The Case for Learned In-Memory Joins [Experiments, Analysis & Benchmark]

Ibrahim Sabek
MIT CSAIL
sabek@mit.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

ABSTRACT

In-memory join is an essential operator in any database engine. It has been extensively investigated in the database literature. In this paper, we study whether exploiting the CDF-based learned models to boost the join performance is practical. To the best of our knowledge, we are the first to fill this gap. We investigate the usage of CDF-based models and learned indexes (e.g., Recursive Model Index (RMI) and RadixSpline) in the three join categories; indexed nested loop join (INLJ), sort-based joins (SJ) and hash-based joins (HJ). Our study shows that there is room to improve the performance of the three join categories through our proposed optimized learned variants. Our experimental analysis showed that these optimized learned variants outperform the state-of-the-art techniques in many scenarios and with different datasets.

PVLDB Reference Format:
Ibrahim Sabek and Tim Kraska. The Case for Learned In-Memory Joins. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ibrahimsabek/learned-joins.

1 INTRODUCTION

The recent advancement of hardware, with increasing main memory capacities and providing a large number of cores, has led to the emergence of in-memory database systems (e.g., Umbra [47], HyPer [22], Quickstep[50], MonetDB [20]), and a lot of research efforts in developing highly-optimized in-memory variants of the core database operators (e.g., [66]). As a fundamental operation, in-memory join has been extensively investigated in the database literature during the last few decades (e.g., [2, 5, 8, 10, 17, 19, 31, 61]). Many recent studies (e.g., [5, 57]) have shown that the design, as well as the implementation details, have a substantial impact on the join performance on modern hardware.

Meanwhile, during the recent years, machine learning started to have a profound impact on automating the core database functionality and design decisions. There has been growing interest in exploiting learned models, such as CDF-based models [28, 29] and Recursive Model Indexes (RMI) [28], to enhance or replace traditional data structures and algorithms, such as indexing [14, 16, 25, 28, 46, 49], sorting [29, 30], and hashing [27, 55]. These learned data structures and algorithms can outperform their traditional counterparts as they explicitly capture trends in the underlying data and instance-optimize the performance. For example, in a recent benchmarking study [41], it has been shown that learned index structures (e.g., RMI [28], RadixSpline [25]), which employ CDF-based learned models, can outperform traditional indexes on practical workloads.

Along this line of research, one idea the authors of [28] introduced is using CDF-based learned models to improve the performance of in-memory join algorithms. However, this idea was neither thoroughly investigated nor supported with experimental evidence by the authors. Surprisingly, though, we are not aware of a thorough study examining the performance of learned join variants against traditional ones. We aim to remedy that here.

In this paper, we study, in detail, whether exploiting the CDF-based learned models to boost the performance of in-memory joins is a beneficial idea or not. In particular, we investigate the performance of three main join categories; indexed nested loop join (INLJ), e.g., [17, 19], hash-based joins (HJ), e.g., [8, 10, 31, 61], and sort-based joins (SJ), e.g., [2, 5], while modifying or replacing their different phases (e.g., indexing, sorting, joining) with CDF-based and RMI-based variants. In our study, we explore all the possible opportunities for integrating the learned models with traditional joins. Our paper has the following main contributions:

Investigating Alternatives of Using Learned Models for Joins. Although the learned model ideas have been introduced in previous works, none of them has been studied in the context of join processing. Therefore, for each join category, we first discuss the straightforward alternatives of directly integrating learned models with the join phases. For example, in INLJ, RMI can be used to replace the built index on the indexed relation. In SJ, the LearnedSort algorithm [29] can be used to replace the sorting phase. In HJ, a CDF-based model can be used to replace the hash function that is used to build the hash table. While investigating the different alternatives, we found an interesting observation: using learned models "as-is" in replacing phases of different join algorithms is sub-optimal. For example, directly using RMI instead of a traditional index in INLJ leads to poor performance (i.e., high overall latency) as RMI requires significant overhead to correct its mispredictions. Another example is using the LearnedSort algorithm [29] as a black-box replacement for the sorting algorithms used in the state-of-the-art SJ techniques, such as MPSM [2] and MWAY [61]. This, unfortunately, leads to repeating unnecessary work and hence increases the overhead of the SJ algorithm.

Optimized Learned Join Variants. To overcome the performance limitations of using learned models as black boxes, we introduce optimized variants of the learned join algorithms in the three join

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

categories, which lead to several performance improvements. In particular, we introduced five INLJ, three HJ, and six SJ learned variants that exploit different optimizations including hierarchical buffering, prefetching-optimized inter-task parallelism (e.g., AMAC [26]), software write-combine buffers (SWWC), non-temporal streaming, NUMA-awareness, and work sharing.

Extensive Experimental Evaluation. To achieve a deeper understanding of the practicality of using learned models with joins, we conducted a detailed evaluation study for both learned and non-learned variants in the three join categories, using different real (SOSD [41]) and synthetic datasets[1]. In particular, we compare 14 optimized learned join variants against seven INLJ baselines (three off-shelf learned (RMI [28], RadixSpline [25], ALEX [14]), two tree-based (Cache-Sensitive Search Trees [19], Adaptive Radix Trie [33]), and two hash-based (bucket chaining and Cuckoo hash) indexes), two SJ baselines (MPSM [2], MWAY [5]), and two HJ baselines (non-partitioned [61], partitioned [57]). Our study also provides a deeper analysis for the joins under different dataset sizes, skewness, tuple sizes, duplicates ratio, parallelism, strategies for shuffling data between NUMA nodes, and via different performance counters (e.g., cache misses and branch misses). The benchmarking code of this study is open-source[2] and can be extended by researchers and practitioners to improve the proposed learned in-memory join variants.

2 BACKGROUND

CDF Models. According to statistics, the *Cumulative Distribution Function (CDF)* of an input key x is the proportion of keys less than x in a sorted array A. Given a sample of keys, we can build a model that estimates the CDF of the distribution from which these keys are sampled. We refer to this model as *learned CDF model*.

For input keys with challenging distributions (i.e., complex CDF), a learned model typically consists of a set of *submodels*, where each submodel (e.g., linear regression) estimates a part of the CDF. These submodels are trained hierarchically across different K layers, as shown later. To predict the CDF of a key, the root submodel yields an initial CDF estimate. Based on this estimate, a submodel is chosen in the next layer for refining such estimate. This process is continued iteratively until the final estimate of the leaf submodel in the last layer is obtained. Assuming that each layer i has M_i submodels, where $0 \leq i \leq K$, and the j-th submodel at the i-th layer is f_i^j (the first layer has a single root submodel f_0^0), the CDF estimate $f_i(x)$ of key x at any non-root layer i (i.e., $0 < i < K$) can be recursively defined as follows:

$$f_i(x) = f_i^{\lfloor M_i \cdot f_i(x-1)/N \rfloor}(x) \qquad (1)$$

where N is the total size of the keys sample used to build the CDF model, and the estimate at the root layer $f_0(x)$ is $f_0^0(x)$. The final CDF prediction of x using the model is $CDF_{pred}(x) = f_{K-1}(x)$.

Algorithm 1 shows how to train a learned CDF model using a sample of keys S. We start by training the root submodel using all keys in the sample after sorting it (Line 5). Then, the keys are assigned to the next layer submodels based on the root submodel's estimates using Equation 1 (Lines 7-9). We proceed by training the submodels of the next layer on the keys that were assigned to them. This process is repeated for each layer until the last layer has been trained. Note that we build each submodel using a simple linear regression (i.e., no complex neural networks). Similar to [28], all submodels are end-to-end trained by minimizing a loss function. Let $(x, y) \in T$ be the training set, where each item in this set is a pair of a sample key x and its actual CDF value y after sorting the sample. The parameters of submodels in all layers are adjusted to minimize the squared error $\sum_{(x,y)\in T}(CDF_{pred}(x) - y)^2$. Finally, error bounds can be computed on trained leaf submodels (Line 10), if the learned CDF model will be used to build a learned index as shown later.

Algorithm 1 Function TrainCDFModel (KeysSample S, LayersNum K, Output *submodels*, Output *errorBounds*)

```
1: $submodelsKeys \leftarrow$ Initialize2DKeysArray ()
2: $submodelsKeys[0,0] \leftarrow$ Sort ($S$)
3: for $i \leftarrow 0$ to $K - 1$ do serially
4:     for $j \leftarrow 0$ to $M_i - 1$ do in parallel /*$M_i$ is the number of submodels at layer $i$ */
5:         $submodels[i,j] \leftarrow$ BuildSubmodel ($submodelsKeys[i,j]$)
6:         If $i < K - 1$ then
7:             for all $x$ in $submodelsKeys[i,j]$ do in parallel
8:                 $p \leftarrow$ GetNextSubmodel ($x$, $submodels[i,j]$, $M_{i+1}$, $|S|$) /*Eq 1*/
9:                 $submodelsKeys[i+1,p] \leftarrow$ AddKey ($x$)
10: $errorBounds \leftarrow$ CalcErrorBoundsForLeafSubmodels ($submodels$)
```

CDF-based Partitioning. CDF can be used to perform logical partitioning of input keys (e.g., [28, 29]) as follows: given an input key x, the partition index of each key x is determined as $CDF_{pred}(x) * |P|$, where $|P|$ is the number of partitions. Such CDF-based partitioning can be done in a single pass over the data. Although keys are not sorted within each output partition, the partitions are still relatively sorted. The same CDF can be used to recursively partition the keys within each partition by just scaling up the number of partitions $|P|$.

Learned Indexes. Learned indexes [28] exploit the CDF model to predict the position of the lookup key in the sorted input keys. In this study, we explore the following three representative learned indexes; RMI [28], RadixSpline [25] and ALEX [14][3]:

RMI is a read-only learned index that builds a CDF model using the whole input keys, not a sample, and rarely has more than two layers (i.e., levels) when the input keys fit into memory. Since the RMI is built using an approximated CDF model, the final prediction at the leaf level could be inaccurate yet error-bounded. Therefore, RMI keeps min and max error bounds for each leaf-level submodel to perform a local search within these bounds (e.g., binary search) and obtain the exact position of the key.

