Information Systems 101 (2021) 101791

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Towards multi-purpose main-memory storage structures: Exploiting sub-space distance equalities in totally ordered data sets for exact knn queries

Martin Schäler[a,*], Christine Tex[b,c], Veit Köppen[d], David Broneske[e,f], Gunter Saake[f]

[a] *Salzburg University, Austria*
[b] *Karlsruhe Institute of Technology, Germany*
[c] *GridData GmbH, Germany*
[d] *Zentral- und Landesbibliothek Berlin, Germany*
[e] *German Centre for Higher Education Research and Science Studies, Germany*
[f] *University of Magdeburg, Germany*

ARTICLE INFO

Article history:
Received 15 May 2020
Received in revised form 21 March 2021
Accepted 28 April 2021
Available online 12 May 2021
Recommended by Dennis Shasha

Keywords:
Data storage structures
k-nearest neighbor queries
Main memory

ABSTRACT

Efficient knn computation for high-dimensional data is an important, yet challenging task. Today, most information systems use a column-store back-end for relational data. For such systems, multi-dimensional indexes accelerating selections are known. However, they cannot be used to accelerate knn queries. Consequently, one relies on sequential scans, specialized knn indexes, or trades result quality for speed. To avoid storing one specialized index per query type, we envision multipurpose indexes allowing to efficiently compute multiple query types. In this paper, we focus on additionally supporting knn queries as first step towards this goal. To this end, we study how to exploit total orders for accelerating knn queries based on the *sub-space distance equalities* observation. It means that non-equal points in the full space, which are projected to the same point in a sub space, have the same distance to every other point in this sub space. In case one can easily find these equalities and tune storage structures towards them, this offers two effects one can exploit to accelerate knn queries. The first effect allows pruning of point groups based on a cascade of lower bounds. The second allows to re-use previously computed sub-space distances between point groups. This results in a worst-case execution bound, which is independent of the distance function. We present knn algorithms exploiting both effects and show how to tune a storage structure already known to work well for multi-dimensional selections. Our investigations reveal that the effects are robust to increasing, e.g., the dimensionality, suggesting generally good knn performance. Comparing our knn algorithms to well-known competitors reveals large performance improvements up to one order of magnitude. Furthermore, the algorithms deliver at least comparable performance as the next fastest competitor suggesting that the algorithms are only marginally affected by the curse of dimensionality.

1. Introduction

In the last decade, main-memory database systems have revolutionized analytical query processing of relational data. The core of such relational systems usually is a columnar main-memory database system. Classically, analytical query processing in relational databases is a synonym for OLAP. However, present-day data analytics involves knowledge extraction from emerging applications, like scientific databases or time series analytics. One prominent example query in this context is computing the k-nearest neighbors (knn) for a high-dimensional data set [1–3].

To efficiently execute knn queries, one currently has three options. First, one can rely on main-memory optimized sequential scans. Hence, one exploits advances in hardware reducing the cost for vising *all* points by orders of magnitude compared to hard-disk environments. This makes sequential scans a powerful competitor. Second, to avoid scanning all points in the data set – complementing main memory storage – a wide range of indexing techniques is known. State-of-the-art approaches, such as iDistance [4], map each point in the data set to its nearest pivot(s) to apply lower-bound-based pruning [5]. Comparative studies [4,6] report large performance increases for various use cases. However, due to the curse of dimensionality, distances between high-dimensional points tend to be very similar reducing the efficiency of pruning [7]. Therefore, for realistic data dimensionality as found, e.g., in the UCI archive [8], such approaches

[*] Corresponding author.
E-mail address: martin.schaeler@sbg.ac.at (M. Schäler).

https://doi.org/10.1016/j.is.2021.101791
0306-4379/© 2021 Published by Elsevier Ltd.

visit a large fraction of the data. Ultimately, they even deteriorate to a sequential scan. Third, in case dimensionality is high, the only approach avoiding the deterioration problem is result approximation [9], i.e., trading result accuracy to improve response time. For instance, the group of Locality Sensitive Hashing (LSH) [10] yields several approaches that deliver good response times and result quality. However, result approximation may not be valid for all use cases.

Objective. To allow for a new option, we study whether novel storage structure layouts, which are already known to work well for multi-dimensional selections, can be exploited to efficiently compute exact knn queries achieving comparable performance as specialized approaches. In the long run, we aim at efficiently computing arbitrary columnar analytical queries as well as knn queries on the same underlying storage structure. That is, one shall not require one additional index per query type storing the data redundantly and increasing system complexity. We call such storage structures multi-purpose storage structures.

A promising candidate for such a storage structure is the Elf approach [11,12]. It shows good performance for queries with columnar access pattern compared to state-of-the-art competitors. In addition, the benefit remains observable when integrated into a full-fledged present-day system, such as MonetDB [13].

To efficiently support exact knn queries, we show how to exploit and tune the *sub-space distance equality* concept naturally occurring when introducing a total order into a data set D. The concept is based on the observation that all points having the same prefix (combination of values), result in the same point when projected to this prefix. To this end, in the subspace of the prefix, they have the same distance to (the projection of) any query point.

The concept implies two effects for accelerating knn queries. We name the first effect *group* lower bound effect. We know already that using prefix distances can effectively bound the distance of individual points [14]. Our core novelty is, relying on Elf, we can easily identify *all* points having the same prefix and tune Elf creation by selecting a good dimension order for a specific data set. This way, the bound generally refers to *multiple points* resulting in the group lower bound effect. The second effect is the re-use of sub-space distances. This effect holds for iteratively computable distance functions (see Definition 2.3) which include, e.g., the Minkowski metric family. There, we compute the distance for each unique prefix only once and, thus, bound the number of required distance function computations independently of the used distance function.

Contributions. Our contributions are:

1. We develop knn algorithms exploiting sub-space distance equalities for knn computation on Elf and present how to tune Elf for a data set D.
2. We show that, for iteratively computable distance functions, including all Minkowski metrics – for the first time for exact approaches [15] – there is a worst-case bound for the number of computed distances smaller than $\mathcal{O}(|D|)$. The bound is independent of the distance function and it only depends on the data set.
3. We examine the influence of dimensionality, cardinality, and distance function on the expected number of distances revealing a predictable and robust behavior of the developed knn algorithms primarily influenced by the dimensionality of the data set.
4. Examining the run time of our algorithms results in competitive performance in any case considering highly potent competitors, including iDistance [4]. We prove that the reason is exploiting the group lower-bound pruning *and* re-use of sub-space distances. Thus, even for high-dimensional data sets, where lower bounding fails due to the curse of dimensionality, we still can exploit the second effect.

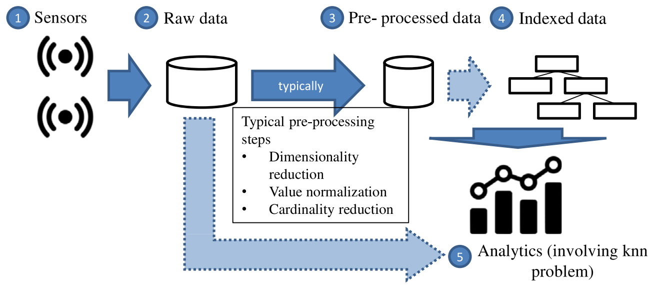


Fig. 1. General data processing pipeline.

The structure of the paper is as follows: Section 2 contains basic assumptions and introduces our notation. In Section 3, we introduce the effects resulting from sub-space distance equalities. In Section 4, we propose knn algorithms for Elf, show how to tune Elf for exploiting the effects, and discuss the worst-case execution bound. Section 5, evaluates our knn algorithms with a systematic intrinsic evaluation as well as a comparison against well-known competitors. Finally, to support repeatability, all data sets and implementations are available open source.[1]

2. Preliminaries and scope

Generally, the way from initial acquisition of the raw data to the desired data analysis results involves a complex processing pipeline with various steps. To this end, we first state the scope of this paper, which is how our solution is involved in this process. Second, we define the knn problem, i.e., the analysis we focus on.

2.1. Scope and assumptions

In the following, we explain the assumptions for the experiments in the remainder, which illustrates the scope of the paper. All assumptions aim at supporting the general processing pipeline of the knowledge discovery process (KDD) [16] depicted in Fig. 1. Typically, the raw data collected, e.g., from some sensors, is not directly used for the analysis. Instead, a pre-processing is recommended [17]. The actual pre-processing is use-case dependent, but typically includes: (i) dimensionality reduction, (ii) value normalization, and (iii) cardinality reduction.

While generally, the knn algorithms presented in this paper are applicable to raw data, we presume that the knn are not computed for the raw data. Specifically, we assume that some combination of dimensionality reduction technique, value normalization, and cardinality reduction on the raw data has been conducted. Giving credit to the vast number of possible pre-processing approaches, we only describe the intended effect on the data set and do not presume that a specific approach is used. This helps to define, e.g., what data we consider for evaluation purposes. Please mind that pre-processing may affect the computing of subsequent standard relational operations (e.g., selections and aggregations). This means that those operators need to be *rewritten*, such that they can be directly executed on pre-processed data. This is a well-known technique, which however does not exist for every possible combination of pre-processing steps as one requires a bijective mapping from raw to pre-processed data.

[1] Source code and data available at www.elf.ovgu.de/KNN.html.

Dimensionality reduction. Frequently, one conducts some dimensionality reduction. The reduction could be done, via Principal Component Analysis (PCA) [18], Piecewise Aggregate Approximation (PAA) [17], Locality Sensitive Hashing (LSH) [10], or Neural Networks (e.g., word embeddings [19]). The objective is to eliminate correlated dimensions or to reduce the data set size. In context of this paper, the assumption that dimensionality reduction is performed helps selecting data sets with meaningful dimensionality in the experiments.

Value normalization. There are various ways to normalize data to fit a desired distribution. A well-known example is Z-normalization to fit a Gaussian distribution. To this end, we focus on data sets having a known distribution (e.g., Uniform), as well as data sets where the distribution is not known. The latter ones are summarized under the term real-world data sets.

