HB+Trie

Thushjandan & François-Xavier

June 02, 2022

Data Management Data Structures

Agenda

1. Motivations
2. Overview
3. Implementation
4. Performance
5. Possible improvements
6. Discussion

Motivations

Motivations

Variable-length sized keys

Disadvantages of B+ trees or LSM-trees:

- Fanout degree decreases if key length increases
- Tree height grows for the same capacity to maintain
- Benefit of prefix B+ tree becomes limited for randomly distributed keys
- B+ tree nodes are randomly scattered on disk when it ages

Overview

Overview

HB+ trie stands for *Hierarchical B+ tree based trie*

Characteristics:

- Key space is divided into buckets. Every bucket has its own HB+ trie
- High disk throughput due to append-only disk layout
  - Regular compaction needed
- Disk updates are delayed with a **Write buffer index**

Overview

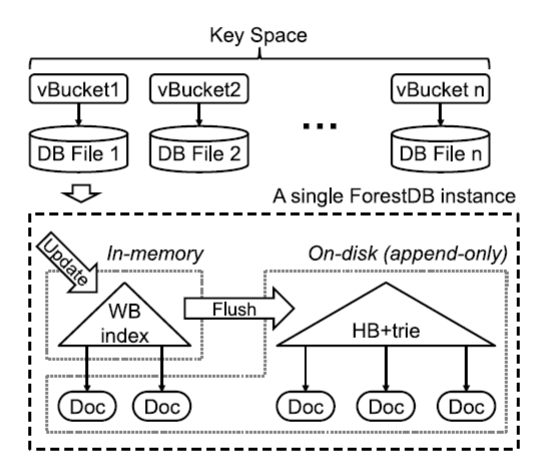


Figure 1: Architecture

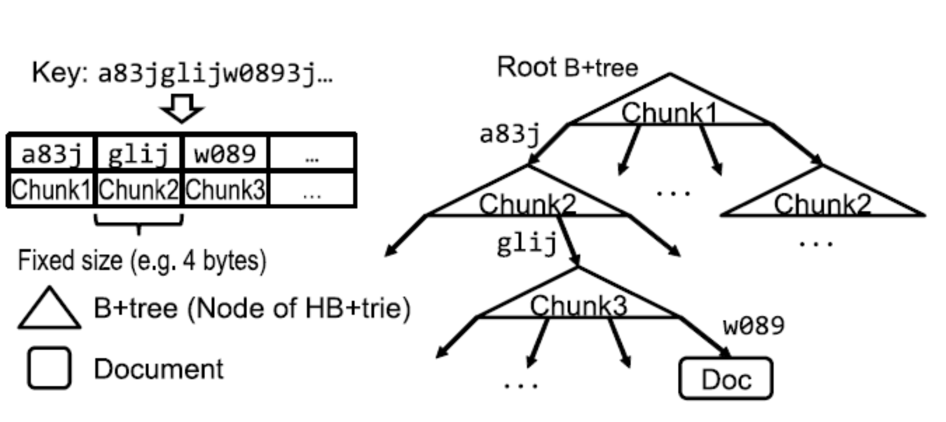
Overview

HB+ trie stands for *Hierarchical B+ tree based trie*

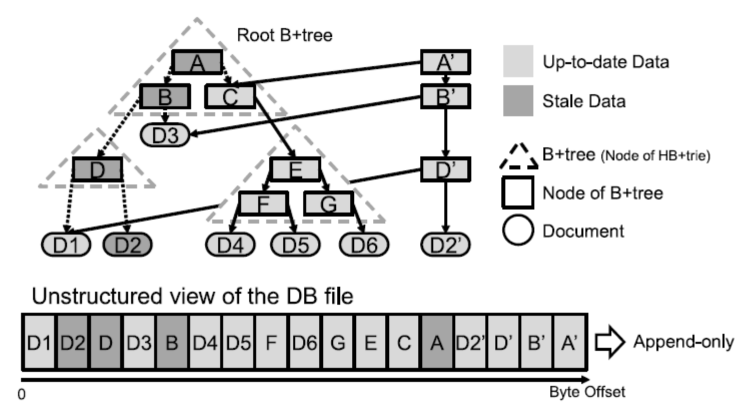
Characteristics:

- Key space is divided into buckets. Every bucket has its own HB+ trie
- High disk throughput due to append-only disk layout
- Disk updates are delayed with a **Write buffer index**
- Fixed size chunking of the key
- Every unique chunk has a dedicated B+ tree

