Toward Variability Management to Tailor High Dimensional Index Implementations

Veit Köppen and Martin Schäler and Reimar Schröter
Faculty of Computer Science
Otto von Guericke University, Magdeburg, Germany

***Abstract*—The increasing amount of complex data requires a solution to store and query these data efficiently. One possibility to speed-up various query types is the application of high dimensional index structures. In prior work, we introduced *QuEval* as platform to evaluate these indexes for user-defined use cases. Our design allows to easily extend *QuEval* with new index structure implementations. However, based on our experiences, we encountered severe challenges by tailoring index structure implementations to specific use cases. In particular, we face challenges to manage several similar implementation variants of the same index. In this paper, we consequently show benefits and drawbacks that emphasize the necessity to tailor index structure implementations with the help of a short evaluation study. Finally, we outline approaches for adequate variability management to address the aforementioned drawbacks.**

***Keywords*—High-Dimensionality, Software Engineering, Index Structures, Databases.**

I. Introduction

Nowadays, we are confronted with an increasing amount of data. These data are often stored, changed, and processed in relational databases or data warehouses. Due to extreme data volumes and the characteristics of high dimensional data, an efficient support for data access is crucial. A well-known approach to query these data efficiently is using specialized index structures. As a result, a large variety of different index structures is proposed either as novel approach or improvement of an existing one (e.g., R-tree family [1]). However, every index structure addresses either a specialized domain or is evaluated to a selected set of existing alternatives. Thus, it is often a challenging task to identify which methods are suitable for a special use case. In high dimensional spaces, this is especially problematic due to different impact factors, such as cardinality of the data space, query types, dimensionality, and multiuser access [2]. In prior work, we proposed *QuEval* as benchmarking and evaluation tool for high dimensional index structures [3]. With our *QuEval* framework, we support fair comparisons of competitive high dimensional index structures. In order to achieve a sound evaluation, we allow to consider a large variety of influence factors, such as (1) different data distributions, (2) different query types, (3) easy extension with new index implementations, (4) index-specific parameter tuning, as well as (5) different optimization targets.

Extending *QuEval* with new index implementations can be done easily. So, we use our insights to tailor existing indexes to new use cases or enrich *QuEval* with new index structures. As a result, we quickly face severe challenges in managing various implementations of the same index. Tailoring an index to a specific use case means optimizing parameters and optimizing on source code level. In prior work [3], we focus on optimizing index-specific parameters. Consequently, this paper addresses optimizations on source-code level. This means to clone an already used index implementation and change details to try a new optimization. Such an optimization may only result in a benefit for a small amount of use cases while it can introduce severe drawbacks for the majority of remaining application scenarios. Thus, an optimization can be abandoned, combined with different optimizations, or refined – quickly creating *numerous variants* of the same index. The question arises: How to manage numerous implementations of the same index, each tailored to a special use case, e.g., to keep commonly shared code consistent without affecting the overall performance?

In this paper, we present results on currently ongoing work to answer the aforementioned question. To give basic background on *QuEval*, we first introduce architecture details and two index structures, each implemented in two ways. In an exemplary evaluation, we show that these simple optimizations result in a major performance difference. Moreover, this evaluation shows that experiences from sparsely populated spaces (used in [3]) cannot be generalized to densely populated spaces. This clearly motivates the need for efficient support of managing various implementations for several index structures. In Section IV, we discuss why intuitive solutions to manage variability are insufficient and usually result in a measurable performance penalty. Finally, we outline an approach to address the aforementioned challenges with the help of novel software product line development techniques and report on our first experiences using this approach.

II. High Dimensional Index Structures

In data warehousing, *Online Analytical Processing (OLAP)* is a method to analyze data in a high dimensional context. Due to performance issues, integration aspects, and availability, relational databases are widely used. Specialized databases, e.g., multimedia or forensic, have to enable queries in a high dimensional context, too. Many development and research efforts focus on efficient index structures for respecting multi-dimensionality (see for instance [4] or [5]). High dimensional data can be described in terms of data distribution. Additionally to [6], we include query types and data distribution aspects in our evaluation. In the following, we briefly present properties of selected index structures and their performance influence.

