First Short Worst-Case Insertion Time Index with Asymptotically-Optimal Query Time

ABSTRACT

With the prevalence of online platforms, data is being generated and accessed by users at a very high rate. Besides, applications such as stock trading or high frequency trading require guaranteed low delays for performing database operations. It is consequential to design databases that guarantee data insertion and query at a consistently high rate without introducing any long delay during insertion. In this paper, we propose Nested B-trees (NB-trees), an index that achieves a consistently high insertion rate on large volumes of data, while providing asymptotically optimal query performance that is very efficient in practice. NB-trees support insertions at rates similar to LSM-trees, the state-of-the-art index for insertion-intensive workloads, while avoiding their long insertion delays and improving on their query performance. They approach the query performance of B-trees when complemented with Bloom filters. In our experiments, NB-trees had worst-case delays up to 1000 times smaller than LevelDB, RocksDB and bLSM, commonly used LSM-tree data-stores, performed queries more than 4 times faster than LevelDB and 1.5 times faster than bLSM and RocksDB, while matching them in terms of average insertion rate.

ACM Reference Format:

. 2020. First Short Worst-Case Insertion Time Index with Asymptotically-Optimal Query Time. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to the rapid growth of the data in a variety of applications such as banking/trading systems [63], social media [29] and user logs [24, 56], massive data comes in at a rapid rate and it is very important for a database system to handle *both* fast insertion and fast query. Facebook with more than 41,000 posts [34] and YouTube with more than 60,000 videos watched per second on average [23] record users' data through interaction with the system, while the data is also accessed by other users at the same time. Data is recorded at such a high rate and it is important to avoid long latencies to provide a smooth user experience. As users notice delays in the order of 100 milliseconds [17], such applications need to use storage systems that guarantee insertion delays smaller than that. In other applications, in Nasdaq Exchange 70,000 shares are traded per second [63], the storage system has to guarantee insertions at that rate, and delays of milliseconds are unacceptable in many trading scenarios such as high-frequency trading, a large component of the market [31] where stocks are traded by milliseconds [60]. Many companies use storage systems optimized for write-intensive applications that support fast queries [51].

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Requirements

We study indexes to achieve the following requirements.

(1) *Short Average Insertion Time Requirement* which is to handle a lot of insertions within a short period of time and is needed due to the rapid data growth nowadays. (2) *Short Maximum Insertion Time Requirement* which is a *stricter* requirement. It requires that each individual insertion has to be completed within a short period of time but the former requirement requires that the index could handle a collective set of insertions within a period of time, allowing some *individual* insertions to be completed with a *longer* delay. (3) *Short Average Query Time Requirement* which is to handle a lot of queries within a short period of time and is needed due to the rapid data access in some applications. (4) *Short Maximum Query Time Requirement* which is needed since it requires that each *individual* query could be answered in a short time. (5) *Theoretical Performance Guarantee Requirement* which is needed to know how good/bad an index is. Based on the first 4 requirements, we are interested in the time complexities of the following.

(a) The *amortized insertion time* (Requirement 1) of an insertion is the average time needed to handle a batch of insertions in the worst-case. (b) The *worst-case insertion time* (Requirement 2) of an insertion is the greatest insertion time of an insertion. (c) The *average query time* (Requirement 3) is the query time of a query on expectation. (d) The *worst-case query time* (Requirement 4) is the greatest query time of a query.

Similar to many recent studies [9, 12, 19–21, 58], we focus on when the data is stored in external memory (e.g., HDD or SSD). Data storage in main memory is volatile and more expensive than HDDs or SSDs. As pointed out in [19] and discussed in [40], main memory costs 2 orders of magnitude more than disk in terms of price per bit. Moreover, main memory consumes about 4 times more power per bit than disk [61]. Thus, designing high performance external memory indices that provide guarantees for real world applications can significantly reduce the cost of operations for many systems. On AWS, any machine with more than 100GB of main memory costs at least US$1 per hour but a machine with 15.25GB of main memory and 475GB SSD costs US$0.156 [3] (Linux machines, US East (Ohio) region). An SSD with 480GB capacity costs US$55 [5] while a 128GB DDR3L RAM module costs about US$393 [4].

1.2 Insufficiency of Existing Indices

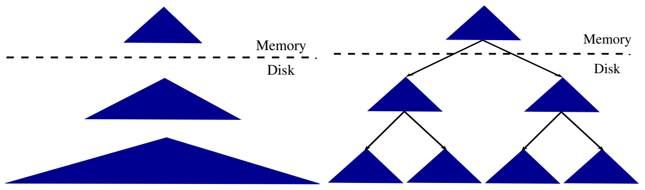
Existing indices do not satisfy the above requirements simultaneously. There are two major branches of indices related to our goal: (1) LSM-tree-like indices [38, 45, 53] and (2) B-tree-like indices [8, 11, 32] .

Consider the first branch. In recent years, LSM-trees [18, 20, 21, 39, 45, 53, 64] have attracted a lot of attention and are used as the standard index for insertion-intensive workloads in systems such as LevelDB[27], BigTable [13], HBase [1], RocksDB [25] (by Google and Facebook [1, 13, 25]), Cassandra [36] and Walnut [15]. LSM-trees buffer insertions in memory and merge them with on-disk components in bulk, creating sorted-runs on disk. Although LSM-trees satisfy the Short Average Insertion Time requirement, they do not satisfy Short Maximum Insertion Time requirement, Short Average/Maximum Query Time Requirement and Theoretical Performance Guarantee Requirement. This is because LSM-tree's worst-case insertion time is linear to data size [38, 58] and their worst-case query time is suboptimal [38]. In fact, in our experiments, although RocksDB [25], the industry standard and common research baseline [19–21], took an order of microseconds per insertion on average, it had its worst-case insertion time in the order of seconds. Such a worst-case insertion time is unacceptable for applications that requires reliability.

There are two major techniques to improve the performance of LSM-trees in the literature. The first technique is *Bloom filters*. They can improve average query time of LSM-trees [53], but their worst-case query time remains suboptimal. Thus, the LSM-trees with Bloom filters still do not satisfy Short Maximum Query Time Requirement. One representative is bLSM [53], a variant of LSM-tree that uses Bloom filters at each level. It also limits the number of LSM-tree levels. Setting the number of LSM-tree levels to a maximum allows for asymptotically optimal query time, but violates Short Average Insertion Time Requirement since the amortized insertion time becomes asymptotically larger than LSM-trees with an unrestricted number of levels. This is because the ratio of the size between LSM-tree components becomes unbounded, causing merge operations to read and rewrite a larger portion of the data. Furthermore, [53] provides methods to improve the worst-case query time of LSM-tree by a constant factor, but the worst-case insertion time remains linear to data size. Thus, they still do not satisfy Short Maximum Insertion Time Requirement. The second technique is *fractional cascading*. It improves the worst-case query time of LSM-trees [38], but their average query time remains high. Thus, LSM-trees with fractional cascading still do not satisfy Short Average Query Time Requirement. One representative is [38] that adds an extra pointer to each component of the LSM-tree, pointing to its next component. This pointer allows for reading one disk page per LSM-tree level. This was not compared in the experimental studies of LSM-trees [18, 20, 21, 53] due to its high average query time. Fractional Cascading and Bloom filters are incompatible [53] and cannot be used together.

Consider the second branch. Traditional B-trees [8] and B^+-trees [55] are among the most commonly used indices for good query performance. They provide optimal query performance and thus satisfy the Short Average/Maximum Query Time Requirement. They do not satisfy Short Average and Maximum Insertion Time Requirements, as they perform no buffering and perform at least one disk access for every insertion, which is very time-consuming.

Later, a write optimized variant of B-trees called B^ϵ-trees (also known as B-trees with Buffer) [11] were proposed, that reserves a portion of each node for a buffer. New data is inserted into the buffer of the root and moved down the levels of the tree whenever



(a) LSM-Tree **(b) Nested B-Tree**

Figure 1: NB-trees break down each level into constant sized B^+-trees and connect the keys in different levels.

the buffer becomes full. However, this method, although faster than B-trees, does not satisfy Short Average/Maximum Insertion Time Requirement. This is because B-tree nodes get scattered across the storage devices and moving this *small* buffer frequently down from a node requires accessing its children which is time consuming.

1.3 Our Index: NB-Tree

Motivated by this, in this paper, we propose the Nested B-tree (NB-tree) index. Fig. 1 shows the structure of an NB-tree compared with an LSM-tree. Intuitively, an NB-tree is a B-tree in which each node contains a B^+-tree. NB-trees can be seen as imposing a B-tree structure across the levels of an LSM-tree and breaking down each level into constant-sized B^+-trees. By imposing a B-tree structure, NB-trees establish a relationship between the keys in different components and provide an asymptotically optimal query cost, which nears the query performance of B-trees when complemented with Bloom filters. This design is based on the observation that different levels need to be connected to avoid suboptimal worst-case query time, which is lacking in the structure of LSM-trees. Although this is also the intuition behind LSM-tree with fractional cascading [38], [38] fails to design an index compatible with Bloom filters or with logarithmic worst-case insertion time.

Furthermore, the B-tree structure ensures that keys in each node only overlap with the keys in its children. This limits the impact of merge operations across levels, causing the merge operations to have the same cost on all the levels, and is used to provide a logarithmic worst-case insertion cost. In essence, the connection created between different levels allows us to bound the cost during the merge operation and provide a per-insertion account of the total insertion cost. Such a worst-case analysis is missing in most of the LSM-tree literature [19–21, 45] where the focus has been on amortized analysis and the few papers that have focused on worst-case performance provide an algorithm with worst-case that is linear to data size [38, 58].

Finally, NB-trees keep their nodes as large constant-size B^+-trees to perform mainly sequential I/O operations during insertions which minimizes seek time and allows them to perform insertions better than B-trees and their variants.

1.4 Contributions and Roadmap

Satisfying today's requirements. Our proposed NB-Tree is the first indexing structure that satisfies all the 5 requirements mentioned for indices on large volumes of data, as summarized in Table 1. Specifically, NB-trees give short average/maximum insertion time which is multiple factors smaller than B-trees and is similar to LSM-trees. They provide worst-case insertion time logarithmic to data size (unlike LSM-tree's linear worst-case insertion time) and multiple factors smaller than B-trees which satisfies the Short Maximum Insertion Time Requirement. They use Bloom filters to provide low average query time, while their structure allows for asymptotically optimal worst-case query time, satisfying Short Maximum Query Time Requirement. This, together with their logarithmic, yet better-than-B-trees, worst-case insertion time shows that NB-trees satisfy the Theoretical Performance Guarantee Requirement.

Optimality. NB-Tree is the first fast-insertion index with asymptotically-optimal query time. To the best of our knowledge, no existing index which could achieve this result.

Better performance in practice. In our experiments, NB-Tree's worst-case insertion time was more than 1000 times smaller than LevelDB, RocksDB and bLSM, three popular LSM-tree databases. They achieved average query time almost the same as B$^+$-trees, while performing insertions at least 10 times faster than them on average.

Organization. Sec. 2 discusses our terminology, Sec. 3 provides the design of the NB-tree, Sec. 4 discusses details on its implementation and analysis, Sec. 5 discusses the final version of NB-tree with better guarantees, Sec. 6 provides our experimental results, Sec. 7 discusses the relevant literature and Sec. 8 concludes the paper.

