RUSLI: Real-time Updatable Spline Learned Index

Mayank Mishra
TCS Research
Mumbai, India
mishra.m@tcs.com

Rekha Singhal
TCS Research
Mumbai, India
rekha.singhal@tcs.com

ABSTRACT

Machine learning algorithms have accelerated data access through 'learned index', where a set of data items is indexed by a model learned on the pairs of data key and the corresponding record's position in the memory. Most of the learned indexes require retraining of the model for new data insertions in the data set. The retraining is expensive and takes as much time as the model training. So, today, learned indexes are updated by retraining on batch inserts to amortize the cost. However, real-time applications, such as data-driven recommendation applications need to access users' feature store in real-time both for reading data of existing users and adding new users as well.

This motivates us to present a real-time updatable spline learned index, RUSLI, by learning the distribution of data keys with their positions in memory through splines. We have extended RadixSpline [8] to build the updatable learned index while supporting real-time inserts in a data set without affecting the lookup time on the updated data set. We have shown that RUSLI can update the index in constant time with an additional temporary memory of size proportional to the number of splines. We have discussed how to reduce the size of the presented index using the distribution of spline keys while building the radix table. RULSI is shown to incur 270ns for lookup and 50ns for insert operations. Further, we have shown that RUSLI supports concurrent lookup and insert operations with a throughput of 40 million ops/sec. We have presented and discussed performance numbers of RUSLI for single and concurrent inserts, lookup, and range queries on SOSD [9] benchmark.

CCS CONCEPTS

• **Information systems → Unidimensional range search**; **Unidimensional range search**.

KEYWORDS

Learned index, Real-time inserts, Spline-based index

ACM Reference Format:
Mayank Mishra and Rekha Singhal. 2021. RUSLI: Real-time Updatable Spline Learned Index . In *Fourth Workshop in Exploiting AI Techniques for Data Management (aiDM'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3464509.3464886

1 INTRODUCTION

Machine learning and deep learning algorithms are playing a crucial role in accelerating the performance of traditional systems. Singhal [17] envisions data-driven enterprise IT systems, where components of the data management system, such as an index to access data, are replaced by machine learning algorithms. Learned Index, first proposed by Kraska [10], builds a machine learning model by learning the cumulative distribution (CDF) on data values with their storage positions and uses this model as an index to access any data element. Precisely, the learned index learns a function 'F' for $(key\_val, position)$ pairs, such that $F(key\_val) = position$, where data is sorted on 'key' and an element with value 'key_val' is stored in storage at location '$position$'.

The in-memory data store used in ML/DL applications often needs both lookups and inserts in real-time, e.g., feature stores in ML pipeline may need to access and insert user-based features. Recommendation applications access a feature store in real-time to retrieve contextual features for a user before inference, while also adding features for new users. Also, episodic memory [11] used in RL needs to access and store rewards for each state. These inserts may be in real-time for multi-agent RL, where agents are accessing the same episodic memory. A new data insert in an in-memory store requires updating the learned index, preferably quickly (in nanoseconds) to serve concurrent users in real-time.

The deep neural networks that can learn non-linear CDFs in a compact representation, are not easy for adapting to new training sets without re-training the CDF model. [7] has proposed to update CDF on a batch of inserts/deletes, model the CDF drift due to insertion, and calibrate the prediction of the learned index. However, we believe in delaying the learning of CDF for the fast update of the learned index. We propose to insert a new data element at the end of the data set, which may disturb the sorting order of data elements. Therefore, the need is for the learned index to maintain the logical sorted order of data elements, while they may be out of order in the physical store. We focus on updates due to insertions only in this paper, so handling delete operations is out of scope for this paper.

Alex [3], PGM [5] and RadixSpline [8] have discussed solutions for updating learned index on inserting new data elements. PGM [5] and RadixSpline [8] employ retraining of the model incurring atleast $O(N)$ time complexity for 'N' number of data elements, which is not suitable for real-time inserts. Alex [3] supports inserts in real-time by employing gaps in the data layout. Like Alex [3], we also employ additional memory to update the learned index on inserts in real-time. However, unlike Alex, the additional memory used can be allocated dynamically based on the frequency of inserts and is not tied to the data layout. This gives complete flexibility on the size and location of the additional memory used for supporting

aiDM'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8535-0/21/06…$15.00
https://doi.org/10.1145/3464509.3464886

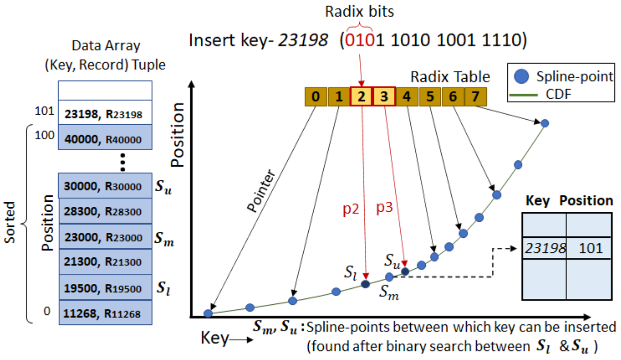

Figure 1: Index on data, as pairs of $(key, record)$, stored in data array$[0, 100]$ is learned as a set of spline keys . A new data element $(23198, r_{23198})$ is inserted at data array$[101]$. Index is updated by inserting $(23198, 101)$ in temporary memory between S_d and S_m

fast inserts in real-time, leading to better throughput and control on the learned index's size.

