Improving the Performance of High-Dimensional *k*NN Retrieval through Localized Dataspace Segmentation and Hybrid Indexing

Michael A. Schuh, Tim Wylie, and Rafal A. Angryk

Montana State University, Bozeman, MT 59717-3880, USA
{michael.schuh, timothy.wylie, angryk}@cs.montana.edu

Abstract. Efficient data indexing and nearest neighbor retrieval are challenging tasks in high-dimensional spaces. This work builds upon our previous analyses of iDistance partitioning strategies to develop the backbone of a new indexing method using a heuristic-guided hybrid index that further segments congested areas of the dataspace to improve overall performance for exact k-nearest neighbor (kNN) queries. We develop data-driven heuristics to intelligently guide the segmentation of distance-based partitions into spatially disjoint sections that can be quickly and efficiently pruned during retrieval. Extensive tests are performed on k-means derived partitions over datasets of varying dimensionality, size, and cluster compactness. Experiments on both real and synthetic high-dimensional data show that our new index performs significantly better on clustered data than the state-of-the-art iDistance indexing method.

Keywords: High-dimensional, Partitioning, Indexing, Retrieval, kNN

1 Introduction

Modern database-oriented applications are overflowing with rich information composed of an ever-increasing amount of high-dimensional data. While massive data storage is becoming routine, efficiently indexing and retrieving it is still a practical concern. A frequent and costly retrieval task on these databases is k-nearest neighbor (kNN) search, which returns the k most similar records to any given query record. While a database management system (DBMS) is highly optimized for a few dimensions, most traditional indexing algorithms (*e.g.*, the B-tree and R-tree families) degrade quickly as the number of dimensions increase, and eventually a sequential (linear) scan of every single record in the database becomes the fastest retrieval method.

Many algorithms have been proposed in the past with limited success for true high-dimensional indexing, and this general problem is commonly referred to as *the curse of dimensionality* [4]. These issues are often mitigated by applying dimensionality reduction techniques before using popular multi-dimensional indexing methods, and sometimes even by adding application logic to combine multiple independent indexes and/or requiring user involvement during search. However,

modern applications are increasingly employing highly-dimensional techniques to effectively represent massive data, such as the highly popular 128-dimensional SIFT features [11] in Content-Based Image Retrieval (CBIR). Therefore, it is of great importance to be able to comprehensively index all dimensions for a unified similarity-based retrieval model.

This work builds upon our previous analyses of standard iDistance partitioning strategies [14, 19], with the continued goal of increasing overall performance of indexing and retrieval for kNN queries in high-dimensional dataspaces. We assess performance efficiency by the total and accessed number of B$^+$-tree nodes, number of candidate data points returned from filtering, and the time taken to build the index and perform queries. In combination, these metrics provide a highly descriptive and unbiased quantitative benchmark for independent comparative evaluations. We compare results against our open-source implementation of the original iDistance algorithm[1].

The rest of the paper is organized as follows. Section 2 highlights background and related work, while Section 3 covers the basics of iDistance and then introduces our hybrid index and heuristics for iDStar. Section 4 presents experiments and results, followed by a brief discussion of key findings in Section 5. We close with our conclusions and future work in Section 6.

2 Related Work

The ability to efficiently index and retrieve data has become a silent backbone of modern society, and it defines the capabilities and limitations of practical data usage. While the one-dimensional B$^+$-tree [2] is foundational to the modern relational DBMS, most real-life data has many dimensions (attributes) that would be better indexed together than individually. Mathematics has long-studied the partitioning of multi-dimensional metric spaces, most notably Voronoi Diagrams and the related Delaunay triangulations [1], but these theoretical solutions can often be too complex for practical application. To address this issue, many approximate techniques have been proposed to be used in practice. One of the most popular is the R-tree [7], which was developed with minimum bounding rectangles (MBRs) to build a hierarchical tree of successively smaller MBRs containing objects in a multi-dimensional space. The R*-tree [3] enhanced search efficiency by minimizing MBR overlap. However, these trees (and most derivations) quickly degrade in performance as the dimensions increase [5, 12].

Research has more recently focused on creating indexing methods that define a one-way lossy mapping function from a multi-dimensional space to a one-dimensional space that can then be indexed efficiently in a standard B$^+$-tree. These lossy mappings require a filter-and-refine strategy to produce exact query results, where the one-dimensional index is used to quickly retrieve a subset of the data points as candidates (the filter step), and then each of these candidates is verified to be within the specified query region in the original multi-dimensional space (the refine step). Since checking the candidates in the actual dataspace is

[1] Publicly available at: http://code.google.com/p/idistance/

costly, the goal of the filter step is to return as few candidates as possible while retaining the exact results to satisfy the query.

