Subspace Clustering for Indexing High Dimensional Data: A Main Memory Index based on Local Reductions and Individual Multi-Representations

Stephan Günnemann Hardy Kremer Dominik Lenhard Thomas Seidl
Data Management and Data Exploration Group
RWTH Aachen University, Germany
{guennemann, kremer, lenhard, seidl}@cs.rwth-aachen.de

ABSTRACT

Fast similarity search in high dimensional feature spaces is crucial in today's applications. Since the performance of traditional index structures degrades with increasing dimensionality, concepts were developed to cope with this curse of dimensionality. Most of the existing concepts exploit *global* correlations between dimensions to reduce the dimensionality of the feature space. In high dimensional data, however, correlations are often *locally* constrained to a subset of the data and every object can participate in several of these correlations. Accordingly, discarding the same set of dimensions for each object based on global correlations and ignoring the different correlations of single objects leads to significant loss of information. These aspects are relevant due to the direct correspondence between the degree of information preserved and the achievable query performance.

We introduce a novel main memory index structure with increased information content for each single object compared to a global approach. This is achieved by using individual dimensions for each data object by applying the method of subspace clustering. The structure of our index is based on a multi-representation of objects reflecting their multiple correlations; that is, besides the general increase of information per object, we provide several individual representations for each single data object. These multiple views correspond to different local reductions per object and enable more effective pruning. In thorough experiments on real and synthetic data, we demonstrate that our novel solution achieves low query times and outperforms existing approaches designed for high dimensional data.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Algorithms, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

1. INTRODUCTION

Similarity search in databases is an active research area with a wide range of application domains. In many of these domains, fast query times are crucial and long waiting periods are prohibited. For example, even minutes can be fatal in medical applications. Similarity search is typically realized in a content-based fashion, i.e. features like histograms are extracted from objects and the similarity between objects is modeled by a similarity function operating in the feature space. Usually distance functions are used for this purpose. The L_p-norm is a family of well suited distance functions: $d(q,p) = \sum_{i=1}^{d} \sqrt[p]{|q_i - p_i|^p}$. For $p = 2$ it corresponds to the Euclidean Distance. An often used query type in similarity search is the k-Nearest-Neighbor(kNN) query that calculates the k most similar objects to a query object.

For efficient query processing on multidimensional feature spaces index structures were introduced, and they are mostly based on hierarchically nested minimal bounding regions. With increasing feature space dimensionality or information content, many of the first solutions, e.g. the R-Tree [14], are inapplicable for the given tasks [7, 9, 31]. This is often denoted as the curse of dimensionality: with rising dimensionality or information content index performance degrades, eventually becoming slower than a sequential scan of the database.

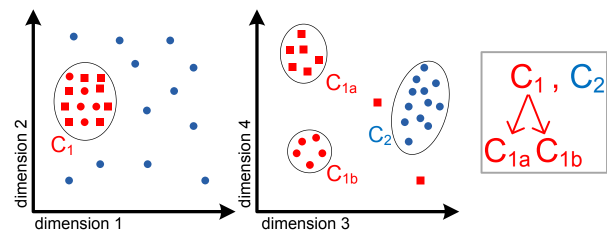


Figure 1: Hierarchically nested local reductions

Filter-and-refine frameworks as well as revised and new index structures were introduced to weaken the effects of high dimensionality. In filter-and-refine frameworks, an index is built on a dimensionality reduced representation of the database allowing for fast computation of possible candidates. The exact distances to these candidates are then computed in the subsequent refinement step. The dimensionality reduction is performed by exploiting correlations between the feature space dimensions corresponding to re-

dundancy in the data. An example is the principal component analysis [17], which transforms the original feature space to a lower dimensional one containing most of the relevant information. Such filter-and-refine frameworks as well as the mentioned index structures weaken the curse of dimensionality, but the general problem is still not solved. The reason is the complex structure hidden in the data: specific correlations between the dimensions are often local, i.e. they are constrained to subsets of the data, and every object in the data can participate in several of these correlations. Reductions based on global correlations, however, cannot reflect the local correlations in an advantageous way for index construction; all objects in the data will be reduced to the same dimensions, and therefore many dimensions are discarded that are important for specific subsets of the data. In the remaining dimensions the values in these subsets have a high variance preventing compact minimal bound regions in the index and resulting in unnecessary large query times. An example is shown in Figure 1 where two coordinate systems represent a 4-dimensional feature space. The values of the shown objects are scattered over the full-dimensional feature space. Examining all objects together, there is no correlation between the dimensions that enables a reduction beneficial for index construction; that is, finding patterns in a single reduced space allowing the construction of compact minimal bounding regions is not possible. It is, however, obvious that the data contains several good patterns when only subsets of the data are considered. They are marked as C_1, C_2, C_{1a}, and C_{1b}. For each of the corresponding object sets a sound reduction can be found corresponding to a local dimensionality reduction. An index constructed based on such local reductions is more effective: dimensions with high variances in the value distributions are avoided and thus more compact bounding regions are achieved winding up in faster query processing due to earlier pruning of candidates. For example, the local reductions for C_1 and C_2 allow compact bounding regions that are good for pruning parts of the data.

Accordingly, for complex data indexing frameworks are needed whose corresponding filter steps are based on local reductions. One possible solution is to build an index for every local pattern in the data, i.e. every index corresponds to a different filter with another set of reduced dimensions [8, 29]. These approaches, however, have a serious drawback: They do not consider that every object can participate in several local reductions, i.e. every object is only assigned to one of these reductions. The potential for several different filters for each data object and better pruning based on these filters is therefore wasted.

In this paper, we introduce a novel index structure that is based on local reductions and that in particular regards the multiple local correlations of single data objects. This is achieved by building a hierarchy of local reductions that corresponds to the structure of the index tree; that is, a local reduction-based minimum bounding region can itself contain local reductions that enable compact bounding regions in lower levels of the index tree. As mentioned before, an index on dimensionality reduced data objects acts as a filter in filter-and-refine frameworks. Accordingly, our *single* index corresponds to a *series* of different filters for every data object; every query passes through an individual cascade of filters until it reaches a leaf node. An example for such a pattern hierarchy is displayed in the right part of Figure 1.

Technically, we realize these new ideas by transferring methods from the data mining domain: we detect patterns in the data which are used to construct a better index. For detecting data subsets combined with a set of dimensions suitable for local reduction, we use subspace clustering. Clustering in general is used to find natural object groupings in data. The objects contained in such groups are good candidates for generating minimal bounding regions in index structures. Subspace clustering goes beyond fullspace clustering: A subspace cluster is a grouping of objects that are only similar in a subset of their dimensions and thus minimum bounding regions that are induced by subspace clusters are very compact. The underlying structure of our proposed index structure is a hierarchy of nested subspace clusters. A key aspect is the recursive use of subspace clustering, i.e. all levels of the index tree are constructed by reclustering existing subspace clusters thus generating subclusters. For example, in Figure 1 our approach can identify the illustrated hierarchical nesting. There are interesting implications: Due to the nature of subspace clustering, it is possible that subspace clusters on deeper levels have other relevant dimensions than their parent clusters. This corresponds to a multi-representation of objects, i.e. on each level objects can be represented by a different set of dimensions defining individual local reductions.

When clustering is used for index construction, there are issues: Clustering algorithms are often unstable, i.e. they are very parameter-sensitive and they deliver different results on every run; therefore we introduce a method that is motivated by train-and-test, a paradigm that is well established in the data mining and machine learning domain, e.g. for decision tree construction. In applications where the underlying distribution of incoming data objects changes, our index adapts its underlying clustering structure dynamically.

