Multi-Space Tree with Incremental Construction for GPU-Accelerated Range Queries

Brian Donnelly
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
brian.donnelly@nau.edu

Michael Gowanlock
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
michael.gowanlock@nau.edu

Abstract—**Performing range queries is prohibitively expensive as the dimensionality of the data increases. Indexing data structures reduce the time complexity of these searches by eliminating superfluous distance calculations. The state-of-the-art utilizes the GPU due to its high distance calculation throughput as compared to multi-core CPUs. Previous state-of-the-art indexes fall into two categories: metric- and coordinate-based indexes, both of which partition the space using different approaches. The indexes partition the space to generate a set of candidate points for a given query which are later refined by distance calculations. Popular metric-based indexes partition the data based on distances to reference points, where the placement of the reference points determines the partitioning of the data space but the effectiveness depends on the distribution of the data. In high-dimensions, coordinate-based indexes typically partition the data based on a subset of the coordinate dimensions. Regardless of the index type there is a tradeoff between index search overhead and the number of distance calculations, where increasing the number of partitions will increase the search overhead but will decrease the number of distance calculations computed. In this paper, we propose Multi-Space Tree with Incremental Construction (MISTIC), a blended approach which uses both metric-based and coordinate-based partitioning strategies coupled with incremental index construction. We evaluate MISTIC on 5 real-world datasets and compare performance to both a state-of-the-art metric-based index, COSS, and a state-of-the-art coordinate-based index, GDS-JOIN. We find that MISTIC outperforms the state-of-the-art methods with an average speedup of 2.53× over COSS and 2.73× over GDS-JOIN.**

I. INTRODUCTION

Advances in science and technology are producing quantities of data that have surpassed our analysis capabilities [1]. Range queries are an important tool that data scientists use to process large volumes of data, as they answer a fundamental question: Which objects in a dataset are similar to my query object(s)? However, range queries are computationally expensive [2], [3], and so reducing the cost of this operation is key for extracting information from large datasets.

Indexing Multi-dimensional Data Points: Indexes are data structures that store the dataset (D) and partition the data space. The index is then *searched* which produces a candidate set of points that may be within the search distance (ϵ), this candidate set is then *refined* using distance calculations. This is the *search-and-refine* strategy and is used for efficiently querying large datasets [4]–[6].

This material is based upon work supported by the National Science Foundation under Grant No. 2042155.

Increasing data dimensionality necessitates a corresponding increase in the search distance because the data space grows exponentially with dimensionality [7]. For a uniformly distributed dataset of a fixed size, an increase in the dimensionality will result in an exponentially larger separation between points, thus the search distance must be increased to find nearby points. This problem with high-dimensional spaces has been termed the *Curse of Dimensionality* [8], where an index may be completely ineffective at pruning searches and degrade into a brute force search (i.e., all of D will need to be examined for a given search).

Coordinate- vs. Metric-based Indexes: Coordinate-based indexes directly use the coordinates of each point in a dataset for partitioning (e.g., a canonical index is a kd-tree [9]). Using the kd-tree as an example, as the search distance increases, an increasing number of partitions in the tree will need to be examined, and in the extreme case, the entire kd-tree will need to be searched, thus degrading into a brute force search. To address this problem, a metric-based indexing strategy should be used instead [4], [5], [10].

A metric-based approach uses a contractive mapping function to embed the dataset into a lower dimensional space [10]. For a metric-based index, the contractive mapping uses distances from points in a dataset to a set of reference points. These distances are the new coordinates in the mapped space. Contractive mapping guarantees that the distances between points in the dataset only decrease so all points within the search radius are obtained and there is no accuracy loss.

Metric-based indexes maintain effectiveness as the distance threshold increases because each coordinate in the mapped space uses all of the coordinate information in the original data space. This creates more partitions which still allow for pruning the search in instances where coordinate-based methods degrade to brute force.

GPU-Acceleration and Distance Calculations: The total work computed is proportional to the search distance (ϵ). Also, increasing the data dimensionality will increase the cost of each individual distance calculation. Higher cost distance calculations incentivize more aggressive index partitioning tailored to each query. To this end, we propose an index with

ϵ-width partitions where the index construction cost is offset by a substantial increase in performance by decreasing the number of distance calculations. In terms of peak performance, GPU hardware has exceeded the capacity of multi-core CPUs. Range queries are an excellent algorithm for GPU acceleration for the following reasons: (i) the algorithm is throughput-oriented, as we are interested in computing a batch of range queries; (ii) each query point can be computed independently by one or more threads, although this leads to other issues regarding the Single Instruction Multiple Thread (SIMT) architecture; and, (iii) the GPU has superior distance calculation throughput compared to the CPU. For these reasons, with the exception of small workloads, the GPU outperforms multi-core CPU range query algorithms [11]–[13].

Drawbacks and Contributions: We outline several drawbacks of prior work in this area (**D1-3**):

D1 There is a vast quantity of work on the CPU outlining efficient indexes but many of those structures, particularly trees, do not perform well on the GPU.

D2 Due to the *curse of dimensionality* problem outlined above, some areas of research have instead focused their attention on approximate range queries [14], [15], which avoids many of the problems associated with searching high dimensional datasets. However, they do not return an exact result, which is often required in scientific and engineering domains.

D3 Previous indexes have used either a metric- or coordinate-based approach, leading to indexes tailored to dataset characteristics which reduce the overall robustness of the methods.

We address the drawbacks with contributions **C1-6**:

C1 We propose a novel multi-space index, the Multi-Space Tree with Incremental Construction (MiSTIC), which combines metric- and coordinate-based approaches which are more robust than using a single indexing type.

C2 The index uses incremental construction to increase the pruning efficiency of the index when coupled with a heuristic for determining the effectiveness of candidate partitions.

C3 We propose a new reference point placement strategy that exploits dataset characteristics, yielding good partitioning.

C4 The index exploits several facets of GPU architecture including good locality and caching behavior and uses instruction level parallelism (ILP) to hide accesses to global memory.

C5 We compare MiSTIC to one metric- and one coordinate-based GPU reference implementation on five real-world datasets. We show that MiSTIC is robust to different dataset characteristics and consistently outperforms the state-of-the-art methods COSS and GDS-Join with a speedup of 2.53× and 2.73×, respectively.

C6 Contrary to other work, we find that minimizing distance calculations does not necessarily lead to the best performance, rather load balancing may be more important.

This paper is organized as follows: Section II outlines the problem statement and related work. Section III presents MiSTIC and associated optimizations. Section IV presents the experimental evaluation. Lastly, Section V concludes the work.

II. Background

A. Problem Statement

We define a dataset, D, which contains $|D|$ points (or feature vectors), where each point has n dimensions. Each point is denoted as $x_i \in D$ where $i = 1, 2, \dots, |D|$. Each point, x_i, is defined by a set of coordinate values in n dimensions denoted as $\{x_i^1, x_i^2, \dots, x_i^n\}$. A range query search finds all of the $x_i \in D$ which are within a distance threshold, ϵ, of a query point.

