Distributed query-aware quantization for high-dimensional similarity searches

Gheorghi Guzun
San Jose State University
San Jose, California
gheorghi.guzun@sjsu.edu

Guadalupe Canahuate
The University of Iowa
Iowa City, Iowa
guadalupe-canahuate@uiowa.edu

ABSTRACT

The concept of similarity is used as the basis for many data exploration and data mining tasks. Nearest Neighbor (NN) queries identify the most similar items, or in terms of distance the closest points to a query point. Similarity is traditionally characterized using a distance function between multi-dimensional feature vectors. However, when the data is high-dimensional, traditional distance functions fail to significantly distinguish between the closest and furthest points, as few dissimilar dimensions dominate the distance function. Localized similarity functions, i.e. functions that only consider dimensions close to the query, quantize each dimension independently and only compute similarity for the dimensions where the query and the points fall into the same bin. These quantizations are query-agnostic. There is potential to improve accuracy when a query-dependent quantization is used.

In this paper we propose a Query dependent Equi-Depth (QED) on-the-fly quantization method to improve high-dimensional similarity searches. The quantization is done for each dimension at query time and localized scores are generated for the closest p fraction of the points while a constant penalty is applied for the rest of the points. QED not only improves the quality of the distance metric, but also improves query time performance by filtering out non relevant data. We propose a distributed indexing and query algorithm to efficiently compute QED. Our experimental results show improvements in classification accuracy as well as query performance up to one order of magnitude faster than Manhattan-based sequential scan NN queries over datasets with hundreds of dimensions.

1 INTRODUCTION

Nearest Neighbor (NN) searches over high-dimensional data are ubiquitous in information retrieval, machine learning, and multimedia data mining. These searches are often performed through k nearest neighbor (kNN) queries over multi-dimensional feature vectors. Spatial and multimedia objects can be represented as feature vectors characterizing their shape and/or content. Social network data objects, for instance can be represented by links with other data objects, history actions, preferences, etc. Application domains such as spatial data-bases[8], computer vision [5], multimedia and social network applications [29] can all benefit from a more efficient method for finding nearest neighbors in high dimensional spaces.

However, due to the rapid advancements in data generation and collection, it is increasingly challenging to process similarity searches in a rapid and meaningful way on these larger and more complex datasets. Existing methods for finding nearest neighbors using tree-based indexing for spatial data [36] and low dimensional data [13, 23, 24], suffer from the curse of dimensionality when applied to high-dimensional spaces. Moreover, not only processing time degrades, but also the ability to characterize similarity using a distance function for high-dimensional spaces is greatly reduced[3]. The reason is that distances between data points in high-dimensional spaces, are usually very concentrated around their average [7]. This makes it difficult to distinguish between the closest and furthest data points [3].

To overcome this limitation, localized distance functions [1, 37, 39] have been proposed in the literature. These functions only consider dimensions that are close to the query point to characterize similarity providing a better distinction between closer and further points and often improving the accuracy of the results. The IGrid index [1] efficiently supports the computation of a partial distance called PiDist and its performance scales well for high dimensional data. IGrid pre-process the data and define equi-populated partitions (bins) over each dimension independently. The points in these partitions are then mapped to buckets and only the points that fall into the same bucket as the query point are considered similar for that dimension. The points with the largest cumulative similarity are then retrieved after all dimensions have been processed. As stated in the paper, the accuracy of the results are affected by the binning strategy and the number of bins used.

In this paper, we expand on the ideas proposed in [1, 15] and define a query-dependent quantization strategy that further improves the accuracy of the results. The main idea is to use the query itself to determine a range for which the points falling within are considered similar. Note that considering a fixed interval around the query value for each dimension is not a viable option. Since the distribution of each attribute varies, a small interval could yield empty bins, while a large interval could include all the points. Determining the range needs to consider the data values in the attribute. We want to define a bin for each dimension where the number of points is roughly the same for each dimension. This novel function, called Query dependent Equi-Depth (QED) quantization, can be used with traditional distance functions (e.g. Euclidean, Manhattan) to improve their accuracy in high-dimensional spaces. In order to efficiently support QED over big-data, we design a distributed indexing and efficient query algorithm. The proposed approach includes the usage of compressed bitmap-based indexing and low level parallel bitwise operations.

Our approach does not require any pre-computations for quantization, or excessive storage or memory. On the contrary, the index requires less storage than the data itself. The index and query algorithms are designed for parallelism and can run on centralized systems as well as cluster systems. In this work we evaluate the distributed indexing and query processing on a Spark cluster, however it is suitable for other distributed environments as well. With QED quantization, as shown in our performance evaluation we observe an improvement in query time of up to one

order of magnitude when compared to Sequential Scan on high dimensional data.

We also evaluated the kNN classification accuracy of the proposed QED quantization on a set of nine high-dimensional datasets and observed an average improvement in accuracy of 2.4% for Manhattan distance and 10.95% for Hamming distance when using QED quantization.

The primary contributions of this paper can be summarized as follows:

- We present a bitmap-based, distributed index for answering similarity searches and nearest neighbor queries. The cumulative distance is computed in parallel through an optimized distributed aggregation that uses task mapping by slice depth at its core.
- We formalize the cost for the distributed aggregation and optimize the partition size to balance the memory shuffling and parallel processing trade-off.
- We propose a novel Query dependent Equi-Depth quantization (QED) for improved similarity searches. This dynamic quantization not only improves accuracy but also execution time because it reduces the amount of data processed by only considering, for each dimension, the closest points to the query.
- We introduce a power function using the number of tuples and the number of dimensions as a heuristic to determine the number of points that are considered similar in each dimension and empirically evaluated it.
- We evaluate the kNN classification accuracy of QED on a number of labeled real datasets and the performance of the proposed index and query algorithms in a distributed setting.

The rest of the paper is organized as follows. Section 2 presents related work on NN searches. Section 3 describes the index structure, the proposed quantization, and the parallel query optimization for similarity searches. Section 4 provides an experimental evaluation of QED over a Spark/Hadoop cluster. Finally, conclusions are presented in Section 5.

2 RELATED WORK

This section presents related work for high-dimensional similarity searches and nearest neighbor queries. The most recent exact nearest neighbor searches are using localized similarity functions. While some of the most recent approximate nearest neighbor searches use hashing or product-quantization techniques.

2.1 High-dimensional Similarity using Localized Functions

As described in [37], for high-dimensional applications where human cognition is the target judge of object similarity, it is more important to closely match a subset of attributes rather than provide some least total distance measurement over all the attributes. The similarity function, called Dynamic Partial Function (DPF) [37], considers only the smallest N distances between all dimensions to compute the similarity. Since the method is so sensitive to N, the k-N-match problem is introduced in [37] and the authors propose the use of a frequent k-N-match algorithm, where the most frequent k objects appearing in the k-N-match solutions for a range of N values. It is worth noting that DPF is not a distance since the triangle inequality does not hold.

In [1], quantization was used to create equi-populated partitions for each dimension to bound the worst case performance of the query. Quantization has been widely used to improve the accuracy of classifiers and clustering algorithms as it reduces the noise of the data and simplifies the models. Supervised methods use the class label and a training dataset to make an informed decision about the optimal split points. Unsupervised methods rely solely on the statistics collected about the data. Examples of unsupervised methods are equi-width (divide into intervals of the same length) and equi-populated or equi-depth (divide into intervals with the same number of data objects). Examples of supervised methods are entropy-Minimum Description Length Principle (MDLP) [9], chi-merge [25], class-attribute interdependent maximization (CAIM) [26], and Class-attribute Contingency Coefficient (CACC) [38], among others. A detail survey of quantization methods can be found in [11].

After quantization, each attribute is represented using discrete values and points are considered "close" if they fall into the same bin. Hamming distance is the preferred distance metric for discrete domains and is defined as:

$$\texttt{Hamm}(\texttt{x}, \texttt{y}) = \sum_{\texttt{i}=1}^{\texttt{d}} \begin{cases} 0 & \text{if } x_i = y_i \\ 1 & \text{otherwise} \end{cases}$$

where x and y are the high-dimensional quantized vectors, d is the number of dimensions and x_i denotes the value of x for dimension i. The evaluation of the Hamming function produces discrete values in the range $[0, d]$. This is a source of ambiguity when defining similarity as the distance score is the same for an increasing number of nearest neighbors. To break these ties a weighted hamming distance function can be used. `PiDist` [1] computes the normalized Manhattan distance over the continuous values for the dimensions where the two points fall into the same discretization range. When equi-populated ranges are used, the attribute is forced to follow a uniform distribution and the distance between the continuous values is used to capture similarity.

