# Evaluating Learned Indexes for External Memory Joins

Yuvaraj Chesetti* Prashant Pandey†

**Abstract**

Joins are among the most time-consuming and data-intensive operations in relational query processing. Much research effort has been applied to the optimization of join processing due to their frequent execution. Recent studies have shown that CDF-based learned models can create smaller and faster indexes, accelerating in-memory joins. However, their effectiveness for external-memory joins, which are crucial for large-scale databases, remains underexplored.

This paper evaluates the impact of learned indexes on external-memory joins for both sorted and unsorted data. We compare learned index-based joins against traditional join methods such as hash joins, sort joins, and indexed nested-loop joins on real-world and simulated datasets. Additionally, we analyze learned index-based joins across multiple dimensions, including storage device types, data sorting, parallelism, constrained memory environments, and varying model error. The detailed evaluation enables us to determine the most appropriate learned index to employ for external-memory joins.

Our experiments reveal that, unlike in-memory settings, learned indexes in external-memory joins can trade off accuracy for space without significantly degrading performance. While learned indexes provide smaller index sizes and faster lookups, they perform similarly to B-trees in external-memory joins since the total amount of I/O, which dominates runtime, remains unchanged. Additionally, the construction times of learned indexes are $\sim$1000$\times$ longer, and although they are 2–4$\times$ smaller than the internal nodes of a B-tree, these nodes only represent 0.4%–1% of the data size and typically fit in main memory.

## 1 Introduction

The join operation is a fundamental operation in database management systems. Data normalization spreads data across multiple relational tables, and the join operation enables combining tuples from two or more relations based on a common attribute. Implementing join efficiently is challenging as it involves iterating through the tuples across multiple relations, incurring large amounts of disk I/Os. The join operation is often critical for the overall database performance as it is expensive and frequently executed. This is especially true for online analytics processing (OLAP) systems where the data is static and large-scale multidimensional analytical queries are frequent. This has resulted in extensive research on optimizing joins in recent decades [4, 6, 7, 10, 31, 29, 21].

Recently, machine learning has significantly influenced the automation of fundamental database functions and design choices. Specifically, researchers have shown that indexes based on learned models that approximate the cumulative distribution function (CDF) [20]—which is effectively the rank function for items in a dataset —- are faster and smaller than traditional indexing data structures [25]. These learned structures and algorithms often outperform their traditional counterparts because they can accurately capture data trends and optimize performance for specific instances. A recent survey [3] identifies close to 100 proposed learned indexes in the last five years.

A recent benchmarking study [25] demonstrates that learned index structures such as the PGM-Index [14], RMI [20] and RadixSpline [18] can deliver superior lookup performance on sorted data compared to traditional indexes. Kristo et al. [22] show that learned models can speed up sorting in main memory for integer keys. Sabek and Kraska [29] demonstrate that learned models can improve in-memory indexed nested loop joins by $5-25\%$ and can improve the partitioning step in parallel hash and sort-joins.

While recent work on in-memory benchmarks has shown promising results, the benefits of learned indexes in the external memory settings do not appear to be as clear. Lan et al. [23] adapt multiple updatable learned indexes to external memory and evaluate performance on read, scan, and write workloads. They observe that in their current iterations, none of the updatable learned indexes are competitive with traditional B-trees. Zhang et al. [35] identify that directly extending learned indexes to disk environments often results in suboptimal performance compared to traditional B-trees. However, they show that employing optimizations such as leaf-page fetching strategies, prediction granularity, and adapting for disk characteristics can help bridge this gap.

Existing benchmarking studies involving learned indexes have not focused on external memory database joins, which are crucial for large-scale databases. External-memory joins offer vastly different trade-offs and challenges compared to in-memory joins. For relations stored in external memory, the dominating cost in performing the join is the I/O cost. Random I/Os are more expensive compared to sequential ones, resulting in different trade-off choices [34, 17, 27]. Additionally, different storage devices such as hard disks and SSDs offer different I/O bandwidths and latencies, leading to a different choice of indexing parameters to achieve optimal performance. Therefore, the insights gained from employing learned models for accelerating in-memory joins are relevant but not directly applicable in the context of external memory.

**This paper.** We investigate the applicability of learned indexes for external-memory joins, a common operation in large-scale databases. Through an extensive empirical evaluation, we assess whether the performance advantages of learned indexes in in-memory settings extend to external-memory environments.

To utilize learned indexes for external-memory joins (learned index-based join), we replace the traditional index in the indexed nested-loop join with a learned index and introduce optimizations tailored for external-memory settings to enhance performance. This approach aligns with the methodology used by Sabek and Kraska [29] in their study on in-memory learned joins. We conduct a comprehensive

*Northeastern University.
†Northeastern University.

empirical evaluation, comparing learned index-based join with various traditional external-memory join algorithms. Our study results in the following key contributions:

- We evaluate the performance of the indexed-nested loop join using learned indexes on real-world and synthetic sorted datasets.
- We evaluate learned indexes across a range of parameter settings, including table-size ratios (selectivity), concurrency, model accuracy, and types of storage devices (SSDs and HDDs).
- We evaluate the performance of learned indexes for joins on unsorted data to produce a sorted output by partitioning the data using the learned CDF model and evaluate its performance under memory pressure.
- We analyze experimental results and make several key observations that will benefit future work in incorporating learned indexes as part of the query execution engine in large-scale databases.

**Key Takeaways.** Here we list the key takeaways from our experimental evaluation:

- For external-memory joins, learned indexes can benefit from using higher errors, and in turn achieve lower memory usage without loss in join performance. Using a larger error window size for the learned model helps increase disk-bandwidth utilization for learned indexes when the join no longer does sequential I/Os, which can happen when the sizes of two sorted tables being joined differ by a large factor. This enables smaller indexes and faster operations.
- While learned indexes provide smaller index sizes and faster lookups, they do not reduce total I/O costs, leading to performance that is comparable to B-trees in external-memory joins.
- On SSDs, learned indexes improve the time to perform a join on sorted data compared to B-trees by $1.2-1.6\times$ when the input tables must be completely scanned. The improvement in performance on SSDs comes from faster CPU performance in querying for items.
- Learned indexes scale similarly to B-trees with increasing threads, but they become I/O-bound at high levels of parallelism.
- Learned indexes have an order-of-magnitude higher build time than B-trees when built on the entire dataset. Sampling keys to build the learned index enables faster index construction without a noticeable impact on the join performance. However, even with sampling, the learned index build time is $10\times$ larger than B-trees.

## 2 Background & Related work

In this section, we briefly describe the various learned index designs and their usage in database applications. We highlight the key insights and unanswered questions pertaining from these studies.

Kraska et al. [20] showed that machine learning models can help reduce the memory footprint and increase the performance of traditional indexing data structures such as B-trees, hash tables, and filters. More specific to databases, there is considerable interest in designing *learned indexes* for sorted data that deliver better performance with lowered space usage than B-trees that are commonly used to implement database indexes.

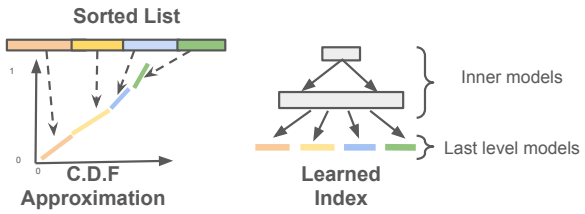



Figure 1: Modeling the CDF using piecewise linear approximation and indexing into the individual linear models.

Various learned index designs have been proposed, including read-only indexes such as the Recursive model index (RMI) [20], RadixSpline [18] and Piecewise geometric model (PGM) index [14] and updatable learned indexes such as ALEX [12], FITing Tree [15] and LIPP [33] as replacements for B-trees in static and dynamic workloads, respectively. For a more detailed history of the work in this area, please refer to [3] which extensively surveys learned indexes.

There has also been considerable research showing the applicability of learned models in other applications. Recent works include improvements to fundamental algorithms such as sorting and joins [22, 29], range index designs [12, 33, 14], query performance in log-structured merge-trees [11, 1], multi-dimensional and spatial indexes [24, 36] and genome sequencing [19, 16].