RadixSpline [8] incurs $O(1)$ amortized cost for an insert. It is efficient for bulk inserts by amortizing the cost of overall inserts. We propose an extension of RadixSpline, real-time updatable spline learned index (RUSLI), to support efficient real-time inserts at the cost of temporary memory to store newly inserted data elements for a short period of time till it gets reorganized, where the temporary memory used for inserts is released. We propose to retrain RUSLI along with the maintenance cycle of its application,

We use a bounded sized temporary memory, between any two spline points, which is searched in parallel to binary search on data elements between two spline points. We exploit parallelism to keep the lookup time unaffected due to extra search in the temporary memory for the newly inserted data elements. RUSLI is compact enough such that the radix table and the temporary memory fit in cache, and hence the performance gains can be maintained. Such an approach potentially saves more time compared to re-training the learned index or switching to a traditional indexing algorithm for real-time update workloads.

RUSLI also supports range queries, mapped as two lookup operations, however, filtered data may be accessed in random order rather than a sequential read. This may impact the application of RUSLI to index data on disk. Also, the data outputted may not be in sorted order. We have modified the SOSD [9] benchmark to evaluate the performance of real-time inserts and lookup in RUSLI and compared it with that of RadixSpline. We have also compared RUSLI throughput using multithreading as presented in [12].

The key contributions of the paper are as follows:

- The real-time updatable learned index, RUSLI, to support uniformly distributed insert operations in constant time. RUSLI needs an additional temporary memory bounded by $N_{sp} * log_2(2 \times max\_err)$, where N_{sp} is the number of spline keys for N data elements and max_err is maximum tolerable position interpolation error (a configurable parameter).
- An algorithm for fast lookup on data keys including the newly inserted keys as well.
- An algorithm to achieve high throughput for concurrent real-time inserts and lookup operations, while updating the index.
- An extension of SOSD [9] benchmark to evaluate the performance of learned index for updation.

The paper is organized as follows. We present the related work in Section 2. We start with background of RadixSpline [8], foundation for RUSLI, in Section 3. Section 4 presents the innovations in data structure employed to support updation of RUSLI on inserts in real-time. The support of delete operation is out of the scope of this paper. The real-time updation and lookup algorithms of RUSLI are presented in Section 5 and 6 respectively. The performance evaluation of RUSLI for both lookup and updation on real-time insert operations is presented in Section 7 and finally, we concludes in Section 8 with future work.

2 RELATED WORK

Learned index employs machine learning algorithms to learn the relation between data distribution and their storage locations to build an index. Their primary focus has been to enable fast lookup. Learned index assumes data to be stored in a sorted manner while learning their distribution with respective storage locations. RMI [10] is the first learned index proposed by MIT for data indexed on a single key (or dimension). Further, the learned index synthesis framework is proposed in [13] to generate a learned index with an optimal size of memory and look-up time. Their performance benchmarks have been discussed in detail in [12] and [9]. These learned indexes are extended for accessing data on multiple keys, using workload knowledge to design optimal storage patterns - the details are presented in Spatial [16], Flood [14] and Tsunami [4]. A survey of these learned indexes has been presented in [2]. We focus on building a learned index on a single key, in this paper, but address the challenges of updating it in real-time with comparable lookup time. Please note, for RUSLI, lookup query spans across newly inserted data elements as well.

IFB-tree[6], PGM [5], Alex [3], and RadixSpline [8] have discussed solutions for updating learned index on new data inserts. IFB-tree[6] is a modified version of B-tree, inherently maintains updates, which uses linear interpolation if the accuracy is guaranteed to be in a predefined error tolerance *max*. However, since the IFBtree still maintains the underlying B-tree, the updates are handled in $O(n \log n)$ time. PGM learns the line equations on the sorted data points. It builds an index recursively to search a line for a given data point. It supports updates with the complexity of $O(num\_lines \times \log num\_lines)$.

Alex [3] supports inserts in Btree by partitioning data and building the model tree bottom up so that any insert may lead to a change in models of a specific path only. It also employs gaps in data layout to provision for future inserts. RUSLI also employs additional memory like Alex, but this additional memory is disjoint to the data layout. This gives flexibility in choosing the size and location of the additional memory required for the faster update of the learned index - Hence, better throughput and latency for look-ups and inserts. Further, Alex uses a hierarchy of models, so the effect of an insert is cascaded upwards to retrain the models in the path. Therefore, an insert's latency is bounded by the length of this path. However, RUSLI, employs the radix hash table approach as discussed below, to keep the insert's latency constant.

RadixSpline [8] incurs $O(1)$ amortized cost for an insert by retraining on a batch of inserts. It is efficient for bulk inserts by amortizing the cost overall inserts. We propose an extension of RadixSpline to support efficient real-time inserts at the cost of configurable additional memory to store newly inserted data elements for a short period of time till retraining occurs. We use bounded-sized memory for inserts between two spline points, as discussed in Section 4, which is searched in parallel to binary search on data elements between two spline points. We exploit parallelism to keep the look-up time same as experienced by RadixSpline [8], where RUSLI lookup includes inserted data elements as well - this is discussed further in Section 6.