RadixSpline is another read-only learned variant consisting of a linear spline to approximate the CDF and a radix lookup table that indexes resulting spline points. Compared to RMI, RadixSpline can be built in a single pass with constant cost per key. Lookups first consult the radix table, which indexes r-bit prefixes of spline points and is used to narrow the search range over the spline points. Then binary search is used on the narrowed range to identify the two spline points surrounding the lookup key. Finally, linear interpolation between the two spline points is used to obtain a prediction.

[1] Similar to previous studies [7, 8, 57], we focus only on equi-join SELECT queries.

[2] https://github.com/ibrahimsabek/learned-joins

[3] We didn't include PGM [16] in our study since it has very slow lookups compared to other learned indexes, as shown in [41]. We also didn't include LIPP [64] as it is dominated by ALEX in multi-threaded environments as shown in [63].

ALEX is an updatable learned index (i.e., can support updates, insertions, and deletes). It utilizes a B-tree-like data structure with fixed internal (i.e., non-leaf) nodes and dynamic leaf ones with gapped arrays to support updates. It recursively partitions the key space through the tree levels, and a partition is represented by a slot in an internal node. A submodel in each internal node is used, during both lookup and update queries, to decide which partition a key belongs to. ALEX uses model-based insertion when building the index (i.e., placing a key at the position where the model predicts that the key should be) and hence reduces the model misprediction during lookups. During lookup, it applies exponential search on the gapped array in the leaf node to correct the model mispredictions, which are already few.

3 LEARNED INDEXED NESTED LOOP JOIN

Indexed nested loop join (INLJ) is the most basic type of join. Here, we assume that one input relation, say R, has an index on the join key. Then, the algorithm simply scans each key in the second relation, say S, uses the index to fetch the tuples from R with the same join key, and finally performs the join checks.

In this section, we start by describing the default multi-core INLJ algorithm and its optimized variants (Section 3.1). Then, we propose different INLJ variants that exploit the learned indexes (Section 3.2).

3.1 Representative Algorithms

The straightforward algorithm first chunks the non-indexed relation S into equi-sized chunks and assigns them to the different worker threads. Then, in parallel, each worker iteratively queries the global index of R (which is shared among all workers) with the keys in its chunk of S to return the tuples with the same join key and finally checks for the join matches. However, this algorithm is slow, whether the index used is hash-based (e.g., [59]) or tree-based (e.g., [19]), due to its key-at-a-time processing on each worker that can not hide the cache misses and will keep the worker idle while handling each miss.

3.1.1 *Optimized Hash-based INLJ.*

To hide the cache misses for hash-based INLJ, we employ a prefetching-optimized inter-task parallelism technique, namely Asynchronous Memory Access Chaining AMAC [26], during the index querying[4]. With AMAC, when an index lookup issues a memory prefetching, the algorithm switches to other query lookups in the pipeline to keep busy (i.e., avoiding key-at-a-time processing), and then returns to process the prefetched data after a while. In our study, we investigate two INLJ variants based on prefetching-optimized bucket chaining [7] and cuckoo [59] hashing, referred to as CHAIN-INLJ and CUCKOO-INLJ, respectively.

3.1.2 *Optimized Tree-based INLJ.*

For tree-based INLJ, instead of the key-at-a-time processing, we adapt an efficient INLJ variant that exploits *hierarchical buffering* [17, 67] to improve the temporal locality of the tree index. It temporarily stores the requests for query keys that will probably be accessed from the same part in the tree (i.e., keys that traverse the same nodes across the tree levels) and then answers them as a batch. This significantly reduces cache misses and, in turn, increases the join throughput.

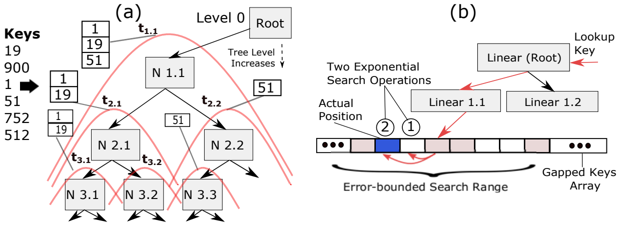


Figure 1: Examples on (a) hierarchical request buffers at some tree-based index nodes, and (b) 2-levels gapped RMI (GRMI).

Buffers Structure. In particular, the index is viewed as a tree with multiple sub-trees rooted at the child nodes, and the root of each node t is associated with a fixed-size buffer. We refer to this buffer as *request buffer*, where it temporarily stores all keys that will be traversed using the corresponding child node. Any sub-tree t can be recursively decomposed into smaller sub-trees at its own child nodes. Figure 1(a) shows an example of nodes at different levels of an index's tree, where the root of each sub-tree has a request buffer. In this example, we show the root's left sub-tree $t_{1.1}$ rooted at its child node $N1.1$, where for example, the sub-tree $t_{1.1}$ itself has two child sub-trees $t_{2.1}$ and $t_{2.2}$ (with their buffers) rooted at the $N2.1$ and $N2.2$ nodes, respectively, and so on.

Buffers In Action. The buffers for all sub-trees of the indexed relation R are created before the INLJ begins. During the INLJ operation, the query keys from the non-indexed relation S are distributed to the buffers based on the internal nodes' traversal until a buffer becomes full. When a buffer is full, we flush it by distributing the buffered query keys to its child buffers recursively (e.g., in Figure 1(a), when the buffer of sub-tree $t_{1.1}$ is filled with the query keys 1, 19, and 51, the node $N1.1$ is then used to distribute these keys to the buffers of its child sub-trees $t_{2.1}$ and $t_{2.2}$, and so on). When a query key reaches the leaf level, the final location of the key is determined. When no more query keys are issued, the whole buffers are flushed in the depth-first order similar to [17]. We also follow the guidelines in [17] to set the buffer sizes at the roots of different sub-trees. First, we set the buffer size at each leaf node as a user-defined parameter. Then, recursively, we set *the buffer size at the root of any sub-tree t as the total sizes of the associated buffers to the non-root nodes in this sub-tree*. By carefully tuning this parameter, the total size of all buffers needed for an index's tree will satisfy the analytical bound of the *cache complexity* for hierarchical indexes [17].

In our study, we investigate two state-of-the-art tree indexes: Adaptive Radix Trie (ART) [33] and the Cache-Sensitive Search (CSS) Trees [19]. We provide two INLJ variants that utilize the hierarchical buffering optimization with ART and CSS trees, referred to as ART-INLJ and CSS-INLJ, respectively.

3.2 Learned Index-based Variants

3.2.1 *INLJ with Black-Box Learned Indexes.*

One straightforward approach is to use a learned index as a drop-in replacement for traditional indexes, without buffering optimization, in INLJ. To guarantee a good accuracy for the learned index, we use the whole keys of relation R, not a sample, to build it[5]. In our study, we

[4] We selected to use AMAC because it outperforms the other prefetching-optimized inter-task parallelism techniques in hash probing as shown in [21].

[5] Using all keys to build the learned index never hurts the INLJ performance since this is done offline (INLJ assumes the index already exists before).

investigate three INLJ variants that use three different learned indexes; RMI [28, 41], RadixSpline [25], and ALEX [14], referred to as RMI-INLJ, RadixSpline-INLJ, and ALEX-INLJ, respectively.

3.2.2 INLJ with Gapped RMI. When using typical RMI [28] with challenging inputs, the lookup throughput deteriorates as the input data is difficult to learn and the accuracy of built RMI becomes poor [41]. In this case, the RMI predicts a location that is far away from the true one, and hence the overhead of searching between the error bounds becomes significant (i.e., last-mile search). This is because when an RMI is built for an input relation, it never changes the positions of the keys in this relation. A crucial insight from ALEX [14] shows that when using the *model-based* insertion to build the learned index (check Section 2), the search over the indexed keys later, using the same model used for insertion, will be improved significantly (i.e., less model misprediction errors). This inspires us to propose an INLJ variant, referred to as GRMI-INLJ, using a new read-only index, namely Gapped RMI (GRMI), that combines the benefits of both RMI and model-based insertion from ALEX. GRMI substantially improves the lookup performance over typical RMI, specially for datasets with challenging distributions.

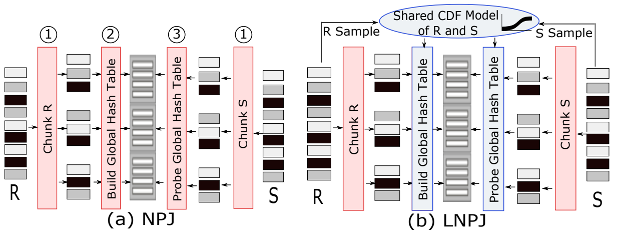
GRMI Structure and Building. GRMI consists of a *typical RMI* and a *gapped array* for keys (there is another corresponding array for payloads). We assume both keys and payloads have fixed sizes. The gapped array of keys is filled by model-based inserts of the input keys during the index building (described later). The extra space used for the gaps is distributed among the array elements to ensure that keys are located close to the predicted position when possible. The number of allocated gaps per input key is a user-defined parameter that can be tuned. For efficient lookup, the gapped array is associated with a bitmap to indicate whether each location in the array is filled or is still a gap. Building a GRMI has two straightforward steps, which happen offline. First, a typical RMI is constructed for the input keys. Then, the built RMI is used to model-based insert all keys from scratch in the gapped array. Figure 1(b) depicts an example of a 2-levels GRMI with a gapped array for keys (white cells are gaps, i.e., empty).

GRMI Lookup. Given a key, the RMI is used to predict the location of that key in the gapped array of keys. Then, if needed, an *exponential search* is applied till the actual location is found. If a key is found, we return the corresponding tuple. Otherwise, we return null. We use the exponential search to find the key in the neighborhood because, in GRMI, the key is likely to be very close to the predicted RMI location, and in this case, the exponential search can find this key with less number of comparison checks compared to both sequential and binary searches. Figure 1(b) presents a lookup example, where red arrows show the lookup flow. In this example, the RMI prediction is corrected by two exponential search steps only.