Cardinality reduction. To reduce data volume and cope with noise, one often conducts a cardinality reduction. That is, one reduces the number of possible values of dimension i to c_i possible integer values or class labels [10]. Here, c_i is a user-defined arbitrarily large granularity, which can also be used as weighting factor. In this paper, we assume that c_i is the same for all dimensions, a common assumption [10] imposing no restriction.

2.2. The knn problem

In the remainder, we refer to approaches involving the knn problem. We subsequently define this problem and state classes of distance functions, we focus on. We give an overview of our notation in Table 1.

2.2.1. Definition of the knn problem

Given a set of points in a d-dimensional space, a knn query returns the k closest points to a query point according to a distance function.

Definition 2.1 (*Knn Problem*)**.** For a query point $q \in \mathbb{N}^d$, a distance function $dist()$, a set of points D from $\mathbb{N}^d$, and $k \in \mathbb{N}^*$, the knn query $knn(q, dist(), D, k)$ returns the smallest set $S \subseteq D$ that contains k points, and for which the following condition holds:

$$\forall p \in S, \forall p' \in D \setminus S : dist(q, p) < dist(q, p').$$

Notably, the definition above means that there are alternative functions than $dist()$ for computing the knn. There may be some function $dist'()$ that is order preserving according to $dist()$, but faster to compute. An example is using the squared Euclidean distance instead of the Euclidean distance to avoid computing square roots.

For lower bound pruning, one utilizes the maximum distance of the points forming the current result set S to the query point.

Definition 2.2 (*Maximum Distance to Query Point*)**.** For a query point $q \in \mathbb{N}^d$ and a result set $S \subseteq D$ of a knn query, the maximum distance of any point $s \in S$ to the query point q is defined by $max\_dist(S, q) := \max_{p \in S} dist(q, p)$ realized by $arg\_max\_dist(S, q) := \arg\max_{p \in S} dist(q, p)$.

2.2.2. Classes of distance functions in the focus

For computing the knn, relying on sub-space distance equalities, one can generally use an arbitrary distance function $dist()$. However, to allow for efficient computation, one usually focuses on a class of distance functions, such as metrics, or even on a single distance function. The rational is that one can exploit their features to speed up computation. In the remainder, we exploit the two mentioned effects, and therefore focus on distance functions allowing to exploit them. While the first effect holds for all norms [20], the second one requires an *iteratively computable distance function*. We introduce such distance functions next. For explanatory reasons, we initially introduce the definition of iterative distance functions. Then, we extend this definition to iteratively computable ones.

Table 1
Overview of notation.

Notation	Semantics
D	A data set from $\mathbb{N}^d$
d	Dimensionality of the data set
q	Query point $q \in \mathbb{N}^d$
p	A point $p \in \mathbb{N}^d$
$p[i]$	The ith value (i.e., dimension) of p
k	Number of points to return
S	Result set of a knn query
$dist(p_1, p_2)$	Function returning the distance between points p_1 and p_2, abbreviated by $dist()$
$f_i(p[i], q[i])$	Local distance function for dimension i
$max\_dist(S, q)$	Function returning maximum *distance* of any point $s \in S$ to query point q
$arg\_max\_dist(S, q)$	Function returning the *point* having largest distance in S to query point q
$pre_u(p)$	Projection of p to its first u dimensions

Iterative distance function. A distance function $dist()$ is *iterative*, if it can be computed as sum of the dimension-wise distances. This is formalized in Definition 2.3. It states that there is a local distance function f_i for every dimension i computing the respective dimension-wise distance. That way, we can consider data sets having numerical *and* categorical attributes. As usual, we assume that f_i returns non-negative distances.

Definition 2.3 (*Iterative Distance Function*)**.** A function $dist()$ is iterative, iff there exists a set of functions f_i such that $\forall(p, q) \in N^d$ holds that

$$dist(p, q) = \sum_{i=1}^{d} f_i(p[i], q[i]).$$

For illustration, consider a data set having several numerical and one categorical dimension. The distance between two points p and q shall mimic the squared Euclidean distance. To this end, one sums up the *dimension-wise distance* in all numerical dimensions computed as $f_i(p[i], q[i]) = (p[i] - q[i])^2$. Then, one adds the distance of the categorical dimension computed by the corresponding local distance function f_i. The simplest case of such a function is a matrix containing all pair-wise distances.

Iteratively computable distance functions. Some common distance functions, like the Euclidean distance, are not iterative. The reason is that, after summing up the dimension-wise distances (i.e., after iterating), one has to additionally compute the square root. However, as stated for the knn problem, to compute the knn it is sufficient to have a function $dist'()$ that is order preserving according to $dist()$. As a result, we say that a distance function is *iteratively computable*, if there is an iterative function that is order-preserving towards $dist()$, which is formalized in Definition 2.4. The definition covers, for instance, all Minkowski metrics, like Euclidean distance, including their weighted variants, Hamming distance, and, after normalizing vector (i.e., point) length, also for cosine distance.

Definition 2.4 (*Iteratively Computable Distance Function*)**.** A function $dist()$ is iteratively computable, iff there is an iterative distance function $dist'()$ that is order preserving towards $dist()$.

$pre_2 = (0,1)$

dim_1	dim_2	dim_3	dim_4	TID
0	1	1	1	T_1
0	1	1	1	T_5
0	1	2	2	T_4
0	2	0	0	T_3
1	0	0	1	T_2

Fig. 2. Running example data. The data is sorted.

3. Knn computation with sub-space distance equalities

In this section, we firstly introduce the concept of sub-space distance equalities and secondly explain two effects allowing to exploit it for efficient knn computation. Moreover, we differentiate the concept from related ones.

3.1. Sub-space distance equalities

To explain the concept of sub-space distance equality, consider a d-dimensional data set, with $d > 2$ and sort it to some dimension order. For simplicity, assume we sort according to $dim_1, \ldots, dim_d$. As a result, all points having the same value in the first dimension (dim_1) are located next to each other. Observably, that also holds 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. Intuitively, 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 query point projected to that sub space. This is what we call a sub-space distance equality.

For illustration, consider the (sorted) example data set in Fig. 2. We observe that points with the TIDs T_1, T_5 and T_4 have the same prefix until dimension dim_2, namely, $pre_2 = (0, 1)$. Consequently, in the two-dimensional sub space consisting only of the first two dimensions, those points are projected to the *same* point $p = (0, 1)$. To this end, they also have the same distance to any query point q: $dist(pre_2(p), pre_2(q)) = \sum_{i=1}^{2} f_i(p[i], q[i])$.

3.2. Two effects for knn computation

Using sub-space distance equalities, we can exploit two effects to accelerate knn computation. The effects are the group lower bound effect and the re-use effect. Subsequently, we introduce both effects giving an intuition how to exploit them and for what distance functions they hold.

Effect 1: Group lower bound effect. Let pre_u be a projection of an arbitrary d-dimensional point to its first u dimensions. Further, let q be a d-dimensional query point. Then, p can only be a query answer (i.e., $p \in S$) if $dist(pre_u(p), pre_u(q)) \leq max\_dist(S, q)$ holds. That is, the distance of pre_u to the corresponding projection of q (sub-space distance) is a lower bound of the full-space distance of p and q. This is the common lower bound effect. However, this lower bound additionally holds for all points having the same prefix as p. Recapitulate that after sorting, all points having the same prefix are located next to each other. Thus, it is easy to prune *all* points whose lower bound exceeds the full-space distance of the currently found kth nearest neighbor. The effect holds for any norm used as distance function due to orthogonality [20].

Effect 2: Re-use effect of sub-space distances. If we use an iteratively computable distance function, we can re-use the previously computed sub-space distance in two ways. For explanation, assume that we already computed the sub-space distance of p to q having the same prefix pre_{l-1}. Firstly, for a different point p' having the same prefix pre_{l-1}, we can also re-use this distance, as it is the same. The second re-use opportunity is as follows. Assume that we aim at computing the sub-space distance from p to q and p' to q in the (next larger) sub space defined by pre_l. To compute the distance for p, we only need to add $f_l(p[l], q[l])$ to the already computed distance. Even in case the dimension values of p and p' are *not* the same in dimension l, we can still re-use the sub-space distance from pre_{l-1}, by simply adding the result of $f_l(p'[l], q[l])$.

3.3. Related work to sub-space distance equalities

We now compare accelerating knn computation with sub-space distance equalities to knn approaches from the literature. Generally, there is a large variety of different knn approaches. Therefore, we do not focus on specific approaches, but on underlying concepts. Nevertheless, for each concept, we name prominent approaches. For each concept, we integrate at least one approach into the evaluation.

Classic indexing approaches. Approaches like R-Tree [21] or kd-Tree [22] have given way to various improvements including, e.g., [23–25]. There are large surveys comparing them comprehensively, like [1–3]. All classical indexing approaches enclose a set of points using a hierarchy of geometric forms, like minimum bounding rectangles. Those forms allow to compute a lower bound for points contained. To this end, they apply lower-bound-based pruning of point groups. However, one cannot re-use previously computed distances. Moreover, the geometric forms are defined on the full space tending to overlap each other even at the leaf level. Therefore, they tend to *degrade* to a sequential scan for high-dimensional data [26].

Optimized sequential scans. As a solution to the degradation problem of classical indexing techniques, optimized sequential scans are proposed. An early variant is the VA-File [26]. Its core idea is using a compressed representation of the data fitting into main-memory allowing to compute a lower bound per point. The data itself resides on hard disk being multiple orders of magnitudes slower than main memory. However, with increased main-memory capacities, usually the whole data set fits into main memory. Nevertheless, for computationally expensive distance functions, such as set similarity, dynamic time warping (DTW), or string as well as tree-edit distance, optimized sequential scans are state-of-the-art [27]. Corresponding approaches apply multiple techniques to avoid computing the full-space distance of two points. To illustrate, the UCR suite [8] – pioneering the combination of techniques – first applies a cascade of lower bounds with increasing tightness (ordered by increasing computational complexity). Second, upon distance computation, the UCR suite abandons the distance computation if the prefix distance exceeds $max\_dist(S, q)$, i.e., the prefix distance serves as lower bound. Finally, in case the data needs to be normalized upon knn query execution, the UCR suite re-orders the values to provoke large prefix distances for short prefixes. In contrast to the concept of sub-space distance equalities, the bounds refer to individual points. Research in this area aims at grouping points based on *similar* bounds [28,29]. Then, the (less tight) bounds refer to multiple points. However, since the prefixes of points in a group is not guaranteed to be the same, one cannot re-use previously computed distances.

