ALEX: An Updatable Adaptive Learned Index

Jialin Ding[†*] Umar Farooq Minhas[‡] Hantian Zhang[∓*] Yinan Li[‡] Chi Wang[‡]
Badrish Chandramouli[‡] Johannes Gehrke[‡] Donald Kossmann[‡] David Lomet[‡]

[†]MIT, [∓]ETH, [‡]Microsoft

ABSTRACT

Recent work on "learned indexes" has revolutionized the way we look at the decades-old field of DBMS indexing. The key idea is that indexes are "models" that predict the position of a key in a dataset. Indexes can, thus, be learned. The original work by Kraska et al. shows surprising results in terms of search performance and space requirements: A learned index beats a B+Tree by a factor of up to three in search time and by an order of magnitude in memory footprint, however it is limited to static, read-only workloads.

This paper presents a new class of learned indexes called ALEX which addresses issues that arise when implementing dynamic, updatable learned indexes. Compared to the learned index from Kraska et al., ALEX has up to 3000× lower space requirements, but has up to 2.7× higher search performance on static workloads. Compared to a B+Tree, ALEX achieves up to 3.5× and 3.3× higher performance on static and some dynamic workloads, respectively, with up to 5 orders of magnitude smaller index size. Our detailed experiments show that ALEX presents a key step towards making learned indexes practical for a broader class of database workloads with dynamic updates.

1. INTRODUCTION

We are currently living through the age of Software 2.0, which refers to the concept of replacing and augmenting human-written algorithms with machine learning (ML) models. Such software has already shown great results for web search, recommendation, speech understanding and generation, video and image processing, robot control, and self-driving vehicles. Recent work by Kraska et al. [17] has moved this revolution to database systems. Their work, which we will refer to as the Learned Index, proposes to replace a standard database index with a hierarchy of ML models. Given a key, an intermediate node in the hierarchy is a model to predict the child model to use, and a leaf node is a model to predict the location of the key in an array. The models for this *learned index* are trained from the data. The results show that compared to the classic B+Tree, the learned index has over an order of magnitude less memory usage and a factor of up to three faster search. However, their solution can only handle static data. This critical drawback makes it only usable for read-only workloads.

[*]Work performed while at Microsoft Research.

In this work, we introduce a new set of main-memory model-based index structures called ALEX, that work beyond static data. Our goal is to create an index structure that has the lookup time of the existing learned index, but has an update time that is competitive with a B+Tree. In addition, we also aim to maintain the index size to be competitive to the learned index and, thus, much smaller than a B+Tree.

Achieving these goals is not an easy task. Implementing updates requires a careful design of the underlying data structure that keeps the records. Kraska et al. [17] use a sorted array which works well for static datasets but can result in excessive costs for shifting records if new records need to be inserted, since the array is always densely packed. Furthermore, the prediction accuracy of the models can deteriorate as the data distribution changes over time, requiring repeated retraining. To address these challenges, this paper makes the following technical contributions:

- We use a node per leaf layout with two different storage structures for the leaf nodes that allow us to efficiently insert data: A gapped array (optimized for search), and the Packed Memory Array [6] (which balances update and search performance). Both structures trade space for time, by stretching the space to store the keys thereby leaving gaps for inserting new data. This strategy amortizes the cost of shifting the array for each insertion. As more keys are inserted, the structures adaptively expand themselves to prepare for more insertions.

- As a good, perhaps surprising, side effect, these structures do not only render the index updatable, but they can improve the search performance even for static data at the expense of a slight increase of storage space. The reason is that they try to ensure that records are located closely to the predicted position. Our analysis shows that there is an elegant trade-off between space and search performance under this strategy.

- We propose techniques to *split* models and adapt the height of the tree as a result of updates. This way the index gains robustness to handle workloads with shifting data distributions.

- We present the results of an extensive experimental analysis with real-life datasets and varying read-write workloads.

On read-only workloads, ALEX beats the Learned Index by up to 2.7× on performance with up to 3000× smaller index size. ALEX achieves up to 3.5× higher performance than the B+Tree while having up to 5 orders of magnitude smaller index size. On certain read-write workloads, ALEX achieves up to 3.3× higher throughput than the B+Tree while having up to 2000× smaller index size. On other read-write workloads, ALEX is competitive with, or slightly worse than, a B+Tree.

The remainder of this paper is organized as follows. Section 2 introduces background on learned index structures. Section 3 presents our new index structure, ALEX. Section 4 presents an analysis of ALEX's search performance when increasing the storage space. Section 5 presents experimental results. Section 6 reviews related work. Section 7 discusses the current limitations of our techniques, and proposes directions for future work, and we conclude in Section 8.

2. BACKGROUND

2.1 Traditional B+Tree Indexes

B+Tree is a classic range index structure, which is a crucial component of database systems. It is a height-balanced tree which stores either the data (primary index) or pointers to the data (secondary index) at the leaf level, in a sorted order to facilitate range-queries.

A B+Tree index lookup operation can be broken down into two logical steps: (1) traverse to leaf, and (2) search within the leaf. Starting at the root, traverse to leaf performs several comparisons with the keys stored in each node, and branches via stored pointers to the next level. When the size of the index is large, the tree is deep, and the number of comparisons and branching can be large, leading to many cache misses. Once the correct leaf page is identified by traverse to leaf, typically a binary search is performed to find the position of the key within the node. Binary search involves further comparisons and branching, and depending on the size of the node and the particular B+Tree implementation, this step might incur additional cache misses.

The B+Tree is a dynamic data structure that supports inserts, updates, and deletes. It has been extensively researched, developed, tuned, and widely deployed over the last 50 years. It is truly a "ubiquitous" data structure which has proven to be indispensable due to its robustness to data sizes and distributions and applicability in many different scenarios, including in-memory and on-disk.

As observed in [17], the general applicability of B+Tree comes at a cost. In some cases the knowledge about input data and its distribution is helpful in improving the performance. As an extreme example, if the keys are consecutive integers, we can simply store the data in an array and perform lookup in O(1) time. A B+Tree does not exploit such knowledge. This is where "learning" from the input data has an edge.

2.2 The Case for Learned Indexes

Kraska et al. [17] observed that B+Tree indexes can be thought of as models. Given a key, they predict the location of the key within a sorted array (logically) at the leaf level. If indexes are models, they can be learned using traditional ML techniques by learning the cumulative distribution function (CDF) of the input data. The resulting *learned index* is optimized for the specific data distribution.

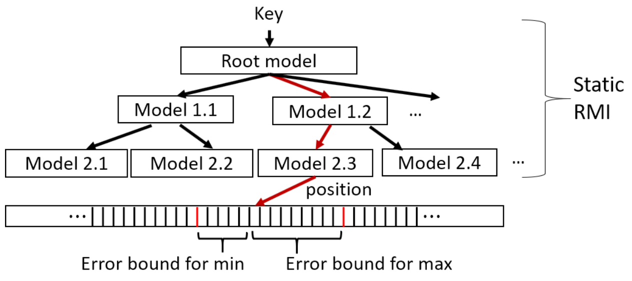

Figure 1: Learned Index by Kraska et al.

Another insight from Kraska et al. is that a single ML model learned over the entire data is not accurate enough because of the complexity of the CDF. To overcome this, they introduce the *recursive model index* (*RMI*) [17]. RMI comprises a hierarchy of models, with a *static* depth of two or three, where a higher level model picks the model at the next level, and so on, with the leaf level model making the final prediction for the position of the key in the data structure, as shown in Figure 1. Logically, the RMI replaces the internal B+Tree nodes with models. The net effect is that comparisons and branching in internal B+Tree nodes during traverse to leaf are replaced by multiplications and additions (model inference) in a learned index, which are much faster to execute on modern processors.

In [17], the keys are stored in an in-memory sorted array. Given a key, the leaf-level model predicts the position (array index) of the key. Since the model is not perfect, it could make a wrong prediction. Here the insight is that, if the leaf model is accurate enough, a local search surrounding the predicted location is still faster than doing a binary search on the entire array. To enable this local search, [17] proposes keeping *min* and *max* error bounds for each model in RMI. A bounded binary search within these bounds proves to be faster.

Last, each model in RMI can be a different type of model. Both linear regression and neural network based models are considered in [17]. There is a trade-off between model accuracy and model complexity. The root of the RMI is tuned to be either a neural network or a linear regression, depending on which provides better performance, while the simplicity and the speed of computation for linear regression model proves to be beneficial at the non-root levels. A linear regression model can be represented as $y = \lfloor a * x + b \rfloor$, where x is the key and y is the predicted position. A linear regression model needs to store just two parameters a and b, so the storage overhead is low. The inference with a single linear regression model requires only one multiplication, one addition and one rounding, which as noted above are fast to execute on modern processors.

Unlike B+Tree, which could have many internal levels, RMI uses two or three levels. Further, the storage space required for models (two or four 8-byte double values per internal non-root model or leaf model) is also much smaller than the storage space needed for internal nodes in B+Tree (which store both keys and pointers). A learned index can be an order of magnitude smaller in main memory storage used for the index (vs. internal B+Tree nodes), while out-performing a B+Tree in lookup performance by a factor of up to three [17].

2.3 Current Limitations of Learned Indexes

The main drawback of the Learned Index [17] is its static nature. The resulting data structure does not support any modifications including inserts, updates, or deletes. Let us demonstrate a naïve insertion strategy for such an index. Given a key k to insert, we first use the model to find the insertion position for k. Then we create a new array whose length is one plus the length of the old array. Next, we copy the data from the old array to the new array, where the elements on the right of the insertion position for k are shifted to the right by one position. We insert k at the insertion position of the new array. Finally, we update the models to reflect the change in the data distribution.

