OPTIMIZING RADIXSPLINE WITH THE POWER OF K-CHOICES

Evangelos Tsimpouris
ID: 1574035
Database Group
Eindhoven University of Technology
e.tsimpouris@student.tue.nl

November 21, 2023

ABSTRACT

Indexes are one of the most essential tools that DBAs utilize to improve the performance of analytics and transactional workloads. Advancements in the field led to the introduction of learned indexes. Learned index structures use machine learning techniques to model the cumulative distribution function (CDF) which maps the keys to their relevant positions. This work aims to optimize one of the SOTA learned index structures, namely Radix Spline [7] with the power of k-choices (an extension of power of 2-choices [10]). We investigate how the combination of the $error$ term and k-choices influences the performance of the resulting index. This endeavor aims to provide a first optimization attempt which can later be extended to other SOTA indexes. Using a combinations of real-world and synthetic datasets, we show that the optimized Radix spline index can effectively reduce the memory footprint and the lookup performance of its original counterpart.

.

1 Introduction

In Database Management System (DBMS), indexing is a technique using special data structures to quickly retrieve records from a database file. Generic index structures include B-Trees, B+ Trees and Bitmap Indexing [4], [13], [5]. These structures require to maintain records for the whole dataset and as such they are considered dataset-agnostic. This is particularly the case with 'traditional' indices, like B-Trees and B+ Trees, which end up scaling linearly to the table size. As is typically the case to have many indices on a single table, their size can grow significantly. Therefore, splitting your table to more than one indexes is one of the usual solutions in this case. This however, comes at the cost of additional operations and storage space to maintain the index data structure [9]. While in most cases a traditional index structure will work and perform adequately, in larger datasets this is not the case. Finally, with the introduction of Big Data, the need to create index structures for larger datasets becomes more and more essential.

Recent research developments led to the replacement of generic index structures with more sophisticated learned index models, which mainly incorporate Machine Learning techniques (RMIs [8], PGM [6] , and RS [7]) . These models aim to learn and store the underlying distribution of the given dataset instead of building an index from a dataset-agnostic algorithm. Recent research showed they exhibit superior performance in contrast with traditional index structures [11]. The overall performance gains of the learned index structures (contrary to the traditional ones) can be mainly attributed to the reduced size of the model and number of instructions. Cache misses and branch misses also play an important role on their performance. Additionally, most of these models are tuned by a fixed $error$, which can also affect the lookup performance of the model.

This work aims to optimize one of the learned index structures, namely Radix Spline [7] with the power of k-choices (an extension of power of 2-choices [10]). Power of k-choices here will act here as an optimization layer, which tries to increase the performance of query search while decreasing the overall size of the index. During creation of the index the purpose is to choose which index fits better for a key/value pair to be added. This will result in k-indexes which will have a decreased memory footprint (as a whole) as well as faster lookup performance. Radix

Spline is chosen here to test the power of k-choices as it is one of the current state of the art learned index and its properties exhibit the potential to maximize memory/ performance gains from the incorporation of k-choices.

This paper will explore how the k-indexes framework affects space-complexity and lookup performance in contrast with the single index counterpart. Results from both synthetic and real datasets are given on both metrics. Since the number of indexes k is a tunable parameter I also derive and present a formula that allows fine-tuning the value of k by estimating the performance of a k-index without having to benchmark the resulting structure each time.

2 Motivation and Preliminaries

2.1 power of k-choices

Here, a general representation of how k-choices work follows. Here k-choices act as an algorithm that aims to make a best decision on every insert request during creation of the index. The new index structure comprises of k-indexes built on the same dataset. In every insert request the algorithm will first calculate the impact of the insert for all the k-indexes. After that, the insert operation will only happen to one of the indexes, namely the one which is less influenced by the insert operation. In [10] the results indicated improved performance for Bloom filters while using the same basic approach Bloom filters employ. In this article we aim to achieve a similar behaviour in our results.

