Benchmarking Learned and LSM Indexes for Data Sortedness

Aneesh Raman
Boston University
aneeshr@bu.edu

Andy Huynh
Boston University
ndhuynh@bu.edu

Jinqi Lu
Boston University
jinqilu@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Database systems use indexes on frequently accessed attributes to accelerate query and transaction processing. This requires paying the cost of maintaining and updating those indexes, which can be thought of as the process of adding structure (e.g., sort) to an otherwise unstructured data collection. The indexing cost is redundant when data arrives pre-structured, i.e., pre-sorted, even if only *up to some degree*. While recent work has studied how classic indexes like B$^+$-trees cannot fully exploit the intrinsic data *near-sortedness* during ingestion, there is a lack of this exploration on index designs like read-optimized learned indexes or write-optimized LSM-trees.

In this paper, we bridge this gap by conducting the first-ever study on the behavior of state-of-the-art learned indexes and LSM-trees when varying the degree of data sortedness in an ingestion workload. Specifically, we build on prior work on benchmarking data sortedness on B$^+$-trees and we expand the scope to benchmark: (i) ALEX and LIPP, which are updatable learned index designs; and (ii) the LSM-tree engine offered by RocksDB. We present in detail, our evaluation framework and present key insights on the performance of learned indexes and LSM-tree designs with respect to data sortedness. Our observations indicate that learned indexes exhibit unpredictable performance when ingesting differently sorted data, while LSM-trees can benefit from sortedness-aware optimizations. We highlight the potential headroom for improvement and lay the groundwork for further research in this domain.

1 INTRODUCTION

Data systems often utilize indexing structures on frequently queried attributes for accelerated query processing to meet the ever-growing demand for real-time analysis of large volumes of data. Typically, indexes establish *structure* or order to the data by paying an indexing cost which allows for efficient data access. However, if data arrives fully sorted or sorted up to some degree, the cost to create indexes is entirely redundant. This becomes a performance bottleneck as modern data requirements entail systems to support fast data ingestion in addition to offering efficient query processing.

Near-Sortedness in Practice. Several applications often generate *near-sorted* data, i.e., data is not entirely ordered but close to being fully ordered. For example, data tables that have correlated attributes [2], intermediate results of previous join operations [3], aggregated time series, event-based data such as sensor failures [14, 34] or naturally generated data such as stock market prices are all near-sorted, but may not necessarily be fully sorted. The degree to which such data is nearly sorted is characterized by *data sortedness* and has been widely explored in sorting and searching algorithms [3, 6, 7, 17, 21, 35].

Are Indexes Sortedness-Aware? When data is available fully sorted and in its entirety, the index construction cost can be significantly amortized through bulk loading [1, 9]. However, the prospect of exploiting intrinsic data sortedness up to any degree during index construction has yet to be fully solved. Recent work provides a framework for evaluating indexes with a varying degree of sortedness through the *Benchmark on Data Sortedness* (BoDS) [28]. Yet, the benchmark and its follow-up work on a Sortedness-Aware Paradigm [29] concentrate on a B$^+$-trees. While B$^+$-trees are often used in commercial data systems due to their balanced read-write performance [8, 16, 22, 23, 25–27], several write [4, 5, 15, 20, 24] as well as read optimized indexes exist [10, 13, 18, 30, 36]. In this work, we expand the scope of benchmarking near-sorted workloads by testing modern index designs, both *learned indexes* and *LSM-trees*.

Learned indexes [10, 13, 18, 36] aim to replace standard index traversals with an alternative strategy using a hierarchy of machine learning models that can inexpensively predict the location of a key (albeit with an error bound). Though initially proposed as a read-only index structure as re-training machine learning models is expensive, recent proposals like ALEX [10] and LIPP [36] provide updatable index designs, enabling wider adoption. ALEX, for instance, uses a gapped array layout for its data nodes that distributes extra space between entries to enable a model-driven insertion policy. Evaluations with random insertions show considerable benefits offered by ALEX over the B$^+$-tree for mixed read-write workloads as the index easily adapts to changes in data distribution [10]. However, ingesting near-sorted data could potentially reduce the advantage learned indexes hold over B$^+$-trees due to less frequent changes in data distribution and risks potential overfitting of learned model.

On the other hand, LIPP stores the exact positions of keys within the data which eliminates model-based predictions during insertions and provides a fallback during lookups if the models are inaccurate. LIPP overcomes the drawbacks of other learned index designs by extending the tree structure during updates caused by the collision of model predictions and combining it with a dynamic height adjustment strategy to bound the tree height. Similarly to ALEX, evaluation of LIPP focused entirely on workloads following a random insertion order. Thus, LIPP may potentially carry similar drawbacks to ALEX for near-sorted data; in particular, overfitting of learned models may have an exacerbated cost as more collisions may cause expensive dynamic tree adjustments to occur.

