ForestDB - A Fast Key-Value Storage Engine for Variable-Length Keys

Overview

A general B+-tree implementation usually has the following drawbacks:

- B-tree stores the entire string of key inside not only leaf nodes but also index (internal) nodes including root node. As the length of key gets longer, the fan-out degree of each node gets smaller, and it makes the height of entire tree longer. Since the number of disk accesses is proportional to the height of tree (because we should traverse the tree from root to leaf), storing the entire string is inefficient for disk I/O. Furthermore, we should compare the entire string of key in each node with query while traversing from root to leaf. This B-tree implementation is inefficient to index variable size string keys.

- If a file layout is not block-aligned, it will require more IO operations that expected. Although logical view of file provided by file system is byte-addressable, the actual device I/O is performed on a per-block basis (note that block size is 4KB in general). When we try to read a node written over two blocks, the device driver have to read entire two blocks even though whatever the node size is. To avoid this additional overhead, the size of a node should be fitted into multiple size of a block, and different types of data should not be interleaved in a block.

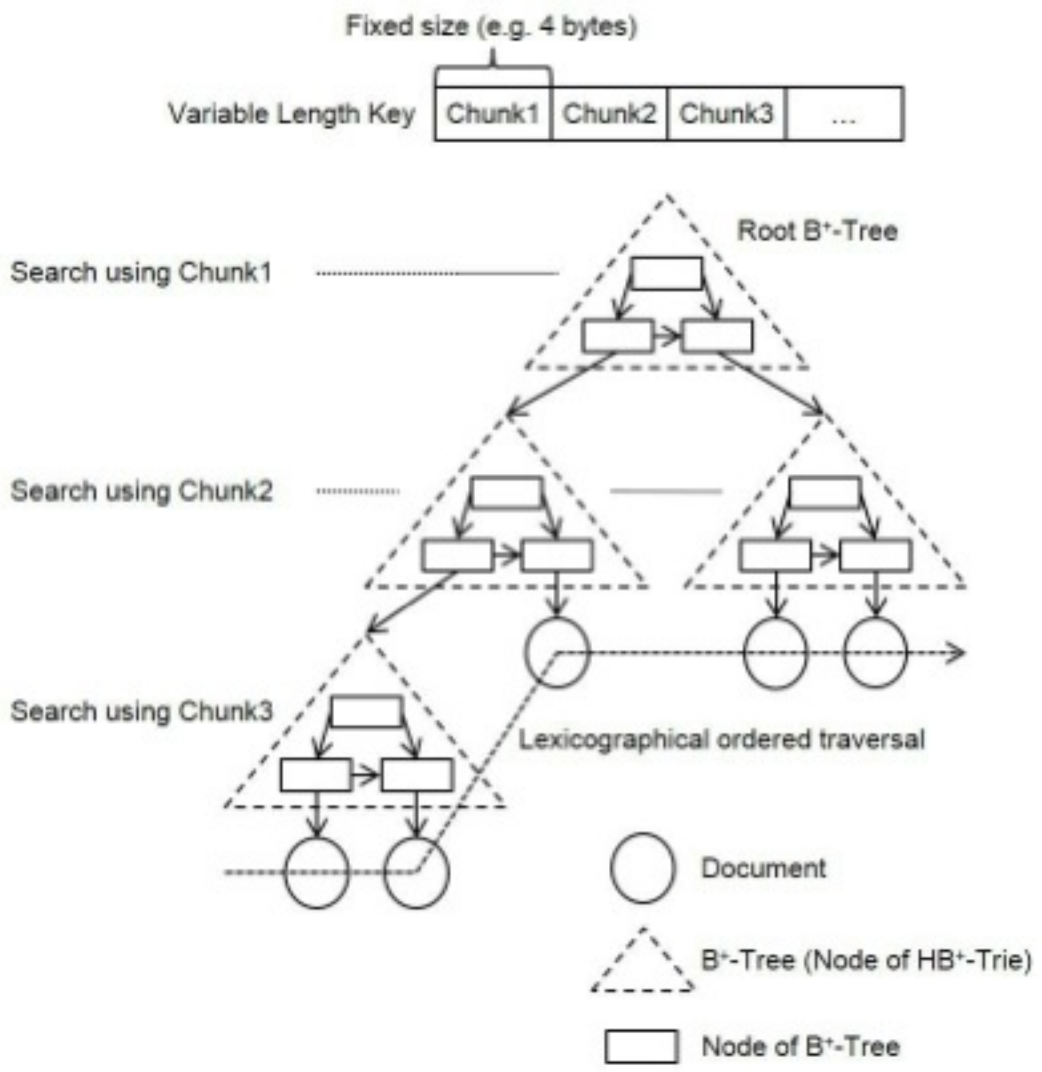
- B+-tree implementation can rely on the OS file system cache to cache btree nodes in memory. However, the file system cache is a shared resource among other processes, and representative NoSQL communities point that the OS page cache significantly increases the latency and slows down the IO performance on SSD especially.

To address these limitations, we propose ***ForestDB***, a fast key-value storage engine that is based on hierarchical B+-tree trie data structure and provides the same API functionalities as Couchstore. In this work, we suggest trie (also known as prefix tree)-like structure, called ***HB+-trie*** (Hierarchical B+-tree based trie), which can support efficient indexing and retrieval on a large number of variable string keys over tree-like structures, in terms of block device I/O. In addition, documents and internal nodes of trie are stored as block-aligned form so that we can minimize the number of block-I/O accesses while indexing the documents. We expect that this new key-value storage can improve the overall I/O performance of EP Engine.

