A Comparative Study of Compressed, Learned, and Traditional Indexing Methods for Integer Data

Lorenzo Bellomo[1]
Department of Computer Science, University of Pisa, Italy

Giuseppe Cianci
Sadas S.R.L., Casalnuovo di Napoli, Italy

Luca de Rosa
Sadas S.R.L., Casalnuovo di Napoli, Italy

Paolo Ferragina
Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
Department of Computer Science, University of Pisa, Italy

Mattia Odorisio
Department of Computer Science, University of Pisa, Italy

Abstract

The rapid evolution of learned data structures has revolutionized database indexing, particularly for sorted integer datasets. While learned indexes excel in static scenarios due to their low memory footprint, reduced storage requirements, and fast lookup times, benchmarks like SOSD and TLI have largely overlooked compressed indexes and SIMD-based implementations of traditional indexes. This paper addresses this gap by introducing a comprehensive benchmarking framework that (i) evaluates traditional, learned, and compressed indexes across 12 datasets (real and synthetic) of varying types and sizes; (ii) integrates state-of-the-art SIMD-enhanced B-Tree variants; and (iii) measures critical performance metrics such as memory usage, construction time, and lookup efficiency. Our findings reveal that while learned indexes minimize memory usage, a feature useful when internal memory constraints are mandatory, SIMD-enhanced B-Trees consistently achieve superior lookup times with comparable extra space. On the other hand, compressed indexes like LA-vector and EliasFano provide very effective compression of the indexed data with slower access speeds (2x–3x). Another contribution of this paper is a publicly available benchmarking framework (composed of code and datasets) that makes our experiments reproducible and extensible to other indexes and datasets.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Algorithm design techniques; Theory of computation → Data compression; Information systems → Database query processing

Keywords and phrases indexing data structures, compression, algorithm engineering, benchmark

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.5

Supplementary Material
Software: `https://github.com/LorenzoBellomo/SortedStaticIndexBenchmark`
archived at `swh:1:dir:c459351991c1a7df827164c4edd4d582a8caaf5e`

Funding *Lorenzo Bellomo*: This work was partially supported by the PON project on "For Fair Justice: Innovation and efficiency in Judicial Offices" (aka *Giustizia Agile*), PON Governance and Institutional Capacity 2014-20 project, Axis I, Specific Objective 1.4, 2022-23.
Paolo Ferragina: This work was partially supported by a 2024 SADAS grant; by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) through the project "SoBigData.it–Strengthening the Italian RI for Social Mining and Big Data

[1] Corresponding Author

Analytics" under Grant Prot. IR0000013–Avviso n. 3264 del 28/12/2021; by the spoke "FutureHPC and BigData" of the ICSC–Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing; by the Alfred P. Sloan Foundation with the grant #G-2025-25193 (`sloan.org`); and by the PON project on "For Fair Justice: Innovation and efficiency in Judicial Offices" (aka *Giustizia Agile*), PON Governance and Institutional Capacity 2014-20 project, Axis I, Specific Objective 1.4, 2022-23.

1 Introduction

The rapid development of learned data structures has significantly enhanced the performance of lookups in database systems, particularly for sorted integer datasets. These data structures have been extensively evaluated in recent years over static and dynamic scenarios through comprehensive benchmarks, such as Search on Sorted Datasets (SOSD) [21, 16] and the Testbed for Learned Indexes (TLI) [32], thus identifying the indexing solutions that perform best under some specific conditions.

In the static context, which is the one of interest in our paper, learned indexes have emerged as the dominant choice due to their low RAM footprint, reduced disk space requirements, and faster lookup times compared to traditional indexes. Despite this prominence, these benchmarks failed to consider other kinds of indexes, such as compressed indexes [8, 7, 19] and compressed+learned indexes [2, 1]. These other data structures are much more interesting because they are *"computation friendly"*, in that they not only minimize the storage space but also incorporate some specific succinct/compressed indexes that allow fast access to compressed data. Comparing these data structures against the ones already experimented in SOSD [21, 16] and TLI [32] is fundamental to offer a comprehensive comparison, and it is challenging due to the wealth of state-of-the-art technical solutions available. Notable contributions include traditional compressed indexes [26] and newer methodologies leveraging learned compressed indexes [11]. Additionally, known benchmarks often employ unoptimized baselines, which, for example, do not include some recent implementations of traditional indexes (i.e., B-Tree) with SIMD (Single Instruction Multiple Data) operations [29, 15, 17] that achieve exceptional lookup times with no extra space.

The present paper aims to build on these premises by experimentally exploring the performance of learned and compressed indexes more thoughtfully, thus offering new insights into their efficiency and applicability in various data contexts. To reach our goal, we designed a comprehensive benchmarking framework for traditional, learned, and compressed indexes working on static integer datasets that: (i) uses eight real datasets and four synthetic ones of different types (graphs, ratings, time series,...) and size (from 1 to 800 million elements); (ii) includes the implementation of a SIMD-BTree and a SIMD-Sampled-BTree based on [29], that achieve state-of-the-art performance on pointwise queries; (iii) provides scripts to run the experiments and obtain comprehensive reports of its results; (iv) extends the previously existing benchmarks (i.e., SOSD and TLI) in the static context by integrating compressed indexes and including the measurement of other useful metrics (i.e., index size, RAM footprint, build time, and missing items lookup times); and, last but not least, (v) it is extremely easy to be extended with new indexes and datasets.

Through this benchmarking framework, we shed some more light on the performance of learned, compressed, and traditional indexes. Indeed, (1) learned indexes are fast, very efficient in space occupancy, and the best in terms of memory footprint at building time; but (2) existing and our proposed SIMD-variants of BTrees achieve faster lookup times over all of the 12 tested (real and synthetic) datasets using none-to-minimal extra space; and (3) when

Table 1 Details on the real and synthetic datasets of integers (32 and 64 bits) used in the experiments. Values with the $\sim$ symbol are approximated for clarity.

Type	Dataset Name	Size	Average Gap	# of Duplicates
Real	CompanyNet	1M	$\sim 2\,801$	2
	Friendster	50M	~ 13	0
	Amzn	200M	~ 21	1
	Wiki	200M	~ 2	109 562 989
Synthetic	Normal	50M	~ 43	710 309
	Lognormal	50M	~ 42	919 544
	Exponential	50M	~ 43	1 009 869
	Zipf	50M	~ 43	6 311 198
64-bit	Amzn64	800M	$\sim 1.2 * 10^{10}$	0
	Facebook64	200M	$\sim 9.2 * 10^{10}$	0
	OSM-Cellids64	800M	$\sim 1.7 * 10^{10}$	0
	Wiki64	200M	~ 1	109 562 989

data compression is mandatory, the best-compressed indexes are LA-vector and EliasFano, but their lookup times are at least 3x slower than that of SIMD-BTrees. We can report in the main article only a few figures/plots that we consider most significant for our discussion because of space constraints, and thus we include a Supplementary Material part and a file named "plot_recap.pdf" in our public code-repository[2] to allow the interested reader finding all experimental figures and further comments.

The paper is organized as follows. Section 2 describes the 12 datasets on which we performed our experiments; Section 3 details our experimental design and settings; Section 4 describes the 15 indexes we tested; Section 5 presents the experimental results conducted on all the described indexes and datasets, and foresees future research directions.

2 Datasets

The experiments conducted on the tested indexes used some datasets that varied widely in terms of size, distribution, nature, and other characteristics. Specifically, we considered three types of datasets: synthetic datasets of 32-bit integers, with a size of 50 million elements; real datasets of 32-bit integers, with sizes up to 200 million elements; and real datasets of 64-bit integers, with sizes of 800 million elements for Amzn64 and OSM-Cellids64, and 200 million elements for Wiki64 and Facebook64. All the datasets are downloadable from our public code repository through proper scripts, and they are also available in Zenodo[3].

The choice of their size was dictated both by data availability and by the memory limitations of the machine on which the experiments were executed. In total, our experiments involved 12 datasets comprising roughly 2.6 billion integers. The detailed description of each dataset is in Section A.1 of the Supplementary Materials for space reasons, but a rough overview is provided in Table 1, and the following bulleted list:

[2] `https://github.com/LorenzoBellomo/SortedStaticIndexBenchmark`
[3] Link to all datasets: `https://zenodo.org/records/15240501`

Four 32-bit real datasets: Two of these datasets are taken from SOSD [21, 16] (i.e., Amzn and Wiki), while the other two are derived from graphs of social or corporate networks, respectively Friendster[4] and CompanyNet, the latter being proprietary to a Sadas client;

Four 32-bit synthetic datasets: All integers in these datasets are generated synthetically using standard routines. Specifically, three of these datasets used generators from the C++ standard library (i.e., normal, lognormal, and exponential), while the Zipf distribution was generated using a specific script [30];

Four 64-bit real datasets: All integers in these datasets are taken from SOSD [21, 16] (i.e., Amzn64, Wiki64, OSM-Cellids64, and Facebook64).