Google [1] has presented a learned index for accessing data stored on disk, where the challenge is to find the correct block, otherwise search within $\pm max\_error$ of approximate block position may involve many disk accesses. We present a real-time update algorithm for the learned index in the context of data being in memory, however, we believe that it may be extended for data stored on disk as well.

3 BACKGROUND - RADIXSPLINE

Consider a data array having (key, record) pairs, as shown in Fig 1. The data is sorted on key and stored from positions pos_0 to pos_{N-1}, such that, key_i is stored at position pos_i and, if $key_i <= key_j$, then, $pos_i < pos_j$. RadixSpline [8] learns the CDF of (key, position) pairs (or tuples) ((key_i, pos_i), where, $0 \leq i \leq N-1$) shown as a curve in Fig 1.

RadixSpline Model Creation: RadixSpline learns the CDF by creating a spline-based model. Fig 1 shows the model in the form of blue circles over the CDF curve known as spline-points. The spline-points are stored in an array, (spline key, position), sorted on their key values. The array of spline-point is accessed by a radix table, shown in the Figures 1. A radix table maps the most-significant-bits (MSB) of a given data key to the closest spline-point in the array of spline-points. A radix table's size is 2^{radix_bits}, where $radix\_bits$ is a configurable input parameter and determines the most significant bits used in the radix table. For example, in Fig 1, spline-point S_l's key having the first 3 bits as 010, is mapped to the third entry in the radix table. There can be multiple eligible spline-points to be pointed by a radix table entry (all having the same first $radix\_bits$ bits in their respective keys), however, the spline-point with the smallest key is selected.

The model employs $GreedySplineCorridor$[15] to create spline-points. The idea behind the spline-based model is that interpolation using positions of two closest spline-points surrounding the lookup key k will give an estimated position p within an error bound of $\pm max\_err$. For more details, please refer to RadixSpline [8].

Lookup using RadixSpline: Consider an example of lookup for a key $k = 23198$ represented as "0101101010011110" in binary. Using RadixSpline, where $radix\_bits = 3$, the first 3 bits of k are 010 (i.e. 2 in decimal). Thus, spline-point pointed by 2^{nd} entry of the radix table is the first spline-point, or the lower bound, in the spline-point range in which k is placed, let this spline-point be S_l (l denotes "lower"). To find the upper bound of the spline-point range the next entry (3^{rd} entry) of radix table is looked up, which points to S_u (u denotes "upper").

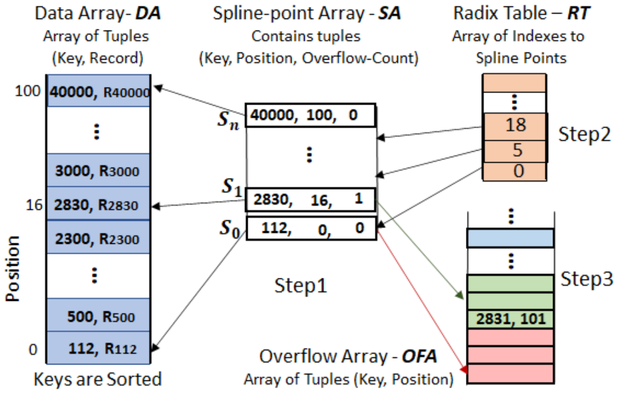

Figure 2: The Fig shows the internal structure of different data-structures used in RUSLI. Spline-point's index in spline-Point array along with "OverflowCount" entry is used as offset to access Overflow Array.

To find the approximate position of key k using interpolation, the spline-points immediately surrounding it are required. Such surrounding spline-points are found by using binary search for k over the spline-point range bounded by S_l and S_u. The binary search returns the S_m such that S_m's key's value $key(S_m) \leq k$ and $k - key(S_m)$ is minimum among all spline points in a range bounded by spline-points S_l and S_u, thus, ensuring immediately surrounding lower spline-point. The adjoining spline-point whose key is greater than that of S_m is the other surrounding spline-point as shown in Fig 1. The approximate position of k is then found by interpolating between the positions of S_m and S_u as follows:

$$p_i = pos(S_m) + (k - key(S_m)) \times \frac{pos(S_u) - pos(S_m)}{key(S_u) - key(S_m)} \quad (1)$$

where, $pos(S)$, $key(S)$ denote position and key of spline-point S.

Binary search is again applied over the keys in position range $p_i - max\_err$ to $p_i + max\_err$, which returns the position p. The position p either is - a) the position of the first occurrence of key k, if it is found, or b) the position of key which is immediately lower than k. A linear scan beginning with p up-till the position where a key larger than k is found completes the lookup.

4 RUSLI DATA STRUCTURES

RUSLI builds upon RadixSpline data structure as shown in Fig 2. It maintains a data array, referred to as DA, the array to store spline-points referred to as spline-point Array SA, and the Radix Hash Table RT. In addition, RUSLI keeps a temporary Overflow Array, OFA to support real-time inserts. Similar to SA, the OFA also stores ($key, position$) tuples for newly inserted keys. Every spline-point in the SA has a fixed number of slots in OFA.