A. Evaluation Scenario – Data-Warehouse Case Study

In our evaluation example, we are interested in efficient data access for multi-dimensional data (approximate 7-12 dimensions, each with a cardinality of about 10 elements per dimension). We want to analyze data spaces that are covered with 0.1‰ up to 10-25%. Our goal is to answer the question which of the three presented index structures fits best to exact match queries. For simplicity, we do not consider data inserts and deletes in this paper, but build the index from the complete given data. Before we give details on concepts and index implementation, to show the versatility of our *QuEval* framework, we introduce assumptions on the queried data, which influence implementation details. As we use a data warehouse scenario, we assume:

1) Each dataset has a unique multi-dimensional key, an arbitrary payload (e.g., aggregated values etc.), and is assigned a unique tuple identifier (TID),
2) For an exact match, a multi-dimensional key is provided. The results are the TIDs of all points that have the same key. As in this case the key is unique, the query result is either *one* TID or *not found*.
3) We assume, there are no online updates. In case there are new data, the index is rebuilt. Thus, query response and index build times are of interest.

B. High Dimensional Index Structures

There exist many ideas for efficient data access that have to be evaluated against existing ones. Due to complexity and optimization criteria, benchmarking plays a vital role within the community. [6] suggests a framework for evaluating index structures in the OLAP domain with a focus on range queries. [7] discusses index structures and algorithms in the domain of multimedia databases. Therefore, the focus is set on similarity searches in high dimensional feature spaces. [8] addresses the challenge of comparing new algorithms and data structures within subspace clustering. Note, a holistic approach for high dimensional index structures is currently missing.

In practice, a data management system has to support users in its best way for a specific workload. Due to heterogeneity of data, queries, and user preferences, it is nearly infeasible to suggest an optimal index structure. With the help of the *QuEval* framework, we support the decision process in identifying an appropriate index structure for customizable workloads. This addresses query types, workloads, as well as algorithms and data structures as competitors. Additionally, used metrics, optimization criteria (e.g., performance), and limitations are included in the benchmarking properties of *QuEval*.

For the remainder of this paper, we focus on a specific evaluation task as an example. We also show implementation issues regarding the Dwarf [9] and the Pyramid technique [10] as examples within our framework.

C. Dwarf

The main intention of the Dwarf [9] is the compressed data storage of high dimensional OLAP data. It uses prefix and suffix redundancies, which are both used for an improved exploitation of dense and sparse data areas respectively. Originally, the Dwarf structure represents the OLAP fact table in a digraph. We change the Dwarf according to our application scenario. As we assume that the result of an exact-match query is a TID, this generalization does not change the Dwarf structure for our evaluation. In Fig. 1, we use three dimensions as an illustrative example, D_1, D_2, and D_3. For each dimension, lists

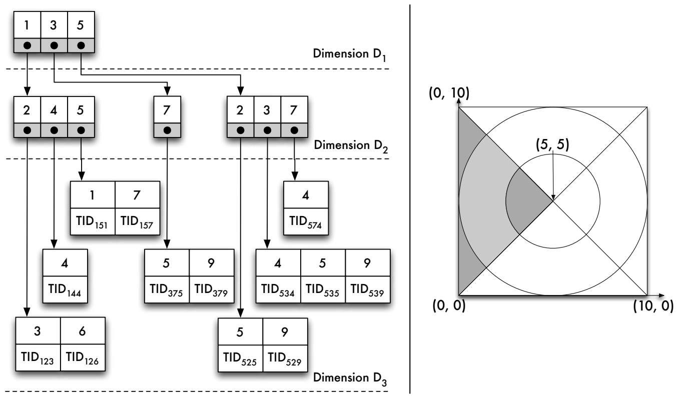

Fig. 1. Dwarf with prefix redundancy utilization (left) and Pyramid technique using spherical segments (right)

are used that consist of a representation of the dimension value and a pointer to the next dimension level. This means, values from D_1 point to D_2 and these point to the last level (D_3). In level D_3, the pointer directs to the tuple identifier. Note, the Dwarf is suggested as a storage method for compressing OLAP data cubes. Instead, we use this method as an index structure for accessing high dimensional data efficiently due to reduced processing cost.