As a note, we point out that we have generated two additional datasets to run a preliminary experiment on performance scalability with the dataset size; specifically, we generated a subset of Amzn64 (containing the first 50 million items), and a larger 800 million integer normal-distribution dataset (using the same parameter as the Normal dataset, and still 32 bits unsigned integers).

3 Experimental design

The experimentation focused on static datasets, where we tested three types of indexes (traditional, learned, and compressed), described in detail in Section 4, and whose results are reported in the file "plot_recap.pdf" in our repository[5]. For each index, we measured the following metrics:

Disk Space Footprint: This is the space taken by each tested index, measured in MB[6]. The results are detailed in the plots of Figures 2, 3. We notice that, for a fair comparison with the compressed indexes, the disk space of the non-compressed ones (i.e., learned and traditional) includes the memory required to store the std::vector containing the data.

Construction Time: In some applications, it is crucial to have an index ready quickly so that it can be rebuilt fast after a few changes. We measured the time (in milliseconds) required to construct each index to evaluate this issue. The results are shown in Figure 1.

RAM Footprint: A fundamental metric for assessing the efficiency of an index is the peak internal memory required to construct it. In our experiments, we calculated the maximum and minimum memory peaks. We reported it as the "ratio" with the memory required by the std::vector, also reporting the dataset where these ratios were obtained.

Pointwise Query: This query evaluated the average time to search for a single element, present or not in the indexed dataset. The average was computed over 1 million distinct queries on each dataset. The sampled distinct queries are fixed for all indexes, and they are chosen as follows: the 1 million query items are sampled uniformly at random from the input, for the experiment on existing items; while they are generated uniformly at random (between the minimum and the maximum key) for the experiment on missing items. Depending on the tested index, the search returns one of the following results: (i) the index or value of the first element greater than or equal to the queried value (Next-GEQ), or a "safe" value indicating that no element greater than or equal to the query exists; (ii) an approximation (with error) of the position of the Next-GEQ in the vector, typical of learned indexes. In this second case, we ensure timing consistency by counting the cost of searching the exact position via the *lower_bound* function of the

[4] Friendster Social Network Dataset: `https://snap.stanford.edu/data/com-Friendster.html`
[5] `https://github.com/LorenzoBellomo/SortedStaticIndexBenchmark`
[6] Values in MB were rounded up to the nearest integer.

C++ standard library. Choosing to use *lower_bound* for the last-mile search has an impact on the final query results timings, but we decided to use it because of the following reasons: (i) the binary search is the default option for all the learned indexes that provide an approximate position; (ii) the implementation of the *lower_bound* function of the C++ standard library is compiler dependent, making it very hard to reliably benchmark; (iii) Marcus et al. [21, 16] analyzed the impact of the search algorithm in their work on SOSD, testing linear, binary, and interpolation search, and found that interpolation search and binary search obtain comparable results (difference of $\sim 2\%$), while linear search performed worse; (iv) the main point of PGM++ [20] is to improve the PGM-index by changing the search algorithm for all the PGM layers, obtaining results that, as seen in Section 5, are in line with those of PGM-index. For those reasons, we deemed this part of the benchmark too focused on the task of "measuring search time", rather than focusing on the indexes, so we decided to ignore it. The results of these experiments are shown in Figures 2, 3, and in Figure 4 of the Supplementary Materials.

Range Query: This benchmark evaluates the time required for each index to perform a random access on the indexed data and a scan of the subsequent x elements, with $x = 10, 100, 1K, 10K$. Time was averaged on 100 000 range queries. This experiment was run on four selected datasets (i.e., CompanyNet, Amzn, Wiki, and Facebook64) and is particularly useful for assessing the performance of compressed indexes and of the SIMD-BTree (which permutes the data). In fact, the other indexes rely on the std::vector, and thus their scan performance depends on this latter, so we refer to it in the plots. The plots are reported in Figure 5 in the Supplementary Materials and in the file "plot_recap.pdf" in our repository.

The final time performance of the construction, range, and pointwise queries was computed by repeating each experiment 10 times and then averaging the individual results.

4 Indexes (traditional, learned, and compressed)

Table 2 summarizes the key information about the tested indexes. Specifically, it shows the citations of relevant works for each index, the year of the last update to the corresponding library (to assess its "maintenance vitality"), the space required to represent the original data, the space needed to construct the indexing structures, and the support for the "Rank" operation (indicating whether it is directly available in the library or easily derivable).

The selection of the test indexes was based on the state of the art in this field [27, 18, 22, 12, 20, 31, 6, 2, 1, 13, 25, 24, 19] and the most relevant benchmarks found in the literature, as previously mentioned (that is, SOSD [21, 16] and TLI [32]). Regarding the state-of-the-art, three main families of indexes can be identified: (i) Traditional and/or Baseline Indexes, (ii) Learned Indexes, and (iii) Compressed Indexes.

It is important to note that while compressed indexes encapsulate both the index and the input data within a space smaller than a std::vector (hence, they could also be called *compressed self-indexes*), traditional and learned indexes occupy the space of the std::vector plus the extra space needed to store the index, although this extra space is often minimal. These solutions are particularly interesting in contexts where the input data vector is read-only and multiple indexes are required for different primary keys, or when the input data is large and stored in external memory while the (small) index is kept in internal memory to support fast searches.

In the next Sections, we describe briefly all the tested indexes. For a more detailed description, we refer the readers to Section A.2 of the Supplementary Materials.

Table 2 Information about all tested indexes. All compressed indexes and PLEX only support integers, while all the others also support floating-point numbers. All the licenses of the listed tools allow commercial use. The "Space" column specifies the space required for the index and the (input) data representation distinguishing among several approaches whose acronyms stand for: ED = Explicit Data (storage), CD = Compressed Data (storage), CSD = Customizable Sample of (indexed) Data, CS = Customizable Size, NCSD = Non-Customizable Sample of (indexed) Data, and PD = Permuted Data. The "Rank Supported" column specifies whether the rank of an element can be extracted from that index, or can be "derivable" through elementary operations, or is "approximate" and thus the programmer has to implement a final step based on binary search.

Type	Data Structure	Citation	Last Update	Space		Rank Supported
				Index	Data	
Index	CSS-Tree	[27]	2019	CSD	ED	YES/Derivable
	SIMD-BTree	**new**	2025	0	PD	YES
	SIMD-Sampled-BTree	**new**	2025	CSD	ED	YES
	FAST	[15]	2010	0	PD	YES
	Static-search-tree	[17]	2025	0	NCSD	NO
	RMI	[18, 22]	2020	CS	ED	YES/Approximate
	PGM-index	[12]	2024	CSD	ED	YES/Approximate
	PGM++	[20]	2024	CSD	ED	YES/Approximate
	PLEX	[31]	2021	CSD	ED	YES/Derivable
	ALEX	[6]	2024	NCSD	ED	NO
Compressed Index	LA-vector	[2, 1]	2024	CSD	CD	YES
	δ-code-vector	[13]	2019	CSD	CD	YES/Derivable
	γ-code-vector	[13]	2019	CSD	CD	YES/Derivable
	EliasFano	[13]	2019	NCSD	CD	YES
	Roaring	[19]	2024	NCSD	CD	YES
	std::vector	–	–	0	ED	YES

4.1 Traditional, SIMD, and Baseline Indexes

This section describes the traditional indexes (CSS-Tree [14, 27]), the two SIMD trees found in the literature (Static-search-tree and FAST), and the baseline (std::vector) used in the benchmark, whereas the next section details our new proposed SIMD implementation of the classical B-Tree

std::vector: The simplest data structure commonly used to store ordered integer keys. To search for an element, we use the `lower_bound` function (binary search).

CSS-Tree: Implementation[7] of the "Cache Sensitive Search Tree" described in [27].

FAST: Implementation[8] of the SIMD-BTree described in [15].

[7] CSS-Tree implementation: `https://github.com/gvinciguerra/CSS-tree/`
[8] FAST: `https://github.com/curtis-sun/TLI/blob/main/competitors/fast/src/fast.h`

Static-search-tree: Extremely recent, highly optimized RUST implementation[9] [17] of the static search tree described in [29]. However, this index was not integrated directly into our benchmark because: (i) it runs only on 32 bits; (ii) its maximum allowed value is below $2^{32} - 1$; and (iii) some optimizations are lost while compiling the functions as a static library. For these reasons, we decided to run the author's benchmark and dedicate a paragraph in Section 5 to comment on its performance.

Our Implementation of a B-Tree for Modern Architectures

The experience gained from and the results obtained with our benchmarks led us to design and experiment with two new indexes[10] that leverage SIMD operations.

