QuEval: Beyond high-dimensional indexing à la carte

Martin Schäler
University of Magdeburg
Magdeburg, Germany
schaeler@ovgu.de

Alexander Grebhahn
University of Passau
Passau, Germany
grebhahn@fim.uni-passau.de

Reimar Schröter
University of Magdeburg
Magdeburg, Germany
rschroet@ovgu.de

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sanschul@tu-bs.de

Veit Köppen
University of Magdeburg
Magdeburg, Germany
vkoeppen@ovgu.de

Gunter Saake
University of Magdeburg
Magdeburg, Germany
saake@ovgu.de

ABSTRACT

In the recent past, the amount of high-dimensional data, such as feature vectors extracted from multimedia data, increased dramatically. A large variety of indexes have been proposed to store and access such data efficiently. However, due to specific requirements of a certain use case, choosing an adequate index structure is a complex and time-consuming task. This may be due to engineering challenges or open research questions. To overcome this limitation, we present QuEval, an open-source framework that can be flexibly extended w.r.t. index structures, distance metrics, and data sets. QuEval provides a unified environment for a sound evaluation of different indexes, for instance, to support tuning of indexes. In an empirical evaluation, we show how to apply our framework, motivate benefits, and demonstrate analysis possibilities.

Keywords

High-dimensional index selection & tuning, evaluation framework.

1. INTRODUCTION

In recent years, the amount of multimedia data increased dramatically. Reasons are proliferation of Web 2.0 technology and digitalization of previously analog processes, such as digital acquisition of biometric data in forensic use cases. Consequently, it is important to access these data efficiently (e.g., for similarity or range queries). To this end, many index structures have been proposed that aim at accelerating the search in high-dimensional data (e.g., [8, 9, 13, 21, 27]).

Due to numerous available index structures, it is a non-trivial task to choose the best index for a given problem. Amongst others, dimensionality and amount of data as well as search facilities (exact vs. nearest-neighbor vs. range queries) play a pivotal role for the performance of index structures. In prior work, we give an overview of existing impact factors for choosing an optimal high-dimensional index structure [20]. Finally, we argue that developing new index structures or building combinations of existing ones, for instance to tailor them for a specific domain, requires a unified environment with a certain amount of alternative index structures to demonstrate benefits and limitations of a new approach.

In this paper, we propose QuEval [1] (**Qu**ery **Eval**uation Framework) an open-source framework to address the aforementioned issues. Our framework supports *researchers* as well as *practitioners*. In particular, we make the following contributions:

- We introduce QuEval, a framework that can be flexibly extended w.r.t. index structures, distance metrics, and data sets.
- With QuEval, we provide a unified environment to benchmark and tailor index-specific parameters to user-specific use cases.
- We demonstrate usability and show the benefits of QuEval based on real-world examples. Furthermore, our evaluation procedure can be seen as a general guideline to fairly benchmark newly proposed index structures.

The remainder of this paper is structured as follows. In Section 2, we briefly explain the core components of our evaluation framework QuEval. Then, in Section 3, we introduce four motivating real-world examples that require high-dimensional index support. Based on these examples, in Section 4, we subsequently define 12 evaluations using eight additional artificial data sets to demonstrate purpose and benefits of our framework. In Section 5, we perform a large experimental case study. The results indicate a totally different index-structure selection to supports the aforementioned use case characteristics (e.g., dimensionality, or amount of data). We conclude the paper with presentation of related work (Section 6) and future work (Section 7).

2. THE QuEval FRAMEWORK

In this section, we give an overview on the architecture of our Java-based framework. We introduce selected features that make QuEval unique compared to existing evaluation frameworks. Finally, we explain how others can extend our framework to their specific purposes and requirements.

2.1 Overview and General Architecture

In the following, we introduce core components and the *test case* facility, which allows configuring QuEval for a specific experiment. Then, we present advanced features of QuEval. In Figure 1, we show the architecture of our framework consisting of three major parts: (1) Data-Generator, (2) Query-Point Generator, and (3) HDI-Tester.

[1] `queval.de`

The *Data-Generator* creates new *data sets*, each with a certain *number of dimensions*, *amount of points*, *value domain*, and *stochastic distribution* (see Figure 1), which can be specified by a user. These data sets can be re-used in multiple experiments and thus, establish a sound basis for comparable evaluations.

The *Query-Point Generator* utilizes a generated data set to create a set of query points with the same properties (e.g., dimensionality) as the data set it belongs to. The framework uses these new points to query all indexes in a test case for evaluation purposes. Our framework can re-use query points for several test cases.

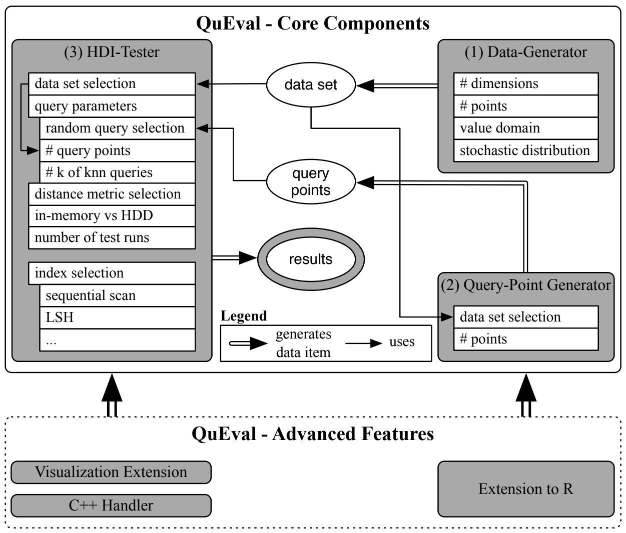


Figure 1: Architecture of our QuEval framework.

The *High-Dimensional Index Tester (HDI-Tester)* as another central component of our framework performs the evaluation of a certain test case using a description that contains:

- *Data-set selection.* A generated data set of the Data-Generator, or a point collection stored as comma separated values (CSV) must be selected.
- *Query parameters.* A user chooses a set of query points that are generated by the Query-Point Generator for exact-match queries. Furthermore, the user can define an amount of query points from the data set for exact-match and knn queries. Additionally, she is able to specify the k for knn queries.
- *Distance-metric selection.* Independent of the index structures, a user can select a distance metric (Minkowsky family, Bray-Curtis, etc.) for knn queries.
- *In-memory vs. HDD.* Since we need to consider different storage devices, it is possible to store the data accessed by an index either in main memory or on HDD. In future, we want to introduce a third option having a user-specific penalty for accessing a point, to simulate for instance SSDs.
- *Number of test runs.* It is possible to state the number of runs for each scenario. Hence, the user can compute robust mean values of response times to ensure statistical soundness.
- *Index-structure selection and index-parameter configuration.* For each selected index, the user can adjust the configuration parameters to find the optimal configuration for a specific scenario. Currently, the user can select between sequential scan as reference, R-Tree variant (covering original R-Tree [21] and R*-Tree [6]), k-d Tree [7], Pyramid Technique [26], Prototype Based Approach [19], VA-File [31], p-stable LSH [15], and M-tree [14]. The extension of the available index structures is subject of current development (e.g., B+Tree extensions based on arbitrary space-filling curves).

A user creates a scenario description either in a graphic user interface (GUI) or uses configuration files, allowing to schedule several test cases (scripted variant). As a result, the user can analyze the behavior of index structures for different scenarios. To evaluate the selected indexes, the *HDI-Tester* determines *accuracy* and *time efficiency*. In the case that our framework computes accuracy (based on the distance metric), it relies on the sequential scan to determine the correct query response and compares this result to the query result of the evaluated indexes. After running a test case, the *HDI-Tester* saves the results in a CSV file for further analyses. Besides the introduced core components of QuEval, we offer *advanced features*, which we describe next (cf. Figure 1).

- *Visualization extension.* This extension provides the opportunity of visualizing *how* the partition of the data space takes place. For instance, for the R-Tree variant, we can use this facility to demonstrate effects of different split algorithms for R-Trees [12] (e.g., for educational purposes).
- *C++ handler.* Since we are aware that many implementations are written in C++, we offer a possibility to include C++ implementation into QuEval similarly to Java implementations.
- *Extension to statistic language R.*[2] For supporting analysis of complex stochastic distributions, such as multivariate Gaussian distribution (MVG) [4], there is a QuEval extension to produce a data set for evaluation purposes via R.

2.2 Possible Extensions

One of our primary design goals of QuEval is to offer several extensibility options. As a result, a user can integrate additional index structures or distance metrics.

