SOSD: A Benchmark for Learned Indexes

Andreas Kipf[*]
TUM
kipf@in.tum.de

Ryan Marcus[*]
MIT CSAIL
ryanmarcus@mit.edu

Alexander van Renen[*]
TUM
renen@in.tum.de

Mihail Stoian
TUM
stoian@in.tum.de

Alfons Kemper
TUM
kemper@in.tum.de

Tim Kraska
MIT CSAIL
kraska@mit.edu

Thomas Neumann
TUM
neumann@in.tum.de

Abstract

A groundswell of recent work has focused on improving data management systems with learned components. Specifically, work on learned index structures has proposed replacing traditional index structures, such as B-trees, with learned models. Given the decades of research committed to improving index structures, there is significant skepticism about whether learned indexes actually outperform state-of-the-art implementations of traditional structures on real-world data.

To answer this question, we propose a new benchmarking framework that comes with a variety of real-world datasets and baseline implementations to compare against. We also show preliminary results for selected index structures, and find that learned models indeed often outperform state-of-the-art implementations, and are therefore a promising direction for future research.

1 Introduction

There has been a recent surge in proposals to replace traditional database system components, such as query optimizers and index structures, with learned counterparts [12, 13, 16, 17]. In particular, learned indexes [14] raised a lot of attention in the database community. Learned indexes replace structures such as B-trees with learned models that can accelerate lookups by predicting the position of a sought key in a sorted array (i.e., approximating the underlying cumulative distribution function, or CDF).

However, [14] lacked an open-source implementation and thus left many in the community skeptical that learned models could outperform optimized in-memory data structures [11, 15]. Thus, we introduce the **S**earch **O**n **S**orted **D**ata Benchmark (SOSD), a framework that allows researchers to compare their new (learned) index structures on both synthetic and real-world datasets. The benchmark is provided as open-source code [4] and comes with diverse datasets and highly-optimized baseline implementations. To the best of our knowledge, our implementation is the first performant and publicly available implementation of the Recursive Model Index (RMI) proposed in [14].[1]

This extended abstract additionally presents preliminary results from an experimental study using SOSD. We show end-to-end latency measurements and explain the results using performance counters (e.g., cache misses, branch mispredictions) for several index structures, including a novel spline-based approach which is trained bottom-up (RMIs are trained top-down). We find that learned index structures can indeed outperform traditional index structures in many scenarios. Finally, we explore the implications of our preliminary findings for practitioners.

[*]equal contribution

[1]Note, this is a novel implementation, not the implementation used in [14].

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

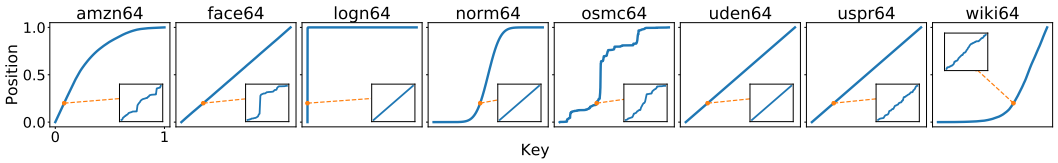


Figure 1: CDFs of datasets

2 Techniques

At a high level, the techniques studied in this work (cf. Table 1) can be categorized into on-the-fly algorithms, auxiliary indexes, approaches that approximate the cumulative distribution function (CDF) of the data (i.e., learned structures), and traditional index structures. In the context of this work, these techniques are used as in-memory secondary indexes that associate each *key* with a 64-bit *tuple identifier* (TID). When queried, each index structure must find all keys (unsigned integers) that qualify under the query predicate, and their associated TIDs.

