Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2018,6
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Exploiting subspace distance equalities in Highdimensional data for knn queries

Martin Schäler, David Broneske, Veit Köppen,
and Gunter Saake

2018

KIT – University of the State of Baden-Wuerttemberg and National
Research Center of the Helmholtz Association

Exploiting sub-space distance equalities in high-dimensional data for knn queries

Martin Schäler[1], David Broneske[2], Veit Köppen[2], and Gunter Saake[2]
[1]Karlsruhe Institute of Technology, [2]OVGU Magdeburg
[1]martin.schaeler@kit.edu, [2]firstname.lastname@ovgu.de

ABSTRACT

Efficient k-nearest neighbor computation for high-dimensional data is an important, yet challenging task. The response times of state-of-the-art indexing approaches highly depend on factors like distribution of the data. For clustered data, such approaches are several factors faster than a sequential scan. However, if various dimensions contain uniform or Gaussian data they tend to be clearly outperformed by a simple sequential scan. Hence, we require for an approach generally delivering good response times, independent of the data distribution. As solution, we propose to exploit a novel concept to efficiently compute nearest neighbors. We name it sub-space distance equality, which aims at reducing the number of distance computations independent of the data distribution. We integrate knn computing algorithms into the Elf index structure allowing to study the sub-space distance equality concept in isolation and in combination with a main-memory optimized storage layout. In a large comparative study with twelve data sets, our results indicate that indexes based on sub-space distance equalities compute the least amount of distances. For clustered data, our Elf knn algorithm delivers at least a performance increase of factor two up to an increase of two magnitudes without losing the performance gain compared to sequential scans for uniform or Gaussian data.

1. INTRODUCTION

In data analysis, efficient computation of the k-nearest neighbors (knn) for high-dimensional data sets has long been an important, yet challenging problem [6, 11]. With emerging applications, like scientific databases or time series analytics, this importance is further increasing. The reason is that solving the knn problem is part of various data analysis methods, including classification and clustering.

To allow for efficient knn computation, a wide range of indexing techniques is known. Early approaches, such as R-Trees [15] or kd-Trees [3] are known to deliver good performance only in low-dimensional spaces [20]. State-of-the-art approaches [8, 16], such as iDistance [17], map each point in the data set to its nearest pivot(s). Comparative studies [8, 17] report large performance increases. However, usually those approaches are difficult to tune as they feature various parameters and are known to be parameter sensitive [23, 27]. More importantly, distances between high-dimensional points tend to be very similar – particularly if the number of dimensions increases. Therefore, the mapping of points to artificial clusters does not solve the fundamental problem of high-dimensional data sets: often a large fraction of the data set is visited upon knn computation.

With the increasing amount of main-memory, a major cost driver for sequential scans, fetching *all* points from hard disk, has vanished. This makes sequential scans as competitor even more powerful. Nevertheless, advances in hardware alone do not allow for efficient analyzes of the fast growing amount of data. Hence, we require for an approach that is expected to result in good response times for knn computation regarding various distributions instead of reaching peak performance for some data sets (cf. Figure 1).

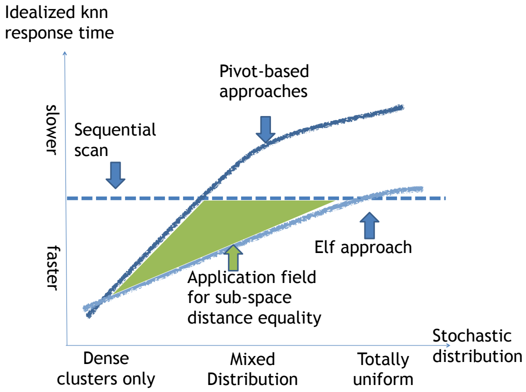


Figure 1: Intuition of the application field of indexes based on sub.space distance equalities

The objective of this paper is to investigate how to design an index that generally results in good performance. That is, for highly clustered data we want to achieve, on average, at least comparable performance to state-of-the-art indexes. In case the data set contains uniform or Gaussian data, the average performance shall be at least comparable to a sequential scan. We explicitly do not target at outperforming any known approach for any possibly existing data set. We deem such endeavor unreasonable considering decades of knn index research and the variety of parameters to consider. Examples for such parameters are different dimensionality, number of points, and stochastic distribution of the data sets as well as different knn distance functions, to name only a few.

A novel concept that gives way to approaches generally resulting in good response times is *sub-space distance equality*. It is based on the observation that all points sharing the same prefix (combination of values) have the same distance to any query w.r.t. the sub space defined by the prefix. Hence, if this concept is well exploited, the number of required distance computations is highly reduced. That is, distance computation is executed only once per sub space (not once per point) and the sub-space distance represents a tight lower bound for any point within that sub space. To our knowledge, the only index that allows exploiting sub-space distance equalities is Elf [7]. Moreover, this index features a main-memory optimized storage layout, which allows to investigate the effect of sub-space distance equality on knn computation in isolation as well as in combination with an optimized storage layout. Therefore, we select Elf for our investigations.

However, so far, Elf is only known to deliver high speedups for multi-column selection predicates (i.e., multi-dimensional range queries) [7]. This query type is fundamentally different to knn queries. Hence, we make non-trivial extensions in order to open this new application field for indexes based on the concept of sub-space distance equality. Altogether, our investigations result in the following contributions:

1. We develop algorithms exploiting sub-space distance equalities for knn computation and integrate them into the Elf index. Thereby, we exploit the index' genuine features at concept level as well as the level of optimized storage layout.
2. We comprehensively examine factors affecting the performance of our knn algorithms allowing to build optimized Elf indexes for arbitrary combinations of data set and distance function.
3. We comprehensively investigate the performance of our algorithms. To this end, we consider twelve data sets with different properties, such as data size and stochastic distribution, revealing that our algorithms result in competitive performance for all data sets considering highly potent competitors. The reason is that exploiting sub-space distance equalities results in the least amount of distance computations for all data sets.
4. We reveal that the observed speedups hold for a well-known instance of the Minkowski metric family.

Finally, to support repeatability, all data sets and implementations are available open source[1].

2. PRELIMINARIES AND RELATED WORK

In this section, we introduce the knn problem including properties of distance functions. Furthermore, we introduce related work.

2.1 The knn problem

Given a set of points in a d-dimensional space ($\mathbb{N}^d$), a knn query returns the k closest points to the query point according to some distance function. Formally, a query `knn(q,dist(),D,k)` has the following input: a query point $q \in \mathbb{N}^d$, a distance function `dist()`, a set of points D all being $\in \mathbb{N}^d$, and some $k \in \mathbb{N}$ with $k > 0$. It returns a set S containing k points from D, where `max_dist` is the largest distance of any point $p \in S$. The following holds for any point p' in $D - S$: `max_dist` $\leq$ `dist(p',q)`.

We map all dimensions in the d-dimensional space ($\mathbb{N}^d$) to integer numbers in interval of [0,c], where c is a user-defined granularity, instead of the unit space $\mathbb{R}^d$ [0,1]. The reason is that many `dist()` functions, such as the Manhattan metric, can be computed using integer arithmetic known to be generally faster than float arithmetic. In addition, this definition of the knn problem implies that one does not necessarily has to use `dist()`. There may be some function `dist'()` that is order preserving according to `dist()`, but easier to compute. A common example is using the squared Euclidean distance to avoid computing square roots.

$$\texttt{dist(X,Q)} = \left(\sum_{i=1}^{d} |x_i - q_i|^p\right)^{\frac{1}{p}} \text{ with } p \geq 1 \qquad (1)$$

Metrics as distance functions. Different distance functions support different intuitions of distance. Metrics are a group of distance functions that are well studied and used frequently. Particularly, the Minkowski metrics also referred to as L_p metrics, such as the Euclidean (p=2) and Manhattan metric (p=1), are frequently used. The general definition is given in Equation 1. Metrics have special algebraic requirements (e.g., the triangle inequality must hold) used by state-of-the-art indexes, such as iDistance [17]. Therefore, we restrict the considered distance functions to metrics, even as this is no special requirement for exploiting our concept.