To define indexes with SIMD operations, we will use the following terminology:

key: A single computational unit. In our case, one integer.

vector: The sequence of keys on which atomic SIMD operations are performed.

block: A group of keys stored contiguously in a B-Tree node, serving as the smallest reading unit of this data structure.

SIMD-BTree is our implementation of a Static B-Tree (a.k.a. S-Tree) as proposed in [29]. It is a pointer-free B-Tree [14], whose elements are placed in a cache-sensitive layout like the one of CSS-Tree. Its construction is obtained by generalizing the more popular Eytzinger layout [28], which is based on a binary tree, to the general case where the fanout of the tree is B (i.e., the size of a B-Tree node). Additionally, it uses SIMD primitives to traverse the tree efficiently. Since the data is permuted to "simulate" a layout resembling that of a B-Tree without explicitly realizing it, the additional space this index requires, beyond the input data (appropriately permuted), is zero. Note that even if the elements are permuted, the items can still be retrieved through their index (rank) in their original sorted order. The library we developed allows us to specify the SIMD extension to use (and consequently the size of the vector of integers to process "atomically" using this mode) and the number of vectors per block in the B-Tree. Our experimental results were obtained using AVX512 with one vector per block, which we found to be the fastest configuration. This is coherent with the expectations, indeed one such vector has the same size as the cache line. In our implementation we included several algorithm engineering choices suggested in [29], (i.e., blocks nodes are ensured to be aligned with the cache lines; huge pages[11] are set to reduce the overhead due to the translation of virtual addresses). The library supports all C++ numeric data types by selecting the appropriate intrinsic primitives at compile time (through extensive use of templates). As mentioned, the tree is stored as a vector of elements, where each node occupies a contiguous block of n_B elements, n_B being the fixed size of a node. To resolve a query, the query value is splatted (copied in all the positions) into a vector register. The algorithm then traverses the tree starting from the root (the first n_B elements of the data vector), loading the node in a vector register, and comparing all of its elements with the query value to determine the child index where the search has to continue. This process repeats until a leaf node is reached. The expert reader may have noticed the similarities in

[9] RUST S-Tree: `https://github.com/RagnarGrootKoerkamp/suffix-array-searching`
[10] SIMD-BTrees repository: `https://github.com/mattiaodorisio/s-tree`
[11] Debian Hugepages Wiki: `https://wiki.debian.org/Hugepages`

between this SIMD-BTree and FAST [15], the main differences reside in the in-node items ordering (sorted for the SIMD-BTree, recursively stored in breadth first layout in FAST), the data-type supported, and other engineering design choices[12].

SIMD-Sampled-BTree is derived by applying the previous data structure onto a subset of the input data extracted via regular sampling with a fixed step. The library we developed allows for specifying, on top of the parameters of SIMD-BTree, an additional one that defines the sampling ratio from the input data. Consequently, a Rank query is answered in two steps by first solving it over the (internal) SIMD-BTree built over the sampled data, and then by computing the direct rank (again via SIMD operations) on the identified block of the input. The advantages of this proposal are: (i) faster direct access to the elements, (ii) easy inclusion of satellite data, which can be left at the leaf level, (iii) further extensibility to the dynamic setting (although the current implementation does not address this case).

4.2 Learned Indexes

We have already introduced the *learned indexes* [18] as data structures that, through more or less sophisticated machine learning techniques, "learn" the data distribution to be indexed to improve search performance and space usage. These indexes differ on the machine learning (ML) models used to "learn" the input data distribution. It is evident that the more complex the ML model is, the more precise the training of the network is and, thus, its ability to find the position of the searched element. However, it is equally true that the space occupied by the model and its usage have become more expensive in terms of computational resources. Therefore, the design of learned indexes involves a sophisticated balance between the complexity of the ML model and its efficiency. Below, we describe five approaches to the design of learned data structures, selected among those that are the most significant from a theoretical perspective, the most efficient in practice, and that offer robust software libraries.

For completeness, it should be noted that some indexes that achieved the best results in certain experiments in SOSD [21, 16] and TLI [32] were excluded from our benchmark because they are limited to support boolean queries, and they cannot return the position of an element in the input dataset[13], or they do not allow indexing datasets with duplicate elements (i.e., LIPP [33]), which are typical in DBMS. The list of tested learned indexes is the following:

RMI (Recursive Model Index): The precursor of all learned indexes [18, 22]. In our experiments, two RMI models are proposed for each dataset: the "large" and the "compact" versions. The RMI optimizer[14] outputs a list of 10 parameter combinations with models of decreasing size. The largest RMI model is always the most efficient in query time but uses a fixed size of 403 MB, so it is named "large." To include a more compact RMI model that aligns most closely with the sizes of the other (traditional and learned) indexes in our benchmark, we added a version of size approximately 6 MB, so named "compact."

[12] FAST adds a padding equal to the last (max) value to build a complete tree, possibly wasting a huge amount of space; conversely, the SIMD-BTree allocates only the needed memory, but incurs in one branch to check the out-of-bounds condition. The more sophisticated layout of [17] allows one to avoid the branch, still limiting the amount of wasted memory to at most one extra page (2MB) per tree layer, thus resulting in an even better compromise.

[13] Note: ALEX also does not support the "Rank" operation, but it supports "Next-GEQ", which is a sufficient requirement for our benchmark.

[14] RUST RMI library: `https://github.com/learnedsystems/RMI`

PGM-index: A learned index [12] [15] based on *Piecewise Linear Approximation*, customized using the parameter ϵ that trades search time and space. The values used for ϵ in our experiments are 8, 32, and 128.

PGM++: A recent (2024) optimized variant[16] of the PGM-index [20] that addresses its search slowdown. We used the same values for ϵ as for the PGM-index.

ALEX: A learned index[17] based on the key-value store model [6] (see footnote 13).

PLEX: A learned index[18] based on the Compact Hist-Tree [5] and a radix-tree. As for the PGM-index, it uses a parameter ϵ (i.e., maximum prediction error, which provides a trade-off between build and lookup time) set in our experiments to the values 8, 32, and 128. PLEX is the only uncompressed index of this article that does not support floating point numbers (this is not a limitation for this benchmark, but it may be an important limiting issue for its wider adoption).

4.3 Compressed Indexes

The compressed indexes described in this section are of the *self-index* type, meaning they encapsulate both the data to be indexed (in compressed form) and the indexing data structure for searching them. It is important to note that many of these data structures are implemented in the SDSL library [13][19]. For completeness, a brief description of each is provided.

EliasFano: A compressed index [8, 9, 10] for non-descending integer sequences. We used the implementation provided by SDSL (sd_vector).

γ-code and δ-code vectors: They are compressed indexes [7] that store non-descending sequences of integers using the corresponding compression algorithms [10], plus some extra information that allows to jump over the compressed data (controlled by a "*density*" parameter, set to 16 and 32 in our experiments). The implementation used in our benchmark is part of the SDSL library (enc_vector), which does not provide the Next-GEQ function. To this end, we derived its implementation from the *Rank* and *Select* test library of the LA-vector data structure described below[20].

LA-vector: It is a compressed index [2, 1] based on Piecewise Linear Approximation, controlled by a parameter *bpc* which was set in our experiments to $6, 8, 10, 12$. Additionally, the library can compute the optimal value to minimize the space of the vector. We call this configuration "opt".

Roaring: It is a highly optimized compressed bitmap [19][21], based on an implementation that heavily utilizes SIMD operations to accelerate queries. Roaring bitmaps can be Memory Mapped natively through the library, but they only work with sets of non-duplicate integers (and thus they could not be applied to most of our datasets; see Table 1).

[15] The PGM-index: `https://github.com/gvinciguerra/PGM-index`
[16] PGM++: `https://github.com/qyliu-hkust/bench_search/`
[17] ALEX: `https://github.com/microsoft/ALEX`
[18] PLEX: `https://github.com/stoianmihail/PLEX`
[19] SDSL library: `https://github.com/simongog/sdsl-lite`
[20] `https://github.com/aboffa/Learned-Compressed-Rank-Select-TALG22.git`
[21] C implementation of Roaring: `https://github.com/RoaringBitmap/CRoaring`

5 Results

The server used for the experiments has 512 GB of RAM and an Intel(R) Xeon(R) Gold 6238R CPU at 2.20 GHz, with a total of 56 cores, running Ubuntu 22.04 LTS on an x86-64 architecture. The following precautions were taken to minimize interference with external factors and ensure a fair benchmarking environment: the CPU frequency was set to the nominal frequency, via the "cpupower" command, and "hyperthreading" was disabled.