Orthogonally, LSM-trees [24] are widely used in modern key-value stores due to their ability to allow high ingestion rates and fast reads. To achieve fast data ingestion, LSM-trees buffer writes in memory, and will periodically flush, merge, and organize data on storage in logical levels (i.e., *compactions*) to help amortize the ingestion cost. Generally, LSM designs—such as RocksDB [12]—are highly tunable [32] and offer fine-grained control of data movement, the flush and merge policies [19, 31], and the overall layout of the underlying index. For instance, RocksDB supports partial compactions, where only a subset of files on disk with overlapping

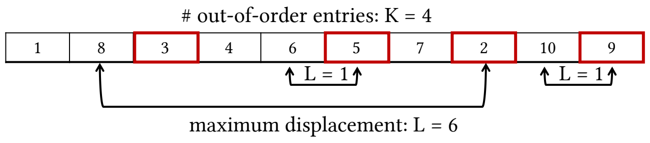

Figure 1: An example of a near-sorted data collection quantified by the (K, L)-sortedness metric.

key ranges between logical levels are merged to reduce write amplification. RocksDB also supports *trivial moves* [33], where files with non-overlapping key ranges can be directly moved between levels through simple pointer manipulation and does not require re-writing and merging. This can potentially benefit workloads that are highly sorted, maintaining the LSM-trees high ingestion rates.

Contributions and Benchmarking Observations. In this work, we benchmark the performance of learned index designs: ALEX, LIPP and the state-of-the-art LSM-tree (using RocksDB); when varying data sortedness. Our evaluation framework puts together the data generator from BoDS along with adapters to the different indexes with the helper scripts written in C++ and Python. We make our framework available on GitHub so that it can be easily extended to incorporate other index designs in the future.[1] Our key observations are highlighted as follows:

(1) We establish that any design must be evaluated over varying degrees of sortedness in the ingestion workload as standard.
(2) Learned indexes can be unpredictable when varying sortedness. Particularly, LIPP can be either 4.4× faster or 1.9× slower than ALEX, depending on data sortedness.
(3) Learned indexes may also overfit their models when learning data distribution with high data sortedness. We find that LIPP fails to ingest fully sorted data through sequential writes due to repeated collisions creating an unbalanced left-deep tree.
(4) LSM-trees benefit from optimizations like trivial moves found in RocksDB, however, this can be further optimized to exhibit ideal sortedness-awareness. Specifically, when transitioning between fully sorted to minorly unsorted workloads, throughput decreases by ≈ 20% while the files trivially moved are ≈ 5× fewer.

The rest of this paper is structured with §2 covering the relevant background, §3 discussing our evaluation framework, and §4 showing our key observations.

2 BACKGROUND

In this section, we provide the necessary background on the data generator from BoDS [28] and the indexes we benchmark.

Benchmark on Data Sortedness (BoDS) [28], provides a framework for evaluation of the performance of data systems when varying data sortedness in the ingestion workload. The benchmark features: (i) a synthetic data generator that creates differently sorted collections, and (ii) a workload executor that converts the data files into workloads having only writes or both reads and writes. The data generator uses an intuitive (K, L)-sortedness metric, inspired by [3], that quantifies data sortedness through two parameters: *how many entries are out-of-order* and *by how much*. The number of out-of-order entries are denoted by K and the maximum displacement of any out-of-order entry in the data collection is denoted by L. Figure 1 shows an example of a near-sorted data stream quantified by the (K, L)-sortedness metric. Here, the unordered entries are marked by the red boxes, and their displacements are visualized by the arrows. Note that a low L signifies local unorderness in the data as entries are displaced closer to their ideal positions, while a high L points to global unorderness.

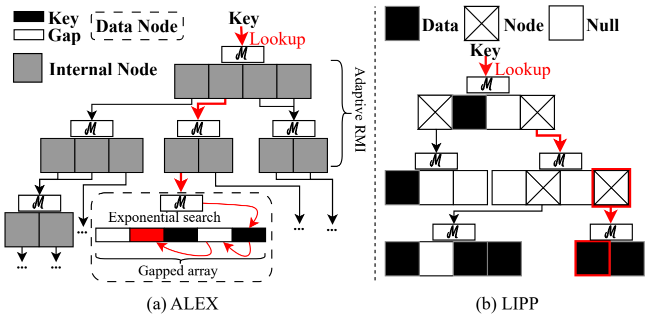

Figure 2: Overview of ALEX and LIPP: (a) ALEX follows model-based insertions into the index. Gapped Arrays in data nodes help amortize write amplification caused by shifting keys during ingestions; (b) Every node in LIPP contains a bit-vector denoting three entry types: gap or empty space, data (containing a single entry), or a link to a child node (created through collisions in predicted locations).