2.2.1 Index-Structure Extensibility

Initially, we provide seven index structures with QuEval. However, for other user-specific index structures we provide a mechanism that allows seamless integration with minimum implementation effort. Indeed, a programmer only needs to provide a class file that extends an abstract class serving as basis for accessing all index structures.[3] Within the abstract class, there are three types of methods that should be implemented: (1) general purpose methods, (2) access methods, and (3) optional glue code for the GUI-based variant of QuEval. General purpose methods provide information about the name of the index and current parameter configuration. These methods require about one line of source code. Access methods include usually an insert method to build the index and methods for each supported query type. Since the existing index structures use the same mechanism, they can be used as a starting point (i.e., template) for adding new index structures. Optional glue is required to set index specific parameters via GUI (cf. Section 4.2). For the GUI variant one additional method needs to be implemented, otherwise default parameters values are used. Note, in the scripted version, we can set parameters without this method.

2.2.2 Distance-Metrics Extensibility

In addition to existing distance metrics (Minkowsky family, Bray-Curtis, etc.), it is possible to extend QuEval with further distance metrics. We provide an abstract class, which can be extended to implement own distance metrics. A user has to implement for an additional metric two methods computing the distance of two points

[2] `www.r-project.org`

[3] `queval.de/docs/AIndexStructure.htm`

w.r.t. the points data type (integer or double). For our examples, these methods require usually five to eight lines of code.

3. PURPOSE OF QuEval

Before we present the details of indexing high-dimensional spaces, we introduce four use cases. With the help of these use cases, we later on demonstrate that our framework is useful for many application scenarios.

3.1 Motivating Examples

High-dimensional indexes may be used for various purposes. To demonstrate the application of our framework, we pick use cases from different domains. This way, we show that using QuEval results in benefits for various application scenarios. In the remainder of this paper, we use these examples to demonstrate how to apply QuEval to select a suitable index structure in an *empirical* evaluation.

In a current research project, we are generally interested in the future of crime-scene investigation. Particularly, we deal with the acquisition of latent fingerprints [30]. Thus, we use two examples derived from our current research project. Besides crime-scene investigation there are additional use cases where indexing can be applied. Therefore, we use two different scenarios derived from scientific data management and physical activity recognition. In the following, we introduce the single use cases ordered by increasing dimensionality.

Latent Fingerprint Identification (LFI). Large scale data-intensive systems, such as the automated fingerprint identification system (AFIS),[4] are used by many authorities to support experts in finding similar fingerprints to those collected at crime scenes. For a given latent print, AFIS returns a fixed number of similar fingerprints that are furthermore examined by a human expert. To improve the throughput and response time, we want to speed up the access time similar to the forensic case database. Similarly to the second test case, we have different partitions of the overall fingerprint basis, divided, for instance, by the reason why a print is in the system (suspect, police officer, witness, etc.) and different classes of fingerprints (arch, left loop, etc.). Because of the different challenges such as query type (knn vs. exact-match) and the demand for efficient query processing, we need a solution (i.e., an index) that can address all of these challenges.

Sensor Parameter Tuning (SPT). When examining a potential crime-scene, an important task is latent fingerprint acquisition. Generally, we favor non-destructive optic technologies. However, due to optic characteristics of a surface containing the fingerprint (e.g., degree of light reflection and transparency), we need to adjust sensor parameters, use a different type of sensor, or classical invasive acquisition techniques. Since there is no trivial way to compute these parameters, we use indexes with stored ones for known materials and compute nearest neighbors to adjust these parameters. Our main task is to avoid unsuitable sensor parameter configurations. Hence, we do not need to find the *optimal* configuration and thus, can use approximative index structures. Furthermore, we need to be aware that sensor devices are (to some extent) mobile embedded devices that are for security reasons not linked to large database servers. As a result, we have limited CPU and memory resources.

Scientific Data Management (SDM). In scientific experiments, classification of results via a certain amount of previously classified nearest neighbors is an important task to interpret the results. For instance, in physics, results of particle experiments to identify new elementary particles or determining the fraction (or amount) of a certain particle type are a field of application. Generally, these data sets used for classification are high-dimensional and low-populated data spaces consisting of representatives forming clusters and removed noise. Furthermore, the amount of data from new experiments is large. Thus, efficient determination of nearest neighbors is crucial. However, we do not need to support exact-match queries.

Physical Activity Monitoring (PAM). Similar, to the SDM use case, classification of different physical activities (e.g., walking) can be done by computing the distance to several nearest neighbors with known activity [28]. In contrast, to the SDM use case, we have many data. So, the data space is not that sparsely populated as for SDM use case. Thus, we hypothesize that we have to apply different index structures as for the SDM use case.

3.2 Adopting the Examples for QuEval

We conduct a case study based on the motivating examples to demonstrate applicability and benefits of QuEval in practice. To this end, we define test cases for each of the exemplary use cases that should be evaluated with QuEval. For our evaluation, the selected data sets must fulfill the following requirements to ensure repeatability and scalability: (1) substantial amount of data (at least several thousands), (2) open access to the data set, and (3) different characteristics w.r.t. properties (e.g., dimensionality) to show differences in our exemplary evaluation. In the following, we provide details on the data sets that we use in our evaluation.

Latent Fingerprint Identification. Due to privacy issues, we cannot publish large fingerprint databases and to the best of our knowledge existing *latent* fingerprint databases, such as the NIST database,[5] contain only a few hundred entries. To circumvent this problem, we perform the evaluation with a publicly available data set based on different hand-writing features, which is also used for forensic evidences [2]. The data set has 16 dimensions and contains 10,992 data points.

Sensor parameter tuning. For this use case, we apply a real-world data set based on the spectral texture features used to identify latent fingerprints [25]. The data set has 43 dimensions and contains 411,961 data points.

Scientific Data Management. Here, we use a particle identification data set for evaluation. Particularly, we use a data set having 50 dimensions and 130,064 points [29].

Physical Activity Monitoring. For the PAM use case we use a PAM benchmarking data set published in [28]. The data set has 51 dimensions and contains 3,850,505 points and thus, is the largest in our evaluation.

4. EMPIRICAL EVALUATION

In engineering of data-intensive systems, selection and optimization of adequate index structures is an important task. To demonstrate the application and benefits of QuEval, we present an empirical evaluation for our motivating examples from Section 3. In the first part of this section, we give an overview of index structures used in our evaluation. Second, we describe the setup of our evaluation.

4.1 Index-Structure Selection

Next, we motivate the selection of index structures used in our evaluation and describe their concepts. Since we are aware that many index structures exist in the literature, we need to restrict the

[4]`https://www.interpol.int/Public/Forensic/Fingerprints/RefDoc/default.asp`

[5]`www.nist.gov/itl/iad/ig/latent.cfm`

amount of structures to promising candidates. First, the considered index structures have to support required query types. As stated in the previous sections, most of our use cases require exact-match *and* knn query capability. Hence, we select indexes that support both query types. Consequently, index structures such as iDistance [24] or M-tree [14], optimized to support knn queries in arbitrary metric spaces, are not considered. Nevertheless, these index structures can be integrated, if needed for user-specific test cases. Overall, we select a combination of tree-based indexes (i.e., *R-Tree* and *k-d Tree*), improved sequential scans (i.e., *VA-File* and *Prototype Based Approach*), and a hashing approach (i.e., *p-stable LSH*) to cover a broad variety of index types based on the classification in [16]. Furthermore, we chose the *Pyramid Technique* which is mentioned as not affected by the curse of dimensionality [8].

4.1.1 R-Tree, k-d Tree, and VA-File

Because R-Tree (including respective variants) and k-d Tree are widely known tree based index structures, we just refer to the papers of Guttman [21] and Bentley [7] presenting these structures. For the same reason, we refer the reader to [31] for details on the VA-File. Our implementation of the k-d Tree is based on an existing variant.[6] Furthermore, we implemented the R-Tree variant and VA-File from scratch. Notably, our R-Tree variant is based on the idea of GIST [22]. Thus, implementing a new R-Tree variant means modifying insert heuristic and split algorithm. As a result, for the subsequent evaluation, insert heuristic and split algorithm are parameters that can be modified.

4.1.2 Pyramid Technique

According to Berchtold et al., the Pyramid Technique is a technique that is not affected by the curse of dimensionality [8], which is an interesting property for our evaluation. The basic idea of this technique is to divide a d-dimensional space into $2d$ pyramids. By definition, a normalized d-dimensional point x is located in the pyramid p_i with the following condition:

$$i = \begin{cases} j_{max} & \text{if } x_{j_{max}} < 0.5 \\ j_{max} + d & \text{if } x_{j_{max}} \geq 0.5 \end{cases}$$

$$j_{max} = (j | \forall k, 0 \leq (j,k) < d, j \neq k : |0.5 - x_j| \geq |0.5 - x_k|)).$$

As a result, a point is inserted according to its dimension that has the largest distance j_{max} to the center of the space. For instance, if the position of this component is j, then the point is inserted in the pyramid p_j if the value is smaller than 0.5 or to the pyramid p_{j+d} if the value is larger or equal 0.5. For efficiently managing the points, the pyramid is divided into several slices.

