FITing-Tree: A Data-aware Index Structure

Alex Galakatos[1*]   Michael Markovitch[1*]   Carsten Binnig[2]
Rodrigo Fonseca[1]   Tim Kraska[3]

[1]Brown University {first_last}@brown.edu   [2]TU Darmstadt {first.last@cs.tu-darmstadt.de}   [3]MIT CSAIL {last@mit.edu}

ABSTRACT

Index structures are one of the most important tools that DBAs leverage to improve the performance of analytics and transactional workloads. However, building several indexes over large datasets can often become prohibitive and consume valuable system resources. In fact, a recent study showed that indexes created as part of the TPC-C benchmark can account for 55% of the total memory available in a modern DBMS. This overhead consumes valuable and expensive main memory, and limits the amount of space available to store new data or process existing data.

In this paper, we present FITing-Tree, a novel form of a learned index which uses piece-wise linear functions with a bounded error specified at construction time. This error knob provides a tunable parameter that allows a DBA to FIT an index to a dataset and workload by being able to balance lookup performance and space consumption. To navigate this tradeoff, we provide a cost model that helps determine an appropriate error parameter given either (1) a lookup latency requirement (e.g., 500ns) or (2) a storage budget (e.g., 100MB). Using a variety of real-world datasets, we show that our index is able to provide performance that is comparable to full index structures while reducing the storage footprint by orders of magnitude.

*Authors contributed equally.

1 INTRODUCTION

Tree-based index structures (e.g., B+ trees) are one of the most important tools that DBAs leverage to improve the performance of analytics and transactional workloads. However, for main-memory databases, tree-based indexes can often consume a significant amount of memory. In fact, a recent study [48] shows that the indexes created for typical OLTP workloads can consume up to 55% of the total memory available in a state-of-the-art in-memory DBMS. This overhead not only limits the amount of space available to store new data but also reduces space for intermediates that can be helpful when processing existing data.

To reduce the storage overhead of B+ trees, various compression schemes have been developed [7, 9, 23, 49]. The main idea behind these techniques is to remove the redundancy that exists among keys and/or to reduce the size of each key inside a node of the index. For example, prefix and suffix truncation can be used to store common parts of keys only once per index node, reducing the total size of the tree. Additionally, more expensive compression techniques like Huffmann coding can be applied within each node but come at a higher runtime cost since pages must be decompressed to search for an item.

Although each of the previously mentioned compression schemes reduce the size of an index node, the memory footprint of these indexes still grows linearly with the number of distinct keys to be indexed, resulting in indexes that can consume a significant amount of memory. This observation is especially true for data such as timestamps or sensor readings that are generated in a wide variety of applications (e.g., autonomous vehicles, IoT devices). Even worse, the number of unique keys to be indexed for such data types typically grow over time, resulting in indexes that are constantly growing. Consequently, a DBA has no way to restrict memory consumption other than dropping an index completely.

To tackle this issue, we present FITing-Tree, a novel index structure that compactly captures trends in data using piece-wise linear functions. Unlike typical indexes which use fixed-size pages on the leaf level that point to the data, FITing-Tree uses piece-wise linear functions to quickly *approximate* the position of an element. By leveraging the trends within the data, FITing-Tree can reduce the memory consumption of

an index by orders of magnitude compared to a traditional B+ tree. At the core of our index structure is a parameter that specifies the amount of acceptable error (i.e., a constant that is the maximum distance between the predicted and actual position of any key). Unlike existing index structures, our error parameter allows a DBA to FIT an index to a given scenario and balance the lookup performance and space consumption of an index. To navigate this tradeoff, we also present a cost model that helps a DBA choose an appropriate error term given either (1) a lookup latency requirement (e.g., 500ns) or (2) a storage budget (e.g., 100MB).

In the basic version of FITing-Tree, we assume that the table data to be indexed is sorted by the index key (i.e., clustered index) but we also show how our techniques extend to secondary (i.e., non-clustered) indexes. Using a variety of real-world datasets, we show that our index structure provides performance that is comparable to full and fixed-page indexes (even for a worst-case dataset) while reducing the storage footprint by orders of magnitude.

Using linear functions to approximate the distribution makes FITing-Tree a form of a learned index [30]. However, in contrast to the initially proposed techniques, our approach allow us to (1) bound for the worst-case lookup performance, (2) efficiently support insert operations, and (3) enable paging (i.e., the entire data does not have to reside in a contiguous memory region). Furthermore, although the problem of approximating distributions using piece-wise functions is also not new [10, 12, 13, 16, 18, 27, 33, 35, 41, 46], none of these techniques have been applied to indexing and therefore do not consider operations that indexes must support.

Another interesting observation is that our compression scheme in FITing-Tree is orthogonal to node-level compression techniques such as the previously mentioned prefix/suffix truncation. In other words, since FITing-Tree internally uses a tree structure for inner nodes, we can still apply these techniques to further reduce an index's size.

In summary, we make the following contributions:

- We propose FITing-Tree, a novel index structure that leverages properties about the underlying data distribution to reduce the size of an index.
- We present and analyze an efficient segmentation algorithm that incorporates a tunable error parameter that allows DBAs to balance the lookup performance and space footprint of our index.
- We propose a cost model that helps a DBA determine an appropriate error threshold given either a latency or storage requirement.
- Using several real-world datasets, we show that our index provides similar (or in some cases even better) performance compared to existing index structures while consuming orders of magnitude less space.
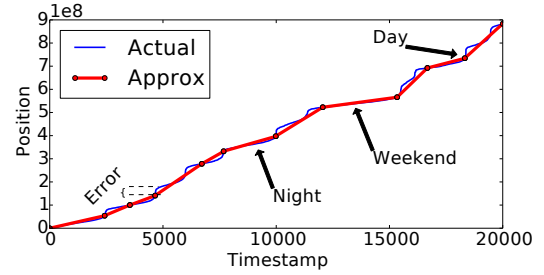

Figure 1: Key to position mapping for IoT data.

The remainder of the paper is organized as follows. In Section 2, we first present an overview of our new index structure called FITing-Tree. Afterwards, we discuss the main index operations: bulk loading (Section 3), lookups (Section 4) and inserts (Section 5). Section 6 then introduces our cost model that allows a DBA to balance the lookup performance and the space consumption of a FITing-Tree. Finally, in Section 7 we discuss the results of our evaluation on real and synthetic datasets, summarize related work in Section 8, and finally conclude in Section 9.

2 OVERVIEW

At a high level, indexes (and B+ trees over sorted attributes in particular) can be represented by a function that maps a key (e.g., a timestamp) to a storage location. Using this representation, FITing-Tree partitions the key space into a series of disjoint linear segments that approximate the true function, since it is (generally) not possible to fully model the underlying data distribution. At the core of this process is a tunable error threshold which represents the maximum distance that the predicted location of any key inside a segment is from its actual location. Instead of storing all values in the key space, FITing-Tree stores only (1) the starting key of each linear segment and (2) the slope of the linear function in order to compute a key's approximate position using linear interpolation.

In the following, we first discuss how we can use functions to map key values to storage locations. Then, we discuss how we leverage this function representation to efficiently implement our index structure on top of a B+ tree for clustered indexes. Finally, we show how our ideas can also be applied to compress secondary indexes.

2.1 Function Representation

One key insight to our approach is that we can abstractly model an index as a monotonically increasing function that maps keys (i.e., values of the indexed attribute) to storage locations (i.e., its page and the offset within that page). To explain this intuition, assume that all keys to be indexed

are stored in a sorted array, allowing us to use an element's position in the array as its storage location.

As an example, consider the IoT dataset [1], which contains events from various devices (e.g., door sensors, motion sensors, power monitors) installed throughout a university building. In this dataset, the data is sorted by the timestamp of an event, allowing us to construct a function that maps each timestamp (i.e., key) to its position in the dataset (i.e., position in a sorted array), as shown in Figure 1. Unsurprisingly, since the installed IoT devices monitor human activity, the timestamps of the recorded actions follow a pattern (e.g., there is little activity during the weekend and at night).

Since a function that represents an index can be arbitrarily complex and data-dependent, the precise function that maps keys to positions may not be possible to learn and is expensive to build and update. Therefore, our goal is to approximate the function that represents the mapping of a key to a position.

To compactly capture trends that exist in the data while being able to efficiently build a new index and handle updates, we use a series of piece-wise linear functions to approximate an arbitrary function. As shown in Figure 1, for example, our segmentation algorithm (described further in Section 3) partitions the timestamp values into several linear segments that are able to accurately reflect the various trends that exist in the data (e.g., less activity during the weekend). Since the approximation captures trends in the data, it is agnostic to key density (a trend with sparse keys can be captured as well as a trend with dense keys).

Although more complex functions (e.g., higher order polynomials) can be used to approximate the true function, piece-wise linear approximation is significantly less expensive to compute. This dramatically reduces (1) the initial index construction cost, and (2) improves insert latency for new items (see Section 5).

The resulting piece-wise linear approximation, however, is not precise (i.e., a key's predicted location is not necessarily its true position). We therefore define the *error* associated with our approximation as the maximum distance between the actual and predicted location of any key, as shown below, where $pred\_pos(k)$ and $true\_pos(k)$ return the predicated and actual position of an element k respectively.

$$error = \max(|pred\_pos(k) - true\_pos(k)|) \ \forall \ k \ \in \ keys \ \ (1)$$

This formulation allows us to define the core building block of FITing-Tree, a *segment*. A segment is a contiguous region of a sorted array for which any key is no more than a specified error threshold from its interpolated position. Depending on the data distribution and the error threshold, the segmentation process will yield a different number of segments that approximate the underlying data. Therefore,
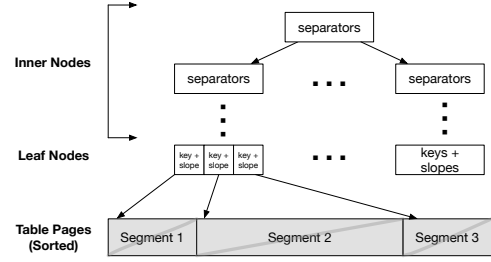

Figure 2: A clustered FITing-Tree index.

importantly, the error threshold enables us to balance memory consumption and performance. After the segmentation process, FITing-Tree stores the boundaries and slope of each segment (instead of each individual key) in a B+ tree, reducing the overall memory footprint of the index.
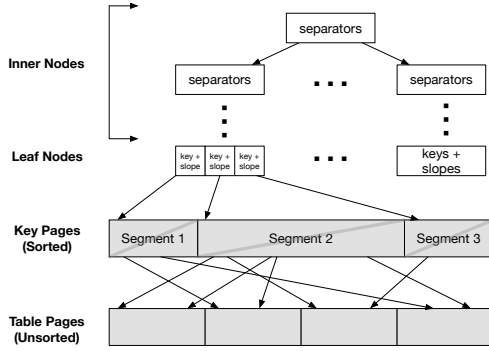
2.2 FITing-Tree Design

As previously mentioned, our segmentation process partitions the key space of an attribute into disjoint linear segments such that the predicted position of any key inside a segment is no more than a bounded distance away from the key's true position. FITing-Tree organizes these segments in a tree to efficiently support insert and lookup operations.