Learned Indexes look to replace index traversals through machine learning models (e.g., regression models) that can accurately predict the location of a key in the dataset [10, 18, 36]. They do so by learning the relationship between the keys and their data positions, i.e., the data distribution. Using models rather than index traversals reduces the number of disk accesses and comparisons needed to lookup data in the index, thereby, improving query performance. While learned indexes were traditionally proposed as read-only structures to avoid expensive re-training overhead during updates [18], recent effort has proposed updatable variants, most notably - ALEX and LIPP.

ALEX [10] at its core uses a Recursive Model Index (RMI) that predicts the position of a key in the data. As with a B$^+$-tree, ALEX builds a tree structure, however, utilizes a different node structure that can grow or shrink at different rates. Data nodes in the index use a gapped array between existing entries in the node, unlike the B$^+$-tree that accumulates gaps at the end of the node, as shown in Figure 2a. The gapped array reduces the cost of shifting keys for each insertion as the gaps can directly absorb insertions. Further, insertions to the index are *model-driven*, i.e., records are expected to be closely located to their predicted positions. ALEX also uses exponential search during lookups that is faster than binary search when the RMI is highly accurate. As with a learned index, insertions cause the model's errors to increase, however, ALEX selectively retrains the model only when needed with the use of simple cost models that account for current workload characteristics.

[1]https://github.com/BU-DiSC/bliss_benchmark

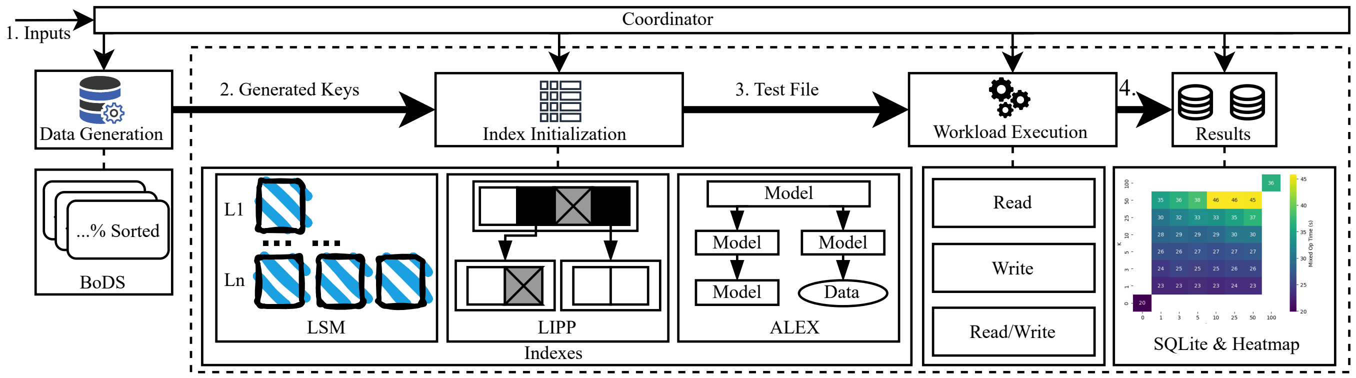

Figure 3: Overview of our benchmarking framework

LIPP [36], illustrated in Figure 2b, also uses model-based insertions for writes to the index, however, with a precise key-to-position mapping in the index. Unlike ALEX, which shifts entries in case of collisions in predicted locations by the RMI model, LIPP chooses to create a new child node linked to the particular position to hold the keys. This way, multiple keys can be mapped to a single predicted location without affecting the prediction errors. Each node in LIPP contains a model, an array of entries, and a bit vector of entry types (as elements for that node). Each entry type can either be a gap, or can contain data (a single entry) or be linked to a child node. The index tightly bounds the tree height with a dynamic, lightweight re-adjustment strategy to $O(logN)$, where N is the size of the ingested data. The strategy chooses the appropriate subtree to determine when and how to readjust and reduce the tree height.