Metric indexing with pivots. The AESA approach introduced the idea of pivot-based indexing [30] performing a lower-bound pruning based on the triangle inequality. Generally, the idea is to determine several pivot points. Then, one maps any point in the data set to its nearest pivot(s). Due to these mappings, the triangle in-equality between query point, pivot, and data

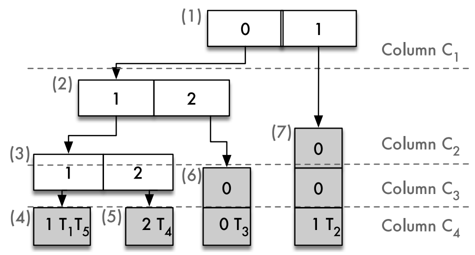


Fig. 3. Conceptual Elf for the running example data.

point can be used for computing a lower bound of the distance between query point and data point using any metric [6,31]. Thus, lower-bound computation is a simple addition and subtraction of pre-computed values. In contrast to exploiting sub-space distance equalities, one cannot re-use previously computed distances, iteratively tighten the bound when dimensionality (i.e., length of the prefix) increases.

Various approaches rely on metric indexing with pivots. A particularly relevant approach is iDistance [4]. It assigns one-dimensional indexes to the pivot-point mappings. Based on the one-dimensional index, it can be used for data on hard disk or in main memory [32,33]. By concept, also revealed in comparisons like [5], for high-dimensional data one still faces the challenge that one tends to scan the entire data set, as the expected size of lower bound becomes too large to prune points. To avoid this Ptolemaic indexing [34] has been proposed. It features tighter bounds than metric indexing and is applicable to various well-known distance functions. However, it is not as general as metric indexing and requires more computational effort.

Approximation. Metric indexing approaches increase the dimensionality for that an indexing approach generally is faster than a sequential scan compared to classical indexing approaches. However, ultimately, if dimensionality is high enough, any of those approaches still converges towards scanning the entire data set [7,15]. A common solution avoiding this problem is to embed the data into a lower dimensional space approximately keeping the point-wise distances, optionally satisfying some correctness bound. Examples for approaches guaranteeing a bound are [17,35]. Any of these approaches are applications of the Johnson–Lindenstrauss lemma [36]. It states that one can embed points from a high-dimensional (Euclidean) space into a lower dimensional one by approximately keeping the distances between them. Thus, in the end, they conduct a dimensionality reduction, which is one of the suggested pre-processing steps.

4. Knn computation with Elf

We use the Elf approach [11] for exploiting sub-space distance equalities. Still, neither the concept of sub-space distance equalities itself, nor the corresponding effects are specific to Elf. What is specific to Elf, being the reason why we use this approach, is that it is particularly easy to find all points having sub-space distance equalities. To this end, we first introduce the Elf approach in Section 4.1 and then develop efficient knn algorithms for Elf (Section 4.2). Furthermore, we theoretically analyze the maximum number of computed distances per knn query (Section 4.3) showing that we can set up a distance-function independent bound. Finally, we show how to tune a specific Elf by selecting a good dimension order.

4.1. Design of the Elf approach

Originally, trie-like Elf is proposed to accelerate OLAP queries focusing on efficiently evaluating multi-column selection predicates. We illustrate the design of Elf relying again on the four-dimensional data set from Fig. 2. Conceptually, Elf incrementally indexes existing dimension values in sub spaces. That is, the first level in Elf refers *only* to the first dimension. The second level refers to the first *and* the second dimension etc. Each node of Elf, called `DimensionList`, contains entries of the form $(value, pointer)$ (i.e., dimension value and pointer to the next deeper tree level) ordered according to *value*. This means, the root level of the tree contains every unique value of the first dimension — for instance, the values 0 and 1 of dim_1 in the Elf in Fig. 3. Each entry in the root is the start of one *path*.

There is an optimization, named `MonoLists`, to reduce storage cost and improve data locality [11]. The idea is as follows: Whenever one encounters a node containing only one point (i.e., the path is unique), the build algorithm creates a `MonoList`. That is, one stores all remaining dimension values (without the need for additional pointers) next to each other, similar to a row store.

Optimized memory layout

It is possible to build Elf with a main-memory optimized storage layout [12] to improve its cache sensitivity (i.e., exploit data locality). To this end, one linearizes the conceptual Elf – explained above – into an array using a preorder traversal. This offers the possibility to investigate the effect of sub-space distance equalities on response time in isolation and in concert with an optimized storage layout. This is important as row-wise sequential scans highly take advantage of data locality.

4.2. Knn algorithms for Elf

We now introduce two knn algorithms exploiting sub-space distance equalities within Elf. The algorithms are optimized for different data distributions. The first algorithm targets at fast convergence of $max\_dist(S, q)$, relevant, e.g., for highly clustered data sets. By contrast, the second algorithm aims at data sets that deteriorate towards scanning an entire Elf. Thus, it aims at optimally exploiting the data layout. We now explain both algorithms separately and relate them to knn algorithms of similar approaches.

4.2.1. Knn algorithm optimizing pruning power

The first knn algorithm `knnConverge` (cf. Algorithm 1) aims at greedily traversing Elf, such that $max\_dist(S, q)$ of the collected points in the intermediate result S converges fast against the distance of the true kth nearest neighbors. To this end, Elf is traversed in a greedy manner starting by invoking `knnConverge` for the first dimension list (i.e., the root). In each `DimList`, the algorithm searches for the entry (best match) having minimum distance to the query q. In case the corresponding sub tree can contain at least one query solution (i.e., its sub-space distance is smaller than $max\_dist(S, q)$), we first recursively invoke `knnConverge` on the root of this sub tree. After having examined the entire sub-tree, the algorithm examines the remaining entries in the current dimension list iteratively. That is, in the first iteration one entry to the left *and* to the right of best match element are examined. Since the $(val, pointer)$ elements are ordered according to val starting from the best match searching inwards and outwards, the sub-space distances, i.e., bounds, monotonically increase. Thus, the algorithm safely stops searching a direction (e.g., to the right, see Line 20) after having found the first element with $subspaceDist + f_{dim}(val, q[dim]) \geq max\_dist(S, q)$.

Algorithm 1 knnConverge(q, k, dim, DimList, $subspaceDist$, S)

```
Input q: query point, $k \in \mathbb{N}^*$, dim: start dimension of the monolist,
      DimList: node in Elf containing $(val, pointer)$ elements.
      subspaceDist: distance of $pre_{dim-1}(p_{tid})$ to $pre_{dim-1}(q)$,
      S: knn result set so far.
Output  S: possibly adjusted knn result set

1:                   ▷ find element having minimum distance named best match
2: $(val, pntr) \leftarrow$ argmin$\{f_{dim}(val, q[dim]) \mid (val, pntr) \in$ DimList$\}$
3: $dist \leftarrow subspaceDist + f_{dim}(val, q[dim])$
4: if $dist \geq max\_dist(S, q)$ then
5:     return $S$                              ▷ no solutions in this node at all
6: end if
7:                                   ▷ First, search sub tree below best match
8: if $isMonoList(pntr)$ then                                ▷ how to descend?
9:     $S \leftarrow$ knnElfMono($q, k, dim+1, pntr, dist, S$)
10: else
11:     $S \leftarrow$ knnConverge($q, k, dim+1, pntr, dist, S$)
12: end if
13:                            ▷ Second, search in- and outwards of best match
14: $done\_right \leftarrow false$, $done\_left \leftarrow false$
15: while not ($done\_right$ and $done\_left$) do
16:     if not $done\_right$ then
17:         $(val, pntr) \leftarrow$ next element to the right
18:         $dist \leftarrow subspaceDist + f_{dim}(val, q[dim])$
19:         if $dist \geq max\_dist(S, q)$ then
20:             $done\_right \leftarrow true$          ▷ stop searching to this direction
21:         else
22:             descend as for best match
23:         end if
24:         if reached last element then
25:             $done\_right \leftarrow true$
26:         end if
27:     end if
28:     analogous treatment of the next element to the left
29: end while
```

As we can see, the algorithm stops in case there are no more solutions to the left and right. Hence, the algorithms aims at avoiding to examine the whole node. This is similar to the knn algorithm of iDistance, but our bound (i.e., the sub-space distance) results in the true full-space distances (leaf level) and the bound refers to multiple points.

The difference to other tree-based knn algorithms is that we can exploit the re-use effect by *adding* the distance in the current dimension to the sub-space distance computed so far. By contrast, using e.g., an M-Tree, one needs to compute the full-space distance of the query point to the closest point that might exist in the corresponding sub-tree. Note, knnElfMono (cf. Algorithm 2) computes the distance for the remaining dimensions of *one* point to q. It stops computing the full distance in case the bound (i.e., the sub-space distance) exceeds the $max\_dist(S, q)$. It is the only algorithm modifying the knn result set S as in the final level all nodes in Elf are MonoLists.

4.2.2. Knn algorithm optimizing data locality

The second knn algorithm (cf. Algorithm 3) is optimized for data sets, where pruning, for instance due to the curse of dimensionality, is difficult. To this end, the only difference to the first algorithm is that it does not search for the best match in each dimension list. Instead, it strictly follows the data layout of Elf by executing a pre-order traversal of the tree. Since the linearization of the tree uses the same traversal, one optimizes data locality. As a result, this algorithm is more similar to a sequential scan which benefits primarily from data locality. However, we still benefit from re-using sub-space distance equalities reducing the number of distance computations and we also check whether we can prune sub trees referring to groups of points. The algorithm ensures correctness, as we only skip sub trees whose sub-space distance exceeds the full-space distance of $max\_dist(S, q)$ (Line 5).