### 2.1 Learned indexes

In this section, we briefly describe various types of learned indexes.

**Range indexes are CDF models.** The key insight that ties machine learning techniques and deterministic problems such as indexing is that all range indexes model the cumulative distribution function (CDF) of the data they index [20]. The cumulative distribution function of a list is a function $F(x)$ that maps the probability that $x$ is greater than an item picked randomly from that list $L$. When the CDF value for an item $x$ is scaled by the number of items in the list, it returns the position where $x$ would fit in that list, i.e. $pos(x) = F(x).|L|$, where $|L|$ is the size of the list.

**CDF-based models.** Learned index implementations follow a common design trend in modelling the CDF. They often employ regression techniques (as opposed to deep neural networks) to model the CDF. As a single model is practically not enough to model the entire CDF, the CDF is modelled in a piecewise manner with a hierarchical structure indexing individual piecewise models. This is illustrated in Figure 1.

The CDF model only approximates the position of a key in the list, i.e., it cannot accurately determine the position of the key. However, this is not a significant issue in practice as the model error is usually bounded and a local *last-mile search* can be performed to return the exact answer to a query (Figure 2). The model error is a training parameter that trades accuracy for space usage, i.e., high-accuracy models require

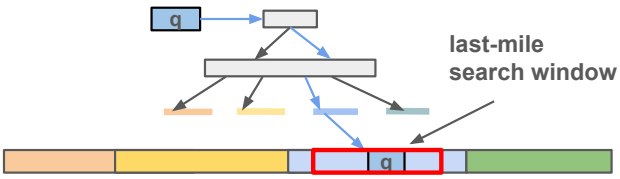



Figure 2: Query path using a learned index. The inner levels direct the query to the approximate location in the dataset. A last-mile search is performed to find the query key.

more space but a smaller last-mile search and vice versa.

**2.2 Learned index designs** Multiple learned index designs have been proposed, each using a unique modelling technique and data layout to implement the index. Below, we go over a few learned indexes that we use as candidates in our external-memory join implementation.

**Recursive model index (RMI).** RMI [20] builds a hierarchy of models on the dataset given a specification by the user. The user specification provides various parameters, such as the type of models and learning techniques to use, the maximum branching factor of each level, and the maximum size of the final model. To help with tuning, the RMI implementation provides an optimizer that performs a grid search of RMI models against the space/performance curve, allowing users to choose the RMI configuration appropriate for their use case.

**RadixSpline.** RadixSpline [18] is a single-pass learned index that uses a linear spline to model the CDF and employs a radix table to speed up the lookup of spline points. To perform a key search, the radix table is first consulted to return a range of linear spline segments. The returned linear spline segments are then used to find the error-bounded position of the lookup key. The RadixSpline is parametrizable in two parameters, the error-bound of the linear spline model and the number of the bits to use for the radix table. A larger error bound reduces the number of linear spline segments used to model the CDF but increases the last-mile search window. Similarly, using more bits for the radix table speeds up finding the correct linear spline model for a query at the cost of a larger radix table.

**Piecewise geometric model (PGM) index.** The PGM index [14] is a learned index that is similar to the RMI in that it builds a hierarchical structure of models. Each level is an error-bounded linear regression model built using piecewise linear approximation (PLA), i.e., the model is a piecewise function where each component is a linear function. The model can be visualized as a list of line segments that approximate a curve. The lowest level model approximates the CDF curve of the data, while higher levels are PLA models approximating the CDF curve of line-segment end points of the next level. Similar to the RadixSpline, the PGM is configurable by two parameters, the error bound and the maximum height of the index.

**Adaptive Learned Index (ALEX).** The aforementioned learned indexes are read-only indexes. ALEX [12] is an updatable learned index that augments the traditional B-tree nodes with learned models. ALEX maintains the invariant that the data inside a node follows the distribution of the learned model associated with the node. ALEX uses a cost model to help with decisions related to rebalancing the tree when new items are inserted or deleted. While ALEX is implemented as an in-memory data structure, we use the implementation adapted for disks by Lan et al. [23].

**2.3 Related work** In this section, we describe related research work evaluating learned indexes.

**Benchmarks for learned indexes.** The Search on Sorted Datasets (SOSD) benchmark [25] evaluates the lookup performance of various learned indexes in main memory on sorted data. Lan et al. [23] adapt multiple updatable learned indexes to external memory and evaluate performance on read, scan, and write workloads. They observe that in their current iterations, none of the updatable learned indexes are competitive with B-trees on all workloads.

**Learned indexes for in-memory joins and sorting.** Sabek and Kraska [29] show that learned indexes can improve the performance of in-memory joins. They show that learned indexes must be carefully adapted for each of the three main types of joins — hash, index-nested loop, and sort join — and using them as black-box replacements for traditional indexes is not optimal. For instance, they observe that index-nested loop joins benefit from removing the last-mile search by redistributing elements according to the learned model predictions. As another example, hash and sort join benefit from using the sampled CDF to uniformly partition the workload across different cores. Similarly, Kristo et al. [22] show that utilizing CDF approximations to recursively partition unsorted data into buckets results in a sorting algorithm that is $1.49-5\times$ faster than various state-of-the-art sorting algorithms.

## 3 Approach and Analysis

In this section, we describe and analyze the usage of learned indexes for external-memory joins. We describe the overall approach of the learned index-based join and then describe specific optimizations. Finally, we analyze the performance in I/O cost using the affine memory model [8].

**3.1 Approach** We use a learned index to support the indexed nested-loop join. During the join, keys from the smaller table are streamed, and each key is looked up in the learned index of the larger table to determine if it contributes to the join result (see Figure 3). Since the join is performed using indexes, the resulting output is naturally sorted.

For sorted tables, the learned index-based join uses two optimizations. The first optimization reduces the overall cost of the learned index look-up. The second optimization minimizes the size of the last mile search. We describe both optimizations below.

**Optimization 1.** The first optimization is to avoid traversing the entire height of a learned index for each query. Typically, a learned index query needs to start from the root and traverses

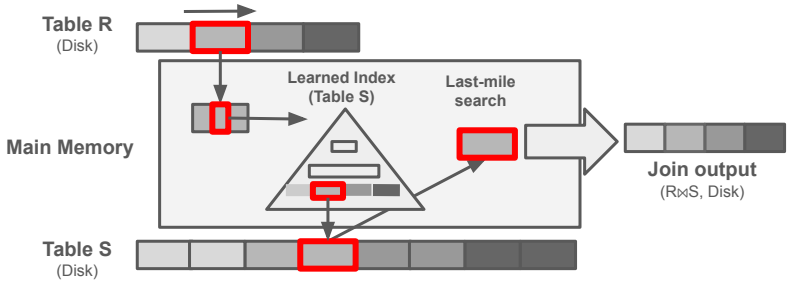



Figure 3: Illustration of the learned index-based join: Tables $R$ and $S$ (with $S$ being the larger table) reside on disk, while the learned index for $S$ is kept in memory. Pages of table $R$ are sequentially loaded into memory, and for each key, the learned index predicts the corresponding page in $S$ to perform the final lookup.

down to the leaf that contains the model assigned to handle queries for the query key (Figure 2). However, we can take advantage of the fact that queries increase monotonically during the join operation to reduce the query cost. Instead of traversing the entire height of the learned index for each query, we use a leaf node iterator that traverses the breadth of the last level. This iterator will advance to the next model when the query (which is the candidate join key) exceeds the range assigned to the model of the current leaf.

**Optimization 2.** The second optimization also takes advantage of the fact that queries during a join on sorted data occur in monotonically increasing order. In other words, the search window only advances forward. For example, if the lower bound for key $K_i$ is at $L_i$, and the learned index returned a search window $[L_j,H_j]$ for key $K_j$ where $j>i$, then the last-mile search for key $(K_j)$ can be constrained to $[\text{MAX}(L_i,L_j),\ H_j]$.

### 3.2 Analysis

We analyze the cost of using a learned index for indexed nested loop joins.