Overflow Array OFA size: The number of slots in OFA for every spline-point in SA, denoted by O_s is given by the following equation:

$$O_s = \lceil log_2(2 \times max\_error) \rceil \tag{2}$$

where max_err denotes maximum interpolation error. The reason for keeping O_s a function of max_err is to keep a bound on the search time. This is discussed in detail in RUSLI's lookup algorithm in Section 6. The total number of entries in OFA is:

$$size(SA) \times O_s = size(SA) \times \lceil log_2(2 \times max\_error) \rceil \tag{3}$$

Addition to "spline-point": RUSLI extends the spline-point from a "$(key, position)$" tuple to a "$(key, position, overflow\_count)$" tuple. The $overflow\_count$ of a spline-point S keeps the count of newly inserted keys, which are surrounded by spline-point S at the lower side (for simplicity, such keys are referred to as keys belonging to S). The $overflow\_count$ is also used as an index to the OFA entries which belong to S. For example, if we want to access all newly inserted keys belonging to i^{th} spline entry S_i in the spline-array SA then the range of OFA's indices will be given by - $[i \times O_s, i \times O_s + overflow\_count - 1]$.

All the spline-points are initialized with $overflow\_count = 0$. A 0 value of $overflow\_count$ for any spline-point S indicates that no new key has been inserted in the space reserved for S in OFA (using as a flag). We employ unsigned 8-bit integers for $overflow\_count$. Note that from equation 2, the 8-bit size of $overflow\_count$ will be sufficient even for very large values of max_err.

RUSLI Model Size: Every entry of OFA needs two 64 bit unsigned integers. Thus, the total amount of extra space needed to support inserts (in bytes) is $size(OFA) \times 16 + size(SA) * 1$. Apart from this, RUSLI also needs $2^{radix\_bits} \times 4$ (RT entries are 32 bit unsigned integers) and $size(SA) \times 16$ to maintain RT and SA respectively. We present RUSLI model sizes for different datasets and configurations using SOSD [9] benchmark, in Table 2 in Section 7. Table 1 lists all the acronyms and functions used for the presentation of RUSLI lookup and insert algorithms[1].

Possible Optimizations on Model Size: RadixSpline [8] keeps two memory structures for spline-points - radix table and spline-points array. The motivation for this seems to minimize the size of the radix table, which could be very large for larger data size. RadixSpline controls the size of radix table by keeping $radix\_bits$ as a configurable parameter. Thus, a poor choice of $radix\_bits$ and non-uniform distribution of spline-points may lead to a large number of empty buckets and chaining in each bucket. We can get rid of the spline-points array and maintain only the radix table storing the spline-points tuple.

To optimize the size of the radix table, we can keep only those buckets in memory that have at least one spline-point. We can avoid the chaining by learning the CDF of spline-points and partitioning the CDF instead of the data range to create the buckets. We can choose the $radix\_bits$ as the minimum number of most significant bits (MSB) required to distinguish between any two buckets of the radix table. For a skewed distribution of spline points, the MSB bits may be large in number. Since the radix hash table has 2^{radix_bits} buckets, many of the buckets may be empty. To reduce the memory required by the radix table, it could be stored as a sequence of dense fragments.

Table 1: Acronyms/Functions and their descriptions.

Acronym	Description
RT	Radix Table.
SA	Array to store Spline-points.
DA	Data Array in memory.
OFA	Overflow Array to store inserted keys, positions.
$radix\_bits$	Number of MSBs of a key used for hashing into RT.
max_err	Maximum permissible interpolation error.
O_s	Number of overflow slots per spline-point.
getSplinePoint (RT, key k)	It finds the spline-points s and $s+1$ surrounding the key k and returns s. The process is discussed as part of RadixSpline Lookup algorithm in Section 3.
getInterpolPos (splinePt s, k)	It uses Equation 1 using spline points s and $s+1$ to find the approximate position of key k.

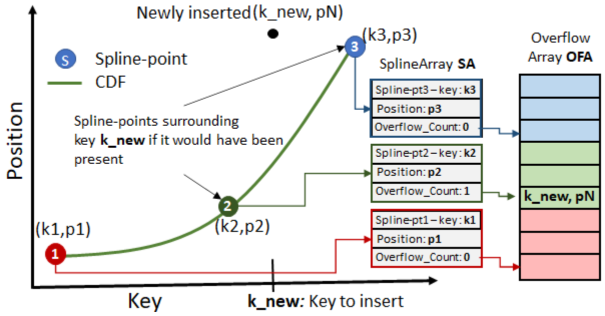

Figure 3: Inserting a new key

We can maintain a 'status' bit per bucket (like a bloom filter) to check if the bucket has some spline-point. If the bucket mapped by a radix number has a spline-point then the corresponding bit value of the radix number is 1 otherwise 0.

Also, to avoid linear search in radix table during lookup to find S_l and S_u, a learned index may perform bitwise operations on 'status' to get the rightmost and leftmost bits, which are set to 1, to the bit corresponding to the mapped bucket for a lookup key.

5 RUSLI INSERT ALGORITHM

We now present an algorithm for updating the learned index during real-time key insert operations, as implemented in RUSLI. Let k_{new}, pN be the key and position tuple to be inserted (pN denotes the position where record affiliated with key k_{new} is stored, this will be the first non-filled location in DA). The first step is to do a lookup for the spline-points which surround the value of the k_{new}. This is because k_{new} should be placed where it will be looked for in the future. For simplicity of explanation, let us assume the spline-points surrounding k_{new} are stored at index 2 and 3 in SA (Refer Fig 3), let us denote them by S_2 and S_3 respectively. Step-1 in Insert Algorithm makes use of the function $getSplinePoint$ to get to spline-point S_2. Function $getSplinePoint$ uses lookup approach as that of RadixSpline explained in Section 3.