In the following, we first discuss clustered indexes, where records are already sorted by the key that is being indexed. Afterwards, we show how our technique can be extended to provide similar benefits for secondary indexes.

2.2.1 Clustered Indexes. In a traditional clustered B+ tree, the table data is stored in fixed-size pages and the leaf level of the index contains only the first key of each of these table pages. Unlike a clustered B+ tree, in a FITing-Tree, the table data is partitioned into variable-sized segments (pages) that satisfy the given error threshold. Each segment is essentially a fixed-size array, but successive segments can be allocated independently (i.e., non contiguously).

Figure 2 shows the structure of a clustered FITing-Tree index. As shown, the underlying data is partitioned into a series of variable-sized segments that approximate the distribution of keys to be indexed. Depending on the error parameter and the data distribution, several consecutive keys can thus be summarized into a single segment. Details of the segmentation algorithm that divides the table data into variable-sized segments are discussed in Section 3.

Unlike a traditional B+ tree, each leaf node in a FITing-Tree stores the segment's slope, starting key, and a pointer to a segment. This allows us to use interpolation search in each segment since the data within this segment is approximated by a linear function given by the slope.

Figure 3: A non-clustered FITing-Tree index.

Figure 4: A segment from (x_1, y_1) to (x_3, y_3) is not valid if (x_2, y_2) is further than *error* from the interpolated line.
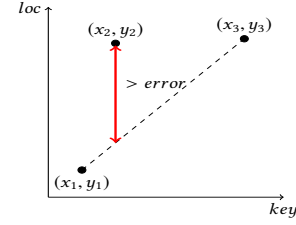
The inner nodes of a FITing-Tree are the same as a B+ tree (i.e., lookup and insert operations are identical to a normal B+ tree). However, once a lookup or an insert reaches the leaf level, FITing-Tree needs to perform additional work. For lookups, we need to use the slope and the distance to the starting key to calculate the key's approximate position (offset in the segment). Since the resulting position is approximate, FITing-Tree must then perform a local search (e.g., binary, linear) to find the item, discussed further in Section 4.

Insert operations also require additional work upon reaching a leaf level page, since we must ensure that the error threshold is always satisfied. Therefore, we present two different insertion strategies (described in detail in Section 5). The first strategy performs in-place updates to the segment (as a baseline) while the second strategy uses a more advanced buffer-based strategy to hold inserted data items.

Finally, instead of internally using a B+ tree to locate the segment for a key, FITing-Tree could instead use any other index structure. For example, if the workload is read-only, other high performance index structures (e.g., FAST [28]) can be used. In Section 7.4 we show how FITing-Tree performs when using different internal data structures, including FAST.

2.2.2 Non-clustered Indexes. Secondary indexes can dramatically improve the performance of queries involving selections over a non-primary key attribute. Without secondary indexes, these queries must examine all tuples, which is often prohibitive. Unlike a clustered index, a non-primary key attribute is not sorted and may contain duplicates.

The primary difference between a clustered and an nonclustered FITing-Tree is that a non-clustered FITing-Tree requires an additional "indirection layer" (called "Key Pages" in Figure 3). This layer is essentially an array of pointers that is the same size as the data but is sorted by the key that is being indexed. For example, the first position in this indirection layer will contain a pointer to the smallest value

of the key being indexed. Note that a secondary B+ tree that uses fixed-size paging also requires this indirection layer.

The first step in creating a non-clustered FITing-Tree is to build the indirection layer by sorting the data by the indexed key (e.g., temperature, age) and materializing the array of pointers to each data item in the sorted order. Next, like in the clustered index case, the segmentation algorithm scans the indirection layer and produces a valid set of segments that are then inserted into the upper level tree.

All operations on a non-clustered FITing-Tree internally operate on the indirection layer. For example, for a lookup, the returned position of the data item is its position in the indirection layer. Then, to access the value, FITing-Tree follows the pointer in the indirection layer at the predicted position.

Although the sorted level of key pages in a non-clustered FITing-Tree introduces additional overhead compared to a clustered FITing-Tree index, this overhead occurs in any non-clustered (secondary) index. However, as we show in our experiments, a non-clustered FITing-Tree is significantly smaller than a non-clustered B+ tree with fixed-size pages since it has fewer leaf and internal nodes.

3 SEGMENTATION

In the following, we describe how FITing-Tree partitions the key space of an attribute into variable-sized segments that satisfy the specified error. After this process, each segment is inserted into a B+ tree to enable efficient insert and lookup operations, described further in Section 4 and Section 5.

3.1 Design Choices

A common objective when fitting a function is to minimize the least square error (minimizing the second error norm E_2). Unfortunately, such an objective does not provide a guarantee for the maximal error and therefore does not provide a bound on the number of locations which must be scanned after interpolating a key's position. Therefore, our objective is to satisfy a maximal error (E_∞), demonstrated in Figure 4.

Algorithm 1 SHRINKINGCONE Segmentation

```
1   sl_high ← ∞
2   sl_low ← 0
3   the first key is the segment origin
4   for every k ∈ keys (in increasing order)
5       do if k is inside the cone:
6           then update sl_high
7               update sl_low
8           else key k is the origin of a new segment
9               sl_high ← ∞
10              sl_low ← 0
```

While several optimal (in the number of produced segments) piece-wise linear approximation algorithms exist, these techniques are prohibitively expensive (e.g., a dynamic programming algorithm [31] has a runtime of $O(n^3)$ using $O(n)$ memory). Second, most existing online piece-wise linear approximation algorithms [17, 27] have a high storage complexity and/or do not guarantee a maximal error. Lastly, even linear time algorithms may not be efficient enough since multiplicative constants have a significant effect.

Therefore, to be able to efficiently (1) construct the index, and (2) support inserts, we need a highly efficient one-pass linear algorithm. This focus on efficiency led us to choose linear piece-wise functions, since higher order approximations often incur additional costs as previously discussed.

In the following, we describe a proposed segmentation algorithm, similar to FSW [33, 46], which is linear in runtime, has low constant memory usage, and guarantees a maximal error in each segment. Importantly, though, we address (1) how to extend these techniques to indexing, including looking up and inserting data items, (2) prove that, in the worst case, segments are bounded in size, and (3) analyze the algorithm and compare it to an optimal algorithm.

3.2 Segment Definition

As previously described, a segment is a region of the key space that can be represented by a linear function whereby all keys are within a bounded distance from their linearly interpolated position. More specifically, a segment is represented by the first point (first key) and by the last point (the last key) in the segment. Using this definition, we can fit a linear function to the locations of keys in the segment (using the start key, end key, and the number of positions).

Recall that every segment must satisfy the maximal error (i.e., a key's predicated position is at most *error* number of elements away from its true position). This leads to an important property (proof in Appendix A.1) of a *maximal segment* (a segment is maximal when the addition of a key will violate the specified error):

THEOREM 3.1. *The minimal number of locations covered by a maximal linear segment is error + 1.*
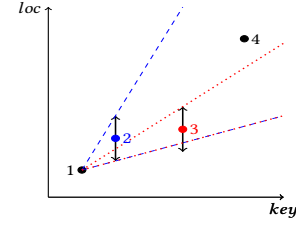
Figure 5: ShrinkingCone - Point 1 is the origin of the cone. Point 2 is then added, resulting in the dashed cone. Point 3 is added next, yielding in the dotted cone. Point 4 is outside the dotted cone and therefore starts a new segment.

This allows us to quantify how bad a "worst case" (i.e., dataset and error threshold that produce maximal number of segments) can be. Since the minimum number of locations covered by a maximal segment is bound by the error, the total size of a FITING-TREE is also bounded. Therefore, in the worst case (every maximal segment covers *error* + 1 locations), a FITING-TREE will not be larger than an index that uses fixed-size pages of size *error* (e.g., B+ tree).

3.3 Segmentation Algorithm

As previously mentioned, we needed a fast and efficient algorithm rather than an optimal one. We therefore chose to use a greedy streaming algorithm SHRINKINGCONE (Algorithm 1) which, given a starting point (key) of a segment, attempts to maximize the length of a segment while satisfying a given error threshold. SHRINKINGCONE is similar to FSW [33] but considers only monotonically increasing functions and can produce disjoint segments. The main idea behind SHRINKINGCONE is that a new key can be added to a segment if and only if it does not violate the error constraint of any previous key in the segment.