2.2 Related work

There is a large variety of different approaches. Therefore, we do not focus on specific approaches, but on underlying concepts and name well-known examples for each concept. For specific approaches, we refer to large-scale comparisons cited in each group.

Classic indexing techniques. Approaches like R-Tree [15] or kd-Tree [3] have given way to various improvements [4, 9, 19]. There are large studies or surveys addressing this topic comprehensively, like [6, 11, 20]. All of these approaches use geometric forms defined on the full-space, which is the primary reason why they tend to degenerate to a sequential scan for high-dimensional data. Nevertheless, to show differences, we include one approach from this group in our comparative studies. The number of dimensions they can efficiently support varies, but usually is deemed between 10 and 20 [6, 26].

As solution, optimized sequential searches are proposed, such as the VA-File [26]. Their core idea is using a compressed representation of the data that fits into main-memory allowing to filter points efficiently. The data itself resides on hard disk being multiple orders of magnitudes slower than main memory. This difference is known as access gap. However, with increased main-memory capacities, usually the whole data set fits into main memory. Conceptually, one could transfer this idea, such that data resides in main memory and the VA-file in a higher cache level (i.e., L3 cache). However, the speed difference between L3 cache and main memory usually does not exceed one order of magnitude, i.e., the access gap is much smaller and its exploitation therefore less promising. To this end, we do not consider such concepts.

Metric indexing with pivots. The AESA approach introduced the idea of pivot-based indexing [18]. Generally, the idea is to determine several pivot points. Then, one maps any point in the data set to its nearest pivots. Due to these mappings the triangle in-equality between query point, pivot, and data point can be used for computing a lower bound of the distance between query point and data point using any metric [8, 25]. Thus, lower bound computation is a simple addition and subtraction of pre-computed values. Various approaches rely on this principle. The most comprehensive comparison is [16]. A particularly relevant approach is iDistance [17], assigning one-dimensional indexes to the pivot-point mappings. To this end, it allows to efficiently traverse promising candidates and prune not relevant ones. Based on the one-dimensional index, it can be used for data on hard disk or in main memory [23, 27]. We are particularly interested in revealing differences between approaches relying on pivots and approaches relying on sub-space distance equalities. To this end, besides response time, we measure the number of points for that the index computes the exact distance, which is also independent of the hardware and programming language used.

Approximate approaches. There are various approximate approaches that deliver an approximation of the correct k-nearest neighbor satisfying certain bounds, such as [1]. A large group of approaches is locality sensitive hashing (LSH) [10, 13]. We do not consider this group, since our objective is to find the correct nearest neighbors.

[1] http://www.elf.ovgu.de/knn

3. KNN COMPUTATION WITH SUB-SPACE DISTANCE EQUALITIES

In this section, we first introduce the concept of sub-space distance equalities and explain two effects allowing to exploit it for efficient knn computation. Moreover, we briefly describe Elf as an index featuring sub-space distance equalities. Finally, we introduce two knn computing algorithms aiming at exploiting sub-space distance equalities in Elf. With the help of these algorithms, we examine how to optimally exploit sub-space distance equalities for knn computation.

3.1 Sub-space distance equalities

To explain the concept of sub-space distance equality and its exploitation for knn computation, assume some n-dimensional data set, with $n > 2$ and sort it to some dimension order. That is, we do not only sort according to one dimension, but according to all. For simplicity of explanation, let us assume that we sort according to $dim_1, \ldots, dim_n$. As a result, we observe that all points having the same value in the first dimension (dim_1) are found next to each other. However, that does not only hold for points having the same value in the first dimension, but for all points sharing the same prefix. A prefix pre_u refers to the first u dimensions, i.e., it projects the n-dimensional point to a u-dimensional one. It is important to note that all points sharing the same prefix pre_u are represented by the same point in the corresponding u-dimensional sub space. Consequently, they also have the same distance to any possible query point in that sub space. For illustration, consider the (ordered) example data set Figure 2b. We observe that points T_1 and T_2 have the same prefix until dimension dim_1. Consequently, in the one-dimensional sub space consisting only of the first dimension, the distance to any query point q is dist(q[1], T_1[1]) = dist (q[1] = T_2[1]), where q[1] refers to the value of q in the first dimension. This is what we call a sub-space distance equality.

3.1.1 Two effects for knn computation

Sub-space distance equality features two *effects* that can be exploited for efficient knn computation. The first effect is *re-use of sub-space distances*. One can compute sub-space distances for each set of points having the same prefix only once, instead of computing it for each point. The second effect is, one can use the sub-space distance in a u-dimensional sub space as *lower bound* for the full-space distance. We now outline both effects and how iteratively computable metrics allow exploiting these effects. Finally, we give examples which metrics are iteratively computable.

A metric is iteratively computable iff, given two arbitrary n-dimensional points p and q, the distance dist(p,q) can be computed as dist(p[1],q[1]) $\oplus$ dist(p[2],q[2]) $\oplus \ldots \oplus$ dist(p[n],q[n]).

Re-use of sub-space distances. In case $\oplus$ is associative, we can re-use the sub-space distance as follows: we compute the $l-1$ dimensional sub-pace distance for all points P with the same prefix pre_{l-1}. Then, we split P, such that we group all points having the same prefix pre_l. Next, to compute the sub-space distance for each newly created group, one only needs to add their distance in l, which is dist(p[l],q[l]), as all points in such a group have the same value in dimension l. To optimally exploit this effect, one needs to maximize the number of sub-space distance equalities, which can be done by selecting a respective dimension order.

Lower bound. One can exploit this effect, if dist(p[n],q[n]) is positive, i.e., ≥ 0. As a result, any sub-space distance is a lower bound for the full-space distance. Consequently, in case the sub-space distance of a set of points exceeds or is equal to max_dist (the distance of the currently found k^{th} nearest neighbor), those points cannot be part of the query answer. To optimally exploit this effect, one needs to provoke large sub-space distances, also by selecting a respective dimension order.

Unknown characteristics of both effects. Considering both effects, we state it is not intuitively clear how to find a dimension order that optimizes either effect. To this end, we investigate how to select the dimension order, which we name the dimension order selection problem in Section 4.1. To examine this problem, we design knn computing algorithms exploiting both effects.

Existing iterative metrics. We find various metrics whose computation is iterative, such as the Manhattan, Chebychev, or Hamming distance. However, generally the class of L$_p$ metrics, such as the Euclidean metric, is not iterative. Nevertheless, due to the definition of the knn problem (cf. Section 2.1), we do not require for the actual distance values, but only the correct k-nearest neighbors (in the right order). Hence, we can use a replacement distance function dist'() that is order preserving w.r.t. dist(). For instance, we can use the squared Euclidean distance, which is iterative and requires less computational effort. As this is possible for every L$_p$ metric, this allows us to benefit from the concept of sub-space distance equality for various, well-known metrics. Note, knn computation is possible for non-iterative distances as well. In such cases, we need to compute the distance considering the whole sub space instead of simply adding dist(p[n],q[n]) per dimension.