Overview - Chunking



Overview - Disk layout

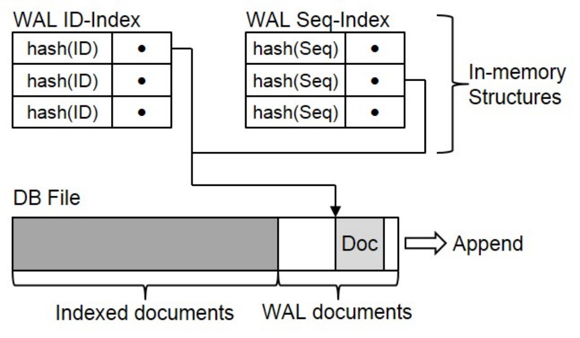


Overview - Write Buffer Index

Entries are added to the disk file directly

Write buffer index consists of a hashtable, where keys are hashed

Returns the offset in the file



Implementation

Implementation

- Using 16 byte chunks for keys
- Each page frame holds a complete B+ subtree.
- Storing pageId in the leaf to reference a B+ subtree

Implementation - Page frame

```go
type Page struct {
    Id uint64 // 8 byte
    Dirty bool // 1 byte
    // Previous page in the frame linked list
    prev *Page // 8 byte
    // Next page in the frame linked list
    next *Page // 8 byte
}

// Node is the unit of the B+ tree and is 3897 bytes long
type Node struct {
    *Page                               // 25 byte
    Next              uint64      // 8 byte
    Prev              uint64      // 8 byte
    Children          [120]uint64 // 960 byte
    Entries           [120]Entry  // 2880 byte
    NumberOfChildren uint64      // 8 byte
    NumberOfEntries  uint64      // 8 byte
}

type Entry struct {
    IsTree bool     // 1 byte
    Key    [16]byte // keys are chunks of 16 bytes
    Value  uint64   // values are pointers to subsequent b+ trees
}
```

Implementation - Chunking

```go
func createChunkFromKey(key []byte) (*[16]byte, *[]byte) {
    chunkedKey := [16]byte{}
    var trimmedKey []byte
    if len(key) > 16 {
        trimmedKey = make([]byte, 0, len(key)-16)
        // Chunked key of 16 bytes
        copy(chunkedKey[:], key[:16])
        // original key removed prefix
        trimmedKey = key[16:]
    } else {
        trimmedKey = make([]byte, 0, len(key))
        copy(chunkedKey[:], key[:])
        trimmedKey = key
    }
    return &chunkedKey, &trimmedKey
}
```

Implementation - Insert

```go
func (hbt *HBTrieInstance) insert(key []byte, value uint64, bpt *bptree.BPlusTree) error {
    chunkedKey, trimmedKey := createChunkFromKey(key)
    // If key is longer than 16 bytes
    if len(key) > 16 {
        subTree, err := hbt.createSubTree(bpt, *chunkedKey)
        if err != nil {
            return err
        }
        // Create recursively a new b+ tree instance
        return hbt.insert(*trimmedKey, value, subTree)
    } else {
        // Key is smaller than 16 bytes => create a leaf node.
        success, err := bpt.Insert(*chunkedKey, value)
        if success {
            return nil
        }
        return err
    }
}
```

Implementation - Search

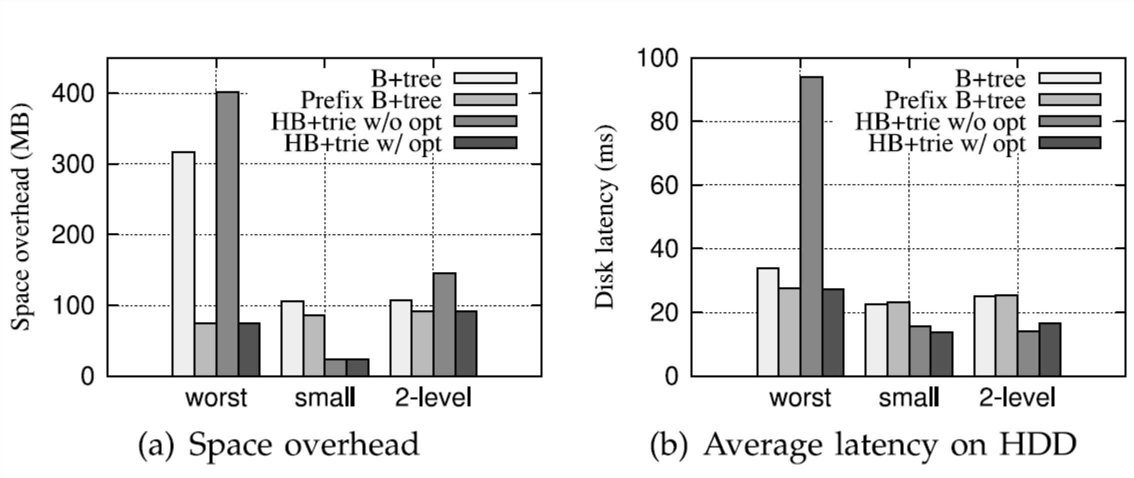
```go
// search recursively search for a key in the node and its children.
func (hbt *HBTrieInstance) search(bpt *bptree.BPlusTree, key []byte) (uint64, []byte,
                                                                    *bptree.BPlusTree, error) {
    chunkedKey, trimmedKey := createChunkFromKey(key)
    // Search in the root tree for the chunked key
    val, err := bpt.SearchTreeEntry(*chunkedKey)
    if err != nil {
        return 0, key, bpt, err
    }
    // Check if the leaf node is a pointer to a subtree.
    if val.IsTree {
        // Decode the frameId from the value field
        // Load b+ tree instance using the frameid
        subbpt := bptree.LoadBplusTree(hbt.pool, val.Value)
        // Call recursively search.
        return hbt.search(subbpt, *trimmedKey)
    } else {
        // it is a leaf entry
        return val.Value, key, bpt, nil
    }
}
```