PiDist is defined as:

$$\texttt{PiDist}(X, Y, k_d) = \left[\sum_{i \in S[X,Y,k_d]} \left(1 - \frac{|x_i - y_i|}{m_i - n_i}\right)^p \right]^{\frac{1}{p}}$$

where k_d is the number of splits for each dimension, $S[X, Y, k_d]$ is the set of dimensions for which the two objects lie in the same range, and m_i and n_i are the upper and lower bounds of the corresponding range in dimension i.

This function accumulates benefit for each attribute for which a data object maps to the same quantization as the query object. It does not differentiate between data and query objects that do not map to the same quantization. Therefore, a data point is not excessively penalized for a few dissimilar attributes.

2.2 Approximate nearest neighbor searches

Given the difficulty of answering exact nearest neighbor searches, approximate nearest neighbor searches were introduced to solve the NN problem in high dimensional spaces [10, 12, 20–22]. Local Sensitive Hashing (LSH)[12] hashes the data so that similar items fall into the same buckets. However, data sets are typically not distributed uniformly over the space, and as a result, the buckets of LSH are unbalanced, causing the performance of LSH to degrade. Data Sensitive Hashing (DSH) [10] aims to keep the buckets balanced with the help of a new hash family, while preserving the nearest neighbor relations.

Hashing techniques require the pre-computation of the hash tables without prior knowledge of the query. The accuracy of a similarity search is determined by the number and the quality of

	Raw Data		Bit-Sliced Index (BSI)				BSI SUM		
			Attrib 1		Attrib 2				
Tuple	Attrib 1	Attrib 2	$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$	$sum[2]$	$sum[1]$	$sum[0]$
t_1	1	3	0	1	1	1	1	0	0
t_2	2	1	1	0	0	1	0	1	1
t_3	1	1	0	1	0	1	0	1	0
t_4	3	3	1	1	1	1	1	1	0
t_5	2	2	1	0	1	0	1	0	0
t_6	3	1	1	1	0	1	1	0	0

Figure 1: Simple BSI example for a table with two attributes and three values per attribute.

the hash functions along with other tuning parameters. A higher number of hash functions can result in higher a probability of grouping similar objects together, however this can also result in a significant storage overhead. Moreover, with addition of new data, the hash index has to be re-computed. While QED uses some of the same concepts by projecting data into smaller ranges, it does not come with the storage overhead required by the hash index, and uses the query directly for data projection. Moreover, QED is an exact similarity function.

We compare QED against a distributed LSH [1] implementation and show that QED could present advantages in terms of smaller index size, and better accuracy for some applications.

3 DISTRIBUTED QED FOR HIGH DIMENSIONAL SIMILARITY SEARCHES

Consider a relation R and query vector Q. Every object in R is represented by m attributes or numeric scores. The query vector $Q = \{q_1, \ldots, q_m\}$ is also represented by m values. The task is to find the k most similar objects to Q in R. This k nearest neighbor (kNN) query can either compute the similarity between each data object and the query and retrieve the k objects with the highest score or, inversely, compute the distance between each object and the query and retrieve the k objects with the smallest distance.

In high dimensional spaces however, many distance functions such as Manhattan and Euclidean metrics are not as effective and the quality of the answer returned by the kNN query degrades. The reason these functions are directly affected by the dimensionality of the data lies in the fact that the dominant components are the dimensions for which two points are farthest apart. With higher dimensionality, the probability of having high discrepancies between two points in at least one dimension increases. The authors of [1, 3] show that for L_p-norm distance functions, the averaging effects of the different dimensions start predominating with increasing dimensionality. To prevent this, the authors of PiDist [1] show that imposing a proximity threshold for each dimension, beyond which the degree of dissimilarity is not relevant, could improve accuracy in nearest neighbor and similarity searches.

They achieve this by quantizing the indexed space into a fixed number of bins which are either equi-width or equi-depth (equi-populated). These quantizations are performed over the dataset without considering the query points. Even when a query point lies close to the boundary of a bin, only the points within the bin are considered for computing the similarity. In this section we describe a novel equi-depth quantization method that considers the query value for defining the bin boundaries and it is done on-the-fly during query execution.

[1] https://github.com/mrsqueeze/spark-hash

3.1 Bit-sliced indexing

In order to efficiently support QED over large datasets, we design a distributed indexing and efficient query algorithm. The proposed approach includes the usage of compressed bit-vector indexing and low level parallel bitwise operations. As shown previously [16, 18], this setup can leverage SIMD instructions and use the processing hardware more efficiently, for arithmetic operations.

Bit-sliced indexing (BSI) was introduced in [30], and it encodes the binary representation of attribute values with binary vectors. Therefore, $\lceil \log_2 values \rceil$ vectors, each with a number of bits equal to the number of records, are required to represent all the values for a given attribute.

Figure 1 illustrates how indexing of two attribute values and their sum is achieved using bit-wise operations. Since each attribute has three possible values, the number of bit-slices for each BSI is 2. For the sum of the two attributes, the maximum value is 6, and the number of bit-slices is $\lceil \log_2 6 \rceil = 3$. The first tuple t_1 has the value 1 for attribute 1, therefore only the bit-slice corresponding to the least significant bit, $B_1[0]$ is set. For attribute 2, since the value is 3, the bit is set in both BSIs. For example, the addition of the BSIs representing the two attributes is done using efficient bit-wise operations. First, the bit-slice $sum[0]$ is obtained by XORing $B_1[0]$ and $B_2[0]$: $sum[0] = B_1[0] \oplus B_2[0]$. Then $sum[1]$ is obtained in the following way: $sum[1] = B_1[1] \oplus B_2[1] \oplus (B_1[0] \wedge B_2[0])$. Finally $sum[2]$, which is the carry, is $majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$, where $majority(A, B, C) = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$.

BSI arithmetic for a number of operations, including the addition of two BSIs, is defined in [34]. Previous work [19, 33], uses BSIs to support preference and top k queries efficiently. BSI-based top k for high-dimensional data [16, 19] was shown to outperform current approaches for centralized and distributed query processing. In this work we adapt the distributed BSI query processing for NN queries. Furthermore, each individual bit-vector is compressed. The compression mechanism is described in section 3.6.

3.2 Query Dependent Equi-Depth Quantization

Further in this section we describe a novel function, called Query dependent Equi-Depth (QED) quantization, which can be used with traditional distance functions (e.g. Euclidean or Manhattan) to improve their accuracy in high-dimensional spaces. QED is implemented using a distributed bitmap-based index, and is designed with performance considerations in mind.

The main idea with localized functions is that for each dimension, if the data point has its respective dimension within threshold x then the distance to the query is considered for that dimension otherwise a dissimilarity penalty larger than x is assigned.

For this work, instead of directly specifying x, we consider parameter p as the minimum number of data points that should be contained within the query bin boundaries. Parameter p is

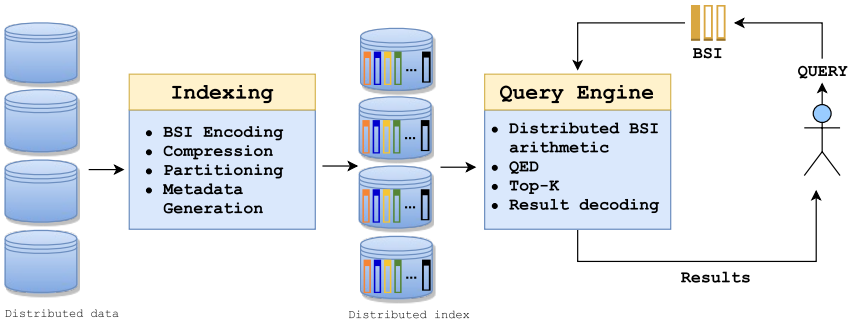

Figure 2: High level system overview

expressed as a percentage. Given the query value for dimension i, q_i, we find the $\lceil pn \rceil$ smallest distances to q_i and define a similar bin around the query point.