LSM-trees [24] buffer entries in an in-memory component to optimize ingestion before flushing them as sorted runs to the disk. The sorted runs are later merged with the existing disk components through a process termed as *compaction*. The compaction policy can also be controlled to either eagerly merge (*leveling*) or lazily merge (tiering). Modern LSM-tree engines like RocksDB [12] also allow for partial compactions [11] with different data movement policies. Textbook LSM-tree designs perform the same amount of merging and re-writing of data during compactions, even for fully sorted data. Meanwhile, partial compactions only compact a subset of files that have overlapping key ranges to reduce write amplification, as the amount of data re-written during every merge is minimized. Further, RocksDB also performs *trivial moves* [33] that are lightweight pointer manipulations to move SST files from one level to the other if there are no overlapping key ranges. This reduces redundant re-writing and merging of files during compactions.

3 BENCHMARK ARCHITECTURE

In this section, we describe our methods and discuss the mechanics of the benchmark. Figure 3 provides a graphical overview of the different components of the benchmark.

Coordinator. For performance, the benchmark is primarily written in C++, however, we provide a coordinator that links together data generation, index initialization, workload execution, and recording results in Python. Performance numbers and logs are stored in a SQLite database for easy analysis and plotting after benchmarking.

Data Generation. As we are concerned with sortedness, we choose to generate data using BoDS [28]. For results shown in this paper, we generate data files with 500 million 64-bit keys ($\approx$ 4GB). If the user prefers a different data generation method, we provide hooks to allow other data generation programs to be integrated.

Index Initialization. Our benchmark integrates with four modern index designs: three in-memory indexes, the B$^+$-tree, ALEX, and LIPP, and the modern LSM-tree through RocksDB. We use the B$^+$-tree from the TLX library, while ALEXand LIPP both provide source code online [10, 36]. Each data structure is encapsulated in a C++ wrapper that specifies how each structure preloads, reads, and writes data. Any index-specific initialization is performed before benchmarking. Users can integrate other indexes or data structures into the benchmark by simply adapting the index wrapper.

Workload Execution. When data files are ingested, the data stream is split up into four distinct execution phases: a preload phase, a write phase, an interleaved operations phase, and a read phase. For example, following the benchmark's default settings, the preload phase operates on 40% of the input stream, the write phase 40%, and the interleaved operation phase 20%. Lastly, the read phase defaults to submitting a number of read queries equal in length to 25% of the input stream. Read keys are generated uniformly at random from the set of ingested keys. During ingestion, every key is associated with a randomly generated payload of fixed length.

Preload phase: By default, the preloading phase will call the equivalent insert operation for each entry in the preload stream. However, certain data structures may have the ability to perform a bulk load operation. In such instances, the preloading behavior can be changed to perform specialized bulk loading that is specific to each data structure. We utilize bulk loading for the preloading experiments presented in this paper.

Write phase: Once the preloading completes, the index is benchmarked with a write phase that consists of sequentially executed insert operations.

Read phase: The read phase consists of sequentially executed read operations. Keys generated for the query are selected uniformly at random from the list of keys already inserted into the data structure.

Interleaved phase: Lastly, the interleaved operation phase combines the read and write phases and randomly selects a uniform mixture of read and write operations.

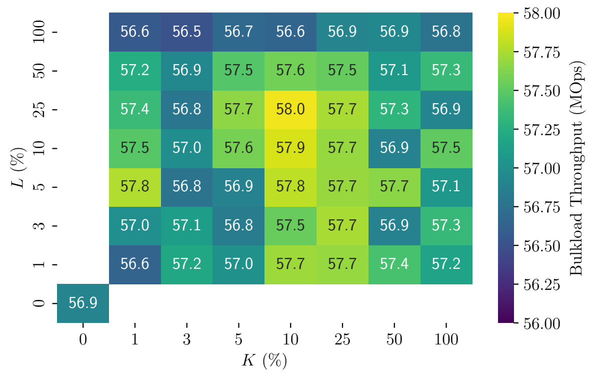

Figure 4: A sample heatmap showing the write throughput in B$^+$-tree when bulk loading the index.

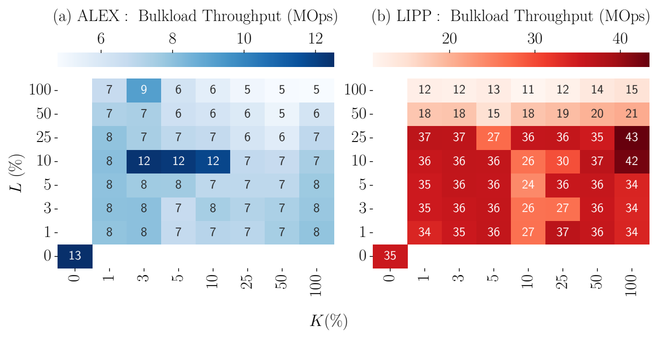

Figure 5: Bulk loading learned indexes with differently sorted data: (a) ALEX benefits from high data sortedness; (b) LIPP works better with scrambled data.