3.2.3 INLJ with Optimized RMI Indexes. Since RMI typically has 2 or 3 levels, it can be easily combined with *prefetching*-optimized inter-task parallelism techniques (e.g., AMAC [26]) to improve its probing throughput as shown in [55]. Therefore, we investigate two INLJ variants based on AMAC-optimized RMI and GRMI, referred to as PRMI-INLJ and PGRMI-INLJ, respectively.

Algorithm 2 Function PRMI-INLJ (AMACInstances s, QueryKeys *input*, QueryKeysNum N, Output *matches*)

```
1:  done ← 0 /* Flag to end INLJ computation */
2:  state ← InitializeFSMInstances (s) /* Initialize s instances of an FSM*/
3:  while done < s do
4:      k = (k == s) ? 0 : k
5:      switch state[k].stage do
6:          case P: /* Predict using the root model, and prefetch next model parameters */
7:              if i < N then
8:                  state[k].key ← LoadKey(input, i)
9:                  pred ← PredictNextLevelModelInd (state[k].key, root)
10:                 state[k].stage = J, i+ = 1
11:                 PrefetchModelParams (pred, state[k].params)
12:             elseState state[k].stage = D, ++done
13:             end if
14:         case J: /* Perform actual join check */
15:             tuple ← GetIndexedTuple (state[k].key, state[k].params)
16:             matches ← JoinCheck (state[k].payload, tuple.payload)
17:             state[k].stage = P /* Initiate prefetching for a new key in P*/
18: end while
```



(a) NPJ

(b) LNPJ

Figure 2: Non-partitioned hash join, NPJ, and its learned variant, LNPJ. Blue steps are learned.

Algorithm 2 shows the pseudo code of our proposed PRMI-INLJ with a 2-level RMI. The core idea is simple: for a key, we map the INLJ computation into a finite state machine (FSM) with two states, where the first state (Lines 6 to 13) uses the root model to predict the index of the second level model, and prefetches its parameters, and the second state (Lines 14 to 17) performs the actual join check after retrieving the indexed tuple using the prefetched model parameters. The algorithm keeps interleaving multiple running instances of the FSM till it finishes join checks with all input keys.

Similar to the optimized tree-based INLJ (Section 3.1.2), we also investigate two other INLJ variants that utilize the hierarchical buffering optimization in both RMI and GRMI, referred to as BRMI-INLJ and BGRMI-INLJ, respectively.

4 LEARNED HASH-BASED JOIN

In many studies (e.g., [57]), hash-based join (HJ) has been shown to be the most efficient type of join in many scenarios. The algorithm assumes the two input relations are not indexed. It simply builds a hash table on the join key of one relation, say R, and then uses the second relation, say S, to probe this built hash table to return the final matches.

In this section, we focus on the two main HJ categories: Non-Partitioned Hash Join (NPJ) [31, 61] and Partitioned Hash Join (PJ) [57, 61]. We start by briefly describing the multi-core variants of the representative algorithms in each category (Section 4.1). Then, we propose different HJ variants that employ CDF-based learned models and partitioning (Section 4.2).

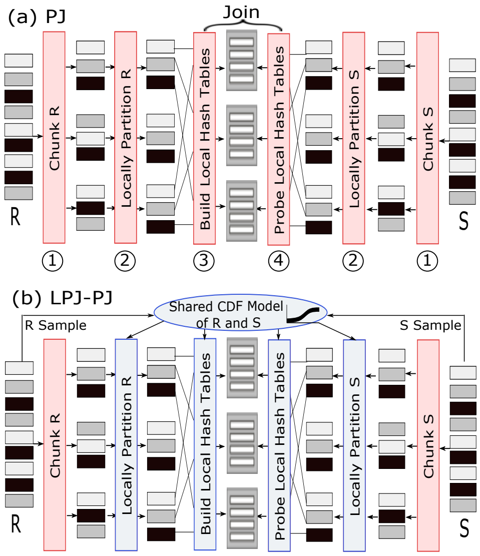


Figure 3: Partitioned hash join, PJ, and one of its learned variants, LPJ-PJ (The learned variant LPJ-P is similar to LPJ-PJ, yet exploiting the shared model for partitioning only).

4.1 Representative Algorithms

4.1.1 Non-Partitioned Hash Join. It is a direct parallel variation of the canonical hash join [31, 61], referred to as NPJ (Figure 2(a)). Basically, the algorithm chunks each input relation into equi-sized chunks and assigns them to several worker threads. Then, it runs in two phases: *build* and *probe*. In the *build* phase, all workers use the chunks of the first relation R to concurrently build a single global hash table (with a bucket chain scheme). In the *probe* phase, after all workers are done building the hash table, each worker starts probing its chunk from the second relation S against the built hash table. In our study, we investigate an NPJ variant with the following optimizations: (1) allocating the hash table among all available NUMA nodes to better utilize memory bandwidth, (2) using an efficient compare-and-swap operation (i.e., lock-free synchronization) in the build phase to avoid concurrent insertions to the same hash table bucket, and (3) employing AMAC [26] (similar to Section 3.1.1) to increase the insertion/lookup throughput in the build/probe phase.

4.1.2 Partitioned Hash Join. The key idea is to improve over NPJ by partitioning the input relations into small pairs of partitions that can fit into the cache. This will significantly reduce the number of cache misses when building and probing hash tables.

The state-of-the-art algorithm proposed in [57], referred to as PJ (Figure 3(a)), first chunks each input relation into equi-sized chunks and assigns them to several worker threads. Then, it runs two main phases: *partition* and *join*. In the *partition* phase, a multi-pass radix partitioning (usually 2 or 3 passes are enough) is applied on each chunk to locally partition it using a local histogram to avoid an excessive amount of non-local writes (i.e., writes on remote NUMA nodes). In the *join* phase, each worker independently builds a local hash table from the corresponding local partitions of R across the different workers. Then, these local hash tables are probed with the corresponding local partitions of S. In our study, we investigate a PJ variant that applies the following optimizations, as shown in [57]: (1) allocating the partitions for each worker on its local NUMA node, (2) applying software write-combine buffers (SWWC) and non-temporal streaming [61] to provide cache-efficient partitioning, (3) careful scheduling of the join tasks such that all memory controllers are utilized simultaneously, and (4) assigning multiple threads to each large partition to avoid skewness.

4.2 CDF-based Variants

4.2.1 Learned Non-Partitioned Hash Join. In our study, we investigate an NPJ variant, referred to as LNPJ (Figure 2(b)), that builds one CDF model for both input relations R and S, and then uses this model to build the global hash table and probe it (i.e., using the model to predict the hash table bucket). The main intuition is that if the input keys are generated from a certain distribution, then the CDF of this distribution maps the data uniformly in the range $[0, 1]$. In this case, the CDF will behave as an *order-preserving* hash function in a hash table [56]. To ensure that the model building overhead is insignificant, we randomly sample and use a small percentage of keys (e.g., 1%) from both relations to build the model as in Section 2.

4.2.2 Learned Partitioned Hash Join. Similar to LNPJ, we also investigate two PJ variants that exploit using a shared CDF model in the different PJ phases. The first variant, referred to as LPJ-PJ (Figure 3(b)), employs the built model in both the partitioning and joining phases. The second variant, referred to as LPJ-P, uses the built model for partitioning only. In both variants, we build the model based on a small sample from R and S as in LNPJ. After the model is built, we proceed directly with the partitioning and joining phases of PJ. Assume that $|P|$ is the total number of partitions that should be output (similar to the number of partitions obtained from histograms in PJ), and $|B_t|$ is the number of hash table buckets per worker t. In the partitioning phase, a key is partitioned by scaling its prediction from the CDF model to be between 0 and $|P|-1$. In the joining phase of LPJ-PJ, a tuple is inserted/probed in a local hash table based on another re-scaling for the obtained CDF prediction from the partitioning phase. This re-scaled prediction is between 0 and $|P| * |B_t| - 1$, and gives the corresponding bucket in the hash table to be used for insertion/probing.

5 LEARNED SORT-BASED JOIN

Assuming the two input relations are not indexed, the idea of sort-based join (SJ) is to first sort both relations on the same join key. Then, the sorted relations are joined using an efficient merge-join algorithm to find all matching tuples. SJ is an essential join operation for any database engine. Although its performance is dominated by PJ [61], it is still beneficial for many scenarios. For example, if the input relations are already sorted, SJ becomes the best join candidate as it will skip the sorting phase and directly perform a fast merge-join only.

In this section, we focus on two efficient multi-core algorithms of SJ, namely MPSM [2] and MWAY [5]. We briefly describe their details (Section 5.1). Then, we propose different variants of them that use CDF-based models and partitioning (Section 5.2).

5.1 Representative Algorithms

5.1.1 Massively Parallel Sort-Merge (MPSM). The idea of MPSM [2] (Figure 4(a)) is to generate sorted runs for the different partitions of

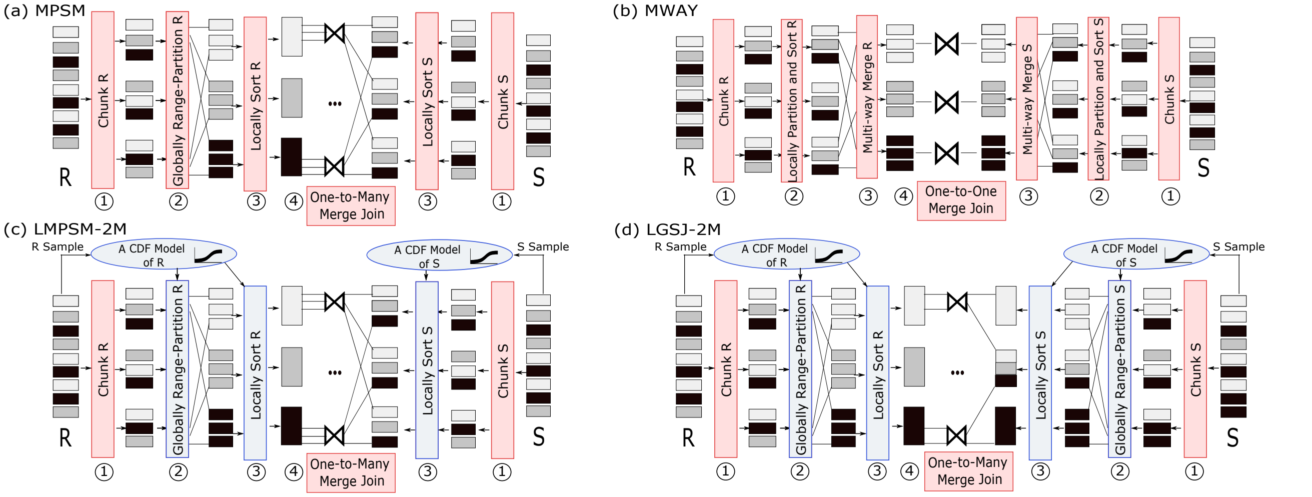


Figure 4: The upper part shows two original sort-based joins: MPSM and MWAY (The learned variants, MPSM-LS and MWAY-LS, are similar to MPSM and MWAY, respectively, yet replacing the traditional sorting step with Learned Sorting [29]). The below part shows two learned variants: LMPSM-2M and LGSJ-2M (The LMPSM-1M and LGSJ-1M variants are similar to LMPSM-2M and LGSJ-2M, respectively, yet using a shared CDF-based model for both relations instead of two separate models).

each input relation in parallel. Then, these sorted runs are joined directly without the need to merge each relation entirely first. The range-partitioned variant of MPSM is considered a state-of-the-art SJ algorithm as it is very efficient for NUMA-aware systems.

