G-Learned Index: Enabling Efficient Learned Index on GPU

Jiesong Liu, Feng Zhang, *Member, IEEE*, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li, Huanchen Zhang, Jidong Zhai, *Senior Member, IEEE*, Hechen Zhang, Yuxing Chen, Anqun Pan, and Xiaoyong Du, *Member, IEEE*

***Abstract*—AI and GPU technologies have been widely applied to solve Big Data problems. The total data volume worldwide reaches 200 zettabytes in 2022. How to efficiently index the required content among massive data becomes serious. Recently, a promising learned index has been proposed to address this challenge: It has extremely high efficiency while retaining marginal space overhead. However, we notice that previous learned indexes have mainly focused on CPU architecture, while ignoring the advantages of GPU. Because traditional indexes like B-Tree, LSM, and bitmap have greatly benefited from GPU acceleration, a combination of a learned index and GPU has great potentials to reach tremendous speedups. In this paper, we propose a GPU-based learned index, called G-Learned Index, to significantly improve the performance of learned index structures. The primary challenges in developing G-Learned Index lie in the use of thousands of GPU cores including minimization of synchronization and branch divergence, data structure design for parallel operations, and usage of memory bandwidth including limited memory transactions and multi-memory hierarchy. To overcome these challenges, a series of novel technologies are developed, including efficient thread organization, succinct data structures, and heterogeneous memory hierarchy utilization. Compared to the state-of-the-art learned index, the proposed G-Learned Index achieves an average of 174× speedup (and 107× of its parallel version). Meanwhile, we attain 2× less query time over the state-of-the-art GPU B-Tree. Our further exploration of range queries shows that G-Learned Index is 17× faster than CPU multi-dimensional learned index.**

***Index Terms*—Learned index, parallel, GPU, acceleration.**

I. Introduction

THE currently increasing amounts of data pose significant hurdles in terms of efficient data accesses. For example, the total data volume stored worldwide reaches 200 zettabytes in 2022 [1], and 11.45 zettabytes of data are generated everyday [2]. Index structures such as B-Tree [3], hash tables [4], and Bloom filters [5] are required for efficient data access. In recent years, Kraska et al. [6] proposed a promising transformative understanding of the index problem – Indexes are models that can be trained to map keys to their locations in a sorted order. More specifically, the position of a key can be estimated in a sorted array if the cumulative data distribution function f of the given dataset can be learned, and accordingly, traditional index structures can be replaced by f. This "learned" approach achieves significant time and space benefits compared to traditional indexes. Further, GPUs are effective and popular accelerators for index structures [7], [8], [9], [10], [11], [12] and machine learning [13], [14], [15]. In particular, the contexts of batch operations in dynamic scenario are perfectly suited for GPU parallelism. In fact, many index structures, such as B-Tree [7], have already been proven to be successfully accelerated by executing batch operations on GPUs, bringing even more than 60× performance improvement [7]. Therefore, we consider it essential to enable the learned index on GPUs to further improve its performance [16], [17], [18].

Enabling efficient learned indexes on GPUs has three major advantages. First, thousands of GPU cores offer huge potentials for dramatically increasing the performance of learned indexes. For example, the NVIDIA GeForce RTX 2080 Ti GPU [19] features 4352 cores, making it highly efficient for executing parallel jobs in specifically designed workloads [20]. Second, comparable workloads, such as B-Tree, matrix computation, and machine learning, have already benefited significantly from GPU acceleration. For example, research [18] shows that GPU LSM can obtain a 13.5× performance boost from GPU. Higher gains in learned index performance can be expected because they advance intelligent structure design in terms of time efficiency and more crucially, space savings. Third, GPUs have been widely used in database operations, and the GPU-based learned

Manuscript received 11 April 2023; revised 9 January 2024; accepted 15 March 2024. Date of publication 2 April 2024; date of current version 12 April 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022ZD0115304, in part by the National Natural Science Foundation of China under Grant 62072458, Grant 62172419, Grant 62225206, and Grant U20A20226, and in part by Beijing Nova Program under Grant 20220484137 and Grant 20230484397. Recommended for acceptance by D. Li. *(Corresponding author: Feng Zhang.)*

Jiesong Liu, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China (e-mail: liujiesong@ruc.edu.cn; fengzhang@ruc.edu.cn; lvlu@ruc.edu.cn; 2018202121@ruc.edu.cn; xiaoguangguo@ruc.edu.cn; duyong@ruc.edu.cn).

Dong Deng is with the Computer Science Department, Rutgers University, New Brunswick, NJ 08854 USA (e-mail: dong.deng@rutgers.edu).

Guoliang Li and Jidong Zhai are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: liguoliang@tsinghua.edu.cn; zhaijidong@tsinghua.edu.cn).

Huanchen Zhang is with the IIIS, Tsinghua University, Beijing 100084, China (e-mail: huanchen@tsinghua.edu.cn).

Hechen Zhang is with the High School Affiliated, Renmin University of China, Beijing 100080, China (e-mail: zhanghechen2@gmail.com).

Yuxing Chen and Anqun Pan are with the Database R&D Department, Tencent, Shenzhen 518000, China (e-mail: axingguchen@tencent.com; aaronpan@tencent.com).

We have made G-Learned Index available at https://github.com/Fred1031/G-Learned-Index.

Digital Object Identifier 10.1109/TPDS.2024.3381214

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

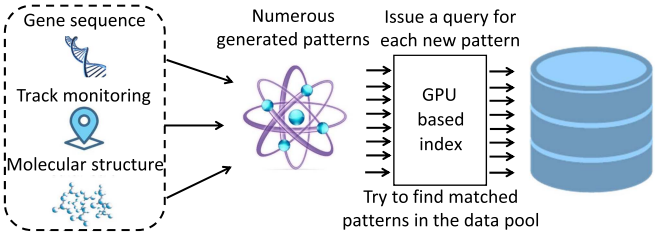

Fig. 1. Use case of GPU-based index in gene retrieval.

index can operate as a sub-component that expands the possibilities for GPU query engines, e.g., MapD [21]. Eventually, we can use an epoch based strategy to integrate our index into a full-fledged database system [22], [23], [24], an approach that has been widely used in database systems, e.g., BatchDB [25]. The basic idea is to split the queries in epochs and concurrently process the queries in each epoch.

We show a use case of applying GPU-based learned index in gene retrieval [26] in Fig. 1. In gene retrieval scenario, a large amount of newly-generated patterns, e.g., gene segments [26] and molecular structures [27], are required for matching in a given pool. For example, the synthesized patterns that need to be matched for a standard laboratory can reach two kilobase pairs each time [27], requiring high throughput of data retrieval. Because learned index, e.g., RMI and PGM-index, and GPU can provide high throughput, similar tasks can benefit greatly from GPU-based learned index. Other application scenarios, such as parallel nested loop join [28], can also benefit. More details are elaborated in Section III-B.

Although enabling learned index on GPUs is both advantageous and necessary, developing an efficient GPU-based learned index faces three major challenges.

First, the performance of GPU-based learned index is highly dependent on the GPU's thousands of cores being fully utilized. This poses substantial difficulty on GPU-based learned indexes because learned indexes entail fundamental dependency between operation phases that is not ideal for parallelism. Namely, for a query, operations on the next level in the recursive structures are dependent on the output of the model in the upper level of learned index [29], which is extremely unfriendly to the SIMT GPU working pattern [30].

Second, the multiple levels of learned index need to adapt to the complicated GPU memory hierarchy. The size of learned indexes varies depending on the dataset and the precision loss ϵ [29]. For example, a huge dataset with a large ϵ can also result in an index structure that is compact enough to fit into the shared memory on GPU. In contrast, if a small ϵ is applied, the produced structure of even a relatively small dataset has to be located on global memory. Therefore, the configurations of learned indexes need to be adjusted.

Third, the complicated index structures and operation algorithms pose difficulties to space occupation and time complexity. In terms of space concerns, a complete index structure can be too large to put onto a GPU device due to space constraints. For example, GPU cannot hold index structures with more than millions of elements. As for execution time, query operations rely on recursive searches for learned indexes, which is time-consuming.

There is a large literature of learned indexes [31], but none can address the above challenges in applying learned index on GPUs. In detail, there are currently two mainstream learned indexes that successfully achieve significant time and space savings: the Recursive Model Index (RMI) [6], an innovative hierarchical structure integrated with machine learning models, and the Piecewise Geometric Model index (PGM-index) [29], which uses algorithmically generated linear regression models to approximate key positions. More recent studies have stated that models on the bottom level of the learned index can be replaced by B-Trees to maintain a balanced partition and improve precision [16], [32], [33]. However, these studies of learned index are mostly concerned with the algorithm on CPU architectures.

In this paper, we propose a **GPU-based learned index**, called G-Learned Index, which enables efficient learned indexes on GPUs. G-Learned Index incorporates three innovative features and can address the three challenges listed above. First, we create a fine-grained synchronization-free approach for processing batch queries with low branch divergence. Second, we develop a novel heterogeneous model to maintain a succinct structure on GPUs and a lock-step working pattern for various query operations. Third, to achieve maximum bandwidth efficiency, we adaptively separate the structures of G-Learned Index and place them into distinct hierarchical memories on the GPU.

We evaluate G-Learned Index on four real-world datasets and four synthetic datasets with different data types and key sizes. Experiments show that G-Learned Index achieves an average of 3050× speedup in throughput over the state-of-the-art learned index. Compared to the state-of-the-art B-Tree on GPU, G-Learned Index achieves 34× performance speedup on average, and saves orders of magnitudes of space on various workloads. Furthermore, G-Learned Index is applicable and indexable in multidimensional space, where it achieves an average of 152× reduction in query time compared to R-Tree and a 97% decrease in space. We summarize our contributions as follows.

- We develop a dynamic learned index on GPUs with an intelligent design that is optimized for queries and updates.
- We design an efficient indexing strategy that achieves coalesced memory access and reduced branch divergence, both for point queries and multi-dimensional queries.
- We extensively assess G-Learned Index on eight datasets and demonstrate its significant advantages over the state-of-the-art methods.

II. Background

A. Learned Index Structures

Learned indexes combine the index structures with the idea of machine learning, and they regard indexes as trainable models. More precisely, assume that we query the position of the key k in the sorted array A; indexes are viewed as models with k as the input and predict the position of k in A. To accomplish this, the Cumulative Distribution Function (CDF) f of A must be learned, and the position of k can be obtained by multiplying N by $f(k)$, where N denotes the size of A. To assist the

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

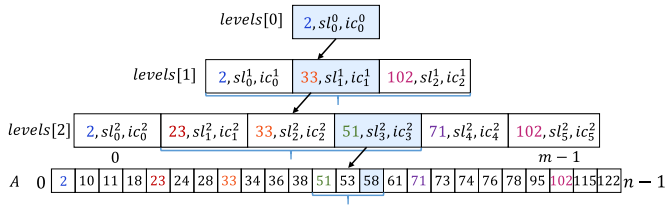

Fig. 2. Recursive search process of $\boldsymbol{x} = 58$ in a three-level PGM-index with $\boldsymbol{\epsilon = 1}$.

learning process of the function, models ranging from the linear functions to more sophisticated machine learning strategies have been explored [6], [29], [34]. Currently, the best-known learned index models are the Recursive Model Index (RMI) [6] and the Piecewise Geometric Model (PGM-index) [29].

Recursive Model Index (RMI): RMI is a multi-level recursive regression model that is organized as a hierarchical Directed Acyclic Graph (DAG). Each node in the graph represents a model that behaves like a CDF. When a key k is queried, we obtain the model denoted by the node in the first level. Note that there is only one node in the first level. We take k as input and keep the model's output, which is used to select the model for the next level. By repeating these operations, we can obtain the output of the entire RMI at the last model in the last level. The output can be interpreted as the estimated position of k in the array A. Finally, we perform the "last mile" search to determine the real location of k.