More specifically, we define a cone by the triple: origin point (the key and its location), high slope (sl_{high}), and low slope (sl_{low}). The combination of the starting point and the low slope gives the lower bound of the cone, and the combination of the starting point and the high slope gives the upper bound of the cone. Intuitively, the cone represents the family of feasible linear functions for a segment starting at the origin of the cone (the high and low slopes represent the range of valid slopes). When a new key is added to a segment, the high and low slopes are calculated using the key and the key's position plus *error* and minus *error* (respectively). In the update step (lines 6-7 of Algorithm 1), the lowest high slope and the highest low slope values are then selected (between the newly calculated and previous slopes). Therefore, the cone either narrows (the high slope decreases and/or the low slope increases), or stays the same. If a new key to be added to the segment is outside of the cone, there

Dataset	error	SHRINKINGCONE	Optimal	Ratio
Taxi drop lat	10	5358	4996	1.07
Taxi drop lat	100	351	271	1.29
Taxi drop lat	1000	51	48	1.06
Taxi drop lon	10	1198	1138	1.05
Taxi drop lon	100	371	325	1.14
Taxi drop lon	1000	40	37	1.08
Taxi pick time	10	6238	4359	1.43
Taxi pick time	100	165	137	1.2
OSM lon	10	7727	6027	1.28
OSM lon	100	101	63	1.6
Weblogs	10	16961	14179	1.2
Weblogs	100	909	642	1.42
IoT	10	8605	6945	1.24
IoT	100	723	572	1.26

Table 1: SHRINKINGCONE compared to optimal.

must exist at least one previous key in the segment for which the error constraint will be violated. Therefore, a new key that is not inside the cone cannot be included in the segment, and becomes the origin point of the new segment.

Figure 5 illustrates how the cone is updated: point 1 is the origin of the cone. Point 2 updates both the high and low slopes. Point 3 is inside the cone, however it only updates the upper bound of the cone (point 3 is less than *error* above the lower bound). Point 4 is outside of the updated cone, and therefore will be the first point of a new segment.

3.4 Algorithm Analysis

While the SHRINKINGCONE algorithm has a runtime of $O(n)$ and only uses a small constant amount memory (to keep track of the cone), it is not optimal. Moreover, for a given maximal error and an adversarial dataset the number of segments that it produces can be arbitrarily worse than an optimal algorithm, as we prove in Appendix A.2.

Although SHRINKINGCONE can be arbitrarily worse compared to optimal segmentation for a given maximal error, there is a limit for how bad it can be in practice since we do have a guarantee that a maximal segment covers at least *error* + 1 locations.

The maximum number of segments SHRINKINGCONE produces is at most $\min\left(\frac{|keys|}{2}, \frac{|D|}{error+1}\right)$, where $|D|$ is the size of the dataset. This guarantee stems from Theorem 3.1: no input with less than 3 keys spanning at least *error* + 2 positions will cause SHRINKINGCONE to create a new segment. Thus, compared to traditional B+ trees, in the worst case, FITING-TREE will produce no more segments (pages) than a B+ tree that uses fixed-size pages (of size *error*).

To evaluate SHRINKINGCONE, we implemented the optimal algorithm (runtime of $O(n^2)$ and memory consumption of $O(n^2)$) using 10^6 elements from real-world datasets: NYC Taxi Dataset [2], OpenStreetMap [3], Weblogs [1], and IoT [1]. Table 1 shows the number of segments generated by the optimal algorithm and by SHRINKINGCONE. As shown, the number of segments that our algorithm produces is comparable to the number of segments in the optimal case.

4 INDEX LOOKUPS

One of the most important operations of an index is to lookup a single key or a range of keys. However, since each entry in the leaf level of FITING-TREE points to a segment, performing a lookup requires first locating the segment that a key belongs to and then performing a local search inside the segment. In the following, we first describe how FITING-TREE performs lookup operations for a single key and then show how we can extend this technique to range predicates.

4.1 Point Queries

The process of searching a FITING-TREE for a single element involves two steps: (1) searching the tree to find the segment that the element belongs to, and (2) finding the element within a segment. These steps are outlined in Algorithm 2.

4.1.1 Tree Search. Since, as previously described, each segment is stored in a B+ tree (with its first key as the key and the segment's slope and a pointer to the table page as its value), we must first search the tree to find the segment that a given key belongs to. To do this, we begin traversing the B+ tree from the root to the leaf, using standard tree traversal algorithms. These steps, outlined in the SEARCHTREE function of Algorithm 2, terminate when reaching a leaf node which points to the segment that contains the key.

Since the B+ tree is used to index segments rather than individual points, the runtime for searching for the segment that a key belongs in is $O(log_b(p))$, where b is a constant representing the fanout of the tree (i.e., number of separators inside a node) and p is the number of segments created during the segmentation process.

4.1.2 Segment Search. Once FITING-TREE finds the segment for a key, it then must find the key's position inside the segment. Recall that segments are created such that an element is no more than a constant distance (*error*) from the element's position determined through linear interpolation. Other techniques for interpolation search inside a fixed-size index page are discussed in [22].

To compute the approximate location of a key k within a given segment s, we subtract the key from the first key that appears in the segment $s.start$. Then, we multiply the difference by the segment's slope $s.slope$, as shown below in the following equation.

$$pred\_pos = (k - s.start) \times s.slope \qquad (2)$$

After interpolating an element's position, the true position of an element is guaranteed to be within the error threshold. Therefore, FITING-TREE locally searches the following region using binary search (as shown in Algorithm 2).

$$true\_pos \in [pred\_pos - error, pred\_pos + error] \qquad (3)$$

Algorithm 2 Lookup Algorithm

LOOKUP(*tree*, *key*)

1   $seg \leftarrow$ SearchTree(*tree.root*, *key*)
2   $val \leftarrow$ SearchSegment(*seg*, *key*)
3   **return** val

SEARCHTREE(*node*, *key*)

1   $i \leftarrow 0$
2   **while** $key < node.keys[i]$
3   **do** $i \leftarrow i + 1$
4   **if** $node.value[i].isLeaf()$
5   **then** $j \leftarrow 0$
6   **while** $key < node.values[j]$
7   **do** $j \leftarrow j + 1$
8   **return** $node.values[j]$
9   **return** SearchTree(*node.values[i]*, *key*)

SEARCHSEGMENT(*seg*, *key*)

1   $pos \leftarrow (key - seg.start) \times seg.slope$
2   **return** BinarySearch(*seg.data*, $pos - error$, $pos + error$, *key*)

However, it is also important to note that any search algorithm, including linear search, binary search, or exponential search can also be used depending on the specific scenario (e.g., hardware properties, error threshold).

Since segments satisfy the specified error condition, the cost of searching for an element inside a segment is bounded. More specifically, the runtime for locating an element inside a segment is $O(log_2(error))$ where $error$ is constant.

4.2 Range Queries

Range queries, unlike point queries, have the additional requirement that they examine every item in the specified range. Therefore, for range queries, the selectivity of the query (i.e., number of tuples that satisfy the predicate) has a large influence on the total runtime of the query.

However, like point queries, range queries must also find a single tuple: either the start or the end of the range. Therefore, FITING-TREE uses the previously described point lookup techniques to find the beginning of the specified range. Then, since segments either store keys contiguously (clustered index) or have an indirection layer with pointers that is sorted by the key (non-clustered index), FITING-TREE can simply scan from the starting location until it finds a key that is outside of the specified range. For a clustered index, scanning the relevant range performs only very efficient sequential access, while for a non-clustered index, range queries require random memory accesses (which is true for any non-clustered index).

5 INDEX INSERTS

Along with locating an element, an index needs to be able to handle insert operations. In some applications, maintaining a strict ordering guarantee is necessary, in which case FITING-TREE should ensure that new items are inserted in-place. However, in situations where this is not the case, we've

developed a more efficient insert strategy that improves insert throughput. In the following, we discuss each of these strategies for inserting new items into a FITING-TREE. Then, in Section 7.1.3, we show how these strategies compare for various workloads and parameters.

5.1 In-place Insert Strategy

In a typical B+ tree that uses paging, pages are left partially filled and new values are inserted into an empty slot using an in-place strategy. When a given page is full, the node is split into two nodes, and the changes are propagated up the tree (i.e., the inner nodes in the tree are updated).

Although similar, insert operations in FITING-TREE require additional consideration since any key in the segment must be no more than specified error amount (*error*) away from its interpolated position. Importantly, in-place inserts require moving keys in the page to preserve the order of the keys.

Without any a priori knowledge about the error of a given key, any attempt to move the key requires checking to see if the error condition is satisfied. To make matters worse, a single insert may require moving many keys (in the worst case, all keys in the page) to maintain the sorted order. Thus, we must have a priori knowledge about any given key to determine if it can be moved in any direction while preserving the error guarantee.

Similar to the fill factor of a page, we divide the specified *error* in 2 parts: the segmentation error e (error used to segment the data), and an insert budget ε (number of locations a key can be moved in any direction). To preserve the specified error, we require that $error = e + \varepsilon$. By keeping an insert budget for each page, FITING-TREE can ensure that inserting a new element will not violate the error for the page.

More specifically, given a segment s, the page has a total size of $|s| + 2\varepsilon$ ($|s|$ is the number of locations in the segment). Data is placed in the middle of the new page, yielding ε empty locations at the beginning and end of the page.

With this strategy, it is possible to move any key in a direction which has free space without violating the error condition. Therefore, to insert a new item using an in-place insert strategy, FITING-TREE first locates the position in the page where the new item belongs. Then, depending on which end of the page (left or right) is closer, all elements are shifted (either left or right) into the empty region of the page. Once all of the empty spaces are filled, the segment needs to be re-approximated (using the segmentation algorithm described in Section 3). If the segmentation algorithm produces more than one segment, we create n new segments (where n is the number of segments produced after running Algorithm 1 on the single segment that is now full). Finally, each new segment is inserted into the upper level tree, and any references to the old segment are deleted.

Algorithm 3 Delta Insert Algorithm

INSERTKEY(*tree*, *key*)

```
1  seg ← SearchTree (tree.root, key)
2  seg.buffer.insert(key)
3  if seg.buffer.isFull()
4     then
5        segs = SEGMENTATION(seg.data, seg.buffer)
6        for s ∈ segs
7           do tree.insert(s)
8        tree.remove(seg)
9  return
```

5.2 Delta Insert Strategy

Since the segments in FITING-TREE are of variable size, the cost of an insert operation using the previously described in-place insertion strategy can be high, particularly for large error thresholds or uniform data that produces large segments with many data items. Specifically, on average, $\frac{|s|}{2}$ keys may need to be moved for a single insert operation, where $|s|$ is the number of locations in the segment. Therefore, a better approach for inserting new data items into a FITING-TREE should amortize the cost of moving keys in the segment.