To illustrate the proposed dynamic quantization, consider a 1-dimensional dataset with values:

$$\{\{r1, 9\}, \{r2, 2\}, \{r3, 15\}, \{r4, 10\}, \{r5, 36\}, \{r6, 8\}, \{r7, 6\}, \{r8, 18\}\}$$

and query $\{q, 10\}$. If using Manhattan distance, the distance between the data points and the query are:

$$\{\{r1, 1\}, \{r2, 8\}, \{r3, 5\}, \{r4, 0\}, \{r5, 26\}, \{r6, 2\}, \{r7, 4\}, \{r8, 8\}\}$$

For QED, if parameter $p = 0.35$ (35% of the population), only the 3 points with the smallest distances, i.e. $\{r1, r4, \text{and } r6\}$, will be considered according to their distance. The rest of the points will be given a larger penalty δ_i to characterize a large dissimilarity. This normalization of the larger differences gives point $r5$ a chance to make it as a NN in the cases where there are other many dimensions for which $r5$ is really close to the query.

The value of the penalty, δ_i, can be assigned a constant larger than the distances computed within the query-dependent interval for each dimension. In the case of PiDist, the penalty assigned to dissimilar points is 1 and the distance for similar points is normalized to less than 1. Another approach could be to make δ_i to represent a number larger than the largest distance between the query and the closest p elements in dimension i.

Equation 1 shows the Manhattan distance between a data point a and the query q after applying the QED quantization.

$$\mathrm{QED}_{\mathrm{Manhattan}}(\mathrm{a}, \mathrm{q}) = \sum_{\mathrm{i}=1}^{\mathrm{d}} \begin{cases} \mid a_i - q_i \mid & \text{if } a \in P_i \\ \delta_i & \text{otherwise} \end{cases} \quad (1)$$

Where P_i is the subset of points closest to the query in dimension i, and δ_i is the penalty for the points outside the similar range in dimension i.

Performing QED for kNN queries without an index would slow the execution time, as it needs to dynamically compute a range for each dimension based on the query, in addition to computing the distance. We choose to implement QED on top of a bit-sliced index because BSI provides a compact representation of numeric values and an implicit ordering of the values as the set bits in the most significant bit-slice represent the largest values. In the next section we show how using the BSI index can in fact improve the performance of the kNN query.

3.3 Distributed Bit-Sliced Index (BSI)

Figure 2 depicts a high level overview of the proposed system. There are two main components: the indexing module and the query engine. The indexing module encodes each attribute into a bit-sliced index (BSI), compresses and partitions them, and generates the metadata required by the query engine to ensure correctness of the execution plan. The query engine encodes the user query into a BSI, runs the distributed kNN query, and returns the k nearest neighbors to the query.

We support both vertical (a subset of the attributes) and horizontal (a subset of the rows) partitioning. Distributed query algorithms are developed to minimize shuffling, improve load balancing, and maximize cluster utilization. For this work, we use Apache Spark and its Java API to distribute the workload across the cluster.

3.3.1 Indexing. Let us denote by B_i the bit-sliced index (BSI) over attribute i. $B_i[j]$ is a binary vector containing n bits (one for each tuple), where j represents the j^{th} bit in the binary representation of the attribute value. j can hold a value between 0 and the number of slices s used to represent values from 0 to $2^s - 1$. The bits are packed into words, and each binary vector encodes $\lceil n/w \rceil$ words, where w is the computer architecture word size (64 bits in our implementation).

For every attribute i in R we create a Bit-sliced index B_i. The BSI index is then partitioned and distributed across the nodes of a cluster. The *BSIAttr* class serves as a data structure for an atomic BSI element included in a partition. Each partition can include one or more *BSIAttr* objects. A *BSIAttr* object can represent all the attribute tuples (in the case of vertical-only partitioning) or only a subset (in the case of horizontal, or vertical and horizontal partitioning). Furthermore, a *BSIAttr* object can carry all of the attribute bit-slices or only a subset of them. The *BSIAttr* metadata generated includes information regarding the data type, encoding, number of slices, partition mapping. An example of attribute partitioning and use of metadata for partition mapping is shown in Figure 3.

We extended the BSI to handle signed numbers (both 2's complement and sign and magnitude) and represent decimal numbers using a fixed point format for each attribute. For every decimal BSI, the position of the decimal point is maintained as metadata for the attribute. To perform arithmetic operations between two attributes with different precision, namely a and b, where $a > b$, the decimal point for the second attribute is moved $(a - b)$ positions by multiplying the second attribute by the appropriate power of 10. Multiplication by a constant, as in this case, can be done efficiently by adding the logically shifted BSI to the original BSI for every set bit in the binary representation of the constant.

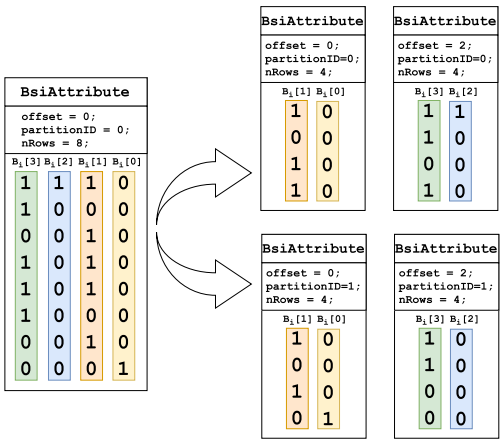

Figure 3: Example of vertical and horizontal partitioning of a BSI Attribute.

At query time, a BSI index is also generated for each partition using the attribute values in the query. Since the query value is constant, compressed bit-slices of all 0s or all 1s are used to generate a BSI with as many bits as objects in the partition in order support the bit-wise operations between the query and the *BSIAttr*. The hybrid query execution model [14] allows us to operate compressed and verbatim bit-vectors together and the results are dynamically compressed/decompressed as needed.

3.3.2 Query Engine. For developing the distributed algorithm we identified three main steps in the kNN query processing: compute the distance between the query Q and each *BSIAttr*, accumulate the partial distances for all dimensions, and retrieve the k closest points as the answer to the query. For an efficient execution of these three steps we are proposing a dynamic query aware quantization method called QED, and integrate it with algorithms for parallel execution of BSI arithmetic operations. In the following sections we describe in more detail the different components of the query engine.

3.4 Distributed kNN Query Processing

Most parts of the three-step process described earlier must be executed in parallel for achieving good scalability and performance. The computation of the absolute value of the difference between the query and each attribute BSI for *all* dimensions in parallel, aggregating all the distances into a single SUM_BSI also in parallel, and performing the top-k operation over the result BSI (can be executed in a single node or in parallel). We apply the same slice mapping distributed BSI aggregation developed for preferences queries [16]. This approach, described next, outperforms other parallel baseline implementations such as tree-reduction (adding pairs of BSIs together and using multiple reduce rounds) and its optimization Group Tree Reduction that reduces together groups of BSIs to reduce the number of rounds and the amount of data shuffled.

3.4.1 SUM_BSI Using Slice Mapping. It is true that the compact representation of the BSI makes the baseline implementations highly competitive versus their array counterparts. However, most of the performance gains, if not all, come from the reduced size of the BSI, and not necessarily because the algorithms are efficient. We propose an aggregation algorithm that promotes the bit-slices as the processing data units and applies the lessons-learned in computer arithmetic optimization to further improve the performance of the parallel aggregation. The basic idea of this approach lies in the use of the bit-slice depth as the mapped key and implement a two-phase aggregation algorithm, shown in Figure 4. In the first phase, the slices are added by bit-depth, producing a weighted partial sum BSI. In the second phase, all the partial sums are added together in a method similar to a carry-save adder.