1) Dwarf Implementation in QuEval*:* An index structure within the *QuEval* framework has to implement the abstract class *AbstractIndex*. For our demonstration two methods have to be implemented: *buildIndex* and *search*. The first method is used to create the complete index using all available data points. To show that tailoring an index on the implementation level to the respective use case, we utilize two implementations. The first one implements the Dwarf using classes. For the first dimension there is a class that contains a list of *DwarfElements*. A *DwarfElement* contains the key (value) in the current dimension and a pointer to the list in the next dimension associated to this key (cf. Fig. 1 left). In the second dimension, there are as many lists as *DwarfElement*s in the first dimension etc. In the final list (dimension), we store the keys and the TIDs, which have to be returned to answer exact-match queries. Using this implementation results into properties that are important for the subsequent evaluation. First, using lists and classes causes that the index is scattered over the heap and the amount of required memory is quite high. Thus, efficient caching is hardly possible, but we can insert additional points at any time without rebuilding the index.

2) Dwarf Implementation – Optimizations: The second implementation takes advantage of the above mentioned assumptions of no online updates. It uses an explicit memory design that linearizes the Dwarf in a consecutive block of memory to require the minimum amount of memory and improve caching behavior. This benefit comes of the cost that inserting new data means rebuilding the whole index. Moreover, we do not use classes to avoid runtime overhead. To compute the linearization, we initially build the first implementation and then linearize it.

D. Pyramid Technique

According to [10], the Pyramid technique is considered to be not effected by the curse of dimensionality. The basic idea of this technique is to divide a d-dimensional space into $2d$ pyramids. By definition, a normalized d-dimensional data point x is located in the pyramid p_i with the following condition:

$$i = \begin{cases} j_{max} & \text{if } x_{j_{max}} < 5 \\ (j_{max} + d) & \text{if } x_{j_{max}} \geq 5 \end{cases}$$

$$j_{max} = (j \mid \forall k, 0 \leq k < d \texttt{ and } j \neq k : |5 - x_j| \geq |5 - x_k|)\,.$$

As a result, a point is inserted according to its dimension that has the largest distance j_{max} to the center of the space $(5, 5)$. For instance, the point $(8, 3)$ results in $j_{max} = 0$ because $|5 - 8| \geq |5 - 3|$ and finally, i is equal to $0 + d = 2$ due to $8 > 5$ (i.e., on the right side of the space center). For efficiently managing the points, the pyramid is divided into several slices. Different ways for partitioning a pyramid are proposed to support different query types. Whereas [10] supports range queries by splitting the pyramid horizontally according to their basis, [11] divides the pyramid in a spherical way, as depicted in Fig. 1 right. Note, [10] uses hyper rectangles and [11] uses hyper spheres. For our evaluation example the spherical approach is more suitable and therefore, we consider this implementation in the following.

1) Pyramid Implementation in QuEval*:* Similar to the Dwarf, we evaluate two different Pyramid implementations. So, we show that variability on code level is required to adapt to properties of a use case. We first implement this index using classic object-oriented decomposition. That means, there is one class implementing the index itself. This index class contains pointers to classes, each representing a *Pyramid*. Additionally, each *Pyramid* contains pointers to classes representing a *Slice* within this pyramid. Finally, each *Slice* has a *HashMap* utilizing the distance of a point to the dimension center as key pointing to a *List* of TIDs of points sharing the same key. Consequently, executing an exact-match query means first identify the pyramid, second get the right slice, and search the slice finally. Note, pyramids as well as slices only exist (are not *null*) in case they contain at least one point to save main memory. In prior work [3], we reveal that this property also improves exact-match query response times for sparsely populated high dimensional spaces. This is because, we stop computation in case a pyramid or slice does not exist. To sum up, this implementation is similar to our first Dwarf implementation.

2) Pyramid Implementation – Optimizations: Taking the properties of the use case into consideration, we can optimize the index on source code level. In fact, our second implementation is based on the same ideas as the second Dwarf implementation. In particular, we observe that in our data spaces, all pyramids and slices exist. Thus, we can create a directory in a consecutive block of main memory containing all pointers to the *HashMaps* storing the TIDs. Moreover, we abandoned the object-oriented decomposition to avoid overhead, because we only need the offset in the directory. Moreover, we can remove if-statements to determine whether the pyramid and slice exist, which avoids flushing the processor pipeline, due to possible branch mispredictions. In contrast to our first implementation, we do not use *Lists*, but *Arrays* to store TIDs in the *HashMaps* by initially building the first variant of the Pyramid and then linearize it. This way, we optimize the index to bulk insert property resulting in better cashing behavior.