[1] data point, data item, data key, and data element are used interchangeably.

Algorithm 1: Insert into RUSLI

```
Input: (key, position) tuple "(k,p)" to be inserted.
Data: Kindly refer to Table 1 for description of the
      acronyms and functions used in this algorithm.
Result: Success or Failure
1 s,id ← getSplinePoint(RT, k)
    // Find the lower of the spline-point pair surrounding k, with
    index id of s in SA.
2 if s.overflowCount == $O_s$ then
3     return Failure   // No overflow space left for spline s.
4 else
5     emptySlot ← id × $O_s$ + e.overflowCount
          // get empty slot index in OFA corresponding to s.
6     OFA[emptySlot] ← k,p
          // Insert tuple "k,p" at location emptySlot of OFA.
7     e.overflowCount++
          // Increment ptr to next empty slot.
8     return Success
```

Step-2 of the algorithm checks whether the space allocated for spline-point S_2 in OFA has room for k_{new}, pN by checking whether $s.overflowCount == O_s$ (indicating full). In the example shown in figure 3, $S_2.overflowCount$ was 0, indicating empty space. In step-5 the absolute index of the free-location in OFA is calculated which is further filled by placing k_{new}, pN in step-6. In step-7 the $overflowCount$ is incremented to maintain the correct state. Figure 3 shows the final state after insertion with k_{new}, pN placed in a green box in OFA.

Any further inserts for data key mapping to the same spline S_2 will happen till $s.overflowCount! = O_s$, after which no more inserts for S_2 are possible. It should be noted that newly inserted keys are not maintained in sorted order in OFA, however, all the newly inserted keys belonging to spline-point S_2 will be smaller than the smallest of keys (including newly inserted ones) belonging to spline-point S_3.

Since space corresponding between two adjoining spline-points in OFA is bounded by O_s, RUSLI can support only as many operations between any two adjoining splines before retraining. This is to keep lookup time unaffected by insert operations as discussed later in Section 6. However, RUSLI may keep multiple OFAs that can be accessed in parallel using multi-threading. The overheads of multi-threading are dominant in single insert operation, however, these overheads get amortized for executing a large number of concurrent operations.

Concurrent real-time Inserts Insert algorithm inherently allows concurrent insert operations corresponding to different pairs of consecutive spline-points because of non-overlapping overflow spaces in OFA. Only the set of inserts between the same pair of spline-points get serialized. RUSLI can exploit multi-threading to support a large number of uniformly distributed concurrent real-time insert operations - the performance numbers are discussed in Section 7.

Skewed Inserts In the case of skewed insert operations, overflow-space corresponding to certain consecutive spline-point pairs will fill up more quickly than others. Keeping more than the bounded size O_s per spline-pair will slow down lookup operations for these newly inserted keys. In such scenarios, retraining of RUSLI is initiated to rebuild the index. If a user expects to see data elements in certain ranges to be more inserted, then RUSLI can appropriately allocate more OFAs proportional to the number of expected insert operations. Alternatively, RUSLI can learn insertions distribution from the history and use linear regression to allocate the right size of overflow arrays assuming the same pattern of future insert operations.

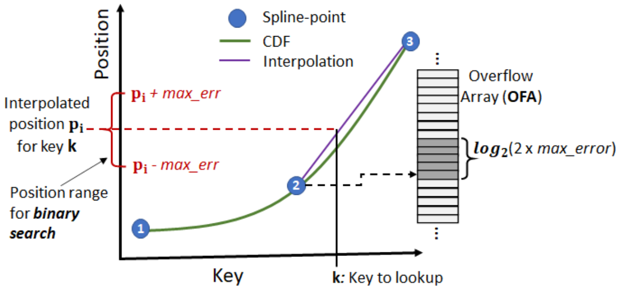

Figure 4: Lookup a key

6 RUSLI LOOKUP ALGORITHM

In RUSLI, a lookup key may reside in the data set or may have been inserted later and is part of OFA (as it is an updatable learned index). Therefore, the RUSLI lookup algorithm has two parts - First part (steps 1-7) involves lookup in the data array (similar to RadixSpline). The second part (steps 8-18) involves a lookup in the overflow array OFA. We begin by explaining the first part followed by the second.

Lookup in Data Array Please refer the RUSLI lookup algorithm 17 and example mentioned in Fig 4. Step-1 is to locate the lower of the surrounding spline-points pair for the lookup key k. Let us denote the surrounding spline-point pair as S_2 and S_3 as shown in example in Fig 4. In step-2, the approximate interpolation position p_i of k is found using function $getInterpolPos$. In step-3, binary search is applied between positions $p_i - max\_err$ and $p_i + max\_error$ for key k as shown in red in Fig 4. Binary search returns the first position p, where key k is found, or position of key just smaller to k. Steps-6,7, and 8 involve a linear scan over DA starting from position p till the position where the keys are same as k.