Table 1: Categorization of techniques

category	technique	layout
on-the-fly	BinarySearch (BS)	array
	InterpolationSearch (IS)	array
	TIP	array
aux. index	RadixBinarySearch (RBS)	array
approx. CDF	RadixSpline (RS)	array
	RMI	array
index	ART	custom
	B-tree	custom
	FAST	custom

As representatives of **on-the-fly** algorithms, we studied binary search (BS), interpolation search (IS), and the recently proposed three point interpolation search (TIP) [20]. These algorithms operate directly on a sorted array, do not build an index, and do not need to examine the data ahead of query time. Binary search takes $O(\log n)$ steps to find the sought key, and can cause as many cache misses. In contrast, interpolation search "guesses" the position of the key by interpolating between the current min and max keys. For dense integer data such as sequential primary keys, this strategy works extremely well and might, in the best case, only incur a single cache miss [10]. For highly skewed data, on the other hand, it can degrade to a linear scan. TIP, an improved interpolation search algorithm, addresses this problem by using linear fractions for interpolation [20].

Similar to on-the-fly algorithms, **auxiliary indexes** operate on the sorted array, but they additionally build small and thus cache-efficient auxiliary structures. RadixBinarySearch (RBS) pre-scans the data to populate a flat *radix structure*: an array mapping fixed-length key prefixes to the first occurrence of that prefix in the sorted array. To process a lookup, RBS determines bounds on the key's position using the radix structure. RBS then preforms a binary search in this much smaller range.

CDF approximation algorithms directly estimate a lookup key's position using pre-trained / fitted models, and can thus be considered learned index structures. The RMI builds (a typically 2-level) tree of models (e.g., linear, linear spline, log linear) to approximate the CDF, and is constructed top-down as described in [14]. In contrast, and similar to [9], RadixSpline (RS) is built bottom-up, by fitting a linear spline to the CDF [18], and indexes the resulting spline segments in a radix structure. At lookup time, RS locates the corresponding spline segment using the radix structure and performs a linear interpolation between the two spline points.

We also include traditional **index structures** as baselines: ART [15] represents radix trees, B-tree is a popular B+-tree implementation [5] (STX, v0.9), and FAST [11] is a cache-optimized binary search tree. Notably, all of these structures reorganize the data into a specialized structure.

3 Search on Sorted Data Benchmark

SOSD is a new benchmarking framework that allows researchers to compare in-memory search algorithms on sorted data. It is provided as C++ open source code [4] that incurs little overhead (8 instructions and 1 cache miss per lookup), comes with diverse synthetic and real-world datasets, and provides efficient baseline implementations.

Table 2: Lookup latencies in nanoseconds per lookup. Top approach in each row is green. Yellow indicates 2-3x degraded performance. Red indicates > 3x degraded performance.

	ART	B-tree	BS	FAST	IS	RBS	RMI	RS	TIP
amzn32	n/a	529	773	**244**	4604	325	264	275	731
face32	**187**	524	771	229	1285	312	274	386	964
logn32	n/a	522	765	294	n/a	471	**97.0**	105	744
norm32	191	522	771	229	10257	355	**71.7**	**70.9**	884
uden32	102	521	771	228	**39.8**	333	54.2	64.2	176
uspr32	n/a	524	771	230	469	301	**153**	200	400
size overhead	47%	16%	0%	123%	0%	< 1%	3%	< 1%	0%
amzn64	n/a	601	804	n/a	4736	387	266	**288**	759
face64	391	592	784	n/a	1893	**337**	**334**	461	1232
logn64	309	597	784	n/a	n/a	753	179	**120**	454
norm64	266	592	785	n/a	10510	405	**71.5**	**70.5**	862
osmc64	n/a	599	785	n/a	95076	492	**402**	437	7186
uden64	112	592	784	n/a	**43.4**	344	54.3	53.9	193
uspr64	287	591	785	n/a	449	313	**169**	214	428
wiki64	n/a	608	802	n/a	7846	364	**222**	**218**	1019
size overhead	25%	16%	0%	n/a	0%	< 1%	3%	< 1%	0%