To reduce the overhead of moving data items inside a page when inserting a new item, each segment in a FITING-TREE contains an additional fixed-size buffer instead of extra space at each end. More specifically, as shown in Algorithm 3, new keys are added to the buffer portion of the segment for which the key belongs to (line 2). This buffer is kept sorted to enable efficient search and merge operations.

Once the buffer reaches its predetermined size (*buff*), it is combined with the data in the segment and then re-segmented using the previously described segmentation algorithm (Algorithm 1) to create a series of valid segments that satisfy the error threshold (line 5). Note that depending on the data, the number of segments after this process can be one (i.e., the data inserted into the buffer does not violate the error threshold) or several. Finally, each of the new segments generated from the segmentation process are inserted into the tree (line 6-7) and the old page is removed (line 8).

Storing additional data inside a segment impacts how to locate a given item, as well as how the error is defined. Since adding a buffer for each segment can violate the error guarantees that FITING-TREE provides, we transparently incorporate the buffer's size into the error parameter for the segmentation process. More formally, given a specified error of *error*, we transparently set the error threshold for the segmentation process to (*error* − *buff*). This ensures that a lookup operation will satisfy the specified error even if the element is located in the buffer.

The overall runtime for inserting a new element into a FITING-TREE is the time required to locate the segment and add the element to the segment's buffer. With p pages stored

in a FITING-TREE, and a fanout of b (i.e., number of keys in each internal separator node), inserting a new key into a FITING-TREE has the following runtime:

$$insert\ runtime : O(log_b p) + O(buff) \qquad (4)$$

Note that when the buffer is full and the segment needs to be re-segmented, the runtime has an additional cost of $O(d)$, where d is the sum of a segment's data size and buffer size. Additionally, if the write-rate is very high, we could also support merging algorithms that use a second buffer similar to how column stores merge a write-optimized delta to the main compressed column. However, this is an orthogonal consideration that heavily depends on the read/write ratio of a workload and is outside the scope of this paper.

6 COST MODEL

Since the specified error threshold affects both the performance of lookups and inserts as well as the index's size, the natural question follows: how should a DBA pick the error threshold for a given workload? To navigate this tradeoff, we provide a cost model that helps a DBA pick a "good" error threshold when creating a FITING-TREE. At a high level, there are two main objectives that a DBA can optimize: performance (i.e., lookup latency) and space consumption [8, 15]. Therefore, we present two ways to apply our cost model that help a DBA choose an error threshold.

6.1 Latency Guarantee

For a given workload, it is valuable to be able to provide latency guarantees to an application. For example, an application may require that lookups take no more than a specified time threshold (e.g., 1000ns) due to SLAs or application-specific requirements (e.g., for interactive applications). Since FITING-TREE incorporates an error term that in turn affects performance, we can model the index's latency in order to pick an error threshold that satisfies the specified latency requirement.

As discussed, lookups require finding the relevant segment and then searching the segment (data and buffer) for the element. Since the error threshold influences the number of segments that are created (i.e., a smaller error threshold yields more segments), we use a function that returns the number of segments created for a given dataset and error threshold. This function can either be learned for a specific dataset (i.e., segment the data using different error thresholds) or a general function can be used (e.g., make the simplifying assumption that the number of segments decreases linearly as the error increases). We use S_e to represent the number of resulting segments for given dataset using an error threshold of e.

Therefore, the total estimated lookup latency for an error threshold of e can be modeled by the following expression,

where b is the tree's fanout, *buff* is a segment's maximum buffer size, and c is a constant representing the latency (in ns) of a cache miss on the given hardware (e.g., $50ns$). Moreover, the cost function assumes binary search for the area that needs to be searched within a segment bounded by e as well as searching the complete buffer.

$$\text{LATENCY}(e) = c \underbrace{\left[log_b(S_e) + \underbrace{log_2(e)}_{\text{Segment Search}} + \underbrace{log_2(buff)}_{\text{Buffer Search}} \right]}_{\text{Tree Search}} \quad (5)$$

Setting c to a constant value implies that all random memory accesses have a constant penalty but caching can often change the penalty for a random access. In theory, instead of being a constant, c could be a function that returns the penalty of a random access but we make the simplifying that c is a constant.

Using this cost estimate, the index with the smallest storage footprint that satisfies the given latency requirement L_{req} (in nanoseconds) is given by the following expression, where E represents a set of possible error values (e.g., $E = \{10, 100, 1000\}$) and SIZE is a function that returns the estimated size of an index (defined in the next section).

$$e = \underset{\{e \in E \; \mid \; \text{LATENCY}(e) \leq L_{req}\}}{\arg \min} \left(\text{SIZE}(e) \right) \quad (6)$$

In addition to modeling the latency for lookup operations, we can similarly model the latency for insert operations. However, there are a few important differences. First, inserts do not have to probe the segment. Also, instead of searching a segment's buffer, inserts require adding the item to the buffer in sorted order. Finally, we must also consider the cost associated with splitting a full segment.

6.2 Space Budget

Instead of specifying a lookup latency bound, a DBA can also give FITing-Tree a storage budget to use. In this case, the goal becomes to provide the highest performance (i.e., lowest latency for lookups and inserts) while not exceeding the specified storage budget.

More formally, we can estimate the size of a read-only clustered index (in bytes) for a given error threshold of e using the following function, where again S_e is the number of segments that are created for an error threshold of e, and b is the fanout of the tree.

$$\text{SIZE}(e) = \underbrace{(S_e \cdot log_b(S_e) \cdot 16B)}_{\text{Tree}} + \underbrace{(S_e \cdot 24B)}_{\text{Segment}} \quad (7)$$

The first term is a pessimistic bound on the storage cost of the tree (leaf + internal nodes using 8 byte keys/pointers), while the second term represents the added metadata about each segment (i.e., each segment has a starting key, slope, and pointer to the underlying data, each 8 bytes).

Therefore, the smallest error threshold that satisfies a given storage budget S_{req} (given in bytes) is given by the following expression where again E represents a set of all possible error values (e.g., $E = \{10, 100, 1000\}$).

$$e = \underset{\{e \in E \; \mid \; \text{SIZE}(e) \leq S_{req}\}}{\arg \min} \left(\text{LATENCY}(e) \right) \quad (8)$$

As we show in Section 7.7, our cost model can accurately estimate the size of a FITing-Tree over real-world datasets, providing DBAs with a valuable way to balance performance (i.e., latency) with the storage footprint of a FITing-Tree.

7 EVALUATION

This section evaluates FITing-Tree and the presented techniques. Overall, we see that FITing-Tree achieves comparable performance to a full index as well as indexes that use fixed-size paging while using orders of magnitude less space.

First, in Section 7.1, we compare the overall performance of FITing-Tree, measuring its lookup and insert performance for both clustered and non-clustered indexes using a variety of real-world datasets. Next, we compare the two proposed insert strategies in Section 7.2. Then, in Section 7.3, we measure the construction cost of FITing-Tree and in Section 7.4 we show how FITing-Tree can leverage other internal index structures. Section 7.5 shows the scalability of our index for different dataset sizes. Finally, Section 7.6 shows how FITing-Tree performs for an adversarial synthetically generated dataset (i.e., worst-case data distribution) and Section 7.7 evaluates our cost model.

Appendix B includes additional experiments that compare FITing-Tree to Correlation Maps [29], show results for range queries, measure the throughput for various buffer sizes, and breakdown the lookup performance of FITing-Tree.

We conducted all experiments on a single server with an Intel E5-2660 CPU (2.2GHz, 10 cores, 25MB L3 cache) and 256GB RAM and all index and table data was held in memory.

7.1 Exp. 1: Overall Performance

In the following, we evaluate the overall lookup and insert performance of FITing-Tree. For these comparisons, we benchmark FITing-Tree against both a full index (i.e., a dense index) as well as an index that uses fixed-size pages (i.e., a sparse index). A full index can be seen as best case baseline for lookup performance and thus gives us an interesting reference point.

For the two baselines (full and fixed-size paging), we use a popular B+ tree implementation (STX-tree [4] v0.9) since our FITing-Tree prototype also uses this tree to index the variable sized segments. Importantly, as we show in Section 7.4, any other tree implementation can also serve as the organization layer.
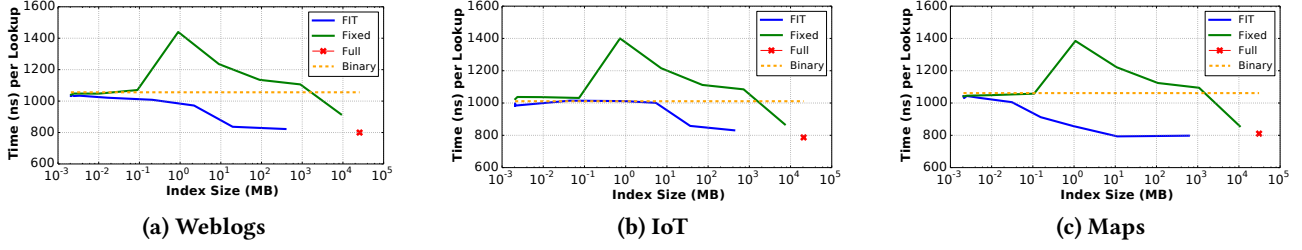
(a) Weblogs   **(b) IoT**   **(c) Maps**

Figure 6: Latency for Lookups (per thread)

(a) Weblogs   **(b) IoT**   **(c) Maps**
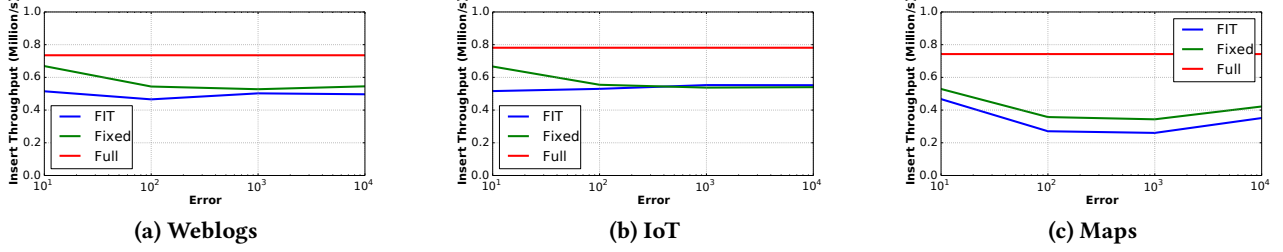
Figure 7: Throughput for Inserts (per thread)