Such a strategy has a linear time complexity with respect to the data size. Further, as data are inserted, the RMI models get less accurate over time, which requires model retraining, further adding to the cost of inserts. Clearly, such a naïve insert strategy is unacceptable in practice. Kraska et al. suggest building delta-indexes to handle inserts [17]. In this paper, we describe an alternative data structure to make insertions in a learned index more efficient.

3. ALEX: ADAPTIVE LEARNED INDEX

The ALEX design takes advantage of two key insights. First, we propose a careful space-time trade-off that not only leads to an updatable data structure, but is also faster for lookups. To explore this trade-off, ALEX supports two leaf node layouts namely, *Gapped Array (GA)* and *Packed Memory Array (PMA)*, which we present in Section 3.3. Second, the Learned Index supports static RMI (SRMI) only, where the number of levels and the number of models in each level is fixed at initialization. This static RMI turns out to be problematic with dynamic inserts, in certain cases. To achieve robust search and insert performance in ALEX, we propose an *adaptive RMI (ARMI)*, which initializes and dynamically adapts the RMI structure based on the workload. We present adaptive RMI in Section 3.4. The overall design of ALEX is shown in Figure 2. We show only adaptive RMI in the figure, but ALEX can be configured to run with static RMI as well.

3.1 Design Goals

While supporting inserts in ALEX, we aim to achieve the following time and space goals. (1) Insert time should be competitive with B+Tree, (2) lookup time should be much faster than B+Tree and competitive to the existing, static learned index, (3) index storage space should be competitive to the existing, static learned index and much smaller than B+Tree, and (4) data storage space (leaf level) should be comparable to B+Tree. In general, data storage space will overshadow index storage space, but the space benefit from using smaller index storage space is still very important in practice because it allows more indexes to fit into the same memory budget. The rest of this section describes how our ALEX design achieves these goals.

3.2 ALEX Design Overview

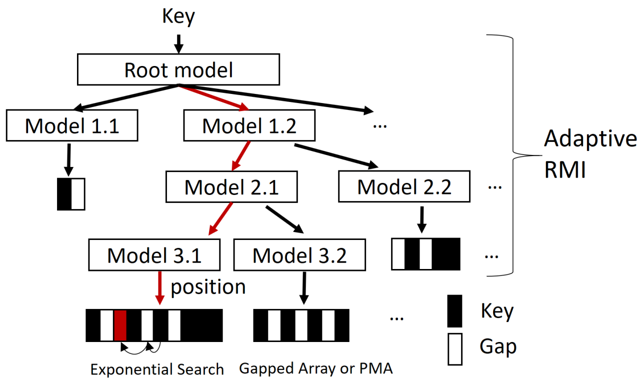

Figure 2: ALEX Design

As noted earlier, ALEX is an in-memory, updatable learned index. The variants of ALEX with static RMI, have a two-layer RMI with a root model and some pre-determined number of leaf models, where each model, including the root model, is a linear regression model. This is similar to the design of the learned index presented in [17]. However, this is where the similarities between the two end.

The first (minor) difference is that ALEX uses exponential search to find keys at the leaf level to compensate for mispredictions of the RMI, as shown in Figure 2. In contrast, [17] proposes to use binary search within the error bounds provided by the models. We experimentally verified that exponential search without bounds is faster than binary search with bounds. This is because if the models are good, their prediction is close enough to the final position. We evaluate the two search methods and show the impact of model prediction error on search performance in Section 5.3.2. More importantly, exponential search obviates the need to store error bounds in the models of the RMI, thereby giving extra room to navigate the space-time trade-off.

The second, more fundamental difference lies in the data structure used to store the data at the leaf level. The Learned Index uses a single, sorted array. As described in the naïve algorithm of Section 2.3, this structure can make it prohibitively expensive to insert new data into the index. Adopting ideas from dynamic index structures such as B+Tree, ALEX uses a *node per leaf.* The goal is to allow the individual nodes to expand more flexibly as new data is inserted. It also limits the number of shifts required during an insert.

In a typical B+Tree, every leaf node has the same layout: It stores an array of keys and values. Furthermore, the node has some "free space" at the end of the array to absorb new inserts. ALEX uses a similar design but more carefully chooses how to use the free space in each node. The key insight is that by introducing gaps that are strategically placed between elements of the array, we can achieve faster insert and lookup time. As shown in Figure 2, ALEX uses two alternative array layouts in each node, Gapped Array vs. Packed Memory Array, each with their own strategy of placing and updating the gaps. Further, ALEX provides the extra flexibility to select which layout to use based on the target workload and the target space-time trade-off. We detail these node layouts in the next sub-section (Section 3.3).

The next fundamental difference is that ALEX dynamically adjusts the shape and height of the RMI depending on the update workload. This adaptive RMI approach is described in detail in Section 3.4.

There is a fourth, subtle yet important difference between ALEX and the learned indexes of [17]. As ALEX inserts records dynamically, ALEX inserts keys at the position at which the models predict that the key should be. We call this effect *model-based insertion* and it comes naturally with ALEX. In contrast, the approach described in [17] bulkloads indexes by taking an array of records as input and producing an RMI on top of that array without changing the position of records in the array. It turns out that *model-based insertion* has much better search performance because it reduces the misprediction error of the models.[1]

In this paper, we focus primarily on the challenges of supporting inserts. We argue that if inserts are supported, then deletes and updates are also straightforward to support. First, deletes are strictly easier to support than inserts; in the same way that ALEX nodes expand upon inserts, ALEX nodes can also contract upon deletes, and the models are retrained in the same way in both cases. However, inserts often require shifting existing keys in a node to create an open slot in which to insert, whereas deletes do not have this challenge, which makes deletion a strictly simpler operation. Second, updates that modify the key can naturally be implemented by combining an insert and a delete into one operation, and updates that only modify the payload value can be implemented by finding the key and writing the new value into the payload.

We next describe the different node layouts followed by a description of our adaptive RMI.

3.3 Flexible Node Layout

This section presents two different node layouts. ALEX can be configured to run with either node layout, depending on the workload, making it flexible.

3.3.1 Gapped Array (GA)

This node layout effectively makes use of the extra storage space by using model-based inserts to "naturally" distribute extra space between the elements of the array. One key detail is that we fill the gaps with adjacent keys, specifically the closest key to the right of the gap, which helps in maintaining good exponential search performance within each leaf node. Gapped arrays not only make ALEX updatable, but they also make search faster by keeping the keys close to the intended position during inserts and, thus, close to the predicted position for a lookup.

To insert a new element into a sorted gapped array, we first find the insertion position. To find the insertion position, we use the RMI to predict the insertion position. If this is the correct position, i.e., inserting the key at this position maintains the sorted order, and is a gap (free space), then we insert the element into the gap and are done. Note that this is the best case for model-based inserts, as we can place the key exactly where the model predicts and thus a later model-based lookup will result in a direct hit, thus we can do a lookup in $O(1)$. If the predicted position is not correct, we do exponential search to find the actual insertion position. Again, if the insertion position is a gap, then we insert the element there and are done. If the insertion position is not a gap, we make a gap at the insertion position by shifting the elements by one position in the direction of the closest gap. We then insert the element into the newly created gap.

[1]Others have independently made a similar observation.

Algorithm 1 *Gapped Array* Insertion

```
1: struct Node { keys[]; num_keys; d; model; }
2: procedure INSERT(key)
3:     if num_keys / keys.size >= d then
4:         Expand()            /* See Alg. 3 */
5:     end if
6:     predicted_pos = model.predict(key)
7:     /* check for sorted order */
8:     insert_pos = CorrectInsertPosition(predicted_pos)
9:     if keys[insert_pos] is occupied then
10:        MakeGap(insert_pos) /* described in text */
11:    end if
12:    keys[insert_pos] = key
13:    num_keys++
14: end procedure
```

The gapped array achieves $O(\log n)$ insertion time with high probability [5].

In practice, insertion performance on the gapped array degrades over time because the number of gaps decreases as we insert more elements into the gapped array, which increases the expected number of elements that need to be shifted for each insert. Therefore, we introduce an upper limit on the density of the gapped array, defined as the fraction of the positions that are filled by elements. If an additional insertion results in crossing the upper density limit $d \in (0, 1]$, then we expand the gapped array by a factor of $1/d$, retrain the linear regression model corresponding to this leaf node in the RMI, and then do model-based inserts of all the elements in this node using the retrained RMI. The models at the upper levels of the RMI are not retrained in this event. Retraining efficiency is one reason why we propose to use linear models for ALEX.

Algorithm 1 summarizes the procedure for inserting into a gapped array node of ALEX. If an additional element will push the gapped array over its density bound d, then the gapped array expands. Pseudocode for expansion can be found in Algorithm 3, which contains operations that are common to both gapped array and the Packed Memory Array. If an additional element does not violate the density bound, then insertion proceeds in the manner described previously.

After the expansion and redistribution, the density of the gapped array will be d^2. The length of the array is $\frac{1}{d^2}$ times the actual number of keys. We define $c = \frac{1}{d^2}$ as the *expansion factor*. When we build the index for the first time, we initialize each leaf node of n keys by allocating an array of length $c * n$ such that the density is also d^2. Given a target budget for storage, we can set c in ALEX accordingly to meet that budget. The upper density limit d is then set to $\sqrt{\frac{1}{c}}$. This becomes useful for our design goal of having data storage space comparable to B+Tree. Section 4 presents a more detailed theoretical analysis of the trade-off between the space and the lookup performance for ALEX.

A B+Tree needs to split nodes to limit the cost for a binary search within a B+Tree node. In contrast, the search effort in an ALEX node is limited by the accuracy of the model, rather than the size of the node. Therefore, ALEX uses node expansions. Expansions provide more fine-grained control of the expansion factor (and space-time trade-off) and are less expensive than splits. Section 3.4.2 describes how to control the accuracy of the models. Indeed, splitting the models is an effective mechanism for that purpose.