Consider a dataset where $m = 128$ attributes are added using 10-nodes. Let us now assume that each attribute's value is within $1M = 2^{20}$, so every attribute i can be further partitioned into a set of 20 vertical bit-slices: $\{B_i[d] \mid 19 \geq d \geq 0\}$. In the proposed two-phase algorithm, the first task is to map all the bit-slices with the same depth (d) to a single node. Then addition is performed over 128 BSIs containing only 1 slice each, producing 20 partial sum BSIs. Each partial sum is in the range $[0, 128]$ and would require at most 8 slices. Next, these partial sums are added using their original depth d as their "weight." For example, the partial sum for the bit-slices of depth $d = 2$ would have a weight of $2^d = 4$. Because the weight is always a power of 2, this weighting scheme can be done efficiently by bit-shifting. Since the BSIs are stored column-wise, this shift can be represented using an offset and never materialized.

It is also possible to perform the parallel aggregation using groups of bit-slices to reduce data shuffling. In the previous example, with a group size of $g = 2$, we could have slices 0 and 1 from all 128 attributes added together in the same node during the first stage. This ability to group the slices and divide the attributes (*e.g.*, half of the depth 0 slices added in one node and the other half in another), allows us to balance the load and keep all the nodes busy longer.

For clarity in describing our algorithms, we use the example illustrated in Figure 4. In the first phase, every BSI attribute has its slices mapped locally to based on their depth d. The splitting of the BSI attribute in individual bit-slices allows for a finer granularity of the indexed data and for a more efficient parallelism during the aggregation phase. The pseudo-code of the mapping step is shown in the first Map() function of Algorithm 1. Every mapper has a *BSIAttr* (containing multiple slices) as input, and outputs a set of *BSIAttr*s that contain *one* bit-slice each. These bit slices are mapped by their depth in the input *BSIAttr*. Although there is an overhead associated with encapsulating each bit-slice into a *BSIAttr*, by creating a higher level of parallelism, we also achieve better load balancing and resource utilization.

Still in the first phase, the aggregation is done by the ReduceByKey() function of Algorithm 1. In this step, all the bit-slices with the same key (depth) are aggregated into a *BSIAttr*. Line 9 of Algorithm 1 performs the summation of two BSIs. We use the same addition logic as the authors in [35]. However, we achieve a parallelization of the BSI summation algorithm by splitting the *BSIAttr* into individual slices and executing their addition in parallel similarly to a carry-save adder. The offset of the resulting *BSIAttr*s are saved in the $offset$ field of each *BSIAttr* object to ensure the correctness of the final aggregated result. The summation is optimized by aggregating the bit-slices on the same node first, then on the same rack, and then across the network. Thus, trying to minimize the network throughput. The aggregation by depth is done locally first.

After aggregating partial local results, the second phase initiates to complete the aggregation by depth through shuffling the partial sums and reducing by their depths. The final step of the aggregation is done by reducing all the BSIs ($pSum$) produced in

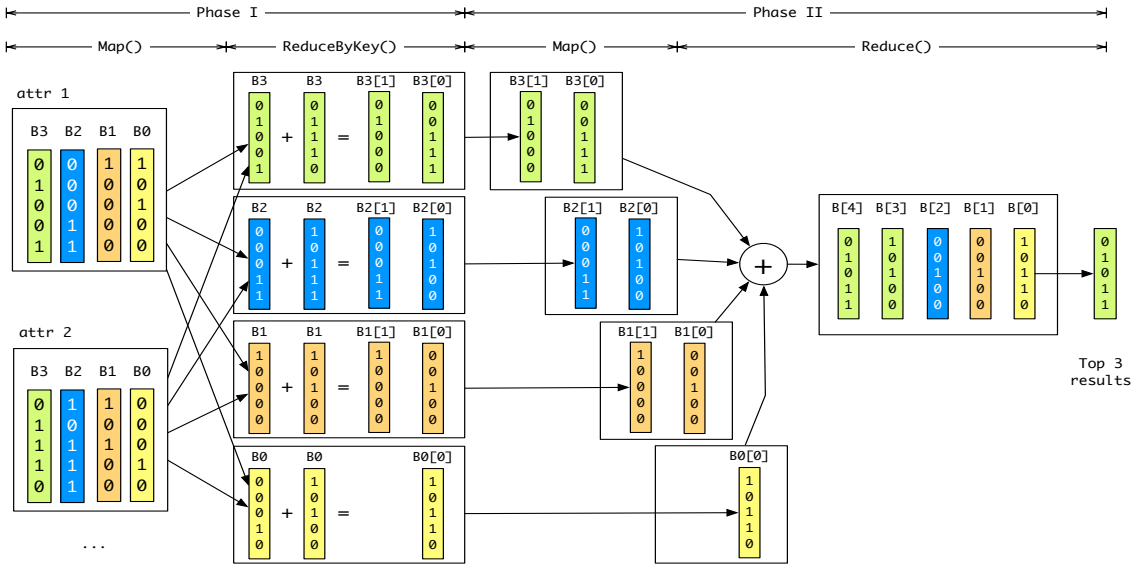

Figure 4: SUM_BSI Using Slice Mapping Example

the previous ReduceByKey() stage, regardless of their key. The final result (*attSum*) of this reduce phase is a single BSI attribute in the case of vertical only partitioning, or a set of BSI attributes, that should be concatenated, in the case of vertical and horizontal partitioning. Concatenation is straight forward, as each BSI in a partition has the same number of bits corresponding to the same rowIds.

3.4.2 Cost Estimations and Query Optimization. It is possible to estimate the complexity of the two-phase distributed aggregation and the expected amount of data shuffling. These estimations can help in choosing the most optimal partitioning strategies (group slicing) for the distributed aggregation, thus finding the best compromise between parallelism and the cost of network communication.

Note that the mapping in the first phase (Figure 4) does not produce any shuffling since it aggregates only the slices from attributes found on the same node. Data shuffling occurs twice in our two-phase aggregation. The first time is between the reducers of phase 1 and the mappers of the phase 2, and the second time data is shuffled between the mappers and reducers of the second phase. The amount of data shuffled depends on the number of nodes, partitions, tasks (or the number of attributes per task), and the number of slices per group. The number of slices per group can vary from 1 to s, where s is the highest number of slices per attribute in the dataset. In Figure 4 the slices are mapped into groups of one.

In order to determine the amount of data shuffled between the phase 1 and the phase 2, we should find first the number of outputs created by the reducers of phase 1. Given m attributes with s maximum slices per attribute, a attributes per node, and g slices per group, each node produces $\frac{s}{g}$ partial aggregations by depth. The size of each of these partial aggregations is in the worst case:

$$\lceil \log_2(g+a) \rceil \tag{2}$$

This represents the number of slices each partial aggregation by depth contains after the reduce phase 1. The total number of slices shuffled at this stage is:

$$Sh_1 = \left[\left(\min \left(\left\lceil \frac{s}{g} \right\rceil, \left\lceil \frac{m}{a} \right\rceil \right) - 1 \right] \cdot \left\lceil \frac{m}{a} \right\rceil \cdot \lceil \log_2(g+a) \rceil \right. \tag{3}$$

Algorithm 1: Two phase distributed BSI aggregation by slice depth

```
Map(): //Map slices by depth
begin
    Input: RDD<BSIAttr> indexAtt
    Output: RDD<Integer, BSIAttr> byDepth
1   int sliceDepth=0;
2   while indexAtt has more slices do
3       bsi = new BSIAttr();
4       bsi.add(indexAtt.nextSlice());
5       byDepth.add(new Tuple(sliceDepth, bsi));
6       sliceDepth++;
7   end
8   return byDepth
end
ReduceByKey()://Reduce by depth - first reduce phase
begin
    Input: RDD<Integer, BSIAttr> byDepth1, byDepth2
    Output: RDD<Integer, BSIAttr> pSum
9   pSum = byDepth1.SUM-BSI(byDepth2);
10  return pSum
end
Map():
begin
    Input: RDD<Integer, BSIAttr> partSum
    Output: RDD<BSIAttr> pSum
11  pSum = partSum._2();
12  return pSum
end
Reduce(): //Second reduce phase
begin
    Input: RDD<BSIAttr> pSum1, pSum2
    Output: RDD<BSIAttr> sumAtt
13  sumAtt = pSum1.SUM-BSI(pSum2);
14  return sumAtt
end
```

The mappers of the second phase produce $\frac{s}{g}$ outputs, each with the size:

$$\lceil \log_2(g+a) \rceil + \left\lceil \log_2(\frac{m}{a}) \right\rceil = \left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \quad (4)$$

The total number of slices shuffled between the mappers and reducers of the second phase is:

$$Sh_2 = \left(\left\lceil \frac{s}{g} \right\rceil - 1 \right) \left\lceil \log_2 \frac{(g+a)m}{a} \right\rceil \quad (5)$$

The total amount of data shuffled is the sum of the results from Equations 3 and 5:

$$Sh = Sh_1 + Sh_2. \quad (6)$$

The amount of data shuffled decreases as g - the number of slices per group increases, or as a - the number of attributes per node increases. However, less data shuffling means a higher load on individual tasks. We further analyze the time complexity for each individual task, and its impact on the total query time in the two-phase distributed aggregation.