Design

HB+-Trie

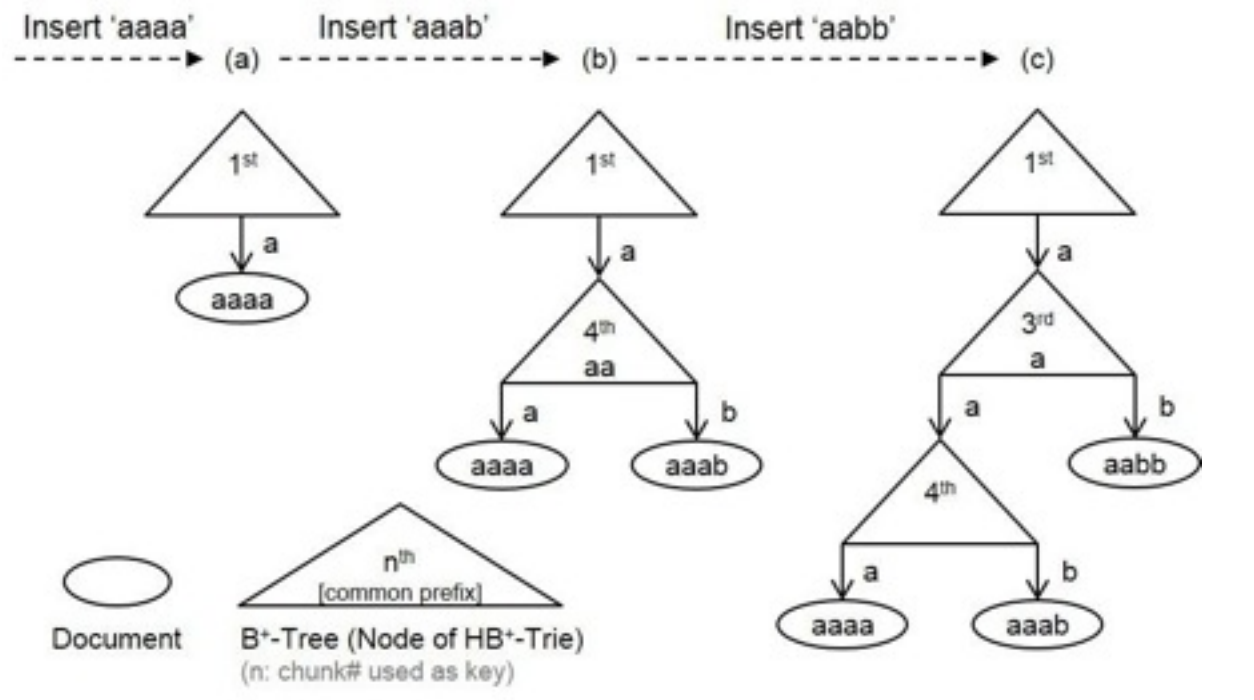
The main index structure is a variant of trie whose nodes are B+-trees. Each B+-tree has its own nodes where all leaf nodes of each B+-tree have pointers to other B+-trees (sub-trees) or actual documents. There is a root B+-tree on top of HB+-trie, and other sub-trees are created on-demand as new nodes are created in trie.



As in the original trie, HB+-trie splits the document key into fixed-length chunks. The size of each chunk is configurable, and we set 4 bytes as default. Each chunk is used as a key for each level of B+-tree sequentially. Searching the index structure starts by retrieving the root B+-tree with the first (leftmost) chunk as a key. After we get a pointer corresponding to the first chunk from the root B+-tree, the searching terminates if a document is pointed to by the pointer. When the pointer points to a sub-tree, we continue the searching at the sub-tree with the next chunk recursively until the target document is found. This retrieval process is basically same as that of the original trie.

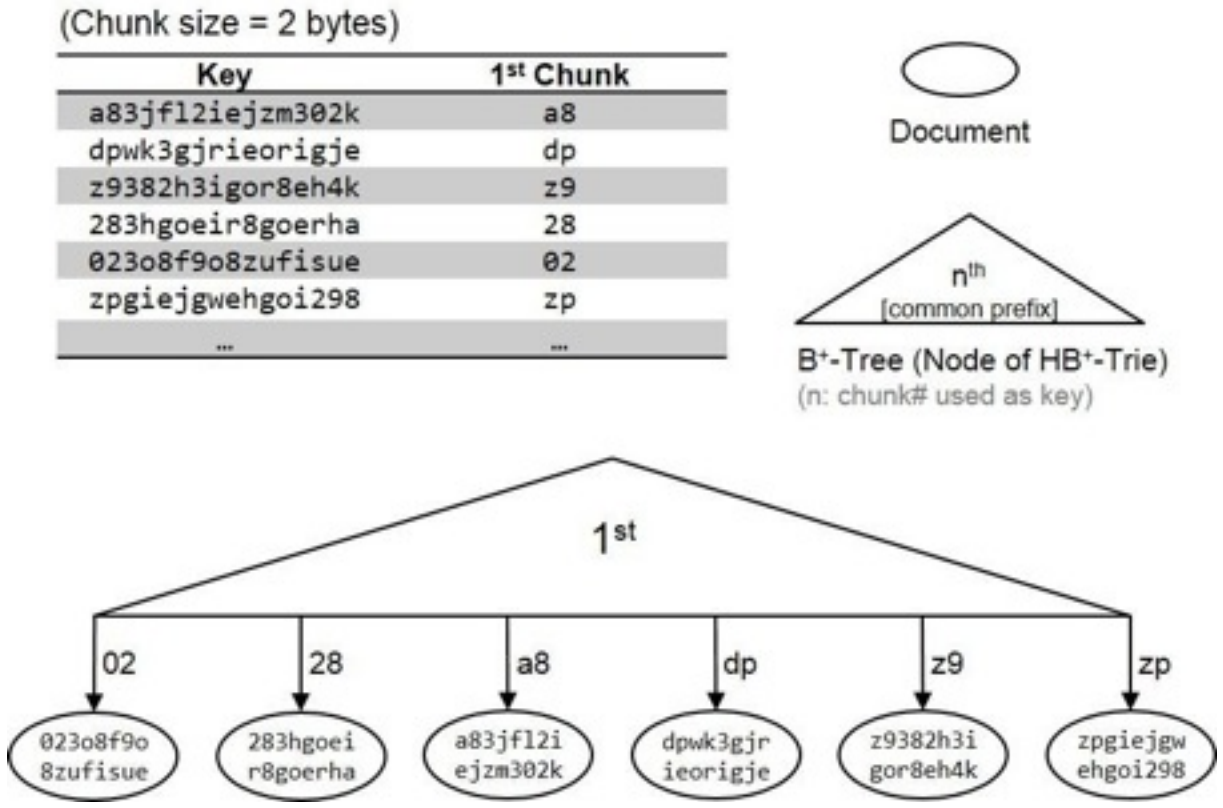
There are several benefits of HB+-trie over tree-like structures. First, dealing with a long key can be fast and space-efficient. Since we do not need to compare the entire string of key for each node, lookup speed can be faster than tree-like structures which compare the entire key for each node on the path from root to leaf. Second, we do not store the entire key in HB+-trie while B+-tree should provide appropriate space (up to key size * fan-out degree) for every node to store keys.