The Pyramid Technique [5, 22] was one of the first prominent methods to effectively use this strategy by dividing up the d-dimensional space into $2d$ pyramids with the apexes meeting in the center of the dataspace. For greater simplicity and flexibility, iMinMax(θ) [12, 16] was developed with a global partitioning line θ that can be moved based on the data distribution to create more balanced partitions leading to more efficient retrieval. Both the Pyramid Technique and iMinMax(θ) were designed for multi-dimensional range queries, and extending to high-dimensional kNN queries is not a trivial task.

First published in 2001, iDistance [10, 20] specifically addressed exact kNN queries in high-dimensional spaces and was proven to be one of the most efficient, state-of-the-art techniques available. In recent years, iDistance has been used in a number of demanding applications, including large-scale image retrieval [21], video indexing [15], mobile computing [8], peer-to-peer systems [6], and video surveillance retrieval [13]. As information retrieval from high-dimensional and large-scale databases becomes more ubiquitous, the motivations for this research will only increase. Many recent works have shifted focus to approximate nearest neighbors [9, 18] which can generally be retrieved faster, but these are outside the scope of efficient exact kNN retrieval presented in this paper.

3 Towards iDStar

Our hybrid index and accompanying heuristics build off the initial concept of iDistance with an approximate Voronoi tessellation of the space using hyperspheres. Therefore we start with a brief overview of the original iDistance algorithm before introducing iDStar.

3.1 iDistance

The basic concept of iDistance is to segment the dataspace into disjoint spherical partitions, where all points in a partition are *indexed by their distance* (hence "iDistance") to the reference point of that partition. This results in a set of one-dimensional distance values, each related to one or more data points, for each partition, that are all indexed in a single standard B^+-tree. The algorithm was motivated by the ability to use arbitrary reference points to determine the (dis)similarity between any two data points in a metric space, allowing single dimensional ranking and indexing of data points regardless of the dimensionality of the original space [10, 20].

Building the Index Here we focus only on *data-based* partitioning strategies, which adjusts the size and location of partitions in the dataspace based on the underlying data distribution, which greatly increases retrieval performance in real-world settings [10, 14, 20]. For all partitioning strategies, data points are assigned to the single closest partition based on Euclidean distance to each partitions' representative reference point.

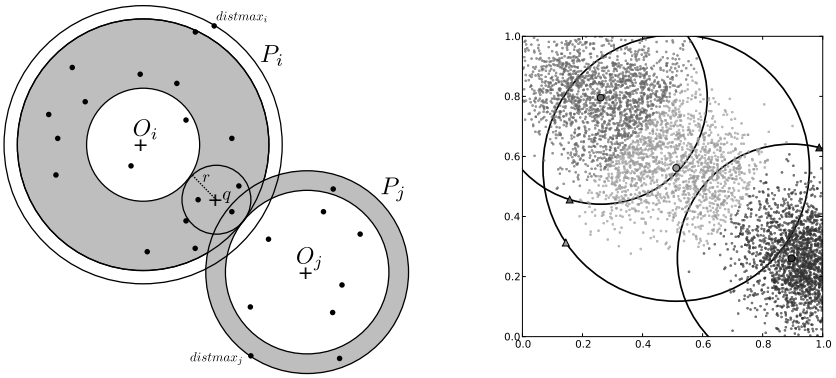

Fig. 1: (a) A query sphere q with radius r and the searched regions (shaded) in the two overlapping partitions P_i and P_j defined by their reference points O_i and O_j, and radii $distmax_i$ and $distmax_j$, respectively. (b) An example two dimensional dataset with three cluster-based partitions and their point assignments and radii.

Formally, we have a set of partitions $\mathbb{P} = \langle P_1, \ldots, P_M \rangle$ with respective reference points $\mathbb{O} = \langle O_1, \ldots, O_M \rangle$. We will let the number of points in a partition be denoted $|P_i|$ and the total number of points in the dataset be N. After the partitions are defined, a mapping scheme is applied to create separation in the underlying B$^+$-tree between each partition, ensuring that any given index value represents a unique distance for exactly one partition.