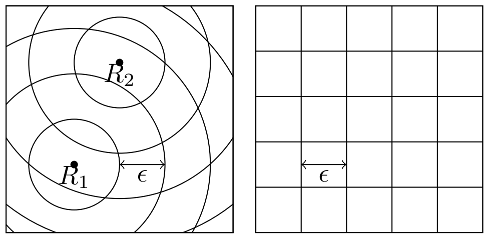
In a self-join operation all of the data points in D are compared to each other. The operation returns $\{q_1, q_2, \dots, q_i\}$, where q_i contains the points in D which are within ϵ of x_i. The total number of returned points, $|Q| = \sum_{i=1}^{|D|} |q_i|$, is most often greater than $|D|$ such that the memory required to store the results from a self-join query exceeds the memory capacity of a GPU, requiring batched computations where intermittent results are transferred back to the host. In the case that D also exceeds the memory capacity of a GPU, the batched computations will only transfer a partition of D which is required for that batch of computations. This enables the self-join to be computed on the GPU regardless of $|Q|$ and $|D|$ and the global memory capacity of a particular GPU model.

We define the Euclidean distance between two points, a and b as $dist(a, b) = \sqrt{\sum_{j=1}^{n}(a^j - b^j)^2}$ and we add a result to the result set when $dist(a, b) \leq \epsilon$. We use the Euclidean distance because it is the standard distance metric that is employed in the literature [11]–[13], [16].

B. Index Supported Range Queries

A distance similarity self-join is straightforward to implement using a nested loop, and when including the dimensionality of the data, n, it has a time complexity of $O(n \cdot |D|^2)$. For even moderately sized datasets, the quadratic time complexity becomes an intractable problem, particularly in high dimensions [8]. To improve performance, indexing methods have been developed to reduce the number of distance calculations needed at the cost of preprocessing overhead [17]. These methods prune the search space so that points only calculate the distance to a subset of other points that are nearby in the data space. This allows range queries to be performed on larger datasets by reducing the total computational cost.

Most indexes use either a grid or a tree structure to store the partitions of the index [8], [12], [18]. Trees create a hierarchical structure where nodes in the tree are decomposed into smaller subsequent nodes. In contrast, a grid typically partitions data using axis-aligned regions [6], [19], [20]. Both trees and grids can be data-agnostic, where the index is constructed statically without using information about the data points (i.e., only using the bounding volume), or they can be data-aware and use the data points during index construction, as exemplified by a kd-tree [9].



(a) Metric-Based (b) Coordinate-Based

Fig. 1. (a) An example of a metric-based index (similar to COSS [12]) partitioning a two-dimensional space with two reference points R_1, and R_2. (b) An example of partitioning with a grid (similar to GDS-JOIN [19]) in two dimensions where each of the dimensions are used for indexing. Both methods use the triangle inequality to exclude points in non-adjacent ϵ-width partitions from the search.

As shown in Figure 1(a), metric-based indexes (also referred to as distance-space, pivot-point, or coordinate-oblivious indexes), use a set of points in the same space as the data as references for partitioning the dataset [5], [18], [21]. These methods tend to have better performance than coordinate-based indexes in high-dimensional spaces because they incorporate every coordinate into a single mapped dimension. Metric-based indexes create indexing structures on a transformed space [8]. Pruning the search occurs in this transformed space but the distance calculations occur in the original data space. While metric-based indexes have proven to be effective in high-dimensional spaces, their additional complexity leads to higher overheads that make them less attractive for low dimensional searches [12].

The time complexity of range queries using an index (such as an R*-tree, kd-tree, or MiSTIC) is an open problem. The number of distance calculations for the self-join operation has an upper bound of $O(|D|^2)$ and a lower bound of $O(|D|)$. Because the time complexity is a function of the search radius, ϵ, and is unconstrained, it is possible that all points will need to be compared to each other. There has been some discussion in the literature regarding practical values of the search radius and the lack of a more robust complexity analysis [22], [23].

C. State-of-the-art & Reference Implementations

We outline the range query algorithms from the literature that we will compare with MISTIC in Section IV. As described in Section I, recent advances in GPU hardware have surpassed the capabilities of multi-core CPUs and so range queries are best carried out on the GPU. Consequently, we do not compare to any multi-core CPU algorithms, as they are typically only advantageous on small workloads for which GPU acceleration is unwarranted as illustrated by previous work [13].

We compare our work to COSS [12] and GDS-JOIN [24] which are GPU-accelerated range query algorithms that use metric- and coordinate-based indexes, respectively. We summarize the algorithms as follows and note that the authors have made their code publicly available such that we can make a comparison to their work[1]. We also compare to a highly optimized brute force implementation, BRUTE, which is a baseline for comparison.

GDS-JOIN performs range queries using the GPU [11], [19], [24]. It constructs a compact coordinate-based grid index on the data as shown in Figure 1(b). The algorithm includes several optimizations, such as reordering the sequence in which query points are processed to limit load imbalance within warps[2]. It indexes the data in $r < n$ dimensions to reach a trade-off between index search overhead and the number of points that need to be refined using distance calculations. The distance calculation kernel takes advantage of the GPU's instruction level parallelism to hide global memory access latency [25]. We compare MISTIC to GDS-JOIN as it is a state-of-the-art coordinate-based index and is therefore suitable for different dataset properties than metric-based indexes.

COSS or Coordinate-Oblivious Index for Similarity Searches, is a metric-based index, designed for GPU acceleration and partitions the space as shown in Figure 1(a). The index design was motivated by the drawbacks of coordinate-based grid indexes to be better for high-dimensional range queries [12].

BRUTE is a brute-force implementation we created for comparison which is highly optimized to use coalesced memory access patterns, good locality to increase cache hits, and reorders the dimensions of the data based on variance to increase the efficiency of short circuiting the distance calculations (which has a substantial impact on brute-force algorithms). BRUTE lacks index construction overhead allowing it to outperform indexing methods on small datasets.

D. Limitations of the State-of-the-Art

We show in Section III that MISTIC addresses several limitations of the state-of-the-art methods, which include:

- COSS and GDS-JOIN search the index to find non-empty partitions on the GPU using binary searches instead of using tree traversals. A drawback of this is that a single binary search has to perform $\log_2(|G|)$ accesses to global memory, where $|G|$ is the number of non-empty partitions in the index (empty space is not indexed to limit global memory usage). In contrast, a tree traversal aborts early when children do not exist, which typically results in fewer accesses to global memory compared to binary searches.
- The reference point placement in COSS and the dimension selection in GDS-JOIN determines how the partitions are generated; however, the strategies are static. MISTIC addresses this limitation by using incremental index construction to examine several sets of candidate partitions that when combined together produce an efficient index structure that prunes the search better than a static method.
- Metric- and coordinate-based indexes have fundamentally different approaches with performance dependent on dif-

[1]https://github.com/bwd29/Coordinate-Oblivious-Similarity-Search/ and https://github.com/mgowanlock/gpu_self_join/.

[2]Warps are groups of 32 threads on the GPU that execute the same instruction in lockstep. Throughout this paper, we use CUDA terminology, but the concepts are the same across GPUs from different vendors.

ferent dataset characteristics [24]. This necessitates the selection of an indexing method based on dataset characteristics, like intrinsic dimensionality, which are non-trivial to discover. MISTIC merges the two approaches to yield an algorithm which dynamically adapts to the dataset without foreknowledge of dataset characteristics. This leads to performance gains over both metric- and coordinate-based indexing methods regardless of the dataset.

III. MISTIC

As described in Section II-D there are three goals for MISTIC, firstly to replace a potently expensive binary search with a tree traversal, secondly to allow for incremental construction, and finally to combine both metric- and coordinate-based partitioning strategies.