Piecewise Geometric Model index (PGM-index): The PGM-index is another learned index structure that occupies a prominent place with its succinct structure for constructing algorithmic models to learn data distributions [29]. By solving the simplified Piecewise Linear Approximation (PLA) problem, the PGM-index develops a Fitting-Tree [35], which provides a more specific and interpretable method to learn the data distribution. In other words, for a given deviation bound $\epsilon \geq 1$, the Fitting-Tree generates a combination of linear models [34]. Specifically, the optimal PLA finds a minimum number of linear model groups for a given array $A[0, b]$, such that each linear model is responsible for an indexing subset $A[a_i, b_i]$. Each linear model is referred to as a *segment*. Note that a *segment* is a tuple of three elements, $(key, slope, intercept)$. For an input $A[x], x \in [a_i, b_i]$, the corresponding linear model returns its estimated position y, which guarantees $|y - x| \leq \epsilon$. Satisfying the optimization principle and Markov property, an $O(n^3)$ Dynamic Programming (DP) algorithm is generated to solve this PLA problem, whereas it admits a streaming algorithm in linear optimal space and time after extensive study [36].

For a query specified by x, the PGM-index retrieves the visited segment in the current level and uses it to estimate the position of x of the next segments. A binary search inside the 2ϵ range centered on the estimated position is then used to determine the accurate position.

Example: An example of PGM-index query $x = 58$ with $\epsilon = 1$ is shown in Fig. 2. At the highest level, we have $\lfloor x \cdot sl_0^0 + ic_0^0 \rfloor = 1$. Then, in $levels[1][1 - \epsilon, 1 + \epsilon]$, we look for x inside the key range $[2, 33, 102]$. Following a binary search, the segment $levels[1][1]$ (encircled by the blue bracket) is returned because x falls within 33 and 102. We conduct a similar process by first calculating $\lfloor x \cdot sl_1^1 + ic_1^1 \rfloor = 2$ and finding segment $levels[2][3]$ in $levels[2][2 - \epsilon, 2 + \epsilon]$ among keys [23, 33, 51]. Up to now, we take four procedures to reach the segment at the lowest bottom level. In the lowest level, we have $\lfloor x \cdot sl_3^2 + ic_3^2 \rfloor = 12$. A search process is then performed in $A[12 - \epsilon, 12 + \epsilon]$ among the keys [51, 53, 58] before the accurate position of 13 is identified.

B. General-Purpose Computing on GPU

GPUs are widely recognized as high-performance computing (HPC) accelerators and have garnered significant attention from the research community. Numerous applications [7], [18], [37], [38], [39], [40] involving a large number of independent parallel tasks have demonstrated substantial performance improvements. A GPU contains several Streaming Multiprocessors (SMs), and each SM contains a large number of light-weight cores. Threads are grouped into blocks, which are then assigned to SMs. During execution, the threads in each SM are executed in *warp* granularity, which means that threads within the same warp must be executed in a single-instruction-multiple-threads mechanism (SIMT). Furthermore, GPU features a special memory called shared memory, referring to a small, fast memory space that is physically located on each SM. It is designed to be accessible by all threads within a CUDA thread block, enabling efficient communication and data sharing among GPU threads during parallel processing kernels. There are also more peripheral storage areas, including the L1 cache located independently on each SM, the L2 cache shared across SMs, and a global memory that is accessible to all threads but has a slower transfer speed.

GPU applications: GPUs have been successfully used in a variety of applications, including data science and machine learning. For example, GPUs have been used in data management systems [37]. Besides, traditional index data structures, such as B-tree [7], B+-tree [8], and LSM tree [18], all can be accelerated by GPUs. Machine learning workloads such as linear regression models [41] and neural network models, including fully connected networks [42], recurrent neural networks [39], and convolutional neural networks [40], can all benefit from GPU acceleration. Hence, we believe that learned indexes also have significant acceleration potentials, although challenges must be overcome.

Directly applying the previous work to GPU is challenging. First, learned indexes involve a fundamental dependency between operation phases, making it difficult to fully leverage the thousands of cores on a GPU. This is because a single thread is not optimal for handling complex instructions. Second, GPU has its own memory hierarchy, such as shared memory that is accessible to the threads within a CUDA block. It requires special designs to effectively utilize the various types of memory. Third, the limited memory space on a GPU presents a hurdle for previous learned index implementations, as the complete index structure may be too large to fit within the device's space constraints.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

III. Motivation

A. Developing Learned Index on GPUs

Guidelines: First, granularity should be an important factor. Second, diving into operator-level and calculation-level design strengthens the efficacy of the GPU-based models. Third, the feasibility and flexibility of thread assignment working on the structure should be considered.

Idea: As discussed in Section I, it is necessary to develop learned index on GPUs. To fully leverage GPU resources, our basic idea is to process queries in parallel. Building machine learning models is well suited for GPUs because matrix multiplications in both training and inference phases can significantly benefit from GPU's high parallelism. Accordingly, our first attempt is to launch multiple GPU threads working cooperatively for a single index in both RMI and PGM-index. However, coordinating different threads under a single query is much harder than a one-thread-one-query working pattern. Furthermore, the overhead of transferring data between CPU and GPU could not be neglected.

The second approach is to assign each query to a single thread and indexing the queries concurrently. In other words, we process indexes in queries at batch granularity. Due to massive numbers of GPU threads, applying GPU has great potential to achieve higher query throughput than CPU-based learned indexes. Accordingly, we need to construct a structure that matches the indexing strategy and supports efficient data transmission between CPUs and GPUs, because data transfer is an important factor affecting performance.

Comparison of RMI and PGM-index: We next analyze the index options. Specifically, the intent is to execute the same operations among different GPU threads according to the SIMD working pattern. In the RMI [6] inference phase, different queries follow varying searching patterns, arriving at different nodes with diverse machine learning models. Operations conducted on respective threads differ from each other, which causes branch divergence, making it hard to fully leverage the GPU resource. Further, the large number of fine-tuned parameters in various RMI machine learning models leads to more uncertain models. In contrast, the PGM-index [29] has only three parameters for a linear model, and maintains a succinct index structure. These attributes make it more suitable for GPUs.

Identifying PGM-index as our focus: In conclusion, we develop a GPU-accelerated learned index based on the PGM-index mainly for two reasons. First, the PGM-index structure is relatively simple and enables efficient data movement between CPUs and GPUs. The cost of time and space of the index structure is marginal. For example, the index constructed on 1e7 numbers with $\epsilon = 32$ only takes the space occupancy of 43 KB, which takes 0.01 ms of data transfer. In addition, the PGM-index has a low search cost that scales logarithmically with ϵ. Second, the PGM-index lends itself to parallelism. In particular, each step involves the same operations between different threads. This is an ideal characteristic for GPU parallel computing to bring out the optimal of GPU efficacy. Moreover, the PGM-index can guarantee a precision loss of ϵ.

B. Case Study

GPU-based learned index encapsulates the field of learned index technologies, which can be used to accelerate many applications. We use parallel nested loop join as a case study to show the motivation for in-memory relational database systems.

Task formalization: Assume we conduct join queries on two relations R and S. R and S are binary tables [43], each of which consists of two integer attributes, $index$ and $value$. The SQL query is "SELECT $R.index = S.index$ FROM R, S WHERE $(R.index = S.index)$ AND $((R.value < S.value)$ OR $(R.value{=}S.value)$ OR $(R.value > S.value))$". R is the outer table and S is the inner table. We can build a GPU-based learned index on the inner relation S, and issue a read-only query for each record in R in parallel. In this parallel indexed nested loop join, we can obtain significant performance speedups by processing concurrent read-only queries with the help of GPU-based learned index.

Parameters: Various parameters that can influence the performance of parallel indexed nested loop joins need to be considered. For example, we denote the percentage of outer records whose corresponding matches can be found in the inner table as $match\,rate$. The GPU-based learned index can work well for situations where a record in an external table does not match an internal record. Besides, we consider $index\,rep$, the average repetitions of matches for each outer record. For example, an $index\,rep$ of 10 indicates that there is an average of 10 matched records in the inner table for each index value.

Integration to databases: GPU-based learned index can be integrated into end-to-end databases such as GPU-accelerated VoltDB [28]. All that needs to do is to replace the index in VoltDB with the GPU-based learned index and process queries in epochs. For table join setting such as the SQL query mentioned in the task formalization, indexing occupies the majority of processing time, and the GPU-based learned index can greatly improve the system performance. More detailed experiments are elaborated in Section VI.

Applying GPU-based learned index to other scenarios: Many applications can benefit from GPU-based learned index with batch indexing. First, *pattern matching in information pools* [44], such as gene retrieval as elaborated in Section I, can be greatly accelerated. Second, for *high concurrency workloads* [45], e.g., Black Friday [46], GPU-based learned index can be used to handle such batching queries. Third, for *machine learning feature stores* [47], analysts need to retrieve training and testing records from diverse datasets among different tasks [48]. We can use GPU-based learned index to select training and testing records concurrently.

Potential analysis: We analyze the proportion of indexing in three applications of nested loop join, pattern matching, and feature store to explore the potential performance benefits of GPU-based learned index, as shown in Fig. 3. For nested loop join, we use two synthetic datasets. For the pattern matching task, we use a real-world dataset. For feature map task, we use the

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

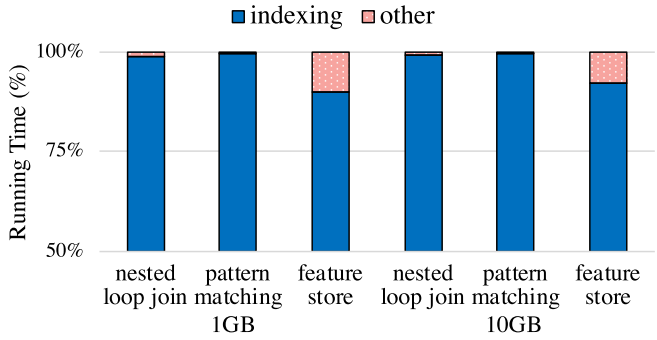

Fig. 3. Running time breakdown of different workloads.

taxi rides made in New York City [49]. The details are covered in Section VI-B. On average, indexing accounts for more than 80% of the time, which means that indexing has become the bottleneck of these applications and the GPU-based learned index can potentially bring significant performance boost.

IV. Overview of G-Learned Index

We propose a GPU-based learned index, called G-Learned Index, which can enable efficient learned index on GPUs. Fig. 4 depicts an overview of G-Learned Index, which consists of four major components: 1) architecture construction, 2) hierarchical memory arrangement, 3) query execution, and 4) multidimensional index. These components combine to generate G-Learned Index.

The first component is **architecture construction**. The architecture comprises two phases. The first phase is *heterogeneous structure preparation*. Specifically, we place all the segments on the GPU and array A containing the original data on the CPU. The motivation for this placement is that A can be too large to fit into the GPU memory. For the smaller size of A, we do the same placement for optimal performance, which is explored in Section V-A. The second phase is *parametric auto-fitting*, and it is an adaptive tune-up method for refining the model parameters, namely the precision loss ϵ.

The second component is **hierarchical memory arrangement**, which fits our heterogeneous model to the memory hierarchy of the GPU. In detail, we separate the storage into a hierarchy based on response time. Storing segments on GPU's shared memory results in faster data transfer because of higher bandwidth. If the size of the segments is too large, those segments exceeding the shared memory limits should be separately stored in GPU's global memory.

The third component of G-Learned Index is concurrent batch **query execution**. This is the component responsible for indexing our G-Learned Index. The query execution for concurrent batch processing should have features of advantageous mutability, accessibility, and flexibility. Specifically, we employ an additional *lookahead strategy selection* to determine the indexing strategy. We discuss our observations and insights in Section V-B.

The fourth component is the **multidimensional index**. We expand our G-Learned Index application to spatial circumstances in order to index two-dimensional objects and perform range queries. Following a *dimension reduction*, each two-dimensional object is assigned to a one-dimensional address. This enables numerical comparison of two objects, thus making data sorting, and ultimately, data distribution learning possible. After that, a *border query* and a *region check* are performed in turn to obtain the final indexing results.

Novelties: There are three major novelties in G-Learned Index. First, it parallelizes the updatable PGM learned index using thousands of GPU cores, considerably improving the speed of learned indexes to an unprecedentedly high throughput for batch queries. Second, our carefully constructed heterogeneous structure caters to varied features of CPUs and GPUs, taking into account both learned index structure characteristics and device architecture properties. Third, we develop a multidimensional learned index on GPUs that can tackle spatial queries in parallel.

We next show the detailed design of G-Learned Index.

V. Detailed Design

In this section, we first describe how the proposed heterogeneous architecture is built and indexed in Sections V-A and V-B. Then, we specify the data structures we employ in Section V-C, accompanied by the discussion of insertions and deletions in Section V-D and multidimensional index in Section V-E.