Given a partition P_i with reference point O_i, the index value y_p for a point p assigned to this partition is defined by Equation 1, where $dist()$ is any metric distance function, i is the partition index, and c is a constant multiplier for creating the partition separation. While constructing the index, each partition P_i records the distance of its farthest point as $distmax_i$. We can safely set $c = 2\sqrt{d}$, which is twice the maximum possible distance of two points in the d-dimensional unit space of the data, and therefore no index value in partition P_i will clash with values in any other partition $P_{j \neq i}$.

$$y_p = i \times c + dist(O_i, p) \tag{1}$$

Querying the Index The index should be built in such a way that the filter step returns the fewest possible candidate points without missing the true k-nearest neighbors. Fewer candidates reduces the costly refinement step which must verify the true multi-dimensional distance of each candidate from the query point. Performing a query q with radius r consists of three steps: 1) determine the set of partitions to search, 2) calculate the search range for each partition, and 3) retrieve the candidate points and refine by their true distances.

Figure 1(a) shows an example query sphere contained completely within partition P_i and intersecting partition P_j, as well as the shaded ranges of each partition that need to be searched. For each partition P_i and its $distmax_i$, the query sphere overlaps the partition if the distance from the edge of the query sphere

to the reference point O_i is less than $distmax_i$, as defined in Equation 2. There are two possible cases of overlap: 1) q resides within P_i, or 2) q is outside of P_i, but the query sphere still intersects it. In the first case, the partition needs to be searched both inward and outward from the query point over the range $(q \pm r)$, whereas in the second case, a partition only needs to be searched inward from the edge ($distmax_i$) to the farthest point of intersection. For additional details, we refer the reader to the original works [10, 20].

$$dist(O_i, q) - r \leq distmax_i \tag{2}$$

3.2 iDStar

Our previous work showed that iDistance stabilizes in performance by accessing an entire partition (k-means cluster) to satisfy a given query, despite dataset size and dimensionality [14]. While only accessing a single partition is already significantly more efficient than sequential scan, this hurdle was the main motivation to explore further dataspace segmentation to enhance retrieval performance. We achieve this additional segmentation with the creation of intuitive heuristics applied to a novel hybrid index. These extensions are similar to the works of the iMinMax(θ) [12] and recently published SIMP [17] algorithms, whereby we can incorporate additional dataspace knowledge at the price of added algorithm complexity and performance overhead. This work proves the feasibility of our approaches and lays the foundations for a new indexing algorithm, which we introduce here as iDStar.

Essentially, we aim to further separate dense areas of the dataspace by *splitting* partitions into disjoint *sections* corresponding to separate *segments* of the B^+-tree that can be selectively pruned during retrieval. The previous sentence denotes the technical vocabulary we will use to describe the segmentation process. In other words, we apply a given number of dimensional *splits* in the dataspace which *sections* the partitions into B^+-tree *segments*.

We develop two types of segmentation based on the scope of splits: 1) **Global**, which splits the entire dataspace (and consequently any intersecting partitions), and 2) **Local**, which explicitly splits the dataspace of each partition separately. While different concepts, the general indexing and retrieval algorithm modifications and underlying affects on the B^+-tree are similar. First, the mapping function is updated to create a constant segment separation within the already separated partitions. Equation 3 describes the new index with the inclusion of the sectional index j, and s as the total number of splits applied to the partition. Note that a partition is divided into 2^s sections, so we must appropriately bound the number of splits applied. Second, after identifying the partitions to search, we must identify the sections within each partition to actually search, as well as their updated search ranges within the B^+-tree. This also requires the overhead of section-supporting data structures in addition to the existing partition-supporting data structures.

$$y_p = i \times c + j \times \frac{c}{2^s} + dist(O_i, p) \tag{3}$$

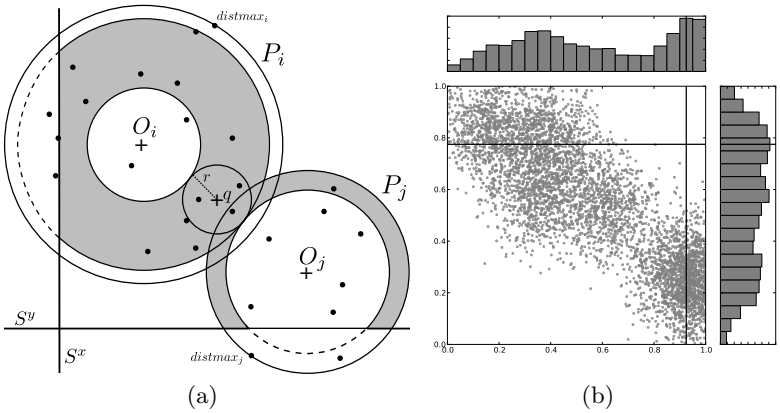

Fig. 2: (a) Conceptual global splits S^x and S^y and their effect on query search ranges. (b) The histogram-based splits applied by the G2 heuristic on the example dataset.