Algorithm 2 knnElfMono(q, k, dim, $monolist$, $subspaceDist$, S)

```
Input & Output: Identical to Algo. 1 except
  MonoList: array containing remaining $d - dim$ values and $tid$.

1: $tid \leftarrow$ MonoList$[d - dim + 1]$
2: $dist \leftarrow subspaceDist$
3:              ▷ $dist$ contains full-space distance $dist(p_{tid}, q)$ after entire loop
4: $d' \leftarrow dim$
5: while $d' \leq d$ do
6:     $dist$+=$f_{d'}$(MonoList$[d' - dim]$, $q[d']$)
7:     if $dist \geq max\_dist(S, q)$ then
8:         return $S$                        ▷ $p_{tid}$ is no result, keep $S$ as it is
9:     end if
10:     $d' \leftarrow d' + 1$
11: end while
12: remove tuple $(p, dist)$ referring to $arg\_max\_dist(S, q)$ from $S$
13: add $(p_{tid}, dist)$ to $S$
14:                  ▷ implicitly adjusts $max\_dist(S, q)$ and $arg\_max\_dist(S, q)$
15: return $S$
```

4.3. Theoretic worst-case distance computation bound

Next, we examine the worst-case bound for the number of distance computations. To this end, we first discuss bounds for existing exact approaches, and why the bound is smaller for Elf. Then, we define a measure quantifying the size of the bound, which is independent of the distance function. Finally, we give further intuitions on the semantics and size of the bound.

Algorithm 3 knnOptLayout(q, k, dim,DimList, $subspaceDist$, S)

```
Input & Output: see Algo. 1

1: while not end of DimList reached do
2:     $(val, pointer) \leftarrow$ next element in DimList
3:     $dist \leftarrow subspaceDist + f_{dim}(val, q[dim])$
4:     if $dist \geq max\_dist(S, q)$ then
5:         go to Line 1                          ▷ no solutions in this sub tree
6:     end if
7:     if $isMonoList(pointer)$ then
8:         $S \leftarrow$ knnElfMono($q, k, dim+1, pointer, dist, S$)
9:     else
10:         $S \leftarrow$ knnOptLayout($q, k, dim+1, pointer, dist, S$)
11:     end if
12: end while
13: return $S$
```

4.3.1. Bounding distance computations of exact approaches

The most relevant factor for the performance of knn approaches is the number of pairwise distances the respective knn algorithm computes per query. Without making assumptions on data distribution, which allows to compute the expected distance of the kth nearest neighbor, the only valid upper bound for the number of computed distances for exact approaches is $\mathcal{O}(|D|)$ [15]. This, in particular, holds for R-tree and all metric-based approaches. Consequently, for these approaches, it is possible that the knn algorithm computes all distances and has the additional overhead for traversing e.g., a tree. This explains why for high-dimensional data sets a simple sequential scan usually outperforms any exact approach [7,37].

This is different for Elf, because we compute the distances sub-space wise (Effect 2) and not point-wise. This means, as long as the first dimension is no primary key, Elf contains less values than the original data set D, which has $|D| \times d$ values. Therefore, in this case, traversing an entire tree (without pruning), results in less distance computations than sequentially scanning the data set. In case the first dimension is a key, Elf contains as many dimension values as the data set, i.e., $\mathcal{O}(|D|)$ holds in any case. However, as we can select Elf's dimension order, this only occurs if *every* dimension is a key.

Naturally, traversing an entire Elf bounds the number of invocations of the local distance functions $f_i()$. In other words, with the exception of having a primary key in the first dimension, Elf compresses the data set. The compression forms an upper bound for the number of invoked distance computations, which is independent of the applied distance function only depending on the data distribution. We define a measure directly related to the upper bound of computed distances, named Elf compression factor.

4.3.2. Elf compression factor as bound

Given a fixed dimension order, we define the Elf Compression Factor such that it indicates the relative cost of a full-Elf scan compared to a full-table scan.

Definition 4.1 (*Elf Compression Factor (ECF_e^D)*)**.** For some d-dimensional data set D with dimension order e and cardinality c, the Elf Compression Factor (ECF_e^D) is defined as

$$\mathrm{ECF}_e^D = \frac{\sum_{l=1}^{d} |supp(D_l)|}{|D| \times c},$$

where D_l is the multiset of all points projected to the first l dimensions in e, and $supp(D_l)$ is the support of D_l.

According to the definition, we compute the compression factor by looping over all d tree levels computing the number of values per level. It follows directly from the definition of Elf that there are as many dimension values per level l as there are unique prefixes pre_l. We compute the number of unique prefixes by projecting the data set D to the first l dimensions resulting in the multiset D_l (containing duplicates). Then, by determining the support of D_l resulting in the *set* of all prefixes existing at least once, we compute the respective cardinality. In case that D and the dimension order is clear from the context, we abbreviate ECF_e^D with ECF.

4.3.3. Intuitions on the semantics and size of the bound

Below, we first illustrate the semantics of the bound. Then, we explain the size of the bound considering the worst-case distribution.

Bound intuitions. To exemplify the semantics of the ECF, imagine a $d = 10$ dimensional data set D consisting of 100,000 points. Then, D contains 1,000,000 ($= |D| \times d$) values. This means to compute the knn with a sequential scan, one invokes the point-wise distance function $dist()$ 100,000, i.e, $|D|$ times, which itself calls d local distance functions $f_i()$. Summarily, this results in 1,000,000 invocations of $f_i()$ – one for each value. Assuming that the corresponding Elf e stores 900,000 values this results in an ECF_e^D of 0.9.

Uncorrelated uniform data as worst case distribution. The worst-case data distribution regarding the ECF for a given cardinality c and dimensionality d is having Uniform uncorrelated data D. The reason is that, in case the data follows a Uniform distribution (and is not correlated), the branch-out in the Elf tree is maximal and computing the number of values per tree level l simplifies to $min(|D|, c^l)$. This means, for a given dimension order, there is a length u where all prefixes of this length (or longer) become unique. We therefore observe no more compression and have one value per point and dimension. For Uniform data, this is the case, when the size of the data space becomes larger than the number of points to distribute, i.e., larger than $\arg\min u$ s.t. $c^u \geq |D|$.

4.4. Tuning Elf indexes

In the following, we focus on how to tune Elf by selecting a good dimension order affecting both effects exploited for efficient knn computation. To this end, we initially specify the tuning objective. Then, we define a measure allowing to identify good dimension orders. Finally, we discuss how this indicator is furthermore used to select the best Elf knn algorithm for a given data set.

4.4.1. Intuitions on the tuning objective

The tuning idea is the same as for any tree-based approach: one wants to maximize the pruning capability of the tree to compute as less distances as possible. For Elf that means, we want to provoke large sub-space distances for short prefixes, i.e., maximize the intrinsic dimensionality [38] in the first sub-spaces by selecting a respective dimension order.

In this context, it is important to know that there are data sets, for which the dimension order has no influence, and that there are pre-processing steps that explicitly aim at producing such data. For instance, as result of removing correlated dimensions and value normalization, one can receive an uncorrelated data set, where all dimensions follow (the same) Gaussian distribution. Then, knowing the distribution, it is obvious that optimizing the dimension order is meaningless.

Nevertheless, in the end one wants to know whether the dimension order is relevant, and determine a good (i.e., fast) dimension order for Elf. Finding a good dimension order is similar to a related problem, which is finding the right sub-space to locate hidden sub-space outliers [39] as follows, meaning that we can rely on similar techniques to find the desired sub spaces.

As a direct result of the curse of dimensionality, finding anomalies (i.e., outliers) in the full space is difficult as the distances tend to be the same and converge towards the length of the space diagonal [7]. To this end, one searches for low dimensional sub spaces where it is easier to locate outliers. In opposition to our problem, the desired sub spaces have the property that the average distance between points is *small* (i.e., highly correlated), named low-contrast sub spaces. This ensures that anomalies are easy to locate. By contrast, we search for sub spaces in which the average distance, i.e., the contrast, is high. Nevertheless, we can rely on sub-space contrast indicator as used, e.g., in [39], by simply reversing the order.

4.4.2. Sub-space contrast indicator

To quantify the expected pruning capability of a specific dimension order, we use the sub space contrast indicator (SSC). It is defined for a prefix p of length u. Definition 4.2 states that SSC_u^p is the average distance of all point pairs. For larger data sets, we use a sample. 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).

Definition 4.2 (*Sub-Space Contrast Indicator*)**.** The Sub-Space Contrast indicator (SSC_u^p) denotes the average distance according to $f_i()$ between two points in D for the first u-dimensional sub space of a given dimension order p.

The best dimension order is the one having the highest SCC. However, computing all SSC_u^p for all possible dimension orders is computationally expensive and practically impossible for large number of dimensions. To this end, we propose to use an estimation. The algorithm `local` (cf. Algorithm 4) aims at fast identification of a fast dimension order, but does not take correlated dimensions into account. It works on the computed SSC_1^p values, i.e., the average distance per point for each value in isolation. Having the SSC value of each dimension, the dimensions are sorted putting the one with the largest average distance first. The resulting order directly forms the dimension order.

Algorithm 4 Algorithm `local`(D)

```
Input D data set Output  dimension order as list of attributes
1: initialize List L as empty list
2: for each dimension d' in D do
3:     compute SSC                          ▷ contrast for this dimension only
4:     insert (SSC, d') into L
5: end for
6: sort L descending                        ▷ according to SSC values
7: return L                                 ▷ containing the order
```

4.4.3. Elf knn algorithm selection

Selecting a good dimension order is relevant for building Elf. Furthermore, to execute a knn query, we need to select one of the two introduced knn algorithms. To this end, one needs to know whether optimizing the dimension order is expected to significantly improve pruning capability. This means, in case the expected difference between points is large, greedily traversing the tree with algorithm `knnConverge`, descending into the subtree having the smallest distance to the query point, is more promising. Otherwise, we use knn algorithm `knnOptLayout` To this end, we rely on sorted SSC values per attribute produced by Algorithm `local` The intuition is, in case the difference between the best and the worst order is large, the tuning is meaningful and we apply the found dimension order in concert with knn algorithm `knnConverge`

5. Evaluation

The evaluation comprises an intrinsic and extrinsic part. We initially give details on the experimental design relevant for both parts. Then, we present the results of the single parts. Finally, we summarize and discuss the primary result of the evaluation.