3.1.2 Distinction of sub-space distance equalities to other approaches

Relying on sub-space distance equalities for knn computation is fundamentally different from existing approaches. For illustration of the difference, let us consider R-trees and their derivatives, like R$^+$-tree [24], R*-tree [2], M-tree [9], X-tree [4], using n-dimensional geometrical objects to index sub spaces. Knn computation with these indexes has two problems: 1) expensive distance re-computation and 2) considerably large tree depths. First, descending the tree, the geometrical objects used for indexing the space shrink in every dimension. This means that also their whole n-dimensional distance to the query point has to be recomputed in each descending step. Second, the depth of the tree is not limited to a specific depth but is restricted to the number of points and fan out. Both points lead to a hardly definable computational complexity converging to visiting all tree nodes as dimensionality increases [14, 5].

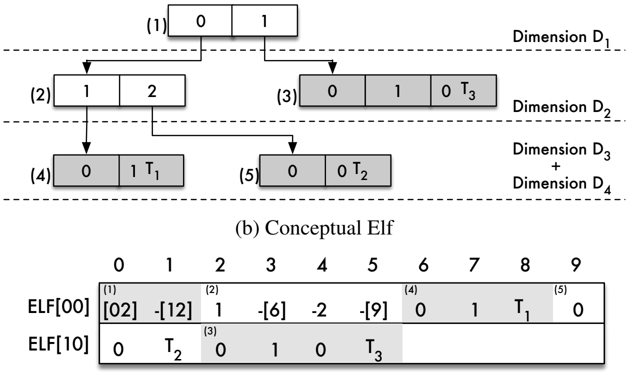
In contrast, this is different when exploiting sub-space distance equalities, as we iterate along dimensions refining the currently found distance. Hence, the previously computed sub-space distance does not change and can be reused for the currently investigated dimension.

3.2 Elf as MCSP index

To our knowledge, Elf is the only index that features sub-space distance equalities and allows to study the effect of sub-space distance equalities in isolation and in combination with a main-memory optimized storage layout. Therefore, it is the starting point for our own investigations. Originally, Elf is proposed to efficiently evaluate multi-column selection predicates named MCSPs. MCSPs are predicates that include several (but not necessarily all) attributes of one table [7]. Thus, it is designed for a different purpose, and we have to extend that approach with knn-computing capability exploiting sub-space distance equalities as defined in Section 3.1. To this end, we introduce its conceptual design and memory layout. Based on that, we develop knn computing algorithms in Section 3.3.

D_1	D_2	D_3	D_4	*TID*
0	1	0	1	T_1
0	2	0	0	T_2
1	0	1	0	T_3

(a) Running example data



(b) Conceptual Elf

	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) [02]	-[12]	(2) 1	-[6]	-2	-[9]	(4) 0	1	T_1	(5) 0
ELF[10]	0	T_2	(3) 0	1	0	T_3				

(c) Memory layout of Elf

Figure 2: Example data, conceptual Elf, and Elf memory layout

3.2.1 *Conceptual design*

We illustrate the design of Elf with the data set in Figure 2a. Each tuple in the data set has 4 dimensions and a *TID* to uniquely identify a tuple. Conceptually, Elf incrementally indexes existing values in sub spaces. That is, the first level in Elf (i.e., the root level) refers *only* to the first dimension. The second level refers to the first and the second dimension. Hence, it indexes a two-dimensional sub space of the whole data set. The third level therefore refers to a three-dimensional sub space etc. The order of dimensions thereby is the only parameter. Each node of Elf, called `DimensionList`, contains entries of the form [Value, Pointer] ordered according to Value. The first level of the tree consists of *one* root node that contains every unique value of the first dimension – for instance, the values 0 and 1 of D_1 in the Elf in Figure 2b. Each entry is the start of one *path*.

All tuples having the same value in the first dimension are in the same path. More generally, all tuples having the same prefix are found in the same path. For example, all tuples represented by `DimensionList` (5) have the prefix 0 in the first dimension and 1 in the second dimension. Constructing the tree in this manner spans a tree with d levels where d corresponds to the number of indexed dimensions. However, due to the curse of dimensionality [5], the more dimensions are indexed, the more sparsely populated (i.e., shorter) are the `DimensionLists` in deeper levels. The worst case is that we have a linked-list like tree, which Elf counters with the concept of `MonoLists` [7].

Whenever one encounters a tree level t in some path such that there only remains one tuple in it (i.e., there is no fanout), the build algorithm creates a `MonoList`. That is, from level t to d (where $t < d$) Elf stores all values adjacent to each other. In Figure 2b, `DimensionLists` (3), (4), and (5) are `MonoLists`, because there is no fanout since each of these `DimensionLists` correspond to only one tuple. The benefits of `MonoLists` are that they reduce storage costs and improve data locality [7].

3.2.2 *Optimized memory layout*

It is possible to build Elf with a main-memory optimized storage layout as shown in Figure 2c. To this end, one linearizes Elf into an array, using a preorder traversal. In Figure 2c, we present the linearized Elf from Figure 2b. The first linearized `DimensionList` is a hash map storing pointers to the underlying `DimensionLists`. The second `DimensionList` stores values *and* pointers of the `DimensionList` (2), where values in brackets represent pointers within the array. To minimize storage consumption, one does not store length information of a `DimensionList`. Instead, the most significant bit (MSB) of the last value denotes the end of a `DimensionList`. We visualize this with a negative value. For example, the -2 at offset 4 denotes the last entry of `DimensionList` (2). Similarly, a negative pointer indicates that the following `DimensionList` is a `MonoList`.

For our own investigations, the optimized storage layout offers two interesting examinations possibilities. First, we can investigate whether different linearizations yield different performance, and second quantify the effect of the best linearization strategy comparing it to its unoptimized version.

3.3 Knn algorithms for Elf

We now introduce two knn algorithms exploiting sub-space distance equalities within Elf. The algorithms are optimized for different properties of the data. The first algorithm aims at minimizing the number of distance computations. To this end, it aims at fast converging towards the final distance of the k^{th} nearest neighbor allowing to prune entire sub trees within Elf. By contrast, the second algorithm aims at data sets that deteriorate towards scanning an entire Elf optimally exploiting the data layout. The results in Section 4.1.4 allow selecting the best algorithm for a given data set.

```
 1 KnnConverge(q, k, dim, dimList, subSpaceDist)
 2   for(each elem in dimList) //find best match
 3     if(dist(elem.v,q[dim])->minimal) break
 4     newDist <- subSpaceDist + dist(q[dim],elem.v)
 5     if(newDist>=max_dist) return //prune all
 6     if(isMonoList(elem.p))
 7       knnElfMono(q,k,dim+1,elem.p,newDist)
 8     else
 9       KnnConverge(q,k,dim+1,elem.p,newDist)
10     elemRight<-elem; elemLeft<-elem
11     while(!done){ //search in- and outwards
12       elemRight<-elemRight.next
13       newDist<-subSpaceDist + dist(q[dim],elemRight.v)
14       if(newDist>=max_dist || elemRight.last)
15         doneRight<-true
16       else
17       //recursive call like in Line 6-9
18       // similar for elemLeft
19       if(doneRight&doneLeft) done <- true
20     }
21
22 KnnOptLayout(q, k, dim, dimList, subSpaceDist)
23   elem <- dimList.first
24   do{
25     newDist<-subSpaceDist + dist(q[dim],elem.v)
26     if(newDist>=max_dist) continue //prune
27     else
28       if(isMonoList(elem.p))
29         knnElfMono(q,k,dim+1,elem.p,newDist)
30       else
31         KnnOptLayout(q,k,dim+1,elem.p,newDist)
32   }while(!elem.last)//reached end}
```

Figure 3: Elf knn algorithms with different optimizations

3.3.1 *Knn algorithm optimizing pruning power.*

The first knn algorithm `KnnConverge` in Figure 3 aims at traversing Elf such that `max_dist` converges fast against the distance of the k^{th} nearest neighbors. This optimizes Elf's ability to prune sub trees that do not contain query solutions. To this end, an Elf is traversed in a greedy manner starting by invoking `KnnConverge` for the first dimension list. In each list, the algorithm searches for the best sub tree (best match), which has minimum distance to the query point q. The algorithm first examines the corresponding sub tree of that element. Then, it examines the remaining dimension elements in the current dimension list iteratively. That is, in the first iteration one element to the left and one to the right are examined. The sub trees are only examined in case their sub-space distance is smaller than the (full-space) distance of the current kth nearest neighbor being `max_dist`. The algorithm stops, in case there are no more solutions to the left *and* right or in case the best match already does not contain a query answer (Line 5). Note, `knnElfMono` computes the distance for the remaining dimension of *one* point.