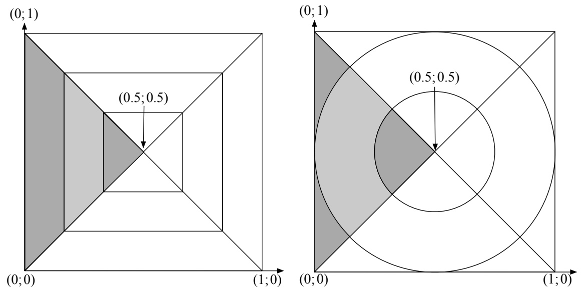


Figure 2: Space partition of a 2-d space by Berchtold et al. [8] (left) and Lee and Kim [26] (right).

Different ways for partitioning pyramids are proposed to support different query types. Berchtold et al. support range queries by splitting a pyramid horizontally according to their basis [8]. In contrast, Lee and Kim divide a pyramid in a spherical way, starting from the top of the pyramid to support knn queries [26]. In Figure 2, we show a visualization of both partitioning approaches for a 2-dimensional space. While the approach of Berchtold et al. refers to hyper rectangles, Lee and Kim split the space into hyper spheres. The latter ensures that all points in one pyramid, having the same distance from top of the pyramid, are located in one slice and thus, efficiently supports knn queries.

Because we are interested in the results of knn and exact-match queries, our own implementation is based on the idea of Lee and Kim [26]. In particular, a d-dimensional point x with a distance of $dist_x$ to the center of the space is located in the slice s where $s = \lfloor \frac{dist_x * |s|}{dist_{max}+1} \rfloor$. A detailed description of the algorithm and proofs that indicate mathematical correctness are given in [26].

4.1.3 Prototype Based Approach

This index structure has been proposed by Gonzalez et al. as *Ordering Permutations* [19]. The main idea of this approach is to compare a specific representation of points in the data set with a representation from the query point. In this case, a specific order of prototypes is used as this representation.

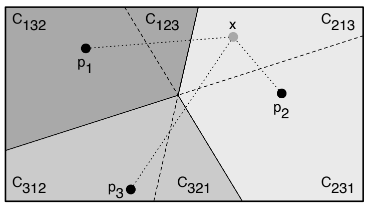


Figure 3: Space partitioning using three prototypes.

Gonzalez et al. define a distinct set of objects P with $P \subseteq X$ where X represents a data set. In Figure 3, the points p_1, p_2, and p_3 are elements of P, which are called *prototypes* [19]. Furthermore, a permutation Π_x, which consists of all prototypes, is computed for each $x \in X$. The order of prototypes in permutation Π_x depends on the distances of point x to the prototypes. For instance, in Figure 3, we insert x into the data space. Prototype p_2 has the smallest distance relative to x and thus, prototype p_2 is in the first position of the permutation. Similarly, the second position is defined by the prototype with the second smallest distance, p_1. As a result, x is assigned to permutation $\{p_2, p_1, p_3\}$. Thus, the data space is divided into a Voronoi diagram [3] using the prototypes as seeds (combination of similar colored cells in Figure 3). So, within a Voronoi cell, all points share the same prototype with the smallest distance. If we extend each line (in three-dimensional spaces it is a plane etc.) of the Voronoi cell with dashed parts, we obtain the cells for the permutations. In this representation, all points in the cells share the same order of prototypes. For instance, all points located in cell C_{213} share the same order of prototypes in their permutation, $\{p_2, p_1, p_3\}$. For our case study, we adopt a C# implementation from [19].

4.1.4 LSH Based on p-stable Distributions

This index structure uses *Locality Sensitive Hashing (LSH)* based on p-stable distributions. The idea of this technique is to map the original data space to a different space by means of hash functions. These hash functions have the property that two points, close to each other, have a higher probability of collision in the hashed space than points that are far away. The LSH technique is introduced by Indyk and Motwani [23] and has been improved and tested by Gionis et al. [18].

[6] `http://home.wlu.edu/~levys/software/kd/`

As a representative of this approach, we implement an index based on p-stable LSH functions first presented by Datar et al. [15]. An example for a p-stable distribution is the Gaussian distribution $N(0, 1)$ where $p = 2$. However, to increase the probability that two points close to each other in the original space have a collision in the hashed space, it is necessary to use different hash functions. For each hash function, we need a random weight vector g_i, with dimensionality of points in the data set. This vector requires independently chosen components from a p-stable distributed function. Additionally, the hashing scheme needs a uniformly random chosen offset o_i between $[0 \ldots w]$, where w is the width of hash buckets. Next, the hash value $h(x)$ of a point x is defined by $h(x) = (\lfloor \frac{\langle g_1, x\rangle + o_1}{w} \rfloor, ..., \lfloor \frac{\langle g_d, x\rangle + o_d}{w} \rfloor)$, where d is the number of hash functions. In Figure 4, we show a 2D space that is partitioned by two hash functions. For instance, point x_2 is a candidate for being a knn of query point q, due to a collision within a bucket of hash function h_2. Although we implement this index from scratch, the implementation is based on the description of Datar et al. [15].

4.2 Case-study Design

In the following, we provide details on important aspects regarding setup and execution of our case study.

4.2.1 Evaluation Setup

Next, we give basic information regarding the setup of the evaluation. This defines, which variation point (e.g., index-specific parameters or stochastic distribution) we need to consider.

Parameters of Selected Index Structures. Most of the introduced index structures exhibit specific parameters that may influence the performance of an index structure. Specifically for the approximative index structures, these parameters have a considerable influence on accuracy of these structures. In the following, we provide an overview of these parameters.

First, the parameters of an R-Tree variant are minimum and maximum number of points in a Minimum Bounding Rectangle (MBR) as well as different insert and split algorithms that try to minimize overlapping MBRs (e.g., R*-Tree). Next, for the Pyramid Technique, the configuration parameter is the number of bounding slices a pyramid is divided into and for the VA-File, it is the number of partitions per dimension. Furthermore, the parameters of the Prototype Based Approach are the number of prototypes and the amount of data points to consider in knn-queries. Finally, the parameters of the LSH based on p-stable distribution are the number of hash functions and the width of the hash buckets. Additionally, both approximative index structures are influenced by randomly created prototypes or weight vectors. For these components, a seed is required, to ensure reproducibility. For a better comparability, we choose the same seed in all test cases. In Table 1, we give an overview of the parameters.

Stochastic Data Distributions. To show differences w.r.t. different stochastic distributions, we add more data sets to the evaluation. Our goal is to determine whether experiences from uniformly distributed or MVG data sets can be taken into account to select an adequate index for real-world data. Thus, for each of the four test cases from Section 3, we add a uniform and an MVG data set having the same number of dimensions and points. For instance, all points in the data sets D^u_{43}, D^{MVG}_{43}, and D^r_{43} have 43 dimensions, but the first data set is uniformly distributed, the second is an MVG data set, and the third is the real-world data set from Section 3.2. As a result, we obtain 12 data sets (three from each test case) of our evaluation (cf. Table 2).

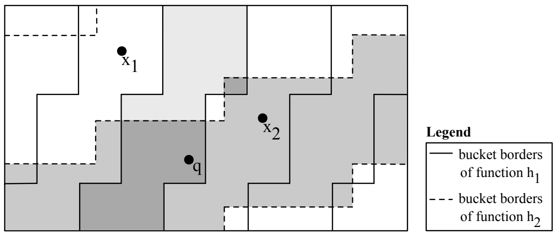


Figure 4: LSH based on p-stable distributions.

Table 1: Parameters of the evaluated indexes.

index structure	parameter
R-Tree variant	`min, max points per MBR,MBR split, insert algorithm`
k-d Tree	`(no parameters)`
Pyramid Technique	`slices per pyramid`
VA-File	`bits per dimension`
Prototype Based Approach	`#prototypes, considered points in %, (random seed)`
p-stable LSH	`#hash functions, width of hash buckets, (random seed)`

Table 2: Overview on our evaluation data sets.

use case	uniform data	MVG data	real-world data
LFI	D^u_{16}	D^{MVG}_{16}	D^r_{16}
SPT	D^u_{43}	D^{MVG}_{43}	D^r_{43}
SDM	D^u_{50}	D^{MVG}_{50}	D^r_{50}
PAM	D^u_{51}	D^{MVG}_{51}	D^r_{51}

Additional Properties. We limit the value domain within a dimension to normalized point data such as [0...255]. For knn queries, we set $k = 10$, based on the experience of other case studies (e.g., [13, 31]). Furthermore, for each experiment run, we determine the 10 nearest neighbors for 100 (pseudo randomly selected) points of the data set. Finally, we state that all indexes are kept in main memory, while the points of the data set (referenced in the index) are located on HDD to consider different access time of different storage devices.