2.2 Radix spline

As previously stated, this article will focus on Radix spline to test the power of k-choices. First, we explain how RadixSpline creates an index by its own, and subsequently we present RadixSpline with the added k-choices layer. During build-time Radix spline creates spline points, which mainly represent the approximate underlying distribution. Radix spline only stores the spline points and uses a sophisticated search mechanism to retrieve the rest of the points.

Radix spline aims to store points (which are key/value pairs) which will end up approximating cumulative distribution function, or CDF of their dataset. These points are called spline points. And the set R where these points are stored is called the spline. Since this algorithm is also error bounded, all points can be retrieved via a query by interpolating the closest to the query spline points.

In Radix spline, the creation of the spline is done via the algorithm GreedySplineCorridor [12], explained in Figure 1. Every time a new point is inserted, it either fits the corridor created by the previous (fitted) points and the original spline point or a new spline point is created. The process repeats itself until all points are checked. The resulting model size is therefore equal to the number of spline points created. The idea here is to split the creation of the spline points between k-indexes.

2.3 Radix spline with k-choices

The process describing the creation of spline points exhibits the greedy property that we exactly aim to optimize via the power of k-choices. Every point now can check k-corridors (instead of one) to see if it fits. This way it is presented that k-indexes will result in creating less spline points in total than 1-index alone. This directly reduces the size of the model and affects its performance.

After the k-indexes are constructed, the algorithm described in [7] is used to retrieve a lookup key. Radix Spline uses a radix table to pinpoint the 2 closest to the key spline points. This is done by first acquiring the prefix of the lookup key to get the relative radix bits and then perform binary search on the resulting range to find the pair of spline points. They are then used to interpolate the estimated location on y-axis of the lookup key. Then the acutal location cannot be more than $\pm error$ from this estimation. Apart from the k-indexes construction the rest of the Radix Spline algorithm is left unchanged. However, the k-indexes will in total have less spline points for the radix table to search than the single index couterpart. To be able to maximize the improvement $error$ and k parameters need to be tuned correctly. Moreover, if the resulting k-indexes fit inside cache (because of their reduced size) the index stays closer to the CPU and therefore lookup performance is improved.

GreedySplineCorridor
Input: a spline S, $|S| = n$ and an error corridor size ϵ
Output: a spline connecting $S[1], S[n]$ through the corridor
$B = S[1], R =< B >$ // $S[1]$ is the first base point
$U = S[2] + \epsilon, L = S[2] - \epsilon$ // error corridor bounds
for $i = 3$ **to** n
  $C = S[i]$
  if $\overline{BC}$ is left of $\overline{BU}$ or right of $\overline{BL}$
    $B = S[i-1], R = R \circ < B >$
    $U = C + \epsilon, L = C - \epsilon$
  else
    $U' = C + \epsilon, L' = C - \epsilon$
    if $\overline{BU}$ is left of $\overline{BU'}$
      $U = U'$
    if $\overline{BL}$ is right of $\overline{BL'}$
      $L = L'$
$R = R \circ < S[n] >$
return R

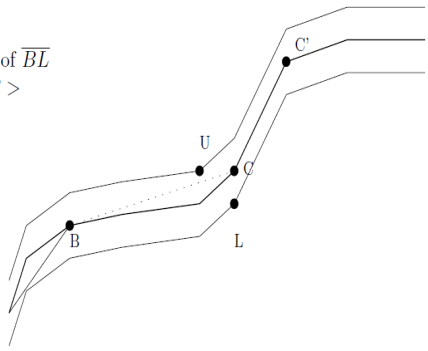


Fig. 1. Greedy Spline Approximation with a Given Error Corridor

Figure 1: Code of approximate spline creation given a fixed error [12].