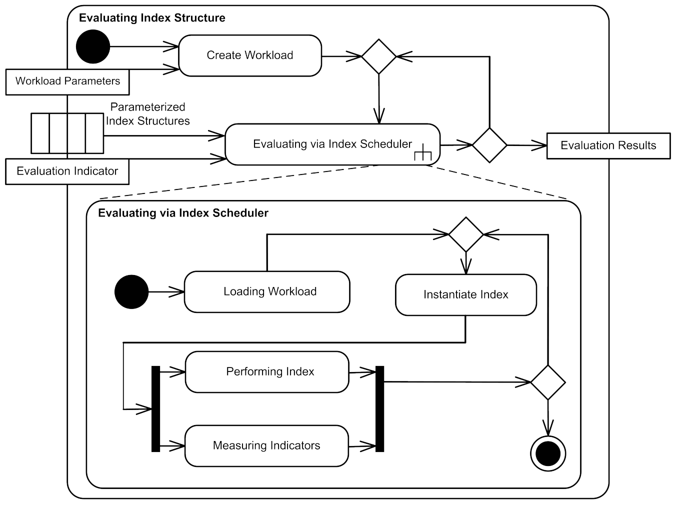

Fig. 2. Evaluation Activities in *QuEval*

III. Evaluation of Index Structures

Every new development can only focus on a restricted or partial aspect. An overall comparison often is not applicable. A benchmark for high dimensional index structures is either not available or very specialized. Therefore, the *QuEval* approach [3] tries to evaluate index structures in different ways. As derived in the last section, different aspects have to be considered for the evaluation of multi dimensional index structures. An experiment is defined with a given set of parameters, e.g., dimensionality of the dataset, query types within the workload, and evaluation indicators.

In Fig. 2, we present the *QuEval* evaluation process. In a first step, we create the workload for the evaluation. The workload parameters define either an existing dataset or dimensionality, number of tuples, and data types. With this information a dataset is ready for use. Furthermore, the workload parameters are used to define queries, e.g., exact match queries or knn search points. Note, the search points can be in or not in the dataset. In the next step, a set of index structures is evaluated. In this context, an index structure is defined including the corresponding parameters. This differentiation enables us to identify an optimal parameter tuning for a given index structure.

Within the *QuEval* framework, we support different programming languages for index implementation. This is achieved with a scheduler as represented by a call behavior action in Fig. 2. As input parameters we need the set of parameterized index structures and the evaluation indicators. The scheduler loads the created workload from the first step and instantiates the index structure. In the next step, the workload is executed by the index structure whereas at the same time the measuring of the required evaluation indicators is performed. In the case there is another index that can be executed by the scheduler the next instantiation is executed, otherwise the scheduler is stopped. If another scheduler (programming language implementation) is part of the evaluation, a new iteration of the *Evaluation via Index Scheduler* is executed. After all executions, the measurements are reported as evaluation results.

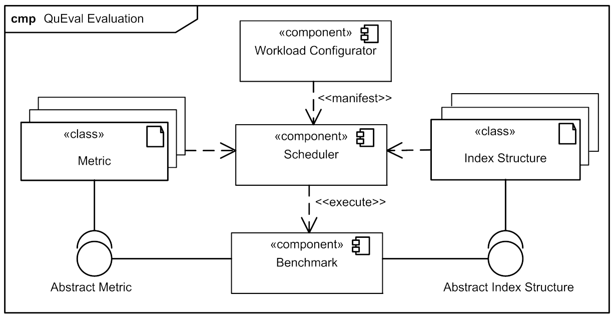

Fig. 3. Components within *QuEval*

In Fig. 3, we present a set of required components of the *QuEval* framework. On the one side, index structures can be implemented. They have to implement methods and properties of the interface *AbstractIndexStructure*. On the other side, new metrics can be added. These implementations should be derived from the *AbstractMetric* interface. The component *WorkloadConfigurator* is responsible for the workload, which is generated by the *Scheduler*. An XML file is used for the description of the workload, which includes information on data and corresponding queries. The scheduler is the central component within our framework. It loads and initializes index structures and metrics. Note, the scheduler is dependent on the programming language implementations. Therefore, we have for each supported programming language an implementation. The scheduler component is responsible for processing the benchmark and controlling workloads, index structures, and metrics. Within the *Benchmark* component, the workload is executed on indexes with respect to metrics. Furthermore, the benchmark evaluation parameters are measured and recorded.

