ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys

Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng

Abstract—Indexing key-value data on persistent storage is an important factor for NoSQL databases. Most key-value storage engines use tree-like structures for data indexing, but their performance and space overhead rapidly get worse as the key length becomes longer. This also affects the merge or compaction cost which is critical to the overall throughput. In this paper, we present ForestDB, a key-value storage engine for a single node of large-scale NoSQL databases. ForestDB uses a new hybrid indexing scheme called HB$^+$-trie, which is a disk-based trie-like structure combined with B$^+$-trees. It allows for efficient indexing and retrieval of arbitrary length string keys with relatively low disk accesses over tree-like structures, even though the keys are very long and randomly distributed in the key space. Our evaluation results show that ForestDB significantly outperforms the current key-value storage engine of Couchbase Server [1], LevelDB [2], and RocksDB [3], in terms of both the number of operations per second and the amount of disk writes per update operation.

Index Terms—Key-value storage system, NoSQL, index structures, B$^+$-tree

1 Introduction

For the past few years, application trends for data managing and indexing have been changing rapidly and dramatically. As we can observe in social networking services such as Facebook and Twitter, the number of concurrent users and the amount of data to be processed keep getting larger and larger, while the data itself is becoming progressively unstructured for flexible analysis by applications. Unfortunately, using relational databases are struggling to deal with these issues due to their scalability and strict data model. For this reason, many companies are using NoSQL technology [4] as a viable alternative to relational databases.

Although there are various NoSQL technologies for high-level sharding, data replication, and distributed caching, their back-end persistent storage modules are not much different from each other. Each single node in NoSQL typically uses a key-value store for indexing or retrieving data in a schema-less way, where the key and value are commonly variable-length strings.[1] Since key-value stores directly interact with persistent block devices such as hard disk drives (HDDs) and solid-state drives (SSDs), their throughput and latency dominate the overall performance of the entire system.

The throughput and latency of key-value storage is bounded by the storage access time, which is mainly affected by two factors: the number of accessed blocks per key-value operation and the block access pattern. The former is basically related to the characteristics and logical design of index structures, while the latter is determined by how the indexing-related data including a key-value pair is actually read from or written to the storage.

There are two popular index structures used for single-node key-value stores: B$^+$-tree [5] and Log-structured merge-tree (LSM-tree) [6]. B$^+$-tree is one of the most popular index structures and has been widely used in traditional databases, due to its ability to minimize the number of I/O operations. Modern key-value databases such as BerkeleyDB [7], Couchbase [1], InnoDB [8], and MongoDB [9] use B$^+$-trees for their back-end storage. In contrast, LSM-tree consists of a set of layered B$^+$-trees (or similar B$^+$-tree-like structures), aimed at improving the write performance by sacrificing the read performance compared to the original B$^+$-tree. Many recent systems such as LevelDB [2], RocksDB [3], SQLite4 [10], Cassandra [11], and BigTable [12] use LSM-tree or its variant for their key-value indexing.

Although those tree-like structures have been successful so far, they perform poorly when they use variable-length strings as their keys instead of fixed-size primitive types. As the key length gets longer, the fanout degree (i.e., the number of key-pointer pairs in a node) decreases if the node size is fixed, and accordingly the tree height should grow up to maintain the same capacity. On the other hand, if the size of each node is enlarged to retain the same fanout, the number of blocks to be read or written for each node access has to be increased proportionally. Unfortunately, since both the average number of disk accesses and the space occupied by the tree are directly influenced by the both tree height and node size, the overall indexing performance degrades as the key length becomes longer.

1. The value can be semi-structured data such as JSON document in some key-value stores, but still it can be treated as a string.

- *J.-S. Ahn and S. Maeng are with the Department of Computer Science, KAIST, Daejeon 305-701, Korea. E-mail: {jsahn, maeng}@camars.kaist.ac.kr.*
- *C. Seo, R. Mayuram, and R. Yaseen are with Couchbase Inc., Mountain View, CA. E-mail: {chiyoung, ravi, yaseen}@couchbase.com.*
- *J.-S. Kim is with the School of Information & Communication Engineering Sungkyunkwan University, Suwon, Korea. E-mail: jinsookim@skku.edu.*

Manuscript received 16 May 2014; revised 28 Apr. 2015; accepted 10 May 2015. Date of publication 19 May 2015; date of current version 10 Feb. 2016. Recommended for acceptance by N. Jha.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2435779

To address this issue, BerkeleyDB and LevelDB use a prefix compression technique, which is similar to Prefix B$^+$-tree [13] and front-coding [14]. However, this scheme is largely affected by the pattern of keys. If keys are randomly distributed on the key space, the remaining uncompressed parts of keys are still sufficiently long so that the benefit of the prefix compression becomes quite limited. We need to devise a more efficient way to index variable-length string keys.

Meanwhile, the block access pattern is another crucial factor in designing key-value stores. The order between blocks to be written and their address distribution can highly affect the I/O performance in both HDDs and SSDs, though the number of blocks to be transferred remains same. The basic update-in-place scheme provides superior read performance but tends to experience the worst case write latency, thus it is unacceptable for the recent write-intensive key-value workloads. For this reason, a lot of databases use an append-only or write-ahead logging (WAL) style to write blocks in a sequential order.

Such designs can achieve high write throughput but suffer from merge or compaction (i.e., garbage collection) overhead. The amount of this overhead is closely related to both the average number of block accesses per index operation and the overall space occupied by the index structure because a number of merging operations are performed during the compaction process. Hence, the overhead also gets even worse when long keys lessen the fanout of an index structure. Since these compaction mechanisms have to be triggered periodically and involve a large number of I/O operations, their overhead is critical to the overall performance of the system.

This paper proposes ForestDB, a per-node key-value storage system for the next-generation Couchbase Server. For indexing variable-length string keys in a space- and time-efficient manner even though the key length can be long and their distribution can be random, we suggest a novel disk-based index structure called *Hierarchical B$^+$-tree-based trie (HB$^+$-trie)*, which is used as a primary index for ForestDB. The logical structure of HB$^+$-trie is basically a variant of patricia trie [15], but it is optimized for reducing the number of block accesses on persistent storage using B$^+$-trees. To achieve high throughput for both read and write operations, and to support highly-concurrent accesses, index updates are written into storage in an append-only manner. This also eases on-disk structures to implement a multi-version concurrency control (MVCC) model [16].

We have implemented ForestDB as a stand-alone key-value store library. Our evaluation results using real datasets show that the average throughput of ForestDB is significantly higher than LevelDB, RocksDB, and Couchstore, the current key-value storage module for Couchbase Server, in terms of the number of operations per second.

The rest of the paper is organized as follows. Section 2 overviews B$^+$-trees, Prefix B$^+$-trees, LSM-trees, and the internal structures of the current Couchstore. Section 3 describes the overall design of ForestDB. Section 4 presents the evaluation results, and Section 5 describes the related work. Section 6 concludes the paper.

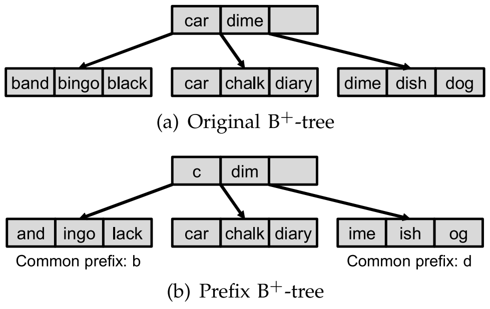


Fig. 1. Examples of B$^+$-tree and Prefix B$^+$-tree.

2 Background

2.1 B$^+$-Tree and Prefix B$^+$-Tree

B$^+$-tree is one of the most popular index structures for managing a large number of records on block devices. There are two different types of nodes in B$^+$-tree: leaf nodes and index (non-leaf) nodes. Unlike B-tree or other binary search trees, the key-value records are stored only in leaf nodes while index nodes contain pointers to their child nodes. There are more than two key-value (or key-pointer) pairs in a B$^+$-tree node; the number of key-value pairs in a node is called *fanout*. Generally the fanout of B$^+$-tree is configured to a large value in order to fit a single node into one or multiple blocks. In this way, B$^+$-tree can minimize the number of block I/Os per key-value retrieval.

However, the fanout is greatly influenced by the length of the keys as we previously mentioned. To moderate this overhead, Prefix B$^+$-tree [13] has been proposed. The main idea of Prefix B$^+$-tree is to store only distinguishable sub-strings instead of the entire keys so as to save space and increase the overall fanout. Index nodes accommodate only minimal prefixes of keys that can distinguish their child nodes, while leaf nodes skip the common prefixes among the keys stored in the same node.

Fig. 1 illustrates an example of B$^+$-tree and the corresponding Prefix B$^+$-tree. In Prefix B$^+$-tree, the root (and also index) node stores the string `c` instead of the entire key `car`, because it is a minimal sub-string that is lexicographically greater than `band`, `bingo`, and `black`, and equal to or less than `car`, `chalk`, and `diary`. However, we cannot use `d` instead of `dime` since `d` cannot be used as a pivot for the comparison of `diary` and `dime`. In this case, Prefix B$^+$-tree uses `dim`, which is a minimal sub-string of `dime` that can be lexicographically ordered between `diary` and `dime`.