5.1. Experimental design

Below, we firstly state the objectives of the intrinsic and extrinsic part of the evaluation. Secondly, we introduce and justify the data sets used in each part. Finally, we explain how we ensure soundness of our experiments.

Objectives of intrinsic and extrinsic part. The *intrinsic* part systemically examines how exploiting sub-space distance equalities affects knn query performance. To this end, it consists of two investigations. Firstly, we aim at getting an understanding how different parameters, like dimensionality and cardinality of the data set, as well as the used distance function, affect knn query performance. Secondly, we examine the strength of each of the two exploited effects to accelerate knn computation in isolation.

In the *extrinsic* part, we compare the Elf approach to well-known exact knn competitors. We do this for four different use cases, each represented by a real-world data set and a corresponding artificial counterpart as stated below. This experiment answers the question whether one can generally expect competitive performance using the Elf approach. In addition, we compare the Elf approach to approaches using approximation.

Data sets. Selecting data sets allowing general conclusions is a challenging issue. This is attributed to the variety of parameters affecting knn performance, as mentioned in the objective of the intrinsic part. Recapitulate that we presume that the data is pre-processed (cf. Section 2.1) affecting dimensionality d of the data set. This allows to give meaningful bounds for the data sets, such that we cover a large fraction of the data sets, which are commonly analyzed. Considering, e.g., the well-known UCI machine learning repository [40], reveals that the majority of data sets have dimensionality $d \leq 256$. This motivates this selection. The data sets in the UCI archive furthermore suggest that majority of data set sizes reach from roughly 100,000 points to several millions.

In the intrinsic part, we use systematically created artificial data sets for each combination of dimensionality $d \in \{16, 32, 64, 128, 256\}$, cardinality $c \in \{128, 256, 1024, 2048, 4096, 8192\}$ and multivariate data distribution $P \in$ {Uniform, Gaussian, Zipf} all having 1,000,000 points. This allows to quantify the influence of either parameter on knn query performance. We use artificial instead of real-world data, because Uniform data represents a worse-case scenario for Elf. In addition, there are pre-processing approaches that target explicitly at producing such data.

In the extrinsic part, we rely on four real-world data sets corresponding to different use cases (cf. Table 2). We use three data sets from [37], as well as one data set from the KDD machine learning challenge (cf. Table 2). With, e.g., D_r^{51}, we abbreviate the 51-dimensional real-world data set. Altogether, the number of dimensions ranges from 43 to 128 and the size ranges from 411,961 points to 11 million points. As common for real-world data, we expect that their intrinsic dimensionality is smaller than the dimensionality of the data set [38]. This means that most approaches should work better than for a comparable data set having the same size, cardinality, and dimensionality, but follows e.g., a Gaussian or Uniform distribution. To illustrate this in our experiments, for each real-world data set, there is an artificial counterpart with same number of points, cardinality, and dimensionality, but containing Gaussian data. With D_G^d, we refer to the d-dimensional artificial counterpart following a Gaussian distribution.

Statistical soundness. We ensure statistical soundness and repeatability, as follows. In each experiment, we execute batches of 1000 randomly selected knn queries. Furthermore, the evaluation is performed on an Intel Core i5 with 2.6 GHz clock frequency having 20 GB RAM ensuring that all data sets and approaches are kept in main memory. We use Java 8 following the benchmarking guidelines from [41]. Additionally, we tested implementation variants for each competitor, selecting the fastest one and repeated the experiments on different hardware all resulting in the same findings. Concerning Elf, we apply knn algorithm `knnOptLayout` for artificial data and `knnConverge` for real-world data sets as discussed in Section 4.4.

5.2. Intrinsic evaluation

In the following, we aim at examining how exploiting sub-space distance equalities affects knn performance in a systematic way. To this end, we examine (1) the influence of different parameters including k and the used distance function on the Elf's knn query performance, and (2) the strength of the two effects in isolation.

5.2.1. Parameter influence analysis

Investigations on parameter k. We first investigate the influence of parameter k determining the number of neighbors to return. To this end, we experimentally determine the number of invocation of the local distance function ($\#f$) per knn query for different

Table 2
Overview of real-world data sets used in extrinsic part.

id	d	$\|D\|$	use case
D_r^{43}	43	411,961	Spectral features [37]
D_r^{50}	50	129,597	Particle experiment [37]
D_r^{51}	51	3,446,019	Physical activity [37]
D_r^{128}	128	11,164,811	SIFT image features [40]

Table 3
Influence of parameter k: $\#f^n$ for $k = 2$ and $k = 50$ for Gaussian data.

k=2

Card/Dim	16	32	64	128	256
128	0.082	0.243	0.424	0.560	0.685
256	0.103	0.259	0.432	0.563	0.687
512	0.129	0.276	0.439	0.565	0.688
1024	0.140	0.283	0.442	0.566	0.689
2048	0.157	0.292	0.447	0.569	0.690
4096	0.174	0.301	0.452	0.571	0.691
8192	0.178	0.301	0.453	0.575	0.691

1 $\#f^n$ 0

k=50

Card/Dim	16	32	64	128	256
128	0.152	0.339	0.513	0.633	0.741
256	0.179	0.356	0.521	0.637	0.743
512	0.210	0.371	0.527	0.639	0.744
1024	0.222	0.378	0.530	0.641	0.745
2048	0.240	0.387	0.535	0.643	0.746
4096	0.258	0.397	0.540	0.645	0.747
8192	0.260	0.399	0.542	0.647	0.748

values of k, specifically $k \in \{2, 5, 10, 20, 50\}$. For better interpretability, we normalize the respective number by dividing it by the number of f_i invocations of a sequential scan resulting in the $\#f^n$ measure. That is, we examine the fraction of distances computed by Elf compared to a sequential scan. A $\#f^n$ value close to 1.0 indicates that nearly all points are examined. In Table 3, we depict the $\#f^n$ values using Euclidean metric for $k = 2$ and $k = 50$, i.e., the smallest and largest tested value for the Gaussian distributed artificial data sets. Similar results hold for Uniform and Zipf distributed data.

As expected, we observe that the $\#f^n$ value slightly increases when the value of k increases. Examining the values in detail, the results reveal that increasing the dimensionality has a significantly stronger effect than increasing the value for k. In particular, we observe the largest $\#f^n$ increase for the most densely populated data set ($c = 128$, $d = 16$). With $\#f^n = 0.082$ for $k = 2$ and $\#f^n = 0.152$ for $k = 50$ the difference is smaller than Factor 2. However, doubling the dimensionality (from $d = 16$ to $d = 32$), we observe for $k = 50$ that the corresponding $\#f^n$ already differs by more than Factor 2. To this end, we argue that the influence of parameter k is small. Thus, it is valid to display only the results for $k = 10$ in the remaining experiments, as commonly done in literature [4].

Influence of the distance function. Next, we examine on the influence of different distance functions. As distance functions, we consider *all* distance functions used in large-scale surveys [2,3]. The functions are three Minkowski metrics, namely, Manhattan, Euclidean, and Chebyshev (max norm), as well as Cosine distance. In the evaluation, we again determine the average number of computed distances per knn query of Elf relative to sequential scan ($\#f^n$) for different cardinality c, dimensionality d, when using these four different distance functions.

In Fig. 4.(a), we depict the resulting $\#f^n$ values for Gaussian data sets only, because the $\#f^n$ values for Uniform and Zipf are similar. Comparing the $\#f^n$ values of all distance functions, we observe that the result patterns are quite different. That means that the $\#f^n$ value highly depends on the used distance function. The most similar patterns are observed for the Euclidean and Manhattan metric being an inclined plane.

Comparing the results for the Euclidean and Manhattan metric in detail reveals that the results for the Euclidean metric are consistently better than those for the Manhattan metric. For instance, for the largest measured $\#f^n$ at $c = 8,192$ and $d = 256$, the $\#f^n$ value using Euclidean distance is 0.732 and 0.831 respectively for the Manhattan metric. We attribute the difference to the way the Elf knn algorithms work. In detail, it is easier to find points having a small distance to the query point q (i.e., good neighbor candidates) with the greedy depth-first strategy and then prune entire sub-trees within Elf. This is consistent to the observations for the Chebyshev metric where the largest difference between the dimension values defines the distance between two points. For those three metrics, we furthermore observe that the $\#f^n$ value significantly differs from the ECF value depicted in Fig. 4.(b). This means that the pruning capability of Elf is the decisive performance factor and Elf scales well for all parameter combinations. The results for Cosine distance significantly differs from the observations of the prior three metrics. In any case, the $\#f^n$ is the same as the ECF value meaning that pruning is not relevant here and the only performance benefit is achieved by the re-use effect.

In summary, the different result patterns per distance function suggest to include all Minkowski metrics when comparing the performance of exploiting sub-space distance equalities to well-known competitors. We do not include Cosine distance as its knn performance is dominated only by the re-use effects and thus, yields no further insights; also the Cosine distance is not applicable to some competitors (e.g., metric indexing approaches).

5.2.2. Investigations on the strength of the effects

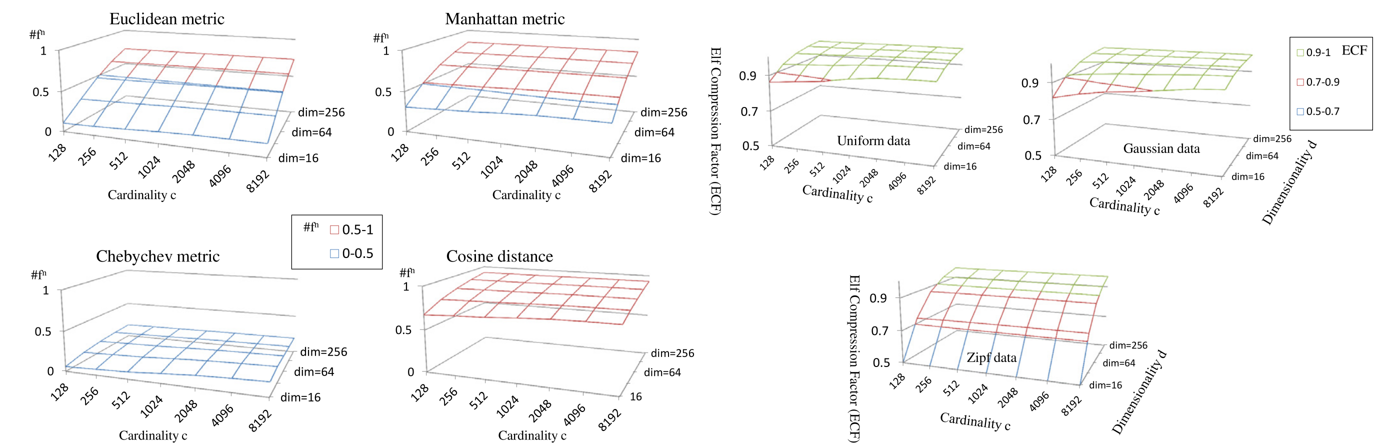
With Elf, we can exploit two effects to accelerate knn computation. Below we aim at systematically examine the strength of the effects in isolation.