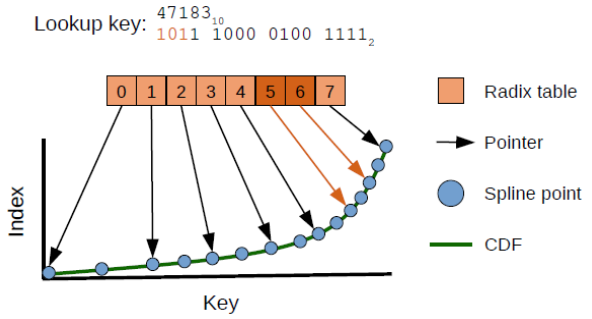


Figure 2: Illustration of Radix Spline search

Subsequently, we show a case where the initial radix-spline algorithm will create an inefficient index. In Figure 3 a dataset created by 3 uniform distributions is presented. As shown by subfigure (b), GreedySplineCorridor will create a spline point for every 3 keys leading to a very large index (for 30 million, 10 million spline points are created). This motivates the research to investigate if k-choices can disentagle these distributions and create one index for every distribution present in a dataset. In this case, 3-choices should solve the issue of repeated spline points creation. In general, GreedySplineCorridor with the property of k-choices should learn k-subdistribution present in a dataset.

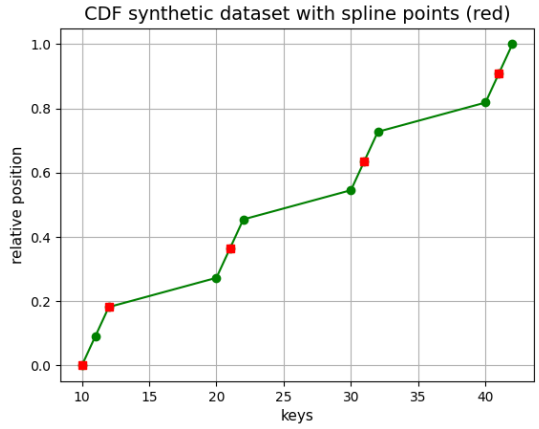


Figure 3: Illustration of spline point creation.

3 Radix spline with the power of k-choices

This part will explain in more detail, how spline points are created and distributed across indexes. In a general view, k-indexes are constructed by maintaining k-Radix splines. Every time a key is inserted, we perform a search to determine the optimal index for its insertion. The process repeats itself by adding the keys to the respective index. After the k-index structure is constructed we explain how queries are performed and explain how the query lookup performance is affected.

3.1 Construction of k indices

The keys are represented in x-axis and their respective in-memory location in y-axis. Figure 4 taken from [11] visually represents this. It is noted that in-memory locations can be normalized between 0 and 1 and the actual locations recreated from the first memory location since the keys are stored sequentially.

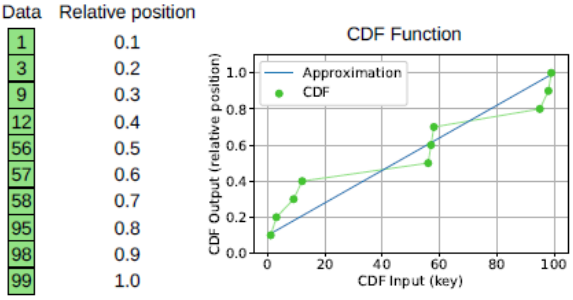
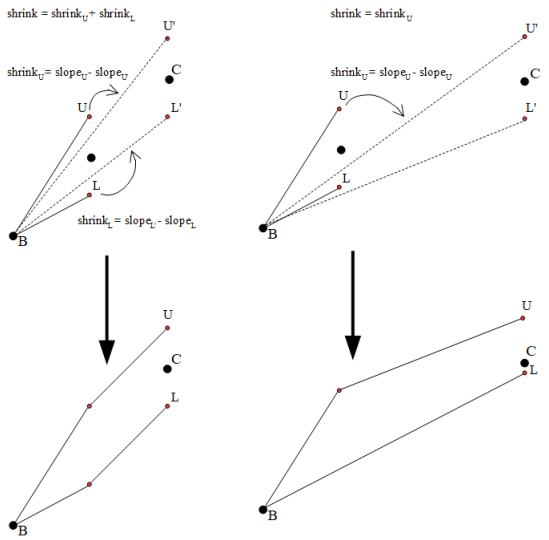