In leaf nodes, `band`, `bingo`, and `black` in the leftmost node share the common prefix `b`, while `dime`, `diary`, and `dog` in the rightmost node share the common prefix `d`. Prefix B$^+$-tree keeps these common prefixes somewhere in the meta-section of each node, and only stores the portion of the strings excluding the common prefixes. This prefix compression scheme efficiently reduces the overall space if a large number of keys share a common prefix.

2.2 Log-Structured Merge-Tree (LSM-Tree)

Although B$^+$-tree can minimize the number of block accesses for indexing a given number of records, one may experience poor performance due to random block accesses. When B$^+$-tree is sufficiently aged, its nodes are randomly

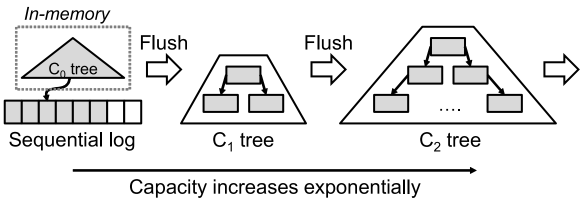


Fig. 2. The overview of LSM-tree.

scattered on a disk. This causes random block accesses for each tree traversal so that the overall performance greatly degrades. Note that the random read performance of other tree-like structures can hardly be better than B$^+$-tree due to its ability to minimize the number of block accesses, while we can still improve the write performance by arranging disk write patterns.

To improve the random write performance over B$^+$-tree, LSM-tree [6] has been suggested. A number of B$^+$-trees or B$^+$-tree-like structures hierarchically organize a single LSM-tree, as illustrated in Fig. 2. There is a small in-memory tree on top of LSM-tree, called C_0 tree. All incoming updates are appended into a sequential log, and each entry in the log is indexed by the C_0 tree for efficient retrieval. If the size of C_0 tree exceeds a certain threshold, a continuous range of entries in the log are merged into C_1 tree. In the same manner, a set of continuous entries are merged from C_1 to C_2, if the size of C_1 tree reaches a certain limit. Merge operations always occur between the C_i tree and C_{i+1} tree, and generally the capacity of the tree grows exponentially as i increases.

Since disk writes are performed sequentially for both appending into the log and merge operations, the write performance of LSM-tree is much better than that of the original B$^+$-tree. However, to retrieve a key-value pair, we have to traverse all trees starting from C_0 in a cascaded manner until the target record is found, and consequently the overall read performance of LSM-tree becomes inferior to that of B$^+$-tree. In order to avoid those unnecessary traversals, and to reduce the overall read amplification, LSM-tree commonly uses Bloom filters.

Note that the problem described in Section 1 also occurs in LSM-tree if the key length gets longer. Smaller fanout lessens the capacities of each tree so that merge operations between trees occur more frequently. Since merge operations involve a large number of disk I/Os, the overall performance degrades rapidly.

2.3 Couchstore

Couchstore is a per-node storage engine of Couchbase Server [1], whose overall architecture inherits from the storage model of Apache CouchDB [17], [18]. The key space is evenly divided into the user-defined number of key ranges, called vBuckets (or partitions), and each vBucket has its own DB file. Each DB file stores key-value pairs belonging to the vBucket, where a key is a string with an arbitrary length and a value is a JSON document. To retrieve the location of a document in the file, there is a B$^+$-tree for each vBucket to store the tuples of the key and the byte offset where the corresponding document is written. Hence, every single DB file contains both

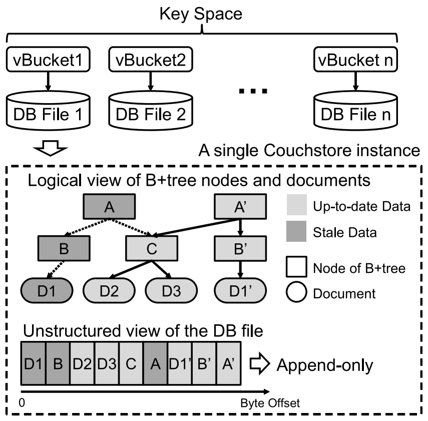


Fig. 3. The overview of Couchstore.

documents and B$^+$-tree nodes, which are interleaved with each other in the file.

Note that all updates in Couchstore are appended at the end of the DB file as illustrated in Fig. 3. `A`, `B`, and `C` denote B$^+$-tree nodes, while `D1`, `D2`, and `D3` in rounded boxes represent documents. If the document `D1` is updated, then the new document `D1'` is written at the end of the file without erasing or modifying the original document `D1`. Since the location of the document is changed, the node `B` has to be updated to the node `B'` and also appended at the end of the file. This update is propagated to the root node `A` so that finally the new root node `A'` is written after the node `B'`.

Compared to update-in-place schemes, the append-only B$^+$-tree can achieve very high write throughput because all disk write operations occur in a sequential order. Furthermore, we do not need to sacrifice the read performance because the retrieval procedure is identical to the original B$^+$-tree. However, the space occupied by the DB file increases with more updates, thus we have to periodically reclaim the space occupied by the stale data. Couchstore triggers this compaction process when the proportion of the stale data size to the total file size exceeds a configured threshold. All live documents in the target DB file are moved to a new DB file, and the old DB file is removed after the compaction is done. During the compaction process, all write operations to the target DB file are blocked while read operations are allowed. Note that the compaction is performed on one DB file at a time.

Same as the original B$^+$-tree dealing with string keys, if the key length gets longer, the tree height should grow up to maintain the same capacity. It can be even worse in this append-only design because the amount of data to be appended for each write operation is proportional to the height of the tree. As a result, compaction is triggered more frequently, and the overall performance becomes worse and worse. We need a more compact and efficient index for variable-length string keys.

3 ForestDB Design

For effective indexing of variable-length keys, we suggest ForestDB, which is a back-end key-value storage engine for a single node of distributed NoSQL systems. ForestDB is designed as a replacement of Couchstore so that the high-level architecture of both schemes is similar. The major

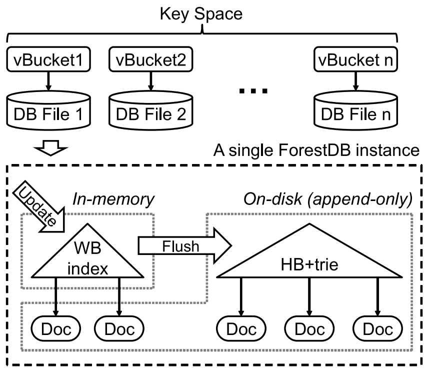


Fig. 4. The overall architecture of ForestDB.

differences between the two schemes are that (1) ForestDB uses a new hybrid index structure, called HB$^+$-trie, which is efficient for variable-length keys compared to the original B$^+$-tree, and (2) the write throughput of ForestDB is improved further by using a log-structured write buffer. The basic concept of the log-structured write buffer is similar to that of C_0 tree and sequential log in LSM-tree, but any further merge operations from the write buffer into the main DB section is not necessary.

Fig. 4 depicts the overall architecture of ForestDB. Same as in Couchstore, there is a single ForestDB instance for each vBucket. Each ForestDB instance consists of an in-memory write buffer index (WB index) and an HB$^+$-trie. All incoming document updates are appended at the end of the DB file, and the write buffer index keeps track of the disk locations of the documents in memory. When the number of entries in the write buffer index exceeds a certain threshold, the entries are flushed into the HB$^+$-trie and stored in the DB file permanently. The detailed mechanism will be described in Section 3.3. The compaction process in ForestDB is the same as in Couchstore. It is triggered when the stale data size becomes larger than a given threshold, and all live documents are moved to a new DB file during compaction.

3.1 HB$^+$-Trie

The main index structure of ForestDB, HB$^+$-trie, is a variant of patricia trie whose nodes are B$^+$-trees. The basic idea of HB$^+$-trie has originated from our prior work [19], and generalized for append-only storage design. All leaf nodes of each B$^+$-tree store disk locations (i.e., byte offsets) of the root nodes of other B$^+$-trees (sub-trees) or documents. Both B$^+$-tree nodes and documents are written into DB files in an append-only manner so that they are interleaved with each other in a file, and maintained in an MVCC model as in Couchstore. There is the root B$^+$-tree on top of HB$^+$-trie, and other sub-trees are created on-demand as new nodes are created in the patricia trie. Fig. 5a presents a logical layout of HB$^+$-trie, and Fig. 5b illustrates how the trie nodes are actually stored in the disk based on the MVCC model.

HB$^+$-trie splits the input key into fixed-size chunks. The chunk size is configurable, for example, 4 or 8 bytes, and each chunk is used as a key for each level of B$^+$-tree consecutively. Searching a document starts from retrieving the root B$^+$-tree with the first (leftmost) chunk as a key. After we obtain a byte offset corresponding to the first chunk from the root B$^+$-tree, the search terminates if a document is stored at this location. Otherwise, when the root node of another sub-tree is written at the byte offset, we continue the search at the sub-tree using the next chunk recursively until the target document is found.

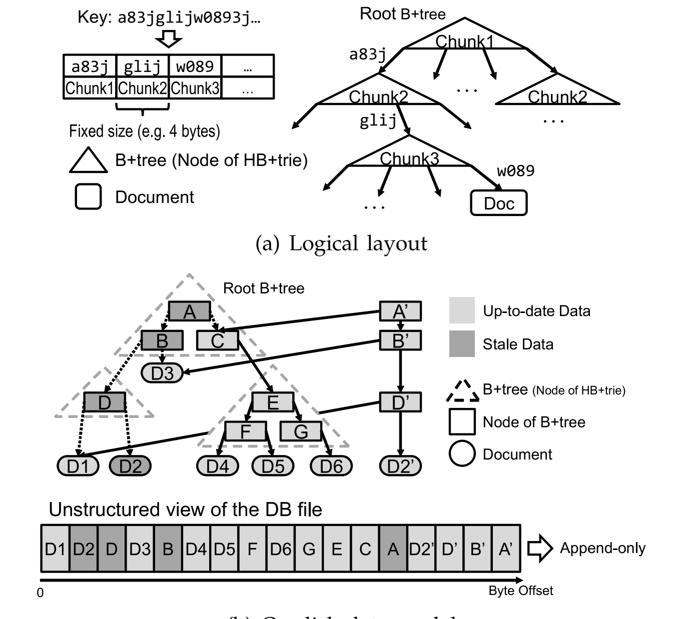


Fig. 5. The hierarchical organization of HB$^+$-trie.

Since the key size of each B$^+$-tree is fixed to the chunk size, which is smaller than the length of the input key string, the fanout of each B$^+$-tree node can be larger than the original B$^+$-tree so that we can shorten the height of each tree. Moreover, in the same way as the original patricia trie, a common branch among keys sharing a common prefix is skipped and compressed. A sub-tree is created only when there are at least two branches passing through the tree. All documents are indexed and retrieved using the minimum set of chunks necessary for distinguishing the document from the others.

Fig. 6 presents insertion examples. Suppose that the chunk size is one byte (i.e., one character), and each triangle represents a single B$^+$-tree as a node of an HB$^+$-trie. The text in each B$^+$-tree indicates (1) the chunk number used as a key for the tree, and (2) the skipped common prefix of the tree. Fig. 6a shows the initial state in which the index stores only one key aaaa. Even though there are four chunks in

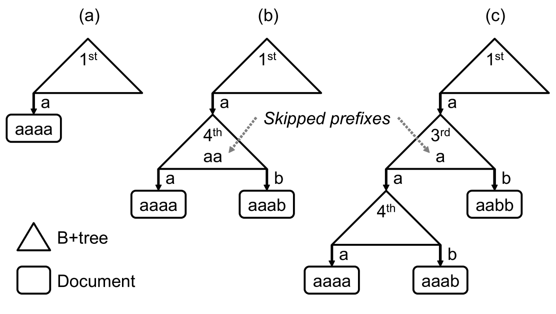
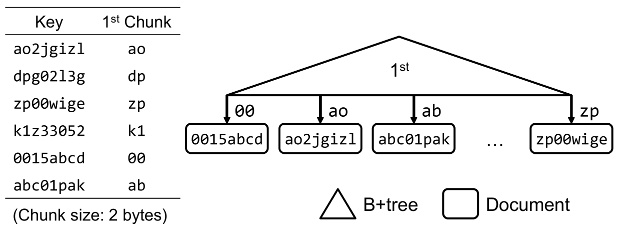


Fig. 6. HB$^+$-trie insertion examples: (a) initial state, (b) after inserting aaab, and (c) after inserting aabb.



Key	1st Chunk
ao2jgizl	ao
dpg02l3g	dp
zp00wige	zp
k1z33052	k1
0015abcd	00
abc01pak	ab

(Chunk size: 2 bytes)

Fig. 7. An example of random key indexing using HB$^+$-trie.

the key aaaa, the document is indexed by the root B$^+$-tree using the first chunk a because the key aaaa is the unique key starting with the chunk a. We can ensure that if the first chunk of the input key is a, the only key corresponding to the input key is aaaa. As a result, the subsequent tree traversals for the remaining chunks can be avoided.

When a new key aaab is inserted into the HB$^+$-trie, a new sub-tree is created, as the key aaab also starts with the first chunk a (cf., Fig. 6b). Since the longest common prefix between aaaa and aaab is aaa, the new sub-tree uses the fourth chunk as its key, and stores aa, which is the skipped prefix between the parent tree (i.e., the root B$^+$-tree) and the new sub-tree. Fig. 6c depicts the situation when another key aabb is inserted. Although the key aabb also starts with the chunk a, it does not match the skipped prefix aa. Hence, a new tree is created between the root and the existing sub-tree. The new tree uses the third chunk as a key because the longest common prefix between aabb and the existing common prefix aaa is aa. The skipped prefix a is stored in the new tree, and the existing prefix kept in the fourth chunk tree is erased because there is no skipped prefix between the two trees.

It is obvious that there are lots of benefits from HB$^+$-trie when the shared common prefix is sufficiently long. Furthermore, we can also obtain a lot of benefit even though their distribution is uniform and there is no common prefix. Fig. 7 shows an example of random keys when the chunk size is two bytes. Since there is no common prefix among the keys, they can be distinguished by the first chunk. In this case, the HB$^+$-trie contains only one B$^+$-tree and we do not need to create any sub-trees to compare the next chunks. Suppose that the chunk size is n bits and the key distribution is uniformly random, then up to 2^n keys can be indexed by storing only their first chunks in the root B$^+$-tree. Compared to the original B$^+$-tree, this can remarkably reduce the entire space occupied by the index structure, by an order of magnitude.

In conclusion, HB$^+$-trie can efficiently reduce the disk accesses caused by redundant tree traversals both when (1) keys are sufficiently long and share a common prefix, and (2) when their distribution is random so that there is no common prefix.

3.2 Optimizations for Avoiding Skew in HB$^+$-Trie

3.2.1 Overview

Since a trie is basically not a balanced structure, HB$^+$-trie could be unnecessarily skewed under specific key patterns, as in the original trie. Fig. 8 depicts two typical examples of the skewed HB$^+$-tries whose chunk size is one byte. If we insert a set of keys where the same chunk is repeated over and over, such as b, bb, and bbb, HB$^+$-trie gets skewed as

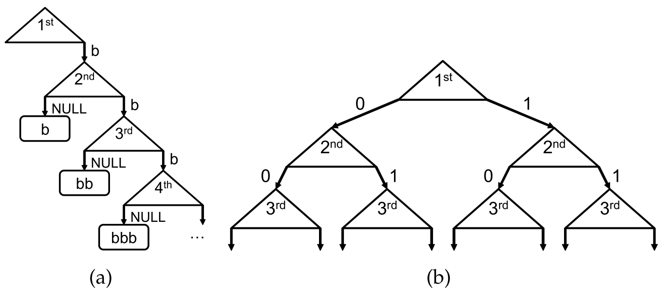


Fig. 8. Skewed HB$^+$-trie examples.

illustrated in Fig. 8a. As a result, the number of disk accesses along the skewed branch largely increases.

Fig. 8b represents another example of a skewed HB$^+$-trie. If the key pattern consists of all permutations of two characters 0 and 1, then each chunk has only two branches so that all B$^+$-trees would contain only two key-value pairs. Since the fanout of the B$^+$-tree node is much higher than two, the overall block utilization becomes very low so that a large number of near-empty blocks will be created. Accordingly, both the space overhead and the number of disk accesses will be very large compared to the original B$^+$-tree.

Note that the use of the longer chunk size lessens the possibility of HB$^+$-trie skew. If we set the chunk size to 8 bytes in Fig. 8b, each chunk has $2^8 = 256$ branches and the B$^+$-tree nodes will be reasonably utilized. However, the overall performance degrades as the chunk size becomes longer due to the contraction of the fanout in each node.

To address this issue, we add an optimization scheme. First we define *leaf B$^+$-tree* as B$^+$-tree that has no child sub-tree, except for the root B$^+$-tree. Instead of a fixed-size chunk, the key of leaf B$^+$-tree consists of a variable-sized string which is a postfix right after the chunk used for its parent B$^+$-tree. Fig. 9 depicts how such leaf B$^+$-trees are organized. The white triangles and gray triangles indicate

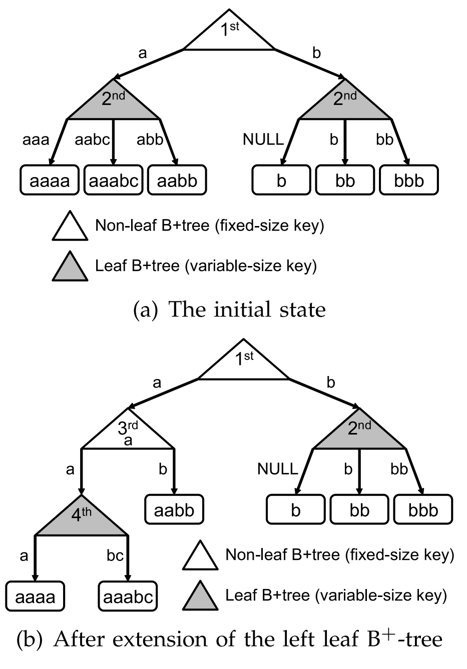

1st
a
b
2nd
2nd
aaa
aabc
abb
NULL
b
bb
aaaa
aaabc
aabb
b
bb
bbb
Non-leaf B+tree (fixed-size key)
Leaf B+tree (variable-size key)
(a) The initial state
1st
a
b
3rd
a
2nd
a
b
NULL
b
bb
4th
aabb
b
bb
bbb
a
bc
aaaa
aaabc
Non-leaf B+tree (fixed-size key)
Leaf B+tree (variable-size key)
(b) After extension of the left leaf B+-tree

Fig. 9. Examples of optimization for avoiding skew.

non-leaf B$^+$-trees and leaf B$^+$-trees, respectively. Non-leaf B$^+$-trees including the root B$^+$-tree index documents or sub-trees using the corresponding chunk as before, while leaf B$^+$-trees use the rest of sub-strings as their keys. For example, the left leaf B$^+$-tree in Fig. 9a indexes documents `aaaa`, `aaabc`, and `aabb` using sub-strings starting from the second chunk `aaa`, `aabc`, and `abb`, respectively. In this manner, even though we insert the key patterns that would trigger skew, no more sub-trees are created and the redundant tree traversals are avoided.

3.2.2 Leaf B$^+$-Tree Extension

This data structure, however, inherits almost all of the problems of the original B$^+$-tree. To avoid this, we extend leaf B$^+$-trees when the total number of keys accommodated in the leaf B$^+$-tree exceeds a certain threshold. For this extension, we first investigate the longest common prefix among the keys stored in the target leaf B$^+$-tree. A new non-leaf B$^+$-tree for the first different chunk is created, and the documents are re-indexed by using the chunk. If there are more than one keys sharing the same chunk, then we create a new leaf B$^+$-tree using the rest of the sub-strings right after the chunk as its key.

Fig. 9b illustrates an example of extending the left leaf B$^+$-tree in Fig. 9a. Since the longest common prefix among `aaaa`, `aaabc`, and `aabb` is `aa`, a new non-leaf B$^+$-tree for the third chunk is created, and the document `aabb` is simply indexed by its third chunk `b`. However, documents `aaaa` and `aaabc` share the same third chunk `a`, thus we create a new leaf B$^+$-tree and index those documents using the rest of sub-strings `a` and `bc`, respectively.

This scheme provides a way to categorize the key space into two types of regions: skew region and normal region. The skew region means a set of keys that is indexed by leaf B$^+$-trees, while the normal region denotes the rest of keys. Since the naive HB$^+$-trie is very inefficient at indexing the aforementioned skewed key pattern, we have to carefully set the extension threshold to prevent the skewed key patterns from being included in normal regions.

3.2.3 Extension Threshold Analysis

To examine the best extension threshold, which is the break-even point of trie-like indexing costs versus tree-like indexing costs for the given key patterns, we mathematically analyze the costs of each indexing. The basic intuition is that the height and space occupation (i.e., the number of nodes) of trie-like structure are greatly influenced by the number of unique branches for each chunk, while only the key length has the most critical impact on those of tree-like structure. Hence, we can derive the point that both the height and space of trie become smaller than those of tree, by using the length of keys and the number of branches for each chunk of the given key patterns.

Table 1 summarizes the notation used in our analysis. Suppose that n documents are indexed by a leaf B$^+$-tree and each B$^+$-tree node is exactly fit into a single block, whose size is B. All keys have the same length k so that the length k is at least equal to or greater than $c\lceil \log_b n \rceil$, where c and b denote the chunk size in HB$^+$-trie and the number of branches in each chunk, respectively. We can simply obtain the fanout of each leaf B$^+$-tree node, f_L, as follows:

TABLE 1
Summary of Notation

Symbols	Definitions
n	the number of documents
B	the size of a block
k	the length of keys used in leaf B$^+$-tree
c	the length of an HB$^+$-trie chunk
v	the size of a value (i.e., byte offset) or a pointer
f_N	the fanout of a node of non-leaf B$^+$-tree
f_L	the fanout of a node of leaf B$^+$-tree
f_L^{new}	the fanout of a node of new leaf B$^+$-trees after extension
s	the space occupied by leaf B$^+$-tree
s_{new}	the space occupied by the new combined data structure after extension
h	the height of leaf B$^+$-tree
h_{new}	the height of the new combined data structure after extension
b	the number of unique branches for each chunk of the given key patterns

$$f_L = \left\lfloor \frac{B}{k+v} \right\rfloor, \tag{1}$$

where v is the size of a byte offset or a pointer. For the given n documents, we can derive the overall space occupied by the leaf B$^+$-tree,[2] s, and the height of the leaf B$^+$-tree, h, as follows:

$$s \simeq \left\lceil \frac{n}{f_L} \right\rceil B, \tag{2}$$

$$h = \lceil \log_{f_L} n \rceil. \tag{3}$$

After extension, b new leaf B$^+$-trees are created since each chunk has b branches. A new non-leaf B$^+$-tree is also created and b leaf B$^+$-trees are pointed to by the non-leaf B$^+$-tree.[3] Recall that the new leaf B$^+$-trees use the rest of the sub-string right after the chunk used as the key of its parent non-leaf B$^+$-tree,[4] thus the fanout of the new leaf B$^+$-tree, f_L^{new}, can be represented as follows:

$$f_L^{new} = \left\lfloor \frac{B}{(k-c)+v} \right\rfloor. \tag{4}$$

Since the non-leaf B$^+$-tree uses a single chunk as key, the fanout of the non-leaf B$^+$-tree, f_N, can be derived as follows:

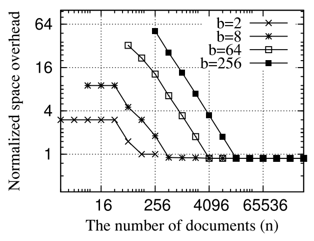
$$f_N = \left\lfloor \frac{B}{c+v} \right\rfloor. \tag{5}$$

By using f_N and f_L^{new}, we can obtain the space overhead and the height of the new combined data structure, denoted as

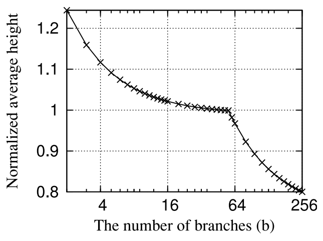
2. For the sake of simplicity, we ignore the space occupied by index nodes of leaf B$^+$-tree because it takes a very small portion compared to that by leaf nodes.

3. We assume the worst case where there is no document directly pointed to by the non-leaf B$^+$-tree.

4. We assume that there is no skipped prefix by the non-leaf B$^+$-tree.



(a) The value of s_{new}/s according to the number of documents n, with various b values



(b) The average value of h_{new}/h when the number of documents n is ranged from $b \cdot f_L$ to $b \cdot {f_L}^2$, according to the value of b

Fig. 10. The variation of (a) the space overhead and (b) the height of the data structure resulting from leaf B$^+$-tree extension, normalized to those values before the extension.

s_{new} and h_{new}, respectively, as follows:

$$s_{new} \simeq \left(\left\lceil \frac{b}{f_N} \right\rceil + b \left\lceil \frac{n}{b \cdot f_L^{new}} \right\rceil \right) B, \tag{6}$$

$$h_{new} = \lceil \log_{f_N} b \rceil + \left\lceil \log_{f_L^{new}} \frac{n}{b} \right\rceil. \tag{7}$$

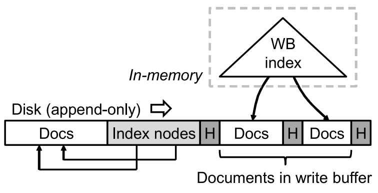
Fig. 10a illustrates the variation of s_{new} normalized to the value of s, when b is 2, 8, 64, and 256. We set the parameter values as follows: $B = 4{,}096$, $k = 64$, $c = 8$, and $v = 8$, and accordingly the values of f_L, f_N, and f_L^{new} are calculated as 56, 256, and 64, respectively. As b gets larger, the space overhead increases. This is because the extension procedure creates b new leaf B$^+$-trees. Since a single B$^+$-tree occupies at least one block, the b new leaf B$^+$-trees occupy at least b blocks, whereas $\lceil \frac{n}{f_L} \rceil$ blocks are occupied by the previous leaf B$^+$-tree. The value of s_{new} gets smaller than s when $b < \frac{n}{f_L} \Leftrightarrow n > b \cdot f_L$, which is the point that the number of blocks occupied by the new leaf B$^+$-trees becomes smaller than the block occupation before extension.

However, although the space overhead shrinks, the overall height would increase after extension. Fig. 10b represents the average height of the combined data structure after extension, h_{new}, varying the value of b. The height is normalized to the value of h, which is the height of the leaf B$^+$-tree before extension. When b is greater than the fanout of the previous leaf B$^+$-tree f_L, the new height h_{new} gets shorter than the previous height h because the number of branches in the non-leaf B$^+$-tree after extension becomes greater than the fanout of the leaf B$^+$-tree before extension.

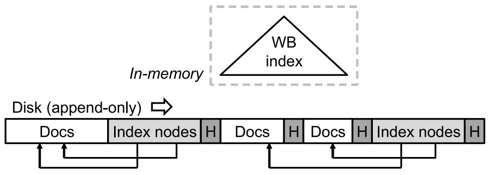
After considering all of these factors, we extend the leaf B$^+$-tree when (1) $n > b \cdot f_L$, and (2) $b \geq f_L$. Note that we need to scan all keys in the leaf B$^+$-tree to get the exact values of k and b, which triggers a large amount of disk I/O. To avoid this overhead, we only scan the root node of the leaf B$^+$-tree. The root node contains a set of keys at near-regular intervals on the entire list of keys, thus we can estimate the approximate values. Since the root node is already cached by the previous tree operation, no additional disk I/O is necessary.

3.3 Log-Structured Write Buffer

Although HB$^+$-trie can reduce the tree height and the overall space overhead, more than one B$^+$-tree node can



(a) Before flushing write buffer



(b) After flushing write buffer

Fig. 11. Write buffer examples.

still be appended into the DB file for every write operation. To lessen the amount of appended data per write operation, ForestDB uses a log-structured write buffer. It is quite similar to the C_0 tree and sequential log in LSM-tree, but the documents inserted in the write buffer section do not need to be merged into the main DB section, since the main DB itself is also based on the log-structured design.

Fig. 11a depicts an example. Docs in the white boxes indicates a set of contiguous disk blocks used for documents, while Index nodes in the gray boxes denotes blocks used for B$^+$-tree nodes organizing the HB$^+$-trie. For every commit operation, a single block containing the DB header information is appended at the end of the file, which is illustrated as H in the dark gray boxes.

All incoming document updates are simply appended at the end of the file, while updates on the HB$^+$-trie are postponed. There is an in-memory index called *write buffer index (WB index)*, which points to the locations of the documents that are stored in the file but not yet reflected on HB$^+$-trie. When a query request for a document arrives, ForestDB looks it up in the write buffer index first, and continues to look it up in HB$^+$-trie next if the request does not hit the write buffer index.

The entries in the write buffer index are flushed and atomically reflected in the HB$^+$-trie when a commit operation is performed if and only if the cumulative size of the committed logs exceeds a configured threshold (e.g., 1,024 documents). After flushing write buffer, the updated index nodes corresponding to the documents in the write buffer are appended at the end of the file, as shown in Fig. 11b. As we aforementioned, the documents themselves do not need to be moved or merged but just need to be linked by the updated index nodes, since ForestDB already uses a log-structured design. This greatly reduces the overall costs required to flush write buffer.

If a crash occurs in the write buffer index before the flush, we scan each block reversely from the end of the file until the last valid DB header written right after index nodes. Once the DB header is found, then ForestDB reconstructs the write buffer index entries for the documents written after the header. We also maintain a 32-bit CRC value for each document to detect a corruption, thus only documents stored normally are recovered.

TABLE 2
The Characteristics of Key Patterns

Name	Description	Key Length
random	Keys are randomly generated so that there is no global common prefix. This is the best case for HB$^+$-trie.	8–256 bytes
worst	There are 20 levels of nested prefixes where each level has only two branches. The average prefix size for each level is 10 bytes. This is the worst case for HB$^+$-trie.	Avg. 198 bytes
small	There are 100 randomly generated prefixes so that each 10,000 keys share a common prefix. The average prefix size is 10 bytes, and the rest of the key string is randomly generated.	Avg. 65 bytes
2-level	There are two levels of nested prefixes where each level has 192 branches. The average prefix size for each level is 10 bytes, and the rest of the key string is randomly generated.	Avg. 64 bytes

By using the log-structured write buffer, a number of index updates are batched so that the amount of disk I/O per document update can be greatly reduced. This makes the write performance of ForestDB comparable to or even better than that of LSM-tree-based approaches.

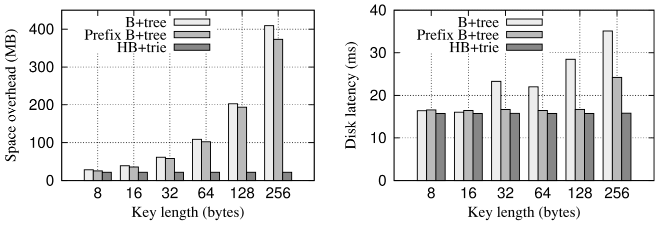
4 Evaluation

In this section, we present evaluation results that show the key features of ForestDB. ForestDB is implemented as a stand-alone key-value store library pluggable to Couchbase Server. The evaluation was performed on a 64-bit machine running Linux 3.8.0-29, equipped with Intel Core i7-3770 @ 3.40 GHz CPU (4 cores, 8 threads), 32 GB RAM, and Western Digital Caviar Blue 1TB HDD,[5] where the disk is formatted using the Ext4 file system. The chunk size of HB$^+$-trie is set to 8 bytes for all evaluations, and we implemented each leaf B$^+$-tree in HB$^+$-trie as a Prefix B$^+$-tree.

4.1 Comparison of Index Structures

First, we investigate the overall space overhead and the average traversal path length of HB$^+$-trie compared to those of the original B$^+$-tree and Prefix B$^+$-tree. As an initialization, one million documents are artificially synthesized and then the corresponding one million keys with 8-byte byte offsets for the document location are inserted into each index. To verify the optimization schemes for avoiding skew, we used four key patterns as described in Table 2: random, worst, small, and 2-level. Each B$^+$-tree node is aligned to 4 KB block size, and all indexes including HB$^+$-trie are written into the storage in an update-in-place manner for exact estimation of the live index size.

Fig. 12a shows the disk space occupied by each index structures after initialization using the random key pattern,



(a) Space overhead (b) Average latency on HDD

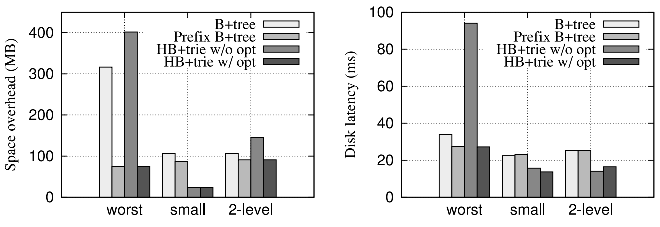
Fig. 12. Comparison of B$^+$-tree, Prefix B$^+$-tree, and HB$^+$-trie using the random key pattern.

with the key length varying from 8 to 256 bytes. The space overhead of B$^+$-tree and Prefix B$^+$-tree increases linearly as the key length gets longer, while that of HB$^+$-trie remains constant. This is because all keys can be indexed by only their first chunks so that the key length does not influence the space overhead of HB$^+$-trie.

To evaluate the average length of traversal path from the root to the leaf, we randomly read key-value pairs and observe the time elapsed for disk accesses during the tree operations. As disk I/O operation is skipped if the request hits the OS page cache, we enable the `O_DIRECT` flag in order to make all tree node reads incur actual disk accesses. Fig. 12b depicts the results. The disk latency of HB$^+$-trie is nearly steady even though the key length increases, while the disk latency of the B$^+$-tree increases rapidly due to the tree height growth caused by the decreased fanout degree. Note that Prefix B$^+$-tree also displays nearly steady disk latency when the key length is less than 256 bytes. This is because the random keys have no (or few) common prefixes, thus the minimal sub-strings stored in index nodes become shorter so that the fanout of each index node becomes larger. Although the small fanout of the leaf nodes enlarges the space overhead, the large fanout of the index nodes can shorten the height of the tree.

Next we repeat the same evaluations using the worst, small, and 2-level key patterns. Fig. 13 illustrates the space overhead and the average disk latency of each index structure. Note that *HB$^+$-trie w/o opt* denotes HB$^+$-trie without optimizations for avoiding skew, while *HB$^+$-trie w/ opt* represents HB$^+$-trie that enables such optimizations.

In the worst pattern, the space overhead of Prefix B$^+$-tree is largely reduced compared to that of B$^+$-tree, since there are a lot of common prefixes and their lengths are sufficiently long. In contrast, the unoptimized HB$^+$-trie occupies even more space than B$^+$-tree because the key pattern leads to the trie skew so that the overall block utilization becomes very low. Even worse, the disk latency of the unoptimized



(a) Space overhead (b) Average latency on HDD

Fig. 13. Comparison of B$^+$-tree, Prefix B$^+$-tree, and HB$^+$-trie using worst, small, and 2-level key patterns.

5. WD10EALX, maximum data transfer rate: 126 MB/s.

HB$^+$-trie is larger than those of B$^+$-tree and prefix B$^+$-tree by 2.8x and 3.4x, respectively. However, the HB$^+$-trie with optimizations effectively reduces the overhead by using leaf B$^+$-trees. Both the space overhead and the disk latency of HB$^+$-trie become similar to those of Prefix B$^+$-tree, as each leaf B$^+$-tree in HB$^+$-trie is implemented as a Prefix B$^+$-tree.

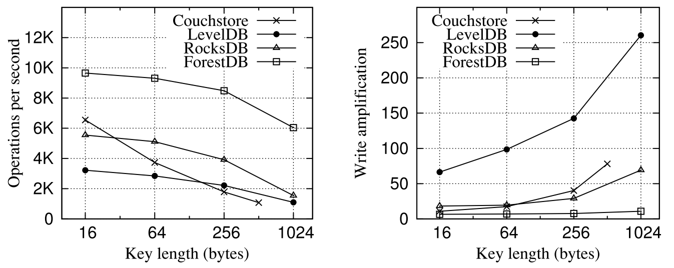
Unlike the worst pattern, there are a number of branches in each chunk of the small pattern, hence trie-like indexing is more advantageous than tree-like indexing. As shown in Fig. 13, the space overhead and the disk latency of HB$^+$-trie without any optimizations are already better than those of B$^+$-tree and Prefix B$^+$-tree. Note that HB$^+$-trie with optimizations also displays a similar performance, because the leaf B$^+$-trees are well extended so that the overall shape of the data structure becomes almost the same as the naive HB$^+$-trie.

Since the 2-level pattern also has a lot of branches for each chunk, the average disk latency of HB$^+$-trie without optimizations can be better than that of B$^+$-tree and Prefix B$^+$-tree. However, every third chunk has only $1,000,000/192^2 \simeq 27$ branches on average, which is much smaller than the maximum fanout of the non-leaf B$^+$-tree node (i.e., 256 in this evaluation). As a result, plenty of near-empty blocks are created so that the space overhead of the naive HB$^+$-trie becomes larger than that of B$^+$-tree and Prefix B$^+$-tree. As we mentioned in Section 3.2.3, the optimization scheme handles this situation well and prevents leaf B$^+$-trees from being extended. Consequently, the space occupation of HB$^+$-trie with optimizations becomes similar to that of Prefix B$^+$-tree.

4.2 Full System Performance

We next evaluate the full system performance of ForestDB compared with that of Couchstore 2.2.0, LevelDB 1.18, and RocksDB 3.5 using the storage operations of Couchbase Server. For fair comparison, API-wrapping layers for LevelDB, RocksDB, and ForestDB are used to convert Couchstore operations into their own operations. The document compression using Snappy [20] in both LevelDB and RocksDB is enabled, but we randomly generate each document body so that only key part is compressible. In Couchstore and ForestDB, compaction is triggered when the size of the stale data becomes larger than 30 percent of the entire data size, while LevelDB and RocksDB continuously perform compaction in the background using separate threads. The write buffer threshold of ForestDB is configured to 4,096 documents by default, and the `O_DIRECT` flag is disabled in all full system performance evaluations. To reduce read amplification for both LevelDB and RocksDB, we add Bloom filters where the number of bits per key is set to 10. We set each module's custom block cache size to 8 GB, except for Couchstore which does not support its own caching functionality.

Each module is initialized using 200 million keys with the corresponding documents whose size is 1,024 bytes (excluding the key size). The total working set size ranges from 190 to 200 GB, which is almost 6x larger than the RAM capacity. For all evaluations, keys are generated based on the 2-level key pattern. Since HDD is too slow (less than 100 ops/second) to initialize each module and to get meaningful results, we instead use a Samsung 850 Pro 512 GB SSD[6] for full system evaluations. We have confirmed that the overall relative results among the schemes on an SSD are similar to those on an HDD.



(a) Overall throughput for randomly ordered operations with 20% update ratio

(b) Average write amplification per a single update operation

Fig. 14. Performance comparison according to various key lengths.

4.2.1 Key Length

First, we investigate the overall performance with key lengths varying from 16 to 1,024 bytes. In order to see the effect of the key length clearly, keys are randomly generated and the document size is set to 512 bytes in this evaluation. A number of read and update operations are randomly performed where the ratio of updates is 20 percent of the total. Multiple updates are applied together using a synchronous write, and we randomly select the batch size from 10 to 100 documents. Since Couchstore does not work correctly when the key length is longer than 512 bytes, we use 512-byte key in Couchstore instead of 1,024-byte key.

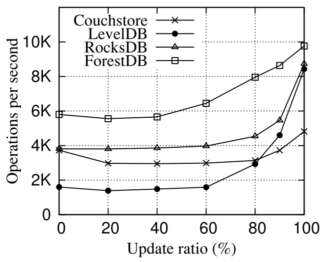
Fig. 14a illustrates the overall throughput in terms of the number of operations per second. As the key length gets longer, the overall throughput of Couchstore, LevelDB, and RocksDB decreases by 3x–11.3x, while that of ForestDB is reduced by only 37 percent due to the use of HB$^+$-trie. Note that although HB$^+$-trie is hardly influenced by the key length, the throughput of ForestDB slightly decreases as the key length gets longer due to an increase in the document size, where the entire key string is included in the document as its metadata.

Since all of these schemes use an out-of-place update, the space occupation increases with more updates. Fig. 14b shows the average write amplification including the overhead of compaction. Couchstore, LevelDB, and RocksDB require more disk writes as the key length gets longer, whereas the amount of disk writes in ForestDB is almost consistent. Note that the amount of disk writes in LevelDB is especially large compared to other schemes, since LevelDB is based on LSM-tree where a number of merge and compaction operations are triggered during the updates. Although RocksDB is also based on the same LSM-tree structure, it reduces the overall write amplification by using various optimizations.

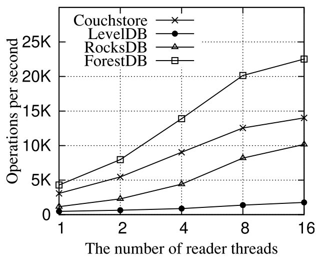
4.2.2 Read/Write Ratio

In order to study the performance characteristics according to the proportion of read and write operations, we evaluate

6. MZ-7KE512B, max. seq read: 550 MB/sec, seq write: 520 MB/sec, random read: 100,000 IOPS, random write: 90,000 IOPS.



(a) Overall throughput for randomly ordered operations with the given update ratio



(b) Total throughput of concurrent reader threads

Fig. 15. Performance comparison with various (a) the update ratios and (b) the number of concurrent reader threads.

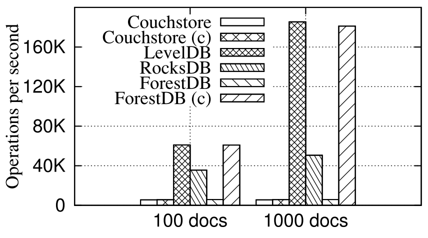


Fig. 16. Range scan performance.

the overall throughput by varying the ratio of update operations from 0 to 100 percent. The key length is fixed to 32 bytes, and the operations are randomly performed in the key space. We use synchronous writes as in the previous evaluations.

Fig. 15a presents the results. The overall throughputs of LevelDB, RocksDB, and ForestDB get better as the update ratio becomes higher, since they use the log-structured approach. All incoming updates are sequentially logged without retrieving or updating the main index, thus the total number of sequential disk accesses increases in proportion to the update ratio. In contrast, the throughput of Couchstore is almost steady regardless of the update ratio because all updates are immediately reflected in B$^+$-tree. Although Couchstore is also based on an append-only design, updating and appending new B$^+$-tree nodes require the old version of the nodes to be read. Since the old nodes are randomly scattered over the disk, the random reads intermixed with the sequential updates cancel the benefits from the append-only logging.

Note that although HB$^+$-trie in ForestDB is hierarchically organized by several B$^+$-trees, the read performance of ForestDB is better than those of LevelDB and RocksDB with Bloom filters. This is because the overall index size in ForestDB is relatively compact, since each level of B$^+$-tree in HB$^+$-trie only stores the corresponding chunks instead of the entire key strings. Consequently, it increases the hit ratio of both the OS page cache and DB module's own cache so that the average number of actual disk accesses required for a single retrieval operation becomes low.

When the workloads become read-only, the overall performance of each scheme becomes slightly better than when the workloads include a few update operations. If there is no update operation, no compaction or merge operations are performed and we can save the overhead of additional disk I/Os triggered by compaction.

4.2.3 Concurrent Read/Write Operation

As both ForestDB and Couchstore are based on the MVCC model, concurrent read and write operations do not block each other. To investigate the multi-threaded performance, we create a writer thread that continuously performs random write operations at 2,000 ops/sec throughput, and also create concurrent reader threads that invoke random read operations at their maximum capacity.

Fig. 15b depicts the total read throughput according to the number of concurrent reader threads. When there is a single reader thread, the overall read throughput of LevelDB and RocksDB is much slower than that in Fig. 15a. This is because write operations and additional merge or compaction operations triggered by the write operations obstruct the concurrent read operations. In contrast, ForestDB and Couchstore show even better performance than the single-threaded results,[7] since reader threads are hardly affected by the concurrent writer thread.

As the number of concurrent reader threads increases, the read throughputs of all schemes get better. This means that concurrent read operations in all schemes do not block themselves. Note that the read throughput is saturated at some point, because the operations are bounded by disk I/O.

4.2.4 Range Scan

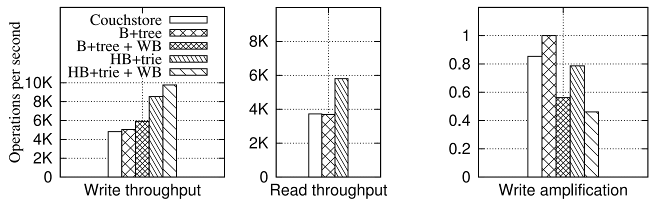
One of the main benefits of LSM-tree-based approaches is high range scan performance, as all tree-like components except for C_0 logs are maintained in a sorted order. In contrast, append-only approaches such as ForestDB and Couchstore suffer from range scan, since incoming updates are just appended at the end of the DB file so that consecutive documents in the key space may not be adjacent on the physical disk. However, during the compaction process, documents are migrated to the new file in a sorted order so that the range scan performance can be much better than before.

In order to evaluate the range scan performance, we randomly pick a document and sequentially scan 100 or 1,000 consecutive documents. Fig. 16 shows the results. Note that *Couchstore (c)* and *ForestDB (c)* denote Couchstore and ForestDB right after the compaction is done. In both cases, LevelDB shows the best throughput. The range scan performance of Couchstore and ForestDB before compaction is not much different from the random read performance, while ForestDB after compaction has almost the same throughput as LevelDB. Note that Couchstore shows poor range scan performance even after the compaction. This is because Couchstore does not maintain its own block cache so that it alternately visits documents and B$^+$-tree nodes that are not physically adjacent on disk.

4.2.5 The Effect of Write Buffer and HB$^+$-Trie

We next observe the separate effect of write buffer and HB$^+$-trie in ForestDB. Fig. 17 illustrates the overall throughput and write amplification of ForestDB with various indexing options. *HB$^+$-trie* denotes ForestDB without write buffer, whereas *B$^+$-tree* indicates ForestDB without write buffer

7. Since results in Fig. 15b excludes the throughput of the writer thread which is 2,000 ops/sec, the overall throughput with a single reader thread is higher than that in Fig. 15a.



(a) Write and read throughputs for randomly ordered operations

(b) Write amplification normalized to B^+-*tree*

Fig. 17. The characteristics of ForestDB with various indexing configurations.

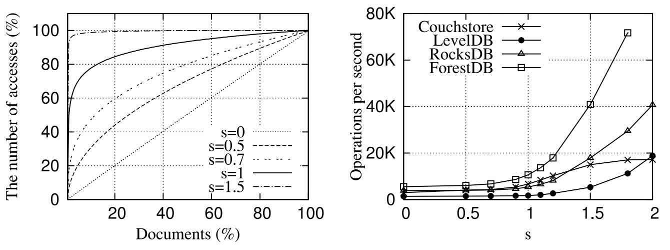
using naive B$^+$-tree as an index structure instead of HB$^+$-trie. B^+-*tree* + *WB* means ForestDB with write buffer using naive B$^+$-tree, and *HB*$^+$-*trie* + *WB* represents the original ForestDB.

With HB$^+$-trie, the overall throughput for both read and write operation gets better due to the reduced number of disk accesses per document retrieval. Fig. 17b reveals that write buffer efficiently lessens the write amplification by buffering the duplicated index node updates, for both B$^+$-tree and HB$^+$-trie. Note that the write amplification of Couchstore is slightly better than that of B^+-*tree*. This is because ForestDB writes more data in each document's meta section, compared to Couchstore.

4.2.6 Locality

To assess the performance variation according to different types of localities, we perform read and update operations based on the Zipf's distribution [21] instead of uniformly random selection. Suppose that there are N elements, the frequency function for the pth element in the Zipf's distribution is given by $f(p, s, N) = \frac{1}{p^s} / \sum_{n=1}^{N} \frac{1}{n^s}$, where s is a parameter value that determines the characteristics of the distribution. The element with $p = 1$ has the highest frequency, and the frequency gradually decreases as the value of p increases. The series of the frequency becomes similar to a uniform distribution as the s value gets closer to zero, whereas the frequency differences among the elements becomes greater with a larger s value.

We organize 10,000 randomly selected documents into one document group; hence there are total 20,000 document groups for 200 million documents. Next we generate 20,000



(a) CDF of Zipf's distribution with $N = 20,000$

(b) Overall throughput for operations based on Zipf's distribution with 20% update ratio

Fig. 18. Performance comparison according to the value of s in Zipf's distribution.

frequencies using the Zipf's distribution with $N = 20,000$, where each frequency is mapped one-to-one to a randomly selected document group. For performing read and update operations, (1) an arbitrary document group is selected based on the frequencies, and (2) we choose a random document in the group.

Fig. 18a plots the cumulative distribution functions of the number of accesses to each document with various values of s. We can control the locality of the workload by varying the value of s; the locality increases as the value of s increases. In the case of $s = 1$, for example, approximately 80 percent of the accesses are concentrated to 10 percent of the total documents.

The overall performance comparison results according to the different types of localities are illustrated in Fig. 18b. The throughput of each scheme improves as the locality becomes higher, since the higher locality makes the hit ratio in the OS page cache better so that the total number of disk accesses for retrieval operations decreases. The increasing rate for the performance of ForestDB according to the locality is much higher than those of the others. This is because the average index size per document in ForestDB is much smaller compared to the other schemes, thus more index data can be kept with the same amount of RAM.

4.2.7 Real Dataset Results

We finally evaluate the overall performance of each scheme using real datasets obtained from an on-line music streaming service and a web spam collection. Table 3 describes the

TABLE 3
The Characteristics of Real Datasets

Name	Description	Avg. Key Length	Avg. Doc Size	# Docs	Working Set Size
profile	Indexes user profile information such as name, e-mail, and password hash value using the corresponding user ID number.	33 bytes	230 bytes	4,839,099	1.7 GB
playlist	Maps playlist ID number to detailed information for the corresponding playlist such as the ID number of the owner user, the name of the playlist, the timestamp when it is created or updated, etc.	39 bytes	1,008 bytes	592,935	700 MB
history	Stores user activity logs such as timestamp and the corresponding activity ID number when they update playlist or play the music.	29 bytes	2,772 bytes	1,694,980	4.7 GB
status	Indexes the current status information of users using their e-mail or ID number.	23 bytes	21 bytes	1,125,352	200 MB
url	URLs in WEBSPAM-UK2007 [22], the collection of web spam datasets. URLs are used as keys, while each document body is randomly generated.	111 bytes	1,024 bytes	105,896,555	112 GB

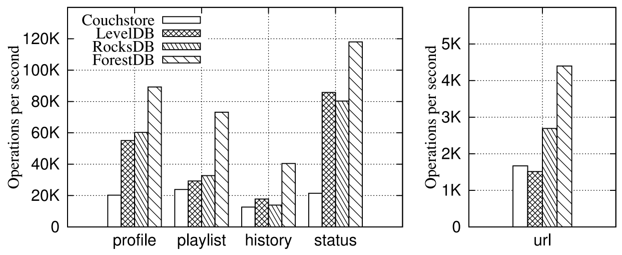


Fig. 19. The overall throughput for randomly ordered operations with 20 percent update ratio, using real datasets.

characteristics of each dataset. Each scheme is initialized using the datasets and we perform uniformly random read and synchronous update operations. The ratio of update operations to the total number of operations is set to 20 percent which is the same as the previous evaluations. Since the working set size of profile, playlist, history, and status is fit into RAM, the overall throughput of those datasets is much faster than that of url. This is because the entire DB files are maintained in the OS page cache so that no disk I/O is triggered during the read operations.

Fig. 19 represents the evaluation results. The overall performance of Couchstore is almost steady regardless of the characteristics of the datasets. In contrast, both LevelDB and RocksDB display better performance as the total size of the document (i.e., the sum of the key length and the document body size) gets smaller. This is because every level of the sorted table in such schemes also stores the body of the document so that the document body size influences the overall merge or compaction overhead. In url, RocksDB outperforms LevelDB due to its optimizations for I/O bounded workloads.

In all datasets, the overall performance of ForestDB is better than those of the other schemes. As HB$^+$-trie reduces both the length of traversal path from the root to the leaf and the overall index size, the cost of index retrieval decreases not only for I/O bounded workloads but also in-memory datasets.

5 Related Work

Key-value stores for a single node can be categorized into two types: (1) in-memory key-value indexes using storage for durability, and (2) general purpose key-value storage engines. In the former case, the key-value store maintains the key-value pairs on storage for durability, but all or most of index data always reside in memory. The main objective of this type of key-value stores is to cache the locations of key-value data in the durable storage as many as possible by reducing the size of the memory footprints in terms of bytes/key.

Masstree [23] is an in-memory key-value store based on a trie-like concatenation of multiple B$^+$-trees. To minimize the DRAM fetch time which is a crucial factor for in-memory queries, Masstree chooses the fanout of B$^+$-trees considering cache lines shared among multiple cores. Incoming updates are logged on a persistent disk for a backup purpose, while all index nodes are maintained in memory only. Although the high-level design of Masstree is similar to HB$^+$-trie, there are significant differences between them. First, Masstree mainly targets workloads that all key-value pairs are fit into RAM, thus it does not need to consider how to organize and write index nodes into disk, which is the most tricky issue for the performance improvement of storage system. Second, Masstree does not provide any solution for balancing skewed trie, as it deals with relatively short and uniformly distributed keys. Since the trie skew issue becomes a major obstacle not only to tree traversals but also to space overhead, it needs to be solved to serve generic workloads.

Lim et al. have proposed SILT [24], which is a partial in-memory key-value store system that is organized by a three-tier indexing model: LogStore, HashStore, and SortedStore. The average size of the memory footprints per document increases from LogStore to HashStore to SortedStore, while the capacity increases in reverse order. All incoming updates are stored in LogStore first, and flushed into HashStore next. When the number of entries in HashStore exceeds a given threshold, the entries are merged into SortedStore, which is based on a trie structure. Note that it is impossible to partially update an entry in SortedStore, thus the merge operation from HashStore to SortedStore always involves a full revision of the entire index.

FlashStore [25] and SkimpyStash [26] have been suggested for key-value indexing in flash memory. The key-value pairs are sequentially logged in storage, and the in-memory hash table points to the location of the key-value logs. The main difference between the two schemes is the amount of memory usage. FlashStore maintains a single hash entry for each key-value log, while several key-value logs organize a linked list in flash memory and a single hash entry is dedicated for each linked list in SkimpyStash. This can be represented as a trade-off between the size of the memory footprint per key-value pair and the number of disk accesses per retrieval.

Note that all aforementioned systems keep their index data in memory, thus recovery after system crash requires a tremendous amount of disk I/Os due to scanning of all logs in storage. Furthermore, shutting down or re-booting the system also involves a number of disk accesses because all of the in-memory index data has to be written back to the disk. This overhead is hardly acceptable for the general-purpose use in the single node storage engine of distributed NoSQL systems.

Recall that another type of key-value stores is a general-purpose back-end engine that embedded into other systems or applications which require key-value indexing functionalities. Generally their overall performance can be worse than the former key-value stores because both the key-value data and the corresponding index data are written into storage for each update operation. However, they can cope with the various kinds of crashes or failures better and commonly show reasonable responses with very little RAM usage.

BerkeleyDB [7] is one of the most popular key-value stores that provides the core back-end key-value store functionalities. It basically uses B$^+$-tree-based indexing, but other indexing methods such as hashing can be used. Various types of transactional features such as full ACID (Atomicity, Consistency, Isolation, Durability) [27] and WAL are also supported. It is known that BerkeleyDB delivers a good performance with a relatively lightweight footprints for various types of applications.

LevelDB [2] is another persistent key-value store, whose main concept is borrowed from Google's BigTable [12]. LevelDB organizes the on-disk data as a variant of LSM-tree, where each component of the LSM-tree is called *level* in LevelDB. Riak [28], a NoSQL DB based on Amazon's Dynamo [29], has embraced LevelDB as one of its storage engine. Facebook has introduced RocksDB [3], which is an extension of LevelDB. It adopts various optimizations to improve a wide range of system factors such as disk utilization, read amplification, and compaction overheads.

Sears and Ramakrishnan have suggested bLSM [30], which is designed to be a viable backing storage for PNUTS [31], Yahoo's distributed key-value store system, and Walnut [32], Yahoo's next-generation elastic cloud storage system. bLSM improves the overall read performance of LSM-trees by adopting Bloom filters and suggests a new compaction scheduler called *spring and gear*, to bound the write latency without impacting the overall throughput.

6 Conclusions and Future Work

This paper presents ForestDB, a per-node key-value storage engine for indexing variable-length string key-value pairs. ForestDB uses HB$^+$-trie as an index structure on persistent storage, which is a combination of a patricia trie and B$^+$-trees. HB$^+$-trie has a low disk access time and small space overhead compared to the tree-like index structures. However, the index can be skewed under rarely-occurring specific input patterns because a trie is basically not a balanced structure. To address this problem, we additionally suggest optimization schemes for avoiding the trie skew. Moreover, ForestDB uses a log-structured write buffer to further reduce the amount of disk writes per document update. We observe that ForestDB achieves significantly faster throughput compared to the other key-value store schemes, in terms of the number of operations per second.

Since ForestDB currently runs on top of traditional file systems, the duplicated overhead of retrieving an index or updating metadata in both the file system and ForestDB is inevitable. To avoid this issue, we plan to propose a volume management layer, which allows ForestDB to access the block devices directly bypassing the file system layer. We believe the overall performance can be greatly improved by performing raw block I/O operations.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441). This work was also partly supported by the ICT R&D program of MSIP/IITP [10041313, UX-oriented Mobile SW Platform].

References

[1] Couchbase NoSQL Database. [Online]. Available: http://www.couchbase.com/, 2011.

[2] J. Dean and S. Ghemawat. (2011). LevelDB: A fast and lightweight key/value database library by Google. [Online]. Available: https://github.com/google/leveldb

[3] RocksDB: a persistent key-value store for fast storage environments. [Online]. Available: http://rocksdb.org/, 2013.

[4] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Proc. IEEE 6th Int. Conf. Pervasive Computing and Applications (ICPSCA)*, 2011, pp. 363–366.

[5] D. Comer,"Ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[6] P. O'Neil, E. Cheng, D. Gawlick, andE. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[7] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, 1999, pp. 183–191.

[8] P. Fruhwirt, M. Huber, M. Mulazzani, and E. R. Weippl, "InnoDB database forensics," in *Proc. 24th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, 2010, pp. 1028–1036.

[9] MongoDB. [Online]. Available: http://www.mongodb.com, 2009.

[10] SQLite4. [Online]. Available: https://sqlite.org/src4/doc/trunk/www/index.wiki, 2013.

[11] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[12] 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," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, p. 4, 2008.

[13] R. Bayer and K. Unterauer, "Prefix B-trees," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 11–26, 1977.

[14] A. A. Moffat, T. C. Bell, and I. H. Witten, *Managing Gigabytes: Compressing and Indexing Documents and Images*. San Mateo, CA, USA: Morgan Kaufmann, 1999.

[15] D. R. Morrison, "Patricia: Practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[16] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, 1981.

[17] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[18] Apache CouchDB. [Online]. Available: http://couchdb.apache.org/, 2005.

[19] J.-S. Ahn. (2011). Third annual SIGMOD programming contest. [Online]. Available: http://goo.gl/yKUoiY

[20] Snappy: A fast compressor/decompressor. [Online]. Available: http://code.google.com/p/snappy/, 2011.

[21] G. K. Zipf, *Human Behavior and the Principle of Least Effort*, Cambridge, MA, Addison-Wesley Press, 1949.

[22] "WEBSPAM-UK2007. [Online]. Available: http://chato.cl/webspam/datasets/uk2007/.

[23] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. 23rd ACM Symp. Oper. Syst. Principles.*, 2011, pp. 1–13.

[25] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," in *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[26] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD Int. Conf. Manage. Data.*, 2011, pp. 25–36.

[27] J. Gray, "The transaction concept: Virtues and limitations," in *Proc. 7th Int. Conf. Very Large Data Bases*, 1981, vol. 81, pp. 144–154.

[28] Riak. [Online]. Available: http://basho.com/riak/, 2009.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[30] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data.*, 2012, pp. 217–228.

[31] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," in *Proc. VLDB Endowment*, 2008, vol. 1, no. 2, pp. 1277–1288.

[32] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: A unified cloud object store," in *Proc. ACM SIGMOD Int. Conf. Manage. Data.*, 2012, pp. 743–754.

Jung-Sang Ahn received the BS and MS degrees in computer science from KAIST in 2008 and 2010, respectively. Currently, he is pursuing the PhD degree in computer science at the same school. His research interests include indexing systems, file and storage systems, mobile devices, and operating systems.

Chiyoung Seo received the PhD degree from the Computer Science Department at the University of Southern California in 2008. He is a software architect at Couchbase and mainly works on caching, replication, and storage engines. His recent focus is to develop the next generation of a storage engine that is optimized for Solid State Drives. Before Couchbase, he worked at Yahoo! for two years and participated in developing the next generation of display advertising platform.

Ravi Mayuram received the MS degree in mathematics from the University of Delhi. He is a senior vice president of engineering at Couchbase. Before joining Couchbase, he was a senior director of engineering at Oracle, leading innovations in the areas of recommender systems and social graph, search and analytics, and lightweight client frameworks. Previously in his career, he has held senior technical and management positions at BEA, Siebel, Informix and HP in addition to couple of start ups including BroadBand office, a Kleiner Perkins funded venture.

Rahim Yaseen received the PhD degree in EE/CSE from the University of Florida, specializing in database systems. He is a former senior vice president of engineering at Couchbase. Prior to joining Couchbase, he served as a vice president of engineering at Oracle Corporation. Prior to Oracle, he held executive engineering positions at SAP, Siebel, and CMGI/AdForce. Prior to that, he led the development of Siebels BPM frameworks including workflow architecture and platform, rule engines and tools and compilers for scripting languages.

Jin-Soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University (SNU), Korea, in 1991, 1993, and 1999, respectively. He is currently a professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor in KAIST from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received the BS degree in electronics engineering from Seoul National University (SNU), Korea, in 1977, and the MS and PhD degrees in computer science in 1979 and 1984, respectively, from KAIST, where he has been a faculty member in the Department of Computer Science since 1984. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include microarchitecture, parallel computer architecture, cluster computing, and embedded systems.