3.3.2 Knn algorithm optimizing data locality.

The second knn algorithm is optimized for data sets, where pruning, for instance due to the curse of dimensionality, is difficult. To this end, this algorithm does not search for the best match in each dimension list. It strictly follows the data layout of Elf and thus, optimizes data locality. Therefore, it starts at the first element of each dimension list, examines its corresponding sub tree, and then, continues with the next element. As a result, this algorithm is more similar to a sequential scan which benefits primarily from data locality. However, we still benefit from sub-space distance equalities reducing the number of distance computations and we also check whether we can prune. Note, traversing Elf results in some overhead that a sequential scan does not have. Particularly, for high-dimensional uniform and Gaussian data, where sequential scans outperform any known indexing approach. So far, we hypothesize that due to this algorithm Elf is more robust towards the curse of dimensionality than other indexing approaches and examine this in Section 5.

3.4 Research questions

Since exploiting sub-space distance equalities is a novel concept, we aim at revealing in-depth insights by means of systematic experiments in the following section. In particular, we firstly investigate how both effects from Section 3.1 using different dimension orders affect knn computation and secondly quantify the influence of the optimized storage layout. Considering Elf as foundation for our investigation results in the following research questions (RQ):

1. RQ1: How to optimally exploit the two effects, sub-space distance equalities induce, using a respective dimension order within Elf?
2. RQ2: How to quantify the effect of Elf's optimized data layout and its interaction with an appropriate knn algorithm?

The results of these micro benchmarks are used in Section 5 for our comparative studies considering different data sets, distributions, and Euclidean metric.

4. PERFORMANCE FACTOR TUNING

In this section, we examine the factors that are relevant for Elf's knn performance. In Section 3.4, the two research questions directly refer to performance factors: RQ1 directly translates to the question of how to order the dimensions in Elf. RQ2 aims at finding an optimal storage layout and quantifying the effects of the storage layout in general. In the following, we conduct a set of micro benchmarks, to answer these RQs. The results allow to build an optimized Elf for any data set.

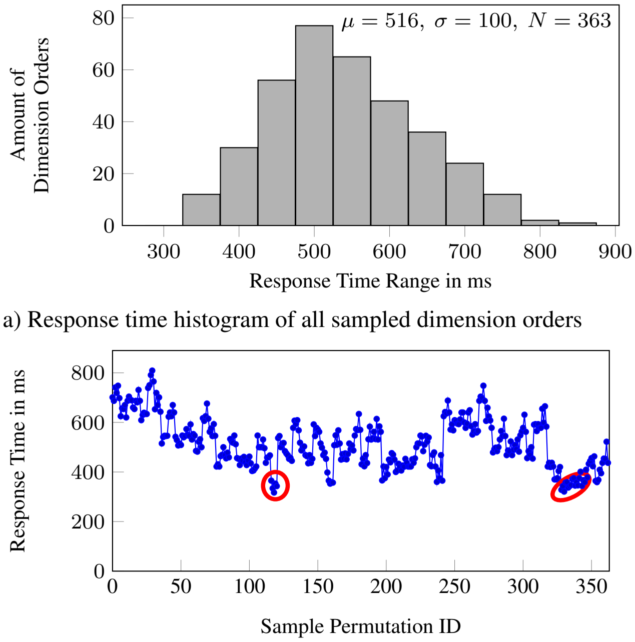
4.1 Dimension-order selection problem

The dimension order is the only parameter of the Elf build algorithm. From [7, 22], we know how to construct an optimized dimension order for MCSP-evaluation and predict Elf's performance. Unfortunately, the situation for knn queries is different. For knn queries, we have to descent *all* paths as long as the remaining sub tree can contain at least one query solution. Moreover, it is unknown whether one needs to optimize Elf for the re-using sub-space equalities effect (i.e., maximizing the number of prefix redundancies) or the lower bound effect that allows pruning. The latter one means maximizing the sub-space contrast generally minimizing the number of prefix redundancies. Thus, both targets are in contrast to each other. Our goal is to find an approach that allows to determine a good dimension order in a reasonable amount of time. What makes this investigation particularly challenging is that the number of possible dimension orders grows exponentially.

Our investigation procedure is as follows. First, we examine one data set in detail and then generalize results. To this end, we take the D_r^{16} real-world data set (cf. Table 1) and create a new data set for that we can sample all possible (n!) dimension orders and construct the respective Elf. We call this data set *micro benchmark data set*. In addition, we restrict the empirical investigation to the Euclidean metric, which is often used for such investigations.

4.1.1 Influence of the dimension order

The first set of experiments aims at studying the problem comprehensively. To this end, we answer the question: Does the order matter, quantify the difference of a good, normal, and bad dimension order, and the probability of finding either class by chance.



a) Response time histogram of all sampled dimension orders

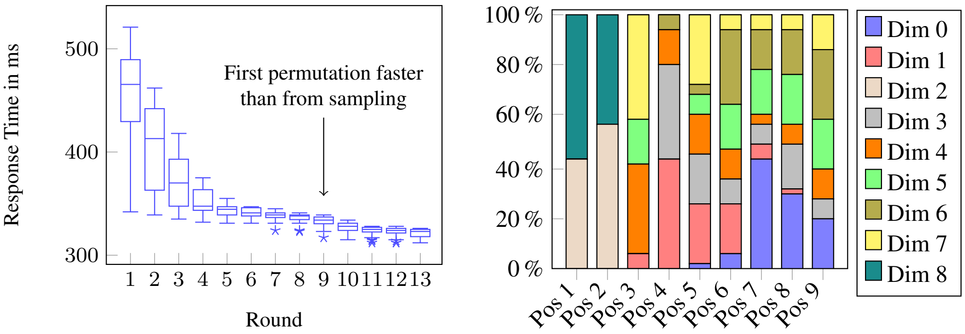
b) Pattern of the response time distribution

Figure 4: Results of the sampling method.