HB+-trie also provide common prefix compression scheme as the original trie. This scheme skips any common branch among keys sharing the common prefix, reducing unnecessary tree traversals. The tree using the first different chunk (i.e. the chunk next to the common prefix) as a key stores the entire common prefix inside its header to reconstruct the original key for all searches passing through the tree. The next figure shows an example when chunk size is one byte (character). The number in the triangle (i.e. n-th) denotes the chunk number used as key in the tree, and the string represents the (common) prefix skipped by the tree.



We use HB+-trie as ID-index which indexes document IDs as its key. For indexing sequence numbers of each document, a B+-tree is used as Seq-index. To make code simple, we share the B+-tree implementation of each node in HB+-trie with Seq-index. Since the key size of this B +-tree is fixed, there is no space overhead to support variable length key.

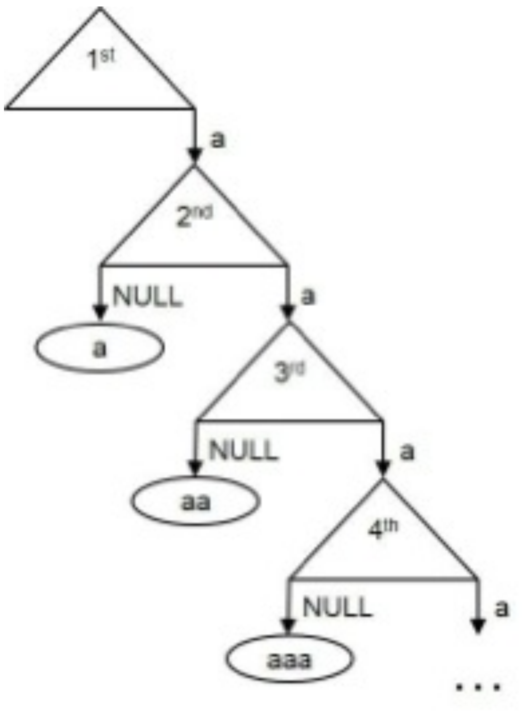
We can get lots of benefit from HB+-trie when keys are sufficiently long and their distribution is uniform random (e.g. in case of hash values). The next figure shows an example of random keys when chunk size is 2 bytes. Since there is no common prefix, they can be distinguished by the first chunk and we do not need to create sub-trees to compare next chunks. Suppose that the chunk size is n-bit and key distribution is (ideal) uniform random, up to 2^n keys can be indexed by storing only their first chunks in the first B+-tree. This makes HB+-trie much more efficient in terms of space occupied by the index structure and the number of disk accesses, compared to the naive B+-tree implementation that stores entire strings in its nodes.


(Chunk size = 2 bytes)