The algorithm has four steps as follows: (1) each input relation is chunked into equi-sized chunks among the workers, then (2) the smallest relation, say R, is "globally" range-partitioned, such that different ranges of R are assigned to different workers (could be located on different NUMA nodes). Meanwhile, each chunk of the largest relation S is only locally partitioned into segments using radix partitioning. After that, (3) all partitions of R and segments of S are locally sorted on their own workers (note that, after this step, R is globally sorted, but S is partially sorted as it was not range-partitioned before). Finally, (4) each partition from R is directly merge-joined with all overlapping sorted segments from S on all workers. Note that the partitioning step utilizes both SWWC buffers and non-temporal streaming to be cache-efficient. Moreover, MPSM provides a 2-step optimization that sacrifices the partitioning cost a bit to handle skewness efficiently. This optimization first estimates the global CDF of S and the global histogram of R. Then, it exploits such estimates to determine globally-balanced partitioning bounds using a complexity approximation that considers both the sort and join costs per worker.

5.1.2 *Multi-Way Sort-Merge (MWAY).*

It is another state-of-the-art SJ that has the following four steps (Figure 4(b)): (1) each input relation is chunked into equi-sized chunks among the different worker threads, where each chunk is further locally-partitioned into few segments using efficient radix partitioning (similar to the PJ's partitioning step in Section 4.1.2, yet locally and with one pass only), then, (2) in parallel, all segments of each partition from R and S are locally sorted on their own workers. After that, (3) all segments of each relation are merged such that different ranges of the merged relation are assigned to different workers. Finally, (4) each partition from the merged R is locally merge-joined with exactly one corresponding partition from the merged S that is located on the same worker. MWAY comes with three optimizations. First, both sorting and merging steps are implemented with SIMD bitonic sorting and merge networks, respectively [5]. Second, a cache-efficient "multi-way merging" technique [5] is utilized to reduce the memory bandwidth bottlenecks. Third, to handle skewness in the merging step, MWAY breaks down any enormous merge task into multiple smaller tasks and inserts them into a NUMA-local task queue shared by the workers in the same NUMA node.

5.2 CDF-based Variants

5.2.1 *MPSM and MWAY with Black-Box LearnedSort.*

These are straightforward variants from both MPSM and MWAY, where we directly use a CDF-based sorting, such as LearnedSort [29], as a black-box replacement for the sorting steps used in the original algorithms. We refer to these variants as MPSM-LS and MWAY-LS, respectively.

5.2.2 *Learned MPSM with One Model Per Relation.*

Similar to the idea of LPJ-PJ in Section 4.2.2, where we re-use the same CDF model of each relation to perform both partitioning and joining phases, we investigate a variant of MPSM, referred to as LMPSM-2M (Figure 4(c)), that builds two CDF models for both relations and re-uses them to implement the range partitioning and sorting steps of MPSM.

Similar to MPSM, the CDF-based partitioning is combined with both SWWC buffers and non-temporal streaming. Also, note that the built CDF model, for each relation, itself is cache-resident since it is shared among all workers that typically use it almost at the same time. This leads to almost no cache misses when accessing the model during partitioning and the overall process becomes highly efficient.

5.2.3 *Learned SJ with Globally-Sorting Two Relations.*

Recent studies [5, 57] have shown that locally-sorting relation chunks and then multi-way merging them across different workers, as in MWAY, is more efficient than firstly range-partitioning the chunks across different workers and then locally-sorting each partition, as in MPSM. This is because typical range-partitioning is costly. However, with having cache-efficient CDF-based range-partitioning and sorting steps (as in LMPSM-2M), it might be interesting to study a SJ variant, referred to as LGSJ-2M (Figure 4(d)), which globally range-partitions and sorts both input relations before directly joining them.

Basically, LGSJ-2M applies the range-partitioning and sorting steps of LMPSM-2M on each input relation using a separate CDF-based model built for this relation. Then, in the join phase, LGSJ-2M performs a join between each sorted partition from the smallest relation, say R, and all its *overlapping* sorted partitions from S. The join operation, which we refer to as Chunked-Join, is performed in two sub-steps: First, for any partition in R, we identify the start and the end positions of its overlapping partitions in S. To do that, we query the CDF model of S with the values of the first and the last keys of this partition from R to return the corresponding first and last overlapping partitions from S, namely f_S and l_S, respectively. Then, we merge-join any partition from R with each partition in S belongs to the corresponding overlapping range $[f_S, l_S]$ (i.e., including f_S and l_S) to find the final matching tuples.

5.2.4 LMPSM-2M and LGSJ-2M with One Shared Model for Both Relations. We push the idea of sharing work further and investigate two variants from LMPSM-2M and LGSJ-2M, referred to them as LMPSM-1M and LGSJ-1M, which build one shared CDF model for both relations, instead of two separate models, and reuse this model across the different algorithm steps of LMPSM-2M and LGSJ-2M, respectively.

6 EXPERIMENTAL EVALUATION

In this section, we experimentally study all learned and non-learned variants of INLJ, HJ, and SJ algorithms in our study (Table 1). Section 6.1 describes the experimental setup, datasets, hardware, and metrics used. Section 6.2 analyzes the performance of the three join categories under different dataset types, sizes, and skewness. Section 6.3 provides a deeper analysis for the joins under different workload characteristics, tuple sizes, duplicates, parallelism, strategies for shuffling data between NUMA nodes, and via different performance counters (e.g., cache and branch misses). Section 6.4 investigates the effect of tuning some specific parameters of the learned indexes on the performance of learned INLJ variants.

6.1 Experimental Setup

Baselines. We used all the different INLJ, HJ, and SJ algorithms discussed in previous sections in our experiments. In INLJ, we used the entire input relation to build each learned index offline as described in Section 3.2.1, and hence its accuracy becomes high. Similarly, for hash-based indexes, we build each index by inserting every key. The implementations of RMI, RadixSpline and ALEX are obtained from their open-source repositories [3, 52, 53]. The RMI hyper-parameters are tuned following the guidelines in [37]. RadixSpline is manually tuned by varying the error tolerance of the underlying models. ALEX is configured according to the guidelines in its paper [14]. For traditional indexes, the implementations of tree-structured ones are provided by their original authors, while the Cuckoo hash index is adapted from its standard implementation that is used in [59]. In SJ, we adapted a variation of MWAY, from its original open-source implementation in [44], that can work with state-of-the-art sorting algorithms (e.g., QuickSort)[6]. However, for MPSM, we provided a best-effort implementation of the original algorithm [2] on our own (unfortunately, the original code is not available to the public). In HJ, we used the reference implementations of non-partitioned and partitioned hash joins from [44]. For all HJ and SJ baselines, we followed the guidelines in a recent study [57] to optimize their performance as well as adapted them to work with variable payload sizes.

Baseline	Description
Non-learned INLJ Baselines (Section 3.1)	
CHAIN-INLJ	Using bucket chaining hash index
CUCKOO-INLJ	Using Cuckoo hash [48] index
ART-INLJ	Using Adaptive Radix Trie (ART) [33] index
CSS-INLJ	Using Cache-Sensitive Search (CSS) Trees [19] index
Learned INLJ Baselines (Section 3.2)	
RMI-INLJ	Using Recursive Model Index (RMI) [28]
RadixSpline-INLJ	Using RadixSpline [25] index
ALEX-INLJ	Using updatable learned index, ALEX [14]
GRMI-INLJ	Using the proposed gapped RMI (GRMI) index
BRMI-INLJ	Using buffer-optimized RMI index
BGRMI-INLJ	Using buffer-optimized GRMI index
PRMI-INLJ	Using AMAC-optimized RMI index
PGRMI-INLJ	Using AMAC-optimized GRMI index
Non-learned HJ Baselines (Section 4.1)	
NPJ	Non-partitioned hash join [61]
PJ	Optimized partitioned hash join [57]
Learned HJ Baselines (Section 4.2)	
LNPJ	NPJ with one joint CDF model, for both relations, in the building and probing phases
LPJ-P	PJ with one joint CDF model, for both relations, in the partitioning phase
LPJ-PJ	PJ with one joint CDF model, for both relations, in the partitioning and joining phases
Non-learned SJ Baselines (Section 5.1)	
MPSM	Massively Parallel Sort-Merge [2]
MWAY	Multi-Way Sort-Merge [5]
Learned SJ Baselines (Section 5.2)	
MPSM-LS	MPSM with LearnedSort [29] in the sorting step
MWAY-LS	MWAY with LearnedSort [29] in the sorting step
LMPSM-2M	MPSM with two CDF models, for both relations, to partition and sort them
LGSJ-2M	Using two CDF models, for both relations, to globally partition, sort, and one-to-many join them
LMPSM-1M	MPSM with one joint CDF model, for both relations, in the partitioning and sorting steps
LGSJ-1M	Using one joint CDF model, for both relations, to globally partition, sort, and one-to-many join them

Table 1: In-memory join baselines in our study.