Figure 4: Keys distribution [11].

$$
\begin{aligned}
shrink_U &= |slope_U - slope_{U'}| \\
shrink_L &= |slope_{L'} - slope_L| \\
shrink &= shrink_U + shrink_L
\end{aligned} \tag{1}
$$



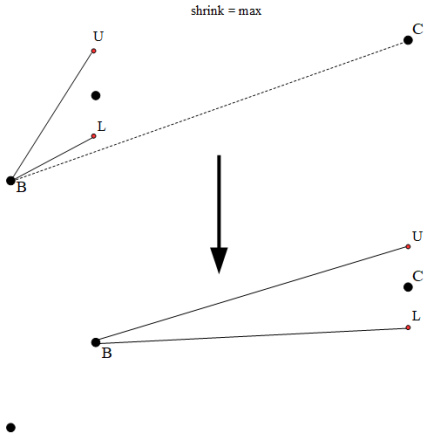


Figure 5: Illustration of calculating the shrinkage after a new key (C) addition.

In Figure 5 we present however 2 different cases (left and right), where the key indeed fits the cone but at the next iteration the new upper and lower limits (U', L') will be different. Between these cases the index which results in a larger cone is chosen (left). With keeping cones large enough, we ensure that more keys will fit without the need to create a new spline point (and therefore increase the index's size). Initially in GreedySplineCoridor U and L are points with a distance of $\pm error$ from the initial key C. However, these limits are updated and actually tend to get closer as more keys are added. (This way GreedySplineCoridor retains the $error$ bound to be able to accurately retrieve the locations of queried keys.)

To be able to distinguish these cases we calculate a-priori at every key for all indexes the potential shrinkage of the corridor. This is done by calculating for both upper and lower limits the difference in slopes (Equation (1)). The resulting metric which we call $shrink$ adds the absolute values of the slope differences. In case a key does not fit the corridor $shrink$ is set to max. The key will be added at the index with the minimum $shrink$.

Initially the pesudocode for the index construction is presented. The first $2 \cdot k$ keys are put to the k-indexes to initialize the process. After this every key is used to calculate each index's $shrink$ and it is put to the index with the $minimum\ shrink$.

for $key\ in\ keys$ **do**
  for $i \leftarrow 1$ to k **do**
    $shrink[i] \leftarrow CalculateShrink(index[i])$
  end for
  $Addkey(key, argmin(shrink))$
end for

The aforementioned code summarizes how keys are distributed and indexes are maintained at each iteration. All the keys that end up in an index form the index's key partition.

To shed some light to our optimization, some more in-depth analysis is needed to clarify the function of $CalculateShrink$. In Figure 5 we show an example of using k = 3 indexes to store spline points. In 5 the new point to be added is C. A cone is created between the upper and lower limits (U, L) of the previous key and the previous spline point (B). The cone is then extended to the location of (C). Clearly C fits the cone in first two cases in Figure 5 but not in the last. In the latter case a new spline point will be created at the location of the previous key (B is moved to the previous key), and a new cone between B and C is created. In this case Radix Spline will store 2 spline points. In Figure 5 Radix Spline will store 1 spline point. Therefore choosing the index which will store 2 spline points is avoided. The resulting corridor in Figure 1 will implicitely be created from the upper and lower limits (U, L) of each iteration.

3.2 k-indexes key lookup

In Section 2 the Radix Spline lookup algorithm is described. In the case of k-indexes, this algorithm is performed k times (or until the queried key is found). Initially it might seem that repeated lookups might cause a performance drop in query times. However, other critical factors such as key partitioning and index size can instead lead to a reduction in times.

The partitioning of keys between indexes does play a role in lookup performance. For example, in the case of 2-indexes, if 90% of keys end up in the first index, then 90% of the times the lookup scheme will avoid repeated search since it will find the key in the first index (if the key exists).

Additionally, as discussed in Section 2, during lookup, the radix bits can return an arbitrary search range of spline points. This range is implicitly affected by the index size. For example a smaller index can lead to narrower search ranges than a bigger one. This can lead to faster binary search of the 2 closest to the key spline points.