Sampling method. For our micro benchmark data set having nine dimensions, testing all 9! dimension order permutations and determining the response times for retrieving all k nearest neighbors for *all* 16k points in a statistical sound manner is not possible (approx. 126 computing days). Hence, we sample the space of the existing permutations taking every 1,000th permutation resulting

	Genetic	Sampling
Fastest	312 ms	317 ms
Avg Top 10	330 ms	335 ms
Slowest	870 ms	809 ms
Avg Slowest 10	850 ms	751 ms

a) Method's example results



b) Results from genetic optimization

c) Commonalities of fast dimension orders

Figure 5: Results from genetic optimization and how they relate to the dimension order.

in 363 samples. We depict a histogram of the sampled response times in Figure 4a) revealing how the response times are distributed. By contrast, Figure 4b) visualizes whether there are regularities in the permutation space regarding the response times. Based on the results shown in Figure 4a), we conclude that the dimension order indeed has a significant influence on the response time. The fastest order requires 317 ms, the average is about 516 ms, and the slowest ones more than 800 ms for finding all nearest neighbors for all points. For comparison, the response time of a sequential scan is 1,380 ms. Hence, guessing a dimension order definitely results in a better performance than a sequential scan. However, as the response times are distributed in a skewed Gaussian manner, probabilities are high not to select a fast order. The pattern of the distribution w.r.t. the linearized permutation space in Figure 4b) (i.e., the order of the permutation and their corresponding response time), reveals a mixture of local and global regularities (i.e., similar orders have similar response times). Hence, the dimension-order selection problem is a well-suited problem for genetic optimization.

Genetic method. Our goals with this method are twofold. First, we intend to find a method that is applicable on larger data sets (i.e., particularly higher number of dimensions) that cannot be sampled efficiently. Second, we want to explore the edges of the response time distribution in more detail. In particular, we investigate commonalities of the fastest and slowest permutation in order to predict a good dimension order analytically.

The results from the genetic method depicted in Figure 5, are consistent with those of the sampling method. Hence, we claim that we studied the effect of the dimension-order selection problem on the response time of Elf for this data set in a comprehensive way. Now, we investigate how to predict a good dimension order. In particular, the best-found dimension order has a response time of 312 ms, which is only 5 ms faster than the fastest permutation from the sampling method. By contrast, the slowest permutation found requires 870 ms compared to 809 ms.

In Figure 5b), we show a box plot visualizing the response time distribution of the resulting population. After 9 rounds of the genetic algorithm, the population already contains a dimension order that yields faster response times than any order observed with sampling. In round 13, the genetic method delivers a set of 50 dimension orders, where the slowest order has a response time of 326 ms. We analyze these permutations for commonalities to find a practical method to predict a good dimension order. As starting point, we visualize for these orders the frequency of occurring dimensions for each position in Figure 5c). From the figure, we see that all good dimension orders start either with [8,2,...] or [2,8,...]. That is either the 8th or 2nd dimension must be first and the other one second. These are the same regions in the permutation space identified as having minimum response time being encircled in Figure 4. Interestingly, inverting a fast dimension order results in a slow dimension order. The slowest ones we found are in fact inverse to the fastest ones.

4.1.2 Investigating the dominant performance factor

Based on the insights of the prior method, we now analyze what is the factor in Elf to optimize to achieve good response times using the dimension order. Therefore, we formulate hypothesis H_1 and H_2, each expecting one performance factor to be the decisive one:

Hypothesis H_1 A high number of sub-space distance equalities is the primary factor for good performance.

Hypothesis H_2 The pruning power of Elf is the primary factor for a good performance.

To confirm or reject the hypotheses, we first compute one performance indicator for each of them and then examine how they are related to fast and slow dimension orders.

H_1: Elf compression factor quantifying sub-space equalities. The first indicator, the Elf Compression Factor (ECF), quantifies how many nodes we do not store compared to simple sequential storage of the data set. Therefore, it counts the number of prefix redundancies and normalizes the result. We explain this indicator with the help of the example data set D having four points: p_1=(1,2,3), p_2=(1,2,2), p_3=(1,1,1), p_4=(2,1,1). First, we count every observed prefix redundancy: There are three points (p_1, p_2, and p_3) having the value 1 in the first dimension. Hence, we can eliminate two prefix redundancies. In the second dimension, only the prefix (1,2) exists twice. Hence, we observe one additional prefix-redundancy. Next, we compute the maximum number of `dimension elements` used to normalize the result, which is equivalent to the worst case. In Elf that would occur in case there are no prefix redundancies at all. Hence, the number is the product of the dimensionality of the data set and the number of points, in this case: 12. As a result, for D we get a value of $\frac{3}{12}$. Thus, a value close to zero indicates high compression, while values close to 1.0 mean almost no compression.

Rejection of ECF as dominant factor. Our experimental results reveal that the ECF of all 9! dimension orders are approximately uniformly distributed between the values [0.173, 0.257] with an average of 0.208. The ECF values for the fastest 50 dimension orders found in the genetic optimization are in the interval [0.215, 0.240] with an average of 0.227. This interval covers 110,000 dimension orders, being roughly one third of all orders. Therefore, there only is *some* tendency for fast dimension orders to have an ECF rate that is higher than the average. However, even for the highest ECF rate, there are more than 16,000 dimension orders that have a higher ECF rate. For the slow dimension orders, we observe an inverted tendency. That is, they generally have lower ECF than the average. As a result, this is not the dominant factor and thus cannot be used to predict a good dimension order.

H_2: Sub-space contrast quantifying the pruning power. The second indicator (SSC_u^p – Sub-Space Contrast) denotes the average distance according to `dist()` between two points in D for the first u-dimensional sub space of a given dimension order p. We compute SSC_u^p as the sum of the distances of all point pairs and normalize by the number of pairs. For larger data sets, we use a sample of the data. Based on the definition of SSC_u^p, its value is the same for all dimension orders if u refers to the whole space (full-space contrast). However, in case pruning power is the decisive property influencing the response time of Elf, we need to maximize the sub-space contrast for small values of u to get large distances in high levels of Elf by selecting a respective dimension order.

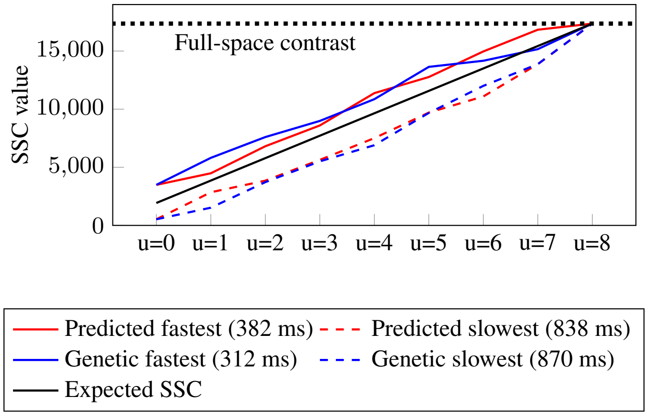


Figure 6: Dimension order selection: Exemplary SSC indicators for different values of u.

Confirmation of SSC_u^p as dominant factor. We confirm the Hypothesis H_2 that the pruning power is the dominant factor in case the fastest 50 dimension orders found in the genetic optimization are among those that converge fast against the full-space contrast for small values of u. By contrast, for the 50 slowest dimension orders, we require to observe the opposite behavior.

In Figure 6, we depict exemplary SSC_u^p values for four dimension orders and a baseline (black line). The baseline assumes that the full-space contrast is uniformly distributed among all sub spaces, i.e., the delta between all SSC_u^p and SSC_{u+1}^p is always the same. We observe for the two fast example permutations that the SSC for small u is significantly higher than the baseline. For the fastest permutation, we found that this is the maximally achievable SSC value for $u < 3$. We make the opposite observation for slow dimension orders. That is, their SSC values are consistently smaller than those of the baseline. For both groups of permutations found in the genetic optimization, these observations hold for all members of the group. In addition, further tests reveal that dimension orders having consistently similar SSC values as the baseline, have a median response times. Hence, we confirm SSC_u^p as the dominant performance factor and use it to analytically predict a good dimension order.