2 TERMINOLOGY AND SETTING

Problem Setting. Key-value pairs are to be stored in an index that supports insertions, queries, deletions and updates. The index is to be stored on an HDD or SSD and the term *disk* is used to broadly refer to the secondary storage device. We use an external memory computational model similar to [38, 42, 45, 53, 57]. The data is written or read from disk in pages of size $\mathcal{B}$ bytes. The index can use up to M pages of main memory. Transferring a contiguous block (i.e., a block where all of its pages are written/to be written next to each other) of one or more pages from disk to main memory (or vice versa) is done in two steps: firstly, a *seek* operation is performed, to find the location of the data on disk, and secondly, a *contiguous-page read/write* operation that transfers the data from/to disk. (1) A seek operation is assumed to take T_{seek} seconds, referred to as *seek time*, and (2) a *contiguous-page* read (write) operation takes $T_{con,R}$ ($T_{con,W}$) seconds/page. (T_{seek} is a constant in seconds, and $T_{con,R}$ and $T_{con,W}$ are constants in seconds/page). Note that seek time is the time difference between the starting time of the read/write request and the starting time of the data transfer. The *contiguous-page* read/write time is the time taken to transfer the data from the disk to the main memory or vice versa. To read (resp. write) a set of disk pages, the time to perform the operations is $\alpha \times T_{con,R} + \beta \times T_{seek}$ (resp. $\alpha \times T_{con,W} + \beta \times T_{seek}$) where α is the number of number of pages read (resp. written) and β is the number of seek operations performed. β depends on how the layout of pages on disk. If the pages are stored contiguously, $\beta = 1$ since only one seek is performed before all the pages are read/written.

In this computational model, we calculate the time it takes to read/write a set of disk pages from α and β. We emphasize that the notion of time used in our theoretical analysis is based on the computational model defined above, and there is an explicit relationship between the number of pages read/written, the number of seeks and the time to perform the operations, as discussed above.

In practice, contiguous-page access time is determined by a device's bandwidth while seek time depends on its internal mechanisms: on HDDs the movement of the disk arm and platter, and on SSDs the limitation of its electrical circuits. When a set of pages is accessed, contiguous-page access time is proportional to the size of the data transferred but seek time depends on how the data is stored, i.e., whether it is stored on contiguous blocks. It is important to account for seek time in our computational model as, per page, it can take much longer than contiguous-page access time. For instance, an HDD (7200rpm and 300MB/s bandwidth) based on the measurements in [52] has a seek time of 8.5 milliseconds and transfer rate of 125 MB/s. Reading a 4KB disk page incurs seek time of 8.5×10^{-3} seconds, but reading it only with contiguous-page read $\frac{4KB}{125MB/s} \approx 3 \times 10^{-5}$ seconds (283 times smaller than the seek time). We observed similar statistics in our experiments (See Sec. 6.1 for our measurement of seek and contiguous-page read/write time).

For the ease of discussion and as is the industry standard for common key-values stores such as RocksDB [26] and LevelDB [27], we consider the keys to be unique. Duplicate keys can be handled similar to B-trees [55] by using an extra bucket or a uniquifier attribute as discussed in [55].

Performance Metrics. For an operation on an index (e.g, an insertion or query on the index), we use the term *cost* only when referring to the *number of pages* accessed during the operation. We use the term *time* when referring to the time calculated based on our computational model, with seconds as its unit, during the operation. The time is composed of the contiguous-page access time and seek time for all disk accesses performed during the operation. Because a *time* measure takes into account seek operations, it is a more realistic measure of the real-life performance of an index compared with the *cost* measures.

Worst-case insertion time is the time, in seconds, it takes to insert an item into the index in the worst case. Moreover, *amortized insertion time* is the worst-case, over all positive integer n and possible sequences of n keys, of total insertion time of the n keys divided by n. *Worst-case query time* is the time an index takes to answer a query in the worst-case. *Average query time* is the expected value of the random variable denoting the query time of a random query key (*average* time is defined over one operation and is the expected time the operation takes while *amortized* time is defined over a sequence of operations, and is the average time an operation takes in the worst-case). We focus on point queries. The metrics are defined in terms of *time*, but their definition in terms of *cost* is analogous.

3 DESIGN OF NESTED B-TREE

Here, we describe a basic version of NB-trees. We provide the final version in Section 5. We use Fig. 2 for illustration.

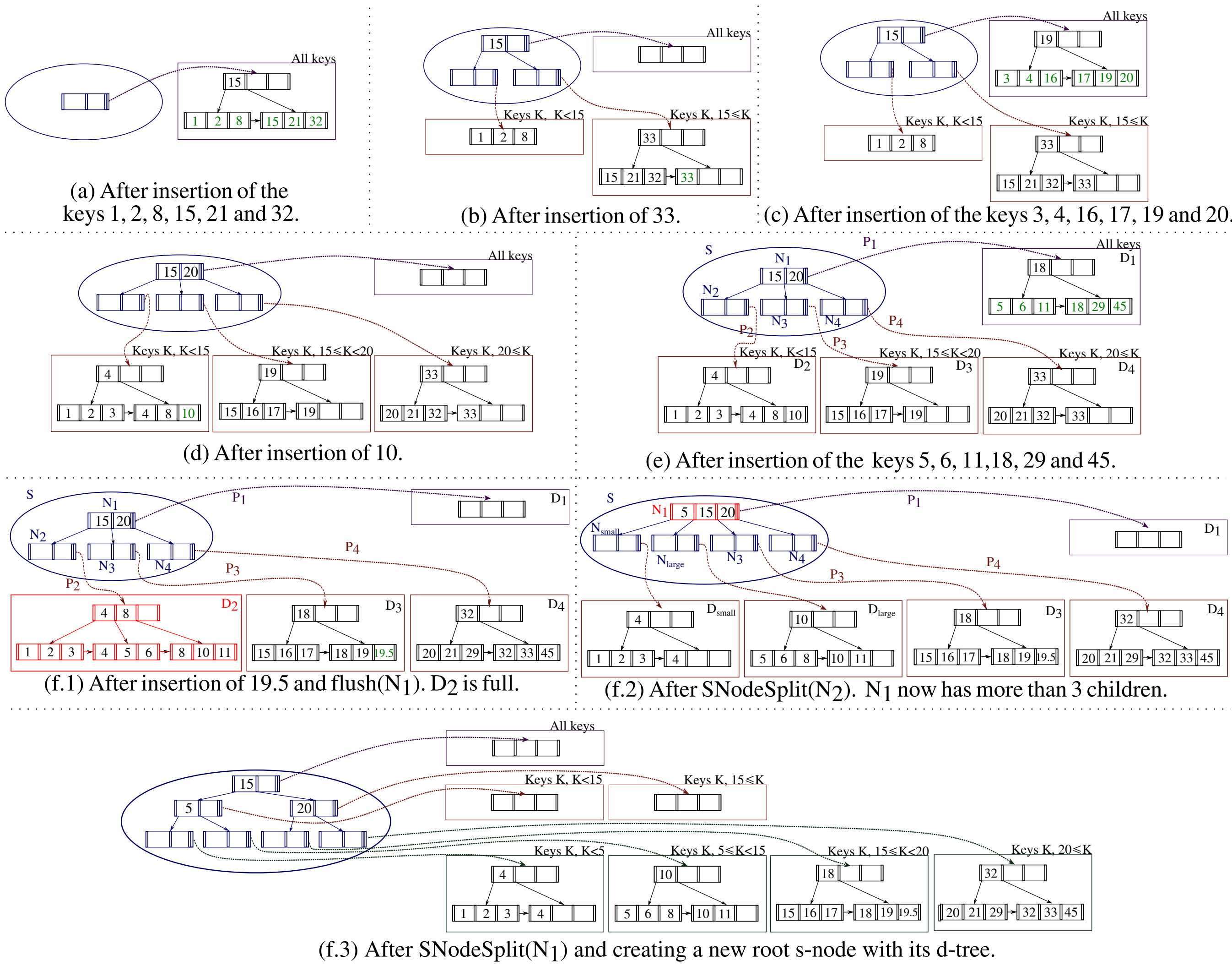
3.1 Overview, Definitions and Properties

An NB-tree is defined as a collection of several tree structures, $\{D_1, ..., D_k, S\}$ for an integer k.

D_1 to D_k are B$^+$-trees that each stores part of the data (i.e., key-value pairs) and is called a *data tree* or *d-tree* for short. Any key-value pair inserted into the index is stored in one of the d-trees, and the key-value pairs are moved between the d-trees throughout the life of an NB-tree. In Fig. 2 (e), D_1, D_2, D_3 and D_4 show four

Data Structure	Amortized Insertion Time	Worst-Case Insertion Time	Average Query Time	Worst-Case Query Time	Asymptotically Optimal Query Time
B-Tree [8] and B^+-Tree	Bad	Medium	Good	Good	Yes
B-Tree with Buffer [11]	Medium	Medium	Medium	Medium	Yes
LSM-Tree (no BF, no FC) [45]	Good	Bad	Bad	Bad	No
LSM-Tree (BF, no FC) LevelDB [27], RocksDB [26], Monkey [19]	Good	Bad	Medium	bad	No
LSM-Tree (no BF, FC) [38]	Good	Bad	Bad	Medium	Yes
NB-Tree (BF) [this paper]	Good	Good	Good	Medium	Yes

Table 1: Comparing NB-trees, LSM-trees and B-tree variants (BF: with Bloom filters, FC: with fractional cascading)



(a) After insertion of the keys 1, 2, 8, 15, 21 and 32.
(b) After insertion of 33.
(c) After insertion of the keys 3, 4, 16, 17, 19 and 20.
(d) After insertion of 10.
(e) After insertion of the keys 5, 6, 11,18, 29 and 45.
(f.1) After insertion of 19.5 and flush(N_1). D_2 is full.
(f.2) After SNodeSplit(N_2). N_1 now has more than 3 children.
(f.3) After SNodeSplit(N_1) and creating a new root s-node with its d-tree.

Figure 2: Insertions in an NB-Tree with parameters $\sigma = 6$ key-value pairs, $f = 3, B = 4$ (where the s-tree is enclosed in an ellipse and each d-tree is enclosed in a rectangle)

different data trees. They are all B^+-trees, i.e., at the leaf level each key is written next to its corresponding value (not shown in the figure). For ease of discussion, we refer to the nodes of a data tree as *data nodes* or *d-nodes* and to the keys in a d-node as *data keys* or *d-keys*.

S is a tree structure similar to a B-tree. S is used to establish a relationship between the keys in the d-trees, and impose a structure on the d-trees. Thus, S is called a *structural tree* or an *s-tree*. A structural tree is a B-tree with some modifications discussed later. In Fig. 2 (e), the ellipse labelled with S shows an s-tree. Similar to a B-tree, an s-tree contains several nodes. For ease of discussion, we refer to the nodes of a structural tree as *structural nodes* or *s-nodes* and to the keys in an s-node as *structural keys* or *s-keys*.