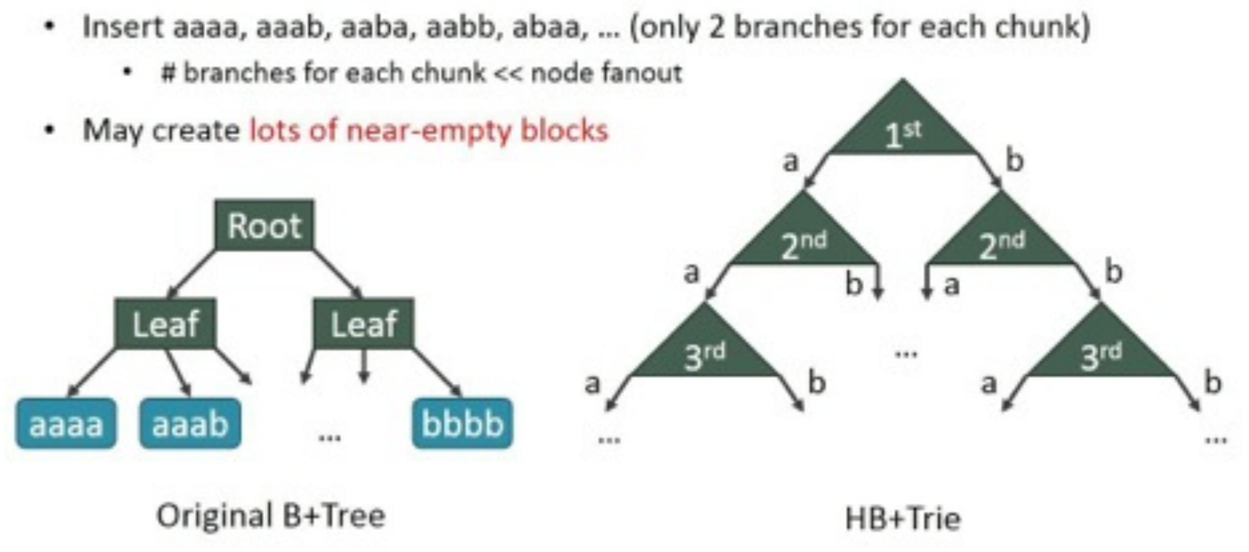
Key	1st Chunk
a83jfl2iejzm302k	a8
dpwk3gjrieorigje	dp
z9382h3igor8eh4k	z9
283hgoeir8goerha	28
023o8f9o8zufisue	02
zpgiejgwehgoi298	zp
...	...

Avoiding Trie Skew

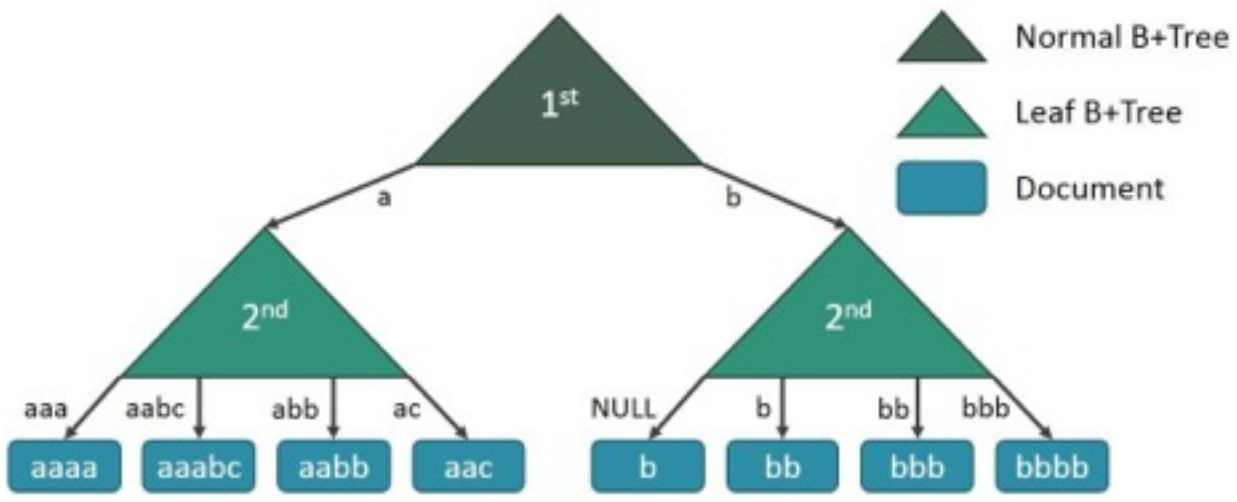
Same as the original trie, HB+Trie can be skewed under specific key patterns. The next figure shows an example of skewed HB+Trie whose chunk size is 1-byte. If we insert a set of keys composed of monotonically cumulating same string pattern (and each pattern is exactly aligned to the chunk size) such as a, aa, aaa and aaaa, HB+Trie is skewed as illustrated in the figure. This makes the node (block) utilization very low, and the number of disk accesses on skewed branch is largely increased.



The next figure represents another example of skew. If all keys have only two branches for each chunk, then all B+Trees will store only two key-value pairs. This makes the overall block utilization very low so that lots of near-empty blocks will be created. As a result, both space overhead and the number of disk accesses will increase rapidly.