**Setup.** We perform the join on two input tables $R, S$ containing $|R|$ and $|S|$ number of keys respectively. We will assume that the tables are large enough such that their combined size exceeds main memory, and that a learned index for $S$ has already been built offline. We will also assume that the learned index fits in memory. Without loss of generality, we will assume that $|R|<|S|$. For a query key $q$, the learned index of table $S$ will return a search window for where $q$ might exist. The size of the window will never exceed $\epsilon$, the maximum error bound of the learned index, and is a training parameter for the learned index.

**The affine model.** We analyze the performance of the learned index-based join in the affine model [8]. Traditionally, the disk-access machine (DAM) [2] model has been used to evaluate the performance of external-memory algorithms and data structures in terms of the number of I/O transfers in the memory hierarchy. However, the DAM model does not assign a cost to each I/O. On HDDs, it does not model the faster speeds of sequential I/O versus random I/O. On SSDs, it does not model internal device parallelism or the incremental cost of larger I/Os.

The affine model [28, 5, 8] makes small refinements to the DAM model, but yields a surprisingly large improvement in predictability without sacrificing ease of use. The affine explicitly account for seeks (in spinning disks) by modelling the cost of an I/O of $k$ words as $1+\alpha k$, where $\alpha \ll 1$ is a hardware parameter.

**I/O cost analysis.** We will divide the I/O cost of the learned index into two components, one for reading $R$, and the other for reading $S$. Assuming $R$ is read in blocks of size $B$, the block transfer size - the cost of reading $R$ is $\frac{|R|}{B}$. For $S$, we assume that the learned index-based join will perform a single I/O of $\epsilon$ words for each query. The cost of I/O is then $O(|R|(1+\alpha\frac{\epsilon}{B}))$, where $\alpha$ is the hardware parameter according to the affine model. This cost is also bounded by $O(\frac{|S|}{\epsilon}(1+\alpha\frac{\epsilon}{B}))$ as we do not read any item in $S$ more than once. Hence, the I/O cost for reading $S$ is $O(\min(|R|,\frac{|S|}{\epsilon})(1+\alpha\frac{\epsilon}{B}))$. The overall I/O cost can now be summarized as

$$O\left(\frac{|R|}{B}+\min(|R|,\frac{|S|}{\epsilon})\cdot(1+\alpha\frac{\epsilon}{B})\right) \tag{3.1}$$

In the case of $|R|\leq\frac{|S|}{\epsilon}$, this analysis shows that increasing the size of the last-mile search window by building less accurate learned indexes does not significantly increase the overall I/O cost as $\alpha<<1$. When the table sizes are very similar ($|R|\geq\frac{|S|}{\epsilon}$), the I/O cost of the learned index is essentially the same as linearly scanning both tables. In this case, the affine model predicts that increasing $\epsilon$ actually decreases the I/O cost - we perform fewer but larger I/Os to completely scan $S$. Note that this analysis also holds for unsorted tables, the only difference being that the indexes are unclustered.

**Index size analysis.** While there are no tight bounds proven for the size of learned indexes, empirically they occupy less space than B-trees. Analyzing the size of learned indexes is not straightforward as it depends on the distribution of the data, the specific design of the learned index, and the training parameters used to build the learned index. The most critical training parameter common to all learned index designs is the size of the last-mile search window. Ferragina et al. [13] find that when the gaps between elements follow a distribution with finite mean and variance, the size of the learned index is $O(n/\epsilon^2)$, where $\epsilon$ is the size of the search window and $n$ is the number of keys.

In general, we can assume that learned indexes occupy less space than B-trees, and that the space decreases by more than a linear factor with the accuracy of the learned index. This is in contrast to B-trees whose size decreases linearly with the node size. We will assume that the size of the learned index is $O(n/f(\epsilon))$, where $f$ is a function that is polynomially greater than a linear function.

**3.3 Takeaways** We now compare the cost of using the learned index compared to ithe B-tree in the indexed nested-loop join (INLJ). We then compare the learned index-based join proposed above with the sort join. We compare these in terms of I/O cost, CPU cost, and index size.

**Learned index vs B-trees in INLJ.** Both indexes have the same I/O cost for the same search window size ($\epsilon$ for learned indexes, node size for B-trees). Increasing $\epsilon$ or the node size increases the I/O cost marginally ($\alpha << 1$) for both the indexes. However, learned indexes occupy less memory and are faster to query compared to B-trees. The learned index also has a better memory performance trade-off curve. Although the size of both the learned index and B-tree can be reduced by increasing the size of the search window, the size of the learned index decreases by more than a linear factor of $\epsilon$ compared to the B-tree size that only decreases linearly with the node size.

**Learned index-based INLJ vs sort join.** The comparison between the sort join and learned index-based join is similar to comparing the sort join with INLJ. The I/O cost for both methods depends on the size of the input tables. When the tables are of similar size, the I/O cost of both the join methods are equal. In this case, the differentiating factor of both the join methods is the CPU cost. The cost of the sort join is one key comparison per element from the larger table, resulting in a total CPU cost of $O(|S|)$. For the index-based approaches, the CPU cost is one index lookup per element from the smaller table, leading to a total CPU cost of $O(|R| \cdot \log|S|)$. When table sizes vary by a factor of more than $\epsilon$ (that is, $|R| \leq \frac{|S|}{\epsilon}$), the learned index-based join, similar to the index nested-loop join, costs less I/O and CPU compared to the sort join.

## 4 Implementation

We now describe our implementation of join methods. We first describe how our tables are stored on disk, followed by the join and merge implementations. We then describe our join implementation for unsorted data. In both cases, the tables are stored on disk.

**4.1 Tables** While fully featured database systems will use more complex data storage layouts and robust page management strategies, we intentionally choose a very simplistic layout to focus on the effects of learned indexes for joins.

**Storage Layout.** A table is a list of key-value tuples. The keys and values are of a fixed size of 8 byte keys and 8 byte values. The keys are distinct in a table. Tables are stored as a dense sorted array on disk in a single file. The first 16 bytes of the file are reserved for a header to store the number and size of the key-value tuples, followed by the key-value tuples themselves. We do not perform any compression of the actual key-value tuples.

**Reading and writing tables.** Tables are logically divided into blocks, each 4096 bytes in size. To read a key-value tuple from the table, the block corresponding to that key must be loaded into memory. The blocks are stored in an internal buffer that holds a contiguous set of blocks in main memory. We configure the internal buffer size to hold enough blocks so that the search window of the learned index can be fully held in memory. If a key that we are searching for is already in the internal buffer, we immediately return the key. Otherwise, the set of contiguous blocks starting with the block containing the key is loaded into memory. Similarly, when writing the output join table, we store keys in an internal buffer and flush them to disk once the buffer is full.

**4.2 Indexes** Indexes are built offline and serialized to disk. When used for a join, the indexes are loaded from disk and stored fully in main memory.

**Index API.** Given a query for a key, all indexes (learned or B-trees) return a search window whose size is bounded, representing the range in which the table contains the lower bound of the query key, which is the largest key in the table that is greater than or equal to the query key. More formally, a table $T$ is an array of $[K_1, K_2, ...K_N]$, such that $K_i > K_{i-1}$. The index is a function $I_T(x)$ that returns the pair $(lo, hi)$ such that $K_{lo} \geq x \leq K_{hi}$ and $(hi - lo) \leq \epsilon$, where $\epsilon$ is the error bound of the learned index.

**B-tree index.** We use the interpretation defined in [20], where the inner nodes of the B-tree are interpreted as a learned model and the leaf nodes are the search window ranges that these learned models ultimately return. When interpreted this way, the B-tree can be decomposed into two distinct entities - a learned index (inner nodes) and the data (leaf nodes). For the sorted tables, we only create the learned index part of the B-tree by evenly sampling keys (these are keys that would have been the first key of their leaf nodes) and insert them into a B-tree.

**Learned indexes.** We consider various learned index implementations (as described in Section 5.1) in our evaluation. We ensure that all learned indexes use the same search window size (except RMI, where this is not configurable). We discuss the exact configuration used for each learned index in the experimental setup (Section 5.1).

**Constructing learned indexes using sampling.** Constructing learned indexes on sorted arrays can be expensive due to the computational cost of modeling the data distribution. In contrast, building a B-tree is more efficient, as pivots can be directly selected from the sorted array.