One of the key contributions of our paper is that we packaged the benchmarking code[22] and datasets into a publicly available repository[23], with the goals of:

- making it extremely easy to run the benchmark and reproduce our experimental results through a set of bash scripts that aid in installing, downloading the sorted datasets, setting up all the RMI models, compiling, and finally running the experiments;
- allowing the easy generation of consistent and valuable plots/reports. We also provide a LaTeX file that generates a visually consistent report (like the file "plot_recap.pdf");
- allowing users to include additional indexes and datasets in the benchmark easily. Specifically, indexes can be added by just integrating it to the CMakeLists file, creating a simple interface file with some simple methods (e.g., build, next-geq), and registering the benchmarks to run.

This article will not explain all the details on running the benchmark because of space constraints, but they will be available in the repository's README file. Moreover, in this section, we report only a subset of the plots and tables (see Figures 1, 2, 3), since the complete set of results is provided in the Supplementary Materials at the end of this article and in the file "plot_recap.pdf" made available in our public repository (see footnote 23).

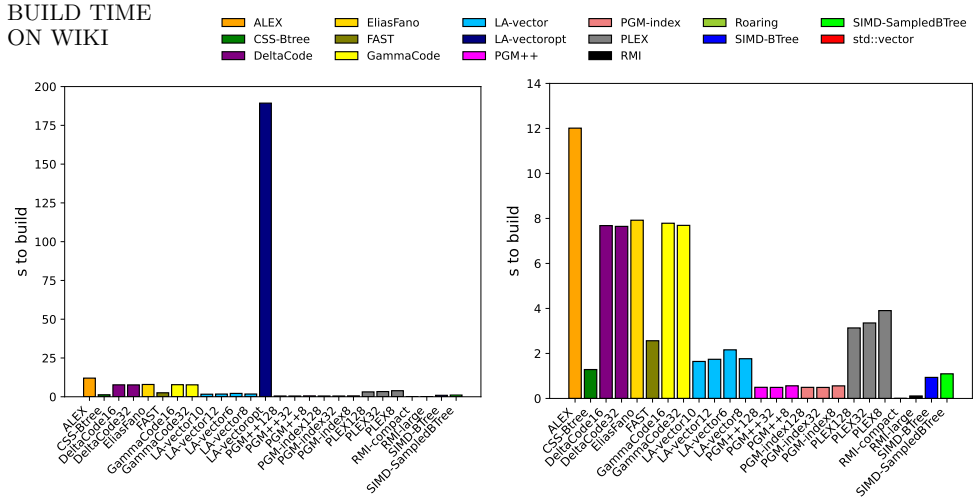
The following pages present the experiments conducted on all the indexes described in the previous Section 4. Some figures are missing for LA-vector because of errors on some datasets, for Roaring because of its limitation to run only on non-duplicate datasets, and for EliasFano because it fails to run on three datasets.

Let us now start commenting on the *construction time* using Wiki as a reference dataset (results tend to be consistent). Figure 1 shows that all indexes require negligible time to index 200M elements (around or less than 2 seconds), except for (in decreasing time cost): the "opt" version of LA-vector, ALEX, γ-code and δ-code vectors, EliasFano, FAST, and PLEX.

We also measured the *memory footprint* used during the construction of each index, obtaining the following results (expressed as a ratio to the one needed by std::vector):

- PLEX, PGM-index, PGM++, EliasFano: require a footprint that does not exceed three times that required by std::vector (specifically, 1x–3x).
- SIMD-BTree, SIMD-Sampled-BTree, γ-code and δ-code vectors, CSS-Tree: require a footprint not exceeding five times that of std::vector (specifically, 1x–5x).
- LA-vector uses a variable amount of memory depending on the configuration used. When "bpc" is fixed, the maximum RAM required is 15.5x, while the minimum is 2x (on the OSM-Cellids64 and Wiki64 datasets obtained with *bpc* set to 12 and 6 respectively). On the other hand, the process of obtaining the "opt" version of LA-vector is more memory-intensive, with a peak RAM consumption exceeding 100x on Facebook64.
- ALEX ranges from a minimum of 5.6x on Wiki64 to 10.7x on CompanyNet.

[22] Based on the library Google Benchmark: `https://github.com/google/benchmark`
[23] `https://github.com/LorenzoBellomo/SortedStaticIndexBenchmark`



The plots above report the time (in ms) required to construct the data structures (indexes) on the Wiki dataset. The left plot shows all indexes, while the right one shows only the indexes whose construction time does not surpass twice the average time.

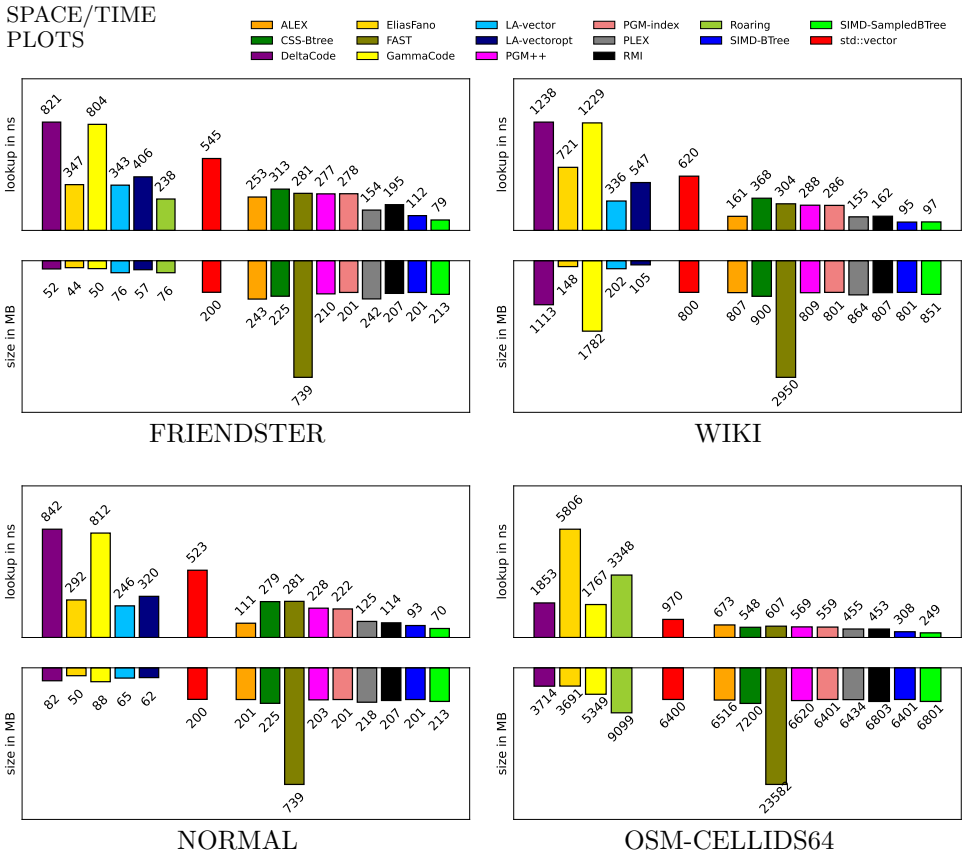
Figure 1 Index construction time on Wiki datasets.

- FAST has a RAM requirement that varies from a minimum of 3.4x on CompanyNet, to a maximum of 5.6x on Books64.
- Roaring is usually lighter but has a high peak on the Amzn64 dataset, needing 19.4x.

Let us now turn our attention to the time and space performance of the indexes on pointwise queries, referring the reader to the details in Figures 2 and 3 (and Figures 4 in the Supplementary Materials). In Figure 2, the red bar represents the time-space performance of std::vector, which can be seen as the reference being the space taken by the input data. The compressed indexes are on its left; to its right are the traditional indexes, the learned indexes, and our two SIMD-BTrees. On the other hand, Figure 3 provides a different visual representation of these results by using a two-dimensional time-space plot, highlighting the Pareto curve (more comments in Section 5.1).

We can observe that the SIMD-BTrees (blue and green bars) significantly outperform all the traditional indexes, and also obtain better time performances on pointwise queries than all learned indexes on all datasets, with only PLEX, RMI, and ALEX obtaining similar results in some experiments. In the case of compressed indexes, the situation is more unstable, as different datasets tend to show significantly different performance patterns. Nevertheless, the indexes that tend to obtain the best results on average are EliasFano (Pareto-optimal in 8 out of 12 datasets), and LA-vector (Pareto-optimal in 7 out of 12 datasets).