The cost of summing two BSI attributes is linear on the number of slices and the number of rows in the attributes. If v is the number of slices of the attribute with a higher number of slices, then the cost of adding the two attributes is equal to the cost of executing v bitwise logical operations between two vectors. Given that the number of slices per group is a constant, g is the number of slices for each depth-shifted attribute in the reduce phase 1. Adding all the depth-shifted attributes within one node has the following complexity:

$$T_1 = \sum_{i=1}^{\log_2 a} (g+i). \quad (7)$$

There are $\frac{m}{a}$ partial sums with the same key per task, to complete the aggregation of partial sums shifted by depth. Thus the cost of this aggregation is:

$$T_2 = \sum_{i=1}^{\lceil \log_2 m/a \rceil} \left(g + \lceil \log_2 a \rceil + i \right) \quad (8)$$

Finally, the cost of aggregating the partial sums shifted by depth into one final attribute, is given by:

$$T_3 = \sum_{i=1}^{\lceil \log_2 s/g \rceil} \left(g + \lceil \log_2 a \rceil + \left\lceil \log_2 \frac{m}{a} \right\rceil + i \right) \quad (9)$$

When taking into consideration the time complexities from Equations 7, 8, and 9, one must account for the different number of tasks executed in these three steps. For example, if T_1 has a weight of one, i.e. $W_{T_1} = 1$, then the number of tasks for T_2 and T_3 is different. For $W_{T_1} = 1$, the weight for T_2 is:

$$W_{T_2} = \frac{1}{\lceil \frac{m}{a} \rceil} \quad (10)$$

since there are fewer tasks for T_2 than T_1 by a factor of $\frac{m}{a}$. While the weight for T_3 is:

$$W_{T_3} = \frac{1}{\lceil \frac{m}{a} \rceil \lceil \frac{s}{g} \rceil} \quad (11)$$

In this case, there are s/g fewer tasks than in the previous step.

Using the time complexities discussed above, together with the data shuffle estimations, it is possible to find the optimum values for the number of slices per group (g) and the number of initial tasks/attributes per task.

3.5 QED over the distributed BSI

QED can be done gracefully with the BSI index without imposing any overhead when compared to the computation of the Manhattan distance without indexing, as shown in Algorithm 2. We operate on top of a BSI index representing the distance between the query and the data points in each dimension. Thus we define the penalty δ_i as the truncation of the most significant bits for the largest distances as depicted in Figure 5.

Algorithm 2: QED Quantization

```
Input: BSI A, int p
Output: BSI S
BitSlice penalty = (A[A.size − 2] XOR A.sign);
for (i = A.size − 2; i >= 0; i − −) do
    penalty = (penalty OR (A[sSize] XOR A.sign));
    if penalty.count() >= n − p then
        sSize = i;
        break;
    end
end
BSI S = new BSI(sSize);
for (i = 0; i < sSize; i++) do
    S[i] = (A[i] XOR A.sign);
end
S.addSlice(penalty);
return S
```

QED can be included in the calculation of the absolute value of the distance between query and each dimension as shown in Algorithm 2. In datasets with large attribute ranges, the output of Algorithm 2 is significantly smaller in size than the size of the actual distance measures, for most distributions. This is very important because the result of this operation is further processed to aggregate and rank similar objects. As a result of reducing significantly the output size of this step, the overall execution time of the kNN query is generally improved. The number of bit-vectors required to encode a difference attribute is equal to the number of bits required to encode the difference range. Where the difference range is the maximum difference between the query dimension and the same dimension of any of the $\lceil pn \rceil$ tuples, and their minimum difference.

In large datasets where the number of tuples is high, p should typically be small, and most of those p closest tuples are much closer than the attribute range. Thus the reduction in size of the result of Algorithm 2. A more detailed discussion on parameter p follows in section 3.5.1.

For a better understanding of Algorithm 2, we show how the distance BSI attribute between the query and the data points used in our previous running example is quantized using QED in Figure 5. For simplicity, all the distances are positive in Figure 5 (leftmost BsiAttribute). Starting from the most significant bit-slice, the bit-slices in the distance attribute are OR-ed until the count of set-bits in the resulting bit-slice (the penalty bit-slice) is equal or greater than $(n-p)$. At this point the bit-slices that were operated are dropped and replaced with one single penalty bit-slice.

The effect of this quantization is the identification of the furthest $(n-p)$ points from the query for one given dimension, and reducing their distance, while keeping an accurate distance for the close points. Hence avoiding over penalizing a point if only a few dimensions are far from the query.

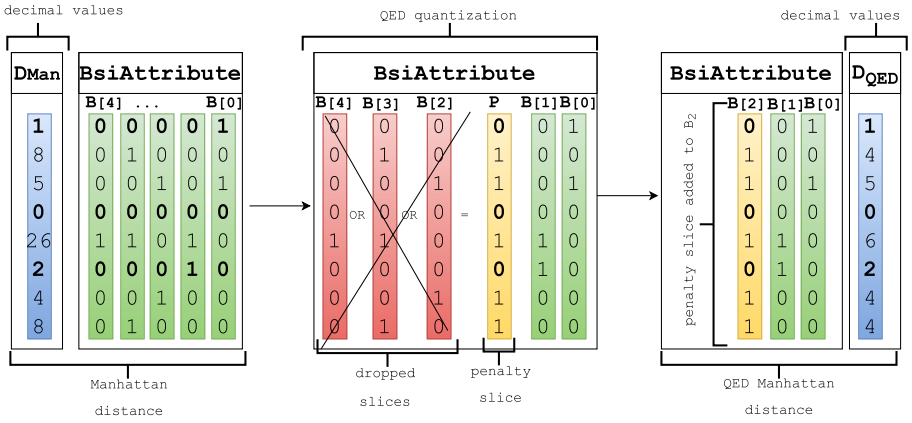

Figure 5: Query dependent Equi-Depth(QED) quantization with population range $p = 35\%$

Figure 5 and Algorithm 2 use Manhattan distance along with QED quantization. However it is also possible to use other distance metrics such as Euclidean or Hamming.

Equation 12 shows the Hamming distance between a data point a and the query q after applying the QED quantization.

$$\mathrm{QED}_{\mathrm{Hamming}}(a, q) = \sum_{i=1}^{d} \begin{cases} 0 & \text{if } a \in P_i \\ 1 & \text{otherwise} \end{cases} \tag{12}$$

Where P_i is the subset of points closest to the query in dimension i.

3.5.1 Estimating parameter p. The main idea of QED is to use the query itself for determining a range in which the points falling within are considered similar. Determining this range needs to consider the data values in the attribute. Thus we want to define a bin for each dimension where the percentage of points p in each bin is roughly the same for each dimension. We define p as a fraction of the total number of rows n in the dataset.

The value of p is directly influenced by the data dimensionality and the total number of rows in the dataset. Intuitively, for large datasets with a large number of tuples, p should be small, as even a small p would represent a large number of candidate points. Conversely, as the number of dimensions increases, p should also increase to prevent all the tuples from being penalized in many dimensions.

Inspired by the Pareto principle [31], where only the vital few produce the majority of results, we define the power function given in Equation 13 as a heuristic to estimate p:

$$\hat{p} = \left(\frac{m}{m+n}\right)^{\frac{1}{\lg(n)}} \tag{13}$$

For this power distribution, we use the number of attributes, m, as the scale, and the number of tuples, n, to derive the shape. We made the power function $\frac{m}{m+n}$ to guarantee a number less than 1.