A. Architecture Construction

Heterogeneous structure preparation: In this module, we split the index into two parts and distribute them to CPU and GPU, respectively. We start by conducting the part-division job. The first part is levels of segments generated by the PLA-model [29], [34], and the second part is a dynamically allocated container array A. This split design takes advantage of the following insight. Although a PGM-index is able to index enormous datasets, G-Learned Index is constrained by the limited capacity available on GPUs. Fortunately, we observe that even when a large amount of space is required for array A, the last level of segments never takes up much space since the number of segments in that level is empirically small (a theoretical bound is $n/(2\epsilon)$). Thus the GPU memory is sufficient for storing those segments. Based on the above analysis, we divide G-Learned Index into two parts — the array A storing keys in the original dataset, and segments produced by the PGM index — and place A on CPU and segments on GPU. For smaller size of A, we take this same placement because it can achieve better performance. The reason is that the binary search range is larger on A than on any other segment levels, and binary search among large ranges is not efficient on the GPU.

Parametric auto-fitting: We focus on optimizing the index in this parametric auto-fitting module, which mainly depends on the choice of ϵ to control the number of segments in our index. The number of segments has a significant impact on space occupancy and query performance. Meanwhile, a varying ϵ itself can influence the search time when pinpointing the accurate position in the range of length 2ϵ. Therefore, the complexity of time and space is complicated in relation to ϵ. In our design, we first conduct a theoretical analysis and then choose an optimal

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

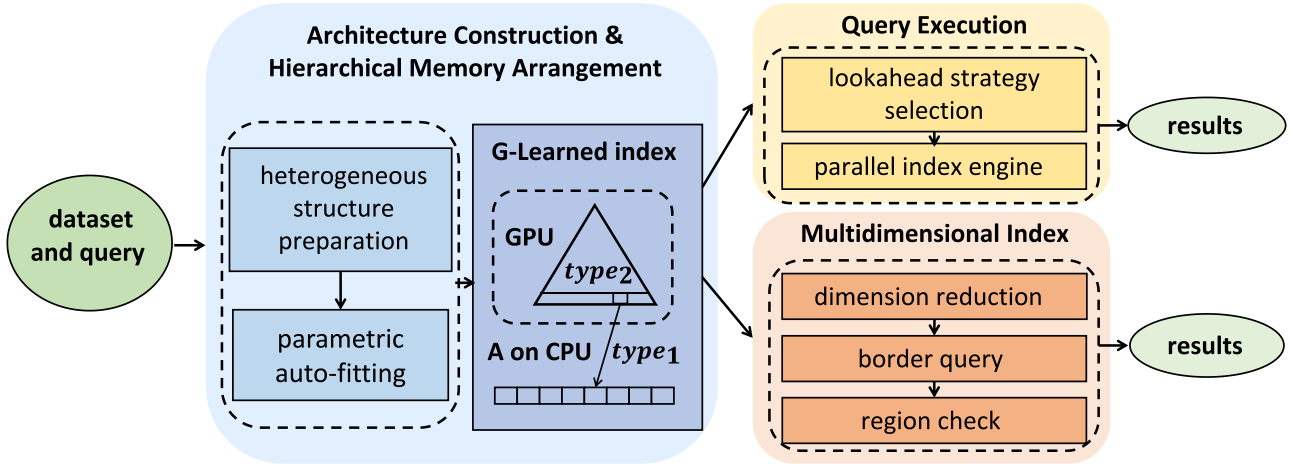

Fig. 4. Overview of G-Learned Index.

ϵ. The theoretical analysis follows two steps. First, we look into the procedures it takes for indexing a key k in the model and how the choice of ϵ can affect those procedures. In the second step, we provide a detailed analysis to support the division of the procedures theoretically.

In the first step, we divide the procedures of a searching process for k into two types. One ($type_1$ procedures in Fig. 4) consists of two procedures: 1) a mapping from the corresponding segment in the last level to a predicted position in A with an error of ϵ on the GPU, and 2) a CPU procedure of pinpointing the final position of k. The other ($type_2$ procedures in Fig. 4) contains procedures beginning at the root segment (which completes a level) and ending with one of the segments at the last level. These procedures are all conducted on the GPU. To sum up, a G-Learned Index structure is made up of these two types of procedures, as well as the two parts discussed in heterogeneous structure preparation.

In the second step, we investigate the effect of varying ϵ on $type_1$ and $type_2$ procedures. In the case of $type_1$ procedures, we first consider the factors of space occupancy. A large ϵ can reduce the number of segments at the bottom level, saving load costs. Furthermore, fewer segments can be stored in the limited GPU shared memory, resulting in more efficient data accesses and thus better time performance. In the case of $type_2$ procedures, we focus on the factors of time efficacy. A small ϵ is preferred because a smaller ϵ leads to a smaller positioning range, resulting in fewer operation discrepancies and less waiting time. This eventually leads to minimal branch divergence, which makes the best of GPU efficiency. Based on the previous discussions, we find that the two procedures have opposing preferences for ϵ. Therefore, we design adaptive auto-fitting for two separate ϵs corresponding to the two types of procedures. We define ϵ_1 as the ϵ used for generating the last level of segments from array A, and define ϵ_2 as the ϵ for internally building segments recursively from the bottom level up to the root segment. Theoretically, we prefer a large ϵ_1 and a small ϵ_2 (in fact, $\epsilon_2 = 4$ yields a competent result). Further tuning and investigation of ϵ_1 are explored in Section VI.

Parallel PLA generation: We use the PLA model [29] to generate the segments that cover the array A. Now, we explore the possibilities for building the segments in parallel.

We apply an $O(n)$ greedy algorithm in PLA for constructing the ϵ-approximate segments for A, where ϵ is given as a parameter. First, we create a Cartesian plane containing points $(k_i, rank(k_i))$ for all k_i in array A. Second, starting from the leftmost point to the rest of points on the right in order, a convex hull is maintained for a set of points until the height of a rectangle enclosing the convex hull exceeds 2ϵ. Then, we obtain a *segment* with corresponding $slope, intercept$, and key (key denotes the value of the leftmost point, which is k_1 here) for the points in the set. This process is repeated until all *segments* are created. We hereby describe our observation by first giving the following definition.

Definition 1: Let P be the point set $\{(k_i, rank(k_i))\}$ sorted according to their x-coordinates. A segment generation problem (SGP) takes the sorted points in P and an initial status (st, le) as inputs. The le clarifies the covering range of the current segment. Namely, it starts at the leth point, and st denotes the next point to process. Accordingly, the SGP (A, st, le) outputs the minimum number of segments needed in PLA-model for covering the points starting from the stth with a current segment status le.

Obviously, constructing the PLA-model on A can be viewed as solving the SGP$(P, 1, 1)$, where $P = \{(k_i, rank(k_i))\}$ for all k_i in A, and a segment starts from and covers only the first point.

Lemma 1: Satisfying the optimization principle and Markov property, we design an $O(n^3)$ dynamic programming (DP) strategy by defining $dp_{i,j}$ as follows. Given a SGP$(P, 1, 1)$, a minimum of $dp_{i,j}$ segments is needed to cover the first i points, with the current segment starting from the point whose $rank = j$. All $dp_{*,*}$ are initialized to ∞ except $dp_{st,le} = 1$. We also maintain a $f_i = \min\{dp_{i,*}\}$ for each i. The algorithm runs by following $dp_{i,j} = dp_{i-1,j}$ if the segment starting from the point j can be extended to i. Otherwise, $dp_{i,j} = \min\{f_k + 1\}$ with $k < i$. In this case, the $(k+1)$th point to the ith point must satisfy the condition to form a segment, because we are forming a new segment. Note that f_n represents the final result.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

Both the greedy algorithm and the above DP strategy [34], [36] achieve the same optimal number of segments. We consider the problem from the DP strategy.

We focus on the matrix C_1 where $c_{i,j} = dp_{i,j}$ for SGP1(P, st, le). We consider the corresponding matrix for SGP2(P, st, le_2) with $le_2 > le$. First, we obtain a copy of C_1 for C_2. Then, we set dp_{st,le_2} to 1 and update C_2. Now, the dp matrix C_2 for SGP2 is going to, in a sense, obtain results no worse than C_1. The copy operation is reasonable because the same state transition routes hold for two matrices. At this time, a looser initial status of $dp_{st,le_2} = 1$ is possibly able to bring a better result in f_n. This finding assists to bring the following theorem.

Theorem 1: Let P be divided into $\omega + 1$ sections, each running PLA-model in parallel before the results are merged together to complete the segment generation ($\omega \geq 1$). This adds at most ω segments on top of the segments needed when we perform the PLA-model on the whole P.

Proof: Consider situations where $\omega = 1$ and it is easy to apply the same procedure to prove for $\omega > 1$. If the split happens where it should be, no extra segments are encountered. In the other case, when performing PLA-model, suppose we are now at the $(cur - 1)$th point with a current segment starting from the loth point. However, we are unable to extend the segment to the curth point due to a split occurring between the $(cur - 1)$th and curth points. The post-split situation fits into an SGP1, where $st = cur$ and $le = cur$. Instead, we regard the original construction process, in the absence of a split, as an SGP2(cur, lo). We obtain an additional $\omega = 1$ segment by deliberately disconnecting the current segment in the parallel PLA-model, regarding it as an extra one compared to the original PLA-model. In this way, we are turning an SGP2 into SGP1. Since SGP1 achieves results no worse than SGP2, we prove the theorem that at most an extra of $\omega = 1$ segment is created by the parallel PLA-model. □

Experimental results have shown that parallel computing significantly reduces the construction time of the PLA model, reducing it from a few seconds to less than one second. However, we cannot allocate a large number of threads to PLA model execution. For example, if the PLA model generates 50 "segments" from a dataset of one million keys, using 10,000 threads for construction can result in an additional ten thousand "segments", consequently leading to a significant degradation in query performance.

B. Query Execution

In this part, we solve the index problem in a batch search context. We use batch granularity because we focus on bulk queries, which can benefit from thousands of GPU cores to index, resulting in massive acceleration.

General design: The parallel query operations work as follows. First, following the heterogeneous design, segment levels are built based on array A and loaded to the GPU device. Second, the queries are initially transferred from the CPU to the GPU. During indexing, each GPU thread independently retrieves individual queries according to the query ID and performs the required task in parallel. Third, a predicted position is returned from GPU for each query and the final position is pinpointed on the CPU.

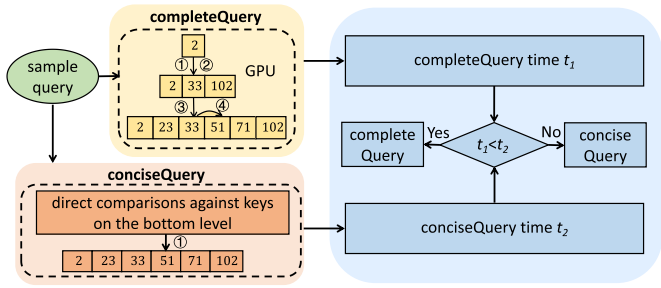

Fig. 5. Lookahead strategy selection module. A query sample is selected and follows *completeQuery* and *conciseQuery*.

Detailed design: Query operations for a single task are performed recursively in GPU from the root segment to the bottom level in G-Learned Index. At each level, a thread uses a referring segment passed down from the upper level to estimate the segment in the next level. Then, a search in a range size of $2\epsilon_2$ is performed to find the right-most segment whose key remains no larger than k. Given that the result position in the range of each thread can differ from one another, causing branch divergence, this operation leads to long time latency. A larger ϵ_2 means more branches. Considering the large number of queries executing in parallel, we select an ϵ_2 that is small enough ($\epsilon_2 = 4$) to compensate the overhead caused by divergent data accesses.

To further improve performance, we design a *lookahead strategy selection* as shown in Fig. 5. We find that for specific cases, such as segments with only three levels, removing the upper levels and conducting direct traversal check search on the bottom level can produce better query time results. This is because we need fewer memory transactions. For this reason, we build the index and use an auto-tuning *lookahead strategy selection* process. The process of selection is as follows. First, we select a sample query at random for the test query. Second, we track the time spent on *completeQuery*, namely indexing this sample query on all levels of segments. Third, we time the *conciseQuery*, i.e., conducting direct comparisons on each key in the last level to find the right-most one smaller than or equivalent to k. Finally, we compare the time spent in the second and third steps and select a more efficient strategy.