One drawback of the gapped array is its worst case performance. Specifically, worst case performance happens when

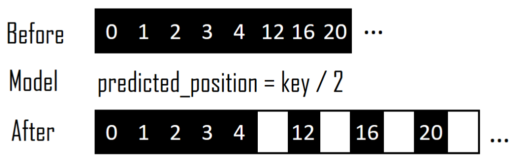

Figure 3: Even though the array expands by 50%, model-based insertion results in a fully-packed region on the left.

model-based insertion results in a long contiguous region without any gaps, which we call a *fully-packed region.* Figure 3 shows an example of a fully-packed region. Inserting into a fully-packed region requires shifting up to half of the elements within it to create a gap, which in the worst case takes $O(n)$ time. In our experience, fully-packed regions can dramatically increase the gapped array's insertion time. We next describe an alterative structure with better worst case insertion.

3.3.2 Packed Memory Array (PMA)

As an alternative to gapped arrays, we have experimented with Packed Memory Arrays (PMA) [6] as a data structure for the ALEX data nodes. PMAs have been shown to adapt and grow better to inserts in theory because a PMA is designed to uniformly space its gaps between elements and to maintain this property as new elements are inserted. The PMA achieves this goal by rebalancing local portions of the array when the gaps are no longer uniformly spaced. Under random inserts from a static distribution, the PMA can insert elements in $O(\log n)$ time, which is the same as the gapped array. However, when inserts do not come from a static distribution, the PMA can guarantee worst-case insertion in $O(\log^2 n)$ time, which is better than the worst case of the gapped array, which is $O(n)$ time.

We now describe the PMA more concretely; more details can be found in [6]. The PMA is an array whose size is always a power of 2. The PMA divides itself into equally-spaced segments, and the number of segments is also a power of 2. The PMA builds an implicit binary tree on top of the array, where each segment is a leaf node, each inner node represents the region of the array covered by its two children, and the root node represents the entire array. The PMA places density bounds on each node of this implicit binary tree, where the density bound determines the maximum ratio of elements to positions in the region of the array represented by the node. The nodes nearer the leaves will have higher density bounds, and the nodes nearer the root will have lower density bounds. The density bounds guarantee that no region of the array will become too packed. If an insertion into a segment will violate the segment's density bounds, then we can find some local region of the array and uniformly redistribute all elements within this region, such that after the redistribution, none of the density bounds are violated. As the array becomes more full, ultimately no local redistribution can avoid violating density bounds. At this point, the PMA expands by doubling in size and inserting all elements uniformly spaced in the expanded array.

Inserting into a PMA node of ALEX is summarized in Algorithm 2. Given a key, the node tries to insert the key into the PMA in the position that the model predicts, which possibly involves PMA logic such as rebalancing according to the density bounds. More details of the `InsertPMA` function can be found in [6]. If the PMA insertion fails, i.e. if the key cannot be inserted without violating a density bound, then the node expands. As we explain shortly, a PMA node of ALEX does not expand according to regular PMA logic. Pseudocode for expansion can be found in Algorithm 3.

Algorithm 2 *PMA* Insertion

```
 1: struct Node {keys[]; num_keys; pma_density_bounds;
    model}
 2: procedure INSERT(key)
 3:     predicted_pos = model.predict(key)
 4:     /* check for sorted order */
 5:     insert_pos = CorrectInsertPosition(predicted_pos)
 6:     insert_status = InsertPMA(key, insert_pos)
 7:     if insert_status == failure then
 8:         /* density bounds violated */
 9:         Expand()              /* See Alg. 3 */
10:         pma.insert(key, insert_pos) /*will succeed*/
11:     end if
12:     num_keys++
13: end procedure
```

The PMA optimizes for inserts by avoiding fully-packed regions. However, since the PMA preemptively spreads the keys in these regions over more space, this behavior can impact search performance because the keys are moved further away from their predicted location. To maintain both good insert and lookup performance, ALEX uses model-based inserts after every PMA expansion, i.e., instead of uniformly inserting elements, ALEX inserts the elements according to the RMI predictions using the same algorithm as for gapped arrays. Over time, as we insert elements into the PMA and rebalances are performed, the elements of the PMA will begin to spread out, with uniformly spaced gaps in between, as per the default PMA algorithm. Therefore this node layout achieves a middle ground between the performances of the gapped array and the regular PMA.

3.3.3 Node Layout Summary

We have described two node layouts, the gapped array and PMA, and how to insert elements into them. In Algorithm 3, we describe how to perform lookups and expansions, which are performed equivalently for both node layouts. Lookups are performed by using the model to predict the position of an element, then using exponential search from the predicted position to find the actual position. Expansions are perfomed by creating an expanded data array whose size is different for gapped array and PMA; retraining the node's linear model on the existing keys; rescaling the model to predict positions in the expanded array; and using the model to perform model-based inserts into the expanded array. One detail is that the node performs model-based inserts of elements in sorted order. If the model tries to insert multiple elements into the same position, every element after the first will instead be inserted into the first gap to the right.

In the case of a "cold start" in which a node is initialized with no keys or very few keys, we do not build a model and instead perform lookups by doing binary search over the node, which is equivalent to what B+Tree does. Once the node contains a sufficient number of keys, we build and maintain a model in order to accelerate lookups.

3.4 Adaptive RMI

The second dimension in our design space is exploring the effect of static vs. adaptive RMI to accommodate for

Algorithm 3 Common Node Operations for GA & PMA

```
struct Node { keys[]; num_keys; model; }
procedure Lookup(key)
    predicted_pos = model.predict(key)
    /*Search target key starting from predicted_pos*/
    actual_pos = keys.exp_search(key, predicted_pos)
    return actual_pos
end procedure
procedure Expand
    if GappedArray == true then
        expanded_size = keys.size * 1/d
    else if PackedMemoryArray == true then
        expanded_size = keys.size * 2
    end if
    /* allocate a new expanded array */
    expanded_keys = array(size=expanded_size)
    model = /* train linear model on keys */
    expansion_factor = expanded_size / num_keys
    model *= expansion_factor                /*scale model*/
    for key : keys do
        ModelBasedInsert(key)
    end for
    keys = expanded_keys
end procedure
procedure ModelBasedInsert(key)
    insert_pos = model.predict(key)
    if keys[insert_pos] is occupied then
        insert_pos = first gap to right of predicted_pos
    end if
    keys[insert_pos] = key
end procedure
```

dynamic workloads. Static RMI suffers from two problems: (1) wasted models and (2) fully-packed regions. To deal with these issues, ALEX adapts the structure of the RMI during initialization of the index, as well as dynamically adjusts the RMI during updates.

Wasted models are a result of a skewed distribution. Consider a highly skewed distribution where majority of the keys are concentrated on one end. Since ALEX uses a linear root model which simply partitions the key space, most of the leaf models will have a small number of keys, hence such models waste index space.

Avoiding fully-packed regions is necessary to maintain robust search and lookup performance with dynamic inserts. The two node layouts described above try to ensure this via expansions of the array, to create more gaps, and retraining of the models, to maintain prediction accuracy. As we show later in Section 5.2.5, if a model is susceptible to producing fully-packed regions, expansions alone do not help since the size of the fully-packed regions grows proportionally, leading to more shifts on inserts. Adaptive RMI limits the size of a leaf data node to prevent such cases.

Next we describe (1) ALEX initialization with adaptive RMI, and (2) how ALEX dynamically changes RMI, in the presence of inserts by doing node splits.

3.4.1 Adaptive RMI Initialization

In this section, we talk about how we bootstrap an ALEX with adaptive RMI. At initialization, to limit the impact of fully-packed regions, we use a pre-determined maximum bound for the number of keys in a leaf data node, and allow the ALEX to determine the RMI depth and number of leaf models adaptively. This maximum bound can be tuned or learned for each dataset. We also want to limit the resulting depth of the RMI; we experimentally verified that traversing down a deep RMI to a data leaf node can be expensive

Algorithm 4 Adaptive RMI Initialization

```
constant: max_keys
procedure Initialize(node)
    node.model = /*train linear model*/
    partitions = node.get_partitions()
    it = /* iterator over partitions */
    while it.has_next() do
        partition = it.next()
        if partition.size > max_keys then
            child = InnerNode(keys = partition)
            Initialize(child)
        else
            begin = it.current()
            accumulated_size = partition.size
            while accumulated_size < max_keys do
                partition = it.next()
                accumulated_size += partition.size
            end while
            end = it.prev()
            child =
            LeafNode(keys=partitions[begin:end])
        end if
    end while
end procedure
```

because going down each layer of the RMI requires following a pointer and likely entails a cache miss. Therefore, adaptive RMI initialization should achieve the desired upper bound on the number of keys for each leaf data node, while also limiting the depth of the RMI.

Algorithm 4 shows the procedure for adaptive RMI initialization. `Initialize` is first called on the RMI's root node, and is then called recursively for all child nodes. To initialize a node, we first train the node's linear model using its assigned keys. We can then use the model to divide the keys into some number of partitions. Specifically, we give the root node a number of partitions such that in expectation each partition will have exactly the maximum bound number of keys. We give each non-root node a fixed number of partitions that is tuned or learned for each dataset. We then iterate through the partitions in sorted order. If a partition has more than the maximum bound number of keys, then this partition is oversized (hence susceptible to fully-packed regions), so we create a new inner node and recursively call `Initialize` on the new node. Otherwise, the partition is under the maximum bound number of keys, so we could just make this partition a leaf node. However, the partition might contain a very small number of keys, which would result in a wasted leaf. Therefore, we instead start merging the partition with the subsequent partitions, until the accumlated number of keys from the merged partitions exceeds the maximum number of keys per node, at which point we drop the latest partition (to push the accumulated number of keys back below the maximum number) and use the merged partitions to create a new leaf node. Because we often merge multiple adjacent partitions before hitting the maximum bound, and each of the merged partitions point to the same child leaf node, the number of children is often significantly lower than the number of partitions.