An s-tree differs from a B-tree in the following ways. (1) An s-tree does not store any key-value pairs. It only contains keys and pointers. Keys in an s-tree are not associated with a value. For this reason, we call it a structural tree (it only specifies a structure). (2) Each s-node, N, contains an extra pointer to the root d-node of a d-tree (which is a B$^+$-tree). We call this d-tree, N's d-tree (each d-tree is pointed to by exactly one s-node). The pointer in an s-node pointing to the root of its d-tree will be referred to as its *d-tree pointer.* In Fig. 2 (e), pointers P_1, P_2, P_3 and P_4 are d-tree pointers for s-nodes N_1, N_2, N_3 and N_4. (3) Leaf s-nodes only contain a d-tree pointer, and no keys or values. This is because an s-tree does not contain any data in its s-nodes. Since leaf s-nodes don't have any children, they do not contain any pointers or keys. In Fig. 2 (e), leaf s-nodes N_2, N_3 and N_4 do not contain any keys and only contain a d-tree pointer.

Specifically, non-leaf s-nodes in an s-tree are of the format $\langle P_{d-tree}, P_1, K_1, P_2, K_2, ..., P_r, K_r, P_{r+1}\rangle$ for an s-node with $r+1$ children. P_i for all i are pointers to the s-nodes in the next level of the s-tree, K_i are the corresponding s-keys and P_{d-tree} is a pointer to the d-tree of the s-node. s-keys are sorted in an s-node. For an s-key, K, in the s-node pointed to by P_i, $1 < i < r+1$, it is true that $K_{i-1} \leq K < K_i$, for $i = 1$, $K < K_i$ and for $i = r+1$, $K_{i-1} \leq K$. The only differences between a non-leaf s-node and a non-leaf B-tree node is that (1) an s-node has an extra pointer P_{d-tree} to the d-tree of the s-node and (2) s-keys are not associated with any value in the s-node. Moreover, a leaf s-node is of the format $\langle P_{d-tree}\rangle$, i.e., it only contains a d-tree pointer.

3.1.1 Properties. **Structural Properties**. The following properties are the structural properties of NB-trees.

S-tree Fanout. Each non-leaf s-node has at most f children and each non-leaf and non-root s-node has at least $\lceil \frac{f}{2} \rceil$ children. We call the parameter f the *s-tree fanout.* In Fig. 2, f is set to 3. Each s-node has at most 3 children, and non-leaf and non-root s-nodes must have at least 2 children.

D-tree Fanout. Each non-leaf d-node has at most B children and each non-leaf and non-root d-node has at least $\lceil \frac{B}{2} \rceil$ children. We call the parameter B the *d-tree fanout.* In Fig. 2, B is set to be 4. Each d-node has at most 4 children, and non-leaf and non-root s-nodes must have at least 2 children.

D-tree Size. For a parameter σ, each d-tree is at most of size σ. D-trees of leaf but not root s-nodes are at least of size $\lceil \frac{\sigma}{2} \rceil$. σ can be specified by the number of bytes used by the d-tree or the number of key-value pairs in the d-tree. The analysis in the paper uses the latter for ease of notation, while the former is used in experiments as its easier to specify in practice. Unless stated otherwise, σ refers to the number of key-value pairs in a d-tree (i.e., number of d-keys in the leaf level). In Fig. 2, σ is set to 6. Each d-tree contains up to 6 keys in their leaves, and d-tree of leaf but not root s-nodes contain at least 3 d-keys in their leaves.

Cross-s-node Linkage Property. This property of NB-trees establishes the relationship between the s-keys in an s-node and the d-keys and s-keys of the s-node's children. Consider an s-node, N with r s-keys. The s-node is of the form $\langle P_{d-tree}, P_1, K_1, P_2, K_2, ..., P_r, K_r, P_{r+1}\rangle$, where the s-keys are in a sorted order (i.e., for each $i \in [1, r), K_i < K_{i+1}$). For each i, $1 < i < r+1$, consider the child s-node, C_i pointed to by P_i. For an s-key in C_i or a d-key in C_i's d-tree, K_{C_i}, it holds that $K_{i-1} \leq K_{C_i} < K_i$. For $i = 1$, it holds that $K_{C_i} < K_i$ and for $i = r+1$, $K_{C_i} \geq K_{i-1}$. In Fig. 2 (e), the d-keys in D_2 are less than 15, the d-keys in D_3 are at least 15 but less than 20 and the d-keys in D_4 are at least 20. In Fig. 2, d-trees are labelled with a possible range that is derived based on this property. The possible range of d-keys for s-nodes in the same level is non-overlapping and covers the entire key space.

3.2 Operations

3.2.1 Insertions. The insertion of a key-value pair (K, V) in an NB-tree starts by inserting the pair in the d-tree of the root s-node and recursively moving the pair down the tree to ensure that the properties mentioned in Section 3.1 are satisfied. We refer to a d-tree as *full* if it has more than σ key-value pairs. In this section, we provide a conceptual discussion on how insertions are performed, and how they are implemented in practice is discussed in Section 4.1.

Intuitively, the d-tree of each s-node can be seen as a storage space for the s-node. The key-value pairs are stored in the d-tree of each s-node. When d-tree of an s-node N becomes *full*, the pairs are distributed down to the d-tree of the children of N based on the N's s-keys such that the Cross-s-node Linkage Property is satisfied. This continues until the d-tree, D of a leaf s-node, N becomes full, in which case D and N are split into two and the median of d-keys in D (i.e., the d-key, K, in D such that half of the d-keys in D are less than K) is inserted into the parent, P, of N. If P now has more than f children, P and it's d-tree are similarly split into two. The splitting may continue until the root of the s-tree, which may result in an increase in the height of the tree. More specifically, insertion works as follows. In this section, we describe that the size of a d-tree temporarily surpasses σ. This only happens *conceptually* and is said only for the ease of disposition. In Sec. 4.1, we discuss that non-root d-trees are not modified in-place (i.e., to insert a new key in a d-tree on disk, the entire d-tree is copied and written, together with the new key, in a new disk location. The d-tree pointers are only updated after we ensure that the new d-tree satisfies the size requirements), and size requirements are never violated. For instance in Fig. 2, by not modifying d-trees in-place, the NB-tree goes directly from Fig. 2 (e) to Fig. 2 (f.3) on the *disk*, i.e., Fig. 2 (f.1) and Fig. 2 (f.2) are *never materialized.*

Insertion Operation. A new key-value pair (K, V) is always inserted into the d-tree, D, of the root s-node, N. We insert (K, V) in D using a B$^+$-tree insertion mechanism. If D has up to σ d-keys, the insertion is finished. Otherwise, we need to ensure *d-tree size requirement* is satisfied. For this, we call $HandleFullSNode(N)$ (described later).

Example. In Fig. 2 (a), insertion of key-value pairs is done in the d-tree of the root s-node. Fig. 2 (a) shows the result of inserting keys 1, 2, 8 ,15, 21 and 32. They are all inserted into the d-tree of the root s-node. Now, inserting a new key (e.g., 33) in the d-tree of the root s-node causes the d-tree to become full and $HandleFullSNode$ is called on the root s-node to restore compliance to the *d-tree size requirement.* Fig. 2(b) shows the result after calling $HandleFullSNode$ and all the properties discussed in Section 3.1 are satisfied.

HandleFullSNode Operation. $HandleFullSNode(N)$ is called to restore compliance to *d-tree size requirement* when the size of a d-tree, D, of an s-node N surpasses σ. It acts differently when N is a leaf s-node and when it is not.

N is a leaf s-node. $SNodeSplit(N)$ (detailed later) is called which splits N into two s-node N_{small} and N_{large} and returns the median d-key, K_M, of the d-keys in D together with pointers P_{small} and P_{large} to N_{small} and N_{large}. Then $HandleFullSNode(N)$ inserts K_M, P_{small} and P_{large} into the parent s-node of N and returns (similar to the insertion of the median into a parent node of a B-tree after the node splits). If N is a root s-node, $HandleFullSNode(N)$ creates a new root s-node and then inserts K_M, P_{small} and P_{large} into this new root (s-tree's height increases by one).

Example. Consider Fig. 2 (a). *HandleFullSNode* splits the d-tree and the s-node into two, one d-tree containing the smaller half and another the larger half of d-keys (seen at the leaf level of Fig. 2 (b)). *HandleFullSNode* also creates a new root s-node and inserts the median of d-keys into it.

N is not a leaf s-node. $HandleFullSNode(N)$ first calls $flush(N)$ operation. $flush(N)$ (detailed later) removes the keys-value pairs from D and inserts them into the d-trees of N's children. After that, for any child s-node C of N, if C's d-tree is now full, $HandleFullSNode(N)$ calls $HandleFullSNode(C)$ recursively. If d-tree of none of the children is full, $HandleFullSNode(N)$ returns.

If N has k children, there can be up to k recursive calls. A recursive call $HandleFullSNode(C)$ may split C into two s-nodes, which increases the total number of children of N. Therefore, if the number of children of N becomes larger than f, $HandleFullSNode(N)$ calls $SNodeSplit(N)$ which splits N into s-nodes N_{small} and N_{large} and returns the median s-key K_M of N together with pointers P_{small} and P_{large} to N_{small} and N_{large}. Then $HandleFullSNode(N)$ inserts K_M, P_{small} and P_{large} into the parent s-node of N and returns (similar to the insertion of the median into a parent node of a B-tree after the node splits). If N is a root s-node, i.e., has no parent, $HandleFullSNode(N)$ creates a new root s-node and then inserts K_M, P_{small} and P_{large} into the new root (s-tree's height increases by one).

Example. Consider Fig. 2 (e). $HandleFullSNode(N_1)$ calls $flush(N_1)$ which moves d-keys from d-tree of the root s-node N_1 to its children. Fig. 2 (f.1) shows the result. Consequently N_2's d-tree becomes full (has more than 6 keys). Thus, $HandleFullSNode(N_1)$ calls itself recursively, i.e., $HandleFullSNode(N_2)$. In the recursive call, since N_2 is a leaf s-node, $HandleFullSNode(N_2)$ calls $SNodeSplit(N_2)$ which splits N_2 into two. It inserts the median d-key into N_1. Now N_1 has more than two s-keys (Fig. 2 (f.2)). $HandleFullSNode(N_1)$ calls $SNodeSplit(N_1)$ which splits N_1 into two, creates a parent for N_1 and inserts N_1's median, 15, into N_1's parent (Fig. 2 (f.3)).

SNodeSplit. $SNodeSplit(N)$ splits an s-node N and its d-tree D into two. If N is a leaf s-node, let K_M be the median d-key of D. If N is not a leaf s-node, let K_M be the median s-key of N. $SNodeSplit(N)$ creates two s-nodes N_{small} and N_{large} with corresponding d-trees D_{small} and D_{large}. It inserts all the d-keys in D less than K_M in D_{small} and the d-keys at least K_M in D_{large} and inserts all s-keys in N less than K_M in N_{small} and the s-keys at least K_M in N_{large}.

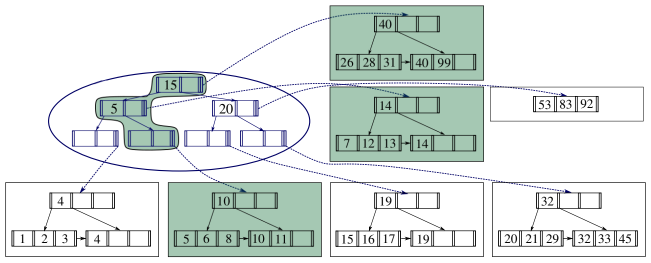


Figure 3: Querying for the key 11 in an NB-Tree. The shaded area shows the part of the tree read during the query.