Our tree is constructed using a combination of the intersections of ϵ-width shells centered on each reference point (the metric-based method) and a grid of ϵ-width axis-aligned cells (the coordinate-based method) which create partitions. Each query requires a search over the index to identify nearby partitions containing data points. There are two methods for searching the partitions; (i) a binary search or (ii) a depth first tree traversals.

A binary search (which is used in the reference implementations) can be used on the last layer of the tree and requires $\log_2(|G|)$ memory accesses [12], where $|G|$ is the number of non-empty index locations (also the size of the last layer of the tree). The worst case for a binary search is if the searched-for partition is empty and therefore not in the array of non-empty partitions, as this requires a full search of the array.

A depth first tree traversal will have a maximum of r memory accesses into the data structure, where r is the number of layers of the tree representing partitions which are either metric- or coordinate-based. The best case for a tree traversal is if the searched-for partition is empty because the search terminates when a branch of the tree has no partitions. This is the opposite of the binary search and results in the tree traversals performing fewer memory access when there are more empty partitions than non-empty. Additionally, tree traversals require fewer memory accesses as compared to a binary search when $\log_2(|G|) > r$ which occurs when the index partitions the data effectively (i.e., the data is separated into numerous partitions which yields good pruning). The number of non-empty partitions of ϵ-width is dependent on the ϵ used in the search, so large ϵ values result in fewer partitions and may therefore be more efficient using binary searches.

In Figure 1, an example of how MISTIC partitions a two-dimensional coordinate space is shown. The non-empty partitions that are created by the overlapping shells from the reference points or the non-empty cells in the grid become leaves on the tree. In high dimensional space the number of partitions will increase to such a level that it becomes intractable to store each partition, therefore only the non-empty partitions are used in the last layer of the tree (as shown by L_r in Figure 2). The non-terminal layers of the tree keep track of the empty shells generated by that layer's reference point, but those nodes on the tree do not have any children nodes, which allows for depth first searches to terminate early.

As ϵ increases, the width of the shells and cells also increase resulting in fewer and larger partitions. While a high dimensional space will have more partitions for a given ϵ, in order to have a practical query the ϵ value will have to increase as the dimensionality increases. This leads to a small subset of partitions which contain the majority of the dataset. This is offset by effective partitioning of the data using a combination of metric- and coordinate-based partitioning strategies.

A. Tree Structure

We outline the tree structure of MISTIC which contains both metric-based partitions created with reference points and coordinate-based partitions created by partitioning dimensions in the coordinate space (see Figure 1). The tree construction is performed on the CPU while the final data structure is transferred to the GPU for searching and distance calculations.

First Layer – Figure 2 shows the structure of an example tree with r reference points/indexed dimensions. The tree has one layer, L, for every reference point or indexed dimension. The first layer of the tree, L_1, is constructed by first calculating the maximum distance from the first reference point to all of the points or the maximum coordinate value of all of the points in the data. This maximum value is used to find the total number of possible partitions for that layer, b_1, by dividing the maximum value by the distance threshold ϵ used in the search. An array of size b_1 is then allocated. The layer L_1 is then populated with an incrementing counter and zeroes to represent non-empty and empty partitions, respectively. A partition is non-empty if the distance from a point to the reference point associated with L_1 falls within the range of a partition or if the coordinate values of a point for an indexed dimension falls within that range. For the first layer, the index of a partition in the array is the distance from the reference point or the coordinate value divided by ϵ. This is not true for all subsequent layers. b'_1 is the number of non-empty partitions in L_1, and is the sum of the array L_1 and $b'_1 \leq b_1$ since there cannot be more non-empty partitions than the total number of potential partitions.

The values in layers L_1 through L_{r-1} represent the current non-empty partition count such that the value in L_{x-1} at a non-empty index will point into the next layer L_x, where x is the layer number. The value does not directly correspond to the index in the next layer but is rather a multiple of the range of partitions for the reference point/indexed dimension associated with the next layer.

Middle Layers – Subsequent layers after L_1 are generated using the previous layer's values for b_x and b'_x. L_x allocates $b'_{x-1} \cdot b_x$ partitions. The idea is to have a section of L_x which has b_x partitions for each non-empty partition in the previous layer L_{x-1}. To populate L_x, each point is compared to the reference point associated with L_x, and this will yield the partition number that will be non-empty based on the previous layer's partition for that point. A point in partition y in the

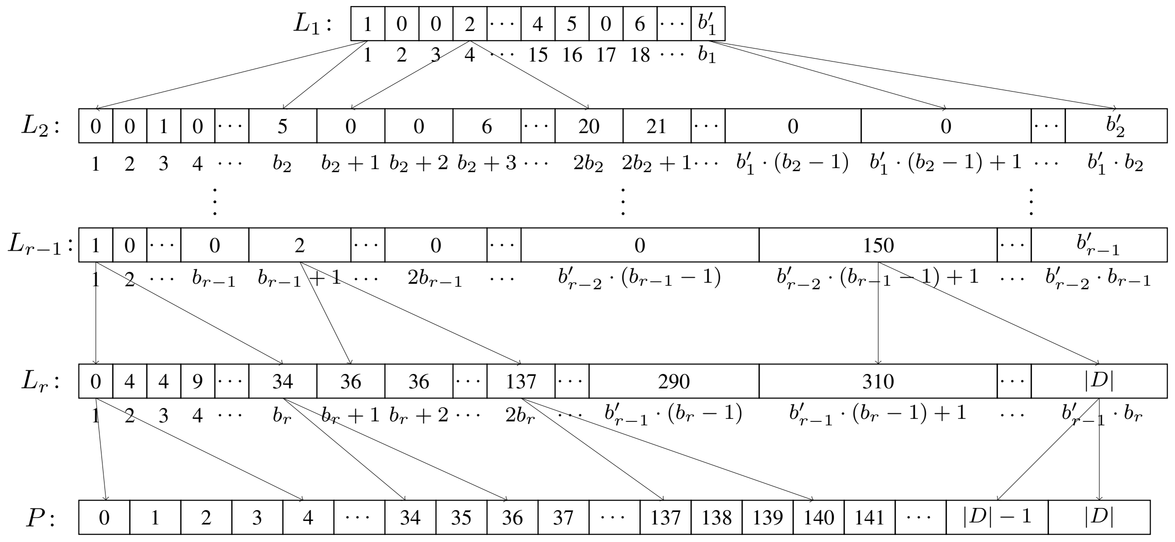

L_1:
1 0 0 2 ··· 4 5 0 6 ··· b'_1
1 2 3 4 ··· 15 16 17 18 ··· b_1
L_2:
0 0 1 0 ··· 5 0 0 6 ··· 20 21 ··· 0 0 ··· b'_2
1 2 3 4 ··· b_2 b_2+1 b_2+2 b_2+3 ··· $2b_2$ $2b_2+1$ ··· $b'_1 \cdot (b_2-1)$ $b'_1 \cdot (b_2-1)+1$ ··· $b'_1 \cdot b_2$
L_{r-1}:
1 0 ··· 0 2 ··· 0 ··· 0 150 ··· b'_{r-1}
1 2 ··· b_{r-1} $b_{r-1}+1$ ··· $2b_{r-1}$ ··· $b'_{r-2} \cdot (b_{r-1}-1)$ $b'_{r-2} \cdot (b_{r-1}-1)+1$ ··· $b'_{r-2} \cdot b_{r-1}$
L_r:
0 4 4 9 ··· 34 36 36 ··· 137 ··· 290 310 ··· $|D|$
1 2 3 4 ··· b_r b_r+1 b_r+2 ··· $2b_r$ ··· $b'_{r-1} \cdot (b_r-1)$ $b'_{r-1} \cdot (b_r-1)+1$ ··· $b'_{r-1} \cdot b_r$
P:
0 1 2 3 4 ··· 34 35 36 37 ··· 137 138 139 140 141 ··· $|D|-1$ $|D|$

Fig. 2. Tree indexing example, where L is the tree structure with r layers/reference points. b_x is the maximum partition range in that layer, b'_x is the number of non-empty partitions where x is the layer number. P is the point array and $|D|$ is the number of points in the dataset.

previous layer L_{x-1} and in partition z of layer L_x will be in the index $y \cdot b_x + z$. This index will be switched to the current non-empty partition count from zero if no other point was found in that partition previously, otherwise the partition will have already been set, and no update is required.