4.2.2 Evaluation Metrics

For comparing different index structures and for optimization of index-specific parameters, we apply several well-known metrics. Note, for different use cases different metrics may be required. For each execution of a test case, we query the index with 100 points pseudo randomly selected from the data set.

T_{resp}. QuEval determines the amount of time required from sending the first query to receiving the last query response.

A_{cc}. For knn queries of approximative index structures, the accuracy is N_f/N_{all}. N_f is the number of correctly found nearest neighbors (in the right position). To determine the correct nearest neighbors, we use a sequential scan. We query 100 points and determine for each the 10 nearest neighbors. So, the total number of nearest neighbors that can be found (N_{all}) is 1000. T^x_{resp} is the smallest T_{resp} configuration of an index with $A_{\text{cc}} \geq x$.

N_{hdd}. This metric describes the average number of points fetched from HDD per query. Our index implementation do not store the data points itself, but a reference, similar to tuple identifiers. Thus, whenever an index needs to fetch a point from HDD, it requests it from QuEval's buffer and storage management system. This allows us to count accessed points. Currently, our storage structures are optimized for sequential reading. Consequently, we can observe different response times for random or sequential access of points. This configuration is especially beneficial for sequential scanning. Hence, we can conclude that each index that outperforms the sequential scan is a valid alternative. In future, we will evaluate different storage and buffer strategies.

M_{heap}. This metric states the maximum amount of main memory allocated by the Java virtual machine (runtime environment). This metric may exclude certain index structures from further considerations. For instance, in embedded environments we may have limited main memory. Consequently, index structures that require more memory cannot be used in this context.

T_{build}. states the index building time. This covers the time span from sending the request of the first point to receiving the confirmation of successful insertion of the last point of a specific data set.

To ensure statistical soundness, we always repeat test cases 120 times and compute a robust mean value per test case using a two-sided γ-trimming ($\gamma = 16.67\%$) to compensate outliers. Mean value computation of execution times has been proved to be appropriate [17] in Java applications.

4.2.3 Test-case Execution

During test-case execution, we have to minimize system-specific effects which may bias our results. Initially, we have to minimize impacts of Java-specific runtime optimization and system effects like garbage collection that can occur while testing performance of Java applications [17]. To this end, we schedule a warm-up phase in advance of each test case. To address the index structure parameters, we divide our case study into two parts. First, we run each test case to determine beneficial parameters for each index structure (*intra-index evaluation*). Second, we use these parameters to run the same test case 120 times for an actual comparison and evaluation of different index structures (*inter-index evaluation*) (cf. Figure 5).

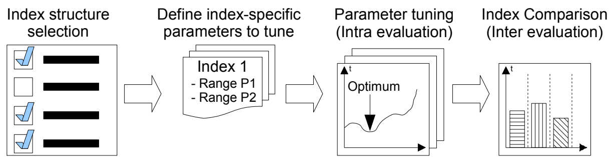

Index structure selection
Define index-specific parameters to tune
Parameter tuning (Intra evaluation)
Index Comparison (Inter evaluation)
Index 1
- Range P1
- Range P2
Optimum

Figure 5: Overall evaluation procedure.

5. EVALUATION RESULTS

In this section, we present the results of our case study. First, we present results of our intra-index evaluation, revealing that parameter optimization is a crucial part for every evaluation of index structures. Second, we present results of our comparison and evaluation of different index structures. All tests are performed on an Intel Core i5 with 8GB main memory using Windows 7 and Java 1.7 (64 Bit). We limit the amount of main memory that can be allocated by our index structures to 6GB. The remaining 2GB are reserved for operating system and different tasks that we cannot control. As a result, index structures that exceed this amount of main memory are unsuitable for this use case and thus, are excluded from this part of the evaluation. Of course, we could use computers having more amount of main memory, but due to our experiences, hardware limitations are still common in practice.

5.1 Intra-Index Evaluation

In the following, we briefly present results of our intra-index evaluation for each index. Our experiences show that this part of the evaluation requires high computation effort. Hence, using QuEval's script functions is helpful to schedule large amounts of tests (in our evaluation several thousands). Furthermore, QuEval optionally notifies the user via email in case that results exist. Generally, we observe large differences among the indexes regarding parameter changes as we point out in the following. To give credit to different classes of indexes, we split the intra-index evaluation into one part for exact and one part for approximative indexes.

5.1.1 Exact Indexes - Verification of Existing Results

In contrast to approximative index structures, exact ones always return the correct knn (the same as the sequential scan does). As a result, we exclude accuracy here and do not consider the trade-off between performance and accuracy. In general, we are able to verify existing results, such as the parameter suggestions from Weber and Blott for the VA-File [31] or the degeneration of tree-based indexes in spaces having more than 16 dimensions [32, 10]. However, determination of the specific influences is still valuable from the engineering perspective.

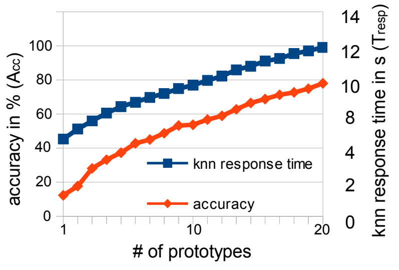
R-Tree Variant. The main challenge of an R-Tree variant is to avoid overlapping MBRs. This problem is even more challenging when the number of dimensions increases, which is a direct consequence of the curse of dimensionality [9]. Furthermore, we have to consider height and fan-out of the tree. Since there are four parameters responsible for the performance of an R-Tree variant, tuning itself is a multi-dimensional problem. However, our results indicate that, for instance, even for D_{16} (affected less by the curse of dimensionality), N_{hdd} indicates that knn search nearly degenerates to a sequential scan (e.g., for D_{16}^{MVG} 85% of the points are retrieved from HDD). Overall, our data reveal that using R*-Tree [6] split and insert delivers best results.

Another interesting/important observation we made is that the R-Tree variant is the only index structure that exceeds the maximum available heap space of 6GB for all three data sets with 51 dimensions (D_{51}). We depict the amount of M_{heap} that is required for this index structure in Figure 9(b) (measured on a computer having 16 GB of RAM). Consequently, we have to exclude this index structure from further considerations for these scenarios. Nevertheless, for the remaining scenarios R-Tree variant is still a valid choice.

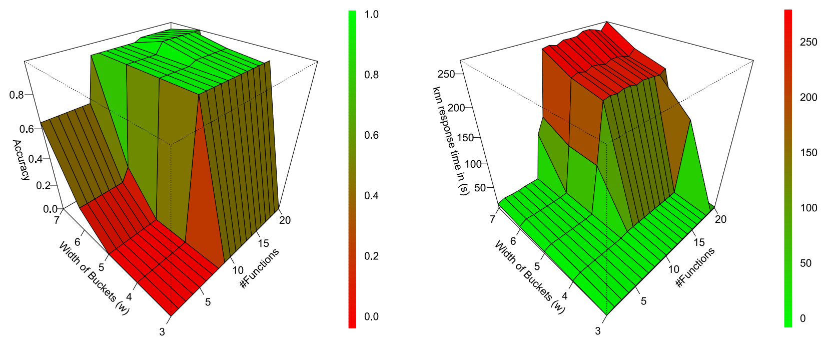
Pyramid Technique. Generally, our implementation verifies that the Pyramid Technique is robust against parameter changes, as stated by [8]. For instance, for D_{50} an exact match requires 1.12 HDD accesses on average for every configuration (8 to 16 `slices`). This is a result of the linear computation effort for determining the (spheric) pyramid slice where a point is located in. We only need the dimension with the largest distance to the center of the pyramid.

Furthermore, for the same scenario, to determine 10 nearest neighbors, the index accesses between 117,363 and 129,155 points from HDD. This is in fact the whole data set having 130,064 points. Thus, increasing the number of `slices` has no benefit. The reason is the curse of dimensionality. Due to the fact that most points are located in one of the corners of the space and `slices` meet in the corners, only a few numbers of `slices` can be excluded for point lookup.