Many existing index structures are designed for secondary storage, and thus technical constraints as block size must be adhered to. In many domains, however, these index structures are becoming obsolete: Main memory capacity of computers is constantly increasing and in many applications persistence of the index structure or even the data set has lost importance compared to the need for fast responses. An example are stream classification scenarios. For main memory indexing, random I/O is no longer a problem allowing the development of more flexible index structures. For example, our unbalanced index structure can reflect the inherent structure of the data better than structures that are constrained by balance properties: different parts of the data set can be represented in different granularities, thus allowing more efficient access.

Summarized, our contributions are:

- Our novel index structure is based on local reductions determined by subspace clustering.
- The objects are multi-represented with different local reductions that are hierarchically nested. Queries have to pass through an individual series of filters allowing for faster pruning of candidates.

- The index construction is enhanced by a train-and-test method providing more compact minimal bounding regions and the underlying structure of the index adapts to new data distributions by reclustering subtrees.

The paper is structured as follows: Section 2 discusses existing work on indexing techniques and subspace clustering. Section 3 introduces our new approach. Section 4 presents experimental evaluation and Section 5 concludes the paper.

2. RELATED WORK

In this section we give a short overview of relevant related work on indexing. Since our novel index structure is based on subspace clustering, we also give a short overview of this research area.

Indexing techniques. Indexing techniques for fast similarity search can be categorized into approximate and correct solutions. Methods from the former category trade accuracy for speed, and one well-known example is Local Sensitive Hashing [30, 12]. Methods from the latter category produce exact results, i.e. there are no false dismissals or false positives in the result set. Since our proposed index is from this category, we focus on exact solutions in the following.

The R-Tree [14] is one of the first multidimensional indexing structures. It is well suited for lower dimensions, but its performance rapidly degrades for higher dimensionalities due to the curse of dimensionality; therefore, dimensionality reduction techniques like PCA [17] or cut-off reduction are used to reduce the dimensionality of indexed objects. Since dimensionality reduction induces information loss, false positives are produced that need to be rejected. This can be achieved by a filter-and-refine-framework (also called multistep query processing), e.g. GEMINI [10] or KNOP [28]. The drawbacks of global dimensionality reduction techniques were discussed in Section 1. Several indexing structures for high dimensional feature spaces that are mostly based on the R-Tree were proposed. The R*-Tree [5] introduces new split and reinsertion strategies. The X-Tree [6] enhances the R*-Tree by introducing overlap-minimizing splits and the concept of supernodes; if no overlap-minimizing split is possible, supernodes of double size are generated, eventually degenerating to a sequential scan. The A-Tree [27] uses quantization to increase the fan-out of tree nodes. The TV-Tree [20] is based on a PCA-transformed feature space: Minimum bounding regions are restricted to a subset of active dimensions in this space. Active dimensions are the first few dimensions allowing for discrimination between subtrees. It has, however, the same drawbacks as the other global dimensionality reduction techniques. Distance based indexing is another type of transformation based indexing and is realized by iDistance [16, 33]: Data points are transformed into single dimensional values w.r.t. their similarity to specific reference points. These values are indexed by a B^+-tree and the search is performed by one-dimensional range queries in these tree. All of the described index structures are optimized for secondary storage, i.e. they have node size constraints. None of these index structures accounts for local correlations in the data. An approach that makes use of local correlations is LDR [8]. It uses subspace clustering to create a single clustering of the whole dataset. These clusters represent local correlations. Dimensionality reduction is performed individually on the clusters and an index is built for each of the reduced representations. LDR has several drawbacks: it applies subspace clustering only to the root of the constructed tree, while the subtrees are built the conventional way (one index for every subtree). These indices are still prone to the curse of dimensionality. Another local-correlation based approach is MMDR [29]. In difference to LDR, a single one-dimensional index is used. In both LDR and MMDR, there is no hierarchical nesting of subspace clusters, i.e. there is no multi-representation of objects according to different dimensions and there is no handling of local correlations in the used indices.

Subspace clustering. Recent research in subspace clustering has introduced several models and algorithms. A summary can be found in [19, 25] and differences between the models are analyzed in [22, 24]. Subspace clustering aims at detecting groups of similar objects and a set of relevant dimension for each of these object groups. A general classification of the models can be done by considering the possible overlap of clusters. Partitioning approaches [2, 26, 32] force the clusters to represent disjoint object sets while in non-partitioning approaches [4, 18, 23] objects can belong to several clusters.

Besides the huge amount of algorithms that keep the original dimensions, some algorithms [3] transform the data space on detected correlations. Initial work has also be done in detecting hierarchies of subspace clusters [1]. However, the complexity of the approaches avoids efficient application.

3. SUBSPACE BASED INDEXING

In this section, we present SUSHI: A means for **SU**b**S**pace based **H**igh-dimensional **I**ndexing. In Section 3.1 we motivate and define the node structure of the tree. The construction of SUSHI based on subspace clustering is presented in Section 3.2; we consider the static case for a given database and the insertion and deletion of objects for dynamic usage. Section 3.3 presents the query processing strategy.

3.1 Tree structure

The general idea of our index structure is to represent each object in multiple ways, such that different information can be used for pruning. To enable efficient query processing we partition the data in a hierarchical structure, allowing us to prune whole subtrees if they are not important for the current query. Each subtree represents a subset of objects annotated with its local reductions avoiding the information loss attended by global reduction approaches. To determine these local reductions we use subspace clustering methods, since in high dimensional data we cannot find meaningful partitions with traditional approaches due to the curse of dimensionality [7].

Unlike existing indexing approaches we recursively apply subspace clustering on smaller subsets of the database, realizing multi-representations for each object. We identify the local correlations in the whole database but also on more fine-grained views of the data. In Figure 1, for example, the subspace cluster C_1 in the dimensions 1 and 2 can further be refined by the clusters C_{1a} and C_{1b}. We can identify these locally nested patterns with subspace clustering on the object set of C_1.

Formally, a subspace cluster is a set of objects together with a set of locally relevant dimensions. The objects show high correlations and thus compactness within the relevant dimensions while in the irrelevant ones we cannot identify a good grouping for this set of objects. For different groups of objects different relevant dimensions are possible, thus we are not constrained to a global reduction of the data. A subspace clustering is a list of subspace clusters together with a list of outliers. The handling of outliers is important because not every object shows a good correlation to other objects. If we prohibited outliers, the clustering quality would get worse and hence we could assume that also our index shows poor performance. In Figure 2 the blue circles form a subspace cluster with the relevant dimensions 1 and 2 while the red squares depict a good grouping in the dimensions 3 and 4. We cannot identify clusters in the full-dimensional space. The green triangle is an outlier, because it shows no similarity to other objects in any of the subspaces.

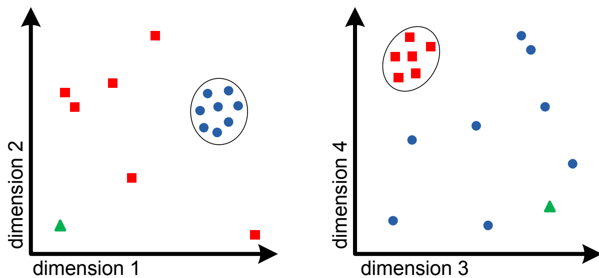


Figure 2: Example of a subspace clustering

Definition 1. *Subspace Cluster and Clustering*
Given a set of dimensions Dim and a database $DB \subseteq \mathbb{R}^{|Dim|}$, a subspace cluster C is defined by $C = (O, S)$ with objects $O \subseteq DB$ and relevant dimensions $S \subseteq Dim$.
A subspace clustering Clus is defined by

$$Clus = (C_1, \ldots, C_k, Out)$$

with subspace clusters $C_i = (O_i, S_i)$ $(i = 1, \ldots, k)$ and outlier list $Out = DB - \bigcup_{i=1}^{k} O_i$.