Figure 6 shows the estimated values of p for four datasets with 1M, 10M, 100M, and 1B tuples as the number of attributes increases. We empirically evaluated this estimations over two large datasets and observed that the estimations for p were at or near the point with maximum accuracy for the task of kNN classification.

3.6 Bitmap Compression

For further optimization, in our setup, we apply compression to each individual bit-vector, when suitable [14]. Most types of

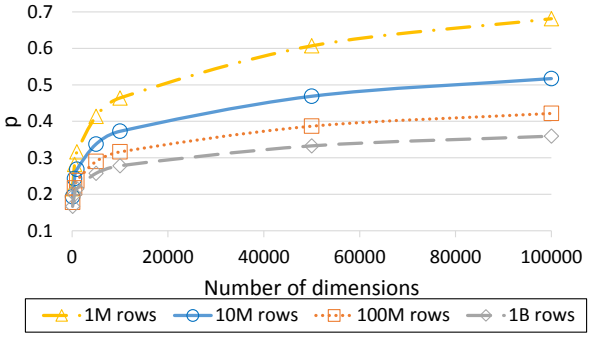

Figure 6: Estimated values of parameter p for maximizing accuracy of the kNN query results with QED

bitmap (bit-vector) compression schemes use specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression.

Word-Aligned Hybrid Code (WAH) [40] proposes the use of words to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH divides the bitmap into groups of length $w - 1$, where w is the CPU's word size. WAH then collapse consecutive all-zeros or all-ones groups into a fill word.

Recently, several bitmap compression techniques that improve on WAH by making better use of the fill word bits have been proposed in the literature [27, 41], and others. Previous work have also used varying segment lengths s within $s \leq w$ encoding [17].

In this work we use our recently proposed bit-vector compression scheme [14], which is a hybrid between the verbatim scheme and the EWAH/WBC [27] bitmap compression. This hybrid scheme compresses the bit-vectors if the bit density is below a user-set threshold. Otherwise the bit-vectors are left verbatim. In our experiments we begin with compressed bit-vectors if the compressed size for the bit-vector is 0.5 or smaller than the size of the uncompressed bit-vector. The query optimizer described in [14] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries. We choose this compression scheme due to its capability of operating with denser bitmaps, which is the case for the bit-vectors inside the bit-sliced index, and it allows for uncompressed bit-vectors to be operated with compressed ones. Nonetheless, it is possible to

apply other compression models, such as the one proposed in [6]. The compression model is orthogonal to the contributions of this work.

4 EXPERIMENTAL EVALUATION

In this section we evaluate the proposed indexing, quantization, and distributed kNN querying algorithms in terms of classification accuracy and query performance. When evaluating the query speed of the kNN query with QED quantization, we set $p = \hat{p}$ as described in Equation 13. In our evaluations we use two distance metrics with QED: QED with Manhattan distance (QED-M), and QED with Hamming distance (QED-H).

4.1 Experimental Setup

We implemented the proposed index and query algorithms in Java, and used the Java API provided by Apache Spark to run our algorithms on an in-house Spark/Hadoop cluster. The Java version installed on the cluster nodes was 1.7.0_79, Spark version 1.6.1. and Hadoop version 2.4.0.

Our Hadoop stack installation is built on the following hardware: There is one Namenode (master) server (Two 6-core Intel Xeon E5-2420v2, 2.2GHz, 15MB Cache; 48 GB RAM at 1333 MT/s Max). The cluster also contains four Datanode (slave) servers (two 6-core Intel Xeon E5-2620v2, 2.1 GHz,15MB Cache; 64 GB RAM at 1333 MT/s Max). As cluster resource manager we used Apache Yarn. The namenode and datanodes are connected to each other over 1 Gbps Ethernet links over a dedicated switch. Unless otherwise noted, we use all the available hardware resources in this cluster for running the experiments.

In our experiments we used a number of real datasets to evaluate the proposed indexing and querying. We used nine datasets from the UCI repository [4] for accuracy evaluation, and two larger datasets (`HIGGS`[2], and `Skin Data` [32]) were used on the Spark/Hadoop cluster for performance measurements. The number of dimensions in the datasets range from 19 to 279 and the number of classes from 2 to 24. The Skin-Images dataset contains integer numbers (image pixel values), while the other datasets contain real numbers. The details of the characteristics of the data and the class distribution can be found in Table 1.

Dataset	Rows	Cols	Classes
anneal	798	38	5
arrhythmia	452	279	13
dermatology	366	33	6
higgs	11M	28	2
horse-colic	300	26	2
ionosphere	351	33	2
musk	476	165	2
segmentation	210	19	7
skin-images	35M	243	2
soybean-large	307	34	19
wdbc	569	30	2

Table 1: Description of the characteristics of the real datasets used in the experiments.

4.2 QED Classification Accuracy

Classification accuracy is computed over the labeled data using the leave-one-out methodology as the number of correct classifications divided by the total number of tuples in the data set. Voting was used to decide the class for each data point. Table 2 shows the best classification accuracy when using kNN classification

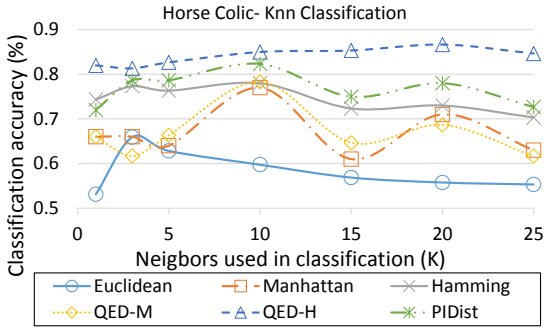

Figure 7: kNN Classification accuracy as the number of nearest neighbors (k) increases for Horse Colic dataset. (Dataset: HourseColic, 300 rows, 26 attributes, 2 classes (99/201))

for each method. We vary the number of nearest neighbors used in classification $k = \{1, 3, 5, 10\}$, and report the best result for each distance function. For quantization, we apply Equi-width and Equi-Poplulated partitioning varying the number of bins/clusters from 3 to 20 $\{3, 5, 7, 10, 15, 20\}$. The same number of bins/clusters was used for all the dimensions. The only case where attributes could be quantized using a different number of bins/clusters than the one provided as a parameter was the categorical attributes with less categories than the number of bins/clusters provided. In that case each value was considered as a bin/cluster. For dynamic quantization we set p as a percentage of the number of rows. We vary $p = \{60\%, 50\%, 40\%, 30\%, 25\%, 20\%, 10\%, 5\%, \text{and } 1\%\}$.

For each function metric we report the best result, and then the best accuracy for each dataset is highlighted in bold in Table 2. As shown in the table, QED is able to improve the results for Manhattan and Hamming in most datasets. QED using Manhattan is consistently better than Manhattan with no quantization (8/9) with up to 7.35% accuracy increase (2.4% on average). For Hamming distance, QED quantization outperformed no-quantization in 7/9 cases with up to 57.7% accuracy improvements (10.95% on average).

4.2.1 Evaluation of Parameter k. When using nearest neighbor searches for classification purposes, the number of neighbors considered is often crucial for the accuracy of the classifier. In this experiment we evaluate the effect of k (the number of nearest neighbors) when QED is used in k-Nearest Neighbor (kNN) classification.

Figures 7 and 8 show the classification accuracy for several distance functions as the number of neighbors k increases for two different datasets. In figure 7 for the Horse-colic dataset, the classification accuracy increases gradually for QED (QED-M with Manhattan distance, and QED-H with Hamming distance), while the other distance functions are more sensitive to the value of k. Regardless of the value picked for k, QED-H has the highest accuracy among the measured distance functions for kNN classification for this dataset.

As Figure 8 shows for the Arrhythmia dataset, QED-M (with Manhattan distance) has the highest accuracy. It is worth noting that while the accuracy performance for other distance functions decreases as k increases, classification accuracy for QED is not significantly affected.