Performance

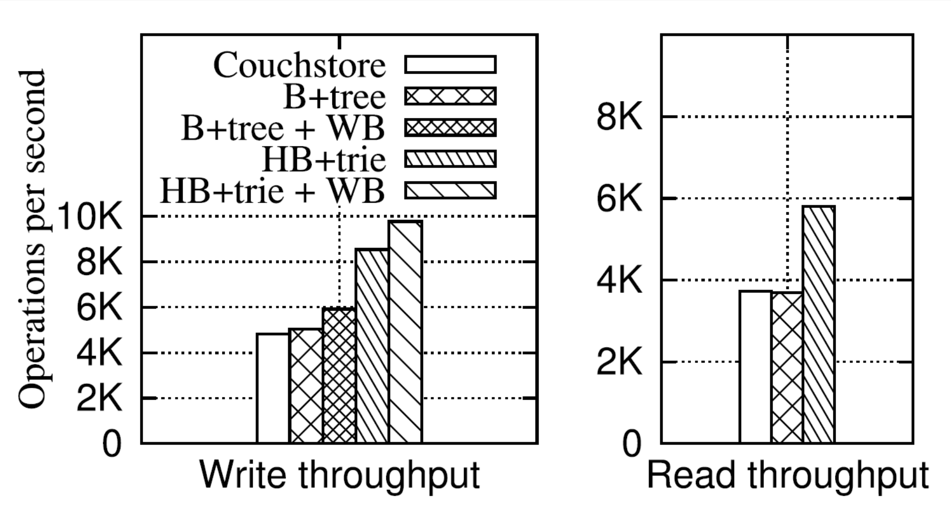
Performance

- **worst** - avg. 198 bytes key length
  - 20 levels of nested prefixes with each 2 branches
- **small** - avg. 65 bytes key length
  - 100 randomly generated prefixes
  - 10000 keys share a common prefix
- **2-level** - avg. 64 bytes key length
  - 2 levels of nested prefixes with each 192 branches

Performance



Performance



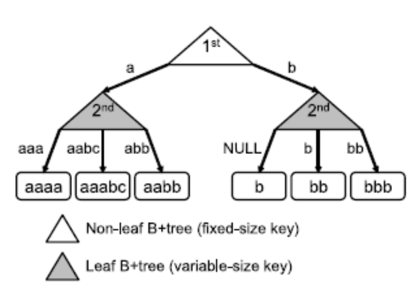
Possible improvements

Possible improvements

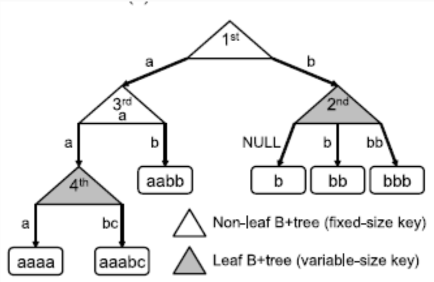
HB+ trie is not a balanced structure

- Leads to key skew under specific key pattern

To address this issue, Leaf B+ tree extension is proposed



(a) Without extension



(b) With extension

Possible improvements

- Performance of Range scans are bad in comparison with B+ tree.
- Write Buffer index improves the write throughput and lowers write amplifications

Discussion

Discussion

- Leaf Node Extension & Write buffer index are required
- Coachbase Server implements ForestDB storage engine (HB+ trie implementation)
- We would use HB+ trie for variable length keys as it combines the best properties of B+ trees and LSM trees