Final Layer – The last layer of the tree, L_r, is unique as it contains the counts of how many points are in each partition of the final layer. The layer is constructed similarly to the previous layer but instead of only noting if the partition is non-empty, a count is incremented in that partition every time a point is located there, and all subsequent partitions are also incremented. This creates a mapping from L_r to P where each value in L_r points to a starting location in P which contains the first point in that non-empty partition. The points in P which belong to that partition are found by accessing the next starting location in L_r which denotes where the next partition begins. The final partition in layer L_r will have a maximum range $|D|$ which corresponds to the size of the dataset D, and will be the maximum index of array P.

1) Partitioning Strategy: The placement of the reference points (metric-based layers) and the selected dimensions for indexing (coordinate-based layers) has a significant impact on the performance of the algorithm. However, finding the optimal reference point placement or selection of dimensions to index that minimize the total number of distance calculations, is intractable. Many attempts have been made to try to model reference point placement strategies [5], [26]–[28] and determine which dimensions to index [24] but because of the complexity of the problem and the dependency on the data distribution there is no single best solution. MiSTIC chooses metric- or coordinate-partitions using a partitioning strategy and evaluates them using a heuristic. Selecting reference points and indexed dimensions to consider for tree construction is carried out with three different partitioning strategies (PS):

PS_1: We give the intuition of an approach that examines the variance of points within partitions. Consider that the greater the variance, the better pruning a reference point should have in general because the points will be separated into more partitions which reduces the total number of distance calculations. To this end, we select initial reference points from a random distribution of values to create a reference point set, R', containing q reference points $R_1, R_2, \ldots, R_q$. The initial set of randomly generated reference points R' are then evaluated using a sample of the data set to find the variance in the distances between R' and a subset of data points, C or $\delta = S^2 = \frac{\sum_{a=1}^{a=|C|}(dist(R'_a, C_a) - \overline{X})^2}{|C|-1}$, where $\overline{X}$ is the mean of distances between R'_a and all points in C.

PS_2: The reference points are generated using the strategy outlined in previous work [12] which places the reference points around the outside of the data. This is effective at reducing partitions that are adjacent in the index but are spatially distant in the original coordinate-space.

PS_3: We select the dimension to index for the coordinate-based partitions based on the variance of each dimension. Previous work has shown that the dimensions with the highest variance are the best dimensions to construct an index [11], [19]. We only evaluate the $r = 6$ highest variance dimensions for each layer to reduce the amount of work needed for index construction. This corresponds to the dimensions used for indexing in the reference implementation, GDS-Join.

Regardless of the strategy used, in the best case the points will be evenly spread throughout the partitions associated with R'_x and in the worst case they will be distributed into only a few partitions. The variance of R'_x with respect to C is a good indication of how clustered the points will be into the partitions, with higher variance correlated with more evenly distributed points.

Algorithm 1 Incremental tree construction.

```
1: procedure TREECONSTRUCTION($D$, $\epsilon$, $R'$, $r$)
2:     for $i \in \{1, 2, \ldots, r-1, r\}$ do
3:         for $j \in \{1, 2, \ldots, |R'|-1, |R'|\}$ do
4:             $M \leftarrow ParValue(R'_j, D)$
5:             $b_i \leftarrow \lceil (max(M)/\epsilon) \rceil \cdot b'_{i-1}$
6:             for $k \in \{1, 2, \ldots, |D|-1, |D|\}$ do
7:                 $ParNum[j,k] \leftarrow \lfloor (M[j,k]/\epsilon) \rfloor$
8:                 $Ofs \leftarrow (Par[i, ParOfs[i-1,k]] - 1) \cdot b_i$
9:                 $ParOfs[i,k] \leftarrow Ofs + ParNumbers[j,k]$
10:                if $ParCnts[i, ParOfs[j,k]] = 0$ then
11:                    $b'_i \leftarrow b'_i + 1$
12:                $ParCnts[i, ParOfs] \leftarrow ParCnts[i, ParOfs] + 1$
13:            $Heuristic[j] \leftarrow CalcHeuristic(ParCnts)$
14:        $R_i \leftarrow R'[minimum(Heuristic)]$
15:        $Count \leftarrow 0$
16:        for $j \in \{1, 2, \ldots, b_i - 1, b_i\}$ do
17:            if $ParCnts[i,j] \neq 0$ then
18:                $Count \leftarrow Count + 1$
19:                $L_i[j] \leftarrow Count$
20:            else
21:                $L_i[j] \leftarrow 0$
22:    $Sum \leftarrow 0$
23:    for $i \in \{1, 2, \ldots, b_r - 1, b_r\}$ do
24:        $Sum \leftarrow Sum + L_r[i]$
25:        $L_r[i] \leftarrow Sum$
26:    Return $L$
```

B. Incremental Tree Construction

To construct the tree index incrementally as described by Algorithm 1, the procedure takes as input the dataset (D), the distance threshold (ϵ), an array of reference points or dimensions to index R' (generated as described in Section III-A1) and the number of layers of the tree (r), as shown on line 1. The algorithm constructs each layer of the tree on the CPU starting with layer L_1 and proceeding to layer L_r in a loop starting on line 2. For each layer of the tree, each potential reference point or indexed dimension in R' needs to generate a layer L_i (line 3). To generate a layer for a given *reference point* in R' first the distance to all the points in D is used to generate a distance vector M with the function $ParValue()$ on line 4. To generate a layer for an *indexed dimension* in R', $ParValue()$ will return the coordinate value for each point which matches the dimension being indexed. The total number of partitions for that layer b_i for a given reference point R'_j are calculated by finding the maximum value in M, dividing it by ϵ, and multiplying it by the previous layer's non-empty bins, b'_{i-1} (line 5).

Each potential layer needs to iterate through every point in D to find: (i) the partition number, $ParNum$, from the floor of the distance from D_k to R'_j divided by ϵ (line 7); (ii) the offsets, Ofs (line 8), of the point from the beginning of L_i which is found from the point's previous partition in L_{i-1} multiplied by the b_i value calculated on line 5; (iii) the offset into the layer for each partition, $ParOfs$, is the previously calculated Ofs added to the $ParNum$ on line 9; (iv) the partition counters, $ParCnts$, which tracks how many points are in each partition for a given layer L_i. If a partition goes from empty to non-empty (line 10), b'_i is incremented to track the non-empty partitions for L_i (line 11).