The $error$ term also plays a significant role here as well. Even in the initial algorithm large $error$ terms can slow down the lookup part. Even more, repeated searches of large $error$ terms can lead to an even larger slow down.

Also, caching can often change the penalty for a random access. Smaller indexes can therefore benefit from a more optimal use of cache. The caching influence however is difficult to exactly be measured.

The total estimated lookup latency for an error term of $error$ and k choices can be modeled by the following expression (Equation 2).

$$search\ range = \frac{splinePoints_j}{radixbits}$$
$$error = e$$
$$key\ partition_i = \frac{keys_i}{totalkeys}$$

$$lookup_k = c\sum_{i=1}^{k}\sum_{j=1}^{i} key\ partition_i(log_2(search\ range) + log_2(e)) \quad (2)$$

Equation (2) serves the purpose of estimating the lookup latency for a choice of k indexes, during their build time. Optimal k is determined by taking the index the minimum $lookup$ was calculated at. Otherwise the resulting k-indexes need to be benchmarked over a number of keys. Benchmarking though can be a costly, time-consuming procedure for larger datasets. Build time however is much faster. Therefore, Equation (2) provides a reasonable estimate of the final k-indexes performance, based on which optimal k can be chosen.

All parameters of Equation 2 can be determined after building the index (that is without performing any query yet). Binary search is assumed both between the search range of a spline point and the area that needs to be searched (bounded by $error$). For simplicity the term $\frac{splinePoints_j}{radixbits}$ expresses the average search range of a spline point, because the actual range can be arbitrary. The parameter $\frac{keys_i}{totalkeys}$ expresses the partitioning of the keys. This term is only use full if the keys we are searching are unique, otherwise this term is set to 1, as the index has to perform k-searches for a non-unique key. Lastly, setting c to a constant value implies that all random memory accesses have a constant penalty but caching can often change the penalty for a random access. This Equation will provide an estimation of the average key lookup time for the provided parameters (k,$error$). Eg. 300ns if $error$ = 70 and k = 3.

In conclusion, the Equation 2 implies that if k-indexes are able to distinguish small subdistributions without increasing the $error$ term too much, lookup time is decreased. In Section 5 it will become apparent that this is not so uncommon.

4 Datasets

To measure the impact of using k-indexes, we introduce a synthetic dataset (previously mentioned in Section 2) with an interesting property and 2 real datasets.

4.1 Synthetic dataset

The first dataset is an artificially created set of keys. The CDF of the dataset exhibits a property where keys continuously switch between 3 lines. To create this distribution three "close" keys with the same distance between them are inserted in the index. The same procedure repeats itself at a constant offset from the previous key. The offset between the steps is some times larger than the distance between each triplet of keys.

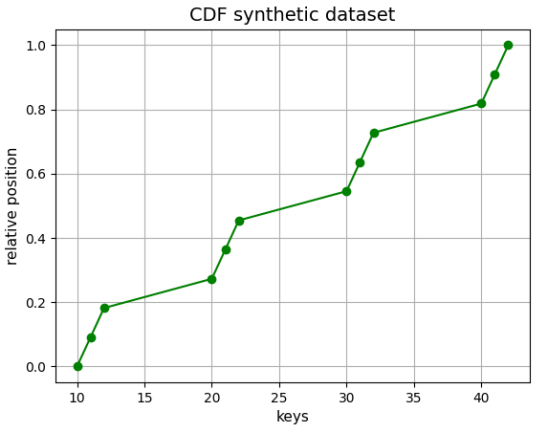
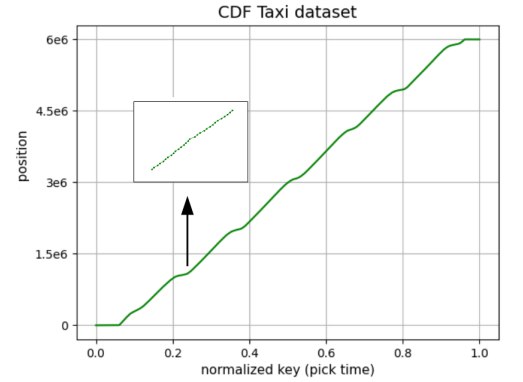