Lookup in Overflow Array In the second part of the algorithm, we begin by using the spline-point S_2 returned by $getSplinePoint$ in step-1 as mentioned above. In step-9, we check whether spline-point S_2 has any overflows associated by checking the value of $S_2.overflowCount$. If the value is not 0, the region of OFA belonging to S_2, shown in a darker shade in the Fig 4, is linearly scanned to find positions corresponding to k (steps 11-16) and a consolidated list of positions is returned (step-17).

Typical size of O_s The reason behind keeping $O_s = \lceil log_2(2 \times max\_error) \rceil$ is to keep the worst-case search time-bound same as RadixSpline [8]. Binary search time taken over $p_i - max\_err$ and

Algorithm 2: Lookup in RUSLI

```
Input: key k to be searched.
Data: Kindly refer to Table 1 for description of the
      acronyms and functions used in this algorithm.
Result: list of positions if k is found, else, empty list
1  s,id ← getSplinePoint(RT, k)
     // Find the lower of the spline-point pair surrounding k, with
     index i of s in SA.
2  $p_i$ ← getInterpolPos(s, k)
     // Get approximate interpolated position of k using
     interpolation over s and s+1.
3  p ← binSearch($p_i$ - $max\_err$, $p_i$ + $max\_err$, k)
     // applying binary search to get the position $p$ of key k's
     first occurrence or last position of key just smaller than k.
4  positionList ← $\phi$   // Initializing empty list for positions.
5  currentPos ← p
6  while DA[currentPos].key == k do
                                              // Scanning DA
7      positionList.add(currentPos)
8      currentPos++
9  if s.overflowCount != 0 then
10     overFlowBegin ← id × $O_s$
           // finding beginning and last index for OFA scan
11     overFlowEnd ← overFlowBegin + e.overflowCount - 1
12     currentPos ← overFlowBegin
13     while key(currentPos) ≤ overFlowEnd do
                                              // scanning OFA
14         if OFA[currentPos].key == k then
15             positionList.add(OFA[currentPos].position)
16             currentPos++
17 return positionList
```

$p_i + max\_error$, in step-3, in worst case is $O(log_2(2 \times max\_error))$, which is same as worst-case sequential search time over an array of size $log_2(2 \times max\_error)$ (same as size of O_s). These two searches over *DA* and *OFA* can be done in parallel after step-1 (locate the surrounding spline-points). For single lookup multi-threading overheads are dominant, however these overheads get amortized for executing a large number of concurrent operations.

Range queries A range query requires to return all data elements within a range, which is specified by k_l and k_u as lower and upper key bounds respectively, i.e., outputs a key k if $k_l <= k <= k_u$. There are two approaches to implement range queries.

First approach involves two lookups, one for k_l and other for k_h which will return position lists Dp_l and Dp_h corresponding to *DA* and Op_l and Op_h corresponding to *OFA*. Lookups for *DA* will involve sequential scan between first position-element of list Dp_l and last position-element of Dp_h. Lookups for *OFA* will involve sequential scan with key comparison at every position between first position-element of list Op_l and last position-element of Op_h.

Second approach involves only single lookup involving k_l which will return position lists Dp_l *DA* and Op_l corresponding to *OFA*. Lookups for *DA* will involve a sequential scan with key comparison at every position starting from the first position-element of list Dp_l till the time we encounter a key greater than k_h. Lookups for *OFA* will involve sequential scan with key comparison at every position starting from first position-element of list Op_l till the time we encounter a key greater than k_h. However, the sequential scan with key comparison continues till O_s more steps to be assured that k_h can not be further found, as, in *OFA* the space of spline-point corresponding to k_h would have surely be crossed. In an experiment using the SOSD benchmark’s Amazon dataset (200M keys), we found that range queries with a single lookup followed by traversal are quicker by (100ns) than range queries involving two lookups. We selected k_l and k_h, such that, there are 10000 keys between them.

Since the data elements including the inserted elements are not in sorted order, so range queries spanning across newly inserted data elements may experience random access to data storage, and also returned set may not be in sorted order.

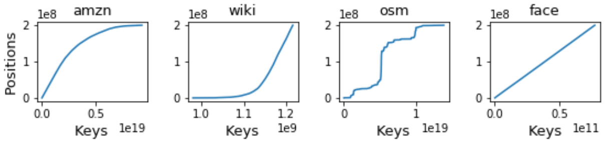

Figure 5: CDFs of SOSD benchmark Datasets (200 M)

7 PERFORMANCE EVALUATION

We have used the SOSD [9] benchmark to evaluate RUSLI’s performance in terms of lookup time and memory usage. SOSD benchmark does not have a mechanism for evaluating insert operations. We have extended the SOSD benchmark to evaluate the time taken by the learned index to update itself for an insert operation. Further, motivated by [12], we evaluated RUSLI for throughput on concurrent insert and lookup operations using multi-threading.

Datasets and System Configuration- We have used datasets, namely, *amzn* (data containing book popularity), *wiki* (Wikipedia edit timestamps), *osm* (Open street Maps cell IDs) and *face* (Ids of Facebook users), each containing 200 Million, 64 bit keys in sorted order to benchmark learned indexes. Fig 5 shows the CDF plots of these datasets. The shape of CDF impacts the number of spline-points and hence the RUSLI model size, which is visible in Table 2 for *amzn* and *face* datasets. We have evaluated RUSLI on a 56 core Intel Xeon server with 64 GB RAM running Ubuntu 18.04.