Algorithm 2 Searching the tree.

```
1: procedure TREESEARCH($A$, $B$, $r$, $I$)
2:     if $I[A[0]] = 0$ then
3:         return $False$
4:     $B_{Total} \leftarrow B[0]$
5:     $Loc \leftarrow A[1] + B[0] + (I[A[0]] - 1)$
6:     for $i \in \{1, 2, \ldots, r-1\}$ do
7:         if $I[Loc] = 0$ then
8:             return $False$
9:         $B_{Total} \leftarrow B_{Total} + B[i]$
10:        $Loc \leftarrow B_{Total} + (I[Loc] - 1) + A[i+1]$
11:    if $I[Loc] = I[Loc-1]$ then
12:        return $False$
13:    return $(I[Loc-1], I[Loc])$
```

After each point has been assigned a partition, a heuristic for that potential layer is calculated using $ParCnts$ on line 13. The best layer for L_i is selected based on which potential reference points or indexed dimension in R' generated the lowest heuristic value (line 14). Once R_i has been selected, layer L_i is constructed as outlined in Section III-A (lines 15 – 21). The final layer, L_r is a special case and needs to have a running total to track the number of points in each partition which are found by keeping a running total of the values in the partitions of L_i when $i = r$. This replaces the values in the array with the running total as shown on lines 22 – 25.

Parallel Incremental Construction – Constructing the tree incrementally (see Algorithm 1) requires that each layer be constructed $|R'|$ times. While this increases the amount of work needed to construct the tree, each layer of the tree is dependent on the previous layer and the amount of parallelization is limited when statically constructing the tree. When constructing the tree incrementally, each layer must be evaluated for each potential reference point or indexed dimension. This allows for parallelization such that a thread is assigned to generate each potential layer on line 3. The overhead from evaluating potential layers of the tree is mostly hidden with concurrent construction, resulting in a negligible increase in the overall construction time.

Heuristic for Incremental Construction – When constructing the tree incrementally there needs to be a heuristic to determine which potential layer will lead to the best performance. Without a good heuristic, the tree may select a given layer which may be effective on an individual basis, but not when considering overlapping partitions with the other layers. This would increase both the construction overhead and search time while failing to offset these costs with increased distance calculation pruning. We define STDDEV as the standard deviation which is calculated as follows: $\sqrt{\sum_{i=0}^{b_r} |L_r[i] - \overline{X}|^2 / |D|}$, where $\overline{X}$ is the average number of points in each partition of the layer, L_r. We select the layer that has the lowest standard deviation to minimize the difference in the number of points between partitions. We examined several heuristics but found that STDDEV outperformed them, so we omit describing them.

C. Searching the Tree

While tree construction occurs on the CPU, searching the tree occurs on the GPU. The search requires four inputs (line 1 in Algorithm 2). A, B, r, and I refer to an array indicating a partition to find, an array of the size of each layer of the tree, the number of layers, and an array that stores all of the tree layers in adjacent memory as $L_1, L_2, \ldots, L_{r-1}, L_r$, respectively. A, has r values each representing an index in I which corresponds to a partition adjacent to the partition of the point that is initiating the search. For each layer of the tree as depicted in Figure 2, the search finds if the value of I corresponding to partition indicated by A that matches the layer to be searched is zero, which indicates an empty partition and terminates the search.

In Algorithm 2, the search starts at the first layer on line 2 which evaluates if the adjacent partition is empty and terminates the search if true. If not, then the location of the adjacent partition for the next layer is calculated on line 5. The algorithm also starts a counter for the total partition sizes B_{Total} on line 4 which will be used for determining the offset into array I.

On line 6 the algorithm enters into a loop to evaluate if the adjacent partition continues to be non-empty (line 7) then calculates the offset (line 9) and location (line 10) for the next layer of the tree. If the adjacent partition is found to be empty for any layer (including the last on line 11 which is an exception because L_r stores the location of points in P, so an empty partition is indicated by no change in the value compared to the previous index) then the search terminates, otherwise it returns the lower and upper bound of points within the adjacent partition on line 13. These bounds correspond to points in the array P (as shown in Figure 2) and define the candidates for the query point for that particular adjacent partition indicated by A.

Binary Searches – Instead of using tree traversals, MISTIC can be configured to use a binary search to locate non-empty partitions in the final layer of the tree. Since the last layer of the tree is sorted we represent it with a linear (one dimensional) ID which a binary search uses to find a specific non-empty partition. As stated in Section II-D, binary searches have drawbacks compared to tree traversals, especially while searching a large number of non-empty partitions. We investigate MISTIC with both search methods and evaluate their impact on performance.

D. Other Optimizations

While the main contribution of this paper is our efficient index, MISTIC, we outline several other optimizations that are needed to ensure that we do not overflow the result set buffer on the GPU, and perform efficient distance calculations.

1) Batching the Computations: The distance calculation kernels need to be batched because of GPU memory limitations. Most result sets, Q, from a self-join operation on a large dataset will need more memory than is available on the GPU to store $|Q|$ pairs of points. The number of batches needed depends on the number of points in the dataset and ϵ, with a larger number of points (or ϵ) requiring a larger number of batches to compute. The limiting factor for how many points are computed in each batch is the global memory needed to store the result set for each batch. Between each batch/kernel invocation, the result set (Q) is sorted and transferred to the host, freeing space for the next batch. The global memory on the GPU is allocated once and data is transferred to the host using a pre-pinned memory buffer to reduce the overhead due to data transfers over PCIe.

2) Instruction Level Parallelism and Short Circuiting: The kernel utilizes instruction level parallelism (ILP) for higher computation throughput by hiding accesses to global memory. This is done by unrolling iterations of the loop which computes the distances between points. The number of iterations unrolled limits the amount of ILP that is possible. We experimentally determined that unrolling by four loop iterations had the best performance with MISTIC on the platform and so the parameter is fixed to four in the experimental evaluation.

In addition to using ILP for computing distance calculations, we also allow for the distance calculations to short circuit and abort early. We use a variable to keep a running total of the distance accumulated, and with ILP this total gets updated every four dimensions. At each update we terminate the distance computation if it has exceeded ϵ. This reduces the amount of work needed to compute distance calculations between points that are far away from each other.

IV. Experimental Evaluation

A. Evaluation Platform

Experiments are executed on a platform with $2\times$ AMD EPYC 7542 CPUs (64 total physical cores) clocked at 2.9 GHz with 512 GiB of main memory and a 40 GiB NVIDIA A100 GPU. The code uses CUDA v.11 and is compiled with the O3 optimization flag. Multi-threaded CPU code is parallelized using the OpenMP library.

B. Implementation Configurations

In the results, we report the response time and variant metrics, such as speedup. The total response time (or end-to-end time) excludes loading the dataset into main memory as this is the same for all experiments, and includes all data transfers to/from the GPU, and host-side overheads such as storing and organizing the final result set in main memory. Each method uses batching on the GPU to allow for result sets that exceed the GPU's memory capacity, which is a necessity for moderately large datasets. All indexing methods are configured to use $r = 6$ reference points/indexed dimensions. While this will not always be the optimal configuration across all datasets, it allows for a fair comparison across all methods. Similarly, prior work showed that indexing in $r = 6$ dimensions (for coordinate-based indexes) and using $r = 6$ reference points (for metric-based indexes) was found to yield good performance on a range of workloads [12], [19], [24].