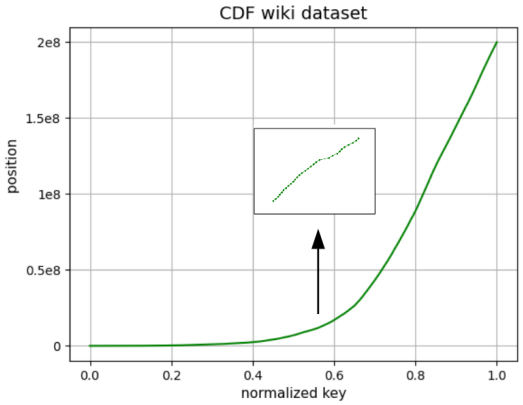
Figure 6: Distribution plot for synthetic dataset. Note that the key pattern is the same for arbitary number of keys. Therefore, the relative position is shown here.

4.2 Real datasets

Two real datasets are also used to measure the impact of k-choices in Radix Spline. Namely, NYC Taxi pick-up timestamps [1] and timestamps of edits from Wikipedia [2]. Their respective distributions are shown in Figure 7. Wiki contains 200M key/value pairs of 32-bit integers, while taxi contains 6M key/value pairs of 64-bit integers.



(a)



(b)

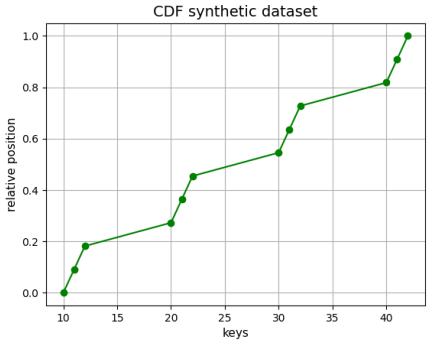
Figure 7: Distribution plots for real datasets.

5 Results

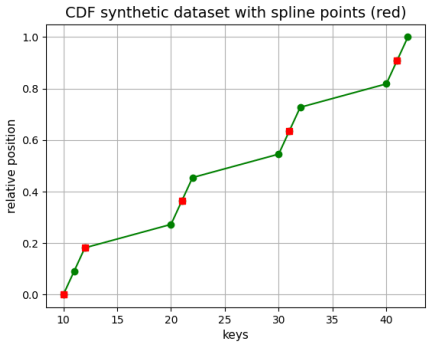
The goal of our experiments is to investigate whether or not k-choices provided significant speedup of query execution and at the same time explore their impact on the size of the index. We conducted all experiments on a single server with an Intel Core i7-4980 Processor CPU (2.2GHz, 4 cores, 6MB L3 cache) and 16GB RAM and all index and table data was held in memory. We provide a repository for the current implementation written in C [3].

5.1 Synthetic dataset

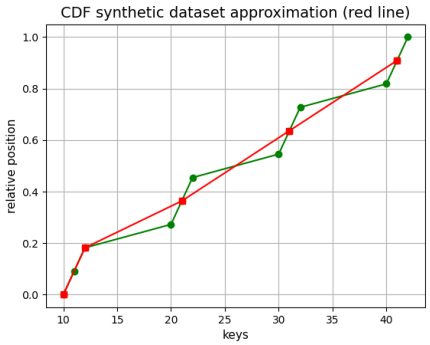
In the case of synthetic dataset producing 30 million keys, the 1 choice index stores 10 million spline points creating an index of size roughly 160 MB. In the same dataset but with 3 indexes, only 6 spline points are created (since every other third point belongs to a single line in Figure 8). The lookup performance is then dropped from 300 ns (1-index) to 60 ns (3-indexes) as probably everything is stored in cache in the latter case.