3.4.2 Node Splitting On Inserts

Adaptive RMI initialization fixes the structure of the RMI after initialization, and does not change the structure during dynamic inserts. If the distribution of keys does not change, an adaptively initialized RMI structure performs well, because inserts will in expectation hit leaf nodes at a uniform rate. However, if the distribution of keys does change, then as inserts occur, some leaves will become increasingly susceptible to fully-packed regions. With dynamic inserts, B+Tree adapts itself by splitting full nodes. Our next optimization which we call *node splitting on inserts* applies the same idea to ALEX, and is a first step towards addressing the challenge of dataset distribution shift. Note that as opposed to a B+Tree, we do not rebalance ALEX when splitting nodes.

The main idea is that if an insert will push a leaf node's data structure over its maximum bound number of keys, then we split the leaf data node. The corresponding leaf level model in RMI now becomes an inner level model, and a number of children leaf level models are created. The data from the original leaf node is then distributed to the newly created children leaf nodes according to the original node's model. Each of the children leaf nodes trains its own model on its portion of the data. The number of children leaf nodes to create on split is a parameter that can be tuned or learned for each dataset, similar to the number of partitions for adaptive RMI initialization.

Node splitting on inserts also allows ALEX to handle "cold starts" in which the data is initially empty and new keys are added incrementally. In this case, the adaptive RMI will begin as only a single node and will grow deeper through splitting as more keys are inserted.

4. ANALYSIS OF MODEL-BASED INSERTS

ALEX tries to place a key in the location predicted by a linear model. We analyze the trade-off between the expansion factor c and the search performance.

Given a leaf node, assume the keys in that leaf node are $x_1 < x_2 < \cdots < x_n$, and the linear model before rounding is $y = ax + b$ when $c = 1$, i.e., when no extra space is allocated. Define $\delta_i = x_{i+1} - x_i, \Delta_i = x_{i+2} - x_i$. We first present a sufficient condition under which all the keys in that leaf node are placed in the predicted location, i.e., search for all the n keys are direct hits, requiring no comparisons.

THEOREM 1. *When $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i}$, every key in the leaf node is placed in the predicted location exactly.*

PROOF. Consider two keys in the leaf node x_i and $x_j, i \neq j$. The predicted locations before rounding are y_i and y_j, respectively. When $|y_i - y_j| \geq 1$, we know that the rounded locations $\lfloor y_i \rfloor$ and $\lfloor y_j \rfloor$ cannot be equal. Under the linear model $y = c(ax + b)$, we can write the condition as:

$$|y_i - y_j| = |ca(x_i - x_j)| \geq 1 \qquad (1)$$

If this condition is true for all the pairs $(i, j), i \neq j$, then all the keys will have a unique predicted location. For the condition Eq. (1) to be true for all $i \neq j$, it suffices to have:

$$\min_{i=1}^{n-1} ca(x_{i+1} - x_i) \geq 1 \qquad (2)$$

which is equivalent to $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i}$. $\square$

This result suggests that once the expansion factor is larger than $\frac{1}{a \min_{i=1}^{n-1} \delta_i}$, the search performance for the keys in this leaf node will not further improve.

In other words, we now understand that $c = 1$ corresponds to the optimal space, and $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i}$ corresponds to the optimal search time (without considering the effect of cache misses). However, since the minimal distance between keys $\min \delta_i$ can be tiny, the setting of c corresponding to the optimal search time can require extremely large gaps which is impractical. So we extend the analysis to bound the number of keys with direct hits when $c < \frac{1}{a \min_{i=1}^{n-1} \delta_i}$.

THEOREM 2. *The number of keys placed in the predicted location is no larger than $2 + \left|\{1 \leq i \leq n-2 | \Delta_i > \frac{1}{ca}\}\right|$, where $\left|\{1 \leq i \leq n-2 | \Delta_i > \frac{1}{ca}\}\right|$ is the number of Δ_i's larger than $\frac{1}{ca}$.*

The proof can be found in Appendix A. This result presents an upper bound on the number of direct hits from the model, which is positively correlated with the expansion factor. The smaller the c, the fewer the direct hits, requiring more comparisons. We can use this result to determine a lower bound on c such that the upper bound of direct hits is above a threshold. As a special case, this upper bound also applies to the previously proposed RMI in [17], where the expansion factor $c = 1$. It explains why the gapped array or PMA has the potential to largely boost the search time.

Following a similar idea, we can calculate a lower bound for the number of direct hits.

THEOREM 3. *The number of keys placed in the predicted location is no smaller than $l + 1$, where l is the largest integer such that $\forall 1 \leq i \leq l, \delta_i \geq \frac{1}{ca}$, i.e., the number of consecutive δ_i's from the beginning equal or larger than $\frac{1}{ca}$.*

The proof is not hard based on the ideas from the previous two proofs. The result is an improvement of Theorem 1, and can be used to reduce c when perfect lookup is not required. This bound is difficult to improve using the same analysis method because once a collision happens, it can cause a chain of movements. It is possible to force all the keys after the collision point to move off the predicted location. If we ignore that effect, we can have an approximate lower bound $1 + \left|\{1 \leq i \leq n-1 | \delta_i \geq \frac{1}{ca}\}\right|$. When Theorem 1's condition is true, the exact and the approximate lower bound, and the exact upper bound all become equal.

5. EVALUATION

In this section, we first describe the experimental setup and the datasets used for our evaluation. We then present the results of an in-depth experimental study that compares ALEX with the Learned Index from [17] and B+Tree, using a variety of datasets and workloads. We conclude this section with a drilldown into the flexible node layout and adaptive RMI, described in Section 3, to give intuition for why certain variants of ALEX work well in certain situations. Overall, this evaluation demonstrates that:

- On read-only workloads, ALEX achieves up to 3.5× higher throughput than the B+Tree while having up to 5 orders of magnitude smaller index size. ALEX also achieves up to 2.7× higher throughput and up to 3000× smaller index size than the Learned Index.
- On read-write workloads, ALEX achieves up to 3.3× higher throughput than the B+Tree while having up to 2000× smaller index size.
- ALEX maintains an advantage over the B+Tree when scaling to larger datasets and remains competitive with the B+Tree under moderate dataset distribution shift.

- Flexible node layout and adaptive RMI allows ALEX to adapt to different datasets and workloads.

5.1 Experimental Setup

We implement ALEX in C++. We perform our evaluation via single-threaded experiments on an Ubuntu Linux machine with Intel Core i9 3.6GHz CPU and 64GB RAM. We compare ALEX against two baselines. The first baseline is a standard B+Tree, as implemented in the STX B+Tree [3]. The second baseline is our own best-effort reimplementation of the Learned Index from [17], using a two-level RMI with linear models at each node and binary search for lookups.[2] Since we focus on supporting range queries, we do not compare to hash tables and dynamic hashing techniques.

ALEX can be configured to use either gapped array (GA) or the Packed Memory Array (PMA) for its node layout, and either static RMI (SRMI) or adaptive RMI (ARMI) for model hierarchy. Therefore, ALEX has four different variants: (1) ALEX-GA-SRMI, (2) ALEX-GA-ARMI, (3) ALEX-PMA-SRMI, and (4) ALEX-PMA-ARMI. We show that each variant of ALEX performs best in different situations. For each benchmark, we note the particular variant of ALEX used. The density bounds for gapped array and PMA are set so that ALEX data storage space is comparable to B+Tree data storage space. The number of models for static RMI and the maximum bound keys per leaf for adaptive RMI are tuned using grid search to achieve the best throughput. Unless otherwise stated, adaptive RMI does not do node splitting on inserts.

For each benchmark, we use grid search to tune the page size used for B+Tree to achieve the best throughput. The STX B+Tree does not have any obvious tunable parameters other than page size. We also make a best effort to tune the number of models for the Learned Index while not exceeding the model sizes reported in [17], using grid search.

To measure index size for ALEX and the Learned Index, we sum the sizes of all models used in the index, as well as pointers and metadata. For ALEX, each model consists of two double-precision floating point numbers which represent the slope and intercept of a linear regression model. The models used in the Learned Index keep two additional integers that represent the error bounds used in binary search. The index size of B+Tree is the sum of the sizes of all inner nodes. To measure data size for ALEX, we sum the allocated sizes of the arrays containing the keys and payloads, including gaps, as well as a bitmap, which we explain in Section 5.2.3. The data size of B+Tree is the sum of the sizes of all leaf nodes.

5.1.1 Datasets

We run all experiments using 8-byte keys from some dataset and randomly generated fixed-size payloads. We evaluate ALEX on 4 datasets, whose characteristics are shown in Table 1. The *longitudes* dataset consists of the longitudes of locations around the world from Open Street Maps [2]. The *longlat* dataset consists of compound keys that combine longitudes and latitudes from Open Street Maps by applying the transformation $k = 180 \cdot \text{longitude} + \text{latitude}$ to every pair of longitude and latitude. The resulting distribution of compound keys k is highly non-linear. The *lognormal* dataset has values generated according to a lognormal distribution, which we use to demonstrate performance on highly skewed distributions. The *YCSB* dataset has values representing user IDs generated according to the YCSB Benchmark [8], which follows a uniform distribution, and uses an 80-byte payload. These datasets do not contain duplicate values. Unless otherwise stated, these datasets are randomly shuffled to simulate a uniform dataset distribution over time. A more detailed description of dataset creation and analysis of dataset characteristics can be found in Appendix C.

[2]In private communication with the authors of [17], we learned that the added complexity of using a neural net for the root model usually is not justified by the resulting minor performance gains.