Global The global segmentation technique is inspired by the Pyramid and iMinMax(θ) methods, whereby we adjust the underlying index based on the data distributions in the space. A global split is defined by a tuple of dimension and value, where the given dimension is split on the specified value. There can be only one split in each dimension, so given the total number of splits s, each partition may have up to 2^s sections. Because the splits occur across the entire dataspace, they may intersect (and thereby segment) any number of partitions (including zero). An added benefit of global splits is the global determination of which sections require searching, since all partitions have the same global z-curve ordering. We introduce three heuristics to optimally choose global splits.

- (**G1**) Calculate the median of each dimension as the proposed split value and rank the top s dimensions to split in order of split values nearest to 0.5, favoring an even 50/50 dataspace split.
- (**G2**) Calculate an equal-width histogram in each dimension, selecting the center of the highest frequency bin as the proposed split value, and again ranking dimensions to split by values nearest to the dataspace center.
- (**G3**) Calculate an equal-width histogram in each dimension, selecting the center of the bin that intersects the most partitions, ranking the dimensions first by number of partitions intersected and then by their proximity to dataspace center.

Figure 2(a) shows our iDistance example with a split in each dimension (S^y, S^x). We can see that a given split may create empty partition sections, or may not split a partition at all. We explicitly initialize these data structures with a known value representing an "empty" flag. Therefore, we only check for query overlap on non-empty partitions and non-empty sections. For the example dataset and partitions in Figure 1(b), the G2 heuristic applies the splits shown in Figure 2(b).

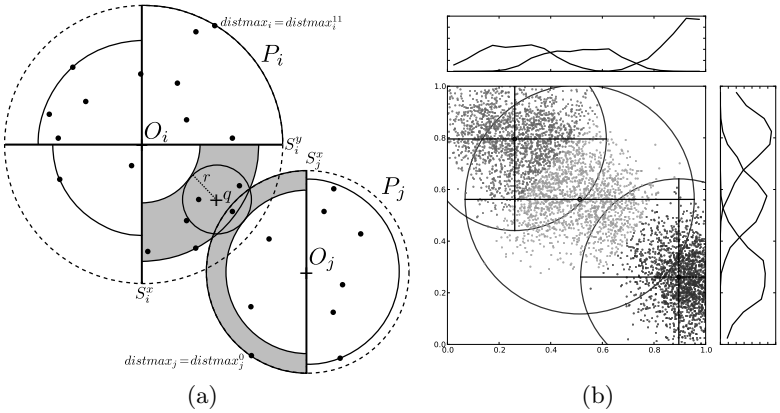

Fig. 3: (a) Conceptual local splits S_i^x, S_i^y, and S_j^x and their effect on query search ranges in overlapping partitions and sections. (b) The local population-based splits applied to each partition by the L3 heuristic on the example dataset.

Local We developed the local technique as a purposeful partition-specific segmentation method based only on local information of the given partition rather than the global data characteristics of the entire space. Local splits are defined by a tuple of partition and dimension, where the given partition is split in the specified dimension. Unlike the global technique, we do not need the values of each dimensional split because all splits are made directly on the partition reference point location. Figure 3(a) shows a conceptual example of local splits.

During index creation we maintain a $distmax_i$ of each partition P_i, and we now we do this for sections too ($distmax_i^j$ for partition P_i and section j). This allows the addition of a section-based query overlap filter which can often prune away entire partitions that would have otherwise been searched. This can be seen in Figure 3(a), where the original partition $distmax$ is now a dotted line and a new $distmax$ is shown for each individual partition section.

We introduce three heuristics to choose local splits. Within each partition, we want to split the data as evenly as possible using the added spatial information. Thus, for all three methods, we rank the dimensions to split by those closest to an equal (50/50) population split. We denote s as the maximum number of splits any partition can have.

- (**L1**) Uniformly apply s splits to each partition using the top ranked.
- (**L2**) Use a population threshold as a cutoff criteria, so each partition could have up to s splits. The intuition here is that if a split only segments a reasonably small portion of the data, it probably is not worth the additional overhead to keep track of it.
- (**L3**) Use s to calculate the maximum number of underlying B$^+$-tree segments created by method L1, but then redistribute them based on partition population. This automatically applies additional splits to dense areas of the dataspace while removing splits where they are not needed and maintain an upper bound on B$^+$-tree segments.