4.2.2 Evaluation of Parameter p. The p parameter determines the number of tuples for each dimension to be considered similar for distance computation, while the other tuples get a dissimilarity penalty. Expressed as a percentage, p falls within the

Dataset	Euclidean	Manhattan	QED-M	Hamming			QED-H	PiDist/iGrid	
				NQ	EW	ED		EW	ED
anneal	.934	.939	.964	.986	.984	.980	**.994**	.990	.990
arrhytmia	.659	.653	**.701**	.602	.686	.646	.650	.695	.635
dermatology	.975	.978	**.986**	.975	.973	.883	.921	.981	.970
horse-colic	.740	.770	.783	.780	.827	.857	**.867**	.833	.843
ionosphere	.866	.909	**.943**	.809	.926	.860	.920	.929	.903
musk	.882	.893	**.916**	.819	.876	.870	.878	.868	.887
segmentation	.843	.886	.881	.586	.871	.857	**.924**	.900	.876
soybean	.873	.899	**.938**	.909	.912	.902	.821	.909	.922
wdbc	.940	.949	.949	.692	**.967**	.951	**.967**	.961	.960

Table 2: Leave-one-out best classification accuracy using k-nearest neighbor ($k \in \{1, 3, 5, 10\}$) classification with different distance functions and quantization methods (NQ=No Quantization, EW=Equi-witdth, ED=Equi-depth, QED=Query-dependent Equi-depth). The best result for each dataset is highlighted in bold.

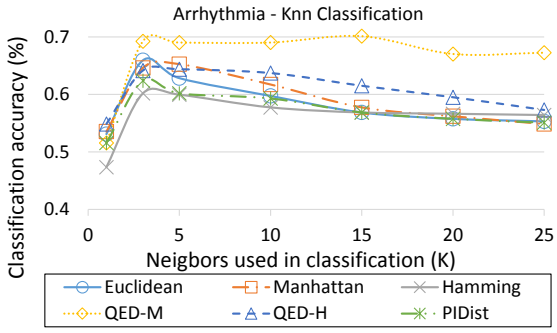

Figure 8: kNN Classification accuracy as the number of nearest neighbors (k) increases for Arrhythmia dataset. (Dataset: Arrhythmia, 452 rows, 279 attributes, 13 classes)

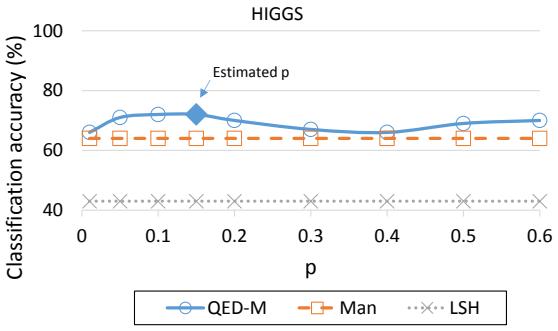

Figure 9: The impact of the p parameter on kNN classification accuracy (Dataset: HIGGS, 11M rows, 28 attributes, 2 classes)

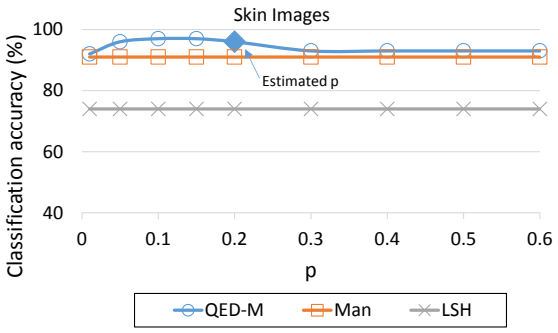

Figure 10: The impact of the p parameter on kNN classification accuracy (Dataset: Skin-Images, 35M rows, 243 attributes, 2 classes)

interval $(0, 1]$. If $p = 1$ then the results of the kNN query are the same for both QED-M and Manhattan distance. Clearly, the value of p affects the accuracy of the kNN query results.

In this experiment we vary the value of p from 0.01 to 0.6 and measure the kNN classification accuracy. We chose the two largest of the datasets: HIGGS and Skin-Images, as a higher number of data objects results in a more robust evaluation of p. The accuracy is reported after running 1000 queries obtained by random sampling. We compare the kNN classification accuracy results of QED against sequential scan Manhattan distance and a distributed implementation of Locality Sensitive Hashing (LSH).

The LSH implementation and parameter choice was largely based on the description in chapter 3 of [28]. The LSH number of bins was set to 10000, number of hash functions: 25, and the number of hash tables: 4. For all three methods 5 nearest neighbors were considered for classification.

Figures 9 and 10 show the kNN classification accuracy results as the value of parameter p varies. The filled marker is the p value computed using Equation 13 from Section 3.5.1, and in both cases it is at, or near the highest accuracy point.

4.3 Index size

Given the rapid advancements in data collection, not only the data becomes more complex and harder to analyze, but also its size requires more computational resources. As described earlier, we make use of the BSI index to represent data in a more compact form. A smaller index size should enable performance gains through less network shuffling, fewer CPU cycles required for processing, and less memory utilization and I/O.

An important advantage of the BSI index is that it has a compact size. The compression comes not only from using a lower number of bit-slices per attribute than the number of bits used in a Long or Double data type, but also from compressing each individual bit-slice (where beneficial) using a hybrid bitmap compression scheme [14]. The compression of the bit-slices occurs only if it can improve the query performance.

Figure 11 shows the size of the BSI index in comparison with the size of the raw data, the LSH index, and the PiDist (10 and 20) index for the HIGGS and Skin Images datasets. Five LSH hash tables were generated using 25 hash functions and 10,000 bins. PiDist-10 refers to the PiDist index with the bin size of 10, while

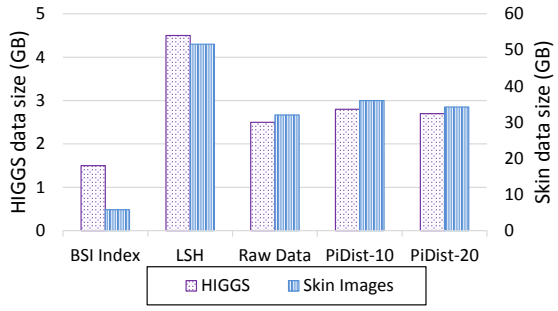

Figure 11: Index sizes for the HIGGS and Skin-Images datasets.

PiDist-20 has the bin size of 20. These are some of the bin sizes that where shown to perform well in [1].

The Skin Images dataset has a higher compression ratio than HIGGS when compared to the raw data size. This is mostly due to the low cardinality of this dataset, which has RGB encoded pixel values. The HIGGS dataset requires approximately 60 slices per attribute to encode its values, while the Skin Images dataset only requires 8 bit-slices per attribute (values from 0 to 255).

Because the BSI index does not require accesses to the raw data, and due to its small size, it is possible to fit more information into memory and less network communication is required when the kNN queries are performed in a distributed setting.

4.4 Performance impact of data cardinality

Given that the BSI index is sensitive to data cardinality, we set up to measure the scalability of the QED quantization method when compared to running NN searches over the BSI index without QED quantization. We use the HIGGS dataset for this experiment as its data has high cardinality. We vary the number of bit-slices per attribute for indexing from 15 to 60. Note that while it is possible to encode any attribute with any number of slices, using less than $\lceil log_2 c_i \rceil$ slices, where c_i is the attribute cardinality, results in a lossy compression where the values are approximated to some degree. This approximation however, could have little effect on the kNN classification accuracy depending on the dataset. The evaluation of the BSI approximation is left as a subject for future work.

The query time is reported in milliseconds per query, and was obtained by averaging the kNN classification query times over 1000 queries. As Figure 12 shows, with the increase in cardinality, the query speed degrades at a much slower pace for QED-M than BSI Manhattan (without QED quantization). As mentioned in the previous section when describing Algorithm 2, the performance improvements is largely due to a smaller output, independent of the attribute cardinality, and consequently less data shuffling and processing in the aggregation phase. Running the same queries with Manhattan distance without any indexing took approximately two seconds. Thus, the kNN query time using BSIs was two to five times faster than sequential scan, while QED-M achieved an improvement in performance of one order of magnitude.

4.5 QED query time performance

Many of the existing indexing techniques fail to run faster than sequential scan when tested against high dimensional data. Thus, the approximate nearest neighbor searches became a solution that improves on query time performance by trading off accuracy.