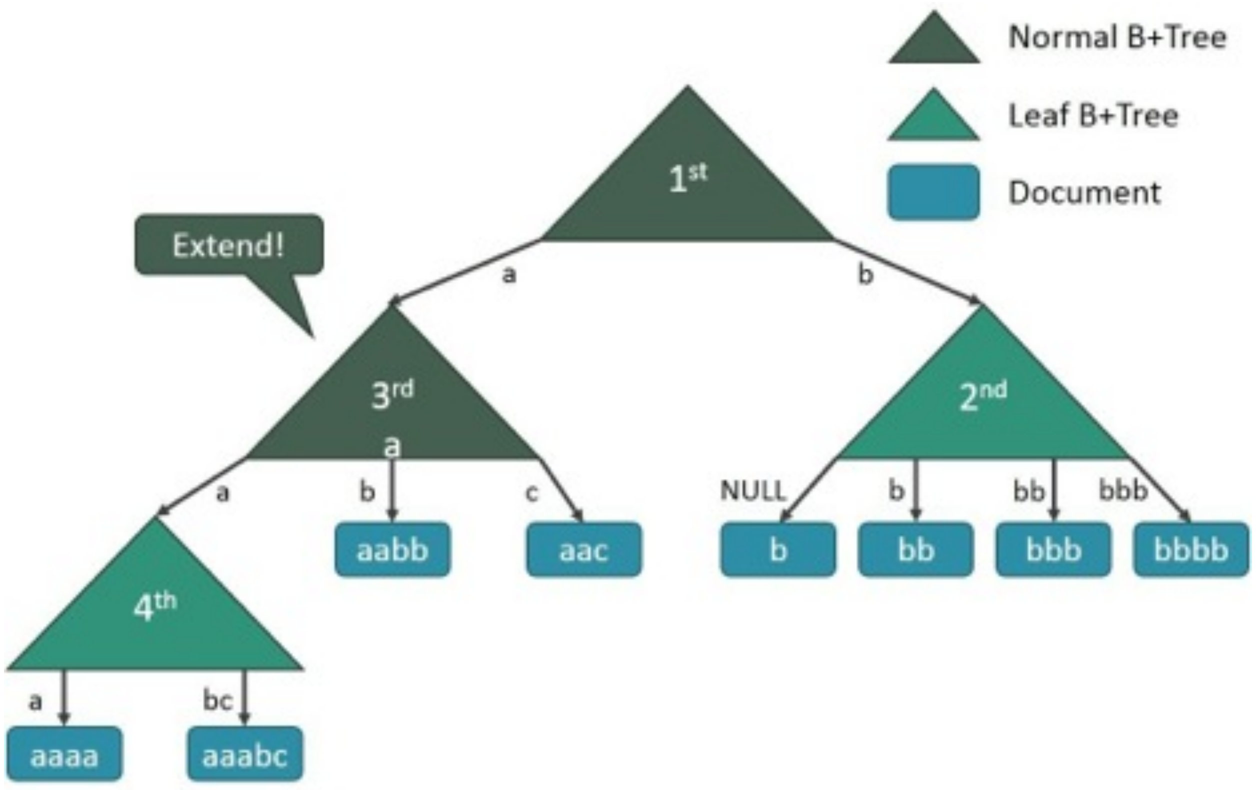

- Insert aaaa, aaab, aaba, aabb, abaa, ... (only 2 branches for each chunk)
  - # branches for each chunk << node fanout
- May create lots of near-empty blocks

To avoid those problems, we first define a leaf B+Tree as a B+Tree that has no child sub-tree, except for root B+Tree. Unlike normal (non-leaf) B+Tree, leaf B+Tree uses the entire sub-strings (postfix) just after the chunk used for its parent B+Tree as its keys. The next figure depicts how they are organized.



Root B+Tree indexes documents using the first chunk as before, while leaf B+Trees use the rest of sub-strings starting from the second chunk as their keys. For example, the left leaf B+Tree indexes document ‘aaabc’ and ‘aabb’ using their sub-strings ‘aabc’ and ‘abb’, respectively. In this way, even though HB+Trie indexes key patterns that may trigger skew, no more sub-tree is created due to leaf B+Tree as shown in the right leaf B+Tree in the figure.

However, this data structure share almost all problems that the original B+Tree has. To address this, we extend leaf B+Tree when the total number of keys stored in the leaf B+Tree exceeds a certain threshold. The next figure describes an example of extension on the left leaf B+Tree.

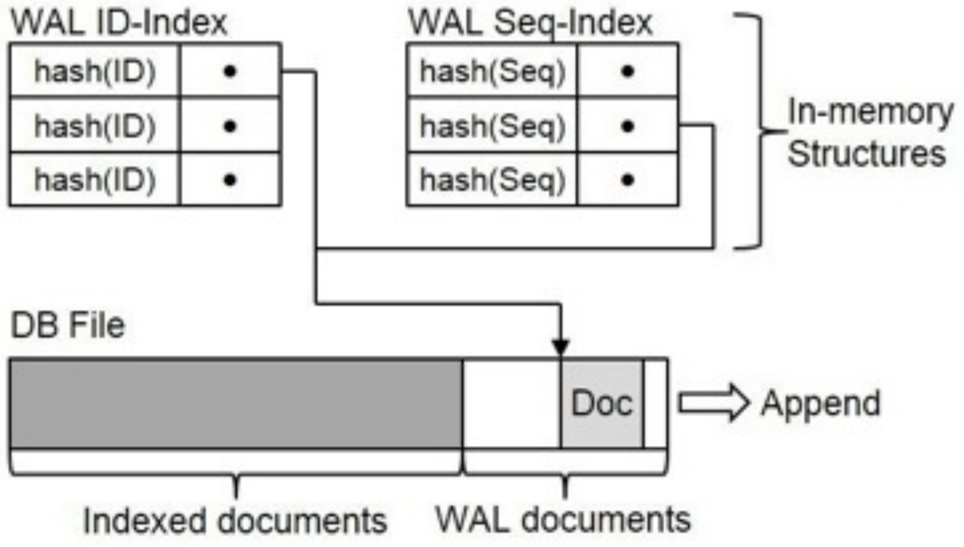


For extension, we first investigate the common prefix among the keys stored in the target leaf B +Tree. A new normal B+Tree using the first different chunk as key is created, and the documents are re-indexed by using the chunk. If there are keys sharing the same chunk, then we create a new leaf B+Tree for the chunk and the leaf B+Tree uses the rest of sub-strings just after the chunk as its key. As presented in the figure, a normal B+Tree using third chunk is created, and document 'aabb' and 'aac' are indexed by their third chunk 'b' and 'c'. Since 'aaaa' and 'aaabc' share the same third chunk 'a', we create a new leaf B+Tree and it indexes the documents using the rest of sub-strings 'a' and 'bc', respectively.