4.1.3 Determining good dimension orders

We revealed that the pruning power, indicated by large SSC_u^p for small values of u, is the decisive performance factor. From investigating the impact of the dimension order problem on the response time, we know there is a set of fast permutations with similar response time. Hence, it is reasonable not to aim at finding the best permutation, but a reasonably good one. To determine good orders, we propose two algorithms.

Local estimates. The first algorithm, `local()`, aims at fast identification of a good dimension order, but does not take correlated dimensions into account. It works either on the computed SSC_1^p values or a respective estimate, which depends on the applied distance function `dist()`. For L_p metrics, we use the variances of the dimensions as estimate. Having, the estimates (or correct SSC values) the dimensions are sorted putting the one with largest estimate first. The resulting order directly forms the dimension order. We can only rely on the variance if the following holds: Given three values c, v_1, v_2: $\text{dist}(c, v_1) > \text{dist}(c, v_2) \longleftrightarrow |c - v_1| > |c - v_2|$. Using an example that means, it is required that according to `dist()` the value $v_1 = 0$ is always further away from $c = 10$ than the value $v_2 = 1$. Using the Hamming distance, v_1 and v_2 would have the same distance, as both values are not equal to the compared value c. In Figure 6, the fast and the slow predicted dimension order (dashed lines) are computed using this algorithm with variance as estimator. Predicting a fast order results in a permutation having a response time of 381 ms, meaning that there are 8.3% of all permutations faster than the predicted one. This is arguably not an optimal result, but the ratio of invested time and achieved response time compared to an average response time of 516 ms is decent.

Greedy determination. The second algorithm `greedy()` offers reasonable execution times and considers correlated dimensions and sub spaces reducing the sub-space contrast. The main difference is that we compute the SSC in an iterative way. First, the dimension with the highest SSC value is determined and forms the already known part of p. Then, by probing all concatenations of the current already known permutation and all remaining dimensions, the algorithm selects the concatenation having the maximum SSC value. Interestingly, for the micro benchmark data set and the Euclidean metric the result is the same as for the first algorithm, but the execution cost are by far higher. For this second algorithm, for each of the d iterations, we compute in every step u the SSC value of d-u possible sub spaces. This requires, for each possible sub space to compute the pair-wise distance for all points in the data set, i.e., this algorithm's complexity is higher than $O(|D|^2)$. As extension, one could also probe the i best concatenations from each iteration. However, the execution time increases exponentially, e.g., in case i = d one would test all permutations.

Table 1: Benchmark data sets from [21]

Dim	Size	real-world	property	uniform	Gaussian
16	11k	D_r^{16}	dense	D_u^{16}	D_G^{16}
43	412k	D_r^{43}	clustered	D_u^{43}	D_G^{43}
50	130k	D_r^{50}	sparse	D_u^{50}	D_G^{50}
51	3446k	D_r^{51}	clustered	D_u^{51}	D_G^{51}

4.1.4 Empiric generalization

So far, we examined the influence of the dimension order parameter with one data set resulting in the insight that we can determine good dimension orders using the SSC value. Now, we aim at generalizing this result. To this end, we use the same benchmark data sets (cf. Table 1) as used in [21]. It consists of 12 data sets. There are four groups, where the dimensionality and size of all data sets within one group is identical. For each group there are three data sets with different stochastic distributions. One data set contains real-world data, the others are uniform and multivariate Gaussian data. We now investigate whether we can rely on the SSC value in general.

We use the `local()` algorithm to find good dimension orders. Hence, a slow dimension order is the inverse permutation of the fast one. In Table 2, we depict the results. For space reasons, we only depict the results for all real-world data sets and the whole 16-dimensional group. This is valid as for any other group the results for the uniform and Gaussian data are, as expected, nearly the same. In particular, this table contains the quotient of the average response times of the slow and fast dimension orders as ΔT_{resp}. In addition, it contains the deviation from the SSC values of a good dimension order to the expected SSC value named ΔSSC. It is defined as $\sum_{\text{dim}=1}^{d} \text{SSC}_{dim}^{\text{fast}}$ - $\text{SSC}_{dim}^{\text{exp}}$ normalized by dividing it by $\text{SSC}_{dim}^{\text{exp}}$. Intuitively, this can be interpreted as the (normalized discrete) integral of the predicted and the expected SSC graphs in Figure 6. Generally, the assumption formulated in Hypothesis H_1 is: the larger the difference between predicted and actual SSC value, the better Elf performs. On the other hand differences close to zero predict no observable difference in ΔT_{resp}, i.e., for this data set the dimension order does not matter.

Table 2: Comparison of delta between SSC and response time

Data set	D_r^{16}	D_u^{16}	D_G^{16}	D_r^{43}	D_r^{50}	D_r^{51}
ΔSSC	0.99	0.17	0.18	21.45	11.65	1.24
ΔT_{resp}	2.50	1.10	1.10	26.60	41.79	1.76

Based on the results in Table 2, we confirm that one can generally rely on the SSC value. First, for small ΔSSC values, such as 0.17, we observe nearly no difference of the response times of the fast and slow dimension orders, i.e., a ΔT_{resp} value close to 1. Second, for a larger Δ SSC value, we generally observe larger ΔT_{resp}. In summary, the answer regarding RQ1 (cf. Section 3.4) is that we need to optimize for large ΔSSC. Then, the corresponding good dimension order in combination with `KnnConverge` forms an optimized Elf. In case, ΔSSC is small, we use `KnnOptLayout` and the dimension order is not relevant.

4.2 The influence of the Elf data layout

Besides the concept of sub-space distance equality, Elf features an optimized storage layout. In this section, we investigate whether the same layout as for MCSP computation shall be used and quantify the effect on response time. To this end, we conduct a second micro benchmark. We use an Elf without optimized storage layout and in addition test an alternative layout with both knn algorithms from Section 3.3. Recapitulate, the main difference between both algorithms is that `KnnConverge` is optimized to converge to the final distance of the k^{th} nearest neighbor faster by iterating first over each dimension list and then traverse the closest sub tree first. By contrast, the `KnnOptLayout` is optimized to exploit data locality in Elf and does not iterate twice over all values.

4.2.1 The memory layouts

We consider three layouts: MCSP, Depth-first, and List. The first is an Elf with optimized layout as shown in Figure 2c. The optimization objective is to speedup examination of one dimension list, by storing all its values and pointers adjacent to each other. The second layout minimizes the cost for jumping to the next level (i.e., next dimension). We depict the resulting Elf for the example data from Figure 2a in Figure 7. The core difference is highlighted in yellow being the second dimension list split into parts. Note, the size of Elf remains the same in both layouts. The third layout considers dimension lists as *lists* potentially scattered across the memory. That is, there is no optimized layout.

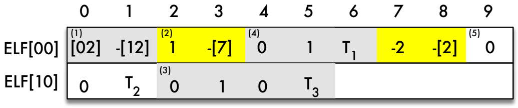


Figure 7: Alternative depth-first memory layout of Elf

4.2.2 Results

We examine for each layout the average response time for the best 50 permutations from the genetic optimization of Section 4.1.1. Again for each permutation, we compute a robust mean value ensuring that numbers are statistically sound.