Datasets. SOSD currently includes eight different datasets. Each dataset consists of 200 million 64-bit unsigned integers (keys) with very few duplicates (if at all): `amzn` represents book sale popularity data [1], `face` is an upsampled version of a Facebook user ID dataset [20], `logn` and `norm` are lognormal (0, 2) and normal distributions respectively, `osmc` is uniformly sampled OpenStreetMap locations [19] represented as Google S2 CellIds [3], `uden` is dense integers, `uspr` is uniformly distributed sparse integers, and `wiki` is Wikipedia article edit timestamps [6]. Figure 1 shows the CDFs of these datasets. In addition, there are 32-bit versions of all datasets (except `osmc` and `wiki`) with similar CDFs. We use different parameters, (0, 1), for `logn` in the 32-bit case to reduce the number of duplicates. When loading the datasets, we generate 64-bit TIDs and store them alongside the keys in row format, although we plan to add support for columnar formats as well.

Queries. We perform 10 million equality lookups on a given dataset. Lookup keys are uniformly chosen from the set of keys. Note that we disallow hashing implementations (e.g., hash tables) since we want to support range-based lookups (e.g., lower bound searches). We ensure that lookup keys have at most 100 matches to limit the impact of result materialization. Lookups are performed one-at-a-time in a single thread. Our framework verifies each query result for correctness with low overhead (two instructions and a cache miss) and allows for out-of-order execution of lookups.

Results. To encourage reproducibility, we use the AWS machine type `c5.4xlarge` for benchmarking. Using the Intel Memory Latency Checker [2], we measured a DRAM latency (LLC miss) of 90 ns. Table 2 shows the lookup performance (in nanoseconds per lookup) for each technique as well as their average size overheads with respect to the data array (the 32-bit array is stored in packed format i.e., no padding). ART performs very well for the 32-bit datasets and even outperforms all others for `face32`. Note that ART does not support duplicate keys without modification. B-tree, BinarySearch, and FAST (which only supports 32-bit keys) are hardly affected by the data distribution. InterpolationSearch produces very low numbers for the dense data (`uden32` and `uden64`) but is heavily affected by skew. RadixBinarySearch shows consistent improvements over BinarySearch. The CDF approximators RMI and RadixSpline both have very low lookup latencies, highlighting the benefit learned index structures receive from fitting a model to the data distribution. TIP improves over the textbook interpolation search in most cases, but is affected by the many steps in `osmc64` (cf. Figure 1).

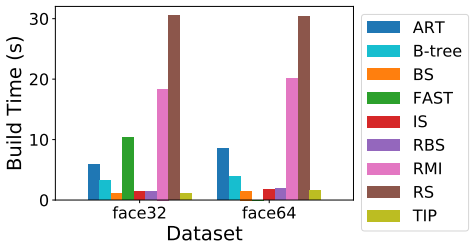
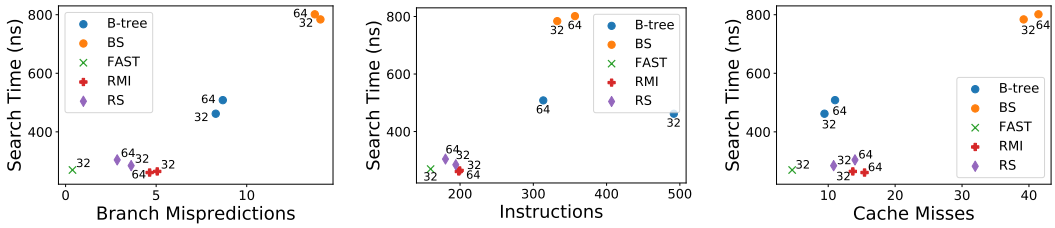


Figure 2: Build times



(a) Branch mispredictions (b) Instructions executed (c) LL cache misses

Figure 3: Performance counter breakdown of amzn32 and amzn64 data

Figure 2 shows a preliminary analysis of build times. Note that we have not optimized build phases: for example, ART could achieve lower build times with bulk loading, and RMI could be improved by building models on samples. BS, IS, and TIP do not spend any time on build, except for copying the data vector due to our framework design (an approximately two second cost, paid once). The same holds for RBS, which only needs to perform a scan over the data to build its auxiliary structure. RS experiences the highest build times for fitting a fine-grained linear spline to the CDF of the data. However, even without optimizations, the build times of the CDF approximators may be acceptable for many applications.