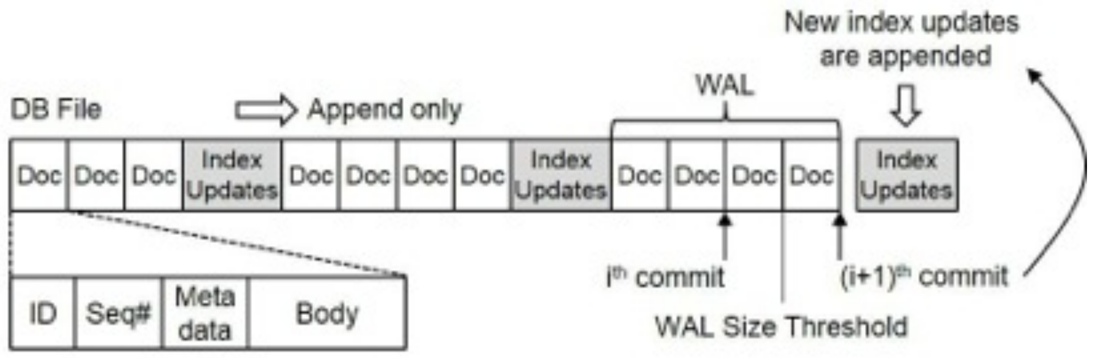
In this way, we can heuristically distinguish between the rarely occurring skewed patterns and hot spots.

Write Ahead Logging (WAL)

For maximizing the overall disk throughput, all data are sequentially written to DB file in a log-structured manner. To avoid random writes, we do not allow in-place update so that old document is not overwritten when there is a new update to the document. Whenever a document is updated, its new metadata and body are written at the end of DB file in an append-only manner. However, the index structures (i.e. ID-index and Seq-index) are not immediately updated to reduce the overall update cost, and the batch of these pended updates is eventually applied to the index structures later.

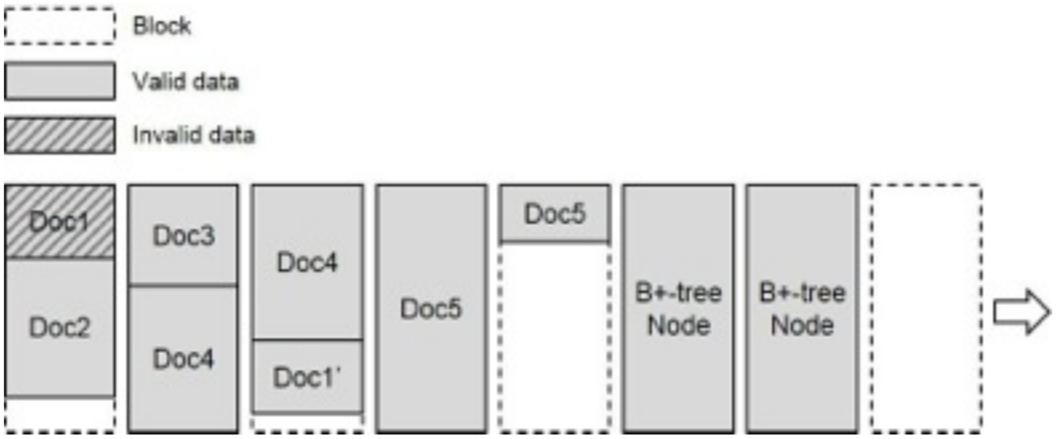


Instead of updating the main index structures (i.e. ID-index and Seq-index) in DB file, our module maintains separate ID-index (WAL ID-index) and Seq-index (WAL Seq-index) for indexing WAL documents that is not indexed by the main index structures. For fast indexing and retrieval, they are organized as hash table that maps from hash value of ID or sequence number of the document to the offset of DB file where the document is located. Since these index structures reside only in-memory, there is no further disk I/O overhead when we lookup WAL documents.

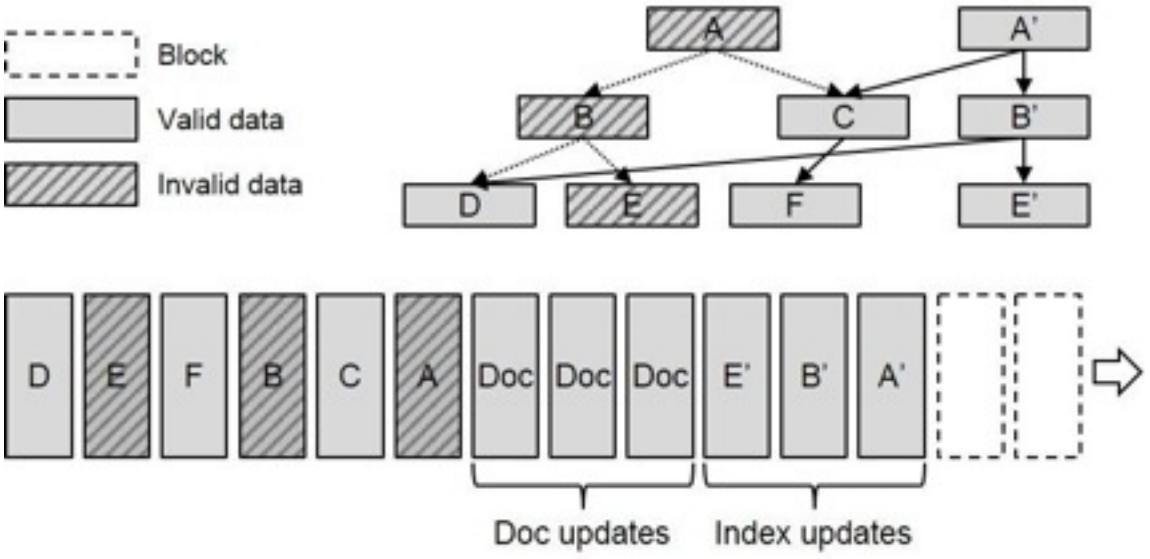


There is a threshold of the total size of WAL documents, and our module checks the WAL size for every commit operation. The batch of main index updates for WAL documents are aggregated and reflected to both ID-index and Seq-index if the total size of WAL documents is larger than the configured threshold. Updated nodes in main index structures are sequentially appended at the end of DB file, and all contents in WAL indexes are removed. The upcoming WAL documents will be appended at the end of DB file again.