Table 3: Average response time for different memory layouts

Layout:	MCSP	Depth-first	List
`KnnOptLayout`	1152 ms	1176 ms	2513 ms
`KnnConverge`	383 ms	460 ms	722 ms

The results indicate that optimizing the convergence of the knn algorithm `KnnConverge` is more important than finding a good dimension order. This is visible in Table 3. Even for the best dimension order from the genetic optimization, the response times using `KnnConverge` are consistently slower than the slowest permutation found in the prior section using `KnnOptLayout`. However, for the List Elf, without optimized storage layout, the response times are by factor 2 slower. Interestingly, there is no observable difference for knn Algorithm `KnnOptLayout` between the two Elf variants with optimized memory layout. However, for `KnnConverge` we observe a difference. It results from iterating twice over a dimension list. Hence, using the depth-first layout means that `KnnConverge` needs to jump twice over the memory for every dimension list, while the sub tree is adjacent. For the default layout the only jump is performed examining the sub tree, while the dimension list values are stored adjacent to each other. To quantify the difference, recapitulate that the response times for the permutations are normally distributed (cf. Figure 4) with a mean of 516 ms using the first layout. With 460 ms for the top 50 permutations of the depth-first memory layout, this value is only slightly better the average of the first layout. In summary, we conclude that using the same layout as proposed in [7] is expected to result in the best performance being the answer regarding RQ2 from Section 3.4. Hence, we apply it in the remainder of the paper.

5. COMPARATIVE STUDIES

The answers regarding the research questions RQ1 and RQ2 in Section 4 allow to build optimized Elf indexes. Based on these insights, we compare the Elf approach to state-of-the-art and well-known competitors using different data sets in a systematic way. To this end, we first introduce the study design ensuring a fair comparison. Then, we present evaluation results.

5.1 Study design

Ensuring a fair comparison of different approaches in main-memory environments is a non-trivial task. To this end, we explain how we ensure a fair comparison first.

5.1.1 Index and measurement selection

Besides the sequential scan, which is known to be a highly-potent competitor due to the curse of dimensionality, we use iDistance [17] as state-of-the-art indexing approach for metric spaces. We use the kd-Tree [3] as classical indexing approach, which is known to work well in main memory settings due to its small size and simple algorithm particularity compared to members of the R-Tree family [15]. Finally, we also include an Elf without optimized memory layout named *List Elf* to examine the effect of the optimized memory layout.

Speedup as comparison measure. We use two different measurements for each triple of index, metric, and data set. The first is the average response time for executing 1,000 knn queries. To ensure statistical soundness, the 1,000 points are randomly selected. We repeat each experiment five times. Each experiment returns the median of ten measurements, i.e., we compute ten executions of 1,000 knn queries. The average response time is the average of all five experiment medians. We normalize this average response time by the average response time of the sequential scan leading to the speedup of each index over the sequential scan. As usual, a speedup value smaller than 1.0 indicates that the corresponding approach is slower than the sequential scan, while higher values indicate the factor of performance improvement over the sequential scan.

Number of distance computations. For a hardware-independent comparison, we measure the average number of points per query for that an index computes the real distance to the query point, named M. As for the response time, we normalize the measurement M^n by the number of points visited by the sequential scan (i.e., the data set size). Hence, a value close to 1.0 indicates that nearly all points are examined. Due to sub-space distance equalities, for Elf, distances are not computed point-, but attribute-wise (cf. Section 3.1). Therefore, we count every distance computation per attribute, i.e., invocation of `dist(elem.v,q[dim])` as found in the algorithms in Figure 3. Then, we divide by the number of dimensions of the data set. This is valid, since for a worst-case Elf without sub-space distance equalities (i.e., the values in the first dimension are unique), Elf performs a sequential scan resulting in the same M^n value the sequential scan has. Note, the M values of Elf and List Elf are identical.

5.1.2 Benchmark data sets

We rely on the same data sets as used in [21], already used in Section 4.1.4, to evaluate different dimensionality, stochastic distributions, and amount of points in the data set. These data sets are particularly well suited to evaluate indexes in a systematic way, as there are four groups of data sets, where each group consists of three data sets. The groups have different number of dimensions reaching from 16 to 51 and different size reaching from 11,000 points to 3,446,000 points. To give an example, in group D^{50} all data sets have 50 dimensions and 130,000 points, whereas in group D^{51} all data sets have 51 dimensions and 3,446,000 points. Within each group, there is one data set containing uniform data, one containing multivariate Gaussian data, and one real-world data set. To improve readability, we use the following notation D^x_y to refer to a specific data set, where $x \in \{16, 43, 50, 51\}$ indicates dimensionality of the data set and $y \in \{u, G, r\}$ its stochastic distribution. For example, D^{51}_r refers to the real-wold 51 dimensional data set, whereas D^{50}_u denotes the 50 dimensional uniform data set.

5.1.3 Intrinsic and extrinsic validity

All index implementations are tuned to the same extent. The same holds for auxiliary structures such as the result set. In addition, we evaluated different number of queries and different values for parameter k revealing that their influence is negligible. For brevity, we only show the results for $k = 10$. Furthermore, we tested implementation variants for each index selecting the fastest one and repeated the experiments on different hardware all resulting in the same findings. Finally, we executed index-specific parameter tuning for each data set. For Elf that is finding a good dimension order. For iDistance, tuning is difficult as there are three parameters (number of partitions, partition center selection, Δr) and their influence is not easy to predict [23]. Our tuning results are consistent with [23, 17].

The evaluation is performed on an Intel Core i5 with 2.6 GHz clock frequency having 20 GB RAM. We use Java 8 following the guidelines from [12] for Java benchmarking.

5.2 Results

The Euclidean metric is one of the most often used L_p metrics with $p = 2$. Hence, we use it in our evaluation.

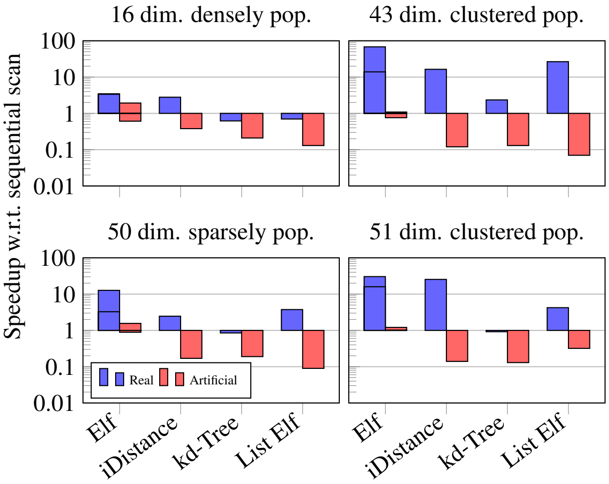


Figure 8: Speedup for Euclidean metric

Table 4: Distance computation number M^n Euclidean metric

M^n	D^{16}_r	D^{43}_r	D^{50}_r	D^{51}_r	D^{16}_G	D^{43}_G	D^{50}_G	D^{51}_G
Elf	0.002	0.001	0.001	0.004	0.265	0.367	0.491	0.371
iDistance	0.013	0.010	0.008	0.061	0.877	1	0.999	1
kd-Tree	0.078	0.023	0.163	0.209	0.961	1	1	1

5.2.1 Real-world data sets

In Figure 8, we depict the speedups for all data sets. Recapitulate that a value of 1 on the y-axis means that the respective approach is as fast as the sequential scan. A value of 10 in the logarithmic scale indicates that the approach is ten times faster than the sequential scan. In the results, we observe for all real-world data sets (in blue) that Elf consistently outperforms the sequential scan by several factors, which also holds for the iDistance. The highest speedup we observe is using Elf for the 43-dimensional real-world data set having a speedup of factor 68.23. In three out of four real-world data sets Elf is the fastest approach. For the 51-dimensional data set using Elf results in a speedup of about factor 10 compared to a sequential scan. However, the iDistance is two times faster for that highly clustered data set. We argue that this is explained by the way the iDistance works (cf. Section 2.2): In case its partition centers (by proper optimization) correspond to existing clusters, we found an optimal data set for such indexes. For the List Elf and the kd-Tree, we observe that even for real-world data the sequential scan outperforms them for two and three data sets, respectively. We also observe a clear superiority of Elf (with optimized memory layout) to its un-optimized counter part.