VA-File. The major challenge for tuning knn queries for the VA-File is the trade-off between better approximations due to longer bit vectors to minimize N_{hdd} and additional complex cell distance computations due to longer vectors. For most scenarios, our results show that the best trade-off is using 8 `bits` per dimension, which is slightly more than the recommendation of Weber and Blott [31],



(a) Correlation between # of prototypes, accuracy, and knn response time for Prototype Based Approach



(b) p-stable LSH - Sensor tuning scenario (D^r_{43}): Excerpt of intra evaluation

Figure 6: Characteristic parameter correlation for Prototype Based Approach D^{MVG}_{50} and for p-stable LSH for D^r_{43}.

who suggest to use 4 up to 6 `bits`. For example, in all D_{50}, the response times with 6 up to 16 `bits` per dimension are very similar (maximum difference 10%). However, the average N_{hdd} differs. For example, to determine the 10 nearest neighbors for D^{MVG}_{50}, the VA-File performs approx. 33,908 HDD accesses for 4 `bits` per dimension, 700 for 8 `bits`, and 181 for 16 `bits`. Thus, we conclude that distance computations due to longer bit vectors compensate the benefit of a more precise approximation, which is the same observation as in [31]. To verify our observation, we analyze the computation time with the help of *VisualVM*,[7] a profiler, which is part of Java development kit. As expected, the time saved for HDD accesses is compensated in functions determining whether we have to consider a point or not. Particularly, for 4 `bits` per dimension, point access requires about 20% of computation time, while for 8 and 16 `bits` HDD access consumes about one percent and thus, can be neglected. For exact-match queries, we observed a similar tendency for the same reason.

5.1.2 Parameter Sensitivity of Approximative Indexes

For approximative structures, we have to take the trade-off between query response time and accuracy into account. Thus, we have two optimization goals. Our results indicate that both index structures are very sensitive to parameter changes. To allow a certain amount of false nearest neighbors, we determine parameter $T^{0.9}_{\text{resp}}$ for each approximative index, meaning that at least 90% correct neighbors are required. Without using our framework for test automation, finding these beneficial parameters manually is hardly possible, because for both approximative indexes we need to test about 300 parameter configurations for each scenario per index.

Prototype Based Approach. For the Prototype Based Approach, there are two parameters: (1) number of `prototypes` and (2) `percent of data searched`. In Figure 6.(a), we show the impact of the number of prototypes with a constant percent of data searched for knn queries for D^{MVG}_{50}. If we increase the number of prototypes, parameter T_{resp} deteriorates, but the index delivers better accuracy. Furthermore, the positive effect becomes weaker and the graph looks similar to square root functions (cf. Figure 6.(a)). Finally, these effects are the same for different values of parameter (2), but starting point and slope of the graphs are different. Currently, these effects are hardly predictable. In fact, the space partition is based on slicing Voronoi cells. Summarily, we can create nearly arbitrary convex shaped regions. However, knowledge about the way of creating beneficial space partitions is still fragmentary. Consequently, tuning parameter configurations is time consuming even with our framework, but still more efficient than manual tuning.

P-stable LSH. For our implementation of p-stable LSH, we consider two parameters: (1) bucket `width(w)` and (2) number of `hash functions`. When optimizing these parameters for exact and knn queries, we made two basic observations. First, small bucket `widths` are beneficial for exact-match queries, but impose a low accuracy for knn queries, because there are only a few numbers of collisions (points in a bucket). Hence, to optimally support exact-match and knn queries, we need two indexes with different parameters. Second, for knn queries, the correlation of the parameters and T_{resp} (or N_{hdd}) and resulting accuracy is hardly predictable (cf. Figure 6.(b)). Since the optimization of parameters is tedious (e.g., for every scenario we test several hundred parameter configurations), we try to exclude and limit the parameter sets to certain bounds. As a result of this intra-index evaluation, we made the following observations: First, the time for computing the hash value is constant and thus, cannot be influenced by parameterization. Second, the number of data points for each bucket is similar regarding uniform and MVG data sets. Hence, we can estimate the response time, because we know the number of buckets and that each bucket contains a similar number of data points. However, we cannot predict the accuracy, because we did not observe a correlation between accuracy and accessed points, due to the sparsely populated space.

The *primary result* of this evaluation is that an intra evaluation for every scenario is inevitable to perform an objective comparison of different indexes. To this end, using QuEval's unified test environment and automation features are *highly beneficial* to schedule large amounts of scenario executions.

5.2 Inter-Index Evaluation: Exact Match

Based on the results of the intra-index evaluation, we can now apply our identified parameters to perform a sound comparison of indexes for our scenarios with the help of QuEval. In Figure 7, we depict the results of our exact-match scenarios by means of a bar chart for each scenario. Furthermore, we provide results for all three data sets for each scenario. Hence, for every index in every figure there are three bars showing the average of measured response time of uniform (D^u_x), MVG (D^{MVG}_x), and real-world data (D^r_x), where x indicates the dimensionality (from left to right). Recapitulate, we have to exclude R-Tree variant from the evaluation for D_{51}, because it exceeds the maximum available heap space in our evaluation.

[7]`http://visualvm.java.net/`

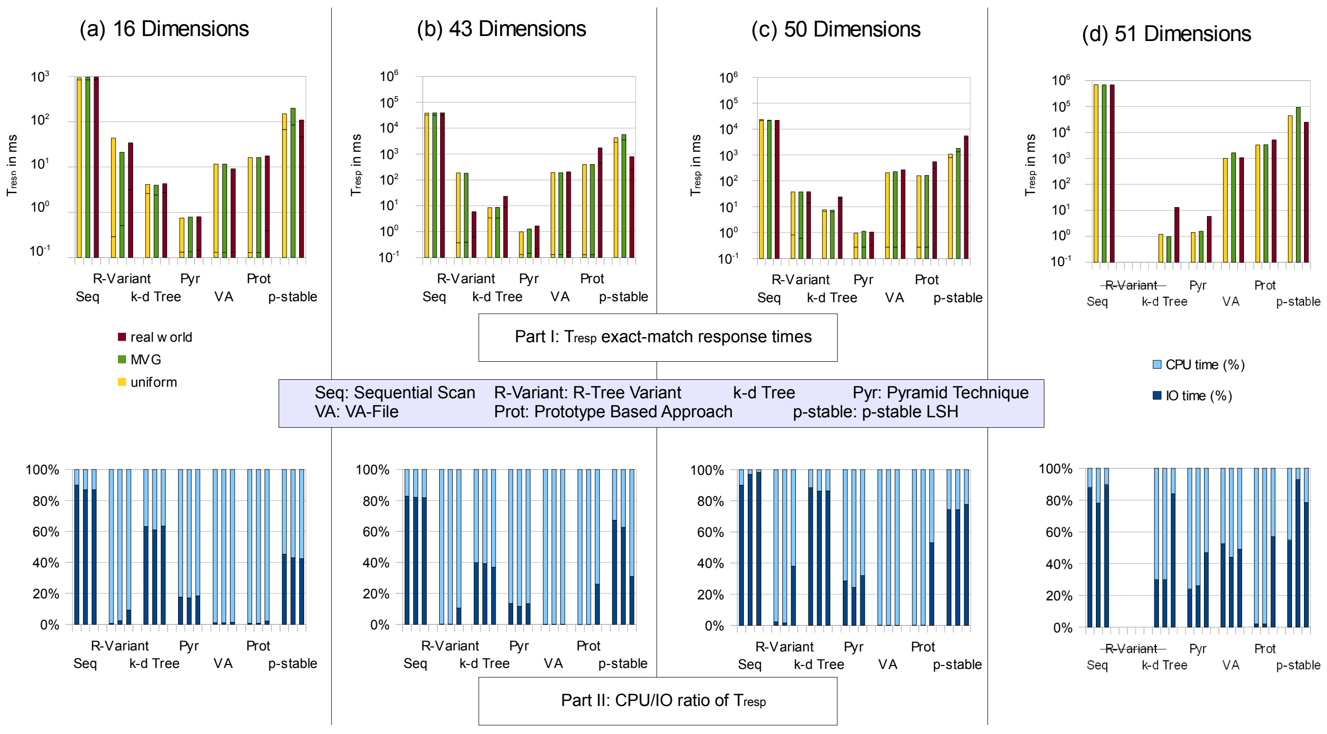


Figure 7: Part I: Exact match T_{resp} in ms (logarithmic scale), Part II CPU/IO ratio.

Thus, we strike out the R-Tree variant in Figure 7(d).