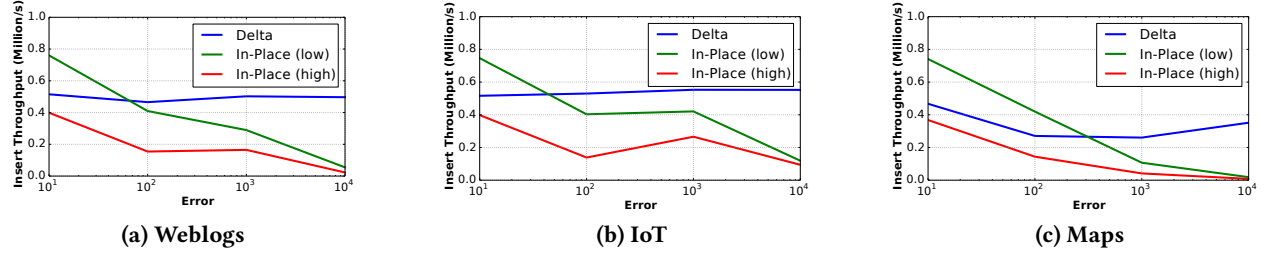**(a) Weblogs**   **(b) IoT**   **(c) Maps**

Figure 8: Insertion Strategy Microbenchmark

7.1.1 Datasets. Since performance of our index depends on the distribution of elements in a given dataset, we evaluate FITing-Tree on real-world datasets with different distributions. Later, in Section 7.6, we show that our techniques are still valuable using a synthetically generated worst-case dataset. For our evaluation, we use three different real-world datasets, each with very different underlying data distributions: (1) Weblogs [1], (2) IoT [1], and (3) Maps [3].

The Weblogs dataset contains $\approx 715M$ log entries for every web request to the CS department at a university over the past 14 years. This dataset contains subtle trends, such as the fact that more requests occur during certain times (e.g., school year vs. summer, day vs. night). On the other hand, the IoT dataset contains $\approx 5M$ readings from around 100 different IoT sensors (e.g., door, motion, power) installed throughout an academic building at a university. Since these sensors generally reflect human activity, this dataset has interesting patterns, such as the fact there is more activity during certain hours because classes are in session. For each of these datasets, we create a *clustered* FITing-Tree using the timestamp attribute (e.g., the time at which a resource was requested). Finally, the Maps dataset contains the longitude

of $\approx 2B$ features (e.g., museums, coffee shops) across the world. Unsurprisingly, the longitude of locations is relatively linear and does not contain many periodic trends. Unlike the previous two datasets, we create a *non-clustered* FITing-Tree over the longitude attribute of this dataset.

For our approach, the most important aspect of a dataset that impacts FITing-Tree's performance is the data's periodicity. For now, think of the periodicity as the distance between two "bumps" in a stepwise function that maps keys to storage locations as shown in Figure 13a (blue line). If the specified error of FITing-Tree is larger than the periodicity (green line), the segmentation results in a single segment. However, if the error is smaller than the periodicity (red line), we need multiple segments to approximate the data distribution.

Therefore, we define a *non-linearity ratio* to show the periodicity of a dataset. To compute this ratio, we first calculate the number of segments required to cover the dataset for a given error threshold. We then normalize this result by the number of segments required for a dataset of the same size with periodicity equal to the error (which is the worst case, or the most "non-linear" in that scale).

To show that all datasets contain a distinct periodicity pattern, Figure 9 plots the non-linearity ratio of each of dataset. The IoT dataset has a very significant bump, signifying that there is very strong periodicity the scale of 10^4, likely due to patterns that follow human behavior (e.g., day/night hours). Weblogs has multiple bumps which are likely correlated to different periodic patterns (e.g., daily, weekly, and yearly patterns). The Maps dataset, unlike the others, is linear at small scales (but has stronger periodicity at larger scales).

7.1.2 Lookups. The first series of benchmarks show how FITing-Tree compares to (1) a full index (i.e., dense), (2) an index that uses fixed-size paging (i.e., sparse), and (3) binary search over the entire dataset. We include binary search since it represents the most extreme case where the error is equal to the data size (i.e., our segmentation creates one segment). For the Weblog and IoT dataset, we created a clustered index using the timestamp attribute which is the primary key of these datasets. We created a non-clustered (i.e., secondary) index over the longitude attribute of the Maps dataset, which is not unique.

The results (Figure 6) show the lookup latency for various sizes of the index for the Weblogs (scaled to 1.5B records), IoT (scaled to 1.5B records), and Maps (not scaled, 2B records) datasets. More specifically, since the size of both FITing-Tree and the fixed-size paging baseline can be varied (i.e., FITing-Tree's error term and the page size influence the number of leaf-level entries), we show the performance of each of these approaches for various index sizes. Note that the size of a full index cannot be varied and is therefore a single point in the plot. Additionally, since binary search does not have any additional storage requirement, its size is zero but is visualized as a dotted line.

In general, the results show that FITing-Tree always performs better than an index that uses fixed-size paging. Most importantly, however, FITing-Tree offers significant space savings compared to both fixed-size paging and a full index. For example, in the Maps dataset, FITing-Tree is able to match the performance of a full index using only $609MB$ of memory, while a full index consumes over $30GB$ of space. Moreover, compared to a tree that uses fixed-size paging, a FITing-Tree which consumes only $1MB$ of memory is able to match the performance of a fixed-size index which consumes over $10GB$ of memory, offering a space savings of four orders of magnitude.

As expected, for very small index sizes (i.e., very large pages or a high error threshold), both FITing-Tree and fixed-size paging mimic the performance of binary search since there are only a few pages that are searched using binary search. On the other hand, as the index grows, the performance of both fixed-size paging as well as FITing-Tree converge to that of a full index due to the fact that pages contain
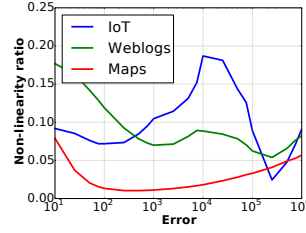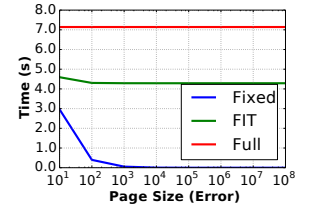

Figure 9: Non-linearity   **Figure 10: Build Time**

very few elements. Note that the spike in the graph for the fixed-size index is due to the fact that the index begins to fall out of the CPU's L2 cache.

Finally, as expected, the data distribution impacts the performance of FITing-Tree. More specifically, we can see that FITing-Tree is able to more quickly match the performance of a full tree with the Maps dataset, compared to the Weblogs and IoT datasets. This is due to the fact that the Maps dataset is relatively linear, when compared to the Weblogs and IoT datasets (shown in Figure 9).

7.1.3 Inserts. Next, we compare the insert performance of FITing-Tree to both a full index, as well as an index that uses fixed-size paging as previously described. To insert new items, FITing-Tree uses the previously described delta insert technique since it provides the best overall performance, which we show in next section where we perform an in-depth comparison of the delta and the in-place insert strategies.

More specifically, to ensure a fair comparison and that FITing-Tree is not unfairly write-optimized, we set the size of the delta buffer to half of the specified error (i.e, for an error threshold of 100, the underlying data is segmented using an error of 50 and each segment's buffer has a maximum size of 50 elements). Similarly, for the index with fixed-size pages, the page size is given by the half of the error threshold we used for FITing-Tree and the same amount (i.e., half of the error used in FITing-Tree) is used as the buffer size. As usual, once the buffer is full, the page is split into two pages. The results, shown in Figure 7, compare the throughput of each index after building the index over the first half of the data and inserting the second half for various error thresholds using shuffled versions of the previously described datasets. As shown, FITing-Tree is able to achieve insert performance that is, in general, comparable to an index that uses fixed-size paging. Unsurprisingly, a full B+ tree is able to handle a higher write load than either a FITing-Tree or an index that uses fixed-size paging since both need to periodically split pages that become full. Additionally, FITing-Tree needs to execute the segmentation algorithm, explaining the performance gap between FITing-Tree and fixed-size paging.

Interestingly, FITing-Tree is faster than fixed-size paging in some cases since the error determines the number of segments in the tree. For a large error, there typically are fewer
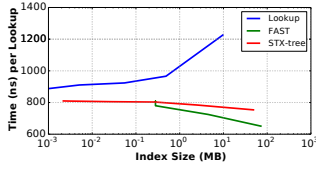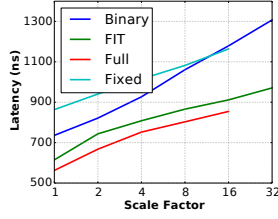
Figure 11: Other Indexes   Figure 12: Scalability

segments generated, which reduces the number of times that FITing-Tree needs to merge the buffer with the segment's data and execute the segmentation algorithm.

7.2 Exp. 2: In-place vs. Delta Inserts

In the following, we compare the two insert strategies described in Section 5 and show how they perform for various datasets, fill factors, and error thresholds.

Figure 8 shows the insert throughput for both the delta and in-place insert strategies for each of the previously described datasets. As mentioned, for in-place inserts, the amount of free space reserved at the beginning and end of page (ε) can be tuned based on the read/write characteristics of the workload. Therefore, we now show results for both a low (i.e., ε is 25% of the error) and high (i.e., ε is 75% of the error) fill factor. For the delta insert results, we use the same setup as the previously described insert experiment (i.e., we set the size of the delta buffer to half of the specified error).

As shown, the delta insert strategy generally offers the highest insert throughput for error thresholds higher than 100. This is due to the fact that the in-place insert strategy must move many data items when inserting a new item since segments created with a higher error threshold contain more keys. However, for low error thresholds, the in-place insert strategy outperforms the delta strategy, since there are significantly fewer data items that need to be shifted.