Example: In Fig. 2, we set ϵ_1 and ϵ_2 to 1. The upper three levels of segments are generated and placed on the GPU, and the array A stays on CPU. Before processing batch queries, we initially pick up the first batch as a sample query set. Assume the batch size is 2^{12}. Then, we launch 2^{12} threads for the sample query set where each thread solves one query. We next launch *completeQuery* and *conciseQuery* in turn as shown in Fig. 5. Suppose there is a thread assigned to index $x = 58$. For *completeQuery*, the thread follows the four procedures as discussed in the example of the PGM-index in Section II-A, and determines the accurate position in the last level of segments denoted by 51. For *conciseQuery*, the only procedure is that the thread directly conducts comparison against keys on the last level of segments and finds that segment. Then, using the segment

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Query Execution.

1: **function** completeQuery($A, n, \epsilon_1, \epsilon_2, levels, k$)
2: $pos = f_r(k)$, where $r = levels[0][0]$
3: **for** $i = 1$ **to** SIZE($levels$) $- 1$ **do**
4: $lo = \max\{pos - \epsilon_2, 0\}$
5: $hi = \min\{pos + \epsilon_2, \text{SIZE}(levels[i]) - 1\}$
6: $s =$ the rightmost segment s' in $levels[i][lo, hi]$ such that $s'.key \leq k$
7: $t =$ the segment right to s
8: $pos = \lfloor \min\{f_s(k), f_t(t.key)\} \rfloor$
9: $lo = \max\{pos - \epsilon_1, 0\}$
10: $hi = \min\{pos + \epsilon_1, n - 1\}$
11: **return** search for k in $A[lo, hi]$ ▷ Processed on CPU
12:
13: **function** conciseQuery($A, n, \epsilon_1, \epsilon_2, levels, k$)
14: $r = \text{SIZE}(levels) - 1$
15: $s =$ the rightmost segment s' in $levels[r]$ such that $s'.key \leq k$
16: $lo = \max\{pos - \epsilon_1, 0\}$
17: $hi = \min\{pos + \epsilon_1, n - 1\}$
18: **return** search for k in $A[lo, hi]$ ▷ Processed on CPU
19:
20: **function** Query($A, n, \epsilon_1, \epsilon_2, levels, k$)
21: load $levels$ to GPUs
22: $p = \arg\max_{p \in P} p(k)$, where $P = \{\text{completeQuery, conciseQuery}\}$
23: **return** $p(k)$ ▷Each thread assigned for a query task

denoted by 51, the thread returns 12 to CPU as the estimated position of x in A. The processing times of t_1 and t_2 are recorded for *completeQuery* and *conciseQuery*, respectively. In this case t_1 is larger than t_2, so G-Learned Index chooses *conciseQuery* as the query strategy. G-Learned Index then executes batch queries using *conciseQuery*. GPU processes queries in a batch and returns the estimated positions to CPU. Collecting all the 2^{12} estimated positions, the CPU processor then conducts binary searches. For the query $x = 58$ for instance, CPU searches among $A[12 - \epsilon, 12 + \epsilon]$ to determine the final position of x in A.

Algorithm: We show the pseudocode of the query execution of G-Learned Index in Algorithm 1, which has four major inputs: 1) pre-trained parameters of ϵ_1 and ϵ_2 in the construction phase, which can achieve high query performance, 2) $levels$ produced by PGM-index, 3) array A containing n keys, and 4) bulk query k. The workflow is as follows. First, $levels$ are transferred to GPUs (Line 21),. Second, we then conduct the *lookahead strategy selection* in Line 22, and then apply k to that choice. Third, we do queries on GPU where $type_1$ procedures are performed (Lines 2-10 and Lines 14-17). Finally, $type_2$ procedures are executed on CPUs (Lines 11 and 18).

Theoretical analysis: Given an array consisting an ordered sequence of $k_i (i = 0, \ldots, n - 1)$, the PLA-model in PGM-index is capable of determining a minimum number m_{opt} of segments for the original array (denoted as A), where $m_{opt} \leq n/(2\epsilon_1)$. For convenience, we use m instead of m_{opt}, and draw the following theorem.

Theorem 2: G-Learned Index with parameters ϵ_1 and ϵ_2 indexes the array A in $\Theta(n + m)$ space and answers queries in amortized $O((\log \epsilon_1)/\omega_1 + \log \epsilon_2(\log_c m)/\omega_2)$ time and $\Theta(\epsilon_1/B + m/(\omega_2 B))$ I/Os with $O(\epsilon_2 \log_c m)$ memory transactions on GPUs. In the context of the above claim, $c \geq 2\epsilon_2$ is the fan-out achieved between the neighboring levels, ω_1 and ω_2 denote the granularity on CPU and GPU, respectively, and B is the block size in the external-memory model.

Proof: A factor of c, which is more than $2\epsilon_2$, represents the reduction in the number of segments of the lower level of segments over the upper. Consequently, a total number $L = O(\log_c m)$ of levels are determined within $\sum_{l=0}^{L} m/c^l = \Theta(m)$ space required in GPUs. For a query task, amortized execution time is bounded by L binary searches with $L - 1$ on GPU and 1 search on CPU. Specifically, $L - 1$ binary searches are conducted on GPU over the $L - 1$ intervals with the size of at most $2\epsilon_2 + 1$. One search is performed on CPU over the range whose size is at most $2\epsilon_1 + 1$. As for the I/O costs, besides one binary search on CPU required for each query, I/O complexity also involves the extra expenses for loading segments from CPU and the transmission of results of batch queries back from GPU. In case of memory transactions on GPU, each thread is responsible for retrieving data that binary searches need in the device memory. □

As a result, each query traverses an equivalent number of levels, ensuring that the upper bound of latency for different queries remains consistent.

C. Data Structures

The data structures of G-Learned Index include a hierarchical self-maintained memory, positioning data structure, and prefix trees for indexing strings.

Hierarchical memory arrangement: When performing query operations, we assign one query task to a thread, and each thread accesses the segments in G-Learned Index. Therefore, data retrieval efficiency influences query performance on GPU, ultimately affecting total time cost. Fortunately, we can make full use of the GPU memory hierarchy to build our index structure. Constant memory located on GPUs can be accessed by all threads on the device and has a limited space of 4 KB, whereas shared memory is a programmable cache on individual SMs and has a size of 64 KB of our platform. Our memory is managed by first loading segment levels to GPU global memory. We then let each block in an SM occupies a copy of the structure in the shared memory if the structure does not exceed the capacity of the shared memory. This takes advantage of shared memory's faster access speed compared to global memory by retrieving data directly from the shared memory for each thread. Another suboptimal choice is to use constant memory for storage pool, which can provide moderate data access speedup. The reason is that constant memory resides on the same chip as global memory, and thus fails to produce the outstanding performance of shared memory.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

Data structure of positioning within the ϵ range: A search operation within an interval for each segment level on GPU whose size is at most 2ϵ ($2\epsilon_2$ specifically) is the key step for query execution on a single thread. This is achieved by a binary search in our design. The binary search outperforms a traversal search because it reduces the number of comparisons from 2ϵ at worst to a fixed $\log(2\epsilon)$. A fixed operation pattern means the avoidance of branch divergence between threads. Although with probability $p = \log(2\epsilon)/(2\epsilon)$, a traversal search behaves better than binary search (approximated to $1/2$ when ϵ is small), this value is diminished by a SIMD pattern of w threads working on a batch. Specifically, the probability is reduced to $p^w = (\log(2\epsilon)/(2\epsilon))^w$ because the processing of a batch finishes only when the last thread in that batch completes its job. For example, assume the concurrent queries be grouped in $m = 10^5$ SIMD batches. Following Chernoff bound, then, the probability that 100 out of 10^5 SIMD batches can obtain better performance on traversal search is smaller than $2e^{-m/(3*10^6 p^w)}$, i.e., $2e^{-1000}$ with $w = 16$, $\epsilon = 4$, and $m = 10^5$.

Indexing strings: Indexing string is commonly used by many databases [6], [29]. However, the existing PGM-index and many GPU-based indexes focus on only indexing numerical values [7], [18], [29]. Extending the model for strings introduces plenty of difficulties. The primary concern is how to convert strings to features of our model, a process known as tokenization. Fortunately, studies [6], [50] on modeling strings have been able to facilitate our practice. Overall, we adopt the advantage of a prefix tree formulated on the dataset dictionary. In an efficient and expressive solution, we represent a string as a feature key $u.key \in \mathbb{R}$, where $u \in T$ and T is the prefix tree generated on the given string dataset. We then follow the same efficient heterogeneous modeling as we perform for numerical values. The difference is that the input is the feature key representation of a string.

D. Updates in G-Learned Index

In this section, we consider random updates, including insertions and deletions, in G-Learned Index. For *insert*, for example, it is more difficult to develop compared to traditional indexes. Specifically, we adopt the logarithmic method for inserts at arbitrary positions in the original array [29]. That is to say, we maintain $\log(n)$ containers $S_0, S_1, \ldots, S_k (k = \log(n))$ at any time having volumes of $2^0, 2^1, \ldots, 2^k$ elements, respectively. In each container, we build a G-Learned Index. When an element is inserted, we first attempt to add it to S_0. If the added element makes the total number of elements exceed the volume of the container, neighbor containers are merged in a cascade way. For example, assume S_i is the first empty container. After inserting the key x, we build a new G-Learned Index over the merged container $S_i = S_0 \cup \ldots \cup S_{i-1} \cup \{x\}$. Overall, we pay $O(\log n)$ amortized time per insertion. To delete a key, we add a tombstone value associated with the key x.

When indexing batch queries, we allocate $\log(n)$ threads for one query and let the threads work simultaneously. There are two levels of parallelism. To index the key x, a group of $log(n)$ threads work in parallel, with thread i indexing x in S_i, where $i = 0, \ldots, \log(n)$. At the same time, different thread groups work in parallel to index different keys.

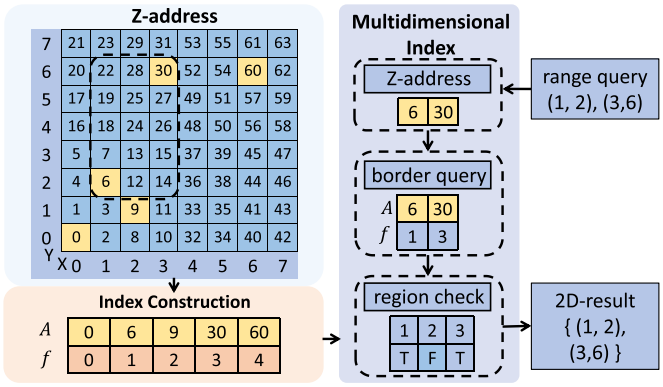

Fig. 6. Example of range query of G-Learned Index.

E. Multidimensional Index

In this part, we discuss our PGM-index extension, which enables it to index spatial objects and process range queries on GPUs. Overall, we expect to learn the distribution of objects scattered in two-dimensional space, which is one of the key elements in a learned index. For one-dimensional situations, reals are ordered intrinsically, so we can learn the CDF function of the keys, whereas this case is not applicable to a spatial context. This makes learning data in a spatial context difficult. One way is to apply a conversion for a set of keys mapped from two-dimensional space to one-dimensional space. Since real numbers are fundamentally ordered, this *dimension reduction* can be obtained by algorithmically assigning orders to points in high-dimensional spaces. In our case, we use a Z-order strategy to convert two-dimensional points to one-dimensional Z-addresses [51] to facilitate data regularity learning. The following property of Z-order is very useful. Given a search range in the Cartesian plane, which is a rectangle framed by the bottom-left and top-right *corner points*, Z-order maintains a monotonicity that all the objects lying in the rectangle are guaranteed to be located within a one-dimensional Z-address range. This range is enclosed by the Z-addresses derived from the two spatial corner points.