Re-use sub-space distances. The strength of the re-use of sub-space distances effect is equivalent to the compression factor within Elf (ECF). Recapitulate, the ECF for a specific Elf refers to the fraction of distances to compute traversing an entire Elf and thus, represents an upper bound for $\#f^n$. As explained in Section 4.3, the bound only depends on the data distribution.

To give an intuition on the size of the bound, we depict the ECF values for all distributions, i.e., Uniform, Gaussian and Zipf data. As we see for the Uniform (worst-case) distribution and for Gaussian data, there is little compression for higher dimensionality and cardinality. Nevertheless, the bound exists for every iteratively computable distance and remains significant for, e.g., Zipf-distributed data. This is a big advantage of exploiting sub-space distance equalities.

Group lower bound. Next, we aim to understand the benefit of the group lower bound effect compared to traditional prefix-sum based pruning. To this end, we compare the $\#f^n$ of Elf with the ones achieved by a sequential scan that uses prefix-sum based pruning. The sequential scan sums up the distances dimension-wise checking after each dimension, whether the sub-space distance already exceeds the (full-space) distance $max\_dist(S, q)$. To eliminate the re-use effect in Elf, as far as possible, we rely on Uniformly distributed data where the strength of the first effect for a given cardinality and dimensionality is minimal.

In Table 4, we depict the $\#f^n$ value improvement when using Elf compared to the prefix-sum sequential scan. To explain the relevance of the difference, the $\#f^n$ values of Elf are given in Fig. 4.(a) in the top left most graph. The results in Table 4, reveal that the benefit *decreases* if cardinality or dimensionality increases, but still is observable for the largest data set. While the general tendency is expected, we emphasize that the group lower bound effect in isolation leads to observable improvements even for the Uniformly distributed data representing the worst-case distribution.



(a) Distance function influence: $\#f^n$ values for Gaussian data.

(b) Exemplary bounds for Uniform, Gaussian, and Zipf data.

Fig. 4. Impact of the different distance functions and data distributions.

Table 4
$\#f^n$ improvement exploiting the group lower bound effect compared to prefix-distance lower bound pruning for Gaussian data.

Card/Dim	16	32	64	128	256
128	0.135	0.077	0.038	0.019	0.009
256	0.110	0.062	0.030	0.015	0.008
512	0.085	0.047	0.025	0.012	0.006
1024	0.074	0.041	0.022	0.011	0.005
2048	0.057	0.031	0.017	0.009	0.005
4096	0.039	0.022	0.012	0.006	0.003
8192	0.034	0.021	0.010	0.004	0.003

0.15 #fⁿ impr. 0

5.3. Extrinsic evaluation

We now compare Elf's knn response time performance to well-known competitors using different data sets and different distance functions. To this end, we first give additional preliminaries. Then, we present the results w.r.t. the speedup, which we validate subsequently with the implementation independent $\#f^n$. Finally, we compare approaches using approximation.

5.3.1. Additional preliminaries

To ensure soundness and reproducibility of our results, we outline relevant additions to the study design below.

Competitor selection. As baseline, we select the sequential scan, which is known to be a highly-potent competitor due to the curse of dimensionality. Additionally, we use the kd-Tree as classical indexing approach and iDistance as state-of-the-art indexing approach for metric spaces relying on pivots. Next, we include R*-Tree, VA-File, and M-Tree all being competitors used in [27]. Finally, to quantify the approach-specific influence of Elf compared to the bare concept of sub-space distance equality, we include an Elf without optimized memory layout named *List Elf*. At implementation level, we tune all approaches to the same extent to ensure a fair comparison. To tune iDistance, we furthermore rely on the method from [32]. For, M-Tree and R*-Tree one needs to specify the node split size, for the VA-File the number of bits per dimension. To this end, we conducted a grid search using the best found parameter value in the evaluation.

Measurements. We rely on two measurements computed for each triple of approach, distance function, and data set. We select speedup as implementation dependent measurement and $\#f^n$ as implementation independent measurement. To compute the speedup, we measure the average response time for executing 1000 randomly selected knn queries and normalize by the average response time of the sequential scan. As usual, a speedup value smaller than 1.0 indicates that the approach is slower than the sequential scan.

5.3.2. Speedup results for all distance functions

In Fig. 5, we display the speedup for the three different distance functions concerning all use cases. We depict the results for real-world data sets D_r^d (in blue) and corresponding artificial Gaussian data sets D_G^d (in red). Since we observe significant performance differences between real-world and associated artificial data set, we discuss their results separately.

Real-world data sets. Considering the Elf approach, we observe response time speedups for all real-world data sets and distance functions. For D_r^{43}, D_r^{50}, and D_r^{51}, the results indicate large performance improvements ranging from several factors up to two magnitudes — independent of the distance function. By contrast, for D_r^{128}, the observed improvement for Euclidean and Manhattan distance is marginal, but existing. However, for this data set, having the highest dimensionality, all other approaches barely reach response time of the sequential scan, which is a known issue [7]. Note, this also holds for the List Elf without main-memory optimized data layout executing exactly the same number of distance computation. This means the response time difference can only be attributed to the optimized layout. This result is in accordance with a line of related work adapting other tree-based approaches, such as B-Trees [42,43], to optimally benefit from modern-hardware environments. Altogether, even if we observe for some combination of data set and distance function slightly higher speedups using iDistance, the performance of Elf is at least comparable.

Artificial data sets. The most obvious observation is that for all combinations of approach, distance function, and data set, the speedup of the artificial data sets is significantly worse than for its real-world counterpart. This is expected, as with increasing dimensionality the distances between points become more similar (ultimately even the same). Thus, every approach depending *only* on lower bound based pruning must fail as it converges towards scanning the entire data set and additionally needs to traverse, e.g., a tree. This also affects the group lower bound effect of sub-space distance equalities, i.e., Elf. Nevertheless, apart from lower bound pruning, the knn algorithm of Elf also exploits the re-use effect. This explains why Elf, in contrast to the competitors, is able to match the speed of the sequential scan. Interestingly, also

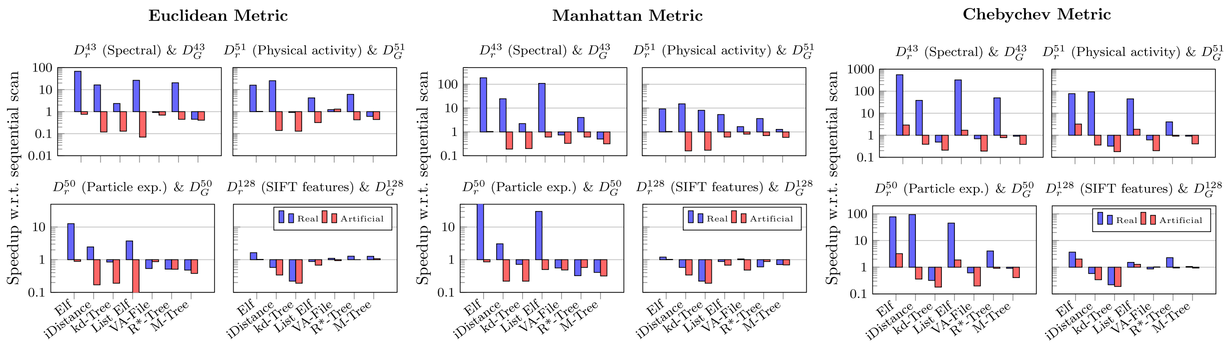


Fig. 5. Speedups for all three distance functions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the VA-File hardly reaches the speed of the sequential scan, in contrast to what one observes in hard-disk environments or for more complex CPU-bound distance functions. Next, we examine the speedup results in more detail by inspecting the correlation of the speedup and computed distances, i.e., whether the number of computed distances explains the response times.

5.3.3. Validation of the speedup with $\#f^n$

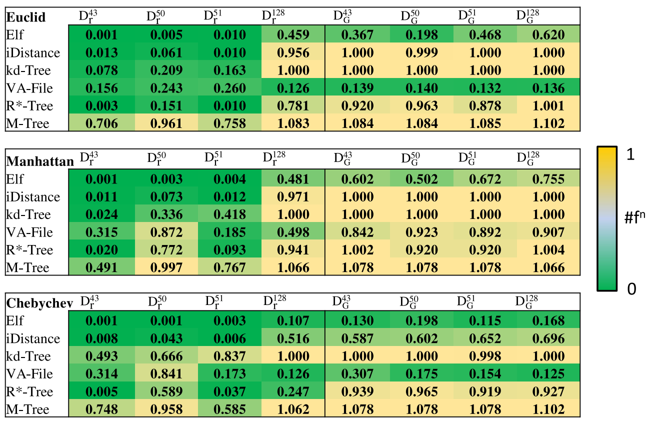
In the following, we aim at validating the above response time results with an implementation independent measure. To this end, we examine whether the $\#f^n$ and speedup values are correlated, i.e., can be considered an explanation. In Fig. 6, we depict a heat map of the $\#f^n$ for all combinations of approach, data set, and distance function. Note, $\#f^n$ values larger than 1.0 may occur for R*-Tree and M-Tree, as descending the tree involves distance computations. For them, worst case number of computed distances per query is: $|D| + \#$inner nodes.