Finally, to distinguish between CPU- and HDD-based indexes as well as to make certain effects explicit, we also depict the amount of in-memory computation time (e.g., comparison of bit strings for VA-File etc.) and the amount of time necessary to fetch points from HDD in Part II of the same Figure. We use an empirically measured approximation of the time necessary for HDD lookups and number of HDD accesses captured by QuEval to compute CPU and HDD ratio. Note that we use logarithmic scales to depict our results.

Main Observations. First, our results reveal that the Pyramid Technique outperforms every other index at least by several factors, except for D_{51} where the T_{resp} of the k-d Tree is quite similar. Furthermore, the response time of this index is very stable even when changing the data distribution, dimensionality, or amount of data (between 0.7 and $7.8ms$). Second, our measurements reveal that Pyramid Technique, Prototype Based Approach, and k-d Tree require more time for real-world data while the p-stable LSH approach seems to benefit from real-world data (except for D_{50}).

Interpretation and Consequences. The explanation for our first observation is that the Pyramid Technique requires only linear effort to determine the pyramid slice where a point is located in [8]. As a result, the Pyramid Technique can *directly* access the searched point within a certain slice. Furthermore, if a slice does not exist, we can conclude that the searched point does not exist in the data set. The reason for the exception in D_{51}^r is a degeneration in the respective real-world data set. Thus, there are several slices having a large amount of points that is also observable in the higher amount of IO time (cf. Figure 7 Part II(d)). Nevertheless, this index requires only 1.45 HDD accesses on average for *all* of our scenarios. Consequently, to speed up exact-match queries in high-dimensional data, we recommend to use Pyramid Technique. For the second observation, we assume that the main reason is that due to the changed distribution the probability of collisions increases. Hence, the average number of points in a bucket or slice increases. For instance, for uniform data, Pyramid Technique requires on average 1.03 HDD accesses, while for real-world data this value is about 1.62. For the Prototype Based Approach this difference is even more obvious. Especially for our real-world data (D_{43}^r), nearly 10% of the data are fetched from HDD, because there is a large number of points having the same permutation (key). To explain this effect, we analyzed the data set with the help of two-dimensional scatter plots. The results reveal a correlation between nearly constant dimensions dominated by some outliers and two-valued dimensions causing this huge amount of collisions.

5.3 Inter-Index Evaluation: knn Queries

Similarly to Section 5.1, we divide the evaluation into a comparison of exact indexes and approximative ones.

5.3.1 *Exact Index Structures*

We depict the results of our knn-query experiments in Figure 8. Analogously to Figure 7, there is a figure for all scenarios having the same dimensionality. Additionally, we explicitly distinguish between exact and approximative indexes, indicated by a vertical, dashed line in each subfigure (i.e., Part I of Figure 8 (a)–(d)). Finally, we also present the results of main-memory consumption in Figure 9, which contains exemplary data of the Sensor Tuning (SPT)and the Physical Activity Monitoring (PAM) scenario. To determine main-memory consumption, we use *jamm*.[8]

Main Observations. First, our data reveal that for sparsely populated spaces (D_{50}) T_{resp} converges to T_{resp} of a sequential scan, independent of stochastic distributions or other properties. Second, the results of our tree-based indexes indicate a considerably better response time for real-world data (at least factor ten) and clearly outperform a sequential scan. Third, for every scenario, the VA-File

[8]https://github.com/jbellis/jamm

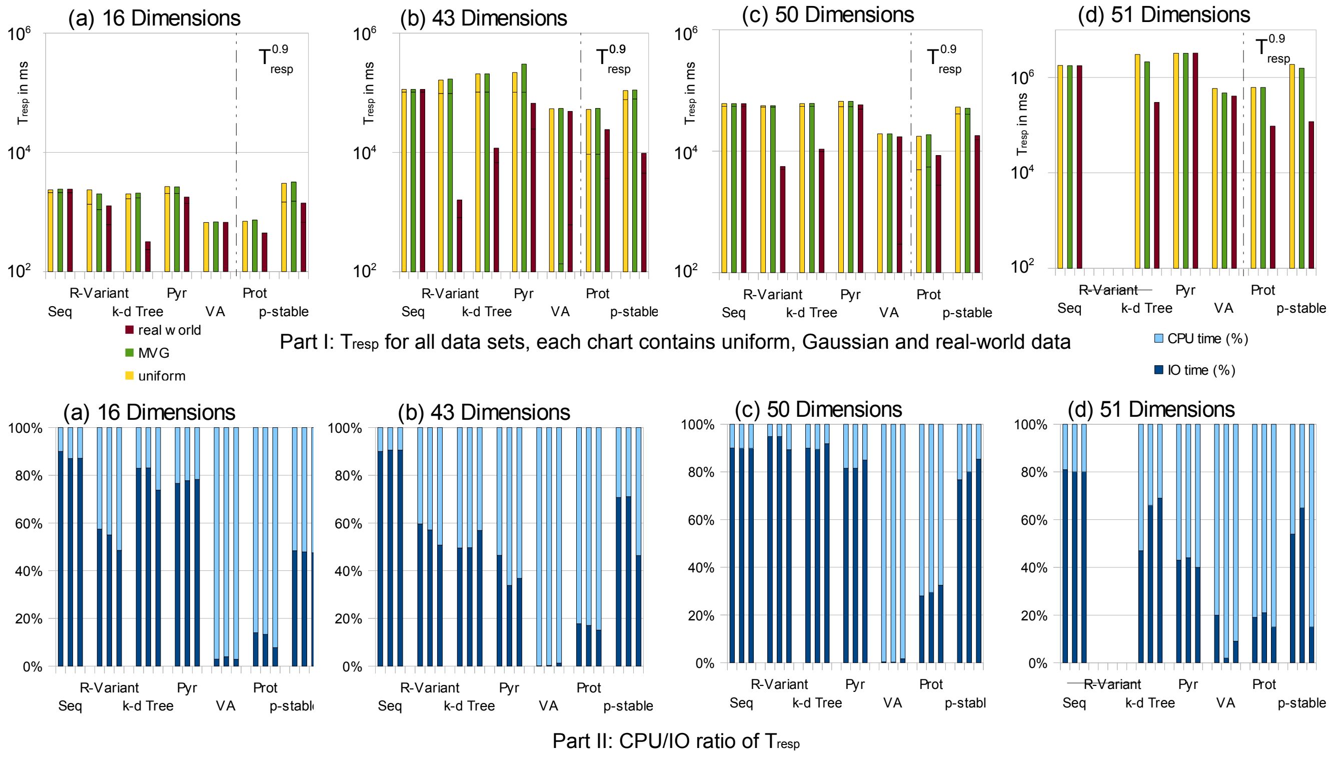


Figure 8: Part I: knn T_{resp} in ms (logarithmic scale), Part II CPU/IO ratio.

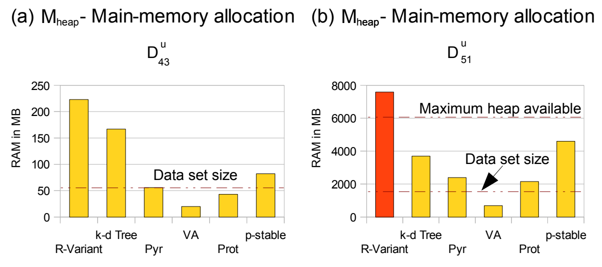


Figure 9: Exemplary RAM allocation for uniform data.

is between 2 and 4 times faster than a sequential scan and is the only index that efficiently reduces the number of HDD accesses (less than one percent of the points are accessed). It also requires least main memory. For instance, for data set D^u_{43} with a total size of 65MB, VA-File requires 20MB compared to 200MB of an R-Tree variant. Finally, for the Pyramid Technique there is only one scenario (Figure 8(b), real-world data) where we observe a clearly faster response time than those of a sequential scan (Pyr: $66s$, Seq: $189s$).

Interpretation and Consequences. First, even for a non-uniform distributed data set, high-dimensional data points are mostly located near to at least one of the borders of the space. As a result, the effect of stochastic distributions diminishes. Furthermore, the average distance of all points is similar and thus, nearest neighbors are spread across the whole space, which means that a large number of points have to be visited to find the k nearest neighbors. Summarily, for some cases using a sequential scan is sufficient since most indexes degenerate to a sequential scan using large amount of main memory and require additional CPU time.

For the second observation, we need to explain why tree-based indexes do not outperform a sequential scan for MVG or uniform data in our test cases. Apparently, N_{hdd} reveals that the R-Tree variant as well as the k-d Tree are in fact similar to a sequential scan since most points are accessed to retrieve the nearest neighbors. For example, data set D^u_{43} consists of 411,961 points. Particularly, the R-Tree variant accesses 388,691 points on average, which means in fact the whole data set. For the R-Tree variant we observed low overlap of leaf nodes. However, due to large distances of currently found nearest neighbors, leafs cannot be excluded even when there is no overlap. Similarly, k-d Tree contains several list-like degenerations. This property causes a performance decrease for exact-match queries, but allows the knn algorithm to exclude large parts of the tree.