Workloads and Datasets. Just like previous studies (e.g., [7, 8, 57]), our study only focuses on equi-join queries like "`SELECT count(*) FROM Rel1 R, Rel2 S WHERE R.k = S.k`". We also use a <key, payload> pair as a tuple. Particularly, we followed [8] in fixing the key size (always 8-byte) and varying the payload one[7]. Only in the case of INLJ, the payload is fixed to 8 bytes to be compatible with traditional indexes (e.g., ART and CSS) that store only a data pointer on the actual record. We generate the tuples in each input relation based on real and synthetic datasets. For real datasets, we use three datasets from the SOSD benchmark [41], where each one is a list of unsigned 64-bit integer keys and we generate a random payload, of a specific size, for each key. The real datasets are:

[6] Since we target (realistic) large join keys and payloads, we can not use the bitonic sorting in [44] that exploit SIMD registers as they are limited to 512-bit data types.

[7] The payload size could be larger than 8 bytes in many scenarios. For example, in row-oriented DBMSs (e.g., PostgreSQL [51], MySQL [45]), the payload stores the actual record (i.e., multiple attributes) and hence becomes very large. Even in column-oriented DBMSs (e.g., MonetDB [20], Vertica [62]), when performing multi-attribute join queries, the payload might become significantly large when stitching multiple columns in it using early materialization techniques [1] during the query processing.

- *face*: 200 million randomly sampled Facebook user IDs.
- *osm*: 800 million cell IDs from Open Street Map.
- *wiki*: 200 million timestamps of edits from Wikipedia.

For synthetic datasets, unless otherwise stated, we generate three datasets with different data distributions. Each dataset has four size variants (in number of tuples): 16, 32, 128, and 640 million tuples (each tuple has similar sizes as in real datasets). All of them are randomly shuffled. The synthetic datasets are:

- *seq_h*: sequential IDs with 10% random deletes (holes).
- *unif*: uniform distribution, with min = 0 and max = 1, multiplied by the size, and rounded to the nearest integer.
- *lognorm*: lognormal distribution with $\mu = 0$ and $\sigma = 1$ that has an extreme skew (80% of the keys are concentrated in 1% of the key range), multiplied by the dataset size, and rounded to the nearest integer.

Hardware and Implementation. Unless otherwise mentioned, we use an Arch Linux machine with 256 GB of RAM and an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with Skylake microarchitecture (SKX) and L3 cache of 55MiB. It has 2 NUMA nodes, each with 40 CPUs. Implementations are in C++ and compiled with GCC (11.1.0).

Metrics. We use the join throughput as the default metric. We define it as in [57] to be the ratio of the sum of the relation sizes (in number of tuples) and the total runtime $\frac{|R|+|S|}{|runtime|}$.

Default Settings. Unless otherwise stated, all reported numbers are produced with running 32 threads. For any INLJ variant based on GRMI, we use 4 gaps per key. To fairly compare such variants with hash-based INLJ, we build the hash indexes to match the gaps setting by (1) using a bucket size of 4 in the block chaining index (CHAIN-INLJ), and (2) creating a 2-table Cuckoo hash index (CUCKOO-INLJ), where the size of these tables are 4 times the size of the input relation (i.e., load factor 25%). For buffer-optimized tree-based INLJ, we set the buffer size at the leaf nodes to 200. For AMAC-optimized INLJ, we set the number of FSM instances (check Section 3.2.3) to be 32. For any HJ or SJ variant that uses radix partitioning, we set the number of partitions to be either 2^{12} (when using *seq_h*, *unif* and *wiki* datasets), and 2^{14} (when using *lognorm*, *face* and *osm* datasets). When building CDF models in HJ and SJ, we use only 1% random sample of the keys in the input relations. For datasets, we remove all duplicate keys in most of our experiments to make sure that some baselines (e.g., ART-INLJ) work properly (a separate experiment for the effect of duplicates exists). In addition, most of our results are reported for tuple sizes of 16 and 128 bytes (i.e., payloads of 8 and 120 bytes).

6.2 INLJ, SJ, and HJ Evaluation Results

Real and Synthetic Datasets. Figure 5 shows the join throughput of all baselines in the three join categories (each row represents a category) when the size of each tuple is 16 bytes.

For INLJ, clearly, BGRMI-INLJ and PGRMI-INLJ outperform all other INLJ competitors in all real and synthetic datasets, except *seq_h*. This is attributed to the effect of building the gapped RMI using "model-based" insertion and employing either the buffering or the AMAC optimization during key lookups. Regardless of the RMI prediction accuracy for any dataset, each key is located very close to the position where it is expected to be found, and by using the exponential search, the key is typically found after 1 or 2 search steps (using binary search in RMI-INLJ takes at least 4 search steps). We also observed that the overhead of maintaining the FSM instances needed by AMAC is a bit higher than maintaining the buffers, and hence PGRMI-INLJ is slightly worse than BGRMI-INLJ. BGRMI-INLJ is also better than GRMI-INLJ in almost all cases with a throughput improvement between 5% and 25%. RadixSpline-INLJ shows a slightly better performance than BGRMI-INLJ with the *seq_h* dataset due to the model over-fitting in the sequential cases [55]. In contrast, RadixSpline-INLJ has a very poor performance with *lognorm* and *osm* datasets. This is expected as these datasets are more skewed than others and lead to building large sparse radix tables in RadixSpline [25]. Therefore, the learned model becomes less effective in balancing keys that will be queried in the join operation. Similarly, ALEX-INLJ suffers from the extra overhead of its B-tree-like structure. This matches the results reported by our online leaderboard [60] for benchmarking learned indexes: all updatable learned indexes are significantly slower. Unlike ALEX, the proposed GRMI variants only injected ALEX's ideas of model-based insertion and exponential search into RMI (without internal nodes).

For HJ and SJ, as expected for small tuple sizes (e.g., 16 bytes), the optimized variant of PJ dominates all other baselines when using uniform and dense datasets such as *seq_h*, *unif* and *wiki*. In this case, applying one pass of logical partitioning in PJ (check Section 4.1.2), combined with SWWC buffers and non-temporal streaming, is enough to have "balanced" cache-fit partitions across different workers (i.e., no need for CDF to balance the partitions). This is highly efficient compared to (1) the excessive non-local writes on remote NUMA nodes when building the hash table in both NPJ and LNPJ, and (2) the expensive sorting step in the different SJ variants. However, such performance gain of PJ over other algorithms significantly decreases by having highly skewed datasets such as *lognorm*, *face*, and *osm*. In these datasets, most tuples fall within a specific range, but there is a small number of outlier tuples causing most of the output radix partitions in PJ to be nearly useless. On the other hand, employing the CDF-based partitioning as in LPJ-PJ and LPJ-P, even based on a small sample, will significantly balance the loads among workers and hence reduces the join bottleneck. For example, LPJ-P has at least 25% better throughput than PJ in both *osm* and *lognorm* datasets.

As shown in Figure 5, all learned SJ variants have higher throughputs compared to MWAY and MPSM due to employing the CDF-based sorting which is efficient than traditional state-of-the-art sorting algorithms when having 64-bit keys or larger [29]. However, we interestingly observe that LGSJ-2M, LGSJ-1M, LMPSM-2M and LMPSM-1M are slightly better than MPSM-LS and MWAY-LS. This is mainly because the later algorithms run the LearnedSort as a black box on different workers independently. Hence, the overhead of allocating/de-allocating the internal data structures in LearnedSort (e.g., over-allocated and spill buckets [29]) is repeated many times as well, which in turn degrades the overall join throughput. In contrast, the former algorithms build one CDF-based model for each input relation and reuse it through the partitioning, sorting, and joining steps, which saves any redundant work and increases the join throughput. Another observation is that among SJ variants that reuse models, LGSJ-2M and LGSJ-1M are better than LMPSM-2M and LMPSM-1M in all datasets because LGSJ variants perform Chunked-join, in which the partitions on each worker are joined

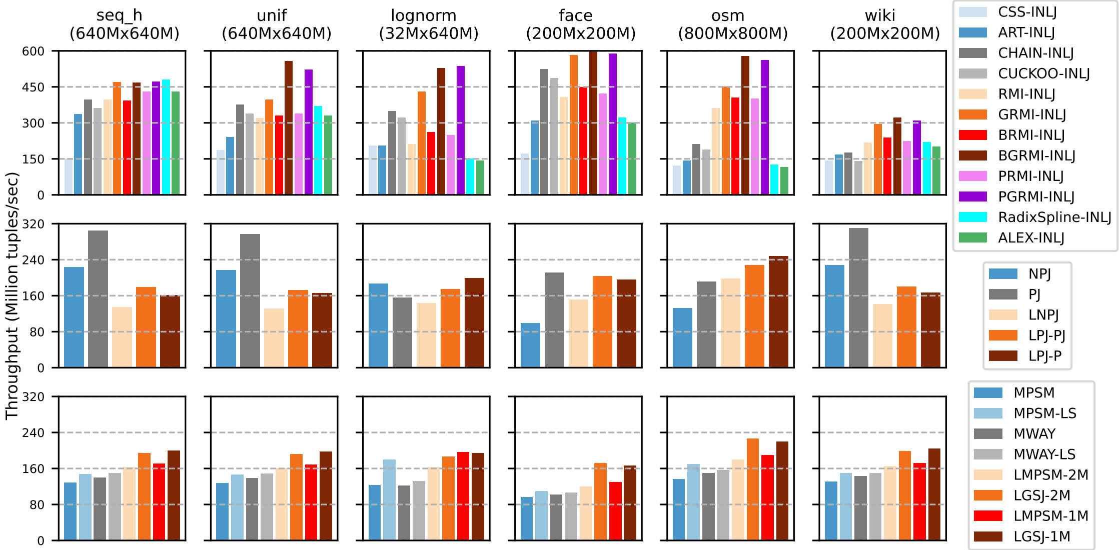


Figure 5: Performance of the three join categories for real and synthetic datasets (with 8 bytes payload).

only with the overlapping partitions on "specific" workers that are determined using the models (check Section 5.2.3). This is unlike LMPSM variants that join the partitions on each worker with the overlapping partitions from "all" other workers and hence suffer from heavy remote NUMA nodes accesses.