Table 1: Dataset Characteristics

	longitudes	**longlat**	**lognormal**	**YCSB**
Num keys	1B	200M	190M	200M
Key type	double	double	64-bit int	64-bit int
Payload size	8B	8B	8B	80B
Total size	16GB	3.2GB	3.04GB	17.6GB
Read-only init size	200M	200M	190M	200M
Read-write init size	50M	50M	50M	50M

5.1.2 Performance Metric & Workloads

Our primary metric for evaluating ALEX is throughput. We evaluate throughput for four workloads, which are in the style of the YCSB workloads [8]: (1) a read-only workload, (2) a read-heavy workload with 95% reads and 5% inserts, (3) a write-heavy workload with 50% reads and 50% inserts, and (4) a range scan workload with 95% reads and 5% inserts. For the first three workloads, reads consist of a lookup of a single key. For the range scan workload, a read consists of a lookup of a key followed by a scan of the subsequent keys. The number of keys to scan is selected randomly from a uniform distribution with a maximum scan length of 100. For all workloads, keys to look up are selected randomly from the set of existing keys in the index according to a Zipfian distribution, so that a lookup will always find a matching key. These four workloads roughly correspond to Workloads C, B, A, and E from the YCSB benchmark, respectively. For a given dataset, we initialize an index with a fixed number of keys, as noted in Table 1. We then run the specified workload for 60 seconds. We report the total number of operations completed in that time, where operations are either inserts or reads. For the read-write workloads, we interleave the operations to simulate real-time usage. Specifically, for the read-heavy workload and range scan workload, we perform 19 reads/scans, then 1 insert, then repeat the cycle; for the write-heavy workload, we perform 1 read, then 1 insert, then repeat the cycle.

5.2 Overall Results

5.2.1 Read-only Workloads

For read-only workloads, ALEX-GA-SRMI is the best variant of ALEX to use; if no inserts occur, then there is less need to avoid fully-packed regions. Therefore, using PMA or adaptive RMI is unnecessary and only adds additional overhead.

Figure 4a shows that ALEX achieves up to 3.5× higher throughput than the B+Tree and up to 2.7× higher throughput than the Learned Index. On the longlat dataset, ALEX achieves only comparable throughput to B+Tree because

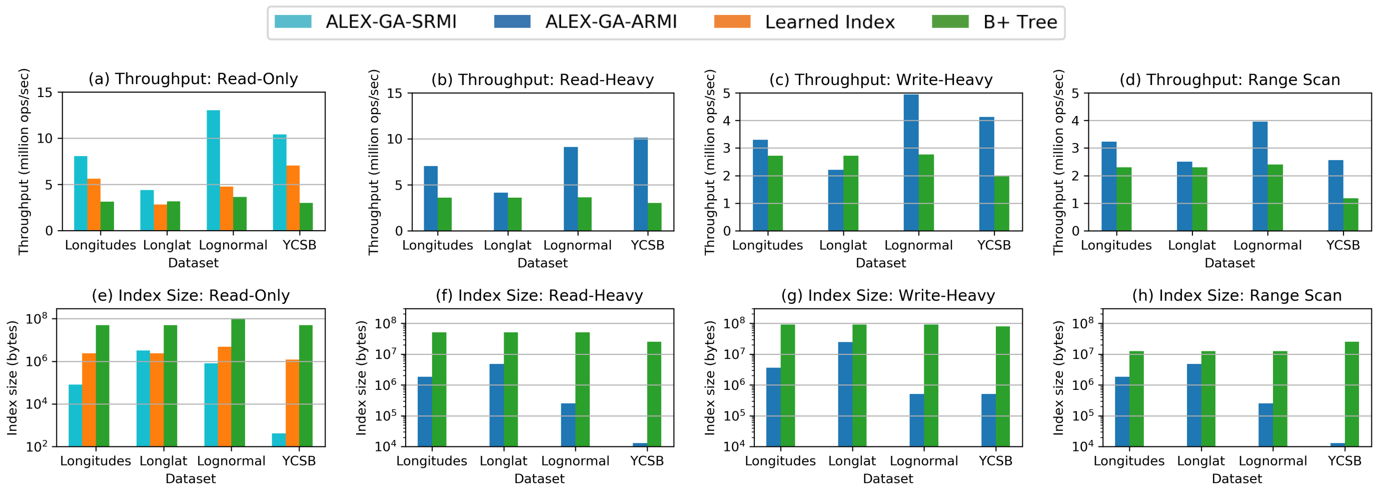

Figure 4: ALEX vs. Baselines: Throughput & Index Size. Throughput includes model retraining time.

the key distribution is more non-linear and therefore more difficult to model.

Figure 4e shows that on read-only workloads, ALEX has up to 5 orders of magnitude smaller index size than the B+Tree and up to 3000× smaller index size than the Learned Index. The index size of ALEX is dependent on how well ALEX can model the dataset distribution. On the lognormal and YCSB datasets, which are more locally linear, ALEX does not require many models to accurately model the distribution, so ALEX achieves small index size relative to the other indexes. This is especially true for the YCSB dataset, whose key distribution is uniform. ALEX has smaller index size than the Learned Index because ALEX uses model-based inserts to obtain better predictive accuracy for each model, which we show in Section 5.3, and therefore achieves high throughput while using relatively fewer models. For example, on the YCSB dataset, Learned Index achieves its best throughput with 50000 models, whereas ALEX achieves its best throughput with 25 models. This difference, combined with the fact that Learned Index uses additional space per model to store error bounds, accounts for ALEX having 3000× smaller index size. However, on datasets that are more challenging to model such as longlat, ALEX is unable to achieve high throughput with few models. Therefore, ALEX tries to use more models, but even then, does not achieve high throughput. This demonstrates a recurring trend: when ALEX achieves higher throughput, it does so with smaller index size.

5.2.2 *Read-Write Workloads*

For the read-write workloads, we initialize with a smaller number of keys so that we capture the throughput as the index size grows. The Learned Index has insert time orders of magnitude slower than ALEX and B+Tree, so we do not include it in these benchmarks.

For read-write workloads, we use the ALEX-GA-ARMI variant; adaptive RMI helps ALEX avoid large fully-packed regions on inserts, and the gapped array helps maintain short lookup times. Figure 4b and Figure 4c show that ALEX achieves up to 3.3× higher throughput than B+Tree on datasets such as lognormal and YCSB that are simpler to model, but also achieves up to 20% lower throughput on datasets such as longlat that are more difficult to model.

Figure 4f and Figure 4g show that ALEX continues to have up to 2000× smaller index size than the B+Tree. Similar to the read-only workloads, higher throughput is generally achieved in conjunction with smaller index size.

5.2.3 *Range Scan Workload*

Figure 4d and Figure 4h show that ALEX maintains its advantage over B+Tree when scanning ranges. However, the relative throughput benefit decreases, compared to Figure 4b. This is because ALEX scans no faster than the B+Tree, so as scan time begins to dominate overall query time, the speedups that ALEX achieves on lookups become less apparent.

In order to avoid incurring overhead by scanning gaps, ALEX maintains a bitmap for each leaf node, so that each bit tracks whether its corresponding location in the node is occupied by a key or is a gap. The bitmap is fast to query and has low space overhead compared to the data size.

5.2.4 *Scalability*

ALEX performance scales well to larger datasets. To demonstrate scalability, we again run the read-heavy workload on the longitudes dataset. For this benchmark, instead of initializing the index with 50 million keys, we vary the number of initialization keys and then run the benchmark for 60 seconds. Figure 5a shows that as the number of indexed keys increases, ALEX maintains higher throughput than B+Tree. In fact, as dataset size increases, ALEX throughput decreases at a surprisingly slow rate. This occurs because ALEX maintains a constant proportion of gaps to keys in the data nodes, so even as the absolute number of keys increases, the time taken to insert a key does not increase by much, as long as the dataset distribution does not shift. Furthermore, performing model-based inserts after every expansion recalibrates the model and restores the high accuracy.

5.2.5 *Dataset Distribution Shift*

ALEX is robust to a moderate amount of dataset distribution shift. To demonstrate this robustness, we create a modified form of the longitudes dataset by first sorting the keys, then shuffling the first 50 million keys and shuffling the remaining keys. We then initialize with the first 50 million keys and run read-write workloads by gradually inserting the

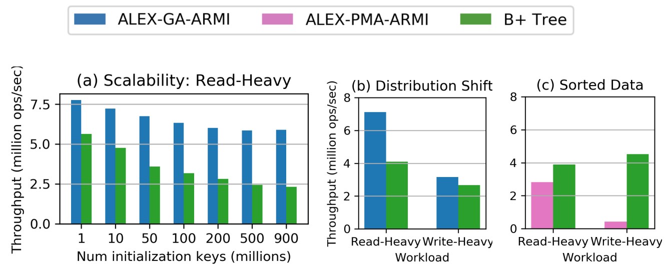

Figure 5: ALEX maintains high throughput when scaling to larger datasets and is competitive on mild distribution shift but has poor performance on sequential inserts.

remaining keys. This benchmark simulates dataset distribution shift because the keys we initialize with come from a completely disjoint domain than the keys we subsequently insert with. Therefore, ALEX must adaptively split its nodes, so ALEX-GA-ARMI here uses node splitting on inserts. Figure 5b shows that ALEX maintains throughput competitive with B+Tree on this modified dataset.

However, ALEX is not robust to adversarial patterns such as sequential inserts, in which new keys are always inserted into the right-most leaf node. Figure 5c shows that ALEX has up to 11× lower throughput than B+Tree in this scenario. If the data is known to be sequentially increasing or decreasing, ALEX design needs to be modified in ways which we discuss in Section 7. For this adversarial pattern, ALEX-PMA-ARMI is the best variant of ALEX; normally, the problem of fully-packed regions can be avoided by using either PMA or adaptive RMI. However, sequential inserts result in larger fully-packed regions that never completely disappear, so both PMA and adaptive RMI are required to address the problem.