We call $Clus$ a subspace clustering for the database DB because each object $o \in DB$ is either in a cluster or in the outlier list. In our approach we use subspace clustering methods which partition the data in each step, i.e. the sets O_i are pairwise disjoint. This is reasonable for constructing an index structure, because otherwise we would include a single object in multiple nodes (on different paths in the tree) leading to a hindered pruning of this object. Apart from that, our index structure is independent from the underlying clustering algorithm. New developments in the research area of subspace clustering can directly be applied to our index. In Section 3.2 we describe the used algorithms.

Index structures are based on the idea of pruning parts of the search space. In our index structure, each node is determined by a subspace clustering containing several clusters. To enable pruning of all the objects within a cluster we have to aggregate a cluster $C = (O, S)$ to some compact information, i.e. a minimum bounding region: we use the idea of enclosing rectangles in the relevant dimensions of the subspace cluster, i.e. each cluster is represented by lower and upper bounds in its subspace.

Definition 2. *Subspace Enclosing Rectangle (SER)*
Given a subspace cluster $C = (O, S)$. The SER R is a list of lower and upper bound values for the relevant dimensions of the cluster, i.e. $R = ([i_1, low_1, up_1], \ldots, [i_d, low_d, up_d])$ with $\{i_1, \ldots, i_d\} = S$ and $low_j = \min_{o \in O}\{o_{|i_j}\}$, $up_j = \max_{o \in O}\{o_{|i_j}\}$.
Where $o_{|k}$ is the restriction of o to the dimensions k.

The irrelevant dimensions of a subspace cluster provide no or little information about the clustered objects. Hence we do not store this information, leading to a local dimensionality reduction, and thus we reduce the computation effort during query processing. In our study we focus on this simple representation to analyze the effect of hierarchically nested subspace clusters and not to figure out the best cluster approximation. Other approximations can easily be included in our index, as elliptical representations or transformation based approaches. However, in Section 3.3 we discuss why these methods are no good choice.

In SUSHI we distinguish between inner nodes and leaf nodes. The root node represents the whole database DB.

Definition 3. *Nodes of SUSHI*
An inner node N representing the objects O fulfills:

- *N is determined by a subspace clustering $Clus = (C_1, \ldots, C_k, Out)$ for the set O.*
- *For each cluster $C_i = (O_i, S_i)$ the node N contains a SER R_i.*
- *For each SER R_i the node N contains a pointer to its child node representing the subset O_i.*
- *N stores a list of objects (or pointers) corresponding to Out.*

A leaf node N representing the objects O fulfills:

- *N stores a list of objects (or pointers) corresponding to O.*

In the leaf nodes full-dimensional objects $o \in DB$ are stored, and each inner node can also contain a list of full-dimensional objects corresponding to the outliers obtained by one clustering step performed on the respective subset of objects. Thus we are able to identify local outliers, i.e. outliers that emerge only by regarding a subset of objects. The potential to store objects also in inner nodes yields compact representations of the remaining clusters. Otherwise one has to add these outliers to the enclosing rectangles which then grow disproportionately large.

Each subspace cluster within an inner node but also across several layers of the tree can have a different set of relevant dimensions. We do not cut off some dimensions globally, and we do not represent all objects in the same way; thus we do not perform the well-known filter-and-refine query processing [10] that is normally based on a single filter step. But

each object is represented in different ways starting from the root down to its full-dimensional object description. Each object has its individual multi-representation enabling us to prune objects along one path of our tree based on their various representations. Thus our index structure has inherently realized the filter-and-refine approach.

In Figure 3 the tree structure is outlined. The root node is determined by three subspace clusters of different sizes. Adjacent to each cluster the relevant dimensions are visualized by green squares, e.g. the first subspace cluster is located in the subspace $\{2, 5\}$. The objects are reduced to this local dimension set. Each cluster is represented by a subspace enclosing rectangle that is stored in the node. Besides the SERs the root node stores a list of outliers; in this example with three objects. Each cluster can either split up in further clusters, e.g. cluster one of the root, or a leaf node is appended as for cluster two. Subspace clusters in different levels of the tree can represent different sets of dimensions, realizing our multi-representation of objects.

Our index is a main memory structure, hence we are not restricted by block sizes. In main memory, the time to access a child node is negligible compared to secondary storage. This gives us the possibility to construct unbalanced trees and hence some objects or regions can be represented more detailed. A couple of objects have many different representations while others have only few.

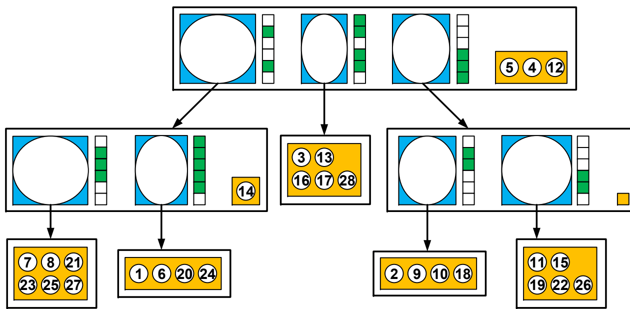


Figure 3: Tree structure of SUSHI

3.2 Tree construction

For index construction, we distinguish two cases: First, the static construction, where we build the index for a database provided in whole. Second, the dynamic construction where we cope with insertion and deletion of objects.

3.2.1 Static construction

Usually an initial set of data is available to construct our index via bulk-loading. To achieve the multi-representation of objects we need a subspace clustering for each inner node of the tree. A simple approach is to calculate only one clustering for the root node and recursively repeat this procedure for each subset to construct further nodes. However, this procedure can lead to instabilities during the index construction. Since there exist several clusterings for a single dataset – each clustering with different quality for grouping the objects – it is unlikely to get a good result with a single run of a clustering algorithm. Furthermore many algorithms are non-deterministic and they inherently construct different clusterings. Thus, in a worst case we get high variations for the index construction and thus the query performance could deviate to a high degree. To improve the stability and quality of our index we calculate multiple clustering per node and we select the one with highest pruning potential.

Since in our hierarchical structure the child nodes depend directly on the upper level clustering, we use the following approach to construct the index. We start with the root for which multiple clusterings are calculated. Consequently, we get a set of partly constructed models/index structures. We select the best model from this set and we try to complete it. Therefore we select one cluster from the root node and we cluster this subset of objects again several times. Based on our first partly constructed model we get a new set of extended but still not complete index structures. The best index is selected and the approach is repeated until the complete index is obtained. Summarized: At each point in time we extend our index structure with one inner node. This inner node is determined based on a set of candidate inner nodes, i.e. subspace clusterings, and in each step the clustering with highest pruning potential is selected.