Figure 3 shows performance counters (per lookup) for selected algorithms on `amzn32` and `amzn64`, generated automatically by SOSD in `perf` mode. Here, we use an Intel Xeon E5-2680 v4 CPU, since AWS does not expose hardware counters. FAST incurs less than five cache misses per lookup, while RMI and RS incur around ten. However, RMI and RS have similar or superior search times, indicating that analyzing cache misses alone is not sufficient to understand search time. Similarly, neither instructions executed nor branch mispredictions fully explain deviations in search time. SOSD automatically analyzes branch mispredictions and instructions executed as well.

4 Takeaways

We have seen that the CDF approximators (RMI, RS) can outperform our baseline implementations. For the tested datasets, we have shown that simple models are sufficient for efficient learned indexes. We next explore potential decision points for choosing between different index structures.

For the trivial case of uniform dense integers, non-surprisingly IS is the clear winner. Otherwise, the optimal search strategy depends on whether a user can afford to manually tune and fit a CDF model, as both RMI and RS require dataset-specific tuning. The upside of these models is that they consume very little space compared to our studied index structures. If users can afford this tuning step, we recommend using either RMI or RS.[2] However, it might be possible to significantly improve the training times of RMI/RS and to do it entirely automatically, without user-intervention. Until then, if users cannot afford the training time, we recommend using ART or FAST for 32-bit keys and ART or RBS for 64-bit keys. Unlike ART, RBS can operator directly on a sorted array (i.e., RBS has low space overhead compared to ART).

A current drawback of learned indexes is the lack of support for efficient updates, an arguably important feature for index structures. However, several recent works have shown progress towards addressing updates [7, 9, 21]. Moreover, it should be pointed out that many of the benchmarked methods (e.g., RBS and FAST) also do not support efficient updates.

We plan to extend SOSD with multi-threaded/vectorized lookups, updates, and integration into real database systems to support measuring the impact of these structures on SQL queries. We will additionally compare with recent variants of learned index structures [7, 8]. We hope SOSD can serve as a platform for testing multi-dimensional index structures, accelerators such as GPUs and FPGAs, and just-in-time compiled custom-tailored data structures for specific queries or datasets.

[2] RMI and RS both represent learned models – the primary difference is that RMI is built top-down (starting with the root model), while RS is built bottom-up (starting by fitting linear splines to the data, though other models could be used as well).

References

[1] Amazon sales rank data for print and kindle books. https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.

[2] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[3] S2 Geometry. https://s2geometry.io/.

[4] Search on Sorted Data Benchmark. https://github.com/learnedsystems/SOSD.

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

[6] Wikimedia Downloads. http://dumps.wikimedia.org.

[7] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. B. Lomet. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.

[8] P. Ferragina and G. Vinciguerra. The PGM-index: A multicriteria, compressed and learned approach to data indexing. *arXiv:1910.06169 [cs]*, Oct. 2019.

[9] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, 2019. ACM.

[10] G. Graefe. B-tree indexes, interpolation search, and skew. In *Workshop on Data Management on New Hardware, DaMoN 2006, Chicago, Illinois, USA, June 25, 2006*, page 5, 2006.

[11] 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 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350, 2010.

[12] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[13] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A learned database system. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[14] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504, 2018.

[15] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49, 2013.

[16] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[17] R. Marcus, O. Papaemmanouil, S. Semenova, and S. Garber. NashDB: An Economic Approach to Fragmentation, Replication and Provisioning for Elastic Databases. In *37th ACM Special Interest Group in Data Management*, SIGMOD '18, Houston, TX, 2018.

[18] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases, BNCOD 25, Cardiff, UK, July 7-10, 2008. Proceedings*, pages 126–138, 2008.

[19] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *PVLDB*, 11(11):1661–1673, 2018.

[20] P. V. Sandt, Y. Chronis, and J. M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 36–53, 2019.

[21] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 1223–1240, 2019.