Considering the results in Table 4, we conclude that the speedups are explainable by the reduction of distance computations using indexing approaches. Interestingly, the ratio of M^n between the approaches approximately is the same as their difference in speedup. For valid interpretation why M^n values are comparable among all approaches, it is required to discuss a detail for the iDistance that also applies to any other metric indexing approach. Elf, kd-Tree, and sequential scan compute distances using squared Euclidean distance (i.e., omit the square root) referred to as `dist'()`. This is not possible for iDistance as the triangle in-equality used for pruning points only works with a metric, which `dist'()` is not. However, upon knn query execution (not upon build) we can in most cases rely on `dist'()`. Only in the case we found a new k^{th} neighbor, i.e., when we need to update `dist_max` of the result S, we have to additionally compute the square root. These updates are by orders of magnitudes smaller than M, so for nearly all distance computation also iDistance can rely on faster `dist'()` instead of `dist()`.

5.2.2 Gaussian and uniform data sets

In Figure 8, we depict the results for the multivariate Gaussian data sets in red. Note, the results for the uniform data sets are almost the same. Hence, for brevity we only depict the Gaussian results, but our observations and interpretations also hold for respective uniform data sets.

Generally, as expected, most approaches have severe issues reaching the performance of the sequential scan. The only exception is Elf. It is the only index that (slightly) outperforms the sequential scan for one of the data sets (the largest one having 51 dimensions and 3 million points) and reaches comparable speedups for the remaining ones. The largest difference (speedup factor of 0.61) is observed for the 16-dimensional Gaussian data, which we deem acceptable for the following reasons. First, all other indexes are by far slower. The next fasted index is the kd-Tree with a speedup factor of 0.13 being nearly an order of magnitude slower. Due to the small size of the data set, the sequential scan highly benefits of the large cache sizes of todays CPU. By contrast, tree-based indexes suffer for the concept related issue that they do not read memory sequentially, but jump across it. This is nevertheless an interesting result, as we expect the cache sizes to grow in future as well as main memory latency to decrease where such issues become relevant also for larger data sets.

Examining M^n in Table 4 reveals that all competitors degenerate to sequential scans for D_r^{43}, D_r^{50}, and D_r^{51}. The same holds for their uniform counterparts. This is different for Elf, which does not compute more than half of the distances for any data set. The maximum M^n value with 0.491 is observed for D_r^{51}, which means that on average 49% of all distances are computed. Hence, even for such data sets using Elf results in comparable performance.

5.2.3 Interpretation

Overall, our results indicate that Elf delivers the best performance for real-world data sets, even if not the best for all data sets, which we argue is hardly possible. However, there is a consistent improvement of several factors compared to a sequential scan. Moreover, for Gaussian and uniform data, we state Elf is the only approach that results in comparable performance to a sequential scan, i.e., is resistant to the curse of dimensionality. This becomes visible as for no data set more than 50% of all distances are computed. Therefore, we conclude that selecting Elf as index to speedup Euclidean metric computation (e.g., within a clustering approach) for data sets with unknown distribution is a good choice.

6. CONCLUSIONS AND FUTURE WORK

For many data analysis tasks, like clustering or outlier detection, computation of the k-nearest neighbors is required and computationally expensive. A novel concept that has the potential to significantly reduce the number of distance computations, independent of the stochastic distribution of the data, is exploiting sub-space distance equalities. The core result of this paper is that, in case one exploits sub-space distance equalities properly, one can expect to compute at maximum about 60% of the distance values. This is important as even state-of-the-art approaches face severe issues to outperform a simple cache conscious sequential scan if dimensionality increases.

To investigate the potential of our novel concept, we rely on Elf, in index featuring sub-space distance equalities combined with a main-memory optimized storage layout. Our results allow to build optimized Elf indexes by maximizing the sub-space contrast with Elf's dimension order parameter. In a comparative study with 12 data sets having different properties, we reveal that using Elf results in competitive performance for all data sets considering highly potent competitors. Our results reveal that for real-world clustered data the minimum performance increases compared to a sequential scan is factor 2, while the largest ones are more than two magnitudes. The competitors do not achieve this. Furthermore, even for Gaussian and uniform data, no more than 61% of the distances are computed and, in any case, the smallest number of distances of all approaches are computed. The reason is that sub-space distance equalities represent tight bounds (i.e., points instead of regions), which are incrementally refined per tree level. Moreover, as the distances are computed attribute-wise instead of point-wise, we can furthermore reduce the number of computed distances. Finally, if a large number of distances have to be computed, Elf benefits from its optimized storage layout, which naturally converges to a row-store-like (i.e., cache conscious) structure. This suggests that one can expect good performance using an index based on sub-space distance equalities, such as Elf, in general.

For future work, we are interested in studying the relationship and existence of order-preserving iterative versions of various well-known distance function. First results indicate that the requirements of Elf towards the applied distance function are less strict than those metric approaches have, which require a metric.

7. REFERENCES

[1] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. Int'l Conf. on on Management of Data (SIGMOD)*, pages 322–331. ACM, 1990.

[3] J. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 28–39. Morgan Kaufmann, 1996.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proc. Int'l Conf. on Database Theory (ICDT)*, pages 217–235. Springer, 1999.

[6] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the

performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[7] D. Broneske, V. Köppen, G. Saake, and M. Schäler. Accelerating multi-column selection predicates in main-memory - the Elf approach. In *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 647–658, 2017.

[8] L. Chen, Y. Gao, B. Zheng, C. Jensen, H. Yang, and K. Yang. Pivot-based metric indexing. *Proc. VLDB Endow.*, 10(10):1058–1069, 2017.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 426–435. Morgan Kaufmann, 1997.

[10] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. An'l Symp. on Computational Geometry (SCG)*, pages 253–262. ACM, 2004.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. Conf. on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 57–76. ACM, 2007.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 518–529. Morgan Kaufmann, 1999.

[14] S. Guhlemann, U. Petersohn, and K. Meyer-Wegener. Reducing the distance calculations when searching an M-Tree. *Datenbank-Spektrum*, 17(2):155–167, 2017.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, 1984.

[16] M. Hetland. *The Basic Principles of Metric Indexing*, pages 199–232. Springer, 2009.

[17] H. Jagadish, B. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[18] M. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15(1):9–17, 1994.

[19] S. Omohundro. Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, 1989.

[20] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[21] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond high-dimensional indexing à la carte. *Proc. VLDB Endow.*, 6(14):1654–1665, 2013.

[22] J. Schneider. Analytic performance model of a main-memory index structure. *CoRR*, abs/1609.01319, 2016.

[23] M. Schuh, T. Wylie, J. Banda, and R. Angryk. A comprehensive study of iDistance partitioning strategies for kNN queries and high-dimensional data indexing. In *Proc. British Nat'l Conf. on Databases (BNCOD)*. Springer, 2013.

[24] T. Sellis, N. Roussopoulos, and C. Faloutsos. TheR+-Tree: A dynamic index for multi-dimensional objects. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 507–518. Morgan Kaufmann, 1987.

[25] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Letters*, 40(4):175–179, 1991.

[26] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 194–205. Morgan Kaufmann, 1998.

[27] C. Yu. *High-dimensional Indexing: Transformational Approaches to High-dimensional Range and Similarity Searches*. Springer, 2002.