Pruning potential of clusterings. To determine this best clustering we use an approach inspired by the train-and-test method used in other data mining applications. On the one hand, one can use train-and-test to evaluate the quality of an approach. On the other hand, train-and-test is used to construct better models itself, e.g. for decision trees [21]. The general idea is to build a set of models based on the training set and choose the best one w.r.t. the test set. In the literature several measures to judge the quality of clusterings, e.g. the compactness, are presented [15]. However, in our approach the clustering with the highest quality need not to be equivalent to the best node for the tree. In our approach, we seek for the clustering resulting in the best query performance for our index, i.e. the clustering with highest pruning potential. Motivated by this we measure exactly those costs that are important during the utilization of the index: the distance calculations (to enclosing rectangles and points) needed to determine e.g. a nearest neighbor. Since we start our index construction at the root node, each partly constructed index is a valid index structure w.r.t. Def. 3. Thus, we can temporarily append a new inner node and check how many distance calculations are performed to find the nearest neighbors of the test sets objects. Then the subspace clustering determining the best inner node based on this objective function is selected.

Selecting the right test set is important while evaluating our objective function. In an ideal case the data distribution of the test set follows the distribution of possible query objects. Since this distribution is unknown we assume that the database itself reflects this distribution. Hence our test set is randomly and uniformly sampled from the whole database DB. Although we optimize our tree locally ('what is the best new node for one subtree'), we should not use a sample of the current subtree as the test set. In this case the test set would not be representative and the subtree is not optimized w.r.t. the performance of the whole tree. To avoid overfitting and enhance the generalization we change the test set periodically.

In Algorithm 1 the overall construction of nodes is described. To generate the root node we use the whole database DB as

Algorithm 1 Node construction algorithm

```
1:  input: set of objects O;
2:  IF(|O| < minSize)
3:      constructLeafNode(); // based on O;
4:      return;
5:  Test = chooseTestSet();
6:  // initial clustering
7:  bestClust = performClusteringOn(O);
8:  bestQuality = evaluateClust(bestClust, Test);
9:  stableCount = 0;
10: WHILE(stableCount < stableSteps)
11:     currClus = performClusteringOn(O);
12:     currQuality = evaluateClust(currClust, Test);
13:     IF(currQuality > bestQuality)
14:         bestClust = currClust;
15:         bestQuality = currQuality;
16:         stableCount = 0;
17:     ELSE
18:         stableCount+ = 1;
19: constructInnerNode(); // based on bestClust;
20: FOREACH(C_i = (O_i, S_i) of bestClust)
21:     run node construction for O_i
```

input. During the hierarchical partitioning of the data the subsets of objects get smaller. If the number of objects for a node is below a certain threshold we create a leaf node (lines 2-4). The train-and-test method starts with the lines 5-8. We choose the test set, perform a subspace clustering on the objects of O and evaluate the quality of this subspace clustering w.r.t. the test set. In the lines 10-18 we try to enhance the clustering quality. Based on the best evaluated clustering we construct the inner node (line 19). At last, we call the node construction method recursively for each cluster (line 21) to realize the hierarchical structure.

Subspace clustering algorithms. As mentioned, our index is able to utilize arbitrary subspace clustering algorithms. In a recent evaluation study of subspace clustering [24] the algorithms PROCLUS [2] and MINECLUS [32] perform best. Both methods show low runtimes and hence are suitable for our index construction. Furthermore, they do not generate arbitrarily widespread clusters but compact approximations, which are meaningful for our index, are ensured. PROCLUS extends the k-means idea to subspace clustering and assigns points to its nearest representative. MINECLUS uses (subspace) rectangles of a certain width to identify dense regions, these boxes approximate the clusters. It must be pointed out that even if the input parameters are fixed, we can use PROCLUS for our train-and-test method. The non-deterministic algorithm generates varying clusterings. However, MINECLUS as a deterministic algorithm is not applicable but we still can generate one clustering per node. We analyze this difference in the experiments.

3.2.2 Dynamic construction

In the next section we describe the dynamic construction of the index, i.e. we consider the insertion of new objects and the deletion of existent objects.

Insertion. The insertion method makes use of an already existent SUSHI index. First, we identify the leaf node whose enclosing rectangle (which is one layer above the leaf) shows the smallest $minDist$ w.r.t. the object we want to insert. The objects in this leaf node are good candidates for clustering with the new object. To assess if this leaf node is truly a good candidate for insertion we check whether the volume of the enclosing rectangle does not increase too much. If it highly increases, we recursively go one layer up in the tree and test the corresponding rectangle. Instead of a leaf node we now analyze inner nodes, thus the object would be inserted in the outlier list. Evidently, all enclosing rectangles including the novel object must be updated to retain a correct approximation of the clusters. With a second step we ensure that our index adapts to new data distributions. Therefore, we include a reclustering step in our insertion method. We monitor the number of objects represented by a node and if this number increases too high, we rearrange the subtree with the methods presented in Section 3.2.1.

Deletion. To delete an object we have to identify the path from the root to one leaf or inner node where the object is stored. Remember that the object could be an outlier and hence is stored in an inner node. The object is removed from the identified node and all enclosing rectangles up to the root are possibly downsized. In the construction phase (cf. Algorithm 1) we do not split up a cluster if the number of objects is below a certain threshold. Thus for the deletion, we have to identify the cluster on the highest level of the currently considered path with too few objects. Its complete subtree is removed and substituted by a leaf node. This procedure prevents that long paths with very small clusters are maintained, leading to large processing times.

3.3 Query processing

We focus on the k-Nearest-Neighbor processing with Euclidean Distance in SUSHI but other types can be easily integrated. In Section 3.1 we approximate the subspace clusters by subspace enclosing rectangles (SERs). To ensure completeness of our index, i.e. no false dismissals are allowed, we have to define a *mindist* which has to be a lower bound for all objects within the underlying subtree: The distance from the query object q to the aggregated information is smaller than the distance to each object in the subtree.

DEFINITION 4. *Minimum distance to SERs*
The mindist between query $q \in \mathbb{R}^{|Dim|}$ and the subspace enclosing rectangle $R = ([i_1, low_1, up_1], \ldots, [i_d, low_d, up_d])$ is defined as:

$$mindist(q, R) = \sqrt{\sum_{j=1}^{d} \begin{cases} (low_j - q_{|i_j})^2 & \text{if } q_{|i_j} < low_j \\ (up_j - q_{|i_j})^2 & \text{if } q_{|i_j} > up_j \\ 0 & \text{else} \end{cases}}$$

Algorithm 2 gives an overview of the query processing. Starting with the root node, a priority queue stores the currently active nodes. At each point in time we refine the node with the smallest *mindist* (line 7). As mentioned before, in each node we are able to store full-dimensional objects. These objects are either outliers or objects from leaf nodes (lines 9 or 10) for which we perform a linear scan to update the temporary nearest neighbors (lines 11-15). Additionally, if the current node is an inner node we add its child nodes

Algorithm 2 kNN queries in SUSHI

```
1: input: query q, result set size k
2: queue = List of (dist, node) is ascending order by dist;
3: queue.insert(0.0, root);
4: resultArray = [(∞, null), ..., (∞, null)]; // k times
5: dist_max = ∞;
6: WHILE(queue ≠ ∅ and queue.nextDist ≤ dist_max)
7:     n = queue.pollFirst;
8:     // scan objects of leaf or possible outliers
9:     IF(n is leaf node) toScan = n.O;
10:    ELSE toScan = n.Out;
11:    FOREACH(o in toScan)
12:        IF(dist(q, o) ≤ dist_max)
13:            resultArray[k] = (dist(q, o), o);
14:            resultArray.sort;
15:            dist_max = resultArray[k].dist;
16:    IF(n is inner node)
17:        FOREACH(SER R in n)
18:            IF(mindist(q, R) ≤ dist_max)
19:                queue.insert(mindist(q, R), R.child);
20: return resultArray;
```

to the queue (lines 16-19). The sorting is based on their *mindist* values. It is important to scan the outliers before analyzing the child nodes. Thereby the value of $dist_{max}$ can be lowered (line 15) and further subtrees can be pruned.