The primary result is that, with exception of the VA-File (relying on a different concept), in all cases, the $\#f^n$ values and speedup are inversely correlated. This is backuped by common correlation measures, such as Pearson's correlation of -0.76 or Kendall's tau of -0.77 for our Elf algorithm though correlation appears not to be entirely linear. This indicates that additional effects, like caching, observably influence the run time. Nevertheless, for Elf, large speedups are explained with small $\#f^n$ values and vice versa. To this end, we draw the conclusion that indeed the reduction of distance computations achieved by the two exploited effects explains the observed good run times of Elf.

The VA-File anomaly is that, firstly, there is no clear increase of $\#f^n$ with increasing dimensionality. This is in line with related work [27] and the intention for its development [26]. However, secondly, despite $\#f^n$ is small, we rarely observe speedups. To rule out insufficient optimization, we compared the implementation to [27] revealing no difference. We hypothesize that we observe a counter-intuitive artifact known from selection predicate evaluation via sequential scans [44]. There, the worst run time is observed for predicates with 50% selectivity, meaning that run time improves in case more result tuples are returned. This effect is caused by branch miss prediction of the CPU and an non-predictable memory access pattern.

5.3.4. Comparison to nearest neighbor approximation

Depending on the use case, trading result quality for runtime might be an option. To this end, we examine how the Elf approach, returning the correct knn, performs in comparison to well-known approximation approaches. To measure result quality, we rely on the $(1+\epsilon)$-nearest neighbor definition [9]. Parameter ϵ is the factor scaling the allowed deviation from $max\_dist(S, q)$. For instance, setting $\epsilon = 1$, one accepts any k points whose distance to q is at maximum twice as large as



Euclid	D_r^{43}	D_r^{50}	D_r^{51}	D_r^{128}	D_G^{43}	D_G^{50}	D_G^{51}	D_G^{128}
Elf	**0.001**	**0.005**	**0.010**	**0.459**	**0.367**	**0.198**	**0.468**	**0.620**
iDistance	**0.013**	**0.061**	**0.010**	**0.956**	**1.000**	**0.999**	**1.000**	**1.000**
kd-Tree	**0.078**	**0.209**	**0.163**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
VA-File	**0.156**	**0.243**	**0.260**	**0.126**	**0.139**	**0.140**	**0.132**	**0.136**
R*-Tree	**0.003**	**0.151**	**0.010**	**0.781**	**0.920**	**0.963**	**0.878**	**1.001**
M-Tree	**0.706**	**0.961**	**0.758**	**1.083**	**1.084**	**1.084**	**1.085**	**1.102**

Manhattan	D_r^{43}	D_r^{50}	D_r^{51}	D_r^{128}	D_G^{43}	D_G^{50}	D_G^{51}	D_G^{128}
Elf	**0.001**	**0.003**	**0.004**	**0.481**	**0.602**	**0.502**	**0.672**	**0.755**
iDistance	**0.011**	**0.073**	**0.012**	**0.971**	**1.000**	**1.000**	**1.000**	**1.000**
kd-Tree	**0.024**	**0.336**	**0.418**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
VA-File	**0.315**	**0.872**	**0.185**	**0.498**	**0.842**	**0.923**	**0.892**	**0.907**
R*-Tree	**0.020**	**0.772**	**0.093**	**0.941**	**1.002**	**0.920**	**0.920**	**1.004**
M-Tree	**0.491**	**0.997**	**0.767**	**1.066**	**1.078**	**1.078**	**1.078**	**1.066**

Chebychev	D_r^{43}	D_r^{50}	D_r^{51}	D_r^{128}	D_G^{43}	D_G^{50}	D_G^{51}	D_G^{128}
Elf	**0.001**	**0.001**	**0.003**	**0.107**	**0.130**	**0.198**	**0.115**	**0.168**
iDistance	**0.008**	**0.043**	**0.006**	**0.516**	**0.587**	**0.602**	**0.652**	**0.696**
kd-Tree	**0.493**	**0.666**	**0.837**	**1.000**	**1.000**	**1.000**	**0.998**	**1.000**
VA-File	**0.314**	**0.841**	**0.173**	**0.126**	**0.307**	**0.175**	**0.154**	**0.125**
R*-Tree	**0.005**	**0.589**	**0.037**	**0.247**	**0.939**	**0.965**	**0.919**	**0.927**
M-Tree	**0.748**	**0.958**	**0.585**	**1.062**	**1.078**	**1.078**	**1.078**	**1.102**

Fig. 6. $\#f^n$ values for all distance functions and approaches.

$max\_dist(S, q)$. There is no general rule for determining a good use-case independent value of ϵ. As a result, it is not meaningful to compare the relationship of speedup and ϵ. Instead, we ask what is the result quality when bounding the fraction of probed points (i.e., $\#f^n$ measure). We contextualize this by stating where Elf delivers the exact result. This allows a domain expert to decide, whether accepting lesser result quality pays off. The $\#f^n$ measure additionally correlates with the speedup being identical for all used approaches and combinations of data sets and distance functions. At $\#f^n = 0.125$ the approaches are as fast as a sequential scan (due to random access of points). Changes in the $\#f^n = 0.125$ bound are directly reflected in the speedup meaning, e.g., that halving the $\#f^n$ doubles the speedup and vice versa.

For comparison, we use LSH approaches giving statistical guarantees regarding ϵ. Specifically, we use p-stable LSH [45,46], offering the guarantee for various distance functions. Furthermore, we rely on the random projection approach [47]. For each approach, we use an LSH forest with ten hash functions and verify the result pattern for different numbers of hash functions. Finally, as interpretation of the approximation factor in high-dimensional spaces is difficult, we include a baseline approach named *pseudo LSH*. This approach iterates over the data set in a random order returning the best k neighbors when having reached the maximally allowed $\#f^n$ bound.

The results in Table 5 show the results for the Euclidean distance. For the remaining distances the results pattern is similar. In detail, the table contains the achieved ϵ for each data set and a pre-defined $\#f^n$ threshold, bounding the maximum number of points an approach may probe. For instance in the upper table, containing the results for all real-world-data sets, on data set D_r^{43}

Table 5
Achieved ϵ for Euclidean distance for increasing number of computed distances $\#f^n$. The highlighted cells 🟩 mark all $\#f^n$ value exceeding the $\#f^n$ values Elf requires to deliver the exact result.

	Approach	$\#f^n$ Bound			Real-world data sets								
		0.001	0.002	0.004	0.008	0.016	0.032	0.064	0.125	0.25	0.5	0.6	0.7
D_r^{43}	Pseudo LSH	>99	55.38	27.31	14.89	7.46	4.25	2.46	1.47	0.78	0.33	0.23	0.16
	Rand. Proj.	>99	>99	>99	>99	>99	>99	>99	>99	82.89	9.96	1.85	0.04
	p-stable LSH	4.72	2.64	1.30	0.70	0.45	0.24	0.11	0.02	0.01	≈0	≈0	≈0
D_r^{50}	Pseudo LSH	20.27	12.24	7.80	5.01	3.31	2.30	1.49	0.97	0.60	0.26	0.19	0.14
	Rand. Proj.	22.59	13.62	8.79	6.75	4.24	3.10	2.31	1.71	0.42	0.14	0.08	0.03
	p-stable LSH	5.65	3.25	2.11	1.25	0.75	0.52	0.30	0.13	0.02	≈0	≈0	≈0
D_r^{51}	Pseudo LSH	>99	>99	62.20	35.44	22.04	14.78	8.55	3.93	1.96	0.60	0.42	0.28
	Rand. Proj.	>99	>99	>99	>99	>99	14.00	2.84	1.23	0.13	0.01	≈0	≈0
	p-stable	8.64	5.56	1.90	0.44	0.20	0.08	0.01	0.01	≈0	≈0	≈0	≈0
D_r^{128}	Pseudo LSH	1.39	1.16	0.93	0.73	0.58	0.46	0.35	0.25	0.16	0.08	0.06	0.04
	Rand. Proj.	0.20	0.17	0.14	0.12	0.09	0.07	0.05	0.03	0.02	0.01	0.01	≈0
	p-stable	0.23	0.19	0.17	0.14	0.12	0.09	0.07	0.05	0.03	0.01	0.01	0.01

	Approach	$\#f^n$ Bound			Artificial data sets								
		0.001	0.002	0.004	0.008	0.016	0.032	0.064	0.125	0.25	0.5	0.6	0.7
D_G^{43}	Pseudo LSH	0.68	0.59	0.49	0.41	0.33	0.26	0.20	0.15	0.10	0.04	0.03	0.02
	Rand. Proj.	0.50	0.41	0.33	0.26	0.19	0.14	0.09	0.05	0.02	0.01	0.01	≈0
	p-stable LSH	0.63	0.53	0.44	0.36	0.28	0.22	0.16	0.11	0.06	0.02	0.01	0.01
D_G^{50}	Pseudo LSH	0.66	0.57	0.47	0.39	0.32	0.25	0.19	0.14	0.09	0.04	0.03	0.02
	Rand. Proj.	0.50	0.40	0.32	0.24	0.18	0.13	0.08	0.05	0.03	0.01	≈0	≈0
	p-stable LSH	0.62	0.51	0.43	0.34	0.27	0.21	0.15	0.11	0.06	0.02	0.01	0.01
D_G^{51}	Pseudo LSH	0.51	0.44	0.37	0.32	0.26	0.21	0.16	0.12	0.08	0.04	0.02	0.02
	Rand. Proj.	0.38	0.31	0.25	0.20	0.15	0.10	0.06	0.03	0.02	0.01	0.01	≈0
	p-stable	0.46	0.39	0.33	0.27	0.22	0.18	0.14	0.10	0.05	0.02	0.01	0.01
D_G^{128}	Pseudo LSH	0.24	0.21	0.18	0.15	0.13	0.10	0.08	0.06	0.04	0.02	0.01	0.01
	Rand. Proj.	0.20	0.17	0.14	0.12	0.09	0.07	0.05	0.03	0.01	0.01	0.01	0.01
	p-stable	0.22	0.19	0.17	0.14	0.12	0.01	0.07	0.05	0.03	0.01	0.01	0.01

p-stable LSH achieves for bound $\#f^n = 0.001$ rather low result quality with $\epsilon = 4.72$ returning highly accurate results after inspecting one eighth (0.125) of the data set with an approximation level of 0.02.