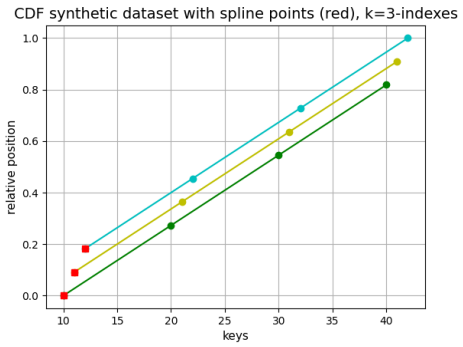
(a)



(b)



(c)



(d)

Figure 8: Illustration of spline point creation and approximation for 1 index (a,b,c) and for 3 indexes (d).

5.2 Real datasets

First, we present results for different real-world datasets, created by keeping the $error$ term fixed.

dataset	k	error	size (MB)	avg lookup (ns)
wiki	1	18	7.71	290
wiki	2	18	7.14	261
wiki	3	18	6.39	248
wiki	**4**	18	**5.72**	**229**
wiki	5	18	5.32	255
wiki	6	18	5.16	268

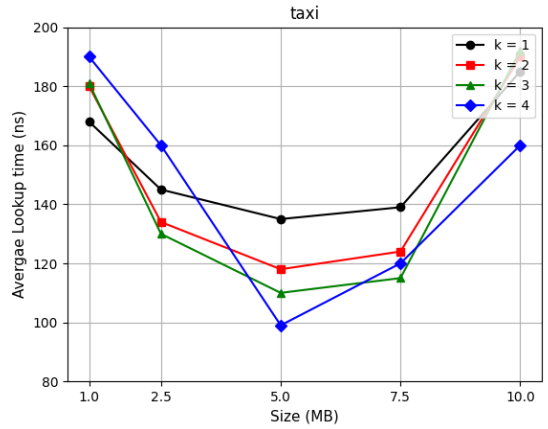
dataset	k	error	size (MB)	avg lookup (ns)
wiki	1	70	2.23	322
wiki	2	70	2.18	290
wiki	**3**	70	**2.06**	**245**
wiki	4	70	1.94	333
wiki	5	70	1.86	344
wiki	6	70	1.76	370

dataset	k	error	size (MB)	avg lookup (ns)
taxi	1	4	9.24	182
taxi	2	4	7.38	131
taxi	3	4	5.89	110
taxi	**4**	4	**5.06**	**99**
taxi	5	4	4.58	104
taxi	6	4	4.21	106

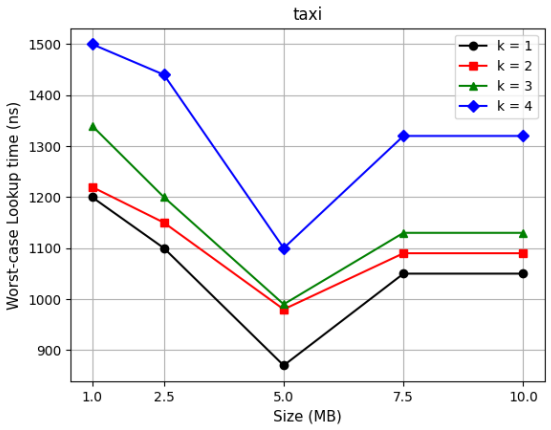
dataset	k	error	size (MB)	avg lookup (ns)
taxi	1	12	2.04	155
taxi	**2**	12	**1.88**	**151**
taxi	3	12	1.56	162
taxi	4	12	1.34	182
taxi	5	12	1.19	199
taxi	6	12	1.08	201

For the same $error$ term, k-choices produce increasingly smaller index size (by summing all k-indexes sizes) as the k term increases. Results show that k-choices outperform the single index counterpart on average lookup time as well.

Additionally, we exhibit the lookup and worst-case performance of the k-indexes at the same size (created by tweaking the error term for their resulting size to be roughly the same). Experimental results for different number of k are shown in Figures 10 and 9. For each experiment 10% of the each dataset keys are selected randomly (without replacement) to calculate the average and worst-case lookup times. For the worst-case times the average of 1% worst times are exhibited.