Model Building- RUSLI, like RadixSpline (RS), uses a single-pass build process and takes same time to build. The only difference being that RUSLI also allocates space for Overflow Array (*OFA*) for future inserts. The size information for allocation (number of spline-points, *max_error*) is available after the single pass itself. During experiments, we didn’t find any difference in build times of RS and RUSLI. As build time of RS (0.5 to 2.5 sec for different datasets) [8] is shown to be much lower than RMI and similar to ART and BTree, the same observation can be made for RUSLI too. Table 2 compares the model sizes for RUSLI and RadixSpline [8] for

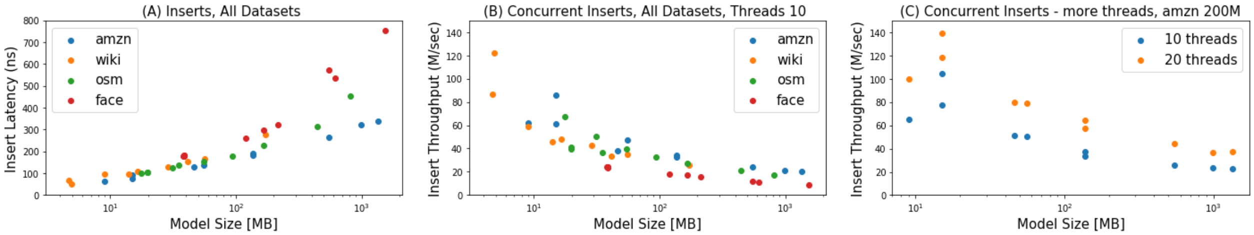

Figure 6: (A) Single Insert Latency for RUSLI, (B) RUSLI Throughput for Concurrent Inserts vs Model Size, and (C) RUSLI Throughput for concurrent inserts vs Model size with varying number of threads.

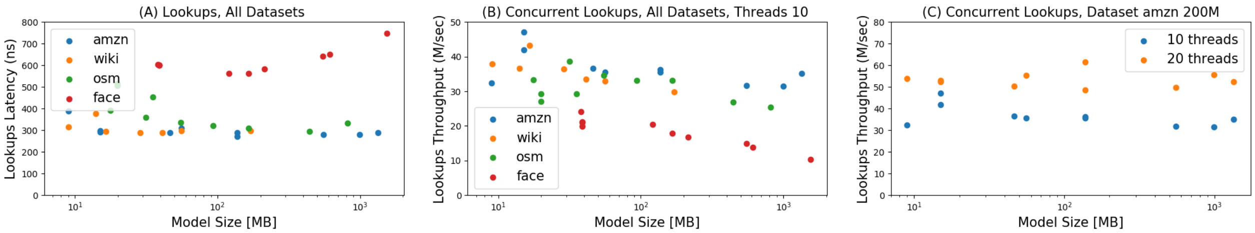

Figure 7: Performance of Lookup in RUSLI: (A) Single Lookup Latency with varying Model size B) RUSLI Throughput for concurrent lookups vs Model size C) RUSLI Throughput for concurrent lookups vs Model size for varying number of threads.

Table 2: Comparison of RadixSpline (RS) and RUSLI model size using SOSD [9] benchmark

Data Set	*radix bits*	*max error*	#Spline pts	RS (MB)	RUSLI (MB)	#max inserts
fb	20	2	31.5M	508.49	1981.37	94.5M
200M	23	35	1.89M	63.81	267.98	13.2M
keys	20	265	0.23M	7.98	44.33	2.3M
wiki	27	8	2.1M	506.86	669.70	10.5M
200M	23	40	140K	31.91	47.72	1M
keys	20	250	18K	3.99	6.61	170K
osm	27	7	6.8M	507	929	27M
200M	24	50	824K	63.09	152	5.7M
keys	21	365	109K	7.99	24.8	1.09M
amzn	25	2	27M	505	1784	82M
200M	22	45	410K	14.95	59	2.8M
keys	12	135	62K	1	9.57	558K

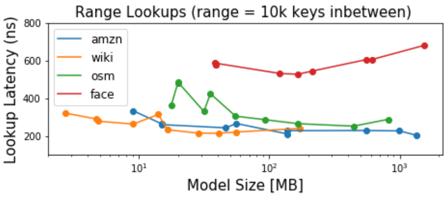

Figure 8: Lookup Time for Range Queries using RUSLI

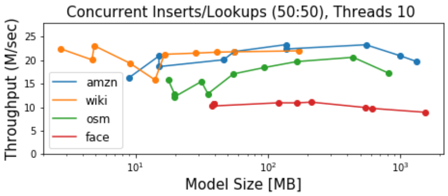

Figure 9: Throughput of RUSLI for mix of concurrent Inserts and Lookups

different configurations of *radix_bits* and *max_error*. RUSLI has a larger model size among the two for all configurations because of the fact that it also has space for future inserts. Column “max inserts” denotes the maximum number of inserts possible for a given model size. This is directly proportional to the number of spline-points and *log* of the configured *max_error*.

Insert Operations We observe in Fig 6, that, an insert operation latency is around 50ns and it increases with increase in the model size. As observed in Table 2, the increase in RUSLI model size is due to increase in number of spline-points. Thus, large number of spline-points implies more time for binary search within the spline-points array *SA*.