Overall, we make three major observations. Firstly, the performance of the LSH algorithms highly differs between the real-world data sets (in the upper table) and their artificial counterparts in the lower table. Every approach delivers better (i.e., more accurate) results for artificial data. This is the opposite behavior as observed for the exact approaches. Secondly, with increasing dimensionality the performance of all LSH approaches improves. That is, they deliver more accurate results probing smaller fractions of the data set. This is again the opposite behavior as observed for the exact approaches. Third, for artificial data sets, the approximation level of all approaches has almost converged towards the exact results before Elf – or any other tested exact approach – has computed the exact result. Similarly, the worst observed approximation level is 0.68 observed for the pseudo LSH approach at D_G^{43}.

These observations are expected, as with increasing (intrinsic) dimensionality the average distances between points become more similar [7]. This explains why simple random sampling of the artificial data sets delivers quite accurate results. Altogether, these results indicate that exact and approximate nearest neighbor determination, are related problems having however different application scenarios. Abstractly, improvements in exact nearest-neighbor research increase the dimensionality of data for that one can efficiently determine the exact knn. Thus, if dimensionality is high and one can accept less result quality considering approximation is suggested. In turn, in case one does require the exact neighbors, approximation is no option by concept.

5.4. Evaluation summary

In our evaluation, we first studied the effect of different parameters affecting knn query performance when exploiting sub-space distance equalities. The results indicate that in contrast to distance function and data distribution, parameter k has minor influence. Moreover, the result pattern in any of the experiments is good-natured indicating a robustness towards parameter changes, i.e., one can generally expect similarly good results on similar data sets.

Furthermore, in the second part of the evaluation, the results suggest that Elf delivers a speedup or a comparable performance as specialized approaches like iDistance. This particularly also includes the artificial data sets, where one otherwise has to accept lesser result quality using approximation to achieve a speedup. Our results indicate that Elf, compared to the competitors, suffers less performance losses. Specifically, it is the only approach delivering, on average, a speedup compared to the baseline of sequential scanning the entire data set suggesting that one can generally expect good knn query performance.

To this end, our general conclusion regarding sub-space distance equalities and the Elf approach is that we found a candidate for the envisioned multi-purpose main-memory storage structures. This holds at concept level, measured by the fraction of computed distances $\#f^n$ compared to a sequential scan, and at implementation level (i.e., knn algorithms of Elf), measured by the speedup (compared to a sequential scan).

6. Conclusions and future work

In this paper, towards using the same index for multiple query types, we study the concept of sub-space distance equalities yielding two effects for efficient knn computation, namely group lower bound effect and re-use effect of sub-space distances. We propose knn algorithms exploiting both effects for the Elf storage structure.

Our investigations reveal that the effects are robust to increasing dimensionality, data set size, and utilized k. Moreover, studying the impact of data distribution and used distance function also indicates robust and good natured results. Comparing our knn algorithms to well-known exact and approximate competitors reveals that they deliver at least comparable performance for real-world data following a complex distribution. Furthermore, for high-dimensional Gaussian or Uniform data, our knn algorithms still deliver competitive performance due to the combined usage of effects.

For future work, we investigate on efficiently supporting further fundamental building blocks of beyond-OLAP analytics like matrix operations.

Funding

This work was partially funded by the DFG (grant no.: SA 465/51-1 and SA 465/50-1).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2006.
[2] C. Böhm, S. Berchtold, D. Keim, Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (3) (2001).
[3] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998).
[4] H. Jagadish, et al., iDistance: An adaptive B+-Tree based indexing method for nearest neighbor search, ACM Trans. Database Syst. 30 (2) (2005).
[5] M. Hetland, The basic principles of metric indexing, in: Swarm Intelligence for Multi-Objective Problems in Data Mining, Springer, 2009.
[6] L. Chen, et al., Pivot-based metric indexing, PVLDB 10 (10) (2017).
[7] K. Beyer, et al., When is ''Nearest Neighbor'' meaningful? in: Proc. Int'L Conf. on Database Theory, ICDT, Springer, 1999.
[8] T. Rakthanmanon, et al., Searching and mining trillions of time series subsequences under dynamic time warping, in: Proc. Int'L Conf. on Knowledge Discovery and Data Mining, SIGKDD, ACM, 2012.
[9] S. Arya, D. Mount, Approximate nearest neighbor queries in fixed dimensions, in: Proc. An'L Symp. on Discrete Algorithms, SODA, ACM, 1993.
[10] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proc. Int'L Conf. on Very Large Data Bases, VLDB, Morgan Kaufmann, 1999.
[11] D. Broneske, et al., Accelerating multi-column selection predicates in main-memory - The Elf approach, in: Proc. IEEE Int'L Conf. on Data Engineering, ICDE, IEEE, 2017.
[12] D. Broneske, et al., Efficient evaluation of multi-column selection predicates in main-memory, IEEE Trans. Knowl. Data Eng. 31 (7) (2018).
[13] P. Blockhaus, et al., Combining two worlds: MonetDB with multi-dimensional index structure support to efficiently query scientific data, in: Proc. Int'L Conf. on Scientific and Statistical Database Management, SSDBM, ACM, 2020.
[14] J. Wang, et al., Can we beat the prefix filtering? An adaptive framework for similarity join and search, in: Proc. Int'L Conf. on Management of Data, SIGMOD, ACM, 2012.
[15] E. Schubert, et al., DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN, ACM Trans. Database Syst. 42 (3) (2017).
[16] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, From data mining to knowledge discovery in databases, AI Mag. 17 (3) (1996).
[17] E. Keogh, et al., Dimensionality reduction for fast similarity search in large time series databases, Knowl. Inf. Syst. 3 (3) (2001).
[18] I. Jolliffe, Principal Component Analysis, Springer, 1986.
[19] T. Mikolov, et al., Distributed representations of words and phrases and their compositionality, in: Proc. Int'L Conf. on Neural Information Processing Systems, NIPS, Curran Associates Inc., 2013.
[20] G. Williams, Linear Algebra, with Applications, Narosa, 2009.
[21] A. Guttman, R-trees: A dynamic index structure for spatial searching, SIGMOD Rec. 14 (2) (1984).
[22] J. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975).
[23] S. Berchtold, D. Keim, H.-P. Kriegel, The X-tree: An index structure for high-dimensional data, in: Proc. Int'L Conf. on Very Large Data Bases, VLDB, Morgan Kaufmann, 1996.
[24] P. Ciaccia, M. Patella, 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, Morgan Kaufmann, 1997.
[25] S. Omohundro, Five Balltree Construction Algorithms, Tech. Rep. TR-89-063, Int'L Computer Science Institute, Berkeley, 1989.
[26] R. Weber, H.-J. Schek, 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, Morgan Kaufmann, 1998.
[27] K. Echihabi, et al., The Lernaean Hydra of data series similarity search: An experimental evaluation of the state of the art, PVLDB 12 (2) (2018).
[28] V. Niennattrakul, P. Ruengronghirunya, C. Ratanamahatana, Exact indexing for massive time series databases under time warping distance, Data Min. Knowl. Discov. (2010).
[29] J. Willkomm, et al., Efficient interval-focused similarity search under dynamic time warping, in: Proc. Int'L Symp. on Spatial and Temporal Databases, SSTD, ACM, 2019.
[30] M. Micó, J. Oncina, E. Vidal, A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements, Pattern Recognit. Lett. 15 (1) (1994).
[31] J. Uhlmann, Satisfying general proximity/similarity queries with metric trees, Inform. Process. Lett. 40(4) (1991).
[32] M. Schuh, et al., 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.
[33] C. Yu, High-Dimensional Indexing: transformational Approaches to High-Dimensional Range and Similarity Searches, Springer, 2002.
[34] M. Hetland, T. Skopal, J. Loko, C. Beecks, Ptolemaic access methods: Challenging the reign of the metric space model, Inf. Syst. 38 (7) (2013).
[35] S. Arya, et al., An optimal algorithm for approximate nearest neighbor searching fixed dimensions, J. ACM 45 (6) (1998).
[36] W. Johnson, J. Lindenstrauss, Extensions of Lipschitz maps into a Hilbert space, Contemp. Math. 26 (1984).
[37] M. Schäler, et al., QuEval: Beyond high-dimensional indexing à La Carte, PVLDB 6 (14) (2013).
[38] K. Pettis, et al., An intrinsic dimensionality estimator from near-neighbor information, IEEE Trans. Pattern Anal. Mach. Intell. 1 (1) (1979).
[39] F. Keller, E. Müller, K. Böhm, HiCS: High contrast subspaces for density-based outlier ranking, in: Proc. IEEE Int'L Conf. on Data Engineering, ICDE, 2012.
[40] D. Dua, C. Graff, UCI machine learning repository, 2017, URL http://archive.ics.uci.edu/ml.
[41] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Proc. Conf. on Object-Oriented Programming Systems and Applications, OOPSLA, ACM, 2007.
[42] G. Graefe, P. Larson, B-tree indexes and CPU caches, in: Proc. IEEE Int'L Conf. on Data Engineering, ICDE, IEEE, 2001.
[43] J. Rao, K. Ross, Making B+-Trees cache conscious in main memory, SIGMOD Rec. 29 (2000).
[44] L. Schulz, D. Broneske, G. Saake, An eight-dimensional systematic evaluation of optimized search algorithms on modern processors, Proc. VLDB Endow. 11 (11) (2018).
[45] M. Datar, et al., Locality-sensitive hashing scheme based on P-stable distributions, in: Proc. An'L Symp. on Computational Geometry, SCG, ACM, 2004.
[46] A. Dasgupta, R. Kumar, T. Sarlos, Fast locality-sensitive hashing, in: Proc. Int'L Conf. on Knowledge Discovery and Data Mining, KDD, ACM, 2011.
[47] M. Charikar, Similarity estimation techniques from rounding algorithms, in: Proc. An'L Symp. on Theory of Computing, STOC, ACM, 2002.