To reduce the cost of learned index construction, we sample every $k^{\text{th}}$ element from the array and learn the distribution over this subset. We then build a learned index on the sampled array with error $\epsilon'$. A query returning the interval $[lo, hi]$ in the sampled array maps to the interval $[k \cdot lo, k \cdot hi]$ in the full array. This effectively gives a learned index on the original array with error $\epsilon = k \cdot \epsilon'$, constructed using only $n/k$ elements.

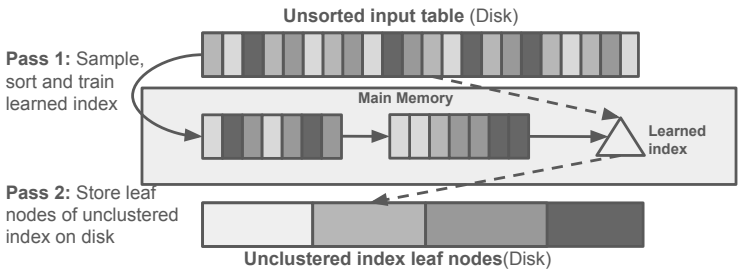



Figure 4: An unclustered index using a learned index is built in two passes: the first pass samples the data to train the learned index, and the second pass uses the index to assign each data item to its predicted position.

**Learned Index Lookup.** Using learned indexes to answer *lower_bound* queries is a two-step process. First, a lookup is performed on the index for the query key $Q$ in table $T$. The index will then return a range $[lo,hi]$, representing the range containing the actual lower bound. The second step is to perform a last mile search in the table to find the exact lower bound in the table.

**Last-mile search.** To perform the actual last-mile search, we use the branchless implementation of the binary search, resulting in a fast and efficient last-mile search. The branchless search algorithm takes advantage of the conditional move (CMOV) instruction to generate assembly code with no branches [30]. We first check if the search range returned by the index is partially loaded in our internal buffer. If the lower bound is not found or the internal buffer contains blocks that do not overlap with the search range, we load the required contiguous set of blocks from the disk into memory.

**4.3 Join on sorted tables** We now describe implementation details of the various join implementations. The join operation will output the common keys from the input tables $(R,S)$ as its own table to disk. We compare the indexed nested loop join with B-tree and learned index-based join, against each other and also with other join methods such as hash join and sort join.

**Indexed nested loop join.** Similar to the hash join, we start first by streaming all keys of $R$. For each key of $R$, we query the index of $S$ which will return a search window. We then do a last-mile search inside $S[lo,hi]$, outputting the key to the final table if it is present.

**Sort join.** If $R$,$S$ are already sorted, we skip the sort phase and proceed directly to the merge phase. We use a standard two-pointer approach to find the common keys. We initialize two iterators, comparing the iterator heads and advancing the iterator with the smaller key. If the keys are equal, the key is added to the output join table. The join ends if any of the iterators reaches the end of its table and cannot advance.

**Hash join.** We first iterate over all keys of $R$ and insert them into an in-memory hash table $H_R$. We use `std::unordered_map` as our in-memory hash table. We then iterate over all the elements from $S$, adding the key to the output if it exists in $H_R$.

**4.4 Join on unsorted tables** We now describe the implementation details of the join methods for unsorted tables. These methods will output the result of the join in sorted order. Here we only implement two variants of the indexed nested loop join, B-tree based and learned index based. We build unclustered indexes for both input tables as part of the join.

**B-tree index.** We first build a complete B-tree for each input table by sequentially inserting keys along with a pointer to its location in the table. We then perform a range scan on the B-tree of the smaller table to stream the keys in sorted order, checking if each key exists in the large table by querying the B-tree of the larger table. The keys which are common to both input tables are added to join table.

**Learned index.** To approximate the data distribution, we build a learned index on a set of sampled keys from the unsorted table. The learned index on the sampled subset acts as a model for the distribution of data in the entire dataset, as uniform sampling captures the overall data distribution and also avoids the need to fully sort the data.

We use the approximate rank of a key in the sampled learned index to partition all keys into the desired number of partitions. Each key $k$ is assigned to the partition $\frac{\hat{R}_k}{N}P$, where $\hat{R}_k$ is the approximate rank of $k$, $N$ is the number of keys in the table, and $P$ is the desired number of partitions. We choose $P$ so that the average partition fits into a small constant number of pages. In practice, if the keys were randomly sampled uniformly and the learned index is reasonably accurate, the partition sizes have low variance. As long as the $\hat{R}_k$ returned by the learned index is monotonic, the partitions will be disjoint. The partitions are essentially leaf nodes of an unclustered index, and the partitioning process is illustrated in Figure 4.

Partitioning the data according to the position returned by the CDF model is an idea that has been explored by Kristo et al. [22] in the context of in-memory sorting, where the data is sorted by recursively partitioning the data using the learned index. Similarly, Abu-Libdeh et al. [1] build tables that are sorted into blocks according to the position predicted by the learned index. We adapt this idea for joins on external memory by only partitioning the data once and not sorting the data inside the partitions. Our partitioning method uses only two passes on the data, one to sample

the keys and build the CDF and the other to assign the key to a bucket according to the built model.

After partitioning keys into buckets, we perform the indexed nested-loop join. We store the learned index and the partition map of both tables in memory. We sequentially load each bucket of $R$ into memory and process the keys in sorted order. For each key, we query the index of $S$ and load the corresponding bucket into memory. The key is added to the output if it is part of both tables.

## 5 Evaluation

| Dataset | Size | Key count | Description |
|---|---|---|---|
| FB | 3.2 GB | 200000000 | Facebook user ids |
| Wiki | 1.44 GB | 90437011 | Wikipedia edit timestamps |
| OSM | 12.8 GB | 800000000 | OpenStreetMap locations |
| Books | 12.8 GB | 800000000 | Amazon book popularity data. |
| udense | 3.2 GB | 200000000 | sequential integers |
| usparse | 3.2 GB | 200000000 | Uniform sparse distribution |
| normal | 3.2 GB | 200000000 | Normal distribution |
| lognormal | 3.2 GB | 200000000 | Lognormal distribution |

Table 1: Summary of datasets

In this section, we evaluate the usage of learned indexes for external-memory joins against traditional join algorithms. For learned index-based join, we employ the learned model to replace the index in indexed nested loop joins. For all indexes, we keep the leaf nodes on disk and the non-leaf nodes in main memory. First, we evaluate various learned indexes on construction time, in-memory space requirements and join performance when used as part of learned index-based join. We then pick the most appropriate learned index and compare the learned index-based join against traditional indexed nested loop joins, sort join (SJ) and hash join (HJ).

We evaluate the performance of the learned index-based join against traditional join algorithms across multiple dimensions: (1) storage device types (HDD/SSD), (2) data ordering (sorted/unsorted), (3) concurrency, (4) constrained memory settings and (5) trade off between error window size and number of threads.

All benchmarking source code and datasets used in our evaluation are available at `https://github.com/saltsystemslab/learnedjoindiskexp`.

### 5.1 Experimental Setup

In this section, we describe the experimental setup we use for our evaluation.

**Environment.** We run our experiments on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 1 NUMA node with 12 cores with a single 12MB L3 cache with 32GB of RAM. We run our experiments on both SSD and HDD. The SSD model used in our experiments is a 512GB SanDisk SD9SB8W5, while the HDD is 2TB TOSHIBA DT01ACA200.

**Datasets.** We use real-world and synthetic datasets used in the unified benchmarking paper by Marcus et al. [25].Table 1 summarizes various datasets.

- **Real-world datasets: Books** is a collection of 800 million keys representing book popularity data from amazon. **Wiki** is a collection of 90 million wikipedia edit timestamps. **OSM** is a collection of 800 million OpenStreetMap locations. **FB** is a collection of 200 million Facebook user ids.
- **Synthetic datasets:** All synthetic datasets contain 200M items generated from a universe of 64-bit unsigned integers. **usparse** represents a dataset of integers picked uniform randomly, while **udense** represents a dense distribution of sequential keys. **normal** and **lognormal** represent data distributions that follow normal and lognormal distributions, respectively.