Let P_{small} be a pointer to N_{small} and P_{large} a pointer to N_{large}. The operation returns $(K_M, P_{small}, P_{large})$.

Example. See Fig. 2 (f.1) to (f.2) and Fig. 2 (f.2) to (f.3).

Flush. $flush(N)$ is called on a non-leaf s-node, N. Intuitively, $flush(N)$ distributes the d-keys in the d-tree of N to the d-tree of its children based on the s-keys of N. Let N contain r s-keys and be of the format $\langle P_{d-tree}, P_1, K_1, P-2, K_2, ..., P_r, K_r, P_{r+1}\rangle$. Let D denote the d-tree of N, pointed to by P_{d-tree}. Furthermore, let C_i be the s-node pointed to by P_i and let D_i be the d-tree of C_i. For every key K in D, we remove it from D and insert it into D_i if $K_{i-1} \leq K < K_i$. We insert K into D_1 if $K < K_1$ and in D_{r+1} if $K \geq K_r$.

Example. See Fig. 2 (e) to Fig. 2 (f.1).

3.2.2 *Update and Deletion.*

Similar to LSM-trees [45], we perform deletion and update by inserting a new record that indicates the modification. To update key k to have value v' instead of its current value v, (k, v') is inserted into the tree using our insertion algorithm. To remove key k, a *tombstone* record, containing k with a *tombstone* bit set to 1, is inserted into the tree. In both cases, if k exists in the root s-node, it is modified or removed. Additionally, to handle update and deletion, $flush(N)$ is modified as follows. If the pair (k, v') exists in N and the pair (k, v) exists in a child, C, of N, $flush(N)$ inserts (k, v') into C and (k, v) is discarded. If the tombstone record is in N and (k, v) in C, (k, v) is discarded and the tombstone record is inserted into C, except when C is a leaf s-node (in which case both the tombstone record and (k, v) are discarded). We keep the tombstone until it reaches the leaf s-node to ensure all the potential update to k are removed from the tree. Since tombstones are discarded at the leaf level, and $flush(N)$ ensures at most one copy of k exists per level, update and deletion do not affect the size of leaf d-trees. Consequently, they do not affect the height of the s-tree, and thus our analysis of query and insertion performance remains the same. The cases when s-nodes become *underfull* as a result of deletion can also be handled using a mechanism similar to deletion in B-trees [55]. We will not provide the details since our focus is on write-intensive workloads, and we assume that deletions are infrequent and such cases can be ignored.

3.2.3 *Queries.*

NB-trees perform queries similar to B-trees. However, in a B-tree, a query traverses the B-tree based on the keys in the nodes. In an NB-tree, a query traverses the s-tree based on the s-keys in the s-nodes. Furthermore, in a B-tree, a query only searches the keys in the nodes visited. However, an NB-tree searches the s-keys in the s-nodes visited and also searches the corresponding d-tree of each s-node visited. Searching a d-tree is exactly a B$^+$-tree search. Fig. 3 shows the query of key 11 on an NB-tree.

4 IMPLEMENTATION AND ANALYSIS

To allow for fast performance, similar to LSM-trees, the d-tree corresponding to the root s-node is kept in memory. The rest of the d-trees are stored on disk.

4.1 Insertion Implementation

Manipulations of the s-tree is straight forward. Here we focus on operations impacting on-disk d-trees. *Insert* and *HandleFullSNode* do not make any modifications to the on-disk d-tree themselves. Modifications are done through *flush* and *SNodeSplit* operations, so we focus on them.

To minimize the insertion time, we aim at minimizing the number of seek operations by performing our mainly contiguous-page accesses. To this end, we maintain the following invariants. Firstly, all the d-nodes in a d-tree are written in contiguous pages and can be retrieved by a contiguous-page scan from the first node written. Secondly, the leaf d-nodes are written on disk in a sorted order. Thus, a contiguous-page scan of a d-tree from the first leaf d-node until the last d-node reads all the key-value pair written in the d-tree in an ascending order.

Flush(N). Assume that N contains $r+1$ children, C_i with respective d-trees D_i, $1 \leq i \leq r+1$ and r keys $\langle K_1, K_2, ..., K_r \rangle$. *flush* starts by scanning D and D_1, merge-sorting them together (contiguous-page scan of D and D_1 retrieves their keys in a sorted order) and writing the output, D'_1 in a new disk location. Note that the two invariants mentioned above now hold for D'_1. From D, we only merge-sort the d-keys that are less than K_1 with D_1. We follow the same procedure and in general merge-sort the d-keys, K such that $K_{i-1} \leq K < K_i$, from D with D_i for $1 < i < r$. For $i = r+1$, we merge-sort the d-keys, K such that $K_i \leq K$, from D with D_{r+1} and for $i = 1$ d-keys, K, such that $K_i > K$. We move down only the first σ d-keys from D if it has more. This is to avoid the size of the full d-trees in deeper levels of the tree getting progressively larger as a result of recursive *flush* calls. Because some of the d-keys may remain in a d-tree, we re-write D starting from the $(\sigma+1)$-th d-key and thus removing the d-keys that were flushed down from D.

The cost of *flush* is $O(\frac{\sigma f}{B})$. Assuming the main memory has enough space to buffer $\Omega(\sigma)$ key-value pairs (to buffer a constant fraction of the parent's d-tree and the d-tree of one child at a time) which is typically in the order of 100MB, *flush* performs a constant number of seek operations for merge-sorting N with each child and thus $O(f)$ seek operations in total. The number of seek operations increases proportionately if there is less space available in memory.

SNodeSplit(N) The *SNodeSplit*(N) operation only performs disk accesses when dividing a d-tree into two. For this, we scan a d-tree and contiguously write it as two d-trees. It costs $O(\frac{\sigma}{B})$ page accesses and $O(1)$ number of seek operations under the same conditions as above. This operation preserves the two invariants mentioned above.

4.2 Analysis

Correctness. Correctness of insertions is shown below.

THEOREM 1. *Performing any sequence of insertions on an NB-tree for which properties in Sec. 3.1.1 hold will result in an NB-tree for which properties in Sec. 3.1.1 hold.*

Proof sketch. Induction on the number of insertion operations shows that the cross-s-node linkage and structural properties are preserved using the insertion algorithm. To see this, note that non-root d-trees and s-keys are only modified through *flush* and *HandleFullSNode* operations, and these two operations by definition preserve cross-s-node linkage and structural properties (See Section A of our technical report[6] for a complete proof). □

The correctness of the query operation follows from the cross-s-node linkage property, and the correctness of updates and deletions follow from the correctness of insertions.

Insertion Time Complexity. There are at most $O(\frac{n}{\sigma})$ *HandleFullSNode* function calls on any level because in the worst case all the keys are moved down to the leaf level and each *flush* moves σ keys. *HandleFullSNode*, excluding the recursive call, requires $O(\frac{f\sigma}{B})$ page accesses for *flush* and *SNodeSplit*. Each operation can be handled with $O(f)$ seek operations. Since the height of the s-tree is $O(\log_f \frac{n}{\sigma})$, the amortized insertion time is $O(\frac{f}{B}\log_f(\frac{n}{\sigma}))(T_{con,W} + T_{con,R}) + O(\frac{f}{\sigma}\log_f(\frac{n}{\sigma}))T_{seek}$. We only modify an s-node if its corresponding d-tree is modified. Thus, assuming each s-node fits in a disk page (f is typically much smaller than B) s-tree manipulations add at most one page write after writing each d-tree, which does not impact the complexity of the operations.

For this version of NB-tree, the worst-case insertion time is linear in n because all the s-nodes may be full at the same time. In Section 5 we introduce a few modifications that reduces the worst-case insertion time to logarithmic in n.

Query Time Complexity. In the worst case, the query will search one s-node in each level of the s-tree. The height of each d-tree is $O(\log_B \sigma)$ and height of the s-tree is $O(\log_f \frac{n}{\sigma})$, thus, the query takes time $O(\log_B \sigma \log_f \frac{n}{\sigma}) \times (T_{seek} + T_{con,R})$. Observe that the query cost of NB-trees is asymptotically optimal. That is, it is within the constant factor $\log_B \sigma$ of minimum number of pages accesses required to answer a query. Note that in-memory caching, to cache a number of levels of each d-tree can be used to reduce query time by a constant factor, similar to B-trees.

5 ADVANCED NB-TREE

We discuss modifications to NB-tree design to reduce the worst-case insertion time from linear in n to logarithmic in n and how to add Bloom filters to NB-trees to enhance their query performance. The version provided here is to be considered the final NB-tree index.

5.1 Modification

We make the following changes to the structural properties of NB-tree. For non-leaf s-nodes, we remove the requirement on the maximum size of its d-tree being σ, and instead put a requirement on the total number of key-value pairs in the d-trees of all sibling s-nodes to be $f(\sigma+1)$ (each s-node can still have up to $f-1$ keys).

Single Recursive Call. All the operations work the same as before, but with one difference. In *HandleFullSNode*(N), after calling *flush*(N), if any s-node is oversized, *HandleFullSNode* will be called recursively on exactly one s-node that has the largest size (i.e.

arg max $|C|$), instead of performing a recursive call for every full s-node. The rest of the operations work as before, but now there is at most one recursive call during *HandleFullSNode* operation.

The above insertion procedure remains correct and satisfies the new requirement on the maximum number of key-value pairs in d-trees of non-leaf sibling s-nodes. This is because each level receives σ keys and flushes down σ keys (see Section 4.1) if any of the d-trees of sibling s-nodes have more than σ keys, and the requirement is already satisfied if none of the siblings has more than σ keys. For leaf s-nodes we still perform splits if their size surpasses σ keys. Thus, we can observe that the total size of siblings is at most $f \times \sigma$.

Deamortization. We use a deamortization technique similar to [38]. If *HandleFullSNode* is not called in an insertion, it is done as before. If *HandleFullSNode* is called on the root, we first calculate an upper bound, or the exact time, T seconds, needed to finish the *HandleFullSNode* operation. Then, *HandleFullSNode* is called in the background using another thread, and each new insertion is forced to take $\frac{T}{\sigma}$ seconds. This way, after σ insertions (which is the number of insertions that can be stored in memory), *HandleFullSNode* finishes. In-memory insertions are performed while the on-disk operations happen in the background, and the on-disk operations finish when memory is full and needs to be flushed down. The upper-bound can be calculated as $O(\frac{f\sigma}{B}\log_f \frac{n}{\sigma})(T_{con,W} + T_{con,R}) + O(f \log_f \frac{n}{\sigma})T_{seek}$ (shown below) or the exact total time can be calculated by by knowing the size of each s-node (we keep track of s-node sizes and store them in memory) and using T_{seek}, $T_{con,R}$ and $T_{con,W}$ (details of the implementation can be found in Sec. B of our technical report [6]). Calculating the exact total time improves the performance at a small book-keeping cost.