In Sec. 3.1 we mentioned the use of transformation based approaches for cluster approximation. Thus, the *mindist* calculation is also based on this transformation. However, due to our multi-representations we would use different transformations for each node/cluster and hence the query object has also to be transformed several times (necessary for line 18) resulting in inefficient processing. Therefore we focus on the method of subspace enclosing rectangles.

4. EXPERIMENTS

This section is structured as follows: Section 4.1 describes the setup. Section 4.2 studies the different construction strategies and parameters. Finally, Section 4.3 compares SUSHI to several competing approaches.

4.1 Experimental setup

We compare our SUSHI with index structures from different paradigms. The R*-tree [5] is used as a competitor for full space indexing. Global dimensionality reduction approaches are realized by using the R*-tree with PCA reduction or with Cut-Off reduction: for the first we use PCA and remove the dimensions with the lowest information content, for the second we simply remove the last dimensions. As an approach exploiting local correlations we implement the LDR index [8]. Distance-based indexing w.r.t. reference points is realized by the iDistance method [16]. Additionally, the sequential scan is used as a baseline competitor. Since SUSHI is designed for main memory and not for secondary storage, node/page accesses are irrelevant. Instead, as an implementation invariant performance measure, we use the number of distance calculations to the bounding regions and to the data objects/outliers (with equal weight) for all approaches.

We evaluate the performance on several real world and synthetic data sets. We use color histograms in the extended HSL color space (ext. by dimensions for gray values) as features obtained from a data set that combines well-known image databases (Corel, Pixelio, Aloi, Hemera). We use the UCI pendigits data [11] and extend it to a 48-dimensional variant by interpolation of the available polylines. Moreover, a 15-dimensional data set reflecting oceanographic characteristics as temperature and salinity of the oceans is used[1]. For synthetic data, we follow the method in [13, 18] to generate density-based clusters in arbitrary subspaces. The generator takes into account that subspace clusters can be hierarchically nested (cf. Fig. 1) by allowing a varying subspace cluster hierarchy depth. Unless stated otherwise, we generate data with 10,000 objects, 64 dimensions, 16 clusters in a hierarchy of depth 4, and 5 percent noise (outliers w.r.t. clustered objects) per level of the hierarchy.

For repeatability and comparison we specify the default parameter settings used in our experiments. We measure the number of distance calculations to obtain the 5 nearest neighbors averaged over 100 queries following the data distribution. We set *minSize* (cf. Alg. 1) and the number of clusters for PROCLUS to 20. The test set contains 50 objects and $stableSteps = 5$. For the LDR approach, we use $Minsize = 10$ and $FracOutliers = 0.1$ as in the original publication [8]. According to [16], we set the number of reference points in iDistance to 64 and they are determined by k-means. The node size for the R*-tree methods is set to 4kb. All dimensionality related parameters (average dimensionality of cluster in PROCLUS; number of retained dimensions in the PCA and Cut-Off approach; maximal dimensionality of clusters in LDR) were optimized for each data set such that the best query performance is obtained.

4.2 Evaluation of construction strategies

We start by evaluating the different construction strategies of SUSHI, i.e. the train-and-test method and the applied clustering algorithms. Furthermore, we study several parameter settings of our index.

Train-and-Test. First, we analyze how the train-and-test strategy influences the query performance of the PROCLUS-based version of SUSHI. Figure 4 compares the efficiency of PROCLUS-based SUSHI with or without train-and-test for varying average dimensions per subspace cluster, a parameter of PROCLUS. The number of average distance calculations for the variant without train-and-test is averaged over 5 constructed indices, while the optimized variant uses just one index. That is, the results are averaged over $5 \cdot 100$ queries or 100 queries, respectively. This procedure was chosen because the variant without train-and-test is very unstable, i.e. the nondeterminism of PROCLUS creates very different indices in each construction phase. Therefore, the query efficiencies of the different indices vary considerably. In the figure, this is pointed out by the whiskers showing high variances in the numbers of distance calculations. The train-and-test-based method clearly outperforms the other variant. Considering the best-case situations of the variant without train-and-test, i.e. the lower ends of the whiskers, the train-and-test approach, averaged over only one index,

[1]provided by the Alfred Wegener Institute for Polar and Marine Research, Bremerhaven, Germany

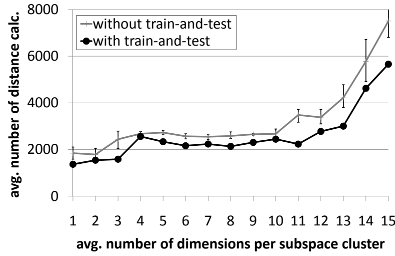
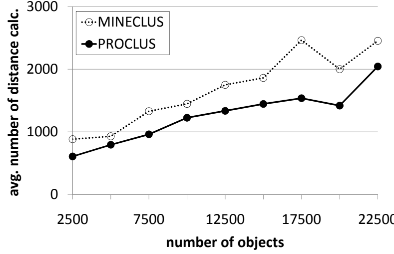


Figure 4: Train-and-test for PROCLUS on the oceanographic dataset



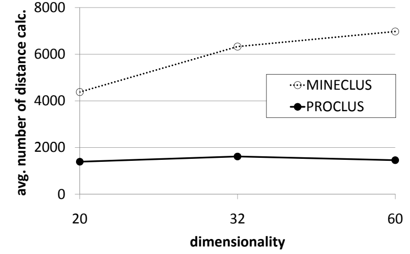
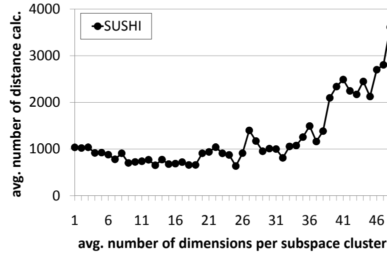


Figure 5: MINECLUS vs. PROCLUS; oceanographic dataset (left); color histograms (right)



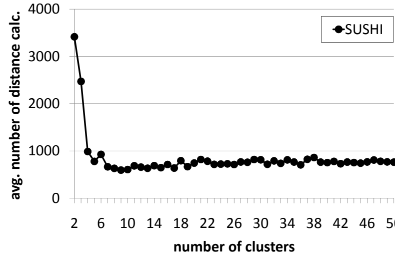


Figure 6: Parameter evaluation of PROCLUS on pendigits; average dimensions (left); number of clusters (right)

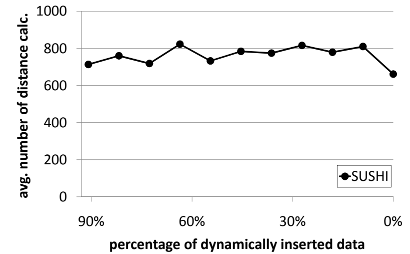


Figure 7: Dynamic inserts on pendigits

dominates over all dimensionalities. Train-and-test significantly improves the stability of the clustering and thus the quality of PROCLUS-based SUSHI. Because of the good results, PROCLUS-based SUSHI is always combined with train-and-test in the rest of the experiments.