Formally, we have a set of splits $\mathbb{S} = \langle S_1 \ldots, S_M \rangle$ where S_i represents the set of dimensions to split on for partition P_i. Given a maximum granularity for the leaves in the B$^+$-tree, we can calculate how many splits each partition should have as a percentage of its population to the total dataset size. Equation 4 gives this assignment used by L3. For example, we can see in Figure 3(b) that the center cluster did not meet the population percentage to have two splits. Thus, the only split is on the dimension that has the closest 50/50 split of the data.

$$|S_i| = \left\lfloor \lg \frac{|P_i|}{N} \cdot M \cdot 2^s \right\rfloor \tag{4}$$

Hybrid Indexing and Optimization The global and local dataspace segmentation generates a hybrid index with one-dimensional distance-based partitions subsequently segmented in a tune-able subset of dimensions. Due to the exponential growth of the tree segments, we must limit the number of dimensions we split on. For the local methods, L3 maintains this automatically. For the spatial subtree segmentation, we use z-curve ordering of each partition's sections, efficiently encoded as a ternary bitstring representation of all overlapping sections that is quickly decomposable to a list of section indices. Figure 3(b) shows an example of this ordering with $distmax_i^{11}$, which also happens to be the overall partition $distmax_i$. Since P_i is split twice, we have a two-digit binary encoding of quadrants, with the upper-right of '11' equating to section 3 of P_i.

4 Experiments and Results

We methodically determine the effectiveness of our hybrid index and segmentation heuristics over a wide range of dataset characteristics that lead to generalized conclusions about their overall performance.

Every experimental test reports a set of statistics describing the index and query performance of that test. As an attempt to remove machine-dependent statistics, we use the number of B$^+$-tree nodes instead of page accesses when reporting query results and tree size. Tracking nodes accessed is much easier within the algorithm and across heterogeneous systems, and is still directly related to page accesses through the given machine's page size and B$^+$-tree leaf size. We primarily highlight three statistics from tested queries: 1) the number of candidate points returned during the filter step, 2) the number of nodes accessed in the B$^+$-tree, and 3) the time taken (in milliseconds) to perform the query and return the final exact results. Other descriptive statistics included the B$^+$-tree size (total nodes) and the number of dataspace partitions and sections (of partitions) that were checked during the query. We often express the ratio of candidates and nodes over the total number of points in the dataset and the total number of nodes in the B$^+$-tree, respectively, as this eliminates skewed results due to varying the dataset.

The first experiments are on synthetic datasets so we can properly simulate specific dataset characteristics, followed by further tests on real datasets. All

artificial datasets are given a specified number of points and dimensions in the d-dimensional unit space $[0.0, 1.0]^d$. For clustered data, we provide the number of clusters and the standard deviation of the independent Gaussian distributions centered on each cluster (in each dimension). The cluster centers are randomly generated during dataset creation and saved for later use. For each dataset, we randomly select 500 points as kNN queries (with $k = 10$) for all experiments, which ensures that our query point distribution follows the dataset distribution.

Sequential scan (SS) is often used as a benchmark comparison for worst-case performance. It must check every data point, and even though it does not use the B^+-tree for retrieval, total tree nodes provides the appropriate worst-case comparison. Note that all data fits in main memory, so all experiments are compared without depending on the behaviors of specific hardware-based disk-caching routines. In real-life however, disk-based I/O bottlenecks are a common concern for inefficient retrieval methods. Therefore, unless sequential scan runs significantly faster, there is a greater implied benefit when the indexing method does not have to access every data record, which could potentially be on disk.

4.1 First Look: Extensions & Heuristics

The goal of our first experiment is to determine the feasibility and effectiveness of our global and local segmentation techniques and their associated heuristics over dataset size and dimensionality. We create synthetic datasets with 100,000 (100k) data points equally distributed among 12 clusters with a 0.05 standard deviation (*stdev*) in each dimension, ranging from 8 to 512 dimensions. The true cluster centers are used as partition reference points and each heuristic is independently tested with 2, 4, 6, and 8 splits. Here we compare against regular iDistance using the same true center (TC) reference points. Due to space limits, we only present one heuristic from each type of segmentation scope, namely G2 and L1. These methods are ideal because they enforce the total number of splits specified, and do so in a rather intuitive and straightforward manner, exemplifying the over-arching global and local segmentation concepts. We also note that the other heuristics generally perform quite similar to these two, so presentation of all results would probably be superfluous.

Figure 4 shows the performance of G2 compared to TC (regular iDistance) over candidates, nodes accessed, and query time. Unfortunately, above 64 dimensions we do not see any significant performance increase by any global heuristic. Note that we do not show Sequential Scan (SS) results here because they are drastically worse. For example, we see TC returning approximately 8k candidates versus the 100k candidates SS must check.