Insertion Time Complexity. An insertion operation performs at most one *HandleFullSNode* function call at each level of the s-tree, resulting in at most $O(\log_f \frac{n}{\sigma})$ number of *HandleFullSNode* calls. Each $flush$ and *SNodeSplit* step take $O(\frac{f\sigma}{B})$ I/O operations. Thus, the total time taken for one insertion call is $O(\frac{f\sigma}{B}\log_f \frac{n}{\sigma})(T_{con,W} + T_{con,R}) + O(f \log_f \frac{n}{\sigma})T_{seek}$. Deamortization reduces the time by a factor of σ and we can achieve the worst-case insertion time $O(\frac{f}{B}\log_f \frac{n}{\sigma})(T_{con,W} + T_{con,R}) + O(\frac{f}{\sigma}\log_f \frac{n}{\sigma})T_{seek}$. The amortized insertion time in this case is the same as the worst-case insertion time.

Query Time Complexity. Maximum size of an s-node is $f(\sigma + 1)$ since that is the maximum total size of sibling s-nodes together. Thus, the query cost is $O(\log_B(f\sigma) \times (\log_f(\frac{n}{\sigma})))$ based on an analysis similar to Section 3.2.3, but by changing the maximum size of an s-node. This is $O((\log_f(\sigma) + 1) \times (\log_B(\frac{n}{\sigma})))$ and is asymptotically optimal.

5.2 Bloom Filter

We use Bloom filters to enhance the average query cost. A Bloom filter uses k bits per key and h hash functions to decide whether a key exists in a data structure. When searching for a key, if the Bloom filter returns negative, the key definitely does not exist in the data structure. When it returns positive, the key may not exist in the data structure with a probability dependant on k and h (e.g., $k = 8$ and $h = 3$ results in a false positive probability of less than 5%).

Algorithm	Memory allocation
`B+-tree (bulk)`	Non-leaf nodes (0.989)
`leveldb-tuned`	Cache (4), mem-table (1.5 × 2)
`rocksdb-tuned`	Cache (2), mem-table (2), index and bloom filter (3.31)
`bLSM`	Mem. comp. (6), cache (1.5)
`NB-Tree`	Cache (2), in mem. s-node (2), index and bloom filter (3.31)

Table 2: Memory content and corresponding size in GB

We use a Bloom filter for d-tree of each s-node. We need to create/modify the Bloom filters during $flush$ or *SNodeSplit* operations. For children s-nodes in $flush$ and all the s-nodes in *SNodeSplit*, as we create a new d-tree for the s-nodes, we create a new Bloom filter for this d-tree and delete the old Bloom filter if it exists. For the parent s-node in $flush$, as mentioned above we use lazy removal, that is, the d-tree is kept until the s-node is a child in a $flush$ operation, when a new d-tree is created and the old d-tree discarded. We similarly keep the Bloom filter and create a new one only when the s-node is a child in the $flush$ operation.

To search for a key q, we start our search from the root s-node. We check if the Bloom filter for the root indicates that the d-tree of the root can contain q or not. If yes, then we search the root. If it does not contain q, then we move down one level according to the pointers and perform the search recursively on the subtree rooted at the node. Overall, in the worst case, we go through all the levels of the s-tree and search the corresponding d-tree, which gives the same worst-case query time as before. However, with high probability, we only search one s-node in total and the cost will be $O(\log_B \sigma)$ with high probability, which is a constant. Thus, NB-trees have a good average query time, as mentioned in Table 1.

6 EMPIRICAL STUDIES

6.1 Experimental Setup

We ran our experiment on a machine with Intel Core i5 3.20GHz CPUs and 8 GB RAM running CentOS 7. This machine has (1) a 250GB and 7200 rpm hard disk and (2) an SSD with the model "Crucial MX500" and the storage size of 1TB. Each disk page is 4KB. For HDD, (1) T_{seek} is about $3.18 \times 10^{-3}s$, and (2) $T_{con,W}$ and $T_{con,R}$ are similar and about $3.79 \times 10^{-5}s$ per page. For SSD, $T_{seek} = 1.78 \times 10^{-4}s$, $T_{con,W} = 2.24 \times 10^{-5}s$ per page and $T_{con,R} = 7.54 \times 10^{-6}s$ per page. Note that the reported value of T_{seek} is the average time of a seek operation among 10,000 random seek operations, and the reported values of $T_{con,W}$ and $T_{con,R}$ are based on writing/reading 100GB on a disk (HDD/SSD). All algorithms were implemented in C/C++.

Dataset. Following [19–21], we conducted experiments on synthetic datasets. Specifically, we generated synthetic datasets with n key-value pairs where each key is 8 bytes and each value is 128 bytes. Following [19–21], we generated keys uniformly to focus on worst-case performance. We observed that results on other distributions were similar and are discussed in Sec. E of our technical report [6]. The largest dataset generated is of size about 250 GB (2×10^9 keys).

Workload. We designed an *insert* workload and a *query* workload to study the query and insertion performance of different indices. Each *insert* workload is a workload which starts from an empty dataset and involves n_I insertion operations. Each *query*

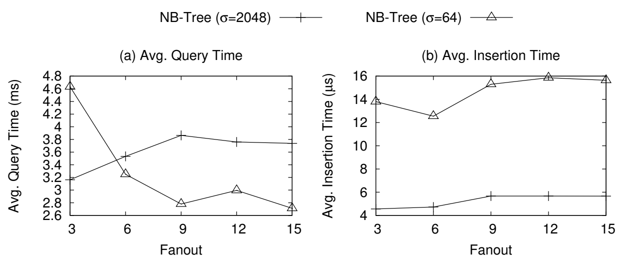


Figure 4: NB-Tree Performance vs Fanout

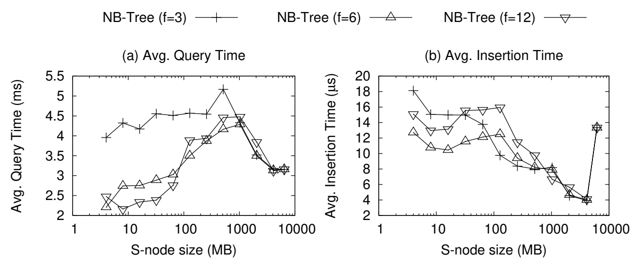


Figure 5: NB-Tree Performance vs. S-Node Size

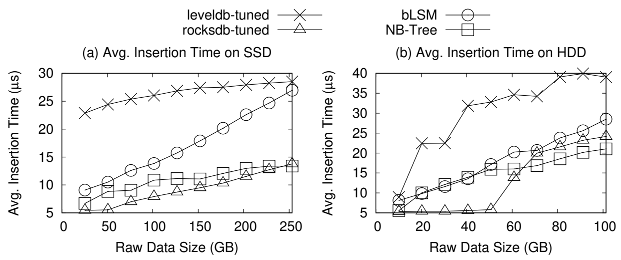


Figure 6: Avg. Insertion Time vs. Data Size

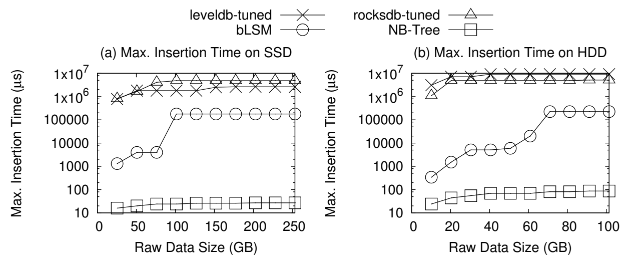


Figure 7: Max. Insertion Time vs. Data Size

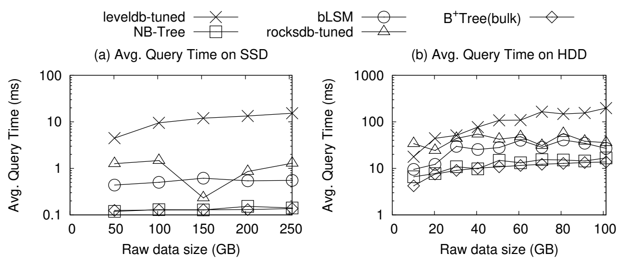


Figure 8: Avg. Query Time vs. Data Size

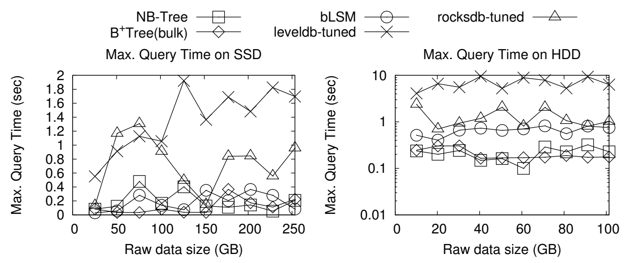


Figure 9: Max. Query Time vs. Data Size

workload is a workload which involves n_Q query operations performed on an index built based on the dataset containing n keys. n_Q is set to 10^4 throughout the experiments. In the query workload, we select keys uniformly from existing keys as the query input. Results on other distribution were similar and are discussed in Sec. E of our technical report [6].

Measurements. Based on the four performance metrics discussed in Section 1, we designed measurements on the indices for each of the two workloads. Consider an *insert* workload involving n_I insertion operations. We have 2 measurements, namely (1) *average insertion time* and (2) *maximum insertion time.* (1) *Average insertion time* is defined to be the average time taken per key to finish the entire insert workload, i.e., $\frac{t_I}{n_I}$, where t_I is the total time taken to complete n_I insertion operations. *Average insertion time* helps us verify our theoretical results on *amortized insertion time.* (2) *Maximum insertion time* is a measure on the entire workload. It is the maximum insertion time of a key over the entire workload. *Maximum insertion time* helps us verify our theoretical results on *worst-case insertion time.*

Consider a *query* workload involving n_Q query operations. We have 2 measurements, (1) the *average query time* and (2) the *maximum query time.* (1) The *average query time* is a measure on the *entire* workload. It is defined as the average time taken per key to finish the entire query workload, i.e., $\frac{t_Q}{n_Q}$ where t_Q is the total time taken to complete n_Q query operations in this workload. The *average query time* helps us verify our theoretical results on *average query time.* (2) The *maximum query time* is a measure on the *entire* workload. It is defined to be the maximum query time of a query in the entire workload. The *maximum query time* helps us verify our theoretical results on *worst-case query time.*

Algorithms. We compared our index, NB-trees, with 6 other indices: (1) LevelDB [27], (2) Rocksdb [25, 26], (3) bLSM [53], (4) B^{ϵ}-tree [11] and (5) B-tree [8] and (6) B^+-tree [55]. The first three indices (i.e., LevelDB, Rocksdb and bLSM) are three different implementations of LSM-trees. We report the memory consumption of all the algorithms in Table 2 based on the parameter setting described below. Note that there exist many other variants of the LSM-trees [20, 39, 67] which optimize the insertion/query performance which will be discussed in detail in Section 7. However, as to be discussed in Section 7, these performance optimization techniques originally designed for LSM-trees could also be applied to NB-trees and are orthogonal to our work. For fairness, we do not include the other variants of the LSM-trees for comparison.

Moreover, we ran a preliminary experiment in which we inserted about 6GB of raw data and measured average insertion time of all the algorithms. If average insertion time was larger than $100\mu s$, we excluded the algorithm from the rest of the experiments. This is because based on this result, we can conclude that the algorithm is not suitable for insertion-intensive workload and it will be infeasible to run such an algorithm on the large datasets in our experiment.