Mineclus vs. Proclus. Next, we study the differences between MINECLUS and PROCLUS-based SUSHI. Figure 5 compares the approaches on two datasets. To give a better overview, we evaluated two different parameters, i.e. the database size for the oceanographic dataset and the dimensionality for the histograms. For MINECLUS, most of the recommended parameter settings from the original publication [32] were applied: $\beta = 0.25$, $maxOuterIterations = 100$, and $numBin = 10$. The side length w of a hypercube was automatically determined by a heuristic proposed in [26] and is based on a sampling set of 2,500 objects. Because we observed a higher performance with lower values of α, we fixed it at $\alpha = 0.001$. In both experiments PROCLUS-based SUSHI shows a substantially higher performance than MINECLUS-based SUSHI. This is especially the case for higher dimensionalities of the histogram dataset; runtimes of PROCLUS-based SUSHI stay relatively constant with increasing dimensionality, while the performance of MINECLUS-based SUSHI degrades. The experiments point out that the subspace clusters generated by PROCLUS are better candidates for SERs than the ones generated by MINECLUS. For the remaining experiments, PROCLUS-based SUSHI is the method of choice.

The experiments in Figure 6 show how the different parameters of PROCLUS influence the performance of SUSHI. The studied parameters are the average dimensionality of subspace clusters and the number of clusters to be found.

Average dimensionality. The experiment to the left of Figure 6 shows that the query performance of SUSHI is highly influenced by the average dimensionality of the found subspace clusters. Most interesting, with higher dimensionalities (≥ 20) query performance rapidly deteriorates; we can infer that the usage of local correlations in the data can improve the performance of our index significantly. Accordingly, higher average dimensionalities should not be used as parameters. Furthermore, very low average dimensionalities are also no good parameter setting. There is not enough information in such low-dimensional subspace clusters that could be used to create an efficient index.

Number of subspace clusters. The experiment to the right of Figure 6 displays how the query performance is influenced by the number of clusters to be found. The general tendency is that with a higher number of clusters the average number of distance calculations becomes more stable, i.e. the results for a low number of clusters are very fluctuant and show no good query performance. As a result, parameterizing PROCLUS with a low number of clusters should be avoided. The effect is, however, very dataset dependent. Based on several experiments with different datasets we made a tradeoff and selected a cluster number of 20.

Influence of dynamic inserts. The experiment depicted in Figure 7 studies how dynamic inserts influence the query performance of SUSHI. The x-axis describes to which percentage the index is based on dynamic inserts. Before the dynamic inserts, the rest of the data objects is inserted statically, a process which is knows as bulk-loading. Accordingly, the database size is always fixed. The dataset is pendigits with 10,992 objects. Overall, the results are very stable. That is, the benefits of bulk-loading the index are negligible. Only if the whole index is built with bulk-loading (i.e. 100% static), a slight performance increase can be noticed. The overall good performance can be explained by the insert strategy based on dynamic reclustering. It adapts to new data distributions making SUSHI well suited for dynamic application domains.

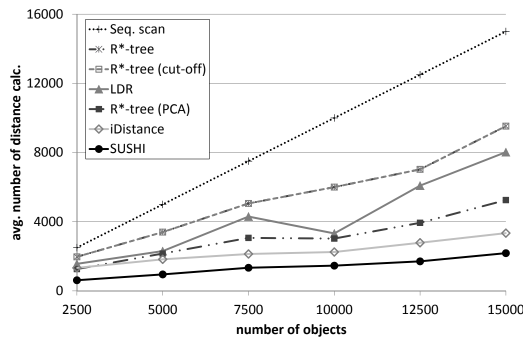


Figure 8: Database scalability on color histograms

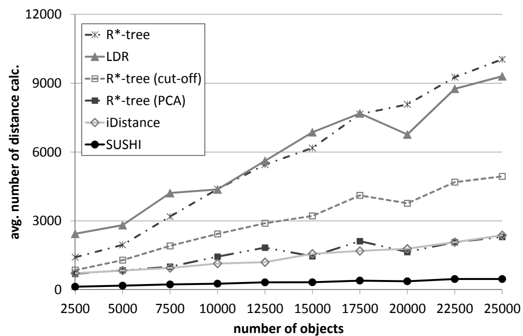


Figure 9: Database scalability on synthetic data

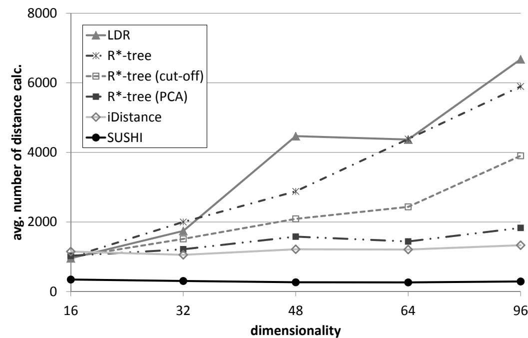


Figure 10: Varying dimensionalities on synth. data

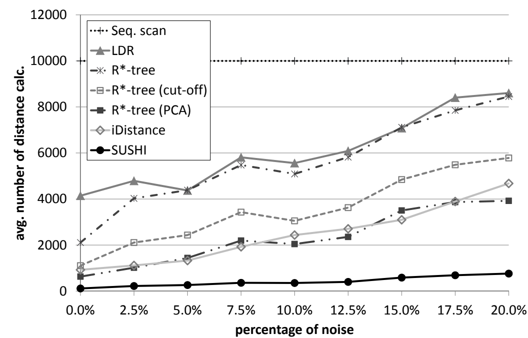


Figure 11: Varying degree of noise on synthetic data

4.3 Comparison with competing approaches

In the following experiments we compare the query performance of SUSHI to the ones of other approaches.

Database size. In Figure 8 we evaluate a varying database size by using subsets of our color histogram database. Our SUSHI approach requires very few distance calculations and the slope of the curve is small. All competing methods show a higher increase w.r.t. the database size. Especially the LDR method, which is able to use local correlations, shows worse performance than the PCA based method, which detects only global ones. Please note, that in this experiment the R*-tree with and without Cut-Off reduction show the same poor performance. This is due to the characteristics of the data set: the dimensions representing gray values have a high information content.

In Figure 9 we analyze the effects of varying database size on a synthetic data set. Comparable to the first experiment, we see that SUSHI clearly outperforms all other approaches. In this diagram, we skip the sequential scan corresponding to a diagonal for clarity. The LDR approach cannot cope with the hierarchy of subspace clusters, it uses subspace clustering only once. The performance degenerates to the one of the R*-tree or is even worse. Furthermore, using classical index structures after the clustering step within the LDR method leads to similar problems as for the R*-tree.

Dimensionality. In Figure 10 we increase the dimensionality of the data set. For SUSHI and iDistance no increase in the number of distance calculations is observed, while all the other approaches have increasing numbers of calculations. Even for the 96-dimensional data set the efficiency of our approach is high. The characteristics of LDR and the R*-tree methods are similar to the previous experiment. The sequential scan constantly needs 10,000 calculations and is orders of magnitude slower than our SUSHI.

Noise. Now we analyze the effects of an increasing percentage of noise in the data. Noise is present in nearly all data sets and the index structures should handle this as well. Figure 11 demonstrates the strength of SUSHI. While its processing time stays low, some competing approaches even converge to the sequential scan. Our SUSHI is able to store outliers, i.e. noise objects, separately in the inner nodes of the index. Thus, our subspace enclosing rectangles are not influenced by the outliers and we reach compact representations of the objects. By contrast, the R*-tree must include all outliers and in iDistance the selection of reference points is sensitive to noise, leading to poor performance of both paradigms. Again and in contrast to SUSHI, the LDR cannot identify the cluster hierarchy.