The same three statistics are shown in Figure 5 for L1 compared to TC. Unlike G2, here we can see that L1 greatly increases performance by returning significantly fewer candidates in upwards of 256 dimensions. Above 64 dimensions, we can see the increased overhead of nodes being accessed by the larger number of splits (6 and 8), but despite these extra nodes, all methods run marginally faster than TC because of the better candidate filtering. It should also be clear that the number of splits is directly correlated with performance, as we can see

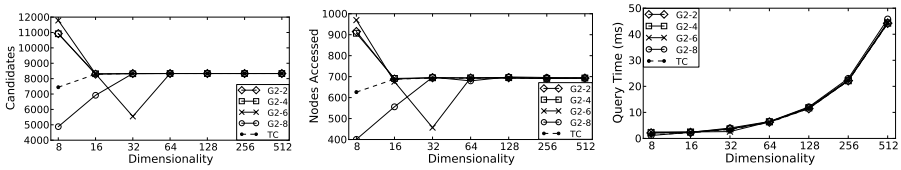

Fig. 4: Global heuristic G2 with 2, 4, 6, and 8 splits over dataset dimensionality.

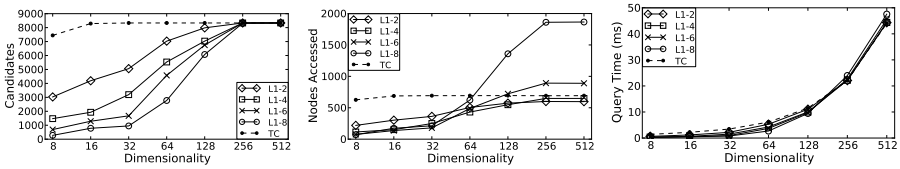

Fig. 5: Local heuristic L1 with 2, 4, 6, and 8 splits over dataset dimensionality.

that 8 splits performs better than 6, which performs better than 4, etc. Of course, the increased nodes accessed is a clear indication of the trade-off between the number of splits and the effort to search each partition section to satisfy a given query. Eventually, the overhead of too many splits will outweigh the benefit of enhanced filtering power.

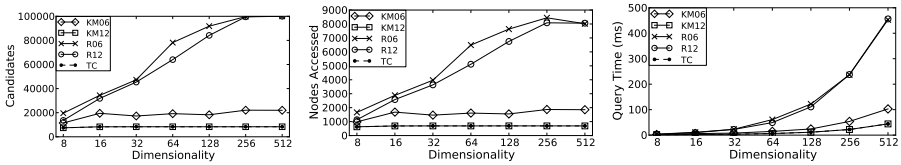

Fig. 6: Comparison of TC to k-means (KM) and RAND with 6 (KM6, R6) and 12 (KM12, R12) partitions over dataset dimensionality. Note that the lines for TC and KM12 entirely overlap.

Lastly, Figure 6 revisits a finding from our previous work [14] to show the general performance comparison between TC and k-means (KM), with the random (RAND) space-based partitioning method included as a naïve data-blind approach – all using regular iDistance. Because we establish 12 well-defined clusters in the dataspace, we can use this data knowledge to test KM and RAND, each with 6 and 12 partitions, and compare them to TC (which uses the true 12 cluster centers). Unsurprisingly, we see that RAND (R6 and R12) quickly degrades in performance as dimensionality increases. We also find that KM12 performs almost equivalent to TC throughout the tests. This provides ample justification for the benefits of optimal data clustering for efficient dataspace partitioning, and allows us to focus entirely on k-means for the remainder of our experiments, which is also a much more realistic condition in applications using non-synthetic datasets.

4.2 Investigating Cluster Density Effects

The second experiment extends the analysis of our heuristics' effectiveness over varying tightness/compactness of the underlying data clusters. We again generate synthetic datasets with 12 clusters and 100k points, but this time with cluster *stdev* ranging from 0.25 to 0.005 in each dimension over a standard 32-dimensional unit space. Figure 7 reiterates the scope of our general performance, showing SS and RAND as worst-case benchmarks and the rapid convergence of TC and KM as clusters become sufficiently compact and well-defined. Notice however, that both methods stabilize in performance as the *stdev* decreases below 0.15. Essentially, the clusters become so compact that while any given query typically only has to search in a single partition, it ends up having to search the entire partition to find the exact results.

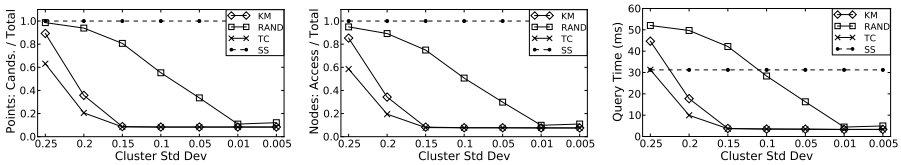

Fig. 7: Comparison of iDistance partitioning strategies over cluster standard deviation.