- *Insert vs Lookup Latency:* An interesting observation is that insert latency (50 ns Fig 6(A)) in RUSLI is lesser than lookup latency (270ns Fig 7 (A)). This appears to be counter-intuitive as, notionally, the inserts require lookups. However, in RUSLI, the inserts only require lookup for the correct spline-point (Insert Algorithm - step-1) and not the position of key. It does not involve the interpolation, binary-search, and scanning operations needed for lookup (steps 2, 3 of lookup algorithm 17). Same argument holds true for concurrent insets and concurrent lookups as well (Fig 6(B), Fig 7(B)).

- *Concurrent Inserts:* We have evaluated throughput of RUSLI for executing concurrent inserts using multiple threads. Figure6(C) shows that throughput of concurrent inserts increases with number of threads. However, throughput decreases with increase in the model size (Fig 6(B)) - this is due to increase in latency of single insert operation with increase in model size as discussed earlier.

Lookup Operations We have evaluated lookup operations soon after the build operation and after insertions as well. As discussed in Section 6, RUSLI lookup operation has two components to span lookup operation across newly inserted data as well. The first part is the same as experienced by RS. We have presented a lookup for data keys, which may have been inserted later ,i.e., after model creation, to bring in more complexity. As shown in Fig 7, because of RUSLI design, and parallel search across *OFA* and *DA*, lookup time remains the same as of part1 (as in RS [8]). Lookup time of RUSLI decreases with an increase in model size as exhibited by [12] as well for other learned indexes. Further, with an increase in the number of threads, lookup throughput increases due to parallel searches across *OFA* and *DA*.

Range Queries We performed range lookups (between k_l to k_u, such that there are $10k$ keys present in-between) using RUSLI. Fig 8 shows that latency incurred is around 300ns per range lookup, which is close to the single key lookup (Fig 7). We have implemented range lookups with just one key lookup (of k_l) followed by sequential scans of the *DA* and *OFA* (refer Section 6). Sequential scanning of arrays is fast and hence minimal extra latency.

Concurrent Insert and Lookup Operations Fig 9 shows the throughput achieved for mixed load of concurrent lookups and inserts (50:50 load split). We observed a mixed throughput of more than 25 Million ops per second. The throughput does not change much with an increase in model size. This is because decrease in throughput for concurrent inserts (Figure 6) is balanced out by increase in throughput due to concurrent lookups (Figure 7).

8 CONCLUSIONS AND FUTURE WORK

Learned index is an example of the use of AI for improving system performance. A learned index learns a model on the data distribution with its sorted storage locations in memory and uses it to access an element. Learned indexes are gaining popularity because of their ability to access data in a time duration comparable to traditional data structures such as B Trees with the benefit of less memory usage. Most of the available learned indexes retrain the model to support updates on the insertion of new data elements.

We have proposed, RUSLI, a real-time updatable learned index in this paper. The proposed algorithm is an extension of RadixSpline [8], which supports only batch inserts.

RUSLI employs a temporary memory to update the index in real-time in constant time (i.e., $O(1)$). To keep the lookup time unaffected by the inserts, RUSLI bounds the size of temporary memory to be $O(log(N/N_{sp}))$, where N_{sp} is the number of spline-points learned on the data set of size 'N'. RUSLI employs multi-threading to increase the throughput for concurrent insert and lookup operations.

We have evaluated the performance of RUSLI for lookup, range queries, and insert operations using SOSD [9] benchmark. We have shown RUSLI can look up a key in 270ns, range query in 300ns, and insert operations takes only 50 ns. We have shown that RUSLI can support 40 million concurrent insert and lookup operations.

We plan to extend RUSLI to support updation of the learned index for delete operations as well. Further, both inserts and deletes operations need to be supported in the context of multiple dimensions as well using ideas discussed in Flood [14] and Tsunami [4]. We will need to implement model size optimizations, especially for the multidimensional index.

REFERENCES

[1] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou (Steve)Li, Olston Christopher, and Andy Ly. 2021. Learned Indexes for a Google-scale Disk-based Database. *ML for Systems Workshop, NeuroIPS* (2021).

[2] Abdullah Al-Mamun, Hao Wu, and Walid G Aref. 2020. A Tutorial on Learned Multi-dimensional Indexes. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. 1–4.

[3] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[4] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.

[5] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1162–1175.

[6] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.

[7] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[8] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[9] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons kemper, Tim Kraska, and Thomas Neumann. 2021. SOSD: A Benchmark for Learned Indexes. *VLDB* (2021).

[10] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[11] Zichuan Lin, Tianqi Zhao, Guangwen Yang, and Lintao Zhang. 2018. Episodic Memory Deep Q-Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 2433–2439.

[12] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.

[13] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2789–2792.

[14] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[15] Thomas Neumann and Sebastian Michel. 2008. Smooth interpolating histograms with error guarantees. In *British National Conference on Databases*. Springer, 126–138.

[16] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. *arXiv preprint arXiv:2008.10349* (2020).

[17] Rekha Singhal, Dheeraj Chahal, Shruti Kunde, Mayank Mishra, and Manoj Nambiar. 2020. A Vision on Accelerating Enterprise IT System 2.0. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. 1–9.