MISTIC: We use a kernel block size of 256 with 1024 blocks per kernel for 262,144 threads per kernel invocation. MISTIC uses $r = 6$ layers, with 38 potential layers (16

TABLE I

THE FIVE REAL-WORLD DATASETS (NORMALIZED TO THE RANGE $[0, 1]$) THAT WE USE IN OUR EVALUATION WHERE THE DATA DIMENSIONALITY n, INTRINSIC DIMENSIONALITY i (ROUNDED) AND DATASET SIZE $|D|$ ARE SHOWN ALONG WITH THE VALUES OF ϵ THAT CORRESPOND TO THE THREE TARGET SELECTIVITY VALUES.

Dataset	n	i	$\|D\|$	S_s	S_m	S_l
Wave [29]	49	15	287,999	0.0054	0.00702	0.008358
MSD [30]	90	29	515,345	0.0076	0.00913	0.011334
Bigcross [31]	57	3	11,620,300	0.0131	0.01994	0.0281
SuSy [32]	18	9	5,000,000	0.01703	0.02078	0.025555
Higgs [32]	28	19	11,000,000	0.049186	0.05558	0.063117

reference points using PS_1, 16 reference points using PS_2 and 6 indexed dimensions using PS_3) per layer as discussed in Section III-A1. The code repository used for evaluation is available to the public [3].

COSS: We experimentally found that a kernel block size of 1024 with 128 blocks per kernel launch (131,072 threads total) with 2 concurrent GPU streams has the best performance.

GDS-JOIN: A kernel block size of 32 with a dynamic number of blocks per launch with 3 concurrent GPU streams was found to have the best performance on our platform.

BRUTE: We configured BRUTE to run with a kernel block size of 256 with 512 blocks per kernel launch for a total of 131,072 threads per launch, which was found to have the best performance on our platform.

C. Selection of the Search Distance ϵ

We will show how performance scales as a function of the search distance (ϵ). To compare the performance between datasets having different sizes and dimensions ($|D|$ and n), the typical convention is to use search distances that span the same range of selectivity values across all datasets. The selectivity, S, refers to the mean number of points found by all searches carried out on a dataset. For the self-join this refers to $|D|$ searches, and so the selectivity $S = (|Q| - |D|)/|D|$, where $|Q|$ is the total number of results returned. In this paper, we select three target (small, medium, and large) selectivity values, corresponding to $S_s = 2^8$, $S_m = 2^{10}$, and $S_l = 2^{12}$, respectively. These selectivity values span several orders of magnitude and so they allow us to examine how the algorithms scale with increasing search distance and thus workload.

Note that there is not an analytical method to calculate what the search distance, ϵ, should be to yield the abovementioned selectivity values because the real-world datasets do not follow known data distributions (e.g., uniform, exponential, or normal) and so we perform a search on all datasets to first determine the ϵ values that correspond to each of the selectivity values above. All of the ϵ values yield a selectivity that is within 1% of the target selectivity values (S_s, S_m, and S_l) and are given in Table I.

D. MISTIC *Performance Analysis*

MISTIC: Tree Construction – We examine the fraction of time spent on tree construction for each selectivity level

[3]Code available at https://github.com/bwd29/self-join-MiSTIC

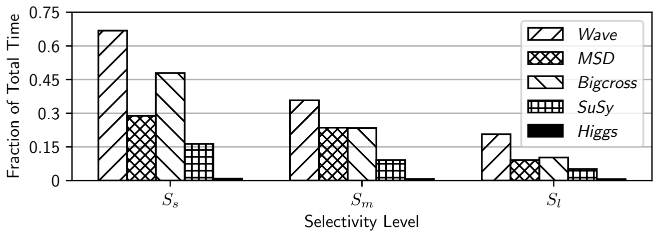


Fig. 3. The fraction of the total response time spent constructing the tree for each of the selectivity values S_s, S_m, and S_l across the five real-world datasets for $r = 6$.

TABLE II

THE PARTITIONING STRATEGY (1-3) DYNAMICALLY SELECTED FOR EACH LAYER OF MISTIC AS DESCRIBED IN SECTION III-A1 FOR THE FIVE REAL-WORLD DATASETS AND EACH SELECTIVITY LEVEL.

Dataset	S_s						S_m						S_l					
	L_1	L_2	L_3	L_4	L_5	L_6	L_1	L_2	L_3	L_4	L_5	L_6	L_1	L_2	L_3	L_4	L_5	L_6
Wave	2	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1
MSD	2	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1
Bigcross	3	2	1	1	1	1	3	2	1	1	1	1	3	2	1	1	1	1
SuSy	3	1	1	1	1	1	3	1	1	1	1	1	3	2	1	1	1	1
Higgs	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

in Figure 3. Tree construction overhead has a large impact on performance on the smaller datasets and lower selectivity levels which are correlated with lower total response times. As the number of distance calculations increase with higher selectivity levels and larger datasets, the tree construction time fraction becomes negligible. Tree construction accounts for up to 65% of the total time; however, we will show that MISTIC maintains a competitive performance level despite the tree construction overhead. Reference implementations COSS and GDS-JOIN have negligible index construction overhead.

The partitioning strategies (PS) used for each layer of the tree are given in Table II. Only L_1 and L_2 did not always select our novel partitioning strategy, PS_1. L_1 uses the reference point placement strategy also used by COSS (PS_2) for *Wave* and *MSD* while MISTIC uses PS_3 (the PS used by GDS-JOIN) on *Bigcross* and *SuSy* for L_1. Additionally, there are 4 cases where PS_2 is used for L_2 and *Higgs* always uses PS_1. The partitioning of the first few layers has a larger impact on overall performance than subsequent layers because of how the heuristics select subsequent layers. Even though PS_2 and PS_3 are rarely selected, they substantially improve MISTIC's performance. Forcing MISTIC to use a single PS reduces performance regardless of which PS is used.

Heuristic Comparison for Construction – MISTIC is evaluated using two different heuristics for incremental construction to reduce total distance calculations as discussed in Section III-B. The incremental construction evaluates 32 reference points for each layer as well as the 6 highest variance dimensions and then selects which reference point or dimension to use for that index layer based on which had the minimum heuristic value. It should be noted that MISTIC is a greedy algorithm and so the heuristics are not guaranteed to find the global minima.

TABLE III

SPEEDUP OF USING THE STDDEV HEURISTIC OVER THE SUMSQRS HEURISTIC, WHERE THE SPEEDUP IS THE TOTAL RESPONSE TIME OF SUMSQRS OVER STDDEV.

Dataset	S_s	S_m	S_l
Wave	0.96	1.08	1.10
MSD	0.86	1.05	1.86
Bigcross	19.64	13.10	9.34
SuSy	3.65	2.66	2.13
Higgs	1.02	0.96	0.98

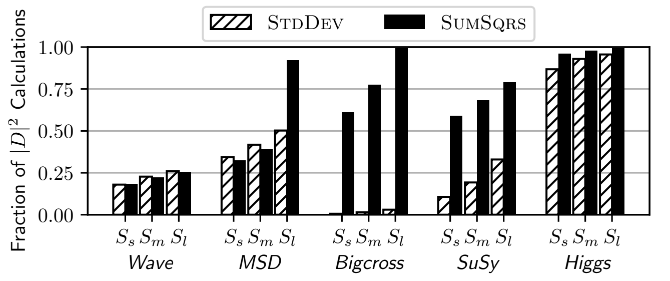


Fig. 4. The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$.