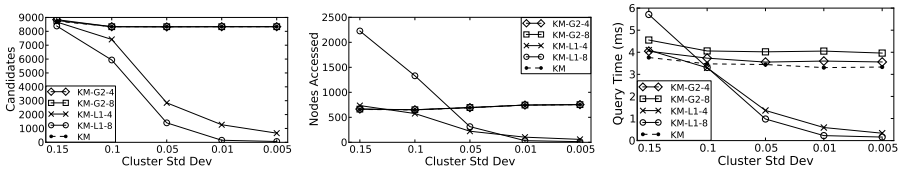

Fig. 8: Heuristics G2 and L1 versus KM over cluster standard deviation.

Figure 8 compares our heuristics G2 and L1 applied to k-means derived partitions (labeled as KM, using regular iDistance). To simplify the charts, we only look at 4 and 8 splits, which still provide an adequate characterization of performance. We also only look at a cluster *stdev* of 0.15 and less, to highlight the specific niche of poor performance we are addressing. Unfortunately, we again see that G2 performs no better than KM, and it even takes slightly more time due to the increased retrieval overhead and lack of better candidate pruning. However, L1 performance drastically improves over the stalled out KM partitions, which more than proves the effectiveness of localized segmentation heuristics.

4.3 A Real-world Comparison

Our last experiment uses a real-world dataset consisting of 90 dimensions and over 500k data points representing recorded songs from a large music archive[2]. First, we methodically determine the optimal number of clusters to use for

[2] Publicly available at: http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD

k-means, based on our previous discovery of a general iDistance performance plateau surrounding this optimal number [14]. Using this performance plateau to choose k represents a best practice for iDistance. Although omitted for brevity, we tested values of k from 10 to 1000, and found that iDistance performed best around $k = 120$ clusters (partitions). We also discard global techniques from discussion given their poor performance on previous synthetic tests.

Figure 9 shows our local L1 and L3 methods, each with 4 and 8 splits, over a varying number of k-means derived partitions. Since this dataset is unfamiliar, we include baseline comparisons of KM and SS, representing standard iDistance performance and worst-case sequential scan performance, respectively. We see that both local methods significantly outperform KM and SS over all tests. Also note that L3 generally outperforms L1 due to the more appropriate population-based distribution of splits, which translates directly to more balanced segments of the underlying B^+-tree.

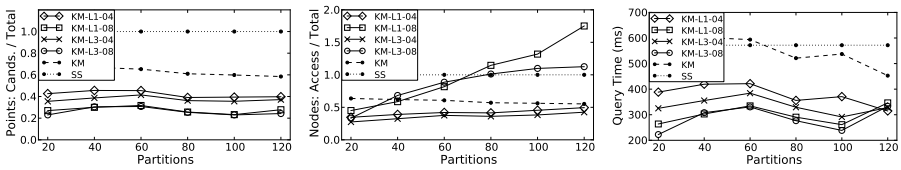

Fig. 9: Results of local methods L1 and L3 versus iDistance on real data.

5 Discussion

One topic worth discussion is the placement of reference points for creating partitions in the dataspace. Building on previous works [10, 14, 20], we continue to use the k-means algorithm to cluster the data and obtain a list of cluster centers to be used directly as reference points. However, Figure 1(b) hints at a possible shortcoming of using k-means – specifically when building closest-assignment partitions. Notice how large the center cluster is because of the farthest assigned data point, and how much additional partition overlap that creates. We hypothesize that just a few outliers in the data can have a tremendously negative effect on index retrieval due to this property of k-means.

While large overlapping clusters may be somewhat mitigated by increasing and fine-tuning the total number of clusters in k-means – as we do in the prior experiment – the more elegant solution tackles the problem of clustering directly. In a related work, we began investigating the idea of *clustering for the sake of indexing*, by using a custom Expectation Maximization (EM) algorithm to learn optimal cluster arrangements designed explicitly for use as reference points within iDistance [19]. Our current findings further the motivation for this type of work with possibly new knowledge and insights of how we might define more optimal clusters for partitions.

6 Conclusions and Future Work

This work introduced the foundations of iDStar, a novel hybrid index for efficient and exact kNN retrieval in high-dimensional spaces. We developed global and local segmentation heuristics in the dataspace and underlying B^+-tree, but only localized partition segmentation proved effective. Results show we can significantly outperform the state-of-the-art iDistance index in clustered data, while performing no worse in unfavorable conditions. This establishes a new performance benchmark for efficient and exact kNN retrieval that can be independently compared to and evaluated against by the community.

