.

9. REFERENCES

[1] The architecture twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.
[2] Couchdb. http://couchdb.apache.org/.
[3] How snaps are stored and deleted. http://blog.snapchat.com/post/50060403002/how-snaps-are-stored-and-deleted.
[4] Leveldb. https://code.google.com/p/leveldb/.
[5] Live commenting: Behind the scenes. https://code.facebook.com/posts/557771457592035/live-commenting-behind-the-scenes/.
[6] Mongodb. http://www.mongodb.org/.
[7] Project website for open source code and workload generator. http://dblab.cs.ucr.edu/projects/KeyValueIndexes.
[8] Rocksdb. http://rocksdb.org/.

[9] Where do pictures and files we send using whatsapp end up? http://kidsandteensonline.com/2013/10/10/where-do-pictures-and-files-we-send-using-whatsapp-end-up/.

[10] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *Proceedings of the VLDB Endowment*, 7(10), 2014.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 26(2):4, 2008.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] D. De and L. Sang. Qos supported efficient clustered query processing in large collaboration of heterogeneous sensor networks. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*, pages 242–249. IEEE, 2009.

[16] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[17] A. Feinberg. Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[18] L. George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[19] Y. He, M. Li, and Y. Liu. Collaborative query processing among heterogeneous sensor networks. In *Proceedings of the 1st ACM International Workshop on Heterogeneous Sensor and Actor Networks*, HeterSanet '08, pages 25–30, New York, NY, USA, 2008. ACM.

[20] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[21] M. T. Najaran and N. C. Hutchinson. Innesto: A searchable key/value store for highly dimensional data. In *CloudCom*, pages 411–420. IEEE, 2013.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[23] W. Tan, S. Tata, Y. Tang, and L. Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.

APPENDIX

A. ALGORITHMS FOR LOOKUP IN EMBEDDED INDEX

Algorithm 1 LOOKUP Procedure using Embedded Index

```
1: Create Min-Heap H;
2: for table in Memtable do
3:     for $\langle k,v \rangle$ in table do
4:         if $val(A_i) == a$ then
5:             if $H.size() == K \wedge H.top.seq < \langle k,v \rangle.seq$ then
6:                 $H.pop()$;
7:                 $H.put(\langle k,v \rangle)$;
8:             else if $H.size() < K$ then
9:                 $H.put(\langle k,v \rangle)$;
10: if $H.size() == K$ then
11:     return List of $K$ pairs in H
12: for $v = 0 \to L$ do
13:     for $sstable_j$ in level-$v$ do
14:         for $block_k$ in $sstable_j$ do
15:             if $block_k.bloomfilter(A_i).contains(a) == F$ then
16:                 NEXT
17:             load $block_k$ of $sstable_j$ in memory;
18:             for $\langle k,v \rangle$ in $block_k$ do
19:                 if $val(A_i) == a \wedge \langle k,v \rangle$ is valid then
20:                     if $H.size() == K \wedge H.top.seq < \langle k,v \rangle.seq$ then
21:                         $H.pop()$;
22:                         $H.put(\langle k,v \rangle)$;
23:                     else if $H.size() < K$ then
24:                         $H.put(\langle k,v \rangle)$;
25:     if $H.size() == K$ then
26:         return List of $K$ pairs in H
27: return List of pairs in H
```

B. ALGORITHMS FOR LOOKUP IN STAND-ALONE INDEX WITH EAGER UPDATES

Algorithm 2 LOOKUP based on Eager Updates

```
1: $H \leftarrow ()$;
2: Primary key list $L \leftarrow$ return of GET(a) on index table $T_i$. // Suppose the
   order is maintained as recent key to older keys
3: for $k$ in $L$ do
4:     $\langle k,v \rangle \leftarrow$ return of GET($k$) on data table.
5:     if $val(A_i) == a$ then
6:         H.add($\langle k,v \rangle$)
7:     if $H.size() == K$ then
8:         BREAK
9: return $H$
```

C. ALGORITHM FOR LOOKUP IN STAND-ALONE INDEX WITH LAZY UPDATES

Algorithm 3 LOOKUP Procedure based on Lazy Updates

```
1: $H \leftarrow ()$
   // starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0
   SSTable.
2: for $j$ from 0 to $L$ do
3:     if $val(A_i)$ is not in $Cj$ then
4:         NEXT
5:     List of primary keys $P \leftarrow$ the value of $A_i(v)$ in $Cj$
6:     for $k$ in $P$ do
7:         $\langle k,v \rangle \leftarrow$ return of GET($k$) on data table.
8:         if $val(A_i) == a$ then
9:             H.add($\langle k,v \rangle$)
10:         if $H.size() == K$ then
11:             return $H$
12: return $H$
```