(1) LevelDB: LevelDB [27] is a widely used key-value store implementing an LSM-tree and has been used in the experiments of many existing studies [19, 39, 53, 54, 64]. In order to have a fair comparison, we adopt two different parameter settings for LevelDB, namely `leveldb-default` and `leveldb-tuned`. `leveldb-default` is LevelDB with the default setting similar to [19, 53] (i.e., multiplying factor = 10, in-memory write buffer size = 4 MB and no Bloom Filter feature enabled). In our preliminary experimental result, we found that the average insertion time of `leveldb-default` is larger than $100\mu s$. In the later experiments, we exclude this algorithm from our experimental results since it could not handle insertions with average insertion time smaller than $100\mu s$. `leveldb-tuned` is LevelDB with the "tuned" setting for the best-insertion performance. Specifically, in `leveldb-tuned`, following [20, 53], we enabled the Bloom Filter feature using 10 bits per key for short query time. Due to the large available memory, we varied the user parameter called "in-memory write buffer size" from 10 MB and 2GB to determine the "best" buffer size which could give the smallest average insertion time. When the buffer size is larger, LevelDB has fewer merge operations resulting in a smaller insertion time but at the same time, each merge takes longer resulting in a larger insertion time. In our experiment, we found that 1.5 GB is the "best" buffer size. Thus, `leveldb-tuned` refers to LevelDB with this setting.

(2) Rocksdb: Rocksdb [25] is a fork of LevelDB with some new features that are not necessarily relevant to our work (e.g., parallelism, see [26] for details). However, we observed that they performed differently under our workloads, so we include both algorithms. Similar to LevelDB, we performed parameter tuning for Rocksdb and observed that setting the write buffer size to 2GB has the best average insertion time. We refer to this algorithm as `rocksdb-tuned`. Bloom Filters are enabled and set to 10 bits per key. The exact setting we used can be found in Sec. C of our technical report [6].

(3) bLSM: bLSM [53] is a variant of an LSM-tree proposed for high query performance and low insertion delay. For a fair comparison, we obtained a parameter setting of bLSM with the best performance. We varied the user parameter of the in-memory component size to determine the "best" in-memory component size with the "best" insertion and query performance (increasing memory size improves both insertion and query performance). We found that 6 GB is the "best" size. In our experiment, we adopted this setting.

(4) B^{ϵ}-trees: We implemented two versions of the B^{ϵ}-trees, namely (a) *Public-Version* and (b) *Own-Version.* (a) *Public-version* is a publicly available version of the B^{ϵ}-trees used in system TokuDB [47]. We adopted the default settings of TokuDB. However, TokuDB's average insertion time in our preliminary experiments is more than $200\mu s$. It was not feasible to run TokuDB in our experiments which requires the insertion time to be at most $100\mu s$. (b) *Own-Version* is our own implementation of B^{ϵ}-tree. *Own-Version* could not handle the insertions with average insertion time less than $100\mu s$. Thus, since B^{ϵ}-tree (both *Public-Version* and *Own-Version*) is not suitable for high-insertion rate workloads, we exclude it from our experimental results.

(5) B-trees and (6) B^{+}-trees: Similar to B^{ϵ}-trees, we implemented two versions of *B^{+}-trees*, namely *Public-Version* and *Own-Version.* Here, *Public-Version* denotes the B^{+}-trees used in *wiredtiger* which is a storage engine in MongoDB [59]. Similarly, we exclude *B-trees* and *B^{+}-trees* in our experimental results since they could not handle insertions with insertion time smaller than $100\mu s$ per insertion. However, since it is well-known that B^{+}-trees are good for fast queries, we implemented a "bulk-load" version of a B^{+}-tree called `B+-tree(bulk)` as a baseline to compare the query performance among all indices in the experiments. We implemented `B+-tree(bulk)` by pre-sorting the data and adopting a bottom-up bulk-loading approach [55]. We do not include any measurement about the insertion statistics for `B+-tree(bulk)` since it does not show the realistic insertion performance for `B+-tree`. The query performance of the bulk-load version of a B^{+}-tree (i.e., `B+-tree(bulk)`) is better than the "normal insertion" version of a B^{+}-tree because `B+-tree(bulk)` could be constructed such that almost all nodes in `B+-tree(bulk)` are full and thus, the data are not scattered across different disk pages, resulting in a lower seek time and a smaller query time. We cache all the non-leaf nodes of the B^{+}-tree in memory to maximize query performance. It is not easy to design a "bulk load" version of B-trees (since some key-value pairs are stored in internal nodes) and thus, we do not include it.

NB-Trees. We implemented the final version of NB-tree discussed in Section 5, referred to as `NB-Tree`. We set f to 3 and σ to 2 GB after conducting experiments to find the "best" parameter for the NB-tree to be shown in Section 6.2.

6.2 Experiment for Parameter Setting

In this section, our experiment measures average insertion time for 25GB of raw data ($n_I = 2 \times 10^8$ keys), and average query time on a database of size 25GB (2×10^8 keys). We ran each experiment on an HDD three times and averaged the results, shown in Figs. 4-5.

Fanout. We studied the effect of fanout f for a small σ value, 64MB, and a large σ value, 2048MB, on NB-trees. Figure 4 (a) shows that when $\sigma = 64$, increasing f causes average query time to decrease. The trend is the opposite when $\sigma = 2048$. This is because when σ is small, increasing f reduces the height by a lot (from 8 levels when $f = 3$ to 4 levels when $f = 15$). When the height is smaller, fewer Bloom filters are checked, decreasing the probability that at least one of the Bloom filters returns a false positive. Thus, increasing f reduces the number of page accesses and the query time. However, for large values of σ, increasing f does not change the height by much (from 4 levels when $f = 3$ to 3 levels when $f = 15$). In this case, most queries perform only one disk access. Note that, d-trees of sibling s-nodes are written contiguously to the disk. Thus, when f is large, keys that are close to each other in the key space are written close to each other on disk. However, the query distribution is uniform, and it is likely that consecutive query keys are not close to each other in the key space. Hence, when f is large, the seek time during the queries becomes larger. This is less of an issue when f is small. Therefore, increasing f increases the seek time for queries. As a result, for $\sigma = 2048MB$, query time worsens when f increases.

Fig. 4 (b) shows that the insertion time increases when f increases. This result generally follows the theoretical model where the factor f in amortized insertion time complexity causes the insertion time to increase when f gets larger.

D-tree size. Fig. 5 shows that, generally, larger σ improves insertion time but worsens the query time, as theory suggests. However, one interesting observation is a local minimum observed at

Algorithms	Amortized insertion time $O(\alpha) \times (T_{con,W} + T_{con,R}) + O(\beta) \times T_{seek}$		Worst-case Insertion Time $O(\alpha) \times (T_{con,W} + T_{con,R}) + O(\beta) \times T_{seek}$		Worst-case Query Time $O(\alpha) \times (T_{con,R} + T_{seek})$
	α	β	α	β	α
B-tree [8]	$\log_B n$	$\log_B n$	$\log_B n$	$\log_B n$	$\log_B n$
B$^\epsilon$-tree [11]	$\frac{f \log_f B}{B} \log_B n$	$\frac{f \log_f B}{B} \log_B n$	$\frac{f \log_f B}{B} \log_B n$	$\frac{f \log_f B}{B} \log_B n$	$\log_f B \log_B n$
LSM-tree [45]	$\frac{f \log_f B}{B} \log_B n$	1	$\frac{n}{B}$	$\log_f B \log_B n$	$\log_f B (\log_B n)^2$
NB-tree (our paper)	$\frac{f \log_f B}{B} \log_B n$	$\frac{f \log_f B}{\sigma} \log_B n$	$\frac{f \log_f B}{B} \log_B n$	$\frac{f \log_f B}{\sigma} \log_B n$	$(\log_f \sigma+1) \log_B n$

Table 3: Summary of the theoretical results (performance in terms of time)

$\sigma = 16MB$ for average insertion time in Fig. 5 (b). This can be attributed to the HDD cache being $16MB$, which improves the contiguous I/O performance during *HandleFullSNode*. As σ gets beyond $4GB$, the insertion time increases since NB-Tree does not fit in main memory. The improvement in query performance when σ is larger than $1GB$ is because the main memory component becomes large compared to data size and some of the queries are answered by just checking the in-memory component.

Parameter Setting. In the rest of the experiments, we optimize NB-tree for an insertion-intensive workload. We select $\sigma = 2GB$ which has the best insertion performance based on Fig. 5 (b) and set $f = 3$ because for $\sigma = 2GB$, in Fig. 4 (b), $f = 3$ has the best insertion performance. Based on this parameter setting, we note that NB-Tree's memory usage is as follows. For data size of 250GB (the maximum data size used in our experiments), about 2.3GB is allocated for caching Bloom filters and 1GB for caching non-leaf node of d-trees. Interestingly, even when optimizing NB-trees for insertions, they perform queries almost as fast as a B$^+$-tree.

6.3 Experiment for Baseline Comparison

Average insertion time. Fig. 6 shows the average insertion time of the indices on HDD and SSD. NB-Tree achieves the lowest time on both HDD and SSD when the datasize is the largest, while bLSM's performance deteriorates when the data size gets larger as it keeps the number of components constant. rocksdb-tuned performs similar to NB-Tree.

Maximum insertion time. Fig 7 shows the maximum insertion time of the indices. NB-Tree achieves the lowest time on both HDD and SSD, outperforming other algorithms by at least 1000 times for some data sizes on both HDD and SSD. Maximum insertion time of rocksdb-tuned, bLSM and leveldb-tuned goes as high more than 0.2 s (for rocksdb-tuned, this number is 4.82s), which is unacceptable for many applications. The superior performance of NB-Tree is due to their logarithmic worst-case time together with the deamortization mechanism suggested in Section 5. Observe that rocksdb-tuned has the maximum insertion time of 4.82 seconds. Even though that happens only once during the insertion processes, it makes the system unreliable.

Average query time. Fig. 8 shows the average query time of the indices. NB-Tree achieves query time almost as low as B$^+$-tree(bulk) (which is worst-case optimal). rocksdb-tuned, leveldb-tuned and blsm have query times larger than NB-Tree, more prominently on SSDs.

Maximum query time. Fig. 9 shows the maximum query time of the indices. rocksdb-tuned has the worst performance while B$^+$-tree(bulk) is generally better. Note that all queries have to wait for at least one disk I/O operation, but an I/O operation can take long if the operating system is busy or if there are disk failures. Thus, maximum query time has a large variance and the comparison among the algorithms is less conclusive (note that insertions do not need to wait for disk I/O operations due to in-memory buffering).

Summary. The average query time of an NB-tree is 4 times smaller than LevelDB and 1.5 times than bLSM and Rocksdb. It is similar to the average query time of a nearly optimally constructed, bulk-loaded B$^+$-tree, where building a B$^+$-tree incrementally takes orders of magnitude longer than an NB-tree. Besides, the average and maximum insertion time of an NB-tree (which are at most 0.0001s) are multiple factors smaller than LevelDB, Rocksdb and bLSM (which could be greater than 0.2s). Overall, an NB-tree provides a more reliable insertion and query performance.