Analyzing the Results. Once benchmarking is complete, we pull performance logs from the SQLite database to produce heatmaps relating the dimensions of sortedness (K and L) to the preload, read, or write throughput. A sample heatmap is shown in Figure 4. By default, the x-axis varies the fraction of out-of-order entries in the data collection (K) while the y-axis varies their maximum displacement (L). The bottom left (K=0, $L = 0$) corresponds to a fully sorted data collection, while the top right corner ($K = 100$, $L = 100$) corresponds to a fully scrambled data collection. Results for $K = 0$ or $L = 0$ are left blank as any data collection with even one of the parameters equalling zero are fully sorted. Every cell in the heatmap corresponds to a particular evaluation metric denoted by the corresponding color bar label. For example, Figure 4 shows the observed throughput while pre-loading a B$^+$-tree index. Note that we exclude detailed analysis of B$^+$-trees in this benchmark as it has been extensively covered in prior work [28, 29].

4 EXPERIMENTAL EVALUATION

We now present the experiments to evaluate the behavior of different indexes in the presence of variable data sortedness. Note that we do not aim to compare the three approaches (for example, the learned indexes are in memory, while RocksDB operates on storage), rather, we attempt to explore the relative behavior of the three index designs as we vary data sortedness, both for their ingestion time and, surprisingly, for their query time.

Experimental Setup. Our server is configured with two Intel Xeon Gold 6230 processors, 384 GB of main memory, a 1 TB Dell P4510 NVMe drive, CentOS 7.9.2009, and a default page size of 4 KB.

4.1 Learned Index

First, we present the results for the read-optimized learned index structures, namely, ALEX and LIPP.

Index Setup. Both ALEX and LIPP are set up with the default settings in accordance to their code base [10, 36]. Namely, ALEX is initialized with a node size of 16MB, and LIPP is initialized with a bitmap width of 1byte.

4.1.1 Bulk Loading. In this experiment, we measure the bulk load throughput for ALEX and LIPP with varying degrees of sortedness.

ALEX Performs Best With Fully Sorted Data. We observe (from Figure 5a) that ALEX offers the best throughput when the ingested data is fully sorted or near-sorted, however, it suffers when both K and L are high ($K \geq 25\%$ and $L \geq 25\%$) as the throughput drops by up to 60%. ALEX bulk loads the tree greedily by recursively partitioning the input stream and deciding independently and locally each internal node's fanout. The index picks the optimal fanout by building a *fanout tree* for every RMI node and calculating the expected costs using a cost model to traverse the tree to the data nodes. Although we bulk load a pre-sorted data stream, the possibility of picking optimal split points without recomputing the fanout tree is higher when the input stream itself has high data sortedness. However, we also observe an anomaly of high throughput for $L = 10\%$ and $K = 3\%, 5\%, 10\%$ which requires further investigation into the internals of the index.

LIPP Performs Best with Local Unsortedness. We observe in Figure 5b that LIPP offers a significantly higher throughput (at least $\approx 1.5\times$ better) when data is locally out-of-order ($L \leq 25\%$). The index is largely unaffected by local unsortedness (small values of L) and its throughput is comparable to ingesting fully sorted data. In fact, even when all entries are out of order ($K = 100\%$) but $L \leq 25\%$, LIPP has its highest ingestion rate. Bulk loading in LIPP starts by creating a *Fastest Minimum Conflicting Degree* (FMCD) model for every node, which is used to insert subsequent entries. If conflicting entries are mapped to the same location, the index collects them and recursively builds partial subtrees to remap their positions correctly. Therefore, the bulk loading performance can be correlated to the number of times the index recursively builds partial subtrees during conflicts. When L is low, the data is more densely packed, making it easier to pick the optimal split points with minimum conflicts during bulk loading. This even holds for $K = 100$ as the FMCD algorithm can build an accurate linear model with minimized conflicts due to well-distributed data. Overall, with local unsortedness, we expect LIPP to be better equipped to densely pack nodes during bulk loading.

4.1.2 Writes. Next, we report our observations when measuring the performance of sequential insertions for both ALEX and LIPP in Figure 6. Note that we execute the sequential insertions after the pre-loading phase (after bulk loading each index).