**Join processing.** Each join operation is invoked as a new process. Our join evaluation setup for **static tables** is similar to previous join studies [4, 6, 7, 10, 31, 29, 21]. The indexes are constructed offline, stored on disk, and loaded into memory before starting the join operation. For multithreaded experiments, the input tables are partitioned evenly across threads and work is distributed evenly. All operations are performed with **cold cache** by dropping the operating system page caches, and the input files are read using `O_DIRECT` to ensure that all data is always retrieved from disk. We do not use `O_DIRECT` for writes as the output cost for all join methods is the same and increases the duration of the experiment. To constrain the memory during join process, we employ `CGroupsV2` utility.

### 5.2 Evaluating learned indexes for joins on disk

In this section, we evaluate the performance of PGM, RadixSpline, RMI, and ALEX against B-trees for index construction, query latency, and in-memory index space. We use the insights from this evaluation to identify the appropriate learned index to employ for external-memory joins on sorted and unsorted data.

Our evaluation of learned indexes extends the SOSD benchmark [25] for disk-based evaluation and performance on the join operation. Furthermore, our evaluation extends the disk-based benchmarking study [23] by evaluating the learned indexes on external memory joins.

**Indexes.** We use the B-tree index for implementing the index nested-loop join. We use the STX-BTree v0.9 [9] library as our B-tree implementation. For each of the learned indexes (PGM [32], RadixSpline [18] and the recursive model index (RMI) [20]), we use the reference implementations provided by the authors. For ALEX [12], we use the disk-based implementation used in the study by Lan et al. [23], which evaluates updatable learned indexes in external memory. The specific configuration used for building each index is described below:

- **B-tree**: B-tree node size is fixed to 4K bytes. To make the comparison fair with static learned indexes, the B-tree is built by bulk loading the keys. Furthermore, we build the B-tree on every $256^{\text{th}}$ key in the dataset. The leaf nodes map their keys to the block they come from in the dataset, where a block is a contiguous set of 256 keys. Thus, the leaf nodes of a B-tree return a search window of size 256, similar to how a learned index returns a search window for the last-mile search.

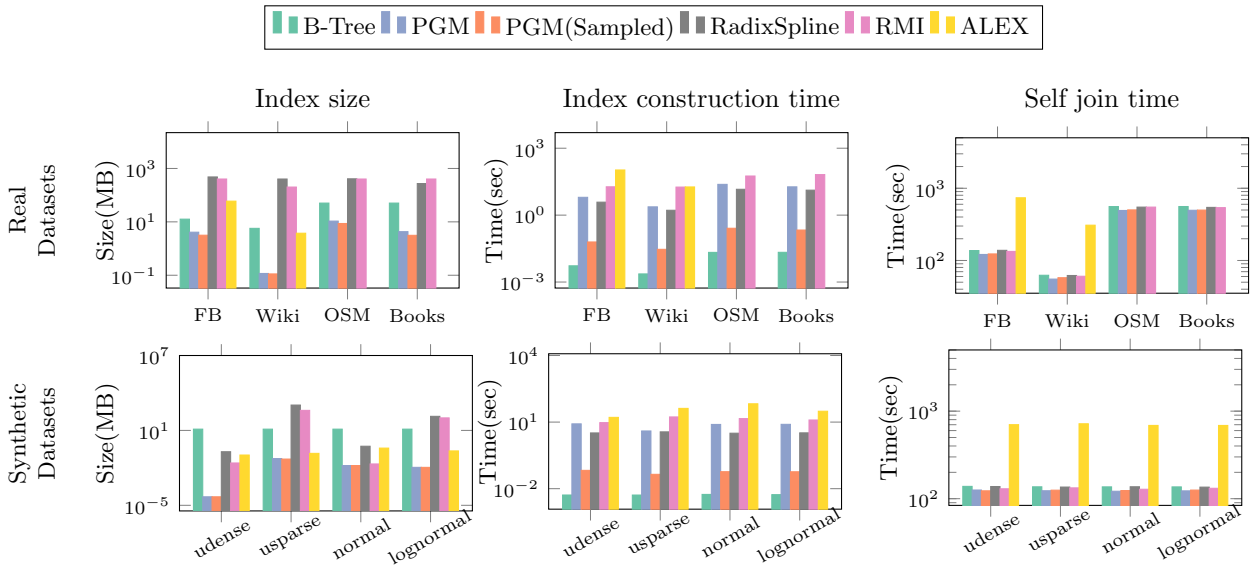



Figure 5: Index size, build time and performance of learned indexes and B-tree. Index Size is the space used by the index in main memory. ALEX did not finish building the index using 32GB of RAM for OSM/Books datasets.

- **Piecewise geometric model (PGM)**: The PGM index is built with an error window of 256 with a single level.
- **Piecewise geometric model (PGM) sampled**: A version of the PGM index that is built on every $128^{\text{th}}$ key with a maximum error of 2. The search window returned by this index is then scaled up to get the search window in the dataset.
- **RadixSpline (RS)**: For each dataset, we choose the *Pareto-optimal* index configuration of the RadixSpline as evaluated by the SOSD benchmark [25]. The *Pareto-optimal* index is an index configuration for which no other index with lower memory usage has better performance. Radix bits range from 16 to 28 across the various datasets. We also set the RadixSpline maximum error to 256.
- **Recursive model index**: For each dataset, we use the RMI model configuration from the SOSD benchmark [25] with hyperparameters tuned using CDFShop [26]. Similar to the RadixSpline, we choose the *Pareto-optimal* index for each dataset.
- **ALEX**: We use the disk-based implementation provided by Lan et al. [23] to evaluate the performance of updatable learned indexes on disk. The data nodes are stored as a contiguous file on disk, while the inner nodes are stored in main memory.

**Setup.** The construction time is measured as the time to build the index over the set of keys. We load all the keys into memory before building the index to avoid counting the disk I/O cost during index construction. All indexes, except RMI, can be built in a single pass by streaming keys from the disk. The query performance is evaluated by measuring the time taken to perform the index nested-loop self-join using the index. All indexes are loaded into main memory while the data is kept on disk and loaded one page at a time. The index evaluation experiments are constructed on flash storage with the operating system page caches flushed before the start of each experiment. Figure 5 plots the construction time, index size, and the time to complete the self join on all datasets using each index.

**Index size.** For real-world datasets, PGM indexes (both sampled and full) have the lowest memory footprint among all learned indexes, being $4\times$ smaller than the B-tree. The RadixSpline and RMI are an order of magnitude ($30-80\times$) larger than B-trees. ALEX was unable to construct the *OSM* and *Books* datasets as it ran out of memory. For synthetic datasets, PGM indexes were an order of magnitude smaller in size compared to B-trees. The RMI and RadixSpline are not able to model the *usparse* and *lognormal* distributions well and needed several orders of magnitude of space to do so.

**Index construction time.** All learned indexes take at least 4 orders of magnitude longer to build than B-trees. Constructing the PGM index with sampling reduces the duration by roughly two orders of magnitude. The higher construction times of learned indexes can be reduced by sampling without sacrificing query performance. As the time to construct the PGM and RadixSpline indexes grows linearly with the size of the dataset, evenly sampling keys to construct the index reduces the number of keys used to train the index while still effectively capturing the distribution of the data. The PGM index built on the sampled data has performance identical to that of the index built on the entire dataset. Additionally, the PGM index is also simpler to construct, requiring only a single parameter (the maximum error), compared to the RMI and RadixSpline, which require tuning multiple parameters to find the pareto-optimal configuration for space and performance.

Despite reducing the construction time using sampling, learned indexes are slower to construct than B-trees. The

B-tree construction is extremely fast when built using bulk loading because the B-tree only needs to perform fixed memory allocations and data copying. On the other hand, learned indexes need to perform more complex processing to model the distribution leading to increased construction time.

**Join performance.** Across all datasets, we find that the PGM index performs the most consistently, being $1.1-1.7\times$ faster than the B-tree. As disk I/O is the bottleneck, all learned indexes (except ALEX) performed very similarly. For ALEX, our results are consistent with the disk-resident learned index study by Lan et al. [23], which showed that the read performance of ALEX on disk is not competitive in read-heavy workloads.