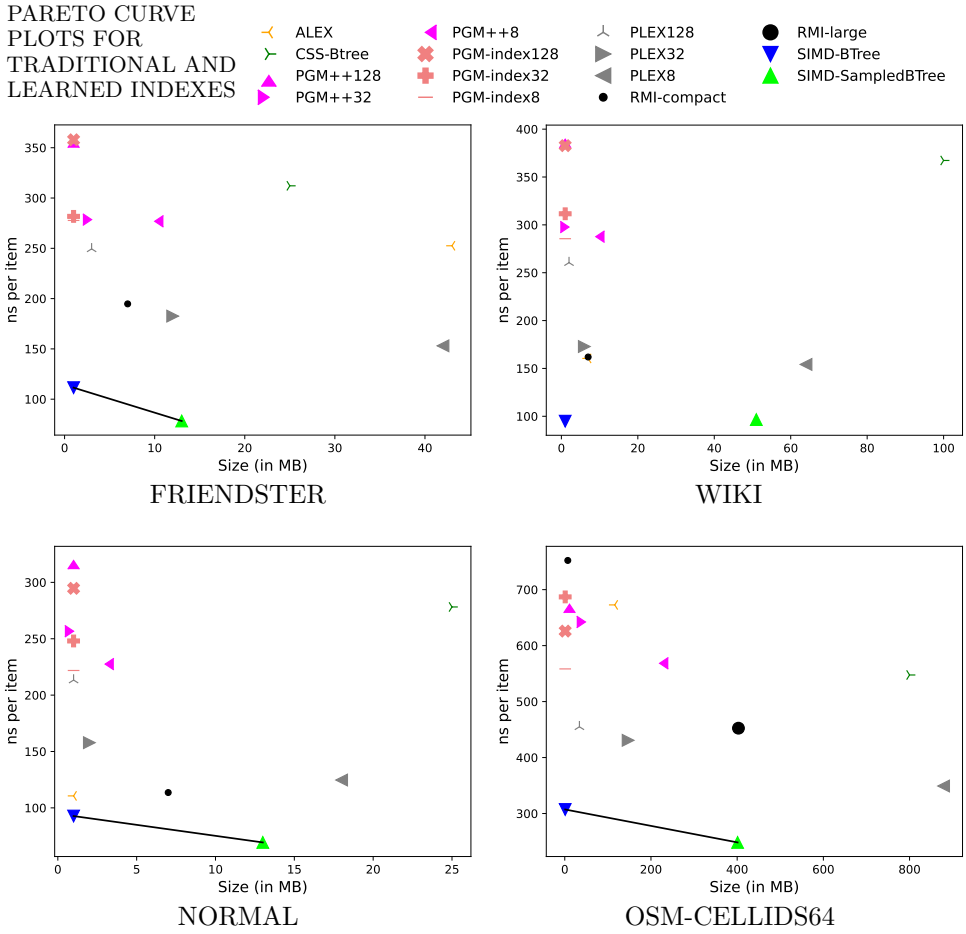
Table 3 shows a compact view of the best-performing indexes on each dataset. For a deeper analysis of the results, we refer the reader to Section A.3 of the Supplementary Materials, whereas all plots are available in the "plot_recap.pdf" file on our code repository. We highlight that the repository contains plots showing the time performance for increasing the dataset sizes (Amzn64 with 50M–800M items and Normal with 50M–800M items). These plots show that the performance of indexes tends to have a consistent slowdown (up to 2x) as the dataset size increases (up to 16x). As future work, we plan to further extend the dataset sizes.



The figures above show the results corresponding to the space occupied and elapsed time for pointwise queries on all tested indexes (traditional, learned, and compressed). For each dataset and index, the top part shows the average time (in ns) needed to make a query on existing items in the dataset, while the bottom part shows the required space in MB (where we added the space of the std::vector for traditional and learned indexes). The results show only the best-performing parameter configurations for each index (in the case of RMI, the best performer is the "compact" configuration).

Figure 2 Space-time plots for pointwise queries on 3 real datasets and 1 synthetic.

As previously mentioned in Section 4.1, the performance of the RUST Static-search-tree [17] is analyzed separately. We ran its benchmark, which performs several Next-GEQ queries on increasing-size datasets. We limit our analysis to: (i) comparing the average ns per query on datasets whose size is comparable to ours (50M, 200M, 800M); and (ii) serial queries without batching, that would substantially improve the throughput, but would not be comparable with all the other experiments. All the results are obtained on uniformly distributed datasets of 32-bit unsigned integers. The JSON file with the performance dump is in our public code repository. The measured times are: (i) on a dataset of roughly 50M items, Static-search-tree requires 83 ns per item on average, while our SIMD-BTree obtains roughly 111 ns per item; (ii) on a dataset of roughly 200M items, Static-search-tree requires 99 ns per item on average, while our SIMD-BTree obtains roughly 110 ns per item; (iii) on a dataset of roughly 800M items, Static-search-tree requires 177 ns per item on average, while our SIMD-BTree obtains roughly 192 ns per item. So the distance between the two implementations shrinks or becomes negligible as datasets grow in size.



The plots recap the experimental results about the occupied space/time needed for pointwise queries for traditional and learned indexes. For these datasets, for each index, and each tested configuration, we plot the extra space occupied (in MB) on the x-axis, and the time (in ns) per pointwise query on existing items in the datasets.
Additionally, a black line shows the Pareto frontier for traditional/learned indexes. The indexes that sit on top of the Pareto frontier offer the best space-time trade-off. Non-compressed indexes whose size exceeds one-fourth of the space required by std::vector were excluded for clarity. None of the excluded indexes were on the Pareto frontier.

Figure 3 Pareto Frontier: Traditional and Learned Indexes.

5.1 Take-home message

From this series of experiments (see Table 3 and Figure 3), we can conclude that there is no single "best performer." Instead, the "best" depends on several factors such as the data distribution, the size of the dataset, and the type of query to be supported.

Excluding the four datasets where significantly different performances were observed (i.e., Wiki, Wiki64, OSM-Cellids64, and Zipf), for all other datasets, **the best space-time trade-off is provided by: EliasFano and LA-vector, for compressed indexes; by RMI, PLEX, and ALEX, for learned indexes; and by the two variants SIMD-BTree and SIMD-Sampled-BTree, proposed by us for traditional indexes.** We point out that

Table 3 Summary of the most performing indexes for each tested dataset: "SIMD-BTrees" refers to both SIMD-BTree and SIMD-Sampled-BTree.

dataset		best-performers		
group	name	traditional indexes	learned indexes	compressed indexes
Real	CompanyNet	SIMD-BTrees	PLEX	EliasFano, LA-vector
	Friendster	SIMD-BTrees	PLEX, RMI	EliasFano, Roaring
	Amzn	SIMD-BTrees	ALEX	EliasFano
	Wiki	SIMD-BTrees	PLEX, ALEX	LA-vector
Synthetic	Normal	SIMD-BTrees	ALEX	EliasFano, LA-vector
	Lognormal	SIMD-BTrees	RMI	EliasFano, LA-vector
	Exponential	SIMD-BTrees	ALEX	EliasFano, LA-vector
	Zipf	SIMD-BTrees	RMI, PLEX	LA-vector
64 bit	Amzn64	SIMD-BTrees	RMI	EliasFano
	Facebook64	SIMD-BTrees	RMI, PLEX	Roaring
	OSM-Cellids64	SIMD-BTrees	RMI, PLEX	γ-code and δ-code vectors
	Wiki64	SIMD-BTrees	ALEX, PLEX	LA-vector

RMI is "compiled" for each dataset, which does not allow using it on dynamically varying datasets; and PGM-index gets the smallest storage occupancy among learned indexes, at the expense of query time.

A deeper analysis of Figure 3 reveals that it is possible to identify specific "best performers" for each dataset (which always include SIMD-BTree and SIMD-Sampled-BTree). In particular, SIMD-BTree is the best choice among all traditional and learned indexes, being the only index always on the Pareto frontier. Among compressed indexes, however, there is no single choice, as this depends on the dataset being indexed.

To facilitate discussion over each specific dataset, we identify some *"clusters"* of them with the same "best performers."

Uniform group: When the data distribution is approximately uniform (i.e., CompanyNet, Amzn, Amzn64, Friendster, Facebook64), EliasFano and Roaring are consistently the best-compressed indexes, both for the space occupied – up to 4x smaller than std::vector– and for the time required for pointwise queries – though still at least 4x slower than our two SIMD-BTrees, but about 1x–2x faster than std::vector.

Normal-Lognormal-Exponential group: On these datasets, the best-compressed indexes are LA-vector and the two implementations of EliasFano, which achieve query times that are 3–4x slower than those of SIMD-Sampled-BTree. On these datasets, RMI and ALEX are the only learned indexes that achieve performance nearly on par with the SIMD-BTrees.

Duplicate group: In this group of datasets (i.e., Wiki, Wiki64, and Zipf), the best-learned indexes are PLEX and ALEX, while the best-compressed index is undoubtedly LA-vector: it obtains great compression rates (4–15x less space than std::vector) and query times that are consistently better than those of std::vector.