5.2.6 *Lifetime Study*

We want to see how ALEX compares to B+Tree over the lifetime of the index, from its initialization with a small number of keys and over the course of many inserts. To evaluate this, we initialize an index on both the longitudes and longlat datasets with 1 million keys and insert until reaching 200 million keys. In order to simulate lookups throughout the lifetime of the index, for every 100 thousand inserts, we pause and do lookups for 10 thousand randomly selected keys. We show performance for ALEX-PMA-SRMI, ALEX-GA-ARMI and ALEX-PMA-ARMI. We do not show ALEX-GA-SRMI because it does nothing to avoid fully-packed region, and therefore inserts are slow.

Figure 6 shows that on the longitudes dataset, ALEX-GA-ARMI outperforms B+Tree on insert time by around 20%. For lookups, ALEX-GA-ARMI achieves up to 4× shorter lookup time than B+Tree, and lookup time remains relatively stable as the number of keys increases. This is because the ALEX RMI does not grow over time, and gaps are maintained in proportion to the number of keys, whereas the B+Tree does grow deeper over time, which makes lookups increasingly expensive.

An interesting observation is that ALEX-PMA-ARMI performance fluctuates periodically for both inserts and lookups. Since adaptive RMI initializes leaves to be around the same size, the leaf data nodes subsequently fill up at similar rates. Furthermore, the PMA always expands its size to the next power of 2. The overall effect is that all the nodes of ALEX-PMA-ARMI fill up and expand in unison, which leads to periodically fluctuating performance. ALEX-GA-ARMI does not see this effect because gapped arrays can expand at different times and to different sizes. ALEX-PMA-SRMI does not see this effect because its nodes begin with different numbers of keys, so they do not expand in unison.

Figure 6 also shows that on the longlat dataset, no variant of ALEX is able to achieve insert time competitive with B+Tree. This is because longlat is a highly non-linear dataset. ALEX can use more leaf models to handle the increased non-linearity, but using more leaf models also means more index space usage, more cache misses, and longer traversal to the leaf nodes for ALEX-GA-ARMI. In fact, the linear increase in insert time for ALEX-GA-ARMI occurs because the gapped arrays still contain fully-packed regions, despite usage of adaptive RMI. The impact of fully-packed regions grows over time; after around 150 million inserts, the ALEX variants that use PMA begin to outperform ALEX-GA-ARMI because PMA helps avoid fully-packed regions. All variants of ALEX still perform faster lookups than B+Tree.

5.3 Drilldown into ALEX Design

In this section, we delve deeper into how node layout and adaptive RMI help ALEX achieve its design goals.

During lookups, the majority of the time is spent doing local search around the predicted position. Therefore, lookup time is mainly determined by the accuracy of the RMI models. To analyze the prediction errors of the Learned Index and ALEX, we initialize an index with 100 million keys from the longitudes dataset, use the index to predict the position of each of the 100 million keys, and track the distance between the predicted position and the actual position. Figure 7a shows that the Learned Index has prediction error with mode around 8-32 positions, and with a long tail to the right. On the other hand, ALEX achieves much lower prediction error by using model-based inserts. Figure 7b shows that after initializing, ALEX-GA-ARMI often has no prediction error, the errors that do occur are often small, and the long tail of errors has disappeared. Figure 7c shows that even after 20 million inserts, ALEX-GA-ARMI maintains low prediction errors. Smaller prediction errors directly contribute to decreased lookup time.

During inserts, time is spent on finding the insert position, which involves a lookup, as well as the number of shifts that occur during insertion. Figure 8 shows that the Learned Index, which uses a single Gap-less Array to store all the data, results in a high number of shifts, which is why it is not well suited for inserts. In the static RMI layout, using PMA instead of the gapped array decreases the number of shifts per insert by 45×, because the PMA avoids fully-packed regions. Alternatively, using adaptive RMI decreases the number of shifts that the gapped array performs by 37× because it limits the size of the leaf nodes, as discussed in Section 3.4, which reduces the size and impact of fully-packed regions. Therefore, PMA and adaptive RMI are two ways of avoiding high shifts per insert resulting from fully-packed regions.

This also explains why ALEX-PMA-SRMI and ALEX-GA-ARMI are generally better than the other two ALEX variants for read-write workloads. When ALEX uses static RMI, the number of shifts for gapped array is high. Therefore, ALEX-GA-SRMI will have poor insert performance. On the other hand, when ALEX uses adaptive RMI, the number of shifts is similar for gapped array and PMA. In this case, we typically

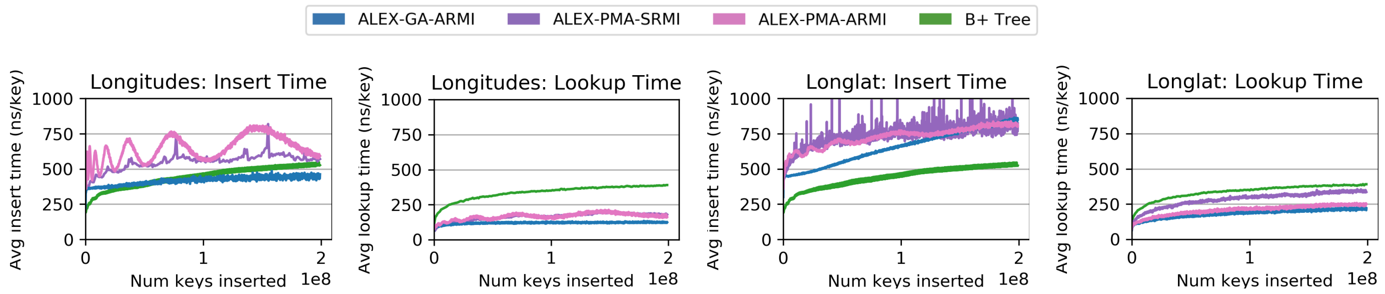

Figure 6: Lifetime Studies

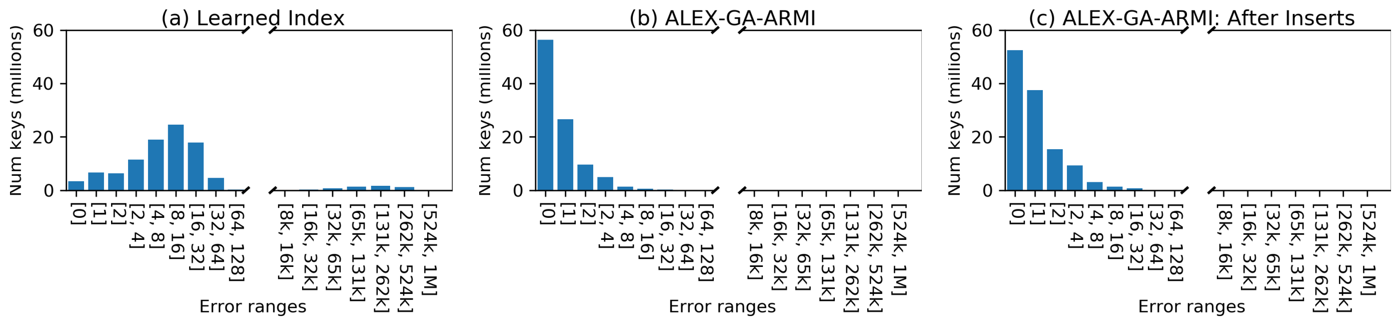

Figure 7: ALEX achieves smaller prediction error than the Learned Index, even after inserts.

prefer to use gapped array, since it achieves faster lookups than PMA and also avoids costly and unnecessary operations such as rebalances.

As a side effect, adaptive RMI avoids high-latency inserts. An insert that triggers an expansion of a large node can be slow. Adaptive RMI avoids this problem by initializing nodes to have at most a maximum bound number of keys, and splitting nodes if they grow too large. Figure 9 shows the impacts of static vs. adaptive RMI on insert latency. Here, we run a write-only workload on the longitudes dataset, where latency is measured for minibatches of 1 thousand inserts. Even though ALEX-PMA-SRMI has low median latency, it can have up to 200× higher tail latencies than ALEX-GA-ARMI. On the other hand, ALEX-GA-ARMI can achieve tail latencies competitive with B+Tree. This is because even though ALEX nodes must perform expensive expansions, B+Tree also has expensive operations, such as splitting leaf nodes and rebalancing.

5.3.1 *Data Storage Space Usage*

So far, we have parameterized ALEX such that the data storage structures have around 43% space overhead, which is similar to what B+Tree has. However, ALEX might achieve better performance if we are willing to allow higher space overhead. Figure 10 shows how throughput on a read-heavy workload changes as we increase the space overhead from 43% to 2× or 3×, and when we decrease to 20%. Having more space does often lead to increased throughput. This is because insert time is dependent on the existence of fully-packed regions, which are less likely to occur when the data node has more space with which to perform model-based inserts. However, more space provides diminishing marginal benefit. For example, the lognormal and YCSB datasets begin to have worse performance with 3× space because the data is relatively easy to model, and unnecessary extra space can lead to more cache misses. Also, the longlat dataset does not see as much improvement as other datasets because its non-linearity is still difficult to model, even with more space. To understand the dataset-specific tradeoff between storage space and performance, one can refer to the analysis discussed in Section 4.

5.3.2 *Search Method Comparison*

In order to show the tradeoff between binary and exponential search, we perform a microbenchmark on synthetic data. We create a dataset with 100 million perfectly uniformly distributed integers. We then perform searches for 10 million randomly selected values from this dataset. We use four search methods: exponential search, and binary search with three different error bound sizes. For each lookup, the search method is given a predicted position that has some synthetic amount of error in the distance to the actual position value. Figure 11 shows that the search time of exponential search increases proportionally with the logarithm of error size, whereas the binary search methods take a constant amount of time, regardless of error size. This is because binary search must always begin search within its error bounds, and cannot take advantage of cases when the error is small. Therefore, exponential search should outperform binary search if the prediction error of the RMI models in ALEX is small. As we showed in Section 5.3, ALEX maintains low prediction errors through model-based inserts. Therefore, ALEX is well suited to take advantage of exponential search.