Figure 6 shows the join throughput of all HJ and SJ baselines when the tuple size becomes 128 bytes (larger than the 64-byte cacheline). Increasing the tuple size results in increasing the number of cache misses in the partitioning phase of any PJ variant (i.e., less number of tuples can be compacted in the cacheline of SWWC), and hence its throughput starts to decrease significantly, compared to the results in Figure 5. On the other hand, NPJ shows better throughput than PJ when increasing the tuple size, especially if the relation used to build the hash table, say R, is either not highly skewed (e.g., *unif* and *wiki*) or relatively small (e.g., in *lognorm*, $|R|$ is 32M). This is aligned with results from recent studies (e.g., [15, 36]). Interestingly, when the input relations have predictable gaps between keys such as *seq_h*, *unif*, and *wiki*, we observe that LNPJ provides better throughput than NPJ (throughput improvement ranges between 6% and 50%). As shown in [55, 56], the CDF models can over-fit to the original data distribution of these datasets and hence result in less number of collisions compared to traditional hashing, which would still have around 36.7% regardless of the data distribution (based on the birthday paradox). This is because the predictability of gaps allows for building accurate learned models, using small samples, that act as a hash function with less number of collisions compared to traditional hashing (e.g., Murmur [4]). Having fewer collisions results in fewer cache misses and hence a better join throughput. We also see the same behavior between LPJ-PJ and LPJ-P, where LPJ-PJ provides slightly better performance than LPJ-P. In LPJ-PJ, the CDF model is used in the joining phase to build and probe local hash tables, where the model over-fitting can reduce the number of collisions as well. That being said, the improvement gain of LPJ-PJ over LPJ-P is limited since the built hash table for each partition in PJ already fits in the cache, and reducing its number of collisions will not significantly reduce the cache misses.

For SJ, all baselines exhibit similar behavior in all datasets as in Figure 5, except in *lognorm*, where the learned variants of MPSM outperform others. In general, the MPSM algorithm is beneficial when the relation S is significantly larger than R, which is the case in *lognorm* ($|R|$=32M, $|S|$=640M), as it avoids the global partitioning/sorting for S. By combining MPSM with the CDF-based partitioning and sorting, the join throughput becomes even better.

6.3 Detailed Analysis and Evaluation

In this section, we show more detailed analysis and evaluation for the most efficient baselines in the three join categories, based on the results in Section 6.2, while changing different workload characteristics and environment settings. Unless otherwise stated, we show the analysis of HJ and SJ with datasets of 128-byte tuples, where the learned variants outperform non-learned ones in many cases Note that we analyze INLJ with typical 16-byte tuples.

Dataset Sizes. Figure 7(a) shows the performance of different algorithms while scaling up the number of tuples to be joined. In this figure, we report the throughput for joining four variants of the *seq_h* dataset: 32Mx32M, 16Mx16M, 128Mx128M, and 640Mx640M. For INLJ, all learned variants can smoothly scale to larger dataset sizes with small performance degradation. In case of RMI-INLJ, the "last-mile" search step will suffer from algorithmic slowdown as it applies binary search, and such slowdown becomes even less in the BGRMI-INLJ as it employs an exponential search instead. Moreover, both tree structures, ART-INLJ and CSS-INLJ, become significantly worse than BGRMI-INLJ at large datasets because traversing the tree becomes more costly compared to the exponential search. For HJ and SJ, all algorithms keep the same relative performance gains regardless of the sizes of input datasets.

Datasets Ratio. Figure 7(b) shows the performance of different algorithms while joining input datasets of different sizes (note that we focus on NPJ and MPSM variants as they are the best when

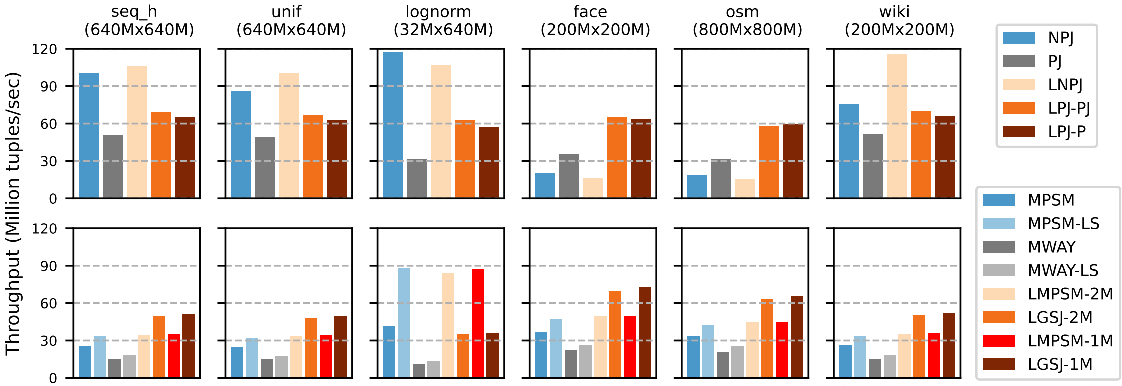


Figure 6: Performance of HJ and SJ categories for real and synthetic datasets (with 120 bytes payload).

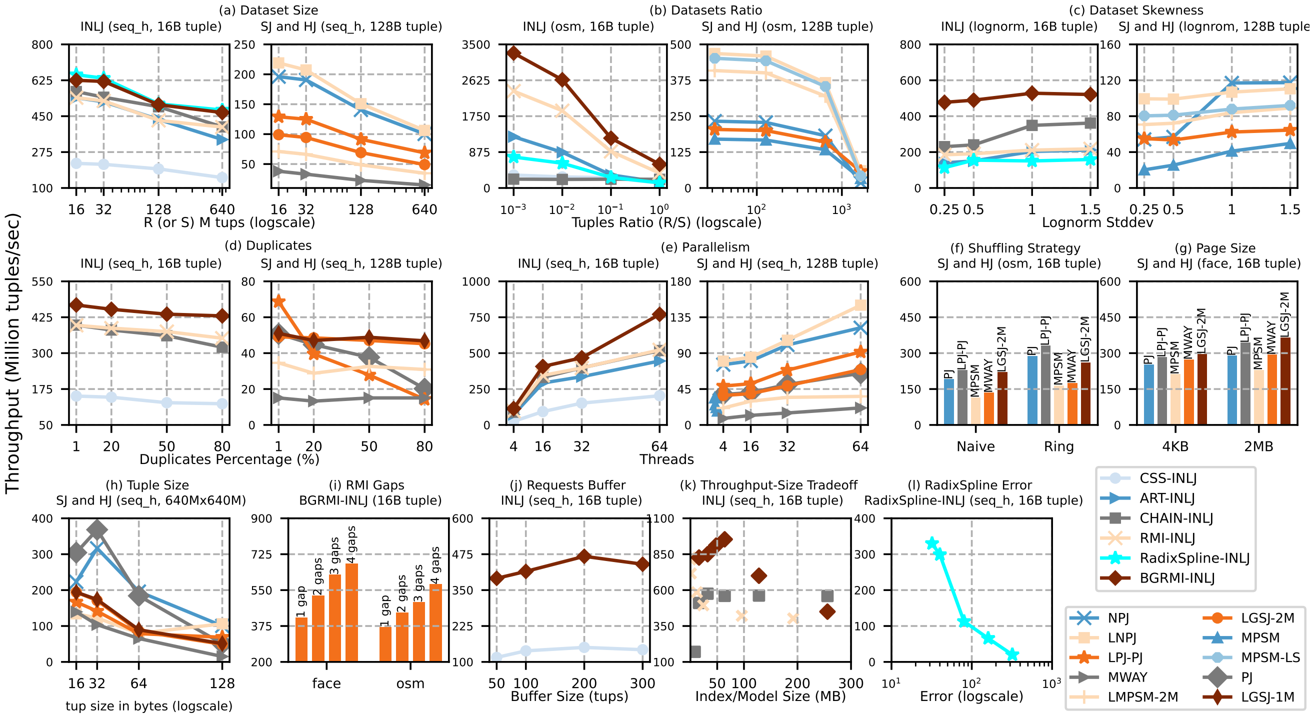


Figure 7: Ablation study for the performance of different baselines from the three join categories.

inputs significantly differ in sizes). In this experiment, we fix the size of one input relation S, which is the *osm* dataset with 800M tuples, while varying the number of tuples of the second input relation R as a ratio from S. We generate four variants of the relation R with the following number of tuples: 0.8M (0.1%), 8M (1%), 80M (10%), and 800M (100%). In case of INLJ, we index relation R to study the effect of changing the index size on the join performance.

Clearly, the throughputs of BGRMI-INLJ and RMI-INLJ at small datasets ratio, including 0.001 and 0.01, are significantly better than the throughputs of other algorithms, specially CHAIN-INLJ and CSS-INLJ (at least 2X better). The main reason for that is the extremely small size of learned indexes at these ratios (compared to the tree structures and hash tables), which can completely fit into the cache. The performance gap between all INLJ techniques significantly decreases at larger dataset ratios, including 0.1 and 1, where the rate of cache misses during index lookups becomes higher. We can also observe that both HJ and SJ baselines follow a similar performance trend as in INLJ ones, where learned NPJ and MPSM variants significantly outperform others at small dataset ratios.

Data Skewness. In this experiment, we study the robustness of different algorithms while changing the skewness degree of input datasets. Figure 7(c) shows the throughput for joining *lognorm* datasets, where one input relation is fixed and the other one is varied according to the skewness degree (i.e., σ). As a fixed relation, we use the *lognorm* dataset, with 32M tuples, that was generated using the default parameters mentioned in the experimental setup (Section 6.1). For the varied relation, we generate four new *lognorm* datasets, each containing 640M tuples, yet, with different σ values:

0.25, 0.5, 1, and 1.5. We index the varied relation when using the INLJ algorithms (note that, for HJ and SJ, we still focus on NPJ and MPSM variants as they are the best performers with the *lognorm* dataset).