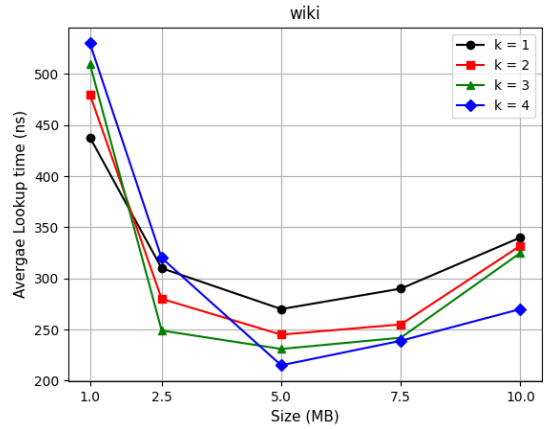


(a)

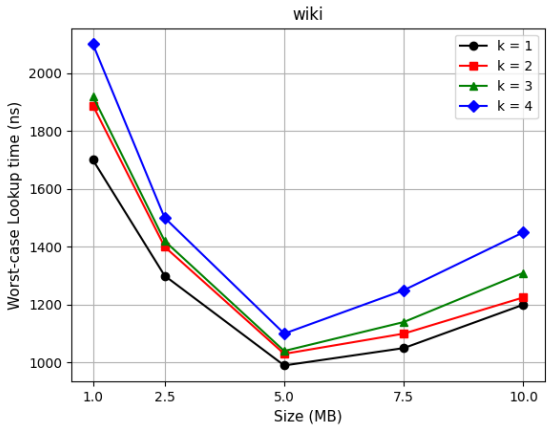


(b)

Figure 9: Illustration of taxi average and worst-case lookup results.



(a)



(b)

Figure 10: Illustration of wiki average and worst-case lookup results.

There are two key takeaways from the results. Firstly, for the same index size the k-choices outperform the single index counterpart on average lookup time as long as the resulting multi-index isn't too small (< 1 Megabyte). Best results were achieved when the full multi-index was able to fit in cache memory (Figures 10 and 9). Worst case lookup was always worse for the k-indexes, but was able to still provide competitive performance. This was to be expected though, since for a worst case scenario a k-index has to iterate all indexes to answer the query.

Secondly, for the same $error$ term, an increased k always produced a smaller index (as a sum of all k-indexes). This exhibits a clear compression property of k-choices. In certain cases (taxi dataset, error = 4) the k-choices were able to compress the same index as much as 45% of its original size. Furthermore, k-choices produced decreased average lookup latencies at all times.

6 Conclusion

In conclusion, optimizing learned indexes with k-choices can yield either lower memory footprint, better lookup performance or both. Equation (2) provides an approximation of the optimal k, however the $error$ term has to be tuned from the user. A main drawback of k-choices in learned indexes is that the implementation has to be model specific. In example, for an algorithm different than Radix spline a k-choices layer will look different. Nevertheless, the increase in performance make it worthwhile for either existing or future learned index structures to investigate similar multi-choice optimizations.

References

[1] `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

[2] `https://dataverse.harvard.edu/api/access/datafile/:persistentId?persistentId=doi:10.7910/DVN/JGVF9A/SVN8PI`.

[3] `https://github.com/vtsimpouris/RS-k_choices`.

[4] R. Bayer. The universal b-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997.

[5] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 355–366, 1998.

[6] P. Ferragina and G. Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[7] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Radixspline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020.

[8] 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, 2018.

[9] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1985.

[10] S. Lumetta and M. Mitzenmacher. Using the power of two choices to improve bloom filters. *Internet Mathematics*, 4(1):17–33, 2007.

[11] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *arXiv preprint arXiv:2006.12804*, 2020.

[12] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In *British National Conference on Databases*, pages 126–138. Springer, 2008.

[13] V. Srinivasan and M. J. Carey. Performance of b+ tree concurrency control algorithms. *The VLDB Journal*, 2(4):361–406, 1993.