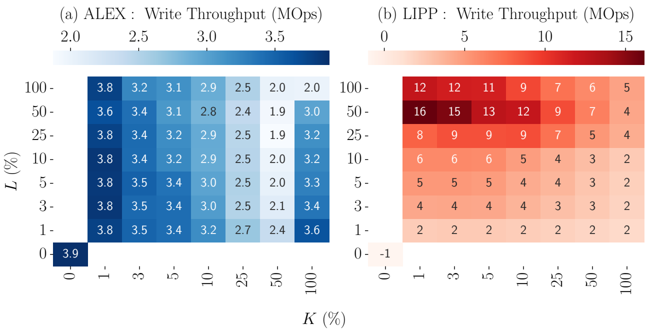

Figure 6: Ingestion Performance of learned indexes in stable-state: (a) ALEX offers better throughput when ingesting data with high sortedness; (b) LIPP performs better when the displacement of unordered entries (L) is larger for a fixed K.

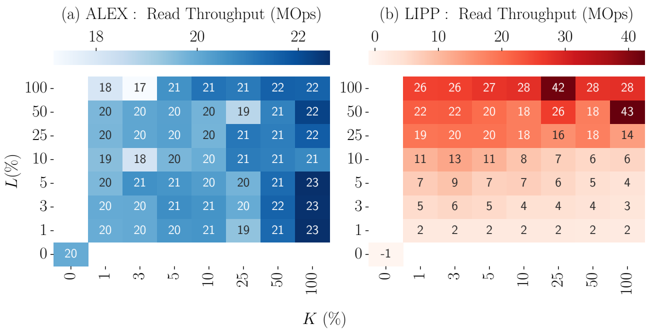

Figure 7: Lookup performance (existing queries) of the learned indexes: (a) Overall, ALEX offers a stable lookup performance; (b) Lookups in LIPP are faster when L is higher.

ALEX is Unpredictable When Varying Sortedness. We observe in Figure 6a that ALEX offers the best throughput when ingesting data with low K and L ($< 10\%$). The indexing effort during stable-state ingestions is expected to be dominated by finding the correct insertion position in the gapped array (through exponential search in case of incorrect predictions). When the ingested data is highly sorted, entries in the data node are likely to be densely packed, leaving long contiguous runs followed by huge terminal gaps, similar to a B$^+$-tree. In this case, exponentially searching for the correct gap is easier, and we believe this is a reason for the higher throughput. On the other hand, ALEX also performs well when all entries in the input stream are out-of-order but have low or moderate displacement ($L \leq 50\%$). In this case, a lower L implies fewer contiguous runs of entries in the data node (leaving more gaps), which may reduce incorrect predictions. The performance degrades as we increase L, and ALEX exhibits its worst throughput for scrambled data ($K = 100$, $L = 100$).

LIPP Offers Better Write Throughput for Higher L. Contrary to pre-loading, we observe from Figure 6b that LIPP performs the best, by up to 8×, when the ingested data has fewer unorder entries ($K \leq 10\%$) but are displaced further away from their ideal positions ($L \geq 25\%$). The insertion cost in LIPP comes from the readjustment strategy that may trigger recalibrations of the model. Generally, for a fixed K, an input stream with a higher L offers a better sample exposing a wider range of entries (i.e., higher data resolution) than one with a low L value. Naturally, the model is better trained to handle conflicts when it is exposed to a coarser sample with a wider domain (compared to a biased fine-grained sample), thus requiring fewer re-adjustments or recalibrations. Further, low L in ingested data leads to dense packed nodes during index creation (as we observe in §4.1.1, which triggers more conflicts in the sequential write phase. Meanwhile, LIPP already performs additional effort in minimizing conflicts during index creation that pays off by leaving considerable gaps to absorb future insertions, and hence, the contradictory trend.

LIPP Fails to Sequentially Write Fully Sorted Data. We also observe in our benchmark that LIPP's current design fails to ingest fully sorted data, denoted by a -1 throughput in Figure 6b. Our investigation reveals that performing sequential writes with fully sorted data produces too many conflicting model predictions. In turn, this creates a deep subtree that fails to balance itself even with the re-adjustment strategy, causing the index to collapse.

LIPP vs. ALEX. As both systems are in memory, and we use the same setup to compare those systems in prior work [36], we can report that we corroborate that LIPP has about 2.5× higher insert throughput (see the K=100%, L=100% points in Figure 6), however, we uncover a much more interesting comparison. Depending on the data sortedness LIPP can be from 4.4× *faster* all the way to 1.9× *slower* than ALEX. Lastly, LIPP surprisingly fails to insert fully sorted data. Overall, we argue that the benchmarking analysis varying data sortedness should be an evaluation standard for all new index designs moving forward.