Information content: Depth of subspace cluster hierarchy. It was shown that high dimensionalities do not necessarily cause the curse of dimensionality but the information content in the data itself [9]. We show that our SUSHI handles this information-rich data better than the other approaches, because we identify and use more structure from the data like hierarchically nested clusters. Measuring the information content of data can be done multifaceted, e.g. in factor analysis one uses the eigenvectors of a covariance matrix to represent factors and the eigenvalues as indicators for the explained variances by each of these factors [17]. Large eigenvalues correspond to factors explaining an important amount of the variability in the data. A data set with low information content has few large eigenvalues and several

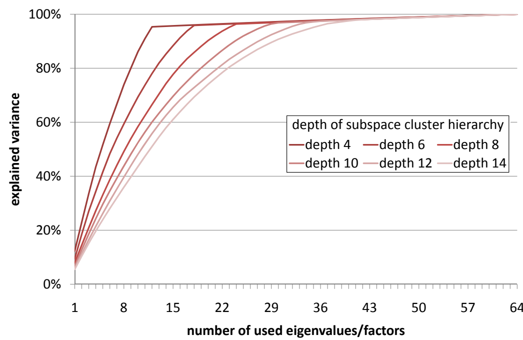


Figure 12: Variance vs. used eigenvalues; synthetic data with hierarchical depths 4-14

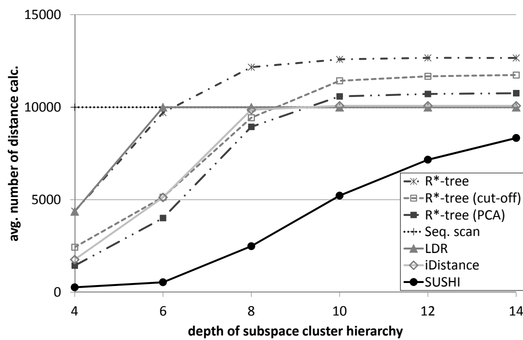


Figure 13: Performance on synth. data of Fig. 12

small ones, and the data can be represented by only few factors. By contrast, data with high information content can result to almost equal eigenvalues. Each factor is important to explain the data.

Figure 12 visualizes the information content for several data sets (one curve per data set). Each data set consists of 64 dimensions and hence 64 eigenvalues are calculated. On the x-axis the number of used eigenvalues/factors to explain the data are presented. The eigenvalues are sorted in descending order such that the factors which explain most of the variances are considered first. The y-axis shows the percentage of the explained variances w.r.t. the overall variance. For example, if we use all factors (right side of the diagram) we can explain the whole data (100%). If we use only two factors we can explain only a part of the variance, e.g. 20%. A data set with low information content reaches nearly 100% with only few factors, while a data set with high information content tends to be a diagonal in the diagram.

To be concrete, in Figure 12 we generate synthetic data sets with varying hierarchy depth for nested subspace clusters. If the hierarchically nesting is only of depth 4 (left curve), the data contains low information. If we increase the nesting of subspace cluster, the curves slide to the right and hence we increase the information content.

In Figure 13 we analyze the query performance of SUSHI and its competitors on these datasets. Keep in mind that the dimensionality and the database size is fixed. We only increase the information content. As expected the performance of all methods drops. However, our SUSHI outperforms all other approaches because we can identify the hierarchies in the data and we use this information for pruning.

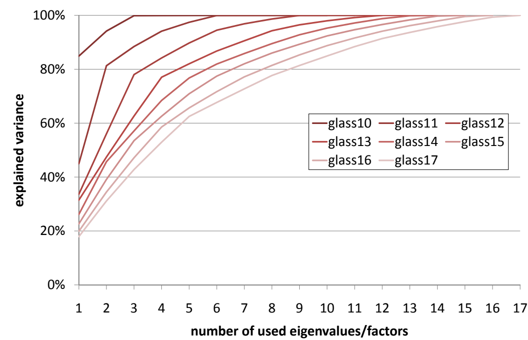


Figure 14: Variance vs. used eigenvalues; semi-real glass data (shifted by i)

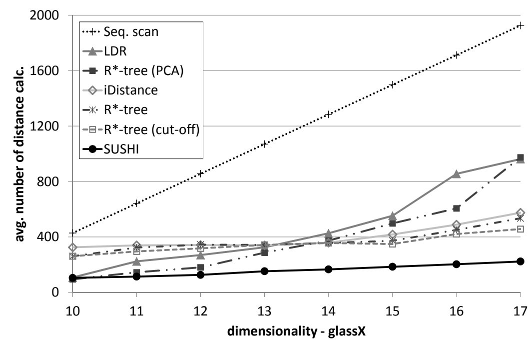


Figure 15: Performance on semi-real data of Fig. 14

All R*-tree variants degenerate and show even worse performance than the sequential scan. The curse of dimensionality becomes apparent. The LDR methods reveals acceptable results only for small hierarchy depths but converges fast to the sequential scan. The method cannot identify meaningful structures and hence nearly all objects are included in the outlier list which results in a sequential scan behavior. Similarly, iDistance fails on these complex data sets since the indexed distances are no longer discriminable and hence all objects need to be processed.

Information content: Real world data. While in the previous experiment synthetic data was used, we now use real world data. Since we cannot directly influence the information content of real world data we modify the dataset and we obtain semi-real data. In the next experiment, our method for varying information content is the following: Assume a database DB with d dimensions is given. To increase the dimensionality by x we replicate the database $x + 1$ times. The new database DB' consists of $x + 1$ different instances DB_i. Thereby, each DB_i is obtained from the old DB by shifting all attribute values of the objects i dimensions to the right, as illustrated in Figure 18. The gray shaded cells are filled with random values. Please note that the dimensionality and the database size is modified by this procedure. In our experiments, we use the UCI glass data [11] to create our semi-real data. We generate different data sets starting with 10 dimensions (glass10) up to 17 dimensions (glass17).

Just as in Figure 12, Figure 14 shows the explained variance by the used eigenvalues/factors. Indeed our method yields an increase in information for the data sets. In Figure 15 the efficiency of the indexing approaches on these data sets

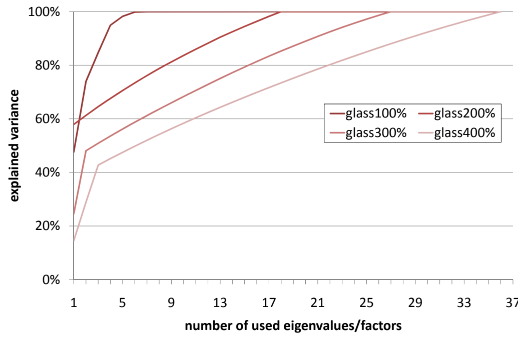


Figure 16: Variance vs. used eigenvalues; semi-real glass data (shifted by $d \cdot i$)

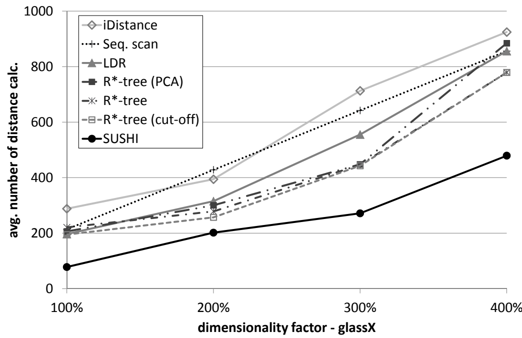


Figure 17: Performance on semi-real data of Fig. 16

is presented. As in the previous experiment SUSHI yields the highest performance. Keep in mind that the dimensionality and database size increase simultaneously and hence the sequential scan does so too. Anyhow, the slope of the curve for SUSHI is very small. An interesting observation is that with increasing complexity of the data set the PCA approach performs worse compared to the classical R*-tree.