As shown in the figure, all algorithms show similar performance at σ values of 1 and 1.5. However, the difference in performance becomes significant at σ values of 0.5 and 0.25 (i.e., highly skewed datasets). For INLJ, all learned variants are less susceptible to changing the skewness degree, except RadixSpline-INLJ. This is due to the ability of learned models to balance keys over the index array and avoid generating large clusters of keys. However, in RadixSpline-INLJ, increasing the skewness degree (i.e., decreasing the σ value) will increase the sparsity of the generated radix tables in RadixSpline [25], and hence leads to large clusters of keys and less join throughput. Conversely, the throughputs of CSS-INLJ and CHAIN-INLJ are significantly decreased by increasing the skewness degree due to the increased tree depth and bucket chains, respectively.

For HJ and SJ, NPJ and MPSM are the most affected algorithms by increasing the skewness. MPSM can easily result in unbalanced partitions among different worker threads, which increases the overall algorithm latency due to its range partitioning technique. Similarly, increasing the skewness will lead NPJ to have extremely unbalanced buckets. In contrast, learned MPSM and NPJ variants are almost not affected due to the balancing coming from employing the models.

Duplicates. Here, we study the performance of the algorithms while changing the ratio of duplicate keys in input datasets (i.e., covering 1-to-*m* and *n*-to-*m* joins). In this experiment, given a percentage of duplicates *x*%, we replace *x*% of the unique keys in each 640M *seq_h* dataset with repeated keys. To support joining duplicates in the RMI-based INLJ variants, we perform a range scan between the error boundaries provided by RMI around the predicted location, similar to [41]. As shown in Figure 7(d), the number of duplicates affects the join throughput of (1) INLJ variants that employ RMI and (2) HJ variants that use CDF models for hashing. This is because the learned models predict the same position/partition to the key, which increases the number of collisions. However, such collisions are less critical in learned SJ variants, such as LGSJ-2M and LGSJ-1M, because they are used for coarse-grained partitioning, not hashing. Although their performance is still degraded with increasing the number of duplicates, they remain better than non-learned algorithms.

Parallelism. Figure 7(e) shows the performance of different algorithms while scaling up the number of threads from 4 to 64. In this experiment, we report the throughput for joining the 640Mx640M variant of the *seq_h* dataset. Overall, all algorithms scale well when increasing the number of threads, although only the learned INLJ variants achieve slightly higher throughput than non-learned INLJ ones. The main explanation for this good performance is that learned INLJ variants usually incur fewer cache misses (Check the performance counters results in Figure 8) because they use learned models that have small sizes and can nicely fit in the cache. Since threads will be latency bound waiting for access to RAM, then threading in typical INLJ algorithms will be more affected by latency than learned variants, and degrades its performance. This observation is confirmed by a recent benchmarking study for learned indexes [41].

Data Shuffling. In this experiment, we study how equipping the HJ and SJ algorithms with NUMA awareness can affect the join throughput. A recent study [35] showed that the implementation details of data shuffling across different NUMA nodes can substantially impact the interconnect bandwidth and hence the overall performance. Since data shuffling (which mainly occurs in partitioning, sorting, and joining steps) is an essential operation in any join algorithm, optimizing the shuffling strategy is expected to improve the join throughput. Here, we experiment with two data shuffling strategies: (1) *Naive* shuffling (the default strategy in all algorithms), which does not consider the NUMA characteristics and lets each thread pull/write data generated by all other threads without any optimization, and (2) *Ring* shuffling, which executes the shuffling operation on multiple steps, as described in [35], to reduce the NUMA interconnect contention. Figure 7(f) shows how the join throughputs of the different algorithms change under these two shuffling strategies while using the *osm* dataset with 16-byte tuples. As expected, all algorithms benefit from the "Ring" shuffling strategy, where the improvement ratios of join throughput range between 17% and 45%.

Page Size. Figure 7(g) shows the effect of changing the page size from 4KB (the default value in Unix systems) to a larger size of 2MB (a.k.a huge page) on HJ and SJ algorithms while using the *face* dataset with 16-byte tuples. Compared to the INLJ algorithms, HJ and SJ algorithms perform a larger amount of random memory accesses (especially when having big working sets, i.e., large datasets), which can be limited by the TLB misses. We can reduce such misses by utilizing larger page sizes as suggested in [57]. As shown in the figure, all algorithms benefit from increasing the page size. We can observe that the improvement ratio in LGSJ-2M is higher than in other algorithms. This is because the overhead of the partitioning phase in LGSJ-2M is a bit higher, and hence reducing the TLB misses will have a greater impact. We skipped reporting the results using larger page sizes (e.g., 1GB) because there was no throughput improvement, as the working set already fits in the TLB when using 2MB pages.

Tuple Size. Figure 7(h) shows the performance of different HJ and SJ algorithms while changing the tuple size from 16 to 128 bytes. In this figure, we report the throughput for joining the 640Mx640M variant of the *seq_h* dataset. As shown in the figure, by increasing the payload size, the performance gap between HJ and SJ significantly decreases. In addition, LNPJ and LGSJ-1M start to show a stable performance when the tuple size exceeds the cacheline (64 bytes), and outperform all other baselines at a size of 128 (this result matches with the reported numbers in Figures 5 and 6).

Performance Breakdown. Table 2 shows the runtime breakdown (in sec) for the different algorithms when joining different datasets. For synthetic datasets (*seq_h* and *unif*), we show the results of joining the 640Mx640M variant. For the learned INLJ case, we breakdown the runtime into RMI prediction (*Pred*) and final "last-mile" search (*Srch*) steps. Since RadixSpline-INLJ does not apply a final search step (it just uses the RadixSpline prediction as a hash value), we consider its total time as prediction (i.e., "0" search step). For all datasets, except *seq_h*, the search step of BGRMI-INLJ is at least

	seq_h (16B tup)		*unif* (16B tup)		*face* (16B tup)		*osm* (16B tup)	
	Pred	Srch	Pred	Srch	Pred	Srch	Pred	Srch
RMI-INLJ	0.3	2.9	0.5	3.4	0.3	0.4	0.9	1.2
RadixSpline-INLJ	1.3	0	1.7	0	1.9	0	5.1	0
BGRMI-INLJ	0.4	2.4	0.5	0.9	0.3	0.1	0.8	0.4

	seq_h (128B tup)						*face* (128B tup)					
	Smpl	Modl	Part	Build	Sort	Join	Smpl	Modl	Part	Build	Sort	Join
NPJ	0	0	0	7.6	0	5.1	0	0	0	12.7	0	6.8
PJ	0	0	11.3	8.3	0	5.5	0	0	6.2	3.1	0	2
LNPJ	1.5	4	0	3.9	0	2.6	0.1	0.5	0	3.2	0	2.1
LPJ-PJ	1.6	4.1	6.1	6.7	0	2	0.1	0.3	1.4	2.4	0	0.3
MPSM	0	0	2.5	0	26.4	1.4	0	0	2.1	0	8.1	0.7
MWAY	0	0	5.7	0	40.5	8.7	0	0	3.6	0	13.1	1.1
LGSJ-2M	1.8	4.2	3.8	0	15.2	0.9	0.1	0.4	1.2	0	3.8	0.1

Table 2: Runtime (sec) breakdown for learned INLJ, HJ, and SJ.

3 times faster than typical RMI-INLJ, while the prediction time is almost the same. For the HJ and SJ algorithms, we breakdown the runtime into sampling (*Smpl*), CDF model building (*Modl*), partitioning (*Part*), hash table building (*Build*), sorting (*Sort*), merging (*Mrge*), and finally hash table probing in HJ or merge-joining in SJ (*Join*). Clearly, the sorting step is dominating in each SJ algorithm, and improving it will significantly boost the SJ performance. On average, LGSJ-2M has a 3X and 2X faster sorting step compared to MWAY and MPSM, respectively. Also, the LGSJ-2M joining step is at least 1.5X faster than its counterparts in MPSM and MWAY. For HJ, the improvement in the partitioning, building, and joining steps of LNPJ and LPJ-PJ ranges between 1.2X and 6X compared to their corresponding steps in NPJ and PJ. In all learned algorithms, building CDF models only represents from 6% to 32% of the total runtime.

Other Performance Counters. To better understand the behavior of the proposed learned join algorithms, we dig deep into their low-level performance counters and try to find any correlation between these counters and the join throughput. Figure 8 presents the plotting of four performance counters (in millions) for different algorithms, where each column represents a counter type. For INLJ, we can see that most of the BGRMI-INLJ cases with high join throughputs occur at low cache misses, whether L1 or LLC. This is explainable as most of such misses happen in the "last-mile" search phase (a typical 2-levels RMI will have at most 2 cache misses in its prediction phase, hopefully only 1 miss if the root sub-model is small enough), and hence improving it by reducing the number of search steps will definitely reduce the total cache misses. For other counters, it seems that there is no obvious correlation between their values and the join throughput.

In contrast, for HJ and SJ, there is a clear correlation between the counter values of LGSJ-2M and LPJ-PJ and their corresponding join throughputs. Their throughputs increase by decreasing the number of cache and branch misses. This is mainly because the CDF-based partitioning employed in them has better caching properties. In addition, the number of comparisons in these algorithms becomes lower, which in turn, reduces branch misses as well.

6.4 INLJ-Specific Evaluation Results

RMI Gaps. Figure 7(i) shows the effect of increasing the number of gaps assigned per key in BGRMI-INLJ, while using the *face* and *osm* datasets. As expected, increasing the gaps degree allows for more space around each key to handle mispredictions efficiently, which in turn reduces the final search steps and increases the throughput. However, having more gaps will come with higher space complexity as well. Thus, this parameter needs to be tuned carefully.

Model/Index Size. Figure 7(j) shows the effect of increasing the buffer size (Section 3.1.2) on the performance of two INLJ variants: BGRMI-INLJ (learned) and CSS-INLJ (non-learned). Increasing the buffer size allows more keys to be answered in batches (reducing the cache misses) and hence the join throughput becomes higher. However, after a certain threshold, the throughput degrades as it makes the index size prohibitively large, and cache misses can not be avoided (i.e., buffers can not fit in the cache anymore). Figure 7(k) shows the tradeoff between the total index size (including the buffers, if any) and the join throughput in three INLJ algorithms: BGRMI-INLJ, RMI-INLJ, and CHAIN-INLJ. Clearly, BGRMI-INLJ has the best throughput at small sizes, while CHAIN-INLJ becomes better at larger sizes. This is because having a large enough hash table significantly reduces the number of collisions (i.e., fewer cache misses) while increasing the model size in BGRMI increases the cache misses.