4.1.3 Reads. Next, we report in Figure 7 our observations when executing existing lookups into the learned indexes.

ALEX Offers Comparable Read Performance. We observe from Figure 7a that the point lookup performance of ALEX is largely comparable for any degree of sortedness in the ingestion workload. However, lookups are slightly (up to 10%) faster for higher unsortedness (both K and $L \geq 25\%$). Lookups traverse the internal nodes by using the predictive models until they reach a data node, where an exponential search may be required to find the accurate position of the key. When the fraction of unordered entries in the ingested data is low, we risk potentially overfitting the model during ingestions, which can result in inaccurate predictions. Overall, ALEX's lookup performance is driven by the accuracy of the model.

Lookups in LIPP Depend on Data Sortedness. Likewise, we observe in Figure 7b that LIPP's lookup performance benefits from a larger displacement (L) of entries in the ingested data. We believe this is a result of exposing the model to unbiased coarser samples during index construction that, in turn, improves accuracy.

4.2 LSM-trees

We now explore the sortedness-awareness of the state-of-the-art LSM-tree engine in RocksDB [12]. Here, we perform sequential writes to the system for every data collection and present our results in Figure 8 and Figure 9.

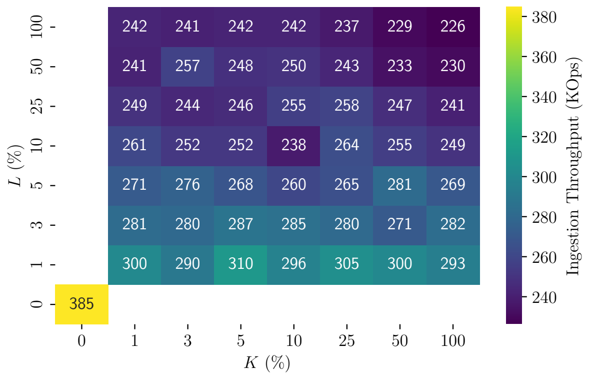

Figure 8: RocksDB offers better performance when ingesting data with high sortedness.

RocksDB Setup. We initialized RocksDB with the following settings; compaction style set to *kCompactionStyleLevel*, buffer size 40MB, entry size 16 bytes, and size ratio 4.

RocksDB Benefits from High Data Sortedness. We observe from Figure 8 that the LSM-tree in RocksDB offers the best throughput when the ingested data is fully sorted, and its performance gradually deteriorates as the data sortedness decreases. Particularly, we find that the system's write throughput highly depends on L, with its performance slowing down by up to 1.7× as the displacement of unordered entries in the workload increases. An LSM-tree works best when it is able to capture any lack of sortedness within its in-memory buffer so that its flushed sorted runs have unique ranges. Thus, a higher degree of sortedness results in lower indexing and merging effort during compactions.

Compactions Increases as Sortedness Decreases. Further, we observe from Figure 9a that the number of compactions performed increases as the sortedness in the ingestion workload decreases. In fact, RocksDB performs up to 2.7× more compactions when ingesting scrambled data ($K = 100, L = 100$). Note that we report the cumulative number of compactions performed during the ingestion cycle for a specific data collection. As sortedness decreases (increase in either K or L), the number of overlapping key ranges among the sorted runs in every level of the tree significantly increases, resulting in an increase in the number of compactions.

Trivial Moves Benefit from High Data Sortedness. Figure 9b shows that RocksDB performs more trivial moves with higher data sortedness, maximizing the metric when ingesting fully sorted data. Trivial moves enable the system to avoid unnecessary rewriting and merging (i.e., compactions) when moving files with non-overlapping ranges between levels through simple pointer manipulation. When the data sortedness is high, more non-overlapping files exist, thus, a higher number of trivial moves is performed.

Trivial Moves are Significantly Affected by Even Minor Lack of Sortedness. We also see from Figure 9b that the number of trivial moves significantly reduces as we slightly reduce sortedness in the data. For example, while RocksDB performs 524 trivial moves when ingesting fully sorted data ($K = 0$, $L = 0$), the number of trivial moves executed (96) are ≈ 5.5× fewer, with $K = 1\%$, $L = 1\%$. This is because the in-memory buffer in the LSM tree is not designed to

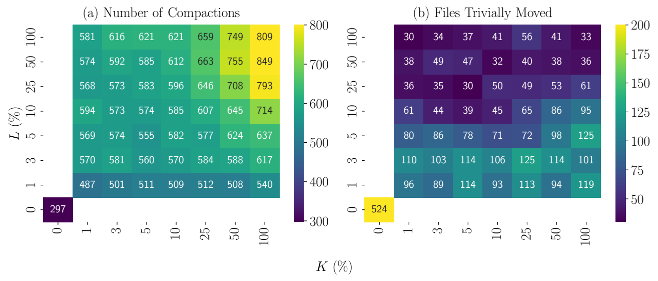

Figure 9: (a) Compactions increase as data sortedness decreases; (b) More trivial moves are executed when ingesting data with high sortedness.

exploit sortedness, and is entirely flushed once full to make space for future insertions. By doing so, RocksDB misses the opportunity to absorb any lack of sortedness in the subsequent buffer cycle, which is very likely to occur when ingesting near-sorted data. This results in overlapping key ranges among the sorted runs, which prohibits them from being eligible for trivial moves and, instead, must undergo compactions. We point out that *LSM systems like RocksDB can benefit by partially flushing the buffer to better capture near-sortedness*, like the SWARE paradigm [29].