6. RELATED WORK

There is a rich body of related work on optimizing indexes which inspired us.

Learned Index Structures: The most relevant work is the learned index from Kraska et al. [17], already discussed in Section 2.2. Although the term "learned index" has been proposed recently in [17], it has some similarities to prior

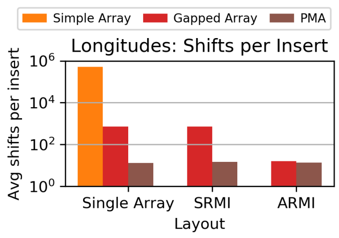

Figure 8: Shifts per insert.

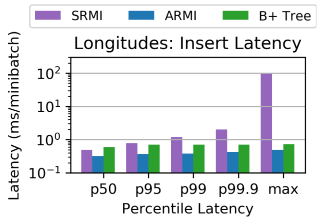

Figure 9: Insert latency.

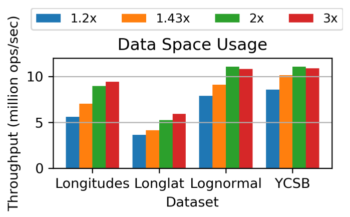

Figure 10: Data space usage.

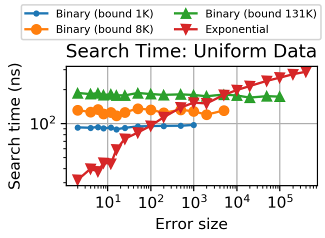

Figure 11: Exponential vs. binary search.

work that explored how to compute the way down a tree index.

Trie [16] is a classic example. Trie uses key prefix instead of the B+Tree splitters to build the tree. Masstree [24] and ART (Adaptive Radix Tree) [20] combine the idea of B+Tree and trie to reduce cache misses. Plop-hashing [18] uses piece-wise linear order preserving hashing to distribute keys more evenly over the pages. Digital B-tree [21] uses bits of a key to compute down the tree in a more flexible manner. [22] proposes the idea that when splitting in B+Tree, we can partially expand the space instead of always doubling. [10] proposes the idea of interpolation search within B+Tree nodes. This is used in SQL Server's implementation of B+Tree. The idea is that instead of using binary search, we recursively use interpolation search to locate the position of a key, or first use interpolation search to find a position, and then do a local search around that position, similar to how search operates in a learned index. This idea was also recently revisited in [1].

Other works propose replacing the leaf nodes of a B+Tree with other data structures. In these works, the inner nodes of B+Tree structure remain unchanged. Since leaf nodes take the most space in a B+Tree, these works change the leaf nodes to compress the index, while maintaining search and update performance. A-tree [9] uses linear models in its leaf nodes, while BF-tree [4] uses bloom filters in its leaf nodes.

All these works, including our index ALEX, share the idea that with some extra computation or data structure, we can reduce the number of binary search hops and the corresponding cache misses to make search faster. Also, this allows larger node sizes and hence a smaller index size. However, how we build ALEX is different in the following ways: (1) We use a model to split the key space, which is similar to a trie, but no search is required until we reach the leaf level. (2) By using linear models with good accuracy, we enable larger node sizes without sacrificing search and update performance and (3) we use model-based insertion to reduce the impact of model's misprediction, without changing the model or the training process.

Memory Optimized Indexes: There has been a large body of work on optimizing tree index structures for main memory by exploiting hardware features such as CPU cache, multi-core, SIMD, and prefetching. For example, Rao et al. proposed CSS-trees [28] to improve B+Tree's cache behavior by (1) having index node size matching the CPU cache-line size, and (2) eliminating pointers in index nodes and instead using arithmetical operations to find child nodes. Later, they extended the static CSS-trees and proposed CSB$^+$-tree [29] to support incremental updates without sacrificing CPU cache performance. The effect of node size on the performance of CSB$^+$-tree was analytically and empirically evaluated in [12]. Chen et al. proposed pB+-tree [7] to use larger index nodes and rely on prefetching instructions to bring index nodes into cache before nodes are accessed. In addition to optimizing for cache performance, FAST [14] further optimizes searches within index nodes by exploiting SIMD parallelism.

ML in other DB components: It is also worthwhile to mention the emerging trend of using machine learning to optimize databases. The most explored area is query optimization. Multiple works [27, 19, 15, 25] propose ideas to use reinforcement learning and convolutional neural networks to predict cardinalities and to optimize join queries. Also, machine learning is used to predict arrival rate of queries [23], do entity matching [26], and to predict memory access patterns for prefetching [13]. These works are orthogonal to our work, but they show that the use of machine learning enables workload-specific optimizations, which also inspired our work. One interesting difference between our work and these prior works is that the models in ALEX are learned from the training data and inference is also done on the training data, i.e., training data and test data are the same, which is not the typical use case of machine learning.

7. DISCUSSION & FUTURE WORK

In this section, we discuss our design and possible future extensions.

Role of Machine Learning: Similar to the original learned indexes, machine learning (ML) currently plays a limited, but important role in the design of ALEX. ALEX uses simple linear regression models, at all levels of the RMI. We found linear regression models to strike the right balance between computation overhead vs. prediction accuracy. We have also found these to work better than the even simpler, pure interpolation search [1]. Polynomial models, piecewise linear splines, or even a hybrid of ML models and B+Tree in the RMI structure, as originally proposed by [17] might be worth exploring, but they have not been the focus of this paper.

Concurrency Control: To use ALEX in a database system requires concurrency control for handling updates with concurrent lookups. For lookups, without Adaptive RMI, only a shared lock on the leaf data node is needed in ALEX. With adaptive RMI, to protect against concurrent modifications of the RMI structure, lookups can use lock-coupling (or crabbing) [11] while traversing the RMI to a leaf data node. Similarly, for inserts, without adaptive RMI, an exclusive lock on the leaf data node is sufficient. For cases which require an expansion, we need to hold an exclusive lock on the leaf data node and the corresponding leaf level model in the RMI, since the model needs to be retrained. With adaptive RMI with node splitting on inserts, the structure of the RMI can be modified as well. Since this is very similar to splits in a B+Tree, we believe lock-coupling [11] can be applied in ALEX, in this case as well.

Data Skew: Data skew is quite common in real-life workloads and hence it is important for any index structure to be able to deal with it. In Section 3.4, we proposed different techniques to handle data skew with dynamic inserts. Figure 5b shows that ALEX is able to handle some data skew gracefully. However, it is also easy to construct an adversarial workload where ALEX's performance degrades significantly, as shown in Figure 5c. Future work could explore even better node layouts for ALEX, for example the adaptive PMA [6] could, in theory, prevent the adversarial case shown in Figure 5c. Better models, or different adaptability strategies for the RMI are other possible directions to pursue.

Secondary Indexes: Handling secondary indexes is straight-forward in ALEX. Similar to a B+Tree, instead of storing actual data at the leaf level, ALEX can store a pointer to the data. The difficulty is in dealing with duplicate keys, which ALEX currently does not support.

Secondary Storage: Handling secondary storage, for data that does not fit in-memory is another important practical requirement. ALEX uses a node per leaf layout, which could be mapped to disk pages, and hence is secondary storage friendly. A simple extension of ALEX could store a pointer to a leaf data page in secondary storage, for every leaf node. However, as observed in [17], supporting secondary storage may require: changes to model training, introducing an additional translation table, or using more complex models.

8. CONCLUSION

This paper builds on the excitement of learned indexes and proposes a new updatable learned index, namely ALEX. We show that a careful space-time trade-off leads to an updatable data structure and also significantly improves search performance. ALEX proposes two leaf node layouts and techniques to adapt the RMI structure for updates. Given this flexibility, ALEX can adapt to different datasets and workloads. Our in-depth experimental results show that ALEX not only beats a B+Tree on three of the four datasets, using read-only and read-write workloads, it even beats the existing learned index, on all datasets, by up to 2.7× with read-only workloads. ALEX is an important case study in this new and exciting space. We believe this paper presents important learnings to our community and opens avenues for future research in this area.

9. REFERENCES

[1] The case for b-tree index structures, 2018. `http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html`.

[2] Openstreetmap on aws, 2018. `https://registry.opendata.aws/osm/`.

[3] Stx b+ tree, 2018. `https://panthema.net/2007/stx-btree/`.

[4] M. Athanassoulis and A. Ailamaki. Bf-tree: approximate tree indexing. *Proceedings of the VLDB Endowment*, 7(14):1881–1892, 2014.

[5] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is o(n log n). *Theory of Computing Systems*, 2006.

[6] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)*, 32(4):26, 2007.

[7] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 235–246, 2001.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[9] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. A-tree: A bounded approximate index structure. *arXiv preprint arXiv:1801.10207*, 2018.

[10] G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd international workshop on Data management on new hardware*, page 5. ACM, 2006.

[11] G. Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010.

[12] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b^+-trees. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, pages 283–294, 2003.

[13] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 1924–1933, 2018.

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[15] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[16] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1997.

[17] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[18] H.-P. Kriegel and B. Seeger. Plop-hashing: A grid file without directory. In *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 369–376. IEEE, 1988.

[19] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[20] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases.

In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[21] D. B. Lomet. Digital b-trees. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, pages 333–344. VLDB Endowment, 1981.

[22] D. B. Lomet. Partial expansions for file organizations with an index. *ACM Trans. Database Syst.*, 12(1):65–84, Mar. 1987.

[23] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645, 2018.

[24] 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.

[25] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212*, 2018.

[26] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 19–34, 2018.

[27] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 4. ACM, 2018.

[28] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89, 1999.