In Figure 4, the fraction of $|D|^2$ distance calculations (the number of distance calculations required of a brute force approach) needed by both heuristics is plotted for each dataset and selectivity level. The number of distance calculations is an estimate for the amount of work that will be performed on the GPU; therefore, a lower number of distance calculations is correlated with higher overall performance. For all selectivity levels of the *Wave* dataset and the first two selectivity levels of the *MSD* dataset, the SUMSQRS heuristic leads to fewer distance calculations. For every other dataset and selectivity level, the STDDEV heuristic results in fewer calculations. The SUMSQRS method fails to create a large number of partitions which yields poor performance particularly on the *Bigcross* and *SuSy* datasets. This is supported by Table III where the speedup of STDDEV over SUMSQRS is up to 19.64×.

The overall performance of MISTIC with each heuristic is not solely based on the number of distance calculations however, as observed in Table III where the speedup of STDDEV over SUMSQRS exceeds 1 even while performing more distance calculations on S_m and S_l of *Wave* and S_m of *MSD*. This is due to the STDDEV heuristic resulting in partitions which have similar number of points and therefore yielding similar workloads for the GPU threads that will search and refine the points in these partitions. On the GPU, if the amount of work between threads is unbalanced then the throughput of the device will be reduced due to the SIMT architecture. This makes it imperative to have a balanced workload and therefore an even distribution of points in the partitions is ideal for achieving the best performance on the GPU. STDDEV is a better heuristic than SUMSQRS because it creates more partitions resulting in fewer distance calculations and it distributes the points evenly across partitions resulting in better load balancing.

TABLE IV

THE AVERAGE SPEEDUP ACROSS SELECTIVITY LEVELS OF PERFORMING TREE TRAVERSALS SEARCHES OVER BINARY SEARCHES FOR EACH DATASET, WHERE SPEEDUP IS THE RATIO OF THE RESPONSE TIME FOR BINARY SEARCHES OVER TREE TRAVERSALS. THE AVERAGE NUMBER OF NON-EMPTY PARTITIONS IS INCLUDED FOR ANALYSIS.

Dataset	Speedup	Partitions
Wave	1.05	2,152
MSD	1.02	5,537
Bigcross	1.04	21,057
SuSy	1.36	27,284
Higgs	0.96	902

TABLE V

THE AVERAGE SPEEDUP ACROSS SELECTIVITY LEVELS FOR MISTIC OVER THE REFERENCE IMPLEMENTATIONS, WHERE SPEEDUP IS THE RATIO OF RESPONSE TIME OF THE REFERENCE IMPLEMENTATION OVER MISTIC.

Dataset	COSS	GDS-JOIN	BRUTE
Wave	1.17	1.43	1.02
MSD	1.17	1.70	1.36
Bigcross	2.65	1.84	5.74
SuSy	5.15	6.89	5.26
Higgs	2.52	1.78	2.36

MISTIC: Binary Search vs. Tree Traversal – In Section III-C we describe the two possible methods for searching the tree; tree traversals or binary searches. The average speedup across the selectivity levels of the self-join using MISTIC which uses the tree traversal described by Algorithm 2 as compared to a binary search is described in Table IV. The tree traversals are more efficient than the binary searches when the number of non-empty partitions in the index is higher due to the increased costs of binary searches as the size of the array being searched increases. A secondary factor in the efficiency of the searches is the ability of the tree traversal to short-circuit and terminate before checking each layer of the tree. This happens in datasets with over-dense regions where there is a higher chance of adjacent partitions being empty, as opposed to a more uniformly distributed dataset where the non-empty partitions are more likely adjacent to other non-empty partitions. We observe from Table IV that *SuSy* has the greatest speedup using tree traversals while also having the largest number of non-empty partitions on average across the selectivity levels. Only *Higgs* is faster with binary searches because it has few non-empty partitions.

E. Comparison to the Reference Implementations

Now that we have demonstrated key facets of the performance of MISTIC we compare it to the reference implementations (COSS, GDS-JOIN, and BRUTE). Figure 5 shows the total response time (s) vs. selectivity level across five real-world datasets. The average speedup for S_s, S_m, and S_l for MISTIC over the other implementations is given in Table V. We observe that MISTIC has the lowest response time in every instance except for Figure 5(a) where BRUTE has a lower response time due to the index construction overhead shown in Figure 3.

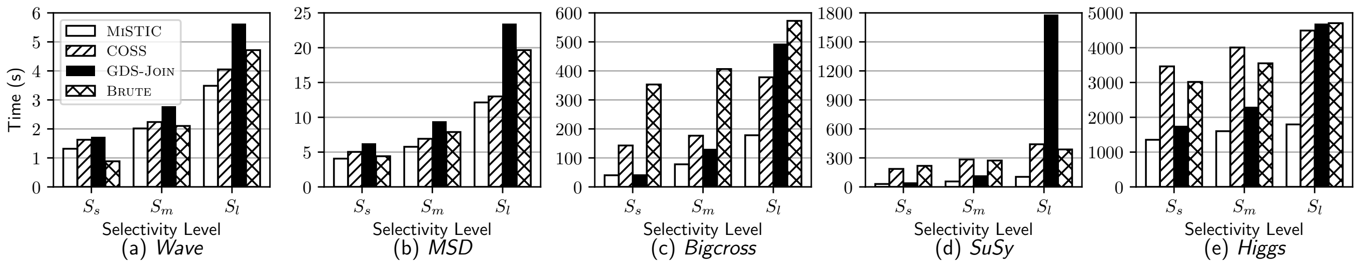


Fig. 5. Response time as a function of the three selectivity values across the five real-world datasets for each reference implementation.

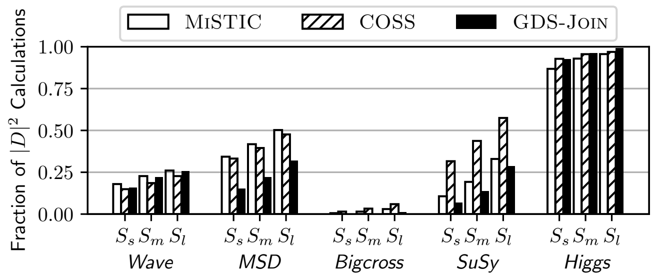


Fig. 6. The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$.

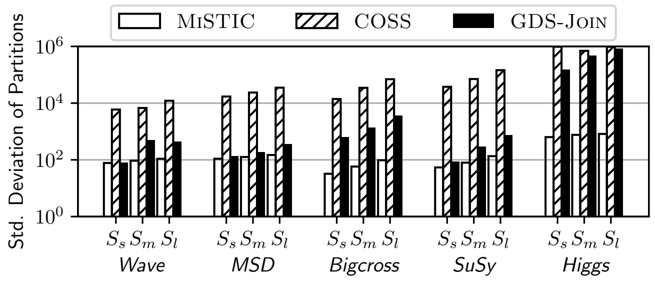


Fig. 7. Standard deviation of the number of points in each partition vs. selectivity values for the five real-world datasets.

Measuring Distance Calculations – In Figure 6, we plot the fraction of $|D|^2$ distance calculations that are performed for MiSTIC, COSS and GDS-Join (Brute is omitted because it performs $|D|^2$ calculations). The fraction of $|D|^2$ calculations represents the reduction of distance calculations due to the index as compared to Brute. Measuring the number of distance calculations needed is a good representation of the amount of work that each algorithm performs but does not directly correspond to the response time. In Figure 6, we observe that MiSTIC performs more distance calculations than either COSS or GDS-Join for the *Wave* and *MSD* datasets but has a lower response time (Figure 5). This behavior is also observed for the *SuSy* dataset at S_l where MiSTIC performs more calculations than GDS-Join but yields a speedup of 17.07× over GDS-Join.