5 CONCLUSION

Indexing data structures establish order to otherwise unstructured incoming data on the indexed attribute to facilitate efficient queries. When such structure i.e., data sortedness already exists, the indexing effort is redundant. Prior work explores the ability of classical B$^+$-trees to exploit data sortedness to improve ingestion performance, while we extend the benchmarking to modern index designs like learned indexes and LSM-trees. We summarize our key takeaways from our experimental analysis as follows:

(1) Benchmarking index ingestion performance while varying data sortedness should be a new standard.
(2) Sortedness drives performance benefits of learned indexes. Specifically, ingestion in LIPP can be anywhere between 4.4× faster to 1.9× slower than ALEX when varying data sortedness in the workload.
(3) Unpredictable performance of learned indexes given a particular degree of sortedness requires further exploration into the internals of the index designs.
(4) LSM-trees by design can absorb some degree of sortedness due to sort-merging during compactions.
(5) The ability of LSM-trees to exploit trivial moves significantly degrades (by ≈ 5×) even when inducing minor unsortedness, leaving potential for further optimizations (e.g., partially flushing the buffer) to bridge the performance gap.

Overall, we highlight the need for a wider analysis of index performance when varying data sortedness. This paper lays the groundwork for further exploration and potential opportunities for improvement in current state-of-the-art index designs.

REFERENCES

[1] Daniar Achakeev and Bernhard Seeger. 2013. Efficient Bulk Updates on Multiversion B-trees. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1834–1845.

[2] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892.

[3] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In *Proceedings of the International Conference on Database Theory (ICDT)*. 256–267.

[4] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bϵ-trees and Write-Optimization. *White Paper* (2015).

[5] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 546–554.

[6] Svante Carlsson and Jingsen Chen. 1992. On Partitions and Presortedness of Sequences. In *Acta Informatica*, Vol. 29. 267–280.

[7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 275–290.

[8] CouchDB. [n. d.]. Online reference. *http://couchdb.apache.org/* ([n. d.]).

[9] Jochen Van den Bercken and Bernhard Seeger. 2001. An Evaluation of Generic Bulk Loading Techniques. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 461–470.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 969–984.

[11] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[12] Facebook. 2021. RocksDB. *https://github.com/facebook/rocksdb* (2021).

[13] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. A-Tree: A Bounded Approximate Index Structure. *CoRR* abs/1801.1 (2018).

[14] Nikolaus Glombiewski. 2023. *Robust Stream Indexing*. Ph. D. Dissertation. Philipps-Universität Marburg.

[15] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[16] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2667–2672.

[17] Donald E. Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms (3rd Edition)*. Addison-Wesley.

[18] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 489–504.

[19] Andrew Kryczka. 2020. Compaction Styles. *https://github.com/facebook/rocksdb/blob/gh-pages-old/talks/2020-07-17-Brownbag-Compactions.pdf* (2020).

[20] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 302–313.

[21] Heikki Mannila. 1985. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers (TC)* 34, 4 (1985), 318–325.

[22] MongoDB. 2023. Online reference. *http://www.mongodb.com/* (2023).

[23] MySQL. 2023. MySQL. *https://www.mysql.com/* (2023).

[24] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[25] Oracle. 2018. Introducing Oracle Database 18c. *White Paper* (2018).

[26] PostgreSQL. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. *https://www.postgresql.org* (2023).

[27] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition.

[28] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A Benchmark on Data Sortedness. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*. 17–32.

[29] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for Near-Sorted Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1475–1488.

[30] Jun Rao and Kenneth A. Ross. 2000. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 475–486.

[31] RocksDB. 2020. Leveled Compaction. *https://github.com/facebook/rocksdb/wiki/Leveled-Compaction* (2020).

[32] RocksDB. 2021. RocksDB Tuning Guide. *https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide* (2021).

[33] RocksDB. 2022. RocksDB Trivial Move. *https://github.com/facebook/rocksdb/wiki/Compaction-Trivial-Move* (2022).

[34] Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Transactions on Database Systems (TODS)* 44, 4 (10 2019).

[35] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 36–53.

[36] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.