RadixSpline Error. Tuning the user-defined error value is an essential step for boosting the RadixSpline performance [25]. Figure 7(l) shows how the throughput of RadixSpline-INLJ is affected while changing the RadixSpline error value (Note that the figure has a logscale on its x-axis). Here, we use the *wiki* dataset. In general, increasing the error value will lead to more relaxed models with a higher misprediction rate, which in turn builds a hash table with a higher collision rate.

7 RELATED WORK

Hash-based Joins (HJ). In general, HJs are categorized into non-partitioned (e.g., [10, 31, 61]) and partitioned (e.g., [8, 57, 61]) hash joins. Radix join [6, 11, 38, 39, 61] is the most widely-used partitioned hash join due to its high throughput and cache awareness (e.g., avoiding TLB misses). [61] and [8] added software write-combine buffers (SWWC) and bloom filter optimizations, respectively, to HJ implementations. Other works optimized the performance of HJs for NUMA-aware systems (e.g., [31, 57]), hierarchical caching (e.g., [18]), and special hardware (e.g.,GPU [19], FPGA [13], NVM [58]).

Sort-based Joins (SJ). Numerous parallel algorithms have been proposed to improve the performance of sort-based joins over years [2, 5, 9, 23, 57]. [23] provided the first evaluation study for sort-based join in a multi-core setting. After that, MPSM [2] was proposed to significantly improve the performance of sort-based join on NUMA-aware systems. Multi-Way sort-merge join (referred to as MWAY) is another state-of-the-art algorithm [5] that improved over [23] by using wider SIMD instructions and multi-way merging.

Nested Loop Joins (NLJ). NLJs have been extensively studied against other join algorithms either on their own (e.g., [12]) or within the query optimization context (e.g., [32]). Block and indexed nested loop joins are the most popular variants [17]. Several works revisited their implementations to improve the performance for hierarchical caching (e.g., [17]) and modern hardware (e.g., GPU [19]).

Machine Learning for Database. The last few years witnessed a big rise in exploring machine learning for automating database operations and design decisions [27, 34]. Examples include indexing

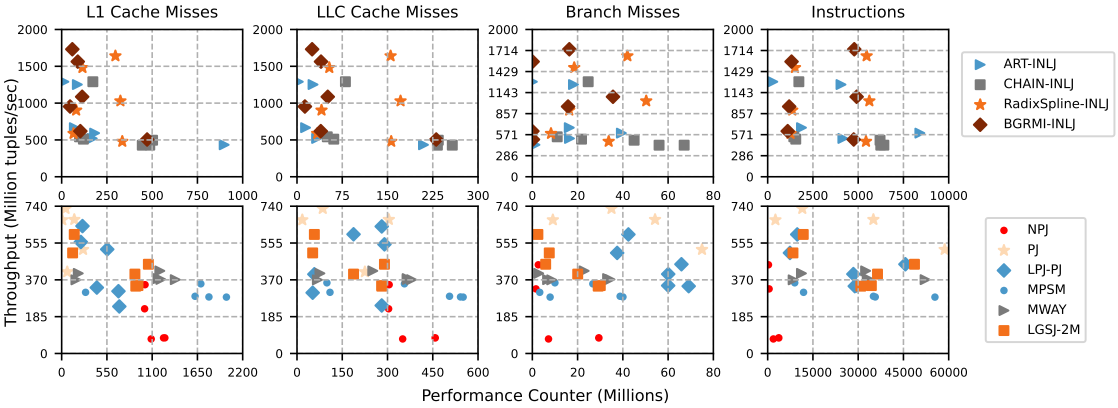


Figure 8: Low-level performance counters for exploratory analysis, where each row represents a join category.

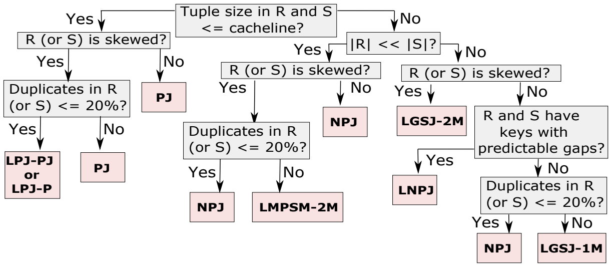


Figure 9: Summarized decision graph for HJ and SJ algorithms.

(e.g., [14, 16, 25, 28, 46]), query optimization (e.g., [42, 43]), cardinality estimation [24], data partitioning [65], sorting [29], hashing [55, 56], and scheduling [40, 54]. However, to the best of our knowledge, there is no existing work on exploring machine learning for the in-memory joins. Our proposed study in this paper fills this gap.

8 CONCLUSION AND LESSONS LEARNED

Here, we summarize the lessons learned by our extensive study. In case one of the two input relations is indexed, BGRMI-INLJ becomes the best INLJ option in almost all scenarios, as long as its gapped arrays and buffer parameters are carefully tuned. For HJ and SJ, selecting the best alternative depends on many input data characteristics including tuple size, data distribution (keys' skewness and distribution of gaps), relations' sizes, and duplicates percentage. Due to the lack of space, we summarized our findings in a decision tree shown in Figure 9. We can see that learned join variants are mostly effective when tuple sizes are beyond the cacheline, specially when datasets are skewed and have a small percentage of duplicates. In these cases, CDF models can capture data distributions accurately and provide better performance in partitioning, sorting, and building/probing hash tables compared to state-of-the-art algorithms. We believe that our findings can help practitioners to decide more easily when learned join algorithms become efficient, and which algorithm to use under different circumstances.

REFERENCES

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2012.

[3] ALEX: An Updatable Adaptive Learned Index. https://github.com/microsoft/ALEX.

[4] Austin Appleby. MurmurHash. https://sites.google.com/site/murmurhash/, 2011.

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2013.

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 27(7):1754–1766, 2015.

[7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 362–373, 2013.

[8] Maximilian Bandle, Jana Giceva, and Thomas Neumann. To Partition, or not to Partition, That is the Join Question in a Real System. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2021.

[9] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed Join Algorithms on Thousands of Cores. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2017.

[10] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2011.

[11] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1999.

[12] Mingxian Chen and Zhi Zhong. Block Nested Join and Sort Merge Join Algorithms: An Empirical Evaluation. In *Advanced Data Mining and Applications*, 2014.

[13] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. Is FPGA Useful for Hash Joins? Exploring Hash Joins on Coupled CPU-FPGA Architecture. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2020.

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

[15] Jian Fang, Jinho Lee, H. Peter Hofstee, and Jan Hidders. Analyzing In-Memory Hash Join: Granularity Matters. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB*, 2017.

[16] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2020.

[17] Bingsheng He and Qiong Luo. Cache-Oblivious Nested-Loop Joins. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*,

2006.

[18] Bingsheng He and Qiong Luo. Cache-Oblivious Query Processing. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2007.

[19] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2008.

[20] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[21] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2018.

[22] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP and OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.

[23] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2009.

[24] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2019.

[25] 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 aiDM@SIGMOD*, 2020.

[26] Onur Kocberber et al. Asynchronous Memory Access Chaining. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2015.

[27] Tim Kraska. Towards instance-optimized data systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2021.

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

[29] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[30] Ani Kristo, Kapil Vaidya, and Tim Kraska. Defeating Duplicates: A Re-design of the LearnedSort Algorithm. In *Proceedings of the AIDB Workshop @VLDB*, 2021.

[31] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-Aware Hash Joins. In *In-Memory Data Management and Analysis, IMDM*, 2015.

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2015.

[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2013.

[34] Guoliang Li, Xuanhe Zhou, and Lei Cao. Machine Learning for Databases (Tutorial). In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2021.

[35] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. NUMA-aware Algorithms: The Case of Data Shuffling. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2013.

[36] Feilong Liu and Spyros Blanas. Forecasting the Cost of Processing Multi-Join Queries via Hashing for Main-Memory Databases. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC*, 2015.

[37] Marcel Maltry and Jens Dittrich. A Critical Analysis of Recursive Model Indexes. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2022.

[38] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2000.

[39] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 14(4):709–730, 2002.

[40] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM*, pages 270 – 288, 2019.

[41] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2020.

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2021.

[43] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2019.

[44] Multi-Way Sort Merge Join. https://systems.ethz.ch/research/data-processing-on-modern-hardware/projects/parallel-and-distributed-joins.html.

[45] MySQL. https://www.mysql.com/.

[46] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-Dimensional Indexes. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[47] Thomas Neumann and Michael Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2020.

[48] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[49] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. The Case for Learned Spatial Indexes. In *Proceedings of the AIDB Workshop @VLDB*, 2020.

[50] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A Data Platform Based on the Scaling-up Approach. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2018.

[51] PostgreSQL. https://www.postgresql.org/, 2019.

[52] RadixSpline. https://github.com/learnedsystems/RadixSpline.

[53] Recursive Model Indexes. https://github.com/learnedsystems/RMI.

[54] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2022.

[55] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When Are Learned Models Better Than Hash Functions? In *Proceedings of the AIDB Workshop @VLDB*, 2021.

[56] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. Can Learned Models Replace Hash Functions? In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2023.

[57] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2016.

[58] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-Scale in-Memory Analytics on Intel Optane DC Persistent Memory. In *Proceedings of the International Workshop on Data Management on New Hardware, DaMoN*, 2020.

[59] SOSD. https://github.com/learnedsystems/SOSD.

[60] SOSD Leaderboard. https://learnedsystems.github.io/SOSDLeaderboard/leaderboard/.

[61] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2013.

[62] Vertica. https://www.vertica.com/, 2023.

[63] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are Updatable Learned Indexes Ready?, 2022.

[64] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable Learned Index with Precise Positions. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2021.

[65] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[66] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 27(7):1920–1948, 2015.

[67] Jingren Zhou and Kenneth A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.