The fill factor impacts (1) how many data items are in a given segment and must be copied when inserting a new element, and (2) how often a segment fills up and needs to be re-segmented. As shown, in general, the in-place strategy with a low fill factor offers the highest insert performance.

7.3 Exp 3: Index Construction

In the following, we quantify the cost to construct a FITing-Tree. More specifically, we measure the amount of time required to bulk load a FITing-Tree and compare it to the time required to construct a B+ tree that uses fixed-size pages as well as a full index. The results, shown in Figure 10, plot the runtime for each approach for various page sizes (Fixed B+ tree) or error thresholds (FITing-Tree) using the Weblogs dataset.

Since a full index must insert every element into the tree, it takes a constant amount of time. However, the time required to bulk load a B+ tree that uses fixed-size pages decreases as the page size increases since only the page boundaries (e.g., every 100th element) must be accessed from the underlying data. As previously mentioned, building a FITing-Tree requires first segmenting the data, and then inserting each segment into the underlying tree structure. Unsurprisingly, since the segmentation algorithm must examine every element in the data, a FITing-Tree incurs a constant amount of extra overhead (≈ 4.2 seconds in the shown experiment) compared to a B+ tree that uses fixed-size pages.

Interestingly, however, this constant amount extra overhead can be avoided in some cases. For example, when initially loading the data into the system, it is possible to execute the segmentation algorithm in a streaming fashion. In this case, it is possible that building a FITing-Tree is actually less expensive than building a B+ tree with fixed-size pages since the resulting segmentation algorithm leverages trends in the data to produce fewer lead-level entries.

7.4 Exp 4: Other Indexes

As mentioned, FITing-Tree internally uses STX-tree [4] to organize the variable sized pages generated through our segmentation process (Section 3). However, depending on the workload characteristics (e.g., read/write ratio), other internal data structures could also be used to index segments to provide additional performance improvements.

Therefore, to show that our techniques are generalizable, we compare using an STX-tree to (1) FAST [28], a highly optimized index structure for read-only workloads and (2) a simple lookup table that simply stores the variable sized pages in sorted order. Figure 11 shows the results for various index sizes (i.e., error thresholds) over a subset of the Weblogs dataset (since FAST requires a power of two elements). As shown, a lookup table provides superior space savings (since there are no internal separator nodes) but with a higher lookup latency. On the other hand, a FITing-Tree that internally uses the highly optimized FAST index can provide faster lookups but often consumes more space. Therefore, a FITing-Tree is able to leverage alternative indexes that may be more performant depending on the workload characteristics (e.g., read-only).

7.5 Exp. 5: Data Size Scalability

To evaluate how FITing-Tree performs for various dataset sizes, we measure the lookup latency for the Weblogs dataset for various scale factors where both the error threshold and fixed page size are set to 100, which is optimal for this dataset.
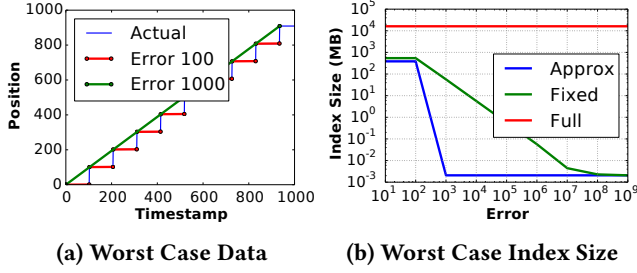
(a) Worst Case Data   (b) Worst Case Index Size

Figure 13: Worst Case Analysis

(a) Latency   (b) Size

Figure 14: Cost Model Accuracy

Since the performance of our index depends on the underlying data distribution, we scale the dataset while maintaining the underlying trends. We omit the result for all other datasets here (IoT and Maps) since they follow similar trends.

Figure 12 shows that the indexes (i.e., FITing-Tree, a full index, fixed-size paging) scale better than binary search due to the better theoretical asymptotic runtime ($log_b(n)$ vs. $log_2(n)$) and cache performance. Additionally, FITing-Tree's performance over various dataset sizes closely follows that of a full index which offers the best performance, demonstrating that our techniques offer valuable space savings without sacrificing performance. Most importantly, neither a full index nor an index that uses fixed-size paging could scale to a scale factor of 32 since the size of the index exceeded the amount of available memory. This again shows that FITing-Tree is able to offer valuable space savings.

7.6 Exp 6: Worst Case Data

Since the data distribution influences the performance of FITing-Tree, we synthetically generated data to illustrate how our index performs with data that represents a worst-case. We define the worst case as as a dataset which maximizes the number of segments given a specific error, described further in Section 3.2. To do this, we generate data using a step function with a fixed step size of 100, as shown in Figure 13a. Since the step size is fixed, an error threshold less than the step size yield a single segment per step. However, given an error threshold larger than the step size, our segmentation algorithm will be able to use a single segment to represent the entire dataset.

Figure 13b shows the performance for various sizes of each index built over this worst case dataset. As shown, for error thresholds of less than 100, the size of a FITing-Tree is the same as a fixed-size index but still smaller than a full index. This is due to the fact that for the error thresholds less than the step size, FITing-Tree creates segments of size 100 (step size), resulting in a large number of nodes in the tree. On the other hand, for an error threshold that is larger than 100, FITing-Tree is able to represent the step dataset with only a single segment, dramatically reducing the index's size. As shown, importantly, a FITing-Tree will not contain more
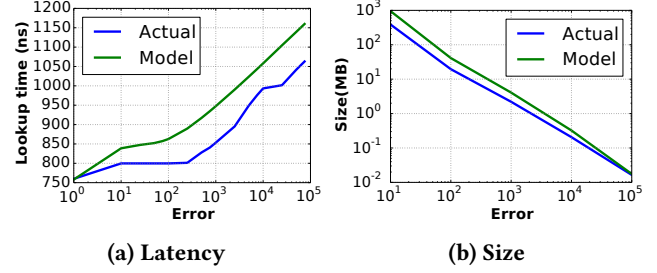
leaf-level entries than an index that uses fixed-size paging, as discussed in Section 3.4.

7.7 Exp. 7: Accuracy of Cost Model

Since, as previously described, the error threshold influences both the latency as well as space consumption of our index, the cost model presented in Section 6 aims to guide a DBA when determining what error threshold to use for a FITing-Tree. More specifically, given a latency requirement (e.g., 1000*ns*) or a space budget (e.g., 2*GB*), our cost model automatically determines an appropriate error threshold that satisfies the given constraint.

Figure 14a shows the estimated and actual lookup latency for various error thresholds on the Weblogs dataset using a value of 50*ns* for *c* (the cost of a random memory access) determined through a memory benchmarking tool on the given hardware. As shown, our latency model predicts an accurate upper bound for the actual latency of a lookup operation. Our model slightly overestimates the latency due to the fact that it does not incorporate CPU caching effects. Since we overestimate the cost, we ensure that a specified latency threshold will always be observed.

To evaluate our size cost model, we show the predicted and actual size of a FITing-Tree for various error thresholds in Figure 14b. As shown, our model is able to accurately predict the size of an index for a given error threshold while ensuring that our estimates are pessimistic (i.e., the estimated cost is higher than the true cost).

8 RELATED WORK

The presented techniques in this paper overlap with work in different areas including (1) index compression, (2) partial/adaptive indexes, and (3) function approximation.

Index Compression: Since B+ trees can often consume significant space, several index compression techniques have been proposed. These approaches reduce the size of keys in internal nodes by applying techniques such as prefix/suffix truncation, dictionary compression, and key normalization [21, 23, 36]. Importantly, these techniques can also be applied within FITing-Tree to further reduce the size of the underlying tree structure.

Similar to B+ tree compression, several methods have been proposed in order to more compactly represent bitmap indexes [6, 11, 26, 39, 42, 45]. Many of these techniques are specific to bitmap indexes, which are primarily only useful for attributes with few distinct values and not the general workloads that FITing-Tree targets.

Correlation Maps [29] try to leverage correlations between an unclustered attribute and a clustered attribute when an existing primary key index exists. Our approach, on the other hand, does not assume an existing index already exists and uses variable sized paging (instead of fixed-sized buckets) that better model the underlying data.

FAST [28] is another more recent tree structure that organizes tree elements in a more compact and efficient representation in order to exploit modern hardware features (e.g., SIMD, cache line size) for read-heavy workloads. Similarly, an Adaptive Radix Tree (ART) [32] leverages CPU caches for in-memory indexing. Another idea discussed in [48] are hybrid indexes which separate the index into hot and cold regions where cold data is stored in a compressed format. Lastly, Log-structured Merge-trees [37] are designed for mostly write intensive workloads and extensions including Monkey [15] balance performance and memory consumption. Each of these techniques can be seen as orthogonal and thus also could be used by FITing-Tree to more efficiently store the underlying tree structure as well as optimize for read-heavy workloads or hot/cold data.

Other indexing techniques have been proposed that store information about a region of the dataset, instead of the indexing individual keys. For example, leaf nodes in a BF-Tree [7] are bloom filters. Unlike FITing-Tree, BF-Tree does not exploit properties about the data's distribution when segmenting a dataset. Another example are learned indexes [30], which aim to learn the underlying data distribution to index data items. Unlike learned indexes, FITing-Tree has strict error guarantees, supports insert operations, and provides a cost model to ensure predictable performance and size.

Sparse indexes like Hippo [47], Block Range Indexes [44], and Small Materialized Aggregates [34] all store information about value ranges similar to the idea of segments in FITing-Tree. However, these techniques do not consider the underlying data distribution or bound lookup/insert latency.

Finally, several approximation techniques have been proposed in order to improve the performance of similarity search [19, 24, 40] (for string or multimedia data), unlike FITing-Tree which uses approximation for compressing indexes optimized for traditional point and range queries.

Partial and Adaptive Indexes: Partial indexes [43] aim to reduce the storage footprint of an index since they index only a subset of the data that is of interest to the user. For example, Tail Indexes [14, 20] store only rare data items in order to reduce the storage footprint of the overall index. FITing-Tree, on the other hand, supports queries over all attribute values but could be extended to index only "important" data ranges as well. Furthermore, database cracking [25] is a technique that physically reorders values in a column in order to more efficiently support selection queries without needing to store secondary indexes. Since database cracking reorganizes values based on past queries, it does not efficiently support ad-hoc queries, like FITing-Tree can.

Function Approximation: The main idea of a FITing-Tree is to approximate the data distribution using piece-wise linear functions and approximating curves using piece-wise functions is not new [10, 16, 18, 33]. The error metrics E_2 (integral square error) and E_∞ (maximal error) for these approximations have been discussed as well as different segmentation algorithms [17, 35, 38]. Unlike prior work, we consider only monotonic increasing functions, E_∞, and potentially disjoint linear segments. Moreover, none of these techniques have been applied to indexing and therefore do not consider looking up or inserting data items.

More recent work [12, 13, 27, 41, 46] specific to time series data also leverages piece-wise linear approximations to store patterns for similarity search. While these approaches also trade-off the number of segments with the accuracy of the approximate representation, they do not aim to provide the lookup and space consumption guarantees that FITing-Tree does, and do not have the analysis related to these guarantees.

Finally, other work [5] leverages piece-wise linear functions to compress inverted lists by storing functions and the distances of elements from the extrapolated functions. However, these approximations use linear regression (which minimizes E_2), and there are no bounds on the error.

9 CONCLUSION

In this paper, we introduced FITing-Tree, a new index structure that incorporates a tunable error parameter to allow a DBA to balance lookup performance and space consumption of an index. To navigate this tradeoff, we presented a cost model that determines an appropriate error parameter given either (1) a lookup latency requirement (e.g., 500*ns*) or (2) a storage budget (e.g., 100*MB*). We evaluated FITing-Tree using several real-world datasets and showed that our index can achieve comparable performance to a full index structure while reducing the storage footprint by orders of magnitude.

10 ACKNOWLEDGEMENTS

REFERENCES

[1] 2019. A Benchmark for Machine-generated Data Management. https://github.com/BrownBigData/MgBench.

[2] 2019. NYC Taxi & Limousine Commission Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[3] 2019. OpenStreetMap database. https://aws.amazon.com/public-datasets/osm.

[4] 2019. STX B+ Tree. https://panthema.net/2007/stx-btree/.

[5] Naiyong Ao et al. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *VLDB* (2011), 470–481.

[6] Manos Athanassoulis et al. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*. 1319–1332.

[7] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: Approximate Tree Indexing. In *VLDB*. 1881–1892.

[8] Manos Athanassoulis and Stratos Idreos. 2016. Design Tradeoffs of Data Access Methods. In *SIGMOD*. 2195–2200.

[9] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-trees. *ACM Trans. Database Syst.* (1977), 11–26.

[10] Dietrich Braess. 1971. Chebyshev Approximation by Spline Functions with Free Knots. *Numerische Mathematik* (1971), 357–366.

[11] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD*. 355–366.

[12] Lu Chen et al. 2017. Efficient Metric Indexing for Similarity Search and Similarity Joins. In *KDE*. 556–571.

[13] Tak chung Fu. 2011. A Review on Time Series Data Mining. *Engineering Applications of Artificial Intelligence* (2011), 164 – 181.

[14] Andrew Crotty et al. 2016. The Case for Interactive Data Exploration Accelerators (IDEAs). In *HILDA*. 11:1–11:6.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*. 79–94.

[16] Frank Eichinger et al. 2015. A Time-series Compression Technique and Its Application to the Smart Grid. *The VLDB Journal* (2015), 193–218.

[17] Hazem Elmeleegy et al. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *VLDB* (2009), 145–156.

[18] R.E Esch and W.L Eastman. 1969. Computational Methods for Best Spline Function Approximation. *Journal of Approximation Theory* (1969), 85 – 96.

[19] Andrea Esuli. 2012. Use of Permutation Prefixes for Efficient and Scalable Approximate Similarity Search. *Inf. Process. Manage.* (2012), 889–902.

[20] Alex Galakatos et al. 2017. Revisiting Reuse for Approximate Query Processing. In *VLDB*. 1142–1153.

[21] Jonathan Goldstein et al. 1998. Compressing Relations and Indexes. In *ICDE*. 370–379.

[22] Goetz Graefe. 2006. B-tree Indexes, Interpolation Search, and Skew. In *DaMon*.

[23] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *ICDE*. 349–358.

[24] Michael E. Houle and Jun Sakuma. 2005. Fast Approximate Similarity Search in Extremely High-Dimensional Data Sets. In *ICDE*. 619–630.

[25] Stratos Idreos et al. 2007. Database Cracking. In *CIDR*. 68–78.

[26] Theodore Johnson. 1999. Performance Measurements of Compressed Bitmap Indices. In *VLDB*. 278–289.

[27] Eamonn Keogh et al. 2001. An Online Algorithm for Segmenting Time Series. In *ICDM*. IEEE, 289–296.

[28] Changkyu Kim et al. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*. 339–350.

[29] Hideaki Kimura et al. 2009. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *VLDB*, 1222–1233.

[30] Tim Kraska et al. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[31] Domine M. W. Leenaerts and Wim M. Van Bokhoven. 1998. *Piecewise Linear Modeling and Analysis*.

[32] Viktor Leis et al. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*. 38–49.

[33] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. 2008. Novel Online Methods for Time Series Segmentation. *IEEE Trans. on Knowl. and Data Eng.* (2008), 1616–1626.

[34] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[35] Thomas Neumann and Michel Sebastian. 2008. Smooth Interpolating Histograms with Error Guarantees. In *BNCOD*.

[36] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: A RISC-style Engine for RDF. In *VLDB*. 647–659.

[37] Patrick O'Neil et al. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* (1996), 351–385.

[38] T. Pavlidis and S. L. Horowitz. 1974. Segmentation of Plane Curves. *IEEE Trans. Comput.* (1974), 860–870.

[39] Ali Pinar et al. 2005. Compressing Bitmap Indices by Data Reorganization. In *ICDE*. 310–321.

[40] K. V. Ravi Kanth et al. 1998. Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *SIGMOD*. 166–176.

[41] Hagit Shatkay and Stanley B Zdonik. 1996. Approximate Queries and Representations for Large Data Sequences. In *ICDE*. IEEE, 536–545.

[42] Michał Stabno and Robert Wrembel. 2009. RLH: Bitmap Compression Technique Based on Run-length and Huffman Encoding. *Inf. Syst.* (2009), 400–414.

[43] Michael Stonebraker. 1989. The Case for Partial Indexes. *SIGMOD Record* (1989), 4–11.

[44] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *SIGMOD*. 340–355.

[45] Kesheng Wu et al. 2006. Optimizing Bitmap Indices with Efficient Compression. *ACM Trans. Database Syst.* (2006), 1–38.

[46] Zhenghua Xu et al. 2012. An Adaptive Algorithm for Online Time Series Segmentation with Error Bound Guarantee. In *EDBT*.

[47] Jia Yu and Mohamed Sarwat. 2016. Two Birds, One Stone: A Fast, Yet Lightweight, Indexing Scheme for Modern Database Systems. In *VLDB*. 385–396.

[48] Huanchen Zhang et al. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. 1567–1581.

[49] Marcin Zukowski et al. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59–.

A SEGMENTATION ANALYSIS

In the following we provide additional information about our segmentation algorithm, SHRINKINGCONE, described in Section 3. First, we prove the minimum size of a segment produced by our algorithm. Then, although efficient in practice, we show that our algorithm can be arbitrarily worse than an optimal algorithm when considering the number of segments it produces.

A.1 SHRINKINGCONE Segment Size

We prove the claim from Theorem 3.1 regarding the size of a maximal linear segment.

PROOF. Consider 3 arbitrary points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, where $x_1 < x_2 < x_3$ and $y_1 < y_2 < y_3$. By definition, the linear function starts at the first point in a segment, and ends at the last point in the segment. The linear segment is not valid if the distance on the y axis (loc) is larger than the specified error. Therefore, given the 3 points, a linear segment starting at (x_1, y_1) and ending at (x_3, y_3) is not feasible if:

$$y_2 - err > \frac{y_3 - y_1}{x_3 - x_1}(x_2 - x_1) + y_1 \qquad (9)$$

By rearranging the inequality we get:

$$err < y_2 - y_1 - \frac{y_3 - y_1}{x_3 - x_1}(x_2 - x_1) \qquad (10)$$

$$= (y_3 - y_1) \cdot \left(1 - \frac{x_2 - x_1}{x_3 - x_1}\right) - (y_3 - y_2) \qquad (11)$$

$$\leq (y_3 - y_1) \cdot \left(1 - \frac{x_2 - x_1}{x_3 - x_1}\right) - 1 \qquad (12)$$

In (11) y_3 was added and subtracted, and in (12) we use the fact that y_2 and y_3 are integers (thus $y_3 - y_2 \geq 1$). This provides a lower bound for the distance between the first point in a segment and the first point in the following segment:

$$y_3 - y_1 > \frac{err + 1}{1 - \frac{x_2 - x_1}{x_3 - x_1}} = (err + 1) \cdot \frac{x_3 - x_1}{x_3 - x_2} > err + 1 \qquad (13)$$

$$\Rightarrow y_3 - y_1 > err + 1 \qquad (14)$$

Since (x_3, y_3) is the first point outside of the segment, the number of locations in the segment is $y_3 - 1 - y_1 \geq err + 1$. □

A.2 SHRINKINGCONE Competitive Analysis

In the following, we prove that SHRINKINGCONE can be arbitrarily worse than the optimal solution when considering the number of segments produced (i.e., SHRINKINGCONE is not competitive).

PROOF. Given the error threshold $E = 100$, consider the following input to SHRINKINGCONE:

(1) 3 keys $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ where $y_1 = 1, y_2 = 2, y_3 = 3$ and $x_3 - x_2 = x_2 - x_1 = \frac{E}{2}$ (this is step 1 in Figure 15).

(2) The key $x_4 = x_3 + \frac{1}{E}$ repeated $E + 1$ times (using $E + 1$ consecutive locations), and the key $x_5 = x_4 + \frac{1}{E}$ without repetitions (using 1 location).
After that repeat for $i \in [1, N]$ the following pattern: the key $x_{2(i+2)} = x_{2(i+2)-1} + E$ repeated $E + 1$ times, and a single appearance of the key $x_{2(i+2)+1} = x_{2(i+2)} + \frac{1}{E}$ (this is step 2 in Figure 15).

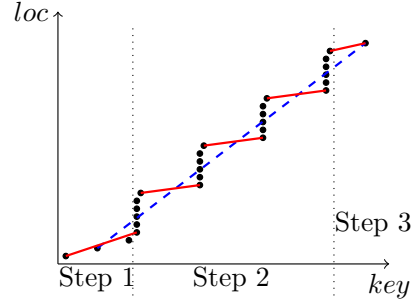(3) The key $x_{2(N+1+2)} = x_{2(N+1+2)-1} + \frac{E}{2}$ (step 3 in Figure 15).

Figure 15: Competitive analysis sketch: the dots are the input, the dashed line is the optimal segmentation (the first dot is a segment), and the solid lines are the segments created by SHRINKINGCONE.

The algorithm will then create the following segments (an illustration is shown in Figure 15):

• $[x_1, x_4]$ (with slope $\frac{3}{E + \frac{1}{E}}$): adding the key x_5 will result in the slope $\frac{3+E+1}{E+\frac{2}{E}}$ which will not satisfy the error requirement for x_4, $y_1 + \frac{3+E+1}{E+\frac{2}{E}} \cdot (x_4 - x_1) - y_4 = 1 + \frac{3+E+1}{E+\frac{2}{E}} \cdot (E + \frac{1}{E}) - 4 = 100.98 > E$.

• Each of the next segments will contain exactly two keys (where the first key appears once, and the second key appears $E + 1$ times), since otherwise the error for the second key will be $\frac{1+E+1}{E+\frac{1}{E}} \cdot E - 1 = 100.98 > E$. Just like before, the $E + 1$ repetitions of a single key will cause a violation of the error (due to the spacing between subsequent keys).

Therefore, the algorithm will create $N + 2$ segments given this input.

On the other hand, the optimal algorithm will need only 2 segments: the first segment is the first key, and the second segment covers the rest of the input since the line starting at the second key and ending at the last key is never further away than E from any key, due to the construction of the input. The slope of the second segment will be $\frac{3+(N+1)\cdot(E+2)}{E+(N+1)\cdot(E+\frac{1}{E})}$, and the first key in the segment is about $\frac{E}{2}$ away on the x axis from the first repeated key. Since the repeated keys are spaced evenly (distance on the x axis of $E + \frac{1}{E}$), the linear function will not violate the error threshold for any key.

Since N can be arbitrarily large, the algorithm is not competitive. □

B FURTHER EVALUATION

In this section, we present additional experimental results that provide a more in-depth study of FITing-Tree and other approaches. More specifically, we compare FITing-Tree to Correlation Maps [29], show how FITing-Tree performs for
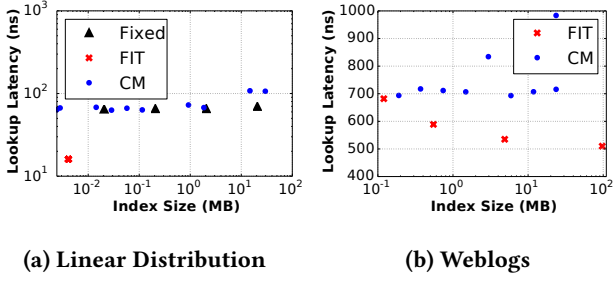
(a) Linear Distribution   (b) Weblogs

Figure 16: Correlation Maps

Figure 17: Range Queries

queries that involve range predicates, show how the buffer size impacts insert throughput, and finally breakdown the lookup latency.

B.1 Correlation Maps

In the following, we compare FITing-Tree to Correlation Maps (CMs) which are designed to exploit correlations that exist between attributes. To ensure a fair comparison and to adapt the described techniques to build a clustered primary index, we assume that each tuple has an implicit position (e.g., ROWID). In addition to the design in the original paper [29] (e.g., bucketing both along the clustered and unclustered dimension), we implemented additional optimizations not described in the paper since our use case has the additional knowledge that the unclustered attribute (i.e., timestamp) is sorted with respect to the clustered attribute (i.e., ROWID). For example, instead of storing several buckets for a given unclustered range (e.g., {100-200} → [b0,b1,b2,b3]) our implementation stores only the first and last bucket (e.g., {100-200} → [b0,b3]). Additionally, when looking up a key, our implementation uses binary search within the entire region instead of searching each bucket individually. We found that these two optimizations improved lookup performance/reduced the size of a CM.

First, to show that CMs are no more efficient for primary indexes than an index that uses fixed-size pages, we consider the simple case of indexing all integer values from 1 to 100M. Although simple, this is not unusual since users often index monotonically increasing identifiers (e.g., customer ID). Figure 16a shows the lookup latency for various index sizes (x-axis) for CMs, FITing-Tree, and fixed-size paging. As previously described, we vary the size of FITing-Tree by selecting various error thresholds and use different page sizes/bucket sizes for CMs and B+ trees that use fixed-size paging. As shown, CMs perform similar to B+ trees with fixed-size pages, since they use fixed-size buckets to partition the attribute domain. FITing-Tree, on the other hand, can use a single segment to represent this data and can locate to the exact position of any element using almost no space.
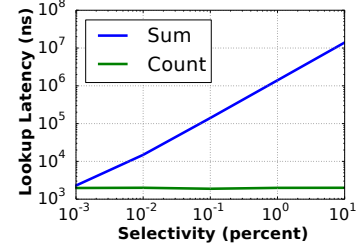
Next, we also used CMs also to build a primary key index on the Weblogs data set and compare it to our FITing-Tree. Figure 16b shows index size (x-axis) vs. the lookup latency (y-axis) for both CMs and FITing-Tree using 400M timestamps from the Weblogs dataset. Since FITing-Tree creates variable-sized segments that better model the underlying data distribution (instead of the fixed-size bucketing approach that CMs use), FITing-Tree is able to provide faster lookup performance using a smaller memory footprint.

B.2 Range Queries

In addition to point queries, FITing-Tree also supports range queries whereby an arbitrary number of tuples must be examined to compute the result. Figure 17 shows the performance of a FITing-Tree for range queries for both a sum and count aggregate for various selectivities using the Weblogs dataset.

Interestingly, to compute the result for a count query, a FITing-Tree can subtract the start position from the end position of the range (i.e., a count aggregate over a range is essentially two point lookups), resulting in a constant lookup latency. On the other hand, computing the sum of an attribute over a range requires examining every tuple in the range, resulting in significantly more work for larger ranges.

B.3 Varying Fill Factor

As previously mentioned, the buffer size of a segment determines the amount of space that a segment reserves to hold new data items. Once the segment's insert buffer reaches this threshold, the data from the segment and the segment's buffer are merged, and FITing-Tree executes the previously described segmentation algorithm to generate new segments that satisfy the specified error threshold.

Therefore, in Figure 18, we vary the buffer size and measure the total throughput using the Weblogs dataset with an error threshold of $e = 20,000$. As shown, the size of the buffer can dramatically impact the write throughput of a FITing-Tree. More specifically, larger buffers result in fewer splitting operations, improving performance. However, a buffer that is too large will result in longer lookup latencies (modeled in the cost model in Section 6).
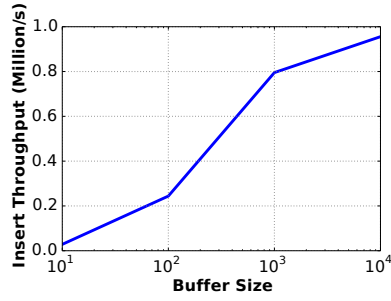
Figure 18: Insert Throughput / Varying Buffer Size

Therefore, the fill factor of an FITing-Tree can be effectively used by a DBA to tune a FITing-Tree to be more read or write optimized, depending on the workload.

B.4 Lookup Breakdown

As described in Section 4, a lookup involves two steps (i.e., locating the segment where a key belongs and then searching the segment's data in order to find the item within the segment). Therefore, we examine the amount of time spent in each of these two steps for FITing-Tree as well as an index that uses fixed-size paging for various error thresholds.

The results in Figure 19 show that in both cases the majority of time is spent searching the tree to find the page where the data item belongs for smaller error thresholds (and page sizes). Since FITing-Tree is able to leverage properties of the underlying data distribution in order to create variable sized segments, the resulting tree is significantly smaller. Therefore, FITing-Tree spends less time searching the tree to find the corresponding segment for a given key.

(a) FITing-Tree
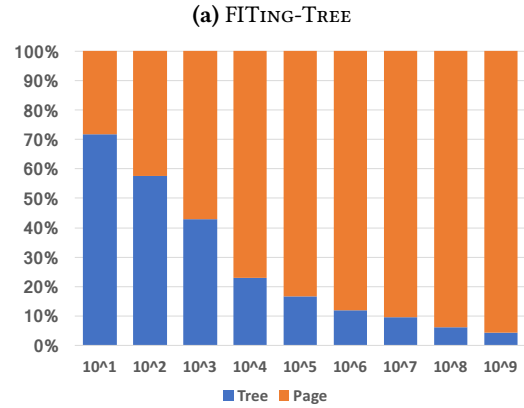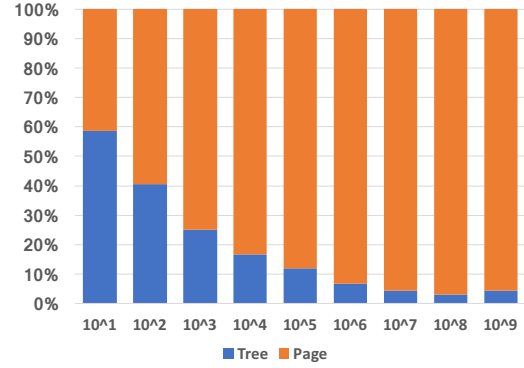
(b) Fixed-size Index

Figure 19: Lookup Breakdown