File Layout



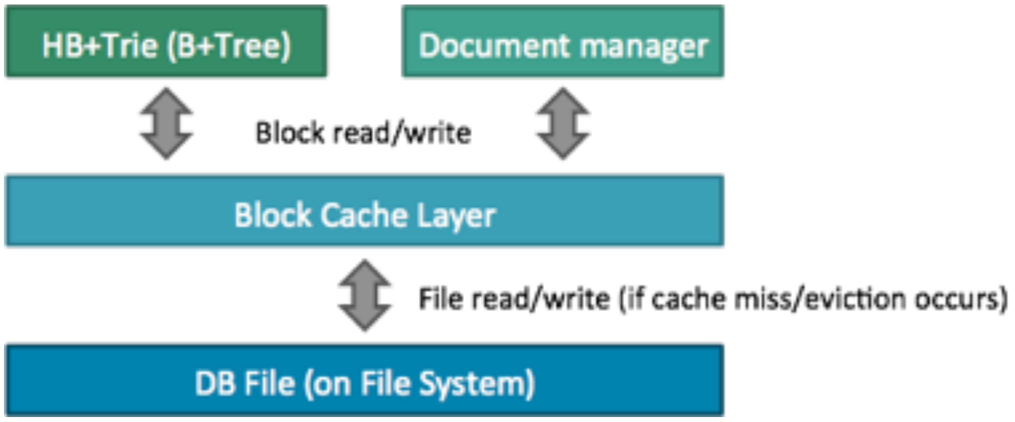
When a document is written in DB file, to avoid a document being written over several blocks, our module first checks the remaining space of current block. The document is simply appended if the remaining space is larger than the size of document. If not, the document is written at the first byte of next block skipping the space of current block. When the size of document is larger than a block size, we compare the total number of blocks to be occupied by this document when the document is written at the end of current position with when the document is written at beginning of next block. The module writes the document at the location that occupies less blocks.



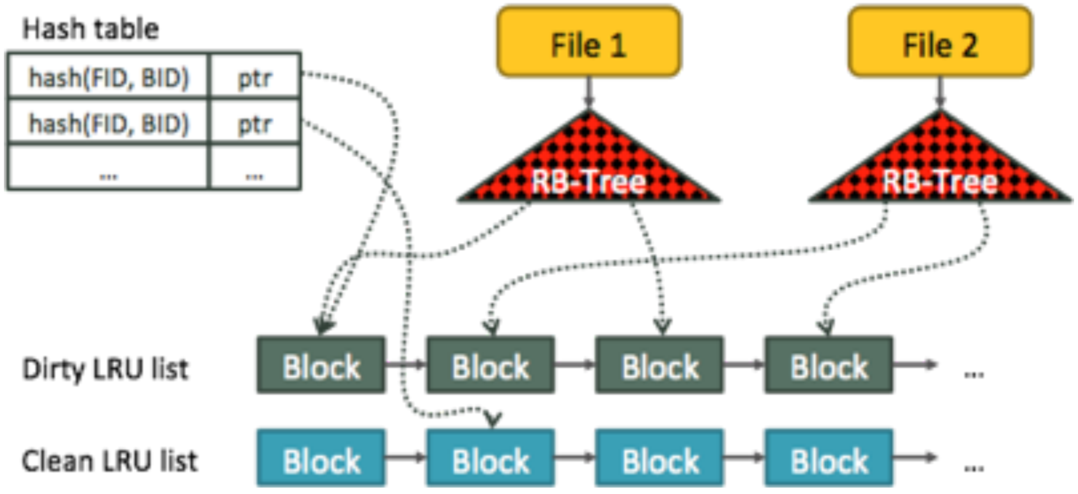
Unlike documents, each node in B+-tree is exactly fitted into a block, and they are always written at the beginning of new block to avoid interleaving with documents in the same block. Since our module does not allow in-place update, modified node is written at the end of file invalidating the old node. If an update occurs on a leaf node, its parent node should be modified and written in new block because the location of the leaf node is moved. For the same reason, all the nodes along the path from the updated leaf node to root node should be updated and moved to the end of file recursively. After writing all index updates, we finally write DB header block at the end of file.

Buffer Cache

We implement a separate global buffer cache to keep frequently accessed data (e.g. index nodes of B+-tree) in memory and not to evict them, and also provide an option for bypassing the OS file system cache through using Direct I/O flag.



This global buffer cache is a shared resource among writers and readers, and consists of two global LRU lists for clean and dirty blocks, respectively. We maintain a separate AVL-tree for each database file to keep track of the list of dirty blocks belonged to that file. All dirty blocks are sequentially written back to the file in an ascending order of their offset values by traversing the AVL-tree, which allows us to maximize the write throughput.
We also maintain a global hash table with a key {file_id, block_id} and a value {a pointer to an cache entry in either the clean or dirty list} for a fast lookup in the global buffer cache.