General design: G-Learned Index range query works as follows. First, all objects in the spatial map are stored in order in a new array after dimension reduction. Second, we build our heterogeneous model based on the new sorted array. Third, a search range is enclosed by the two bottom-left and top-right *corner points* of a rectangle. After this, the traversal range is encircled by the *corner points* that have been transformed to Z-address representations and indexed by GPU query execution. Finally, we traverse the objects in that range to extract those objects that are also contained in the originally given rectangle. The monotonicity of Z-order ensures that the targeted objects in the spatial map together form a subset of the one-dimensional traversal range.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: Range Query for Multidimensional Index.

```
1:  function rangeQuery(Map, n, ε1, ε2, p1, p2)
2:    A = an empty dynamic array
3:    for i = 0 to SIZE(Map) - 1 do
4:      A[i] = Z-address(Map[i])
5:    conduct in-place sort of A according to key value
6:    levels = BUILD-PGM-INDEX(A, n, ε)
7:    st = Query(A, n, ε1, ε2, levels, p1)
8:    ed = Query(A, n, ε1, ε2, levels, p2)
9:    result = an empty dynamic array; count = 0
10:   for i = st to ed do
11:     if checkRegion(p1, p2, A[i]) then
12:       result[count] = A[i]; count = count + 1
13:   return result
```

Example: We show an example of range query in Fig. 6. In the dataset $\{(0,0),(1,2),(2,1),(3,6),(6,6)\}$, we perform range query to index objects within the range $(1,2)$ and $(3,6)$. We first project each point to its Z-address representation, which is indicated in every grid for each individual point, so that a Z-address $\{0,6,9,30,60\}$ of respective points in the dataset can be occupied and stored in array A. We then build our G-Learned Index based on A, and with $\epsilon = 0$, we develop a function mapping the original keys to their positions in A as f. We search for the keys in the range $(1,2)$ and $(3,6)$ by first obtaining their Z-addresses, which are 6 and 30, respectively, and then determining the starting position ($f(6) = 1$) and ending position ($f(30) = 3$) of the traversal range in A. Finally, we traverse the keys in that range in A to extract all points with two-dimensional positions satisfying the range query of $(1,2)$ and $(3,6)$. This is equivalent to $\{6,30\}$ in Z-address representation and $\{(1,2),(3,6)\}$ in the spacial map.

Detailed algorithm: The pseudocode is shown in Algorithm 2. Initial input data of spacial objects are stored in Map and a range query is indicated by p_1 and p_2 as two corners. First, representation transformation is applied in Line 4, resulting in a new array A for sorting the data and learning the distribution, respectively (see Lines 5 and 6). We then turn to the parallel query execution of G-Learned Index to index the positions of $p_1$and p_2 in A by Lines 7 and 8. Next, we extract all the answer objects by traversing the objects between $A[st]$ and $A[ed]$ and checking whether they are situated in the original rectangle in Map. Note that it is possible for objects to reside in the range between $A[st]$ and $A[ed]$ but not to appear in the rectangle.

Theoretical analysis: We next deploy analysis in point query and prove bounds on the time and space complexities of multi-dimensional index.

Theorem 3: Assume A is an ordered array of n keys converted from objects in two-dimensional space, where each dimension is drawn from universe $\mathcal{U}$ and $\epsilon > 1$ is a fixed parameter. In the transformation stage, $\Theta(n)$ time and $\Theta(n/B)$ I/Os are required where B is the block size transferred between memory levels in external-memory model. The stage concerning PGM-index construction consumes $\Theta(m)$ in space and $O(n)$ in time, where m denotes the minimum number of ϵ-approximate segments. The multidimensional index with parameter ϵ indexes a range between p_1 and p_2 in the array A in an extra $O(K)$ time in addition to point query, where K is the number of objects satisfying the query range, with a minimum of extra $\Theta(K/B)$ I/Os required.

Proof: Space reduction can be efficiently conducted because of continuous access pattern. The query time $O(n)$ is reported because a factor of "$2\log w$" for Z-address transformation is dismissed as a constant factor, where w refers to the range of the key universe $\mathcal{U}$. We observe a continuous access pattern of traversal check in the range between $A[st]$ and $A[ed]$, so that the index time complexity and I/O costs are bounded assuming st and ed are evenly distributed in the range of $[0, n-1]$. □

F. Implementation

We integrate G-Learned Index and PGM-index [29] implementations together, to help users use the learned index in a variety of heterogeneous application scenarios. The G-Learned Index consists of two major parts. 1) a CPU-located part handling data input and work assignments on the CPU, and 2) a GPU-situated working kernel mainly responsible for acceleration. G-Learned Index is written in C++ and CUDA, while OpenMP has also been used to facilitate our development. First, the queries are transferred from the CPU side to the GPU side according to the batch size through the CUDA memory copy API. Second, after each thread retrieves the query it is responsible for, the kernel initiates query execution to complete indexing on the GPU side and subsequently transfers the results back to the CPU. Third, the CPU processes the results with minimal overhead and presents the final index to the user. With a user-friendly interface, it is easily applicable to databases in both one-dimensional and spatial contexts.

VI. Evaluation

A. Experimental Setup

Methodology: We compare G-Learned Index to four methods. The first method is "ALEX" [33], the most recent updatable Recursive Model Index (RMI). The second method is the PGM-index [29], which is the state-of-the-art learned index denoted as "PGM". However, we find that the original PGM is a sequential version that uses only one thread on CPU. For fair comparison, we use OpenMP to develop a parallel PGM-index version that can take advantage of CPU's parallelism. Accordingly, the third method is a parallel version of the PGM-index, denoted as "PGM (parallel)". The fourth method is the state-of-the-art GPU-based B-Tree [7], denoted as "GpuBTree". Our GPU-based learned index is denoted as "G-Learned Index". We synchronize the issued CUDA API calls before stopping the timer.

Datasets: We refer to SOSD [52] for evaluation datasets. SOSD is a recent proposal on benchmarking learned index, which encompasses a wide range of datasets. In detail, we use eight datasets. Four datasets come from the real world, including *Books* that contains popularity data of 200 M book sales on Amazon [53], *Facebook* that represents 200 M sampled IDs of users drawn from the Facebook dataset [52], *Street* that

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

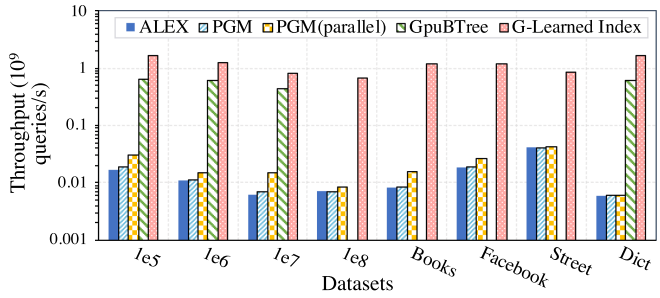

Fig. 7. Throughput of different methods.

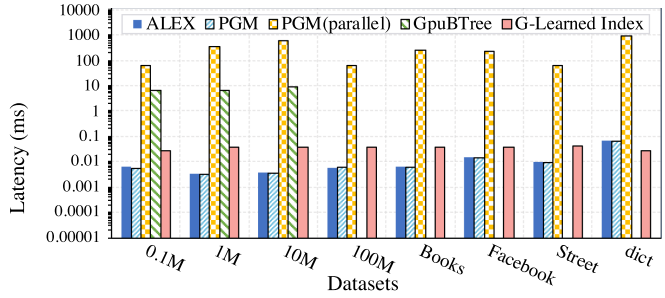

Fig. 8. Latency of different methods.

consists of 800 M location information out of OpenStreetMap presented in Google S2 CellIds [54], and a string dataset *Dict* that consists of 341 K English words from a dictionary [15]. Four synthetic datasets of 0.1 M, 1 M, 10 M, and 100 M are generated randomly, denoted as "*1e5*", "*1e6*", "*1e7*", and "*1e8*", respectively. "*1e5*" is generated according to the uniform distribution in the interval $[0, u)$, "*1e6*" to the normal distribution with mean μ and standard deviation σ, "*1e7*" to the exponential distribution with parameter λ, and "*1e8*" is randomly generated. We set $u = 2^{32}, \mu = 2^{16}, \sigma = 2^{10}$, and $\lambda = 2^{-5}$. The randomly generated datasets involve a high level of uncertainty and reveal a wide range of difficult-to-identify data patterns. As a result, we believe that these datasets present challenges for learned indexes in grasping data schemes.

Platform: Our experiments are conducted on a platform equipped with an Intel i9-9900 k CPU and an NVIDIA GEFORCE RTX 2080 TI GPU. The CPU has 8 cores, each of which can support 2 threads. The GPU has 4,352 cores of Turing architecture with a computing capability of 7.5. The GPU can achieve a maximum memory bandwidth of 616 GB/s, 0.4 tera floating-point operations per second (TFLOPS) on double-precision, and 13 TFLOPS on single-precision.

B. Performance

We compare the performance of various methods in terms of both throughput and latency. We include the PCIe data transfer time between CPU and GPU for G-Learned Index. For each dataset, we execute a different set of 2^{22} key lookup queries. Each query is assigned with a random key. A batch size of 4096 KB is used and ϵ is set to 128.

Throughput: We show the throughput of different methods in Fig. 7, and have the following observations.

First, G-Learned Index achieves the highest throughput among all methods. It can achieve an average of 1.2×10^9 queries per second, which is 174× the performance of PGM-index, and 107× the performance of its parallel version. Compared to the state-of-the-art GPU-based B-Tree, G-Learned Index still achieves an average speedup of 1.9×. We only have the results of the four small datasets for GPU-based B-Tree, as the other datasets resulted in errors due to illegal memory accesses. The learned index has a more compact learned structure and more succinct representation (PGM-index can achieve 4× speedups over the B-Tree with 1140× less space occupancy) compared to the traditional B Tree. Therefore, G-Learned Index can outperform GPU B-Tree with careful heterogeneous architecture optimization.

Second, we find that parallel technology proves highly effective in accelerating the learned index. The CPU platform can accommodate up to eight cores, and the parallel version of the PGM-index demonstrates an average 52% higher throughput compared to the sequential version. GPUs provide much more lightweight cores. With proper optimizations, including coalesced access, minimization of branch divergence, and effective use of memory hierarchy, as discussed in Section V, G-Learned Index can provide a further significant performance improvement.

Third, we can see from Fig. 7 that the G-Learned Index throughputs are comparable on different datasets. When G-Learned Index processes a greater number of elements, its performance stays stable (statistically from 0.4% to 9.6% reduction at most). The reason is that a $segment$ can carry large volumes of information. This suggests promising scalability of G-Learned Index. However, we observe that the parallel PGM-index and GPU-based B-Tree suffer significant time efficacy loss when the number of elements to maintain is increased (from *1e5* to *1e7*). The parallel PGM-index and GPU-based B-Tree, in particular, see a reduction in throughput from *1e5* to *1e7* by 19% to 30%, and 14% to 43%, respectively. This is because the functions of the C++ STL library used in PGM-index are sensitive to the original dataset size. Instead, for G-Learned Index, GPU kernel operations are processed on segments, and ensure that the growing size of datasets can lead to a sub-linear rise in the number of segments, kernel operation time.[1] As for GPU-based B-Tree, throughput is hinged on the height of the B-Tree, mainly influenced by the expanding size of the index structure linear in the number of input keys.

Latency: We show the latency results of different methods in Fig. 8. We define latency as the end-to-end time for a query to complete from the beginning to the end. For methods using parallel batching, a query ends when all queries in the same batch finish. The batch size is 2^{10}. We have the following observations.

First, G-Learned Index exhibits an average latency of less than 0.04 ms. Although G-Learned Index targets high-throughput scenarios, its latency is close to that of ALEX and PGM among

[1] A proven loose bound is $\log(N \times 1/(2\epsilon))$, but empirically quantified investigation shows much better results. Our case here is a compelling justification.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

most workloads. For *dict* dataset, G-Learned Index can provide even shorter latency. This suggests that the batch query strategy does not significantly affect the latency, even at batch granularity.

Second, we can see that the GPU parallelism can provide much lower latency than the CPU parallel version. One reason is that the GPU is equipped with GDDR6 memory, which has much higher bandwidth and lower latency. Another reason is that workloads are scheduled more evenly on different GPU SMs.

Third, G-Learned Index latencies are similar across datasets. In fact, its reassuring latency on datasets of various sizes is mainly due to the predictable and balanced performance of various queries on GPU kernels. The minimum branch divergence is achieved by G-Learned Index. Consequently, only a minimal amount of additional time is needed to wait for the last thread in a *warp* to finish its job.

Currently, the CPU and the GPU are connected via PCIe. When advanced connection is enabled, we can achieve higher performance gains. The data transmission and computation can also be pipelined. It is worth noting that the CPU running time accounts for less than 10% of the total latency and is not sensitive to the different choices of ϵ in experiments. Therefore, we choose ϵ mainly based on GPU. For a different generation of GPU, e.g., NVIDIA GEFORCE 3090, 128 is chosen based on the *1e7* dataset in experiments.