A. Evaluation Indicators

Efficiency can be estimated in terms of resources. This includes memory consumption, CPU usage, or HDD accesses. In our application domain, response time of a query as well as memory consumption play a vital role. Due to different phases and actions respectively, we divide the indicator measurement into build and run-time dependent. We define build time as the creation of an index structure given the dataset. Run-time indicators include all actions that are required to execute a query. For a given exact match query, we measure response time including CPU time and all necessary HDD accesses. The most prominent evaluation indicators are: (1) memory consumption at build time and run-time, (2) index build time, (3) query response time (e.g., exact matches), and (4) if applicable query precision (e.g., for knn queries).

For a fair comparison, an index tuning for each index structure has to be applied on the benchmark dataset. In our exemplary evaluation, we present these results, whereas we use for the Dwarf two implementations as well as for the Pyramid. We also present the k-d tree as a competitor to visualize an established tree index structure.

B. On the Influence of Implementation Optimization

In this section, we briefly present an evaluation with respect to index structure implementation. We use two different implementations for the Dwarf and the Pyramid technique. As a target of the application scenario, we want to optimize exact queries, within a multi-dimensional space. For our evaluated datasets, we use for each dimension a cardinality of 10. An important question, we want to answer in this study, is the influence of the population space. Our data space is between 7 and 10 dimensions that means an overall population from 10^7 to 10^{10} elements. Our benchmark data is randomly generated within this space with one to nine million data entries. We estimate the response time of 1,000,000 exact match queries for the following index implementations: (1) Dwarf, (2) Dwarf with linearization, (3) k-d tree, (4) Pyramid technique, and (5) Pyramid technique with linearization.

Note, the building of the corresponding structure is not in the focus of this evaluation. However, the build times are for both implementations with linearization higher than their established index implementations. This is due to the fact that the linearization is done on the existing data structure, which has to be built first. Regarding the build times, the Pyramid outperforms all others with about 100 ms per one million data entries. The Dwarf requires slightly more time than the construction of the k-d tree, with an increase of the population space. This increase depends on the search for the insertion points in the index structure.

In Fig. 4, we present response times for one million exact match queries for all evaluated index structures. Note, the response time is depicted in a log-scale representation. We only show an excerpt of our evaluations for the population space from 2 to 8‰. For all other degrees of the population space a similar behavior is at hand. As it can be easily seen, both Pyramid techniques require much more time to answer the queries. The reason is that the cardinality of each dimension is quite low which results in full buckets. However, an improvement of the performance is achieved due to linearization of the underlying pyramid data structure at a factor of about two. The Dwarf is quite stable in the response times whereas the response times of the k-d tree increase with an increasing population space. For higher filled spaces, the Dwarf in the standard implementation also outperforms the k-d tree. Furthermore it can again be easily seen that the optimization of the implementation of the Dwarf has a significant influence on the performance measure.

In Fig. 5, we present a comparison of both linearized implementation versions regarding dimensions and the number of tuples within the dataset. The increase of the response times for the Pyramid is higher than in the Dwarf which is a result from the overfull buckets due to the densely populated space. Note that the Dwarf is about factor 40 faster for answering the exact match queries for low populated spaces, which also increases up to 200 for 5% populated space at 7 dimensions. The increase is even higher with an increasing number of dimensions. This is a significant result for not sparsely populated high dimensional data spaces. Therefore, we conclude to use the Dwarf structure as an index for these applications.