Third, as known from literature, for high-dimensional data spaces most indexes degenerate to sequential scans [31]. Thus, accelerating sequential scanning results in a constant benefit that requires only low amounts of main memory. Furthermore, stochastic distributions have only a minor effect on the VA-File, because this structure considers approximation of all points. However, according to our results, the key factor that currently limits the performance of our implementation of VA-File are main memory computations and thus, processor speed is an optimization potential (e.g., parallelization). Finally, for the Pyramid Technique, as discussed in the intra-index evaluation, the major performance issue is that we can exclude only a small number of pyramids and their corresponding slices. Although we state that Pyramid Technique is up to three times faster for real-world data (because most of the points are *not located* in one of the corners of the space). Nevertheless, we measured faster response times of different indexes. Hence, we conclude that for all of our test cases, to support knn queries, we would use a different index structure for each of them.

Overall, we cannot give a clear recommendation which index to use, in contrast to exact-match queries. However, according to our results, applying a VA-File is always a good choice independent of data distribution or number of dimensions. Using VA-File is more beneficial if the HDD accesses are even more costly than in our test cases. Reasons which may prevent a developer from applying a VA-File are low CPU performance or little point look up times (e.g., SSD instead of HDD). Furthermore, our data reveal that using tree-based indexes for highly clustered or real-world data is a good choice as well. However, before applying for instance an R-Tree variant, analysts need to know how data are distributed, which is occasionally time consuming.

5.3.2 *Approximative Index Structures*

In contrast to exact indexes, we have to consider accuracy for approximative index structures as well. Hence, in Figure 8, we depict the fastest query response that delivers at least 90% of the correct nearest neighbors ($T^{0.9}_{\text{resp}}$).

Main Observations. First, for 11 out of 12 test cases, $T^{0.9}_{\text{resp}}$ of the Prototype Based Approach is faster than $T^{0.9}_{\text{resp}}$ of p-stable LSH (except for D^r_{43}) and for all test cases faster than a sequential scan. Second, both indexes have a faster response time for real-world than for MVG or uniform data. This is also observable in the amount of accessed points from HDD (N_{hdd}). In particular, knn-query processing with p-stable LSH for uniform or MVG data results in accessing a large percentage of points from the data set (e.g., 73% for D^u_{51}). In contrast, for real-world data the number of points fetched from HDD is significantly reduced (e.g., 0.74% for D^r_{51}).

Finally, we observed best $T^{0.9}_{\text{resp}}$ of our approximative index structures compared to T_{resp} of a sequential scan for the three data sets derived from the Physical Activity Monitoring use case (D_{51}). Especially for real-world data (D^r_{51}), our measurements reveal a speed up by factor 15 for Prototype Based Approach and for p-stable LSH. We can observe a similar, but not that strong tendency, for D_{50}. However, for any other scenario the approximative index structures are outperformed by exact ones.

Interpretation and Consequences. The first observation is due to parameter adjustment in the intra-index evaluation. For instance, we can use smaller bucket widths for p-stable LSH and have to consider less data for Prototype Based Approach. Furthermore, our analysis reveals that the LSH approach works best for real-world data with many dimensions containing nearly constant values. In our real-world data sets there are, for instance, dimensions dominated by outliers, that contain highly clustered data, or dimensions having only few values. All these properties of our real-world data result in relatively small distances of neighbors. This is the reason why p-stable LSH works best for 51 dimensions, because we can select small buckets and still have a high probability of containing the correct neighbors. Our analysis of the real-world data set confirms that, despite some multivariate skew and Gaussian-like distributions, data are dominated by outliers, highly clustered data, and few data values. As a result, distance of points is relatively small and thus, the probability of collisions is higher than for MVG or uniform data. This also explains the good $T^{0.9}_{\text{resp}}$ times for D^r_{51}. Due to small distances between the single points, collisions in hash buckets for p-stable LSH as well as similar permutations for the Prototype Based Approach are more likely than for uniform or MVG data. As a consequence, we recommend these index structures even for our high-dimensional use cases (e.g., D_{43}), in case that a small amount of false nearest neighbors is acceptable.

5.4 Inter-Index Evaluation: Building Times

In many scenarios, such as data warehousing, indexes are not updated, but rebuild to contain the most recent data. Thus, index building times may be an additional criterion to select an optimal index structure for a given use case. In particular long building times may furthermore exclude index structure despite of having a reasonable performance. To this end, we depict the index building times (T_{build}) in Figure 10.

We observed very similar building times for uniform, MVG, and real-world data, thus we only present the results for uniformly distributed data sets as representative to improve understandability. For every index, we use the optimized parameters determined in the intra index evaluation.

Main Observations. In particular, we observed larger differences between the single data sets than between the index structures of one data set. For instance, creating the VA-File for D^u_{50} required about 480ms while for D^u_{16} only 50ms are required. Furthermore, our results indicate that for one data set the difference between the fastest and the slowest index building time is about one magnitude. Finally, we observed that ranking the different building times for the different data sets results in very similar results. For instance, building the VA-File required, except for D^u_{51}, the least amount of time due to its easy computation of bit vectors. In turn, creating the Prototype Based Approach index resulted in the largest measurements, because for every insert the distance to all prototypes has to be computed.

Interpretation and Consequences. Summarily, although our results show only a maximum difference for index building times about one magnitude, the intra-index evaluation and additional test indicate that building times may be crucial. For instance, the anomaly in the VA-File building time for D^u_{51} reflects an implementation detail of our VA-File. To be independent of sorted data, we decided to compute the borders of single cells on all points of the data set, which is no problem for smaller data sets. However, for several million points computing these borders is costly, but minimizing the index building times was not our main target. Nevertheless, small modifications of the VA-File for large data sets result in similar query performance and a by two magnitudes smaller index building time. In the same sense, in the intra evaluation we used several split algorithms (e.g., Guttman's quadratic split algorithm) for our R-Tree variant that resulted in building times about a magnitude higher than selected variant. Even more costly were spherical variants, such as the SS-tree. To this end, QuEval evaluates and stores the index building times for further examination as well.

5.5 Overall Suggestions and Conclusions

Subsequently, we summarize the selection of indexes for our four use cases, to demonstrate a possible result of an evaluation performed with the help of QuEval. First, for the SPT test case, we recommend to apply the VA-File that clearly outperforms a sequential scan and minimizes the required main memory. This is an important property for our embedded devices (i.e., the sensors). Second, in the fingerprint example, we have no resource limitations. So, we can apply an R-Tree variant for that we observed far the best response times although it also requires most amount of main memory. Moreover, for the forensic use-case test case, that is dominated by exact-match queries, we suggest to use the Pyramid Technique, due to our results using Pyramid Technique results in fastest exact-match response times for all of our scenarios. Finally, for real-world high-dimensional data (e.g., D^r_{51}), we can recommend using approximative index structures, such as Prototype Based Approach.

On a more abstract level, software engineers shall use our suggestions as a starting point, which index structures are valid candidates to accelerate a certain use case. Therefore, they need to map their specific use case to the most similar one of our scenarios. More-

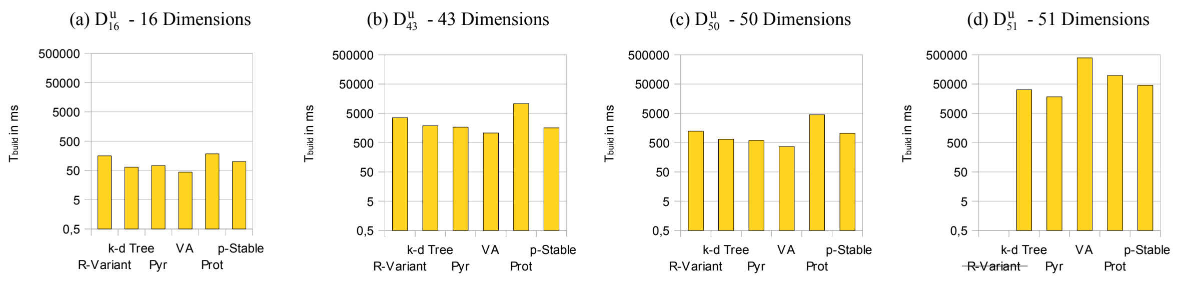


Figure 10: Index building time for uniform data.

over, the introduced evaluation procedure, consisting of intra and inter-index evaluation and QuEval's unified environment, supports software engineers to select one of the candidate index structures in a reasonable way. This is especially beneficial when designing new data-intensive solutions from scratch or optimizing existing ones. Finally, for non-database experts, the detailed description of parameters and influence factors, as well as the agglomeration of our experiences in our framework are useful to understand the complex nature of indexing high-dimensional spaces.