We also observe in Figure 6 that the number of distance calculations needed for the highest selectivity level on the *Higgs* dataset approaches $|D|^2$. This is the effect of the *curse of dimensionality*, which is difficult to mitigate for these selectivity levels, even for the metric-based index COSS.

Workload Balancing – The GPU uses a Single Instruction Multiple Thread (SIMT) execution model which requires all threads within a warp to execute operations in lockstep. MiSTIC assigns one thread to each point in the dataset; therefore, threads belonging to separate partitions that are assigned to the same warp may have load imbalance due to differing numbers of comparisons between candidate points. Therefore having a similar number of points assigned to each partition improves workload balancing, and reduces the time it takes for all threads in a warp to complete their distance calculations. We hypothesize that one reason MiSTIC is faster than COSS and GDS-Join is because the variance of points in each partition is lower for MiSTIC than these reference implementations. To examine load balancing, Figure 7 shows the standard deviation of the number of points in each non-empty partition. MiSTIC has the lowest standard deviation, and therefore the best load balancing and most even distribution of work among threads. This is in part due to the StdDev heuristic used during incremental construction which prioritizes this distribution of points among the partitions. COSS has the highest standard deviation in every scenario but as described in prior work [12], workload imbalance is partially offset by assigning multiple threads to each point and batching the computations based on the amount of work.

Selecting Between Brute and MiSTIC– There are some cases where MiSTIC performs worse than the brute force approach, Brute (Figure 5). This is due to index construction overhead which requires a large fraction of the total time when there at low selectivity levels and/or when $|D|$ is small. Brute has a time complexity of $O(|D|^2)$ which is penalized less by smaller datasets. In these scenarios, the response times are low and occur when GPU acceleration is unwarranted.

Why is MiSTIC Faster? – As described above, MiSTIC sometimes performs more distance calculations than COSS or GDS-Join, but is faster than those algorithms across all datasets and selectivity levels. We summarize why MiSTIC is faster than the other approaches as follows: (i) MiSTIC uses tree traversals instead of binary searches to perform index searches. (ii) The StdDev heuristic leads MiSTIC to distribute the points evenly across the partitions, which results in a more balanced workload and increased performance. (iii) MiSTIC is robust to data characteristics because of the blending of metric- and coordinate-based partitioning strategies.

V. Conclusion

MiSTIC demonstrates that a blended approach to partitioning yields a more robust index than either a metric- or coordinate-based indexing method. Consequently, MiSTIC can be used instead of selecting a metric- or coordinate-based index based on the dataset characteristics. We show that MiSTIC's incremental construction substantially improves performance when coupled with the StdDev heuristic. Additionally, the novel reference point placement strategy improves the pruning efficiency of MiSTIC by intelligently placing reference points to maximize the variance of the points in the associated partitions. The experimental evaluation shows that MiSTIC is the best GPU index for range queries on large high-dimensional datasets, with an average speedup over the state-of-the art reference implementations of 2.53× and 2.73× for COSS and GDS-Join, respectively.

References

[1] S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, 2013, pp. 42–47.

[2] P. Čech, J. Lokoč, and Y. N. Silva, "Pivot-based approximate k-NN similarity joins for big high-dimensional data," *Information Systems*, vol. 87, p. 101410, 2020.

[3] D. Amagata, T. Hara, and C. Xiao, "Dynamic Set kNN Self-Join," in *IEEE 35th International Conference on Data Engineering*. IEEE, 2019, pp. 818–829.

[4] G. Navarro and N. Reyes, "Dynamic spatial approximation trees for massive data," in *Second International Workshop on Similarity Search and Applications*. IEEE, 2009, pp. 81–88.

[5] R. Mao, P. Zhang, X. Li, X. Liu, and M. Lu, "Pivot selection for metric-space indexing," *International Journal of Machine Learning and Cybernetics*, vol. 7, no. 2, pp. 311–323, 2016.

[6] D. V. Kalashnikov and S. Prabhakar, "Similarity join for low-and high-dimensional data," in *Eighth International Conference on Database Systems for Advanced Applications*. IEEE, 2003, pp. 7–16.

[7] A. Zimek, E. Schubert, and H.-P. Kriegel, "A survey on unsupervised outlier detection in high-dimensional numerical data," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 5, no. 5, pp. 363–387, 2012.

[8] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Computing Surveys*, vol. 33, no. 3, pp. 322–373, 2001.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, 2001.

[11] M. Gowanlock and B. Karsin, "Accelerating the similarity self-join using the GPU," *Journal of Parallel and Distributed Computing*, vol. 133, pp. 107–123, 2019.

[12] B. Donnelly and M. Gowanlock, "A coordinate-oblivious index for high-dimensional distance similarity searches on the GPU," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[13] B. Gallet and M. Gowanlock, "Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations," in *IEEE 29th International Conference on High Performance Computing, Data, and Analytics*, 2022, pp. 135–144.

[14] J. Zhou, Q. Guo, H. V. Jagadish, L. Krcal, S. Liu, W. Luan, A. K. H. Tung, Y. Yang, and Y. Zheng, "A Generic Inverted Index Framework for Similarity Search on the GPU," in *IEEE 34th International Conference on Data Engineering*, 2018, pp. 893–904.

[15] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[16] D. V. Kalashnikov, "Super-EGO: fast multi-dimensional similarity join," *The VLDB Journal*, vol. 22, no. 4, pp. 561–585, 2013.

[17] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching," *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[18] G. R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces (survey article)," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 517–580, 2003.

[19] M. Gowanlock and B. Karsin, "Gpu-accelerated similarity self-join for multi-dimensional data," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–9.

[20] D. V. Kalashnikov and S. Prabhakar, "Fast similarity join for multi-dimensional data," *Information Systems*, vol. 32, no. 1, pp. 160–177, 2007.

[21] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 379–388, 2001.

[22] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[23] J. Gan and Y. Tao, "Dbscan revisited: Mis-claim, un-fixability, and approximation," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 519–530.

[24] M. Gowanlock, B. Gallet, and B. Donnelly, "Optimization and Comparison of Coordinate- and Metric-Based Indexes on GPUs for Distance Similarity Searches," in *International Conference on Computational Science*. Springer, 2023, pp. 357–364.

[25] V. Volkov, "Better performance at lower occupancy," https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf, 2010, accessed June 22, 2024.

[26] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search," in *IEEE 31st International Conference on Data Engineering*, 2015, pp. 591–602.

[27] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang, "Pivot-based metric indexing," in *Proceedings of the VLDB Endowment*, 2017, pp. 1058–1069.

[28] B. Bustos, G. Navarro, and E. Chávez, "Pivot selection techniques for proximity searching in metric spaces," *Pattern Recognition Letters*, vol. 24, no. 14, pp. 2357–2366, 2003.

[29] U. M. L. Repository, "Wave Energy Converters Data Set," https://archive.ics.uci.edu/ml/datasets/Wave+Energy+Converters, accessed June 22, 2024.

[30] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval*, 2011, pp. 591–596.

[31] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++ a clustering algorithm for data streams," *Journal of Experimental Algorithmics*, vol. 17, pp. 2.1–2.30, 2012.

[32] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.