Applying G-Learned Index to other scenarios: As discussed in Section III-B, G-Learned Index can be applied to many applications. For the use case of parallel nested loop join [28], assume we have two tables: the outer table R and the inner table S. We add one million records to table R; $match\,rate$ is set to 100% and $index\,rep$ is five. The index is built on S. We then conduct experiments on both the CPU learned index and G-Learned Index. With the help of G-Learned Index, the parallel nested loop join with G-Learned Index achieves 101.2× performance speedup over its original version, and also achieves 21.8× speedup compared to the CPU-learned-index version. For the use case of pattern matching on dataset *Street*, we have an 16.6× faster end-to-end total execution. For the use case of feature store task on one million taxi rides in New York City [49], suppose we need to select 70% of the records as training set randomly; with concurrent batch indexing, the end-to-end processing time speedup is 11.3×.

C. Space Savings

In this part, we investigate the space savings of G-Learned Index by tracking the footprint of our implementation, which to a large extent depends on the amount of space required to store the bottom level of segments. We compare the size of G-Learned Index with that of GPU-based B-Tree.

Overall, experiments show that given $\epsilon = 128$, for the four real-world datasets, G-Learned Index occupies 36 KB, 5.9 MB, 16 MB, and 0.23 KB, thus saving space by $4258\times$, $528\times$, $783\times$, and $11333\times$, respectively. In addition, for the synthetic datasets, G-Learned Index saves 4 orders of magnitudes space on average. The reason is that *segments* of the index are compact structures that take up space sub-linear in the number of keys in a dataset.

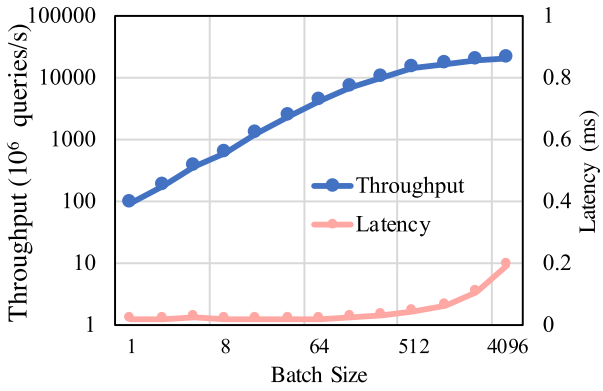

Fig. 9. Performance of G-Learned Index with various batch sizes.

To explore the influence of ϵ on the space savings, we vary ϵ from 32 to 128, 512, and 2048 over the eight datasets. The average space occupancy of G-Learned Index for different ϵ is 12 MB, 2.7 MB, 604 KB, and 108 KB, resulting in space savings of $1130\times$, $1.4e4\times$, $1.5e5\times$, and $9.4e4\times$, respectively. The space benefits increase along with ϵ. The reason is that the PLA model generates *segments* dependent on choices of ϵ, which denotes the tolerance of precision that is allowed when using a *segment* to approximate the position of a key. Specifically, as ϵ increases, fewer *segments* are needed resulting in greater space savings.

D. Design Tradeoffs

Batch size analysis: Next, our navigation is concerned with the effect of different batch sizes, namely the number of queries completed within a batch. Because a large number of queries arrive at a close time, we organize these queries at batch granularity. Combining queries into a batch aids in making full usage of computing cores on GPU. To investigate the effect of batch size on performance, we vary the batch size from 1 KB to 4096 KB, and measure the throughput and latency of G-Learned Index.

We show the performance based on various batch sizes in Fig. 9. We have the following observations. First, the throughput increases as the batch size grows. When the batch size is less than 512 KB, we can see that the throughput grows steadily. Following that, the growth of throughput slows down, indicating that the utilization of parallel GPU resources is close to saturation, and there is contention for memory access that arises at high batch sizes. Second, the latency remains low until the batch size reaches 512 KB. After that, the newly arrived batch cannot be processed immediately, and the processing time increases. Accordingly, there is a noticeable difference in latency after 512 KB. Third, we conclude that properly increasing the batch size when resources are abundant can increase the throughput without affecting the latency. We are also interested in determining the number of queries required to achieve acceleration over CPU-based methods. G-Learned Index has been shown to achieve higher throughput for datasets larger than *1e6* when $batch_size$ is set to 256 or higher. In cases where query batches exhibit superior latency on CPUs, our system can leverage the CPU parallel processing engine.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

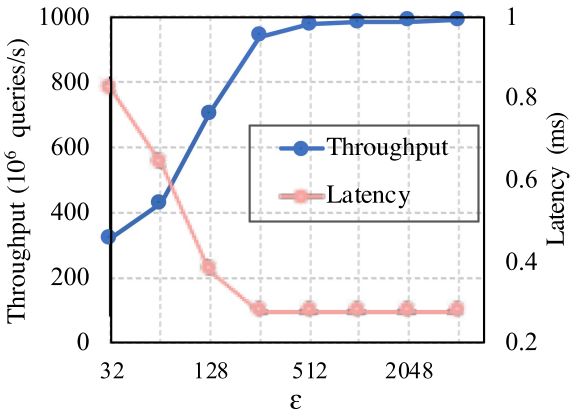

Fig. 10. Influence of ϵ on throughput and latency.

Influence of different configurations: We report the influence from different ϵ configurations on ten million keys in Fig. 10. First, we can see that the throughput grows dramatically when ϵ increases from 32 to 256. This is because a larger ϵ leads to a significant decrease in the number of *segments* at the bottom level, which saves a considerable amount of time during data retrieval and query execution on *conciseQuery*. Although it takes more time for binary search on GPU to find the real position of a key, this time increase accounts for only a small portion of the total time consumed. When ϵ is larger than 256, the throughput does not increase significantly. The reason is that further increases in ϵ have less of an impact even as the number of segments decreases. Additionally, latencies behave in the opposite direction to throughputs. The latency starts out high for $\epsilon = 32$, and gradually decreases until it reaches 256. This is due to faster query execution when ϵ becomes larger. The choice of ϵ also affects index size. Generally, the space occupancy decreases as ϵ increases. If the space occupancy is small enough, the index can reside in the shared memory to accelerate querying. For the 1e5 dataset, we can always store the index in GPU shared memory for $\epsilon \geq 32$. For the *books* dataset, however, $\epsilon \geq 512$ is required for the index structure to reside in the shared memory.

Influence of memory hierarchy: Each GPU features a memory hierarchy. We can take advantage of the shared memory because the index structure on the GPU side can be shared and reused across threads. The index can be fetched from global memory to shared memory and all the threads in the CUDA block can use the same index structure in the shared memory. We investigate the memory hierarchy by explicitly assigning structures of segment levels for the *1e6* dataset with a varying ϵ to specific memory types. We measure the query time and show the performance results in Table I. Experiments show that storing the index structures in shared memory outperforms global memory by 5% to 18%, and up to 7% over constant memory. The reason is that shared memory, also known as programmer-controlled cache on the GPU, can provide faster read and write bandwidths. Besides, the pre-fetch from global memory to shared memory guarantees a maximum degree of coalescing.

Index construction: As discussed in Section V-A, G-Learned Index can be built on GPU in parallel. The index construction process on GPU of G-Learned Index is much faster than that on CPU. To construct the *1e7* dataset, for example, the CPU parallel version of PGM needs 592 ms while G-Learned Index needs only 118 ms. On average, G-Learned Index is 5.54× and 3.05× faster than PGM (parallel) and ALEX, respectively. Moreover, we find that the index construction of G-Learned Index on large datasets exhibits higher benefits. For example, G-Learned Index achieves 8.35× (686 ms against over 5700 ms) speedup in construction on the "*1e8*" dataset and 2.28× (2.24 ms over 5.12 ms) on the "*1e5*" dataset over PGM.

TABLE I
QUERY TIME WITH REGARD TO DIFFERENT TYPES OF MEMORIES ON THE GPU ($\boldsymbol{us}$)

Memory Type	ϵ			
	32	64	128	256
global memory	17.184	17.152	16.64	14.88
constant memory	15.104	15.104	14.4	14.176
shared memory	15.808	14.016	13.888	14.112

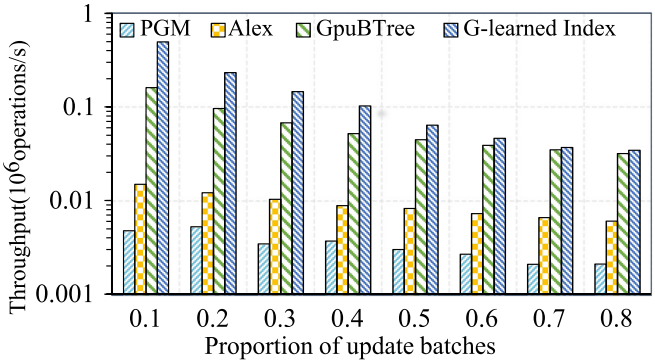

Fig. 11. Performance of random updates of different methods.

E. Random Updates

To measure the performance of random updates in G-Learned Index, we generate a dataset of *1e7* 32 b integers from $u(0, 2^{32})$ and simulate a dynamic scenario where 10^7 operations are randomly generated. Operations of insertions, deletions, and queries are executed in batches of 2^{12}. Among these batches, we vary the proportion of update operations to explore the performance of G-Learned Index in different situations. We set ϵ to 128, and compare G-Learned Index to the dynamic PGM-index, ALEX, and GPU B-Tree. We show the comparison results in Fig. 11. The horizontal axis represents the different proportions of update operations. Fig. 11 shows that G-Learned Index outperforms PGM, ALEX, and GPU B-Tree by an average of 32×, 11×, and 1.6× in terms of throughput, respectively.

F. Multidimensional Indexing

We have enabled the G-Learned Index in multidimensional contexts as well. In this part, we evaluate G-Learned Index

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

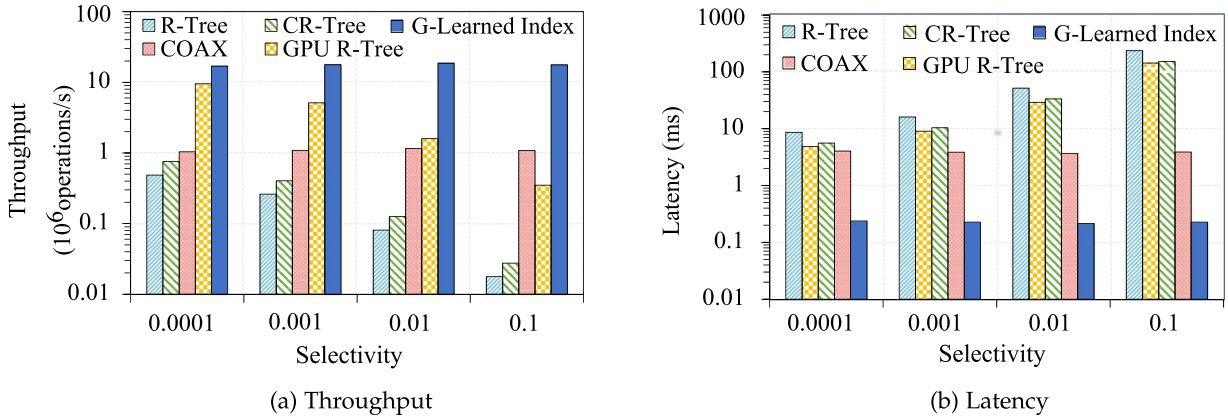

Fig. 12. Performance of multidimensional indexing on R-Tree and G-Learned Index. Selectivity is defined as a ratio of the number of objects selected to the total size of the dataset.

on range queries and compare it to R-Tree [55], GPU R-Tree [56], CR-Tree [57], and COAX [58]. R-Tree is the traditional spatial index suitable for range queries. GPU R-Tree is a multi-dimensional indexing R-Tree structure built on GPU architecture, whereas CR-Tree is a cache-aware R-Tree. COAX is a learned index for multidimensional data, which learns the correlation between the attributes of the dataset rather than the distribution of keys. We compare G-Learned Index with the R-Tree based multi-dimensional index in terms of throughput and latency. COAX is used for performance stability comparison. To create a multidimensional setting, we randomly generate a spatial dataset with 100,000 two-dimensional objects. We create 500 range queries at random. Each query is given a bottom-left and top-right corner, and spatial keys within the range are expected to be retrieved. To gain a comprehensive understanding of the multidimensional performance of G-Learned Index, we apply a *selectivity* parameter to generate range queries. This *selectivity* is a fixed value that represents the ratio of the number of objects chosen in a query to the total size of the dataset.