**Takeaways.** Overall, the PGM index with sampling offers the best tradeoff in terms of query latency, construction time, and space usage. Using sampling, the join can be sped up by $1.1-1.7\times$ compared to B-trees and also uses $4\times$ less space. Although the PGM index takes $10\times$ longer to build compared to the B-tree, this is often an acceptable tradeoff in large-scale analytics systems where the indexes are built offline and are used several times to perform fast joins. **Therefore, we employ sampled PGM index in our implementation of the learned index-based join.**

**5.3 Join methods on sorted data** In this section, we compare the single-threaded performance of the hash join (HJ), indexed nested-loop join (INLJ), sort join (SJ), and learned index-based join on sorted tables stored on disk (both HDD and SSD) across varying table size ratios.

**Setup.** We use the sampled PGM index as the index for the learned index-based join based on the analysis presented in Section 5.2. The indexed nested-loop join uses a B-tree as the index. The sorting phase of sort join is skipped as the data is already sorted on disk. The hash join uses STL `std::unordered_map` as the hash table of the smaller table. All indexes and hash tables are loaded in memory before starting the join, while the data is streamed from disk in pages using file I/O. The join time experiments on SSD and HDD is plotted in Figure 6 and Figure 7, respectively.

**Join selectivity.** We employ different table ratios to evaluate join algorithms for different selectivity values. A table size ratio of 1 is a self-join. For other table size ratios (10, 100, and 1000), we sample a fraction of the keys uniformly randomly from the table to create the smaller table for the join.

**Index for indexed-nested loop join.** We employ ***sampled PGM index*** in our implementation of the learned index-based join. This is based on the conclusions drawn from an extensive study of learned indexes for construction time, size, and query time detailed in Section 5.2. A sampled version of the PGM index is built on every $128^{\text{th}}$ key with a max error of 2. The search window returned by this index is scaled up to get the actual search window in the dataset. We use the B-tree index to implement the index-nested loop join. We use the STX-BTree v0.9 [9] library as our B-tree implementation.

**Join method evaluation on SSDs.** The learned index-based join is faster by $1.2-1.6\times$ compared to the indexed nested-loop join with B-tree when the table size ratio lies between 1 and 100. The learned index-based join is also faster compared to the sort-join($1.1-1.4\times$) when the table size ratio is between 10 and 100. When the table size ratio is between 1 and 100, the I/O cost is identical for all methods as items from the table are fetched in blocks of size 4KB that contain 256 items. Thus, every block is expected to contain a join key. At higher table size ratios (such as 1000), both indexed-based joins, learned and B-tree based, perform random I/Os on the inner table. In our tests, the learned index-based join performed slightly worse than the B-tree based index nested-loop join by about ($1.1-1.6\times$). However, the performance is still very similar in absolute terms due to the small output size. **In cases where both tables have to be scanned completely, the learned index-based join offers better performance compared to the B-tree by $1.2-1.6\times$, and the sort join by $1.1-1.4\times$ (except when both input tables are of similar size).**

**Join method evaluation with HDDs.** On hard disks, the performance of the B-tree and learned index-based join was similar across all table-size ratios and datasets. Both the index-based methods were also faster than the sort-join except for when the table sizes were equal. **Indexed-based methods have similar performance on HDDs and are faster than sort-join except for when table sizes are similar.**

**Takeaways.** Learned indexes offer performance mostly similar to the traditional B-tree-based index nested-loop join in external memory settings. The PGM index is much smaller in size compared to the B-tree. However, that does not result in improved performance as the join operation is bounded by the disk I/O. Using learned indexes for join does not help in reducing the total I/O. This is especially true when there is enough working memory to store the B-tree. This is unlike in main-memory joins where learned indexes can help speed up join performance [29].

**5.4 Join methods on unsorted data** In this section, we compare the single-threaded performance of various joins on unsorted tables to produce a sorted join output in external memory.

**Setup.** We generate input tables for a dataset by shuffling keys from the FB dataset and storing them on disk. We compute the join using an indexed nested-loop join using unclustered indexes (B-tree, PGM) on the keys. The index stores keys and a pointer to its table entry. We use the pointer to fetch the associated value of a join key from the table on disk. We run the test under different memory constraints using `CGroupsV2` to limit the amount of memory that a process is allowed to use and plot the time to complete the join for the B-tree and PGM index in Section 5.3. Similar to joins on sorted data, we test for different table size ratios of the input tables. We summarize the index construction and implementation of the join for each index.

- **B-tree**: We build the B-tree by streaming keys from disk. The leaf nodes of the B-tree are stored on disk, while the

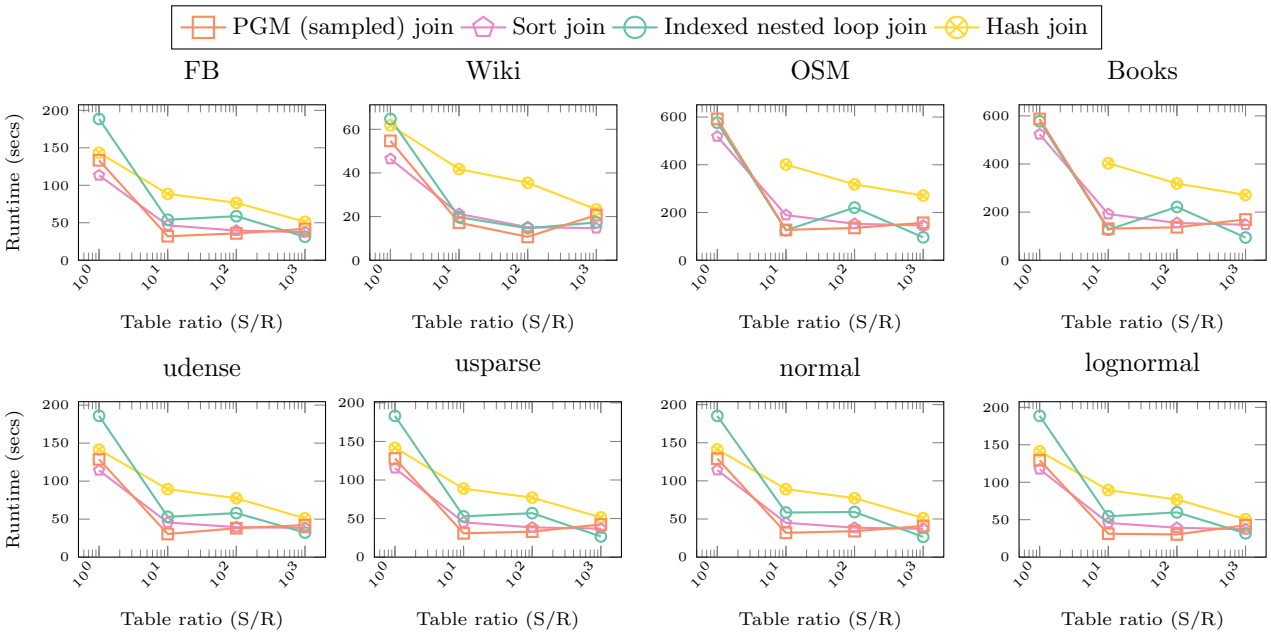



Figure 6: Performance of various join methods for sorted tables on flash-based storage devices (SSD).