OSM-Cellids64: The performance of the indexes on this dataset is part of its cluster, probably because of the extremely particular dataset distribution. On this dataset, RMI and PLEX are the best-learned indexes; whereas we notice that this is the only dataset where, among the compressed indexes, γ-code and δ-code vectors are Pareto-optimal.

We conclude the "take-home message" by noting that our benchmarks assume the data is already sorted and indexes can "copy" it to work on. In other cases, we observe that:

- If the data is already sorted and cannot be copied ("read-only" mode), the use of compressed indexes or indexes that use an internal copy of the data (i.e., SIMD-BTree, CSS-Tree, SIMD-Sampled-BTree, FAST, and ALEX) is pointless, as this would increase space consumption. In this scenario, learned indexes are the best choice, offering efficient time performance and a very low memory footprint.
- If the data is not sorted (also known as secondary key indexing), it is necessary to maintain their sorted permutation via pointer arrays or succinct encoding [23], thus requiring the consideration of this extra space in the disk-space footprint of the indexes.

5.2 Conclusions and Future Work

From this extensive range of experiments, it is evident that one of the characteristics that made *learned indexes* appealing, as mentioned in the original RMI paper [18][24], and related to their "significant benefits" over classical indexes, is challenged by SIMD version of BTrees (including the one of [17]). These prove to be extremely fast and compact when it is possible to modify the "layout" of the input data and leverage the power of SIMD processing. Indeed, the SIMD-BTree resides on the Pareto curve in every dataset, demonstrating its extreme effectiveness in both time and space efficiency. The Pareto plots also highlight that, in the case of compressed indexes, the situation is more "varying," but the indexes that consistently come closest to the Pareto curve are EliasFano and LA-vector. The only aspect where learned indexes still outperform all other indexes (even SIMD-based ones) is their *memory footprint* at building time, which is indeed negligible and thus extremely advantageous if the data to be indexed is read-only, sorted, and stored on memory-constrained clients.

It should be noted that this wide range of trade-offs is very interesting and useful for designing *optimizers* capable of selecting *on-the-fly* the indexes that best adapt to the distribution of input data, user requirements, and possible "constraints" imposed by underlying applications. In this context, the speed of index construction, already emphasized earlier for learned indexes, is crucial to enable their adoption *on-the-fly* following their selection. This aligns perfectly with current trends in scientific research [4, 3, 34], where the focus is shifting from developing (individual) data structures (learned and/or compressed) to designing "single-/bi-/multi-criteria optimizers" capable of selecting, from a family of potential data structures, the one that achieves the best trade-off among a series of computational criteria: typically space, time, and energy.

Our intention, moving forward, is to leverage these recent developments and our experimental results to design and implement an optimizer operating on a family of indexes whose possible configurations take into account the time efficiency of SIMD-BTree, the compactness of learned indexes, and the computation-friendly compression achievable with the encoding of EliasFano, Roaring, or LA-vector. Additionally, we aim to investigate the use of SIMD instructions and batching queries in the context of *learned indexes*, further accelerating query times while maintaining a low memory footprint and drawing inspiration from the work of Zhang et al. [34]. As a final note, we plan to provide a more comprehensive experiment to measure the impact of dataset size on the time-performance of queries in compressed/learned/classic indexes.

[24] "...we have demonstrated that machine-learned models have the potential to provide significant benefits over state-of-the-art indexes..."

References

1. Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A "learned" approach to quicken and compress rank/select dictionaries. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59, 2021. doi:10.1137/1.9781611976472.4.
2. Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Trans. Algorithms*, 18(3), 2022. doi:10.1145/3524060.
3. Subarna Chatterjee, Mark F. Pekala, Lev Kruglyak, and Stratos Idreos. Limousine: Blending learned and classical indexes to self-design larger-than-memory cloud storage engines. *Proceedings ACM Manag. Data*, 2(1), March 2024. doi:10.1145/3639302.
4. Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. Airindex: Versatile index tuning through data and storage. *Proceedings ACM Manag. Data*, 1(3), November 2023. doi:10.1145/3617308.
5. Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. In *Conference on Innovative Data Systems Research*, 2021. URL: https://api.semanticscholar.org/CorpusID:231400989.
6. Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 969–984. ACM, 2020. doi:10.1145/3318464.3389711.
7. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.
8. Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974. doi:10.1145/321812.321820.
9. R.M. Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971. URL: https://books.google.it/books?id=07DeGwAACAAJ.
10. Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
11. Paolo Ferragina and Giorgio Vinciguerra. *Learned Data Structures*, pages 5–41. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-43883-8_2.
12. Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings VLDB Endow.*, 13(8):1162–1175, 2020. doi:10.14778/3389133.3389135.
13. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.
14. Goetz Graefe. More modern b-tree techniques. *Foundations and Trends® in Databases*, 13(3):169–249, 2024. doi:10.1561/1900000070.
15. Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, 2010. doi:10.1145/1807167.1807206.
16. Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.
17. Ragnar Groot Koerkamp. Static search trees: 40x faster than binary search. https://curiouscoding.nl/posts/static-search-tree/, 2024.
18. Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 489–504. ACM, 2018. doi:10.1145/3183713.3196909.

19 Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018. doi:10.1002/spe.2560.

20 Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. Why are learned indexes so effective but sometimes ineffective? *arXiv*, 2024. doi:10.48550/arXiv.2410.00846.

21 Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proceedings VLDB Endow.*, 14(1):1–13, 2020. doi:10.14778/3421424.3421425.

22 Ryan Marcus, Emily Zhang, and Tim Kraska. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2789–2792. ACM, 2020. doi:10.1145/3318464.3384706.

23 Gonzalo Navarro. *Compact data structures: a practical approach*. Cambridge University Press, 2016.

24 Giuseppe Ottaviano, Nicola Tonellotto, and Rossano Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM, pages 47–56. ACM, 2015. doi:10.1145/2684822.2685297.

25 Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 273–282, 2014. doi:10.1145/2600428.2609615.

26 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6), December 2020. doi:10.1145/3415148.

27 Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 78–89. Morgan Kaufmann Publishers Inc., 1999. URL: http://www.vldb.org/conf/1999/P7.pdf.

28 Sergey Slotin. Binary search. https://en.algorithmica.org/hpc/data-structures/binary-search/, 2021.

29 Sergey Slotin. Static b-trees. https://en.algorithmica.org/hpc/data-structures/s-tree/, 2021.

30 StackOverflow. How to generate zipf distributed numbers efficiently? https://stackoverflow.com/questions/9983239/how-to-generate-zipf-distributed-numbers-efficiently, 2012. Accessed: 03/10/2024.

31 Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. PLEX: towards practical learned indexing. *CoRR*, abs/2108.05117, 2021. arXiv:2108.05117.

32 Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. Learned index: A comprehensive experimental evaluation. *Proceedings VLDB Endow.*, 16(8):1992–2004, April 2023. doi:10.14778/3594512.3594528.

33 Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. Updatable learned index with precise positions. *Proceedings VLDB Endow.*, 14(8):1276–1288, 2021. doi:10.14778/3457390.3457393.

34 Jiaoyi Zhang, Kai Su, and Huanchen Zhang. Making in-memory learned indexes efficient on disk. *Proceedings ACM Manag. Data*, 2(3), May 2024. doi:10.1145/3654954.

A Supplementary Materials

Section A.1 provides all the details, links, and parameters used for extracting, downloading, and generating the datasets. Section A.2 describes in more detail the tested indexes, giving a short glimpse at their inner workings. Section A.3 provides a textual deep dive into the results in all the single datasets.

A.1 Dataset Details

In this section, we describe all the methods used for extracting and generating the datasets used in this benchmark:

CompanyNet: Dataset derived from a graph via concatenation of its adjacency lists. The first list is represented *as is*; each subsequent i-th list is stored by adding each of its values to the maximum value in the previous list, incremented by 1.

Friendster: Dataset derived from the Friendster[25] social network, provided by Stanford, using the same method as CompanyNet.

Amzn: Book ratings from the Books_200M_uint32 dataset from SOSD.

Wiki: Wikipedia page edit timestamps from the wiki_ts_200M_uint32 dataset.

Normal: Synthetic dataset generated using the C++ standard library. The parameters used are μ = UINT32_MAX/2, and σ = UINT32_MAX/4.

Lognormal: Synthetic dataset generating using the C++ standard library. The parameters used are $\mu = 0$ and $\sigma = 0.5$. All values are then multiplied by UINT32_MAX/5.

Exponential: Synthetic dataset, generating using the C++ standard library. The parameter of the distribution is $z = 2$, and each value is multiplied by UINT32_MAX/5.

Zipf: Synthetic dataset where the parameters used are max_value: UINT32_MAX/2 and $q = 0.7$, generated using the tool in [30].

Amzn64 : books_800M_uint64 dataset from SOSD.