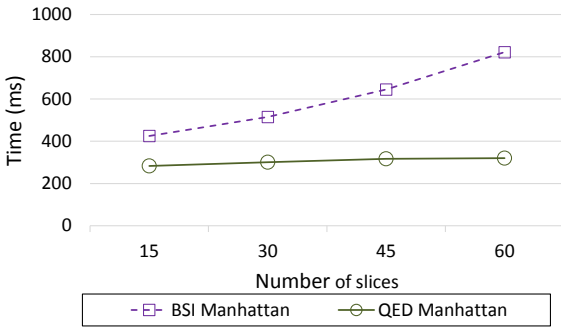

Figure 12: BSI Manhattan and QED Manhattan kNN query performance when increasing data cardinality (Datasets: HIGGS)

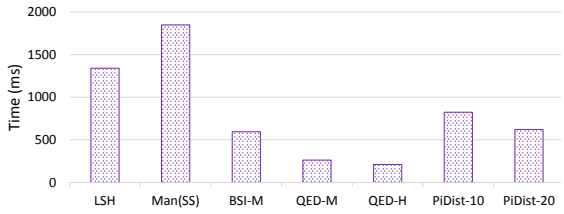

Figure 13: kNN query performance comparison. (Dataset: HIGGS, 11M rows, 28 attributes, 2 classes)

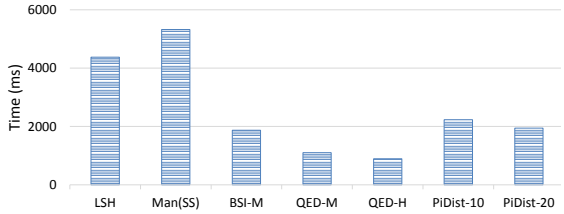

Figure 14: kNN query performance comparison. (Dataset: Skin-Images, 35M rows, 243 attributes, 2 classes)

We ran a total of 1000 kNN classification queries over the HIGGS and Skin Images datasets, with the configuration for LSH and PiDist described earlier. Figures 13 and 14 show the average query time per query in milliseconds. Because the number of nearest neighbors k in kNN classification applications is generally low, we set k in our query speed evaluations to 5 and do not vary it. Increasing k, however, doesn't impact the query performance in any significant way because the scores are computed for all the points in the dataset regardless of k. The best query times were achieved when using the QED quantization over the BSI index. The average query time for QED-M was only 14% of the average Sequential Scan query time for the HIGGS dataset. For the Skin Images data set the QED Manhattan query time was 20% of Sequential Scan query time.

5 CONCLUSION

In this work we described the indexing structure and the methods for on-the-fly Query dependent Equi-Depth (QED) quantization to improve high-dimensional similarity. The quantization is done for each dimension at query time and localized scores are generated for the closest p% of the points. A constant penalty is applied for the rest of the points. By normalizing the penalty for values outside the similarity range we are able to improve nearest neighbor searches in high dimensional spaces.

We evaluated the kNN classification accuracy of the proposed QED quantization on a set of nine high-dimensional datasets and observed an average improvement in accuracy of 2.4% for Manhattan distance and 10.95% for Hamming distance when using QED quantization.

The index structure and the query algorithms that support the kNN searches were designed with distributed processing in mind. We implemented several BSI arithmetic operations such as: addition with a constant, absolute value, and various transformation of the BSI attribute. The index can be partitioned vertically as well as horizontally and makes for a fine level of task granularity and load balancing. Because each dimension is indexed independently, this approach is also scalable for high dimensional data. We evaluated the scalability for datasets up to 243 dimensions on a Spark/Hadoop cluster. We also show that when using QED quantization, the kNN query performance is very robust and does not decrease significantly with the increase in data cardinality.

Due to a smaller index size, the ability to partition the index vertically and horizontally, and the fast bitwise operations, the BSI index proves to be a good data structure for performing distributed Nearest Neighbor searches with query aware quantization at run time. With QED quantization, in our performance evaluation we observe an improvement in query time of up to one order of magnitude when compared to Sequential Scan on high dimensional data.

As future work we plan to investigate further the penalty applied for dissimilar dimensions and under what conditions the normalization of the penalty or the distance would improve the accuracy of nearest neighbor searches. More work is also required in expanding the distance metrics for which the QED quantization can be applied.

6 ACKNOWLEDGEMENTS

This work was partially supported by the NIH National Cancer Institute/Big Data to Knowledge (BD2K) Program under grant R01CA214825 and joint NSF/NIH Initiative on Quantitative Approaches to Biomedical Big Data (QuBDD) (R01) grants NSF DMS-1557578 and NIH R01CA225190.

REFERENCES

[1] C. C. Aggarwal and P. S. Yu. The igrid index: reversing the dimensionality curse for similarity indexing in high dimensional space. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 119–129. ACM, 2000.

[2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Commun*, 5, 2014.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[4] C. Blake and C. Merz. Uci repository of machine learning databases [http://www. ics. uci. edu/ mlearn/mlrepository. html], department of information and computer science. *University of California, Irvine, CA*, 1998.

[5] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[6] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *arXiv preprint arXiv:1402.6407*, 2014.

[7] D. L. Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, 1:32, 2000.

[8] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[9] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[10] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1127–1138. ACM, 2014.

[11] S. García, J. Luengo, J. A. Sáez, V. López, and F. Herrera. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Trans. Knowl. Data Eng.*, 25(4):734–750, 2013.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.

[13] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[14] G. Guzun and G. Canahuate. Hybrid query optimization for hard-to-compress bit-vectors. *The VLDB Journal*, pages 1–16, 2015.

[15] G. Guzun and G. Canahuate. Supporting dynamic quantization for high-dimensional data analytics. In *Proceedings of the ExploreDB'17*, ExploreDB'17, pages 6:1–6:6, New York, NY, USA, 2017. ACM.

[16] G. Guzun, G. Canahuate, and D. Chiu. A two-phase mapreduce algorithm for scalable preference queries over high-dimensional data. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS, 2016.

[17] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 484–495. IEEE, 2014.

[18] G. Guzun, J. C. McClurg, G. Canahuate, and R. Mudumbai. Power efficient big data analytics algorithms through low-level operations. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 355–361. IEEE, 2016.

[19] G. Guzun, J. Tosado, and G. Canahuate. Slicing the dimensionality: Top-k query processing for high-dimensional spaces. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIV*, pages 26–50. Springer, 2014.

[20] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[22] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[23] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

[24] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Record*, volume 26, pages 369–380. ACM, 1997.

[25] R. Kerber. Chimerge: Discretization of numeric attributes. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 123–128, 1992.

[26] L. A. Kurgan and K. J. Cios. Caim discretization algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):145–153, 2004.

[27] D. Lemire, O. Kaser, and E. Gutarra. Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems*, 37(3):20:1–20:29, 2012.

[28] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[29] F. Liu and H. J. Lee. Use of social network information to enhance collaborative filtering performance. *Expert systems with applications*, 37(7):4772–4778, 2010.

[30] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM Press, 1997.

[31] V. Pareto. Manual of political economy, 1906.

[32] S. L. Phung, A. Bouzerdoum, and D. Chai. Skin segmentation using color pixel classification: analysis and comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(1):148–154, Jan 2005.

[33] D. Rinfret. Answering preference queries with bit-sliced index arithmetic. In *Proceedings of the 2008 C 3 S 2 E conference*, pages 173–185. ACM, 2008.

[34] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001.

[35] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. *SIGMOD Rec.*, 30(2):47–57, 2001.

[36] H. Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.

[37] K. shy Goh, B. Li, and E. Chang. Dyndex: A dynamic and non-metric space indexer. In *IN ACM MULTIMEDIA*, pages 466–475, 2002.

[38] C. Tsai, C. Lee, and W. Yang. A discretization algorithm based on class-attribute contingency coefficient. *Inf. Sci.*, 178(3):714–731, 2008.

[39] A. K. H. Tung, R. Zhang, N. Koudas, and B. C. Ooi. Similarity search: a matching based approach. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 631–642. VLDB Endowment, 2006.

[40] K. Wu, E. J. Otoo, and A.Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of the 2002 International Conference on Scientific and Statistical Database Management Conference (SSDBM'02)*, pages 99–108, 2002.

[41] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, 2001.