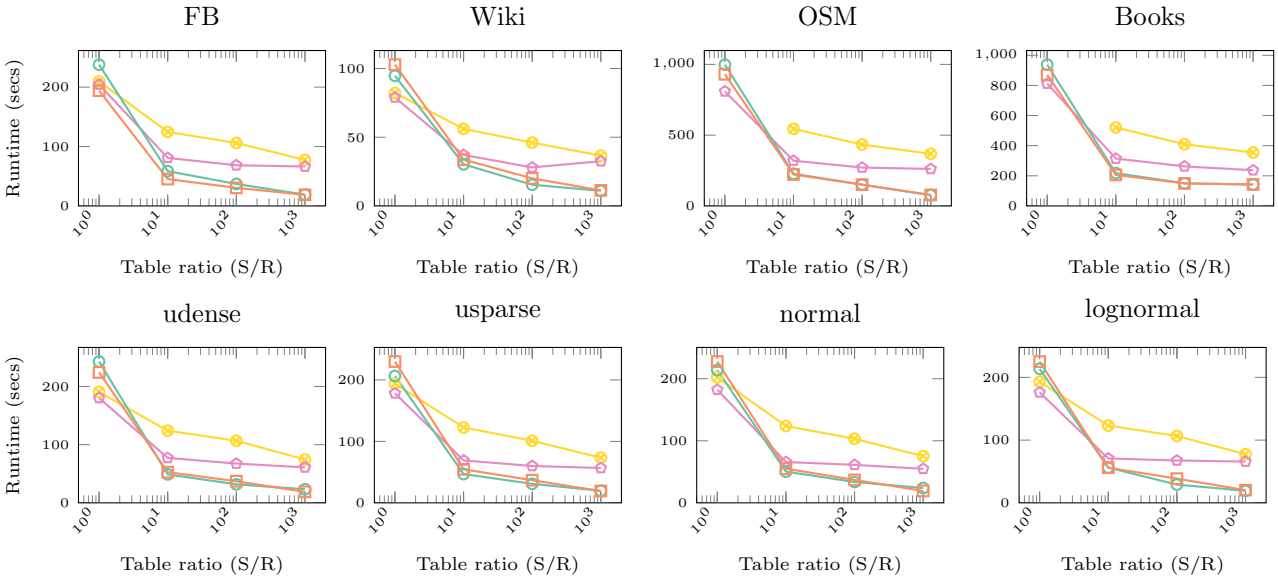
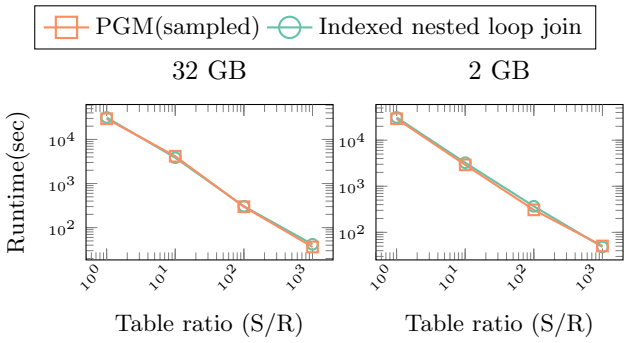



Figure 7: Performance of various join methods for sorted tables on hard disks (HDD).

intermediate nodes are held in memory. Nodes are 4KB in size. To compute the join, we scan the keys from the smaller table and for each key perform a lookup in the B-tree of the larger table.

- **PGM Index**: As the tables are not sorted, we partition the data into disjoint ranges with the help of a PGM index built on a sampled subset (1%) of the dataset. The rank of a key according to the sampled PGM index is used to determine its partition. As the rank returned is only an approximation, it is not necessary that partitions are equally sized. The more accurate the learned index, the less the variance in the size of the partitions. We set the expected partition size to be that of a single page (4KB). Partitions are flushed to disk 8 keys at a time to improve write efficiency. The index and partition map for each table are stored in main memory, and during the join only a single partition per table is kept in memory. To compute the join, we sequentially process the partitions of the smaller table. For each key in the partition, we use the PGM index of the larger table to determine the corresponding




(a) Join duration on unsorted data after index creation under different memory constaints using the FB dataset.

| | Index creation (sec) | |
|---|---|---|
| Memory Limit | B-tree | PGM |
| 2GB | 10777.06 | 2993.49 |
| 32GB | 1406.33 | 411.28 |

(b) Time to create the index on unsorted data under memory constraints using the FB dataset.

Figure 8: Join performance on unsorted data

partition in the larger table. Our approach for learned joins on unsorted data is based on learned sorting [22].

**Results summary.** The performance of different join approaches is largely similar. Joins on unsorted data incur more I/O operations than those on sorted tables. Since I/O cost dominates the join performance, faster queries of the learned index-based join do not yield significant improvements. The PGM index also performs similarly to the B-tree when memory is constrained using `CGroupsV2`. Although the PGM index is significantly smaller at 3MB compared to 12MB for B-tree, this 75% reduction in index size is insufficient to reduce the I/O performed to swap pages to disk due to constrained memory. However, partitioning the data to partially sort the data using the PGM index is faster ($3-3.5\times$) compared to constructing a B-tree with random inserts. This is due to the higher write amplification of B-tree to keep items sorted under random inserts.

**Takeaways.** Although partitioning and constructing an unclustered index for a unsorted table using the CDF model is upto $3.5\times$ faster compared to building the B-tree index, the join itself performs similar. The benefits of smaller indexes and faster queries are not apparent, as the memory savings of the learned index-based join are only over the inner nodes of B-tree, which is tiny compared to the space required to store the dataset. **The benefits of smaller indexes and faster queries on the join itself are not apparent even when operating under constrained memory settings.**

**5.5 Multithreading** In this section, we evaluate the effect of scaling up external-memory join methods using multiple threads. We will further study the performance

| | Index size (MB) | | |
|---|---|---|---|
| Dataset | $\epsilon=256$ | $\epsilon=2048$ | $\epsilon=4096$ |
| FB | 3.1623 | 0.7314 | 0.0910 |
| Wiki | 0.1121 | 0.0507 | 0.0064 |
| OSM | 8.6735 | 2.4492 | 0.6454 |
| Books | 3.140 | 0.092 | 0.024 |
| udense | 48 KB | 48 KB | 48 KB |
| usparse | 0.0505 | 0.0034 | 0.0002 |
| normal | 0.0109 | 0.0054 | 0.0027 |
| lognormal | 0.0152 | 0.0076 | 0.0038 |

Table 2: Index size of PGM index as search window size varies.

tradeoff between search window size and number of threads.

**Setup.** We partition the smaller table into equal size partitions based on the number of threads and assign a single partition to each thread. We build an index on the larger table and query it for keys from the smaller table to find a match for the join. Each thread writes its output to a separate file on disk. The threads are synchronized to block until all threads finish writing their join output. Once all threads are finished, we merge the output file for the final join output. To do this, each thread computes the offset of where its output lies in the merged output and writes it to the final join output. We test for thread sizes of 1, 2, 4, 6, and 8. We compare the learned index-based join with the indexed nested-loop join with B-tree, hash join, and sort join and plot the results in Figure 9.

**Results summary.** Adding more threads makes the join operation more I/O bound. Both the learned index-based join and B-tree indexed joins scale linearly with increasing threads before becoming I/O bound at some point. For example, when the table size ratio is 100, the learned index-based join becomes completely I/O bound with 4 threads. At this point, adding more threads does not improve the overall process as the join is I/O bound. The performance of sort join does not scale with more threads as it is already I/O bound. The hash join is mostly CPU bound and almost linearly scales with increasing threads all the way up to 8 threads. The runtime does not include the time to build the hash table. Thus, the time for hash join measurement avoids the time to perform I/O on the smaller table. This makes the hash join less I/O bound compared to the other joins. Note that the hash join uses much more memory compared to indexed and sort joins as it stores a hash table of size $O(|R|)$ in memory. It is only included in the evaluation only as a baseline for comparison.