Furthermore, we argue that our evaluation procedure is valuable for database researchers as it simplifies argumentation for which scenarios newly introduced index structures are beneficial and what are imposed drawbacks when referring to our proposed evaluation procedure.

Summarily, we conclude that using our QuEval framework delivers valid and interesting results. Due to the open nature of our framework, we invite the community to:

- design and implement new index structures and benchmark them with QuEval,
- tailor existing approaches for new use cases, such as indexing of encrypted data, that can be integrated into our framework,
- provide new or improved implementations of existing index-structures.

We describe the details of contributing own implementations to QuEval on our website (`www.queval.de`).

5.6 Threats to Validity

In this section, we discuss threats to validity that may affect the results of our case study. To this end, we distinguish between internal (relation of input to output) and external (generalization of our results) validity.

5.6.1 Internal Validity

For the implementation of the selected index structures, we choose Java 7 as programming language, which may influence our results. First, we had to reimplement some index structures in Java. Second, with Java a fine-grained memory management, such as with C/C++ is not possible. This may influence optimization of our index structure implementation (regarding memory issues). However, since all index structures are implemented in Java, we have a comparable foundation for an objective comparison. Furthermore, for all index structures, we optimize the implementation in a same way and our C++ Handler allows for inclusion of C++ indexes as well.

Finally, Java-specific optimization could render our results meaningless. As a countermeasure, we use a warm-up period at the beginning of each test. Consequently, our presented results in previous sections do not consider this period and thus, the effects of Java-specific optimization can be omitted. We perform reproducibility and stability studies showing that we can reproduce stable results, which is conform to [17].

5.6.2 External Validity

Within our case study, we compare and evaluate seven indexes for high-dimensional data. Since this is just an excerpt of possible index structures that currently exist, our results may not be generalizable for other structures. However, we have chosen index structures with different characteristics to cover a wide variety of existing approaches. For instance, amongst the presented index structures, there are indexes using space partitioning as well as data partitioning methods. Furthermore, the considered index structures are based on different internal data structures such as trees, hash functions, or bitmap indexes. Hence, we argue that our case study, although not totally comprehensive, is generalizable to some extent. Finally, we argue that, because of the open nature of our framework, it is possible to easily integrate further indexes to conduct additional experiments, for instance with GPU-based implementations [11].

6. RELATED WORK

In recent years, many new index structures have been developed, mainly due to limitations of existing ones such as [8, 9, 13, 21, 27]. However, all of them evaluate their new or improved index structure against a relatively small set of other index structures. For example, Berchtold et al. evaluate the Pyramid Technique against the X-Tree, the Hilbert R-Tree variant, and the sequential scan [8]. The purpose of our framework is to ease handling this large amount index structures and helping to quantify the benefits of newly proposed index structures.

A different approach is to introduce index structures that can be used for various purposes. For instance, M-tree [14] and iDistance [24] are designed to speed up knn queries in arbitrary metric spaces. Additionally, GIST offers a generalization for tree-based indexes [22]. However, we argue, that due to specific requirements of a test case, generalized solutions often cannot provide the same benefits as specialized ones.

For evaluating different high-dimensional index structures, there already exist some frameworks, such as ELKI [1] and MESSIF [5]. However, these frameworks offer only a subset of QuEval's functionality (e.g., amount of index structures, distance metrics, and extensibility properties) or are tailored to specific domains. ELKI is mainly designed for cluster analysis optionally supported by index structures. MESSIF only focuses on metrical data structures and considers no in-memory storage of indexes.

7. CONCLUSION AND FUTURE WORK

Due to versatile characteristics of indexing high-dimensional data, choosing a suitable index structure for a specific use case is very difficult. Hence, we introduce QuEval, an extensible framework to evaluate and to support research in the domain of high-dimensional index structures. To show the benefits of our framework, we conduct an empirical evaluation based on four real-world examples (i.e., with optimized parameters). Within our evaluation, we perform an intra-index evaluation to guarantee an objective and unbiased comparison of index structures. Our results verify existing knowledge and suggestions for well-known index structures as well as point out parameter sensitivity of approximative indexes. Summarily, choosing the right parameters plays a pivotal role for the performance of some indexes and thus, should be a vital part of each evaluation. In summary, we conclude that applying QuEval for large empirical evaluation studies is beneficial and produces highly relevant results, relevant in practice as well as in research especially if data-intensive solutions need to be embedded in complex information systems. Consequently, we invite the community to extend our framework and benefit from its evaluation functionality.

In future work, we want to improve our method for evaluating newly introduced index structures to establish it as standardized procedure. Moreover, we want to collect and publish statistics for a multitude of indexes and large variety of test cases that allows a user to choose a suitable index for a given test case or at least limits the number of index structures for evaluation. We intend to extend our framework (e.g., storage structures). For instance, we want to add further index structures to the framework. Furthermore, we want to add specific workload containing for instance fixed ratios of different query types. Finally, we will evaluate the impact of new trends in databases, such as GPU co-processing or cloud computing, which is generally possible due to the open nature of our framework.

ACKNOWLEDGMENTS

This work has been funded in part by the BMBF No.: 13N10817 and DFG grant No.: SA 465/34-2. We also like to thank Martin Tobies for his helpful support.

8. REFERENCES

[1] E. Achtert, S. Goldhofer, H.-P. Kriegel, E. Schubert, and A. Zimek. Evaluation of clusterings – metrics and visual support. In *ICDE*, pages 1285–1288, 2012.

[2] F. Alimoglu and E. Alpaydin. Methods of combining multiple classifiers based on different representations for pen-based handwriting recognition. In *TAINN*, pages 637–640, 1996.

[3] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23:345–405, 1991.

[4] A. Azzalini and A. Dalla Valle. The multivariate skew-normal distribution. *Biometrika*, 83(4):715–726, 1996.

[5] M. Batko, D. Novak, and P. Zezula. Messif: Metric similarity search implementation framework. In *DELOS*, pages 1–10, 2007.

[6] 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.

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

[8] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyramid-Technique: Towards breaking the curse of dimensionality. *SIGMOD Rec.*, 27:142–153, 1998.

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

[10] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33:322–373, 2001.

[11] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Inf. Sys.*, 38(8):1084–1096, 2013.

[12] D. Broneske, M. Schäler, and A. Grebhahn. Extending an index-benchmarking framework with non-invasive visualization capability. In *BTW Workshop ISDE*, LNI, pages 151–160, 2013.

[13] G.-H. Cha, X. Zhu, P. Petkovic, and C.-W. Chung. An efficient indexing method for nearest neighbor searches in high-dimensional image databases. *IEEE Trans. on Multimedia*, 4(1):76–87, 2002.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[15] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.

[16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, 1997.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

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

[19] E. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(9):1647–1658, 2008.

[20] A. Grebhahn, D. Broneske, M. Schäler, R. Schröter, V. Köppen, and G. Saake. Challenges in finding an appropriate multi-dimensional index structure with respect to specific use cases. In *Foundations of Databases*, pages 77–82, 2012.

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

[22] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[23] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *SAC*, pages 604–613, 1998.

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

[25] T. Kiertscher, R. Fischer, and C. Vielhauer. Latent fingerprint detection using a spectral texture feature. In *MMSec*, pages 27–32, 2011.

[26] D.-H. Lee and H.-J. Kim. An efficient technique for nearest-neighbor query processing on the SPY-TEC. *IEEE Trans. Knowl. Data Eng.*, 15:1472–1486, 2003.

[27] K. Lin, H. Jagadish, and C. Faloutsos. The TV-Tree: An index structure for high-dimensional data. *VLDB Journal*, 3:517–542, 1994.

[28] A. Reiss and D. Stricker. Creating and benchmarking a new dataset for physical activity monitoring. In *PETRA*, pages 40:1–40:8, 2012.

[29] B. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu, and G. McGregor. Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nucl. Instrum. Meth.*, 543(2-3):577–584, 2005.

[30] M. Schäler, S. Schulze, R. Merkel, G. Saake, and J. Dittmann. Reliable provenance information for multimedia data using invertible fragile watermarks. In *BNCOD*, volume 7051 of *LNCS*, pages 3–17, 2011.

[31] R. Weber and S. Blott. An approximation-based data structure for similarity search. Technical Report ESPRIT project, no. 9141, ETH Zürich, 1997.

[32] 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.