Facebook64: fb_200M_uint64 dataset from SOSD

OSM-Cellids64: osm_cellids_800M_uint64 dataset from SOSD

Wiki64: It uses the same values as in dataset "Wiki", but storing them in 64-bit integers.

A.2 A Detailed Description of Tested Indexes

This Section expands the description provided in Section 4 of the main article, and it offers a brief description of the inner workings of all of them. The detailed description of the SIMD-BTree and SIMD-Sampled-BTree is provided in Section 4.1 of the main paper.

std::vector is the simplest data structure commonly used to store ordered integer keys and is provided by the C++ standard library. To search for an element (integer) within the vector, the `lower_bound` function (also part of the standard library) is used, which returns the NextGEQ of the queried key.

CSS-Tree is a Header-Only implementation of the "Cache Sensitive Search Tree" described in [27]. The core idea behind this data structure is to create B-Tree nodes sized to fit a machine's cache line, avoiding using internal pointers and allowing navigation within the structure through arithmetic operations on offsets.

FAST is one of the earliest (2010 [15]) attempts at a SIMD-BTree, where the tree is logically organized to optimize for architecture features like page size, cache line size, and SIMD width of the underlying hardware.

[25] Friendster Social Network Data: `https://snap.stanford.edu/data/com-Friendster.html`

Static-search-tree recently released RUST library [17] implementing the S-Tree described in Algorithmica [29]. This implementation proposes the S-Tree along with all the proposed ideas for future works, and heavily optimizes everything. One of the most valuable additions to the original Algorithmica article is the added possibility of batching queries to increase the throughput dramatically.

RMI (*Recursive Model Index*) is undoubtedly the precursor of all learned indexes (2018) [18, 22]. It is a learned index based on two levels: The first level estimates the position of the searched element, directing the search to a second level composed of multiple ML models, each tasked with providing a more precise estimate. The type of ML model used in these two levels is customizable (e.g., linear, cubic, etc.), as are other parameters (such as the branching factor of the first level and the number of models in the second). The authors also proposed an optimization technique [22] to select the best combination of these parameters for a given input dataset. RMI is implemented in *Rust* and, for each dataset, creates three files (with extensions .cpp, .h, and .hpp) that can be used as a model interface.

PGM-index (Piecewise Geometric Model) [12] is historically the second learned index proposed in the literature (2020). It is based on a hierarchical data structure, à la B-Tree, where the hierarchy is obtained by calculating at each level a *Piecewise Linear Approximation* (i.e., a sequence of segments) of the data to be indexed, defined based on a parameter ϵ. The idea is to construct a sequence of segments $s_j = (k_j, p_j, i_j)$ such that every element x at position r to be indexed is "covered" by a segment s_j where the distance between x and the prediction obtained with s_j, i.e., $p_j \times r + i_j$, is at most ϵ. It has been shown [12] that this representation is compact in practice, with interesting theoretical worst-case bounds.

PGM++ is a recent (2024) optimized variant of PGM-index [20] that addresses its slowdowns in certain contexts, particularly where it performs inefficiently compared to RMI. The authors focus on the *last-mile search*, i.e., the final binary search step required after predicting the position of the searched element using ML models.

ALEX is one of the most performant learned indexes available in the literature (2020), developed by Microsoft Research [6]. Its API is of the key-value store type, where keys and values can have different types. ALEX is implemented in standard C++14, a version unsuited for our benchmark, so we modified some lines of code that were made obsolete[26].

PLEX is one of the most recent learned indexes [31] available in the literature (2021), based on a single hyperparameter (ϵ, the maximum prediction error) that controls the trade-off between search time and the space occupied by the index. PLEX approximates the input data using a *spline* function. It then constructs a second level consisting of an ML model based on Compact Hist-Tree (CHT). This second level is essentially a hybrid between a traditional radix-tree (which enables the binary representation of data) and a learned index (which approximates the underlying data distribution using buckets/histograms).

[26] Some indices exploit features of more recent standards than C++14. PGM-index, PGM++, LA-vector, and SIMD-BTree use a default mechanism of range-based for loops, library functions (i.e. `std::is_same_v`, `std::is_floating_point_v`), and `if constexpr` expression available starting C++17; SIMD-BTree also uses `consteval` expressions, available starting C++20. ALEX uses `std::allocator<T>::destroy` that was removed in C++20, but is still available with a different signature, thus requiring a tiny adjustment.

EliasFano is a compressed index [8, 9, 10] based on the corresponding algorithm for compressing ascending (or non-descending) integer sequences. The compression algorithm first represents each integer in binary form using $\log m$ bits, where m is the maximum integer in the collection. The approximately $\log(m/n)$ least significant bits (where n is the number of input elements) are then stored consecutively in a vector L, while the number of integers sharing the first $\log n$ bits are represented in the unary form in a vector H. The total space occupied by this compressed representation can be estimated as $n(2 + \log(m/n))$ bits. This representation also guarantees constant-time access to integers and logarithmic search costs.

γ-code and δ-code vectors are compressed indexes [7] that store non-descending sequences of integers using the corresponding compression algorithms [10] and then add appropriate and succinct extra information for random access to the data. Specifically, the γ-code technique represents each integer in binary, removes the most significant bit, and then prepends the unary representation of the remaining bit count. The δ-code technique recursively applies the γ-code to that bit count. The extra information consists of an offset vector, suitably encoded, that points to blocks of integers compressed using the previous two codes and of a size chosen by the programmer (called *density*).

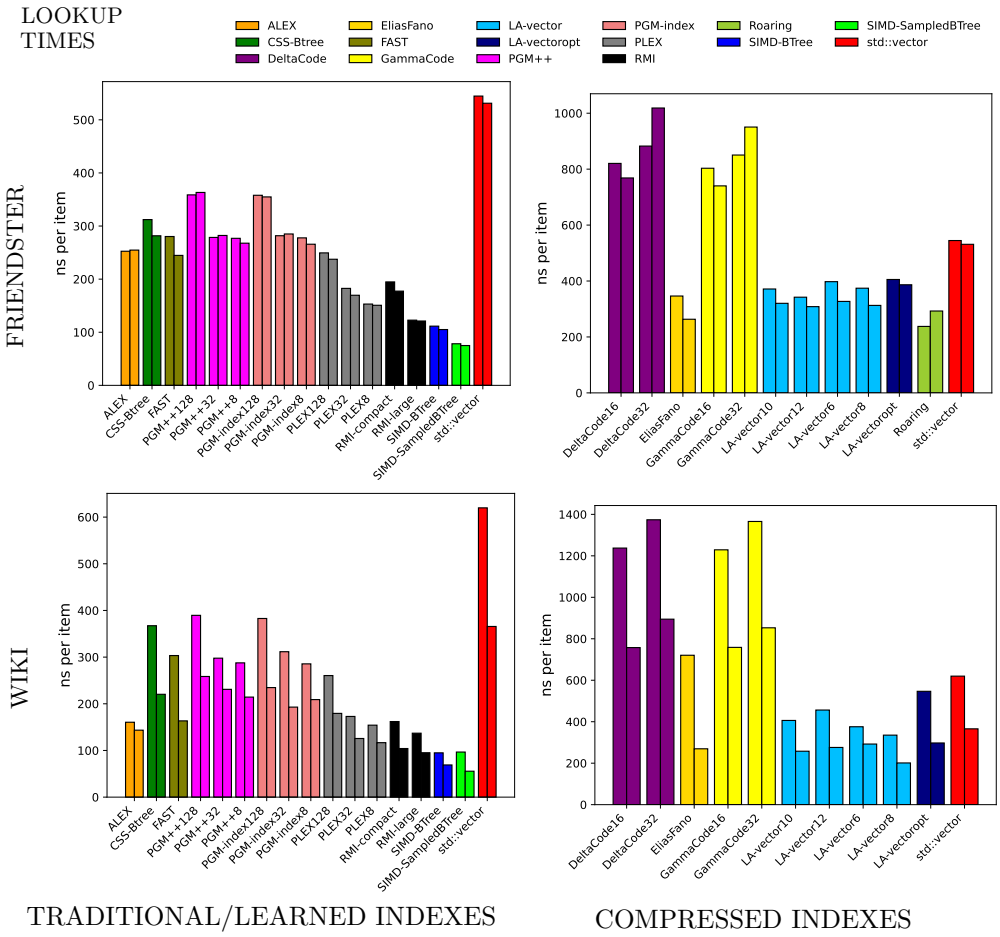
LA-vector is a compressed index [2, 1] based on the same algorithmic principles as PGM-index. It approximates the distribution of the input data using a Piecewise Linear Approximation (PLA) with parameter ϵ, and then stores for each element of the input vector only the difference between it and the value returned by the segment approximating it, using $\log_2(2\epsilon + 1)$ bits. The LA-vector library also utilizes some compressed data structures from the SDSL library to support efficient access and search (Next-GEQ) on the compressed representations of the elements. Additionally, the library offers the ability to minimize the space occupied by LA-vector by appropriately selecting the value of ϵ, potentially varying it throughout the input sequence to account for variations in data regularity. The final result of this optimization (abbreviated as "opt" in the experiments) is an index that is highly compressed but requires more query time.