7 RELATED WORK

We discuss indices used for insertion intensive workloads.

LSM-trees. LSM-tree (see [42] for a detailed survey on the topic) is an index used for insertion-intensive workloads used in many systems such as BigTable [13], LevelDB[27], Cassandra [36], HBase [1], RocksDB [26, 44], Walnut [15] and Astrix DB [2]. By using an in-memory component and several on-disk B-tree components, LSM-trees [45] perform very few seek operations during insertions. However, this design causes a sub-optimal number of I/O operations during queries, and linear worst-case insertion time that causes long insertions delay (see [38, 58] for a discussion of LSM-tree's performance). Many improvements have been proposed to LSM-trees' design as discussed below.

Query improvement. [53] uses Bloom filters to improve the query time and [19] tunes the Bloom filter parameters. Compared with LSM-trees, we showed that Bloom filters adopted by NB-trees provide better theoretical and empirical performance. Method of [19] can also be used by NB-trees to optimize the Bloom filter parameters. Moreover, [18, 33] partition an LSM-tree into several smaller LSM-tree components which provides a constant factor improvement, and [65] optimizes the caching performance.

[38] uses fractional cascading [14] to provide asymptotically optimal worst-case query time. Fractional cascading connects different LSM-tree components to each other. Consider the B$^+$-tree of the i-th level of the LSM-tree. In each leaf node, N, of the B$^+$-tree, some key-value pairs have extra pointers pointing to a node, N', of the $(i+1)$-th level. The pointers from the i-th level to the $(i+1)$-th level are called *fence pointers*. Fence pointers satisfy the properties that (1) the first key-value pair k of node N must have a fence pointer pointing to a next-level node N' and every node N' at level i+1 must have a fence pointer pointing to it from level i. (2) Consider two keys, k_s and k_l, in level i that have fence pointers to nodes N_s and N_l in level $i + 1$, such that there does not exist another key k

in level i that has a fence pointer and that $k_s < k < k_l$. Let r_s be the smallest key in N_s and r_l the smallest key in N_l. It holds that $r_s \leq k_s < k_l \leq r_l$. These properties help in performing a constant number of disk-page accesses at each level.

LSM-trees with fractional cascading suffer from large worst-case insertion time and are not compatible with Bloom filters [53]. Thus, they provide a worse query performance in practice. The reason for their incompatibility is that to search the $(i+1)$-th level using the i-th level fence pointers, we need to have searched the i-th level. Based on this deduction, we need to have searched all the levels of the LSM-tree. However, using Bloom filter is only advantageous when we do not need to search all levels of the LSM-tree.

Insertion improvement. Most of the focus has been on optimizing the merge operation, divided into *leveling* and *tiering* categories. *leveling* is the category discussed so far, which sorts each LSM-tree component during the merge. *Tiering*, during a merge operation, appends the data to the lower levels and only sorts a level after it is full. This avoids rewriting the lower level component during the merge operation at the expense of the query time. [20] uses the leveling merge policy at some levels of the tree and tiering merge policy at other levels. In [21] unlike the original design, the ratio of the size across different adjacent levels of the LSM-tree is not constant. More variations of tiering are discussed in [7, 46, 64, 66, 67, 70]. [10] discusses in-memory optimization for faster writes. These improvements are orthogonal to our work and can be adopted by NB-trees in the future. [39] discusses a theoretical model to analyze insertion performance of LevelDB and provides methods for parameter optimization. Their methods require knowledge of probability distribution of the keys in advance and performs time-consuming optimizations not feasible in the real-world. Thus we did not include their method in our experiments. [38, 53, 58] discuss reducing the worst-case insertion time, but their methods take linear time to the data size compared with the logarithmic worst-case time of NB-trees.

B-tree and B-tree with Buffer. B-trees [8] are read-optimized indices, performing optimal number of I/O operations during queries[11]. But they perform a seek operation for every page access, sacrificing their insertion performance. B-trees with Buffer [11] (also known as B^{ϵ}-trees) are a write-optimized variant of B-trees where part of each disk page allocated to each node is reserved for a buffer. The buffer is flushed down the tree when it becomes full. B-trees with Buffer can be seen as a special case of NB-trees where s-node size is one disk page and their analysis of query and insertion performance follows from that of NB-trees. In that case, all disk accesses involve a seek operation, worsening the insertion performance, as our experiments confirmed. They also have worse space utilization since they allow half full nodes and worse range query performance since their nodes are not written contiguously on the disk. NB-trees keep their d-nodes full and write them contiguously for each s-node.

Other data structures. Many write optimized data structures such as [9, 30, 41, 62] have been proposed for a variety of settings and we do not have space to cover them all. Among them, Y-tree [32] is similar to B-trees with Buffer but allows for larger unsorted buffers at each non-leaf level of the B-tree that reduces the number of seek operations performed during insertions (can also be seen as a form of *tiering*). For a buffer similar in size to that of B-tree with Buffer, their performance will be similar to B-trees with Buffer and with the same weaknesses. However, a larger buffer worsens the point query performance (although range queries will not be affected as adversely), since it requires searching multiple pages of the unsorted buffer at each level of the tree by long scans. Y-trees also suffer from the issues mentioned above regarding space utilization and seek operations during range queries of B-trees with Buffer. Partitioned B-trees [28] divide a B-tree into multiple partitions by appending an artificial prefix to each key inserted in the tree. Since the prefix is the same within a partition, the data within each partition is sorted. The design allows for storing an LSM-tree within a B-tree (each partition is equivalent to an LSM-tree level), but it inherits the same disadvantages: large worst-case insertion time (merging the partitions will be done in the same fashion as LSM-trees) and suboptimal query time (querying each partition will take logarithmic in data size). Finally, mass-tree [43] is an in-memory data structure that is similar to this paper using a nested index, but the structural tree for mass-tree is a trie which, although works well in memory, can be unbalanced and cause large insertion and query cost if adopted for secondary storage.

In-memory optimization is outside the scope of this paper, but in-memory optimizations for B-trees such as [37, 49] improve the in-memory performance. However, their on-disk insertion performance is the same as B-trees, which is worse than NB-trees in terms of amortized insertion time.

Summary. Table 3 shows the theoretical performance of the indices mentioned above (written as multiples of $\log_B n$ for easier comparison). For amortized insertion time, NB-trees perform $\frac{\sigma}{B}$ times fewer seek operations than B^{ϵ}-trees, $\frac{\sigma}{f \log_f B}$ times fewer than B-trees, and similar to LSM-trees (σ is typically in the order of 10,000 times larger than B). NB-trees have worst-case insertion time logarithmic in data size while LSM-trees' worst-case insertion time is linear in data size. NB-trees' query time is a factor $\log_{\sigma} n$ smaller than LSM-trees and is asymptotically optimal. Overall, NB-trees have a better worst-case insertion and query time (considering the number of seek operations) than existing indices while maintaining practical properties, such as compatibility with Bloom filters and high space utilization.

8 CONCLUSION

We introduced Nested B-trees, an index that theoretically guarantees logarithmic worst-case insertion time and asymptotically optimal query time, and thus supports insertions at high rates with no delays while performing fast queries. This significantly improves on LSM-trees' linear worst-case insertion time and suboptimal query time and avoids long delays that frequently occur in LSM-trees during insertions. We empirically showed that NB-trees outperform RocksDB [25], LevelDB [27] and bLSM [53], commonly used LSM-tree databases, performing insertions faster than them and with maximum insertion time of 1000 smaller and lower query time by a factor of at least 1.5. NB-trees perform queries as fast as B-trees on large datasets, while performing insertions at least 10 times faster. In the future, a more detailed study can be done on optimizing the parameter setting of Bloom filters, and using different flushing schemes such as tiering.

REFERENCES

[1] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind facebook messages: Using hbase at scale. In *IEEE Data Eng. Bull.*, 2012.

[2] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. In *VLDB'14*, 2014.

[3] Amazon. Aws pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2019.

[4] Amazon. Ram cost, 2019. https://tinyurl.com/u7u3nna.

[5] Amazon. Ssd cost. https://tinyurl.com/wpzhw4u, 2019.

[6] A. Authors. First short worst-case insertion time index with asymptotically-optimal query time. Available at https://github.com/anonymoussubmitter/project2/NB-tree-technical.pdf, 2020.

[7] O. M. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *USENIX ATC' 17*, 2017.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In *Acta Informatica*, 1972.

[9] M. A. Bender, M. Farach-Colton, R. Johnson, S. Mauras, T. Mayer, C. A. Phillips, and H. Xu. Write-optimized skip lists. In *PODS'17*, 2017.

[10] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi. Accordion: Better memory organization for lsm key-value stores. *Proc. VLDB Endow.*, 11(12):1863–1875, Aug. 2018.

[11] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *ACM-SIAM'03*, 2003.

[12] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 275–290, New York, NY, USA, 2018. ACM.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *ACM Transactions on Computer Systems*, 2008.

[14] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. In *Algorithmica*, 1986.

[15] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. In *SIGMOD'12*, 2012.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *ACM symposium on Cloud computing*, 2010.

[17] J. Dabrowski and E. V. Munson. 40 years of searching for the best computer system response time. *Interacting with Computers*, 23(5):555–564, 2011.

[18] L. DAI, J. FU, and C. FENG. An improved lsm-tree index for nosql data-store. In *International Conference on Computer Science and Technology*, 2017.

[19] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD'17*, 2017.

[20] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD'18*, 2018.

[21] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In *SIGMOD*, 2019.

[22] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[23] R. Diorio, V. Timóteo, and E. Ursini. Testing an ip-based multimedia gateway. *INFOCOMP*, 13(1):21–25, 2014.

[24] DMR. Dropbox statistics. https://expandedramblings.com/index.php/dropbox-statistics/, 2017.

[25] Facebook. Rocksdb documentation. https://github.com/facebook/rocksdb, 2018.

[26] Facebook. Rocksdb features not in leveldb. https://github.com/facebook/rocksdb/wiki/Features-Not-in-LevelDB, 2018.

[27] Google. Leveldb documentation. https://github.com/google/leveldb/blob/master/doc/impl.md, 2017.

[28] G. Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, volume 3, pages 5–8, 2003.

[29] http://wersm.com/. Facebook statistics. http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/, 2017.

[30] J. Iacono and M. Pătraşcu. Using hashing to solve the dictionary problem. In *ACM-SIAM'12*, 2012.

[31] B. Insider. Credit suisse: Here's how high-frequency trading has changed the stock market. https://www.businessinsider.com/how-high-frequency-trading-has-changed-the-stock-market-2017-3#higher-trading-volumes-1, 2018.

[32] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB*, 1999.

[33] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. In *VLDB'17*, 2007.

[34] C. Kim and S.-U. Yang. Like, comment, and share on facebook: How each behavior differs from the other. *Public Relations Review*, 43(2):441–449, 2017.

[35] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2385–2388, 2011.

[36] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. In *ACM SIGOPS Operating Systems Review*, 2010.

[37] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[38] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. In *VLDB'10*, 2010.

[39] H. Lim, D. G. Andersen, and M. Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *FAST*, 2016.

[40] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[41] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *ACM Transactions on Storage*, 2017.

[42] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[43] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[44] Y. Matsunobu, S. Dong, and H. Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[45] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). In *Acta Informatica*, 1996.