As main contribution of the evaluation, we sum up that the optimization of implementations plays an important role for the performance of high dimensional index structures. An optimal data structure can be constructed regarding the application scenario. This means that besides the data distribution, query types, performance indicators, and so on also implementation

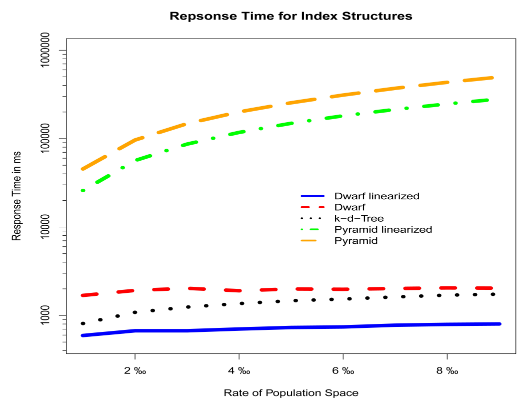

Fig. 4. Response times for Dwarf, linearized Version of the Dwarf, k-d tree, Pyramid technique, and the linearized Version of the Pyramid technique

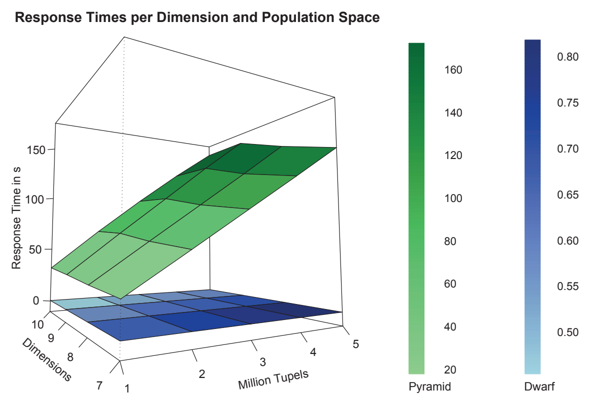

Fig. 5. Response Time of linearized Dwarf and Pyramid regarding number of Dimensions and Data Tuples

details influence the response time significantly. Consequently, we have to tailor index implementations on source code level, resulting in a huge number of variants that have to be managed. In the following, we depict challenges with respect to variability management.

IV. Toward managing implementation variants

In this section, we motivate the need for tailoring index-structure implementations recapitulating the results of our prior evaluation and generalize the gained insights. Based on these explanations, we then explain why intuitive solutions to manage this variability are insufficient. Consequently, we finally outline how to use novel programming techniques and discuss their expected benefits as well as we report on first results.

A. On the Necessity for Tailored Implementations

As a consequence of the previous evaluation, we consider tailoring index-structure implementations to a use case as one of the most important aspects to achieve optimal results. In Section II, we explain how to modify index implementation to characteristics of our use case. Using this background knowledge, it is possible to optimize index structure implementations resulting in a large performance benefit. However, these optimizations are solely possible for our use case and thus, are not applicable for many other use cases (e.g., privacy considerations [12]). As a result, we need a new tailored variant on source code level of the existing index structure and not a simple one-fits-it-all adaption of the existing one. Therefore, the usage of parameters for conditional execution (e.g., if-statements) of program code to implement all variants in one implementation unit does not represent an appropriate solution. Initial results indicate that this procedure imposes severe performance penalties. Consequently, alternative solutions strategies are necessary to address the existing challenges that we illustrate and compare in the remainder of this section.

B. Why Intuitive Solutions are Insufficient

Intuitively, several strategies are known to create similar variants of a program, e.g., index structures. In the following, we explain limitations of intuitive solutions motivating the need for more advanced ones.

Object-Oriented Inheritance: Analogously to parameterization, using inheritance to provide source code variability negatively affects the overall performance due to the object-oriented decomposition. For instance, we used inheritance for the implementation of the Dwarf *DimLists* so that the lists of the last dimension contain the TID and do not contain the pointer to the list in the next dimension, which is the default decomposition approach in the object-oriented paradigm. However, the general problem is that usually multiple inheritance hierarchies have to be resolved in order to get the actual class. This results in significant performance penalties. Consequently, we abandoned this decomposition within parts of the indexes that are relevant for query processing.

Code Cloning: To avoid the drawbacks of the aforementioned approaches, we use a code-clone strategy. To this end, we clone an initial implementation and tailor new variants to our requirements and then start this procedure again to try additional or alternative optimizations. Although this is easily applicable, we quickly realize severe drawbacks of this approach. The basic problem is that there are multiple implementations largely sharing the same (or similar) code base [13]. For instance, if we detect an implementation error in one of the variants, we have to search for similar errors in *all* other variants. Note, this is a major challenge because of different implementation structures.

C. Our Solution: Novel Programming Paradigms

Our experiences in optimizing index-structure implementations reveal that conventional implementation approaches are insