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

April 18, 2025

1 Introduction

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

2 Background

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

However, the fanout is greatly influenced by the length of the keys as we previously mentioned. To moderate this overhead, Prefix B^+-tree [BU77] 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.

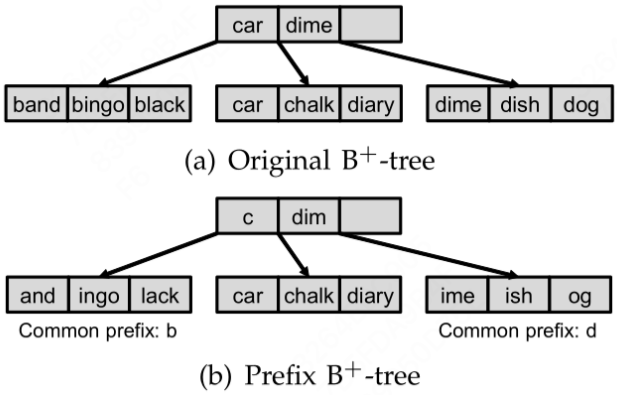


Figure 1: Examples of B^+-tree and Prefix B^+-tree

2.2 LSM-Tree

2.3 Couchstore

Couchstore is a per-node storage engine of Couchbase Server, whose overall architecture inherits from the storage model of Apache CouchDB.

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

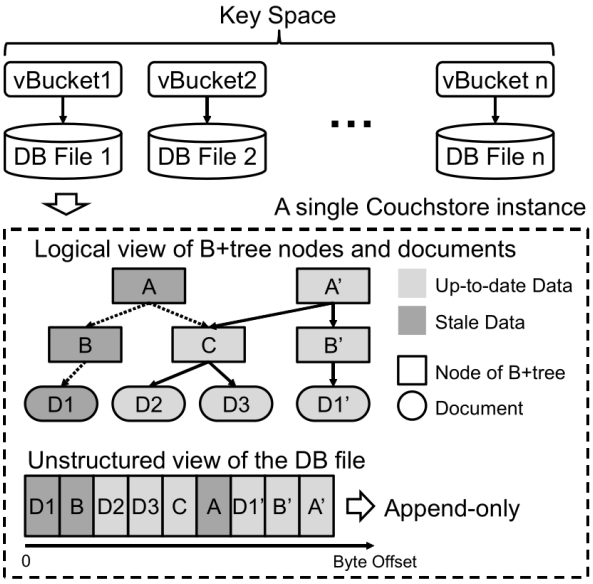


Figure 2: The overview of Couchstore

Note that all updates in Couchstore are appended at the end of the DB file as illustrated in Figure 2. `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

ForestDB is designed as a replacement of Couchstore so that the high-level architecture of both schemes is similar. The major 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

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.

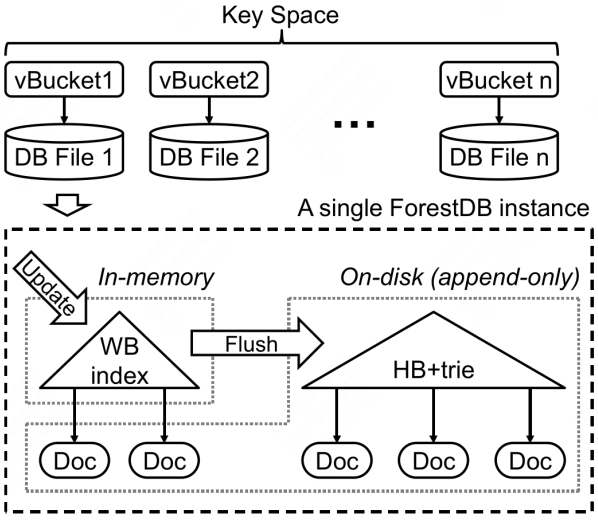


Figure 3: The overall architecture of ForestDB

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.

3.1 HB$^+$-Trie

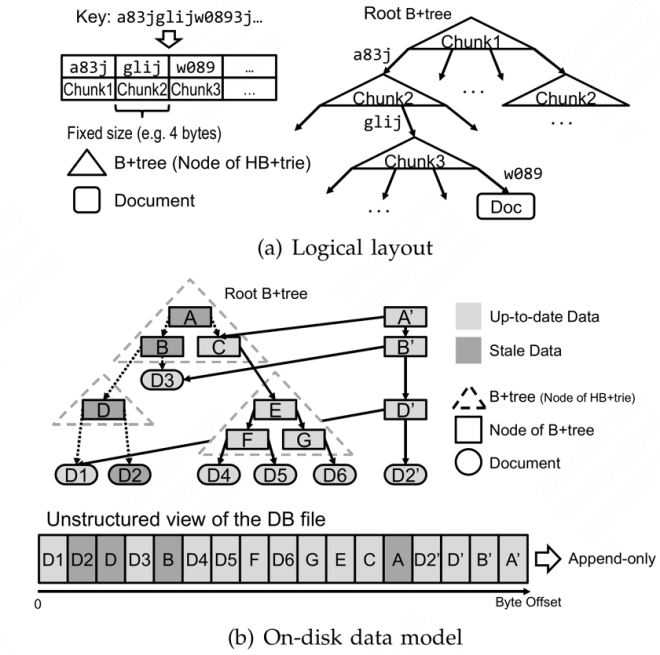


Figure 4: The hierarchical organization of HB$^+$-trie

The main index structure of ForestDB, HB$^+$-trie, is a variant of patricia trie whose nodes are B$^+$-trees. 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. Figure 4(a) presents a logical layout of HB$^+$-trie, and Figure 4(b) 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 bytes 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.

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.

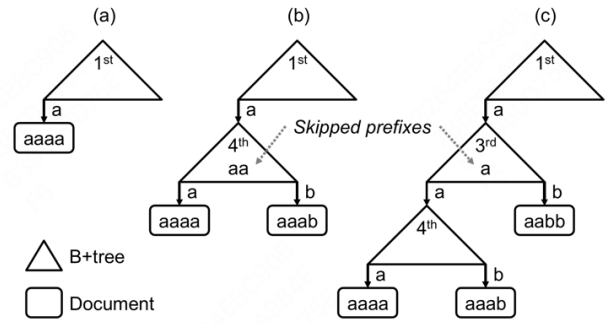


Fig. 6. HB$^+$-trie insertion examples: (a) initial state, (b) after inserting `aaab`, and (c) after inserting `aabb`

Figure 3.1 presents insertion examples. Suppose that the chunk size is one byte, 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

2. the skipped common prefix of the tree.

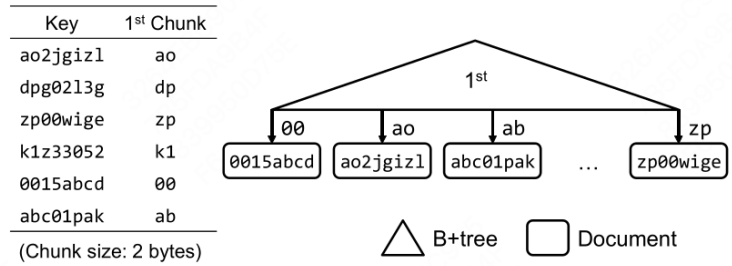


Figure 5: An example of random key indexing using HB$^+$-trie

Figure 5 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.

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

3.2.1 Overview

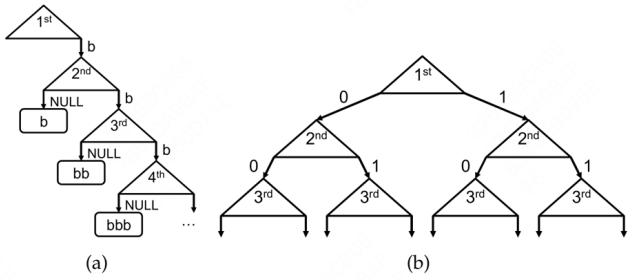
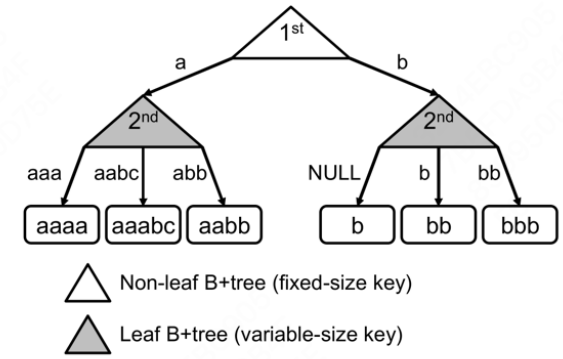
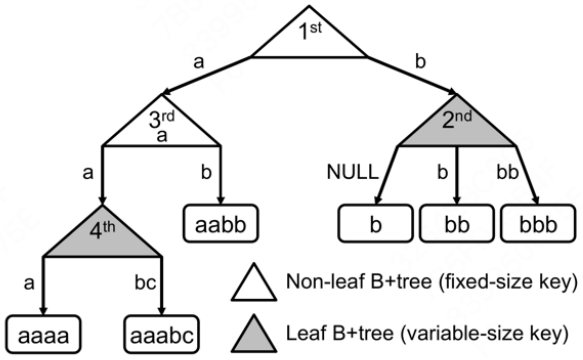


Figure 6: Skewed HB$^+$-trie examples

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.



(a) The initial state



(b) After extension of the left leaf B^+-tree

Figure 7: Examples of optimization for avoiding skew

Figure 7 depicts how such leaf B^+-trees are organized. The white triangles and gray triangles indicate 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 Figure 7(a) 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

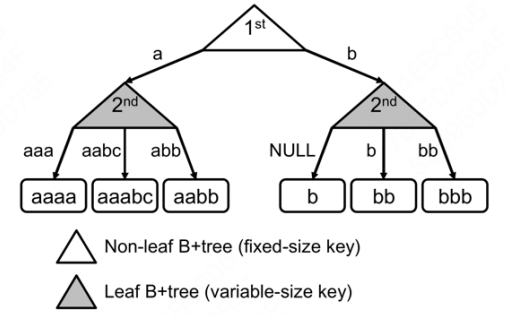
we extend leaf B^+-trees when the total number of keys accommodated in the leaf B^+-tree exceeds a certain threshold. For this extension, we first investigate the longest common prefix among the keys stored in the target leaf B^+-tree. A new non-leaf B^+-tree for the first different chunk is created, 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.

Figure 7(b) illustrates an example of extending the left leaf B^+-tree in Figure 7(a). 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.

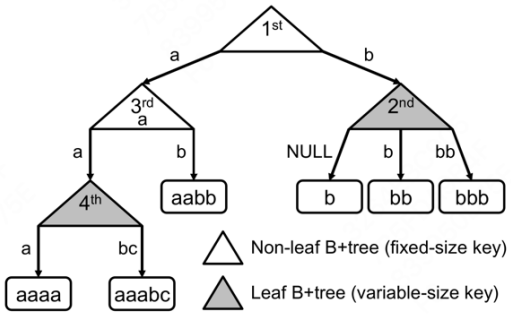
This scheme provides a way to categorize the key space into two types of regions: skew region and normal region. The skew region means a set of keys that is indexed by leaf B^+-trees, while the normal region denotes the rest of keys. Since the naive HB^+-trie is very inefficient at indexing the aforementioned 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

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.



(a) The initial state



(b) After extension of the left leaf B$^+$-tree

Figure 8: Summary of notation

Figure 8 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 $k \geq c \left\lceil \log_b n \right\rceil$ where c and b denote the chunk size in HB$^+$-trie and the number of branches in each chunk. We can obtain the fanout of each leaf B$^+$-tree node, f_L, as follows:

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

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, s, and the height of the leaf B$^+$-tree, h, as follows:

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

$$h = \left\lceil \log_{f_L} n \right\rceil$$

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$^+$-tree is also created and b leaf B$^+$-trees are pointed to by the non-leaf B$^+$-tree. 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, thus the fanout of the new leaf B$^+$-tree, f_L^{new}, can be represented as follows:

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

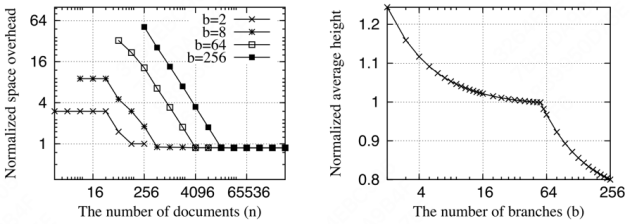
Since the non-leaf B$^+$-tree uses a single chunk as key, the fanout of the non-lefa B$^+$-tree, f_N, can be derived as follows:

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

Now we can obtain the space overhead and the height of the new combined data structure, denoted as s_{new} and h_{new}, respectively, as follows:

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

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



(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

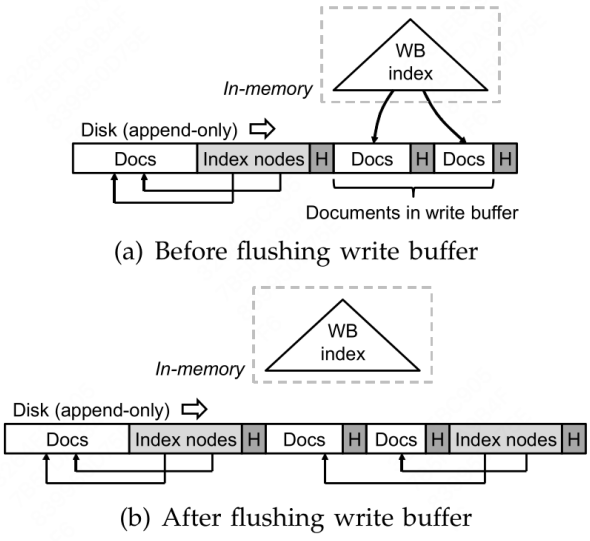
We extend the leaf B$^+$-tree when

1. $n > b \cdot f_L$ ($s_{new} < s$)
2. $b \geq f_L$ ($h_{new} < h$)

We only scan the root node of the leaf B$^+$-tree to get k and b.

3.3 Log-Structured Write Buffer

More than one B+ -tree node can 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.



(a) Before flushing write buffer

(b) After flushing write buffer

Figure 9: Write buffer example

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 Figure 9(b).

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.

4 Evaluation

4.1 Comparison of Index Structures

4.2 Full System Performance

ForestDB/Couchstore/LevelDB/RocksDB

5 Problems

1. Why not consider lsm?

6 References

References

[BU77] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, March 1977.