We report the performance of throughput and latency on different selectivities in Fig. 12. First, G-Learned Index has an average throughput of 17.7×10^6 queries/s, which is 152×, 8×, 98×, and 16× greater than that of R-Tree, GPU R-Tree, CR-Tree, and COAX, respectively. Second, G-Learned Index has a low latency of 0.23 s, which is 159×, 90×, 103×, and 17× lower than that of the R-Tree, GPU R-Tree, CR-Tree, and COAX. Third, we can see that the G-Learned Index's performance behavior is quite stable in terms of both throughput and latency. This is because the time spent on G-Learned Index does not increase significantly when *selectivity* increases. Specifically, the major time consumption is incurred by indexing the two corners, and the following asymptotic linear traversal time of *region check* makes little difference. It can be seen that COAX without tree structure is also stable, which verifies our conclusion. In contrast, the query time of R-tree appears to be dependent on *selectivity*, with longer search time as a result of traversing a larger proportion of the R-tree. That is to say, R-tree does not guarantee a worst-case time complexity, possibly $O(n \log n)$ where n is the number of keys.

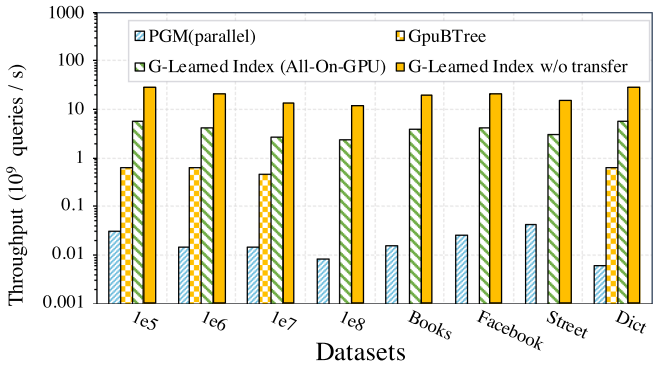

Fig. 13. Performance of different methods.

G. Additional Experiments

G-Learned Index with all data on GPU memory: Following the same setting in Section VI-B, we show the throughput of G-Learned Index for the dataset *1e7* where all the index structure and data are stored on GPU. G-Learned Index can achieve 396× speedups compared to the PGM-index (and 185× speedups against its CPU parallel version), and 6.12× speedups against GPU B-Tree.

Performance without data transfers between CPU and GPU: Fig. 13 illustrates the throughput and latency of G-Learned Index when excluding data transfers between the CPU and GPU. Fig. 13 shows that G-Learned Index can achieve 3050× higher throughput compared to the PGM-index (and 1880× higher than its CPU parallel version). The results are useful, especially for GPU-CPU integrated devices, where the CPU and GPU share the same memory [59].

Latency breakdown of CPU and GPU: We show the breakdown of the whole latency between the GPU side and the CPU side in Table II. Through Table II, we can see that the GPU side takes the majority of jobs and costs over 90% processing time. For example, the 2^{22} queries can be broken down to a detailed level for batch size = 1 KB. In detail, the upper layers take around 12 us (over 90% of the total execution time) for the *1e7* dataset, while the last layer takes less than 2 us.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

TABLE II
LATENCY BREAKDOWN OF CPU AND GPU FOR *1E7*

Batch Size (KB)	Total execution time (us)	GPU portion	CPU portion
1	12.7	91.2%	8.8%
4	13.2	89.9%	10.1%
16	14.1	90.2%	9.8%
64	14.4	90.6%	9.4%
128	19.2	93.0%	7.0%
256	21.0	93.6%	6.4%

VII. Related Work and Discussion

This section discusses related work of learned indexes and traditional indexes on GPU.

Learned indexes: The development, refinement, and application of learned indexes have been broadly studied in recent years [6], [13], [60], [61], [62], [63]. Kraska et al. [6] proposed the learned index, claiming that indexes are fundamentally trainable models to output the position of the input key in a sorted array. Many researchers have focused on improving the performance of learned indexes [14], [33], [51], [64], [65], [66]. Li et al. [32] and Qu et al. [16] constructed variations and extensions of RMI by excluding keys that caused large errors in the array, which are typically called outliers, storing them in a B-Tree and constructing RMI on the resting non-outliers. ALEX [33], [67], [68] focused on fitting a linear learned index into a tree design. In AIDEL [64], RMI is further improved with guaranteed latency. Xun et al. [69] implemented a simple index on GPU; however, their work neither supports index construction and updates, nor does it support range query for multidimensional index. The application of learned indexes is another central research topic [15], [70], [71], [72], [73]. By incorporating learned indexes to the hash table, learned hash is generated [6], [72]. Another category of learned index application is the learned bloom filter. Kraska et al. [6] and Macke et al. [74] used a binary classifier to replace the Bloom filter, and Mitzenmacher [60] provided a detailed analysis of the model size. Dai et al. [70] modified the learned Bloom filter using a predicted probability score. Other researchers used learned indexes to solve the heavy hitter problem. Hsu et al. [71] applied the learned indexes to the frequency estimation problems, and Zhang et al. [75] used machine learning to acquire important input patterns and proposed the Learned Augmented Sketch.

Traditional indexes on GPU: There is a large body of literature that addresses index structures on GPU. Many efforts have been made to improve both the traditional B-Tree and its variant, the B+-Tree [7], [8], [9], [76]. Awad et al. [7] and Kaczmarski [8] optimized the performance of B-Tree and B+-Tree on GPU. When queries arrive, the GPU B-Tree family uses different searching methods – completing a search query in a single thread [8], or locating a key by fetching all keys in a page in parallel to leverage GPU bandwidth [7]. G-Learned Index has different designs. First, G-Learned Index features a different model structure from GPU B-Tree. While G-Learned Index uses machine learning models as sub-components in the index to form a Directed Acyclic Graph (DAG), GPU B-Tree features nodes constructed as a tree. Second, G-Learned Index delves into the intricate relations of the influences of different parameters and tunes the model to achieve optimal performance. Specifically, we have parameters like ϵ to balance the model levels and additional search overhead. Third, G-Learned Index encapsulates further optimizations that exploit the collaboration between threads, such as the lookahead strategy. Other studies have been concerned with GPU performance gains using different index structures [18], [77], [78], [78], [79]. Liu et al. [80] proposed a lock-free parallel solution for the T-Tree, and GPU LSM [18] is implemented with a dynamic dictionary data structure for GPU using a set of sorted arrays. Lopresti et al. [77] developed a GPU permutation index for similarity search on databases with different data distribution, and Zhou et al. [81] succeeded in supporting batch queries. Attempts have also been made on GPUs to parallelize multidimensional index structures [17], [82], [83]. For example, G-Tree is a GPU-aware parallel R-Tree indexing method proposed by Kim et al. [83]. It combines the efficiency of the R-tree with the massive parallel processing power of the GPU. GPU R-Tree, implemented by You et al. [17], offers consistent and stable performance in high-dimensional space.

Discussion: First, one limitation of this work is its focus on PGM-index, while there may be acceleration opportunities in other learned index structures for GPUs, which we leave for future exploration. Second, our current system, G-Learned Index, is designed for the CPU-GPU heterogeneous platform, but there may be potential opportunities on other devices such as TPU. This suggests that additional architectural designs will be necessary to account for the properties of TPUs. Third, the current CPU database stores data on the CPU end. When considering a point query, a performance model may be necessary to maximize the benefits of G-Learned Index against the data transfer cost between different devices.

VIII. Conclusion

This paper proposes an efficient GPU-based learned index to enable efficient learned indexes on GPUs. This paper shows how learned indexes have been materialized on GPUs, and discusses the primary challenges in applying GPU in learned indexes. This is addressed by efficient performance-guided bandwidth utilization, optimizations of multi-thread organization, and memory dependency removal of the upper layers following the heterogeneous model accompanied with pre-trained parametric selection and on-the-fly strategy adaptation. Moreover, we provide a library to help users adopt our work in GPU-based data management systems. G-Learned Index achieves an average of 174 $\times$ speedup over the state-of-the-art method (and 107 $\times$ of its parallel version). It is also worth noting that compared to GPU-based B-Tree, G-Learned Index saves 4 $\times$ query time and takes up two orders of magnitude less space occupancy. These experiments show that using GPU in conjunction with machine learning in the index of database systems has tremendous promise.

References

[1] "How much data is created every day?," 2022. [Online]. Available: https://seedscientific.com/how-much-data-is-created-every-day/