Roaring is a highly optimized compressed bitmap [19], based on an implementation that heavily utilizes SIMD operations to accelerate queries. To compress the bitmaps, Roaring partitions the data space into sets: $[0, 2^{16})$, $[2^{16}, 2^{17})$, $[2^{17}, 2^{18})$, $\cdots [2^{31}, 2^{32})$. The 16 least significant bits of each integer are represented in one of the following three ways: (i) a bitset container of about 8 kB; (ii) an array container of 4096 sorted 16-bit integers; (iii) "runs" of consecutive integer pairs from s to $s + l - 1$ represented as (s, l).

A.3 Detailed Experimental Results

In this Section, we comment in detail on the experimental results on all datasets, pointing out the "best performers" in terms of space-time tradeoff, as reported in Table 3, shown in the main article. Figure 4 shows the lookup plots for all indexes on the Friendster and Wiki datasets. In these plots, for each index, it is possible to see the slight difference between making queries about existing and missing items (left and right columns, respectively). Most datasets show slight differences between those two kinds of queries (see the Friendster row), but the ones that differ most are those on datasets with a high amount of duplicates (see the Wiki row). In those cases, all indexes tend to have a lower query time for missing items. We recall that all plots, tables, and error reports are available in the "plot_recap.pdf" file in our public code repository[27].

[27] `https://github.com/LorenzoBellomo/SortedStaticIndexBenchmark`



The plots above report the time (in ns) required to perform a pointwise query (on average) for traditional/learned indexes on the left and compressed indexes on the right. The left/right bars show the average time to search for an existing/missing item in the collection.

Figure 4 Average lookup times (in ns) on the Friendster and Wiki datasets for traditional/learned indexes (on the left) and compressed indexes (on the right).

On the real datasets, we can observe the following "best performers":

CompanyNet: This is the smallest dataset, and its performance should not be considered fully "reliable." Among the non-compressed indexes, we highlight the two variants of SIMD-BTree; among the compressed indexes, we mention EliasFano and LA-vector; and among the learned indexes, certainly PLEX.

Friendster: Among the traditional indexes, we again highlight the two variants of SIMD-BTree; among the compressed indexes, we mention EliasFano and Roaring; and among the learned indexes, definitely PLEX and RMI.

Amzn: Among the non-compressed indexes, we again highlight the two variants of SIMD-BTree; among the compressed indexes, we mention EliasFano; and among the learned indexes, certainly ALEX.

Wiki: This dataset yields index performances that are significantly different from the others, likely due to the many duplicates (see also the results on the Zipf dataset). Here, LA-vector obtains an exceptional space/time tradeoff. For traditional indexes, the SIMD-BTrees are the best, while PLEX and ALEX are the best learned indexes. It is worth noting that on this dataset (and all others with many duplicates), searches for items absent from the dataset are much faster on almost all indexes, up to three times faster.

On the synthetic datasets, we can observe the following "best performers":

Normal, LogNormal, and Exponential: Among the non-compressed indexes, we highlight the two variants of SIMD-BTree, among the compressed indexes, we mention LA-vector and EliasFano, while among the learned indexes, certainly ALEX (best on Normal and Exponential) and RMI (best on Lognormal).

Zipf: As observed for Wiki, on this dataset too, the performance of the indexes is significantly different compared to the other synthetic datasets, likely due to the presence of many duplicates. There are only two "best performers," which are LA-vector and the two variants of SIMD-BTree. The latter achieve a 3x improvement in time performance but lose in space due to the need to explicitly store the keys.

On the 64-bit datasets, we can observe the following "best performers":

Amzn64: Among the non-compressed indexes, we highlight the two variants of SIMD-BTree; among the compressed indexes, we mention EliasFano; while among the learned indexes, RMI stands out.

Facebook64: Among the non-compressed indexes, we highlight the two variants of SIMD-BTree; among the compressed indexes, we mention Roaring; and among the learned indexes, we cite RMI and PLEX.

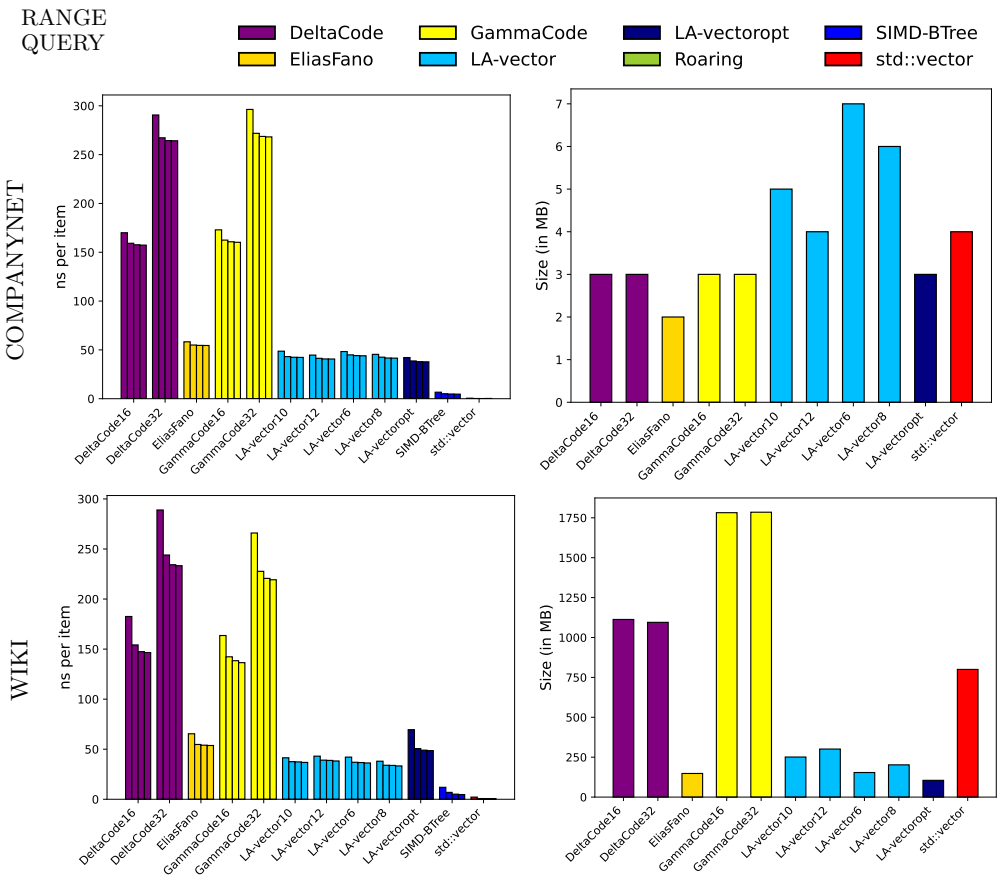
OSM-Cellids64: This dataset has a peculiar distribution that results in highly varied performance, especially among the compressed indexes, which are often very slow. The only compressed index that maintains "acceptable" time performance compared to std::vector is δ-code-vector. Regarding learned indexes, RMI and PLEX are the best, while we mention the SIMD-BTrees as the best performers overall.

Wiki64: The same observations made for the Wiki dataset with 32-bit integers apply here. The winners are the SIMD-Btrees, ALEX, PLEX, and LA-vector.

We conclude the experimental analysis by focusing on the performance of the indexes on "range" queries, shown in Figure 5.

In this experiment, we consider four different sizes for the number of scanned elements (i.e., 10, 100, 1 000, and 10 000), and we limit the discussion to the most performant compressed indexes (i.e., EliasFano and LA-vector), comparing them with std::vector as a baseline:

- Performance slightly improves as the number of adjacent scanned elements increases, as the most costly step is due to the random access to the first element, but the improvements stop being noticeable after 100 elements, where the decompression time becomes the main bottleneck.
- the std::vector obtains results that are orders of magnitude better than all the compressed indexes in this task, while the SIMD-BTree is a close second, obtaining a slight slowdown but outperforming by far all the compressed indexes.



The plots above show the performance of compressed indexes in time and space relative to *range queries*, where starting points are randomly sampled, and the width of the scan is set to 10, 100, 1K, and 10K. The plot on the left shows the time (in ns) required for every interrogation, describing the average time required per access with $x = 10, 100, 1K, 10K$. The plot on the right shows the space occupied by each compressed index (in MB).

Figure 5 Average time (in ns) for range queries on compressed indexes, on the CompanyNet and Wiki datasets.