[46] F. Pan, Y. Yue, and J. Xiong. dcompaction: Delayed compaction for the lsm-tree. In *International Journal of Parallel Programming*, 2017.

[47] Percona. Tokudb documentation. https://www.percona.com/doc/percona-tokudb/ft-index.html, 2017.

[48] G. Prasaad, A. Cheung, and D. Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[49] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[50] Rocksdb. Rocksdb memory usage. https://github.com/facebook/rocksdb/wiki/Memory-usage-in-RocksDB , 2020.

[51] Rocksdb. Rocksdb use-cases. https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases , 2020.

[52] Seagate. Product manual. https://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369h.pdf, 2018.

[53] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *SIGMOD'12*, 2012.

[54] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with vt-trees. In *FAST*, 2013.

[55] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997.

[56] I. L. Stats. Google search statistics. http://www.internetlivestats.com/google-search-statistics/, 2017.

[57] D. Teng, L. Guo, R. Lee, F. Chen, Y. Zhang, S. Ma, and X. Zhang. A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. *ACM Transactions on Storage (TOS)*, 14(2):1–26, 2018.

[58] R. Thonangi and J. Yang. On log-structured merge for solid-state drives. In *ICDE'17*, 2017.

[59] W. Tiger. Wired tiger website. http://www.wiredtiger.com/, 2019.

[60] N. Y. Times. Stock traders find speed pays, in milliseconds. https://www.nytimes.com/2009/07/24/business/24trading.html, 2018.

[61] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242. ACM, 2010.

[62] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *European Conference on Computer Systems*, 2014.

[63] M. Watch. Trading volume slows sharply even as stock market hits highs. https://www.marketwatch.com/story/trading-volume-slows-sharply-even-as-stock-market-hits-highs-2017-07-18, 2018.

[64] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *USENIX ATC' 15*, 2015.

[65] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang. Leaper: a learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[66] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *International Conference on Massive Storage Systems and Technology*, 2017.

[67] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive key-value store with skip-tree. In *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[68] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 511–526, 2020.

[69] H. Zhang, X. Liu, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1601–1615, 2020.

[70] W. Zhang, Y. Xu, Y. Li, and D. Li. Improving write performance of lsmt-based key-value store. In *International Conference on Parallel and Distributed Systems*, 2016.

A PROOFS

Correctness of insertions. We first show that the following invariant holds for insertions on an NB-Tree

Lemma 1. *Assume cross-s-node linkage and structural properties are satisfied for an NB-tree T. After inserting a key-value pair (k, v) using the algorithm describe in Sec. 3.2.1, cross-s-node linkage and structural properties are satisfied will be satisfied for the NB-tree obtained after the insertion, T'.*

Proof. Cross-s-node linkage property is concerned with the relationship between s-keys of an s-node N and the d-keys and s-keys of N's children. If insertion of (k, v) does not trigger $SNodeSplit$ or $flush$ then the theorem trivially holds. Next, consider any $flush(N)$ operation that is triggered. By definition of flush, if Cross-s-node linkage property holds for N and its children before the flush, it will also hold for N and its children after the flush. Now consider $SNodeSplit(N)$. We need to show the same for N_{small} and N_{large} and their children, as well as the parent of N and its children. For N_{small} and N_{large} their s-keys are that of N, and since the property was true for N it will be true for both of them. Now consider N's parent. A new s-key is inserted into N's parent, but by definition it is selected such that cross-s-node linkage holds. The same line of reasoning and using definitions of $SNodeSplit$ or $flush$ shows the same for structural properties. □

Finally, to see the correctness of insertions, consider induction on the number of insertions. Consider the base case of an NB-tree containing exactly one element and inserting the second element. Based on 1, the properties will be satisfied by the tree after the insertion. The inductive step similarly follows from Lemma 1, which completes the proof.

B IMPLEMENTATION DETAILS

The s-tree data structure contains data as discussed in Secs. 3, 5. Furthermore, to be able to calculate the time needed to perform the $HandleFullSNode$ operation when performing deamortization, we keep, in memory, the size of the d-tree for each s-node. Moreover, for each s-key in an s-node, we keep, in memory, the size of the keys-value pairs in the s-node's d-tree that are smaller than the s-key. By keeping this meta-data, we can decide the cost of each $flush$ operation, as well as which s-nodes the flush will be called on.

Performing HandleFullSNode in the background. When $HandleFullSNode$ in the background is performed in the background, NB-tree should be able to perform insertions and queries. Recall that insertions modify the in-memory s-node. However, $HandleFullSNode$ may be in the process of moving the data from the in-memory s-node to disk when new insertions happen. To avoid such conflicts, after $HandleFullSNode$ is called, new insertions are performed in a different in-memory d-tree. Thus, two d-trees exist in memory at the same time, one where new insertions are performed (the *new* d-tree), and the other from which previous keys are moved to disk (the *old* d-tree). After $HandleFullSNode$ moves the data from the old d-tree to disk, the old d-tree is discarded.

We need to ensure that queries can be performed correctly when $HandleFullSNode$ is being done in the background. To ensure that, the pointers from s-nodes to their d-tree is only updated after the new d-tree is created from $flush$ or $SNodeSplit$. Thus, when $HandleFullSNode$ starts, queries may need to be performed on both the new in-memory d-tree and the old in-memory d-tree. After the old in-memory d-tree is flushed completely to disk, the d-tree pointers are updated, at which point the newly written d-trees on disk become accessible. Only then the old in-memory d-tree is discarded, making sure that all the keys-value pairs in the NB-tree are accessible at all points.

Handling variations of in-memory insertion time. Although in our computational model in-memory insertion time is assumed to be negligible, in practice we observed sudden increases in in-memory insertion time in the order of $100\mu s$ (potentially due to maintenance of the in-memory data structure, the processes being swapped out, cache misses, etc.). To stop this from affecting the maximum insertion time in practice, we use a small unsorted buffer to buffer the insertions, before moving them to the in-memory data structure. The buffer helps dampen the effect of such spikes of insertion time increase. Since the buffer is small, an unsorted scan of the buffer per query does not impact the query time, as it is kept in memory.

Lazy Removal. Recall that during the flush operation, we need to remove the d-keys that were moved from the parent s-node to its children. In Section 4.1 we discussed a method that required rewriting of the parent s-node. Here, we discuss a lazy removal approach that removes this overhead. Consider the scenario when $flush(N)$ is called, assume that N's parent is P and N's d-tree is D. Some of the d-keys of D are flushed to the d-tree of children of N. At this stage, we create a pointer to the location of the smallest d-key K in D that is not flushed to N's children, that is, all the d-keys in D smaller than K are now present in the d-tree of N's children and need to be removed from D. Now instead of removing these d-keys from D at this point, we postpone this removal to when $flush(P)$ is called (i.e., when N is a child s-node during the flush operation). When $flush(P)$ is called, we need to flush the d-keys from P's d-tree to N's d-tree. In doing so, we only merge d-keys in D that are at least equal to K (using the pointer to K we remembered). Because of the contiguous-page property, these keys can be retrieved by a contiguous-page scan, and the existence of the keys smaller than K in N does not incur any extra cost for $flush(P)$. After $flush(p)$ is called, an entirely new d-tree is created for N and we discard the previous d-tree now, removing the keys smaller than K. This lazy removal does not incur any extra cost for insertions as the d-keys whose removal where postponed will not be read by the insertion algorithm. Moreover, the total size of siblings will be $f(\sigma + 1)$ because one s-node can now have at most σ more s-keys than was discussed in the above paragraph.

C DETAILED EXPERIMENTAL SETUP

Here we provide the detailed setup for the algorithms used in our experiments for ease of reproducibility,

`leveldb`.

```
block_cache = NewLRUCache(4GB)
compression = kNoCompression
filter_policy = NewBloomFilterPolicy(10)
write_buffer_size = 1.5GB
```

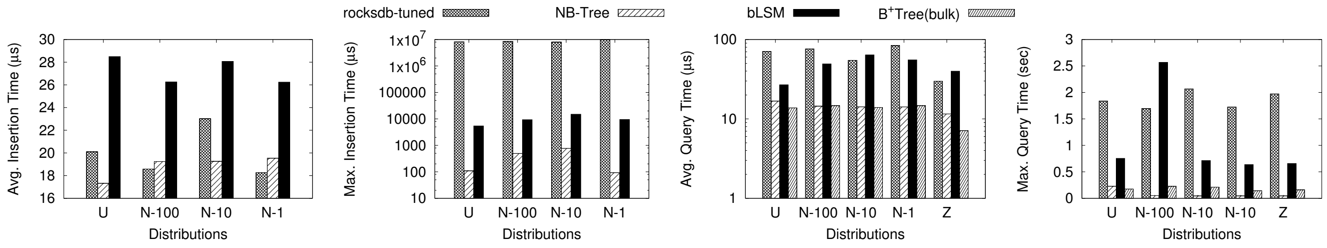


Figure 10: Impact of data distribution (insertion time of `B+-tree(bulk)` not reported as it is bulk-loaded for query comparison)

```
max_file_size = 0.75GB
```

`rocksdb`. Note that according to `rocksdb`'s documentation [50] setting max_open_files = -1 while setting *open files limit* to unlimited in the operating system (seen by the ulimit command in linux) results in caching the index and the boom filter by `rocksdb`.

```
filter_policy.reset(NewBloomFilterPolicy(10))
block_cache = NewLRUCache(2GB)
compression = kNoCompression
options.max_open_files = −1
options.max_bytes_for_level_base = 2GB*10
options.write_buffer_size = 2GB
```

`bLSM`.

```
stasis_buffer_manager_size = 1.5GB
ltable = new bLSM(0, 6GB, 4KB, 4KB, 1)
```

D PARAMETER SETTING

NB-trees have three parameters, f, σ and B. B is set similar to B-trees so we focus on the other two. f provides a trade-off between insertion cost and query cost while σ provides a trade-off between the number of seek operations per insertion and query cost. σ depends on how expensive seek operations are, but typically, for fast insertions, it is set to the order of tens or hundreds of mega bytes. Typically, f is set to a number in the order of 10 for write intensive workloads, and its increase affects insertions much more than queries, as the insertion cost linearly depends on f but query cost's dependence is only logarithmic. Section 6.2 provides an empirical analysis of parameter setting.

E EXPERIMENT WITH OTHER DISTRIBUTIONS

We considered a dataset with keys from a Gaussian distribution with standard deviation of 1, 10 and 100 to see the performance on a biased distribution. We also considered queries following Zipfian distribution from YCSB [16] used commonly in the literature [22, 35, 48, 68, 69] .Fig. 10 shows the impact of different distribution on the performance. The experiments are performed on an HDD. The insertion measurements are for the insertion of 8×10^8 key-value pairs (about 101GB or raw data) and query measurements are for querying the index after the insertions. U refers to uniformly distributed insertions and queries, N-x refers to insertions with Gaussian distribution with standard deviation x for $x \in \{1, 10, 100\}$ and uniformly distributed queries and Z refers to uniformly distributed insertions and queries with Zipfian distribution. The results with other distributions are similar to that of uniform distribution, with `NB-Tree` providing the average insertion time similar to `rocksdb-tuned`, but with the max. insertion time multiple orders smaller and query time close to that of `B+Tree(bulk)`.