Future work surrounds the continued development of iDStar into a fully-fledged indexing algorithm, with specific focus on real-world applications, extended optimizations, and a public repository for the community. Several directions of continued research include testing other heuristics to better guide segmentation and exploring new hybrid components to enable other filtering capabilities. We are also investigating the feasibility of dynamic index updates to efficiently respond and tune to online and unpredictable retrieval environments.

Acknowledgements

This work was supported in part by two NASA Grant Awards: 1) No. NNX09AB03G, and 2) No. NNX11AM13A. A special thanks to all research and manuscript reviewers.

References

1. Aurenhammer, F.: Voronoi diagrams – a survey of a fundamental geometric data structure. ACM Comput. Surv. 23, 345–405 (September 1991)
2. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Informatica 1, 173–189 (1972)
3. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. of ACM SIGMOD Inter. Conf. on management of data. pp. 322–331 (1990)
4. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
5. Berchtold, S., Bhm, C., Kriegal, H.P.: The pyramid-technique: towards breaking the curse of dimensionality. In: Proc. of ACM SIGMOD Inter. Conf. on management of data. vol. 27, pp. 142–153 (1998)
6. Doulkeridis, C., Vlachou, A., Kotidis, Y., Vazirgiannis, M.: Peer-to-peer similarity search in metric spaces. In: VLDB '07 (2007)
7. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of the ACM SIGMOD Inter. Conf. on Management of data. pp. 47–57 (1984)
8. Ilarri, S., Mena, E., Illarramendi, A.: Location-dependent queries in mobile contexts: Distributed processing using mobile agents. IEEE TMC 5, 1029–1043 (2006)
9. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proc. of the 30th annual ACM Sym. on Theory of computing. pp. 604–613. STOC'98, ACM (1998)
10. Jagadish, H.V., Ooi, B.C., Tan, K.L., Yu, C., Zhang, R.: iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. 30(2), 364–397 (jun 2005)

11. Lowe, D.: Object recognition from local scale-invariant features. In: The Proc. of the 7th IEEE Inter. Conf. on Computer Vision. vol. 2, pp. 1150–1157 (1999)
12. Ooi, B.C., Tan, K.L., Yu, C., Bressan, S.: Indexing the edges: a simple and yet efficient approach to high-dimensional indexing. In: Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Sym. on Principles of db systems. pp. 166–174. PODS '00 (2000)
13. Qu, L., Chen, Y., Yang, X.: idistance based interactive visual surveillance retrieval algorithm. In: Intelligent Computation Technology and Automation. ICICTA'08, vol. 1, pp. 71–75 (Oct 2008)
14. Schuh, M.A., Wylie, T., Banda, J., Angryk, R.A.: A comprehensive study of idistance partitioning strategies for knn queries and high-dimensional data indexing. In: Proc. of the 29th BNCOD Conf. (2013)
15. Shen, H.T.: Towards effective indexing for very large video sequence database. In: SIGMOD Conference. pp. 730–741 (2005)
16. Shi, Q., Nickerson, B.: Decreasing Radius K-Nearest Neighbor Search Using Mapping-based Indexing Schemes. Tech. rep., University of New Brunswick (2006)
17. Singh, V., Singh, A.K.: Simp: accurate and efficient near neighbor search in high dimensional spaces. In: Proc. of the 15th Inter. Conf. on Extending Database Technology. pp. 492–503. EDBT '12, ACM (2012)
18. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: Proc. of the 2009 ACM SIGMOD Inter. Conf. on Mgmt. of data. pp. 563–576. SIGMOD'09, ACM (2009)
19. Wylie, T., Schuh, M.A., Sheppard, J., Angryk, R.A.: Cluster analysis for optimal indexing. In: Proc. of the 26th FLAIRS Conf. (2013)
20. Yu, C., Ooi, B.C., Tan, K.L., Jagadish, H.V.: Indexing the Distance: An Efficient Method to KNN Processing. In: Proc. of the 27th Inter. Conf. on Very Large Data Bases. pp. 421–430. VLDB'01 (2001)
21. Zhang, J., Zhou, X., Wang, W., Shi, B., Pei, J.: Using high dimensional indexes to support relevance feedback based interactive images retrieval. In: Proc. of the 32nd inter. conf. on Very large data bases. pp. 1211–1214. VLDB '06 (2006)
22. Zhang, R., Ooi, B., Tan, K.L.: Making the pyramid technique robust to query types and workloads. In: Proc. 20th Inter. Conf. on Data Eng. pp. 313–324 (2004)