**Takeaway.** The conclusions made in the previous section regarding which join method to use at different table ratios hold true even for multithreaded join processing. **Learned indexes scale up with more threads similarly compared to other join methods.**

**5.6 Error window size analysis** In this section, we study the effect of the size of the error window on the join performance with an increasing number of threads. We also study how the build time and index size varies as the error

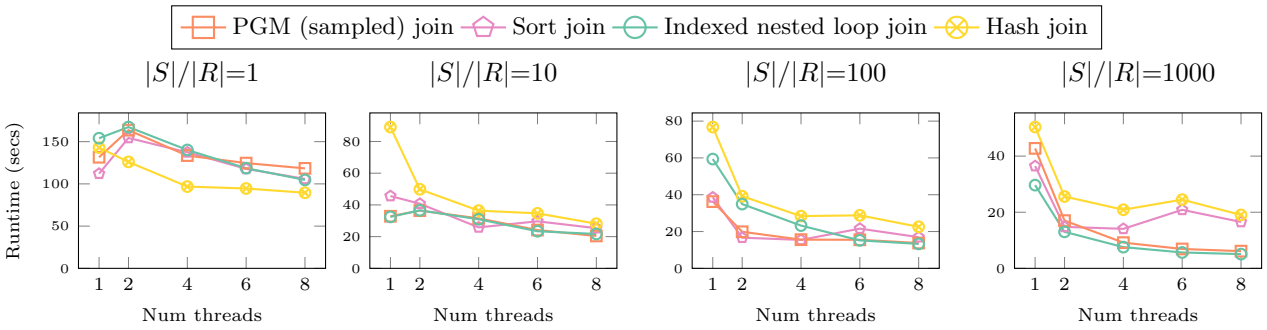



Figure 9: Performance of join methods as the number of threads increase (FB dataset)

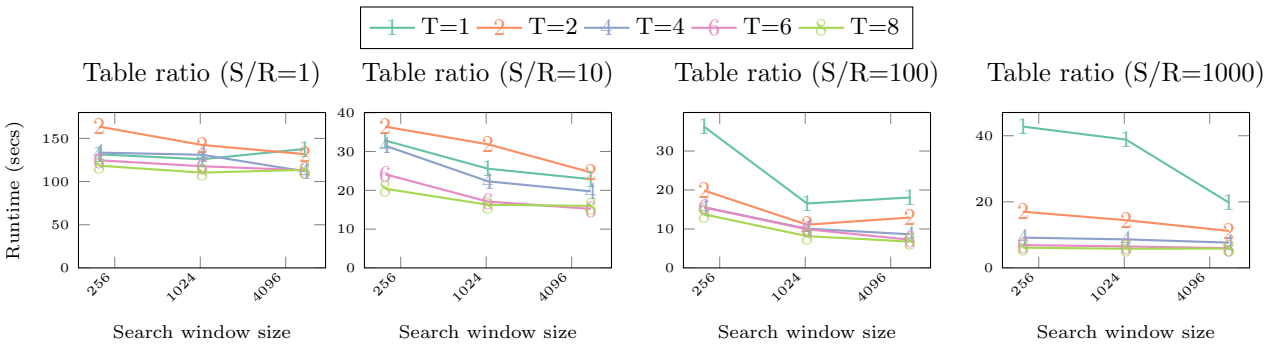



Figure 10: Performance of learned index-based join with various search window sizes (FB dataset)

window changes.

**Setup.** We use the sampled PGM index with error window sizes of 256, 1024 and 4096. The results of this experiment are plotted in Figure 10, while Table 2 shows the index size of the sampled PGM. We increase the page fetch size of a single I/O call to match the size of the error window. Note that this does not change the total number of bytes fetched from disk, only the number of I/O calls performed to fetch those bytes.

**Result Summary.** For the sampled PGM index, the index size is reduced by a factor of $30-130\times$ when the error window size is increased from 256 to 2048 and 4096 respectively. The index build time is independent of the index error window size and depends only on the sampling rate (which is fixed at 128 in this case).

Figure 10 shows that the performance of the PGM index remains consistent with increasing the size of the search window. We perform I/Os of larger block sizes to ensure that we perform no more than a single I/O call per query. For lower table size ratios, performance remains consistent with increasing the search window size from 256 to 4096. At these table size ratios, the join disk access pattern is sequential. Thus, requesting larger I/O block sizes across a varying number of threads does not have a significant effect on overall performance. As the table size ratios increase, the join access pattern is no longer sequential. When run with a low number of threads, a larger search window and I/O block fetch sizes lead to higher disk bandwidth utilization, resulting in disk saturation and consequently better performance. With a high number of threads, the disk utilization is already high and performing larger I/O block fetches has no effect on performance.

## 6 Conclusion

This study presents an extensive evaluation of learned indexes for external-memory joins, analyzing their impact across varoius database configurations. Unlike the main-memory setting, where learned indexes provide clear advantages, our findings indicate that their benefits in external-memory joins are less pronounced due to I/O dominance.

While learned indexes offer smaller index sizes and faster lookups, they do not reduce the total I/O costs, resulting in similar performance to B-trees-based joins in most cases. However, by tuning parameters such as search window size and error bounds, learned indexes can achieve comparable or slightly better performance in specific workloads, particularly on SSDs. The significant index construction overhead (up to $1000\times$ slower than B-trees) further limits their practicality for dynamic workloads, but remains acceptable in offline analytics scenarios.

Our results suggest that practitioners must carefully assess I/O constraints when integrating learned indexes into database engines.

## Acknowledgments

This research is funded in part by NSF grant OAC 2339521 and 2517201.

## References


[1] Hussam Abu-Libdeh, Deniz Altinbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. Learned indexes for a google-scale disk-based database. *CoRR*, abs/2012.12501, 2020.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. A survey of learned indexes for the multi-dimensional space. *CoRR*, abs/2403.06456, 2024.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, 2012.

[5] Matthew Andrews, Michael A. Bender, and Lisa Zhang. New algorithms for the disk scheduling problem. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 550–559. IEEE Computer Society, 1996.

[6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013.

[7] Maximilian Bandle, Jana Giceva, and Thomas Neumann. To partition, or not to partition, that is the join question in a real system. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 168–180. ACM, 2021.

[8] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara Mcallister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. External-memory dictionaries in the affine and pdam models. *ACM Trans. Parallel Comput.*, 8(3), sep 2021.

[9] Timo Bingmann. STX B+ Tree C++ Template Classes v0.9, November 2023.

[10] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48. ACM, 2011.

[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 155–171. USENIX Association, 2020.

[12] 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 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 969–984, New York, NY, USA, May 2020. Association for Computing Machinery.

[13] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theoretical Computer Science*, 871:107–120, June 2021.

[14] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, April 2020.

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, June 2019. Association for Computing Machinery.

[16] Darryl Ho, Jialin Ding, Sanchit Misra, Nesime Tatbul, Vikram Nathan, Vasimuddin Md, and Tim Kraska. LISA: towards learned DNA sequence search. *CoRR*, abs/1910.04728, 2019.

[17] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 301–315. USENIX Association, 2015.

[18] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, pages 1–5, New York, NY, USA, June 2020. Association for Computing Machinery.

[19] Melanie Kirsche, Arun Das, and Michael C. Schatz. Sapling: accelerating suffix array queries with learned data models. *Bioinform.*, 37(6):744–749, 2021.

[20] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, May 2018. Association for Computing Machinery.

[21] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(11):2132–2145, 2020.

[22] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1001–1016. ACM, 2020.

[23] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proceedings of the ACM on Management of Data*, 1(2):139:1–139:22, June 2023.

[24] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A learned index structure for spatial data. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Ab-

dussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2119–2133. ACM, 2020.

[25] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[26] Ryan Marcus, Emily Zhang, and Tim Kraska. Cdfshop: Exploring and optimizing learned index structures. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2789–2792. ACM, 2020.

[27] Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry, Martin Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. Timely reporting of heavy hitters using external memory. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1431–1446. ACM, 2020.

[28] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[29] Ibrahim Sabek and Tim Kraska. The case for learned in-memory joins. *Proc. VLDB Endow.*, 16(7):1749–1762, mar 2023.

[30] Malte Skarupke. Beautiful Branchless Binary Search, 2024.

[31] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. Distributed GPU joins on fast rdma-capable networks. *Proc. ACM Manag. Data*, 1(1):29:1–29:26, 2023.

[32] Giorgio Vinciguerra. gvinciguerra/PGM-index, November 2023. original-date: 2019-10-18T11:48:12Z.

[33] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment*, 14(8):1276–1288, April 2021.

[34] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *ACM Trans. Storage*, 13(1):3:1–3:26, 2017.

[35] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. Making in-memory learned indexes efficient on disk. *Proc. ACM Manag. Data*, 2(3):151, 2024.

[36] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. SPRIG: A learned spatial index for range and knn queries. In Erik Hoel, Dev Oliver, Raymond Chi-Wing Wong, and Ahmed Eldawy, editors, *Proceedings of the 17th International Symposium on Spatial and Temporal Databases, SSTD 2021, Virtual Event, USA, August 23-25, 2021*, pages 96–105. ACM, 2021.