Compaction

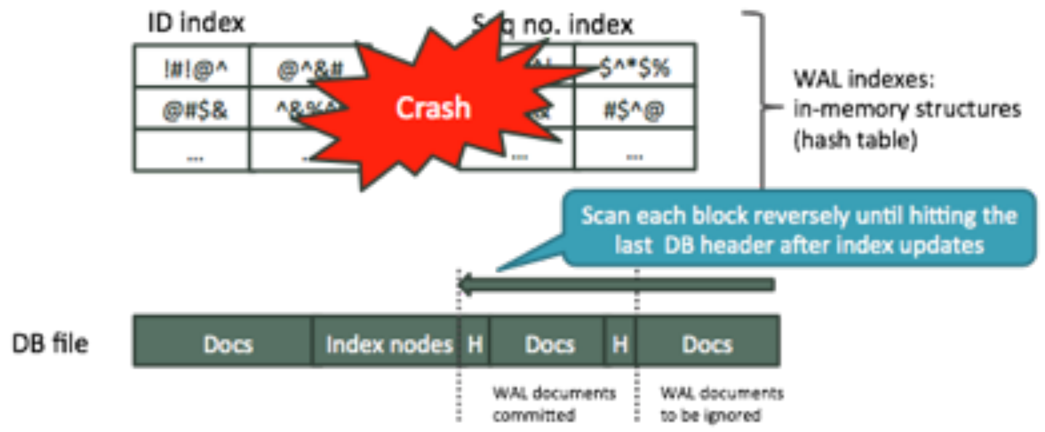
The compaction can be performed by either calling the compact public API manually or having a daemon thread manage the compaction automatically. When the compaction task starts, it scans a given ForestDB file to retrieve each (key, value) pair and then writes it into a new file to construct a new compacted HB+-Trie.

While the compaction task is running, a writer thread can still write dirty items into the WAL section of a new file, which allows the compaction thread to be interleaved with the writer thread. When the compaction task is completed, the writer thread can reflect the WAL items into the new file's HB+-Trie by flushing the in-memory WAL index entries.

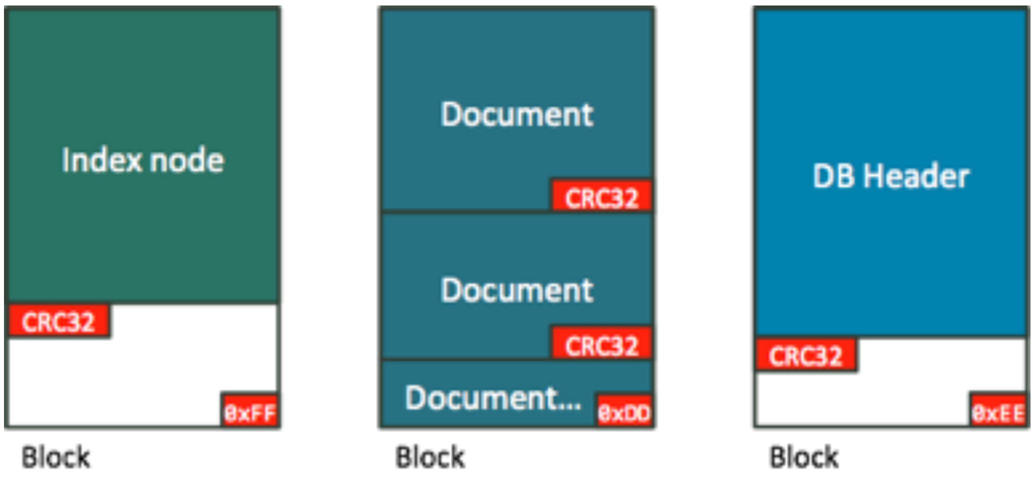
If the compaction task fails due to system crash or other reasons, we open the corrupted compact file and reverse-scan it to retrieve the list of WAL items written by the write thread, and finally move those items back to the original uncompacted file as part of the database open operation.

Recovery

We detect a database file corruption by reading the last block of the file. If the last block is B+-Tree index node, document or incorrect DB header, then the file is corrupted. In this case, we scan each block reversely to find the last valid DB header written after index updates, and then reconstruct WAL entries normally committed until the last header in the file.


ID index
Seq no. index
!#!@^
@^&#
*%
@#$&
#$^@
Crash
WAL indexes: in-memory structures (hash table)
Scan each block reversely until hitting the last DB header after index updates
DB file
Docs
Index nodes
H
Docs
H
Docs
WAL documents committed
WAL documents to be ignored

We also maintain a 4-byte CRC value for each B+-Tree index node, document, and DB header to detect a corruption in a database file. 1-byte block marker is added to the end of each block (4KB by default) to identify a block type (i.e., index node, document, or DB header) as shown in the following figure.



Remaining Tasks

Compaction Daemon (Sundar)

Compaction is currently performed by invoking the compact API manually. This seems reasonable in the ep-engine / cluster manager side because the compaction decision is made by considering various aspects. However, having a separate compaction daemon inside ForestDB can be useful in some use cases. This should be optional, which means that if the compaction daemon is enabled, the compact API should be disabled so that it is not invoked from the outside components.

Read-Only mode and Snapshot support (Sundar)

Allow for a read-only mode in forestdb where no writes are allowed and compaction never runs. A snapshot can be created explicitly by calling snapshot API and only allows us to perform read-only operations. A snapshot is destroyed if its corresponding database handle is closed.

Network byte ordering (Done)

ForestDB should support the endian-safe format, so that ForestDB files can be transferred and used across various platforms.

Disk access pattern optimizing for DGM (Done)

Flushing WAL and Compaction are currently performed by scanning documents in an index-specific order by traversing hash table or HB+Trie. This may not be a problem on large RAM, but extremely degrades the overall performance when working set size is greater than RAM size because disk accesses are performed in a random order. We have to sort the disk patterns by byte-offset.