In the following experiment we change the information content by a different method: Instead of incrementally shifting the replicated databases by one dimension, we directly shift the databases by a factor of d. Thus, each DB_i is obtained from the old DB by shifting all attribute values of the objects $d \cdot i$ dimensions to the right. By this method, the dimensionality of the obtained semi-real data increases much faster compared to the previous method. In the experiment we generate data starting with the dimensionality of the original glass data (100%) up to a factor of 400%.

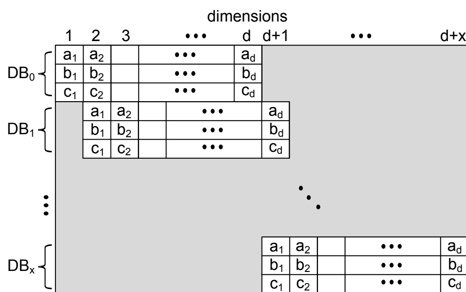


Figure 18: Generation of semi-real data

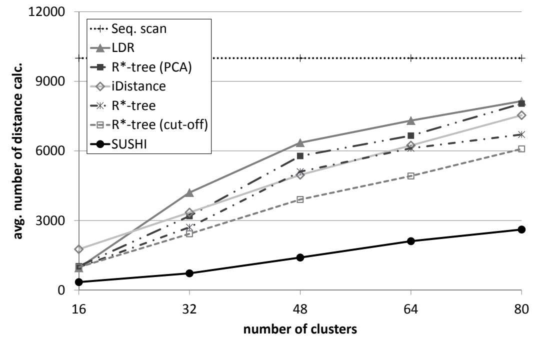


Figure 19: Varying number of clusters on synthetic data sets

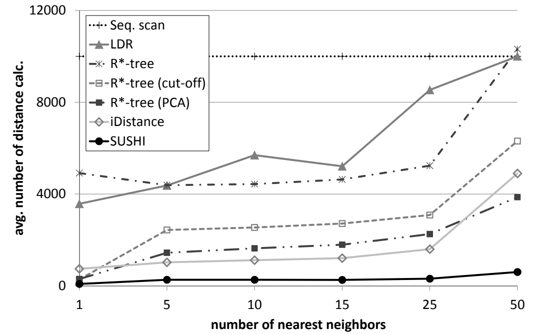


Figure 20: Performance for different kNN queries

In Figure 16 the explained variances by the used eigenvalues/factor are depicted. As illustrated, this data generation procedure also increases the information content of the data. Figure 17 shows the corresponding efficiency of the indexing approaches on these data sets. Similar to the previous experiments, SUSHI performs best on data sets with a high amount of information. All competing approaches show higher numbers of calculations and most of them quickly converge to the poor performance of the sequential scan. A special case is the iDistance, which performs even worse than the sequential scan.

Number of clusters. In Figure 19 we evaluate the query performance when the number of subspace clusters in the data is increased, thus yielding an increase in the local correlations of the data since each cluster has its individual relevant dimensions. Our SUSHI can detect these local correlations resulting in the highest efficiency. The global dimensionality reduction PCA exhibits a high number of calculations, even worse than the classical R*-tree. With increasing number of subspace clusters global correlations become negligible and cannot be used for effective pruning.

Number of nearest neighbors. The performance of the methods under varying result size for the kNN query is studied in Figure 20. SUSHI shows a nearly linear behavior while the other approaches rise faster. The LDR even degenerates to the poor performance of the sequential scan. Some index structures keep up with SUSHI for the 1-NN query. However, the potential of SUSHI becomes apparent for nearest neighbor queries with larger result size. These queries are more relevant in practical applications.

5. CONCLUSIONS

In this work, we introduce SUSHI for indexing high dimensional objects. Our novel model uses subspace clustering to identify local reductions that achieve higher information content than global reductions. By a hierarchical nesting of local reductions we generate a multi-representation of objects, so that queries have to traverse a cascade of different filters in the index. Our index construction is optimized via a train-and-test method that provides compact descriptions for regions in the feature space and we ensure that our index adapts to new data distributions. Thorough experiments on real and synthetic data demonstrate that SUSHI enables fast query processing and reliably outperforms existing approaches.

Acknowledgment. This work has been supported by the UMIC Research Centre, RWTH Aachen University, Germany.

6. REFERENCES

[1] E. Achtert, C. Böhm, H.-P. Kriegel, P. Kröger, I. Müller-Gorman, and A. Zimek. Finding hierarchies of subspace clusters. In *PKDD*, pages 446–453, 2006.

[2] C. Aggarwal, J. Wolf, P. Yu, C. Procopiuc, and J. Park. Fast algorithms for projected clustering. In *SIGMOD*, pages 61–72, 1999.

[3] C. Aggarwal and P. Yu. Finding generalized projected clusters in high dimensional spaces. In *SIGMOD*, pages 70–81, 2000.

[4] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*, pages 94–105, 1998.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[7] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *IDBT*, pages 217–235, 1999.

[8] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *VLDB*, pages 89–100, 2000.

[9] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *PODS*, pages 4–13, 1994.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[11] A. Frank and A. Asuncion. UCI machine learning repository http://archive.ics.uci.edu/ml, 2010.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[13] S. Günnemann, E. Müller, I. Färber, and T. Seidl. Detection of orthogonal concepts in subspaces of high dimensional data. In *CIKM*, pages 1317–1326, 2009.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[15] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2-3):107–145, 2001.

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

[17] I. Joliffe. *Principal Component Analysis*. Springer, New York, 1986.

[18] K. Kailing, H.-P. Kriegel, and P. Kröger. Density-connected subspace clustering for high-dimensional data. In *SDM*, pages 246–257, 2004.

[19] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *TKDD*, 3(1), 2009.

[20] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.

[21] T. Mitchell. *Machine Learning*. McGraw Hill, New York, 1997.

[22] G. Moise, A. Zimek, P. Kröger, H.-P. Kriegel, and J. Sander. Subspace and projected clustering: experimental evaluation and analysis. *Knowl. Inf. Syst.*, 21(3):299–326, 2009.

[23] E. Müller, I. Assent, S. Günnemann, R. Krieger, and T. Seidl. Relevant subspace clustering: Mining the most interesting non-redundant concepts in high dimensional data. In *ICDM*, pages 377–386, 2009.

[24] E. Müller, S. Günnemann, I. Assent, and T. Seidl. Evaluating clustering in subspace projections of high dimensional data. In *VLDB*, pages 1270–1281, 2009.

[25] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Explorations*, 6(1):90–105, 2004.

[26] C. M. Procopiuc, M. Jones, P. K. Agarwal, and T. M. Murali. A monte carlo algorithm for fast projective clustering. In *SIGMOD*, pages 418–427, 2002.

[27] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.

[28] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165, 1998.

[29] H. T. Shen, X. Zhou, and A. Zhou. An adaptive and dynamic dimensionality reduction method for high-dimensional indexing. *VLDB J.*, 16(2):219–234, 2007.

[30] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[31] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[32] M. L. Yiu and N. Mamoulis. Frequent-pattern based iterative projected clustering. In *ICDM*, pages 689–692, 2003.

[33] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.