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

[2] J. Bulao, "How much data is created every day in 2022?," 2022. [Online]. Available: https://techjury.net/blog/how-much-data-is-created-every-day/#gref
[3] G. Graefe, "B-tree indexes, interpolation search, and skew," in *Proc. 2nd Int. Workshop Data Manage*, New York, NY, USA, 2006, pp. 5–es, doi: 10.1145/1140402.1140409.
[4] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Proc. 6th Int. Conf. Very Large Data Bases*, 1980, pp. 212–223.
[5] K. Alexiou, D. Kossmann, and P.-R. Larson, "Adaptive range filters for cold data: Avoiding trips to siberia," in *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1714–1725, Sep. 2013.
[6] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2018, pp. 489–504, doi: 10.1145/3183713.3196909.
[7] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance GPU B-tree," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2019, pp. 145–157, doi: 10.1145/3293883.3295706.
[8] K. Kaczmarski, "B + -tree optimized for GPGPU," in *On the Move to Meaningful Internet Systems: OTM*, R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, Eds., 2012, pp. 843–854.
[9] K. Kaczmarski, "Experimental b+-tree for gpu," *ADBIS*, vol. 2, no. 11, p. 122, 2011.
[10] G. Wang, Y. Lei, Z. Zhang, and C. Peng, "A communication efficient ADMM-based distributed algorithm using two-dimensional torus grouping allreduce," *Data Sci. Eng.*, vol. 8, pp. 61–72, 2023.
[11] J. Wang, W. Pang, C. Weng, and A. Zhou, "D-cubicle: Boosting data transfer dynamically for large-scale analytical queries in single-GPU systems," *Front. Comput. Sci.*, vol. 7, 2023, Art. no. 174610.
[12] J. Liu et al., "Space-efficient TREC for enabling deep learning on micro-controllers," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 644–659.
[13] A. Kipf et al., "RadixSpline: A single-pass learned index," in *Proc. 3rd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, New York, NY, USA, 2020, pp. 1–5, doi: 10.1145/3401071.3401659.
[14] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," in *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 74–86, 2020.
[15] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 1001–1016, doi: 10.1145/3318464.3389752.
[16] W. Qu, X. Wang, J. Li, and X. Li, "Hybrid indexes by exploring traditional b-tree and linear regression," in *Proc. Web Inf. Syst. Appl.*, W. Ni, X. Wang, W. Song, and Y. Li, Eds., Berlin, Germany: Springer, 2019, pp. 601–613.
[17] S. You, J. Zhang, and L. Gruenwald, "Parallel spatial query processing on GPUs using r-trees," in *Proc. 2nd ACM SIGSPATIAL Int. Workshop Analytics Big Geospatial Data*, New York, NY, USA, 2013, pp. 23–31, doi: 10.1145/2534921.2534949.
[18] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, and J. D. Owens, "GPU LSM: A dynamic dictionary data structure for the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 430–440.
[19] Design Guide, "CUDA C Program. Guide," *NVIDIA*, p. 31, Jul. 2013.
[20] Part Guide, "Intel 64 IA-32 architectures software developer manuals," *Syst. Program. Guide*, vol. 3B, no. 2.11 pp. 1 64, 2011.
[21] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. ACM SIGGRAPH Talks*, New York, NY, USA, 2016, pp. 1–2.
[22] Y. Lu, X. Yu, L. Cao, and S. Madden, "Epoch-based commit and replication in distributed OLTP databases," in *Proc. VLDB Endowment*, vol. 14, no. 5, pp. 743–756, 2021.
[23] K. Zeng, S. Agarwal, and I. Stoica, "iOLAP: Managing uncertainty for efficient incremental OLAP," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1347–1361.
[24] A. Raza, P. Chrysogelos, A. C. Anadiotis, and A. Ailamaki, "Adaptive htap through elastic resource scheduling," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2043–2054.
[25] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso, "Batchdb: Efficient isolated execution of hybrid oltp+ olap workloads for interactive applications," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 37–50.
[26] R. A. Hughes, A. E. Miklos, and A. D. Ellington, "Gene synthesis: Methods and applications," in *Methods in Enzymology*, vol. 498. Amsterdam, Netherlands: Elsevier, 2011, pp. 277–309.
[27] J. Tian, K. Ma, and I. Saaem, "Advancing high-throughput gene synthesis technology," *Mol. Biosyst.*, vol. 5, no. 7, pp. 714–722, 2009.
[28] A. Nguyen, M. Edahiro, and S. Kato, "Gpu-accelerated voltdb: A case for indexed nested loop join," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2018, pp. 204–212.
[29] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," in *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020, doi: 10.14778/3389133.3389135.
[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.
[31] P. Li, Y. Hua, J. Jia, and P. Zuo, "FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems," in *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 321–334, Oct. 2021, doi: 10.14778/3489496.3489512.
[32] X. Li, J. Li, and X. Wang, "ASLM: Adaptive single layer model for learned index," in *Database Systems for Advanced Applications*, G. Li, J. Yang, J. Gama, J. Natwichai, and Y. Tong, Eds., Berlin, Germany: Springer, 2019, pp. 80–95.
[33] J. Ding et al., "ALEX: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 969–984, doi: 10.1145/3318464.3389711.
[34] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2019, pp. 1189–1206, doi: 10.1145/3299869.3319860.
[35] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2019, pp. 1189–1206, doi: 10.1145/3299869.3319860.
[36] J. O'Rourke, "An on-line algorithm for fitting straight lines between data ranges," *Commun. ACM*, vol. 24, no. 9, pp. 574–578, Sep. 1981.
[37] E. Furst, M. Oskin, and B. Howe, "Profiling a GPU database implementation: A holistic view of GPU resource utilization on TPC-H queries," in *Proc. 13th Int. Workshop Data Manage. New Hardware*, 2017.
[38] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[39] A. Redd, K. Khin, and A. Marini, "Fast ES-RNN: A GPU implementation of the ES-RNN algorithm," 2019, *arXiv:1911.13014*.
[40] B. G. Soos, A. Rak, J. Veres, and G. Cserey, "GPU powered CNN simulator (SIMCNN) with graphical flow based programmability," in *Proc. Int. Workshop Cellular Neural Netw. Their Appl.*, 2008, pp. 163–168.
[41] J. B. Kulkarni, A. Sawant, and V. S. Inamdar, "Database processing by linear regression on GPU using CUDA," in *Proc. Int. Conf. Signal Process. Commun. Comput. Netw. Technol.*, 2011, pp. 20–23.
[42] H. Xu, L. Zeng, X. Cai, and S. Li, "GPU-accelerated feature extraction and multi-resolution visualization for complex 3D fluid field," *J. Comput.-Aided Des. Comput. Graph.*, vol. 21, no. 7, pp. 893–899, 2009.
[43] B. He et al., "Relational joins on graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2008, p. 511–524, doi: 10.1145/1376616.1376670.
[44] M. Najam, R. Rasool, H. Ahmad, U. Ashraf, and A. Malik, "Pattern matching for dna sequencing data using multiple bloom filters," *Biomed. Res. Int.*, vol. 2019, pp. 1–9, 2019.
[45] Q. Fan and Q. Wang, "Performance comparison of web servers with different architectures: A case study using high concurrency workload," in *Proc. IEEE 3rd Workshop Hot Topics Web Syst. Technol.*, 2015, pp. 37–42.
[46] "Black friday figures," 2022. [Online]. Available: https://black-friday.global/
[47] "Feature store for ML," 2022. [Online]. Available: https://www.featurestore.org/
[48] L. Orr, A. Sanyal, X. Ling, K. Goel, and M. Leszczynski, "Managing ML pipelines: Feature stores and the coming wave of embedding ecosystems," in *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 3178–3181, 2021.
[49] "A billion taxi rides in redshift," 2021. [Online]. Available: https://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html
[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA, USA: MIT Press, 2001.
[51] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *Proc. 20th IEEE Int. Conf. Mobile Data Manage.*, 2019, pp. 569–574.
[52] A. Kipf et al., "SOSD: A benchmark for learned indexes," 2019, *arXiv:1911.13014*.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

[53] P. Van Sandt, Y. Chronis, and J. M. Patel, "Efficiently searching in-memory sorted arrays: Revenge of the interpolation search?," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2019, pp. 36–53, doi: 10.1145/3299869.3300075.
[54] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?," in *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1661–1673, Jul. 2018.
[55] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 1984, pp. 47–57.
[56] J. Kim, S.-G. Kim, and B. Nam, "Parallel multi-dimensional range query processing with r-trees on GPU," *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1195–1207, 2013.
[57] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," *ACM SIGMOD Rec.*, vol. 30, no. 2, pp. 139–150, 2001.
[58] A. Hadian, B. Ghaffari, T. Wang, and T. Heinis, "COAX: Correlation-aware indexing on multidimensional data with soft functional dependencies," 2020, *arXiv: 2006.16393*.
[59] J. Liu et al., "Exploring query processing on CPU-GPU integrated edge device," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4057–4070, Dec. 2022.
[60] M. Mitzenmacher, "A model for learned bloom filters, and optimizing by sandwiching," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, 2018, pp. 462–471.
[61] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing, "Updatable learned index with precise positions," in *Proc. VLDB Endowment*, vol. 14, no. 8, pp. 1276–1288, 2021.
[62] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, *BAO: Making Learned Query Optimization Practical*. New York, NY, USA: ACM, 2021, pp. 1275–1288.
[63] L. Cen, A. Kipf, R. Marcus, and T. Kraska, "LEA: A learned encoding advisor for column stores," in *Proc. 4th Workshop Exploiting AI Techn. Data Manage.*, New York, NY, USA, 2021, pp. 32–35.
[64] P. Li, Y. Hua, P. Zuo, and J. Jia, "A scalable learned index scheme in storage systems," 2019, *arXiv:1905.06256*.
[65] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 985–1000.
[66] R. Marcus, E. Zhang, and T. Kraska, "CDFShop: Exploring and optimizing learned index structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 2789–2792.
[67] G. Vinciguerra, P. Ferragina, and M. Miccinesi, "Superseding traditional indexes by orchestrating learning and geometry," 2019, *arXiv:1903.00507*.
[68] A. Hadian and T. Heinis, "Considerations for handling updates in learned index structures," in *Proc. 2nd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, New York, NY, USA, 2019, pp. 1–4.
[69] X. Zhong, Y. Zhang, Y. Chen, C. Li, and C. Xing, "Learned index on GPU," in *Proc. IEEE 38th Int. Conf. Data Eng. Workshops*, 2022, pp. 117–122.
[70] Z. Dai and A. Shrivastava, "Adaptive learned bloom filter (ADA-BF): Efficient utilization of the classifier," *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 11700–11710, 2020.
[71] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–20.
[72] W. Xiang, H. Zhang, R. Cui, X. Chu, K. Li, and W. Zhou, "PAVO: A RNN-based learned inverted index, supervised or unsupervised?," *IEEE Access*, vol. 7, pp. 293–303, 2019.
[73] K. Vaidya, E. Knorr, T. Kraska, and M. Mitzenmacher, "Partitioned learned bloom filter," 2020, *arXiv:2006.03176*.
[74] S. Macke, A. Beutel, T. Kraska, M. Sathiamoorthy, D. Z. Cheng, and E. Chi, "Lifting the curse of multidimensional data with learned existence indexes," in *Proc. Workshop ML Syst. NeurIPS*, 2018, Art. no. 6.
[75] M. Zhang, H. Wang, J. Li, and H. Gao, "Learned sketches for frequency estimation," *Inf. Sci.*, vol. 507, pp. 365–385, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025519307856
[76] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided B+ tree searches on a GPU with CUDA," in *Proc. 2nd Workshop on Appl. Multi Many Core Processors: Anal., Implementation, Performance*, 2011.
[77] M. Lopresti, N. Miranda, F. Piccoli, and N. Reyes, "Solving multiple queries through a permutation index in GPU," *Comput. Y Sistemas*, vol. 17, no. 3, pp. 341–356, 2013.
[78] H. Cheng, Z. Yong, and X. Yun, "BitMapper2: A GPU-accelerated all-mapper based on the sparse q-gram index," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 16, no. 3, pp. 886–897, May/Jun. 2019.
[79] B. Tran, B. Schaffner, J. M. Myre, J. Sawin, and D. Chiu, "Exploring means to enhance the efficiency of GPU bitmap index query processing," *Data Sci. Eng.*, vol. 6, no. 2, pp. 209–228, 2021.
[80] H. Lu, Y.-Y. Ng, and Z. Tian, "T-tree or b-tree: Main memory database index structure revisited," in *Proc. 11th Australasian Database Conf.*, 2000.
[81] J. Zhou, G. Qi, H. V. Jagadish, W. Luan, and Y. Zheng, "Generic inverted index on the GPU," National Univ. Singapore, Tech. Rep., 2015.
[82] J. Kim, S. Hong, and B. Nam, "A performance study of traversing spatial indexing structures in parallel on GPU," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun, IEEE 9th Int. Conf. Embedded Softw. Syst.*, 2012, pp. 855–860.
[83] M. Kim, L. Liu, and W. Choi, "A GPU-aware parallel index for processing high-dimensional big data," *IEEE Trans. Comput.*, vol. 67, no. 10, pp. 1388–1402, Oct. 2018.

Jiesong Liu is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), in 2020. His major research interests include database systems, and parallel and distributed systems.

Feng Zhang (Member, IEEE) received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is a professor with DEKE Lab and School of Information, Renmin University of China. His major research interests include database systems, and parallel and distributed systems.

Lv Lu is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. She joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) from 2019 to 2021. Her major research interests include high performance computing, artificial intelligence and machine learning systems.

Chang Qi is a graduate student, presently majoring in computer applications and technology, with the School of Information, Renmin University of China. She is currently conducting academic research in the Database & Intelligence Information Retrieval (DBIIR) Lab under the guidance of professor Zhang Feng. Her research interests include GPU acceleration, Big Data systems, high performance computing, and machine learning systems.

Xiaoguang Guo is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), in 2020. His major research interests include database systems and distributed systems.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.

Dong Deng received the bachelor's degree from beihang University, in 2011, and the PhD degree in computer science from Tsinghua University, in 2016. He is an assistant professor in the Computer Science Department with Rutgers University-New Brunswick. His research interests include data management, database system, data curation, and data-centric AI.

Guoliang Li received the PhD degree in computer science from Tsinghua University, Beijing, China, in 2009. He is currently working as a professor in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests mainly include data cleaning and integration, spatial databases, crowdsourcing, and AI & DB co-optimization.

Huanchen Zhang received the BS degree in computer engineering from the University of Wisconsin, Madison, in 2013, and the PhD degree in computer science from the Computer Science Department, Carnegie Mellon University, in 2020. He is an assistant professor in the Institute for Interdisciplinary Information Sciences at Tsinghua University. His research interests include database management systems, indexing/filtering data structures, data compression, and cloud databases.

Jidong Zhai (Senior Member, IEEE) received the BS degree in computer science from the University of Electronic Science and Technology of China, in 2003, and the PhD degree in computer science from Tsinghua University, in 2010. He is a professor in Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis and modeling of parallel applications.

Hechen Zhang is currently working toward the High School Affiliated to Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) as a research assistant since 2022. His research interests include parallel and distributed systems, and machine learning systems.

Yuxing Chen received the PhD degree in computer science from the University of Helsinki, Finland, in 2021. He currently works as a senior research engineer in the database R&D department with Tencent, China. His research interests focus on database performance and evaluation, HTAP database design, and distributed system design.

Anqun Pan is the technical director of the Database R&D Department with Tencent in China. With more than 15 years of experience, he has specialized in the research and development of distributed computing and storage systems. Currently, he is responsible for steering the research and development of the TDSQL distributed database system.

Xiaoyong Du (Member, IEEE) received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from Renmin University of China, Beijing, China, in 1988, and the PhD degree from Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 25,2026 at 11:32:46 UTC from IEEE Xplore. Restrictions apply.