[29] J. Rao and K. A. Ross. Making b+-trees cache conscious in main memory. In *Acm Sigmod Record*, volume 29, pages 475–486. ACM, 2000.

A Proof of Theorem 2

Restatement of Theorem 2:

THEOREM 2. *The number of keys placed in the predicted location is no larger than* $2 + \left|\{1 \le i \le n-2 | \Delta_i > \frac{1}{ca}\}\right|$, *where* $\left|\{1 \le i \le n-2 | \Delta_i > \frac{1}{ca}\}\right|$ *is the number of* Δ_i*'s larger than* $\frac{1}{ca}$.

PROOF. We define a mapping $f : [n-2] \to [n]$, where $f(i)$ is defined recursively according to the following cases:

Case (1): $y_{i+2} - y_i > 1$. Let $f(i) = 1$. Case (2): $y_{i+2} - y_i \le 1, \lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor, f(i-1) \le i$ or $i = 1$. Let $f(i) = i+1$. Case (3): Neither case (1) or (2) is true. Let $f(i) = i+2$.

We prove that $\forall 1 \le i < j \le n-2$, if $f(i) > 1, f(j) > 1$, then $i+1 \le f(i) \le i+2, j+1 \le f(j) \le j+2$, and $f(i) < f(j)$.

First, when $f(i) > 1, f(j) > 1$, we know that case (1) is false for both i and j. So $f(i)$ is either $i+1$ or $i+2$, and $f(j)$ is either $j+1$ or $j+2$.

Second, if $i+1 < j$, then $f(i) \le i+2 < j+1 \le f(j)$. So we only need to prove $f(i) < f(j)$ when $i+1 = j$. Now consider the only two possible values for $f(j)$, $j+1$ and $j+2$, when $i+1 = j$. If $f(j) = j+1 = i+2$, by definition we know that case (2) is true for $f(j)$. That means $f(j-1) = j$ or 1. But we already know $f(j-1) = f(i) > 1$. So $f(i) = f(j-1) = j = i+1 < i+2 = f(j)$. If $f(j) = j+2$, then $f(i) \le i+2 < j+2 = f(j)$.

So far, we have proved that $f(i)$ is unique when $f(i) > 1$. Now we prove that the key $x_{f(i)}$ is not placed in $\lfloor y_{f(i)} \rfloor$ when $f(i) > 1$, i.e., either case (2) or case (3) is true for $f(i)$. In both cases, $y_{i+2} - y_i \le 1$, and the rounded integers $\lfloor y_{i+2} \rfloor$ and $\lfloor y_i \rfloor$ must be either equal or adjacent: $\lfloor y_{i+2} \rfloor - \lfloor y_i \rfloor \le 1$. That means $\lfloor y_{i+1} \rfloor$ must be equal to either $\lfloor y_{i+2} \rfloor$ or $\lfloor y_i \rfloor$.

We prove by mathematical induction. For the minimal i s.t. $f(i) > 1$, if case (2) is true, $\lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor$. That means x_{i+1} cannot be placed at $\lfloor y_{i+1} \rfloor$ because that location is already occupied before x_{i+1} is inserted. And $f(i) = i+1$ by definition. If case (2) is false, since we already know $y_{i+2} - y_i \le 1$, $f(i-1) = 1$ or $i = 1$, it follows that $\lfloor y_{i+1} \rfloor > \lfloor y_i \rfloor$. That implies $\lfloor y_{i+1} \rfloor = \lfloor y_{i+2} \rfloor$. So x_{i+2} cannot be placed at $\lfloor y_{i+2} \rfloor$. And $f(i) = i+2$ because case (3) happens.

Given that the key $x_{f(i-1)}$ is not placed at $\lfloor y_{f(i-1)} \rfloor$ when $f(i-1) > 1$, we now prove it is also true for i. The proof for case (2) is the same as above. If case (2) is false, and $\lfloor y_{i+1} \rfloor > \lfloor y_i \rfloor$, the proof is also the same as above. The remaining possibility of case (3) is that $\lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor$, and $f(i-1) = i+1$. The inductive hypothesis states that x_{i+1} is not placed at $\lfloor y_{i+1} \rfloor$. That means x_{i+1} is placed at a location equal or larger than $\lfloor y_{i+1} \rfloor + 1 = \lfloor y_i \rfloor + 1$. But we also know that $\lfloor y_{i+2} \rfloor \le \lfloor y_i \rfloor + 1$. So x_{i+2} cannot be placed at $\lfloor y_{i+2} \rfloor$ which is not on the right of x_{i+1}'s location. Since case (3) is false, $f(i) = i+2$.

By induction, we show that when $f(i) > 1$, the key $x_{f(i)}$ cannot be placed at $\lfloor y_{f(i)} \rfloor$. That means when we look up $x_{f(i)}$, we cannot directly hit it from the model prediction. Since we also proved that $f(i)$ has a unique value when $f(i) > 1$, the number of misses from the model prediction is at least the size of $S = \{i \in [n-2] | f(i) > 1\}$. By the definition of $f(i)$, $S = \{i \in [n-2] | y_{i+2} - y_i \le 1\}$. Therefore, the number of direct hits by the model is at most $n - |S| = 2 + |\{i \in [n-2] | y_{i+2} - y_i > 1\}| = 2 + \left|\{1 \le i \le n-2 | \Delta_i \ge \frac{1}{ca}\}\right|$.

□

B Adaptive RMI and Leaf Sizes

In Section 3.4, we described why static RMI can result in wasted leaves and large fully-packed regions, and why adaptive RMI improves performance by bounding the size of a leaf node after initialization. Figure 12 shows that when initializing ALEX on 200 million keys of the longitudes dataset, the static RMI layout results in wasted leaves, as well as leaves that are big and therefore more likely to have large fully-packed regions. On the other hand, adaptive RMI initialization results in leaves that a maximum number of keys and fewer wasted leaves.

C Dataset Characteristics

The CDFs of the four datasets are shown in Figure 13. Even though the longitudes and longlat datasets look similar at a global scale, zooming into the CDF in Figure 14 shows that the longlat dataset is much more non-linear at a smaller scale.

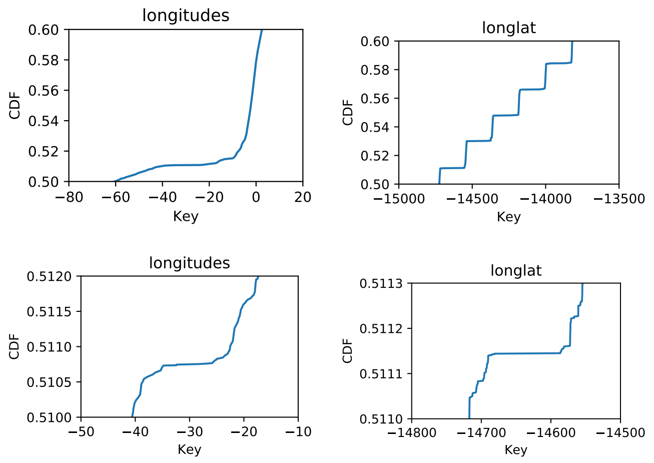

Figure 14: Zooming in on CDFs of longitudes and longlat. Both plots on the first row show 10% of the CDF. The plots on the second row zoom in even further, showing 0.2% of the CDF for longitudes and 0.03% of the CDF for longlat.

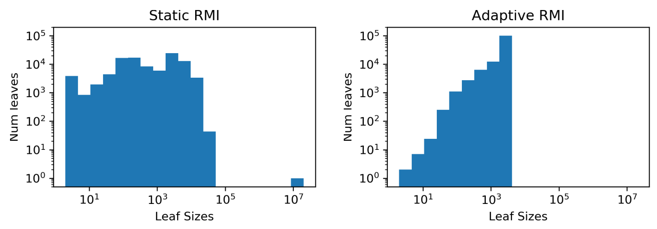

Figure 12: Adaptive RMI achieves more consistent leaf size.

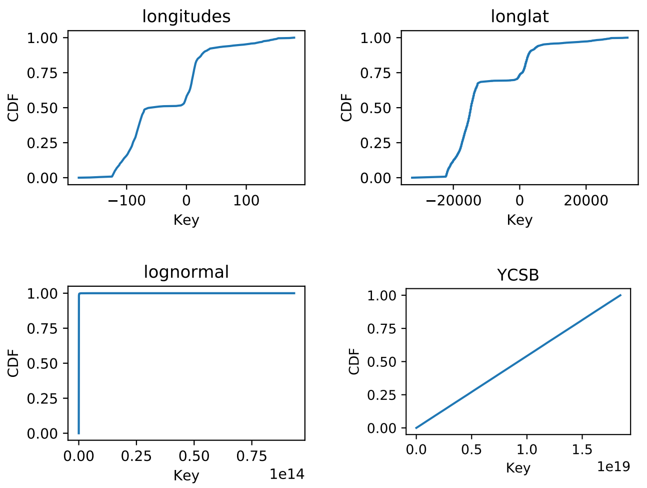

Figure 13: Dataset CDFs.

In particular, the CDF for longlat looks like a step function at local scale. This is due to the way in which the longlat dataset is constructed. In order to concatenate a longitude with a latitude, we round the longitude to the nearest integer degree, multiply by 180 (which is the size of the domain of latitudes), and add the latitude. In effect, this concatenation first groups locations into their nearest degree of longitude, then sorts within each group by latitude. If we iterate over the longlat values in sorted order, we traverse locations around the world one longitude "strip" at a time. Each strip shows up as a step function in the CDF. This results in a much more uneven distribution of values than the longitudes dataset, which traverses locations in globally sorted order.

The lognormal dataset is created by uniformly selecting 190 million values according to a lognormal distribution with $\mu = 0$ and $\sigma = 2$, multiplying by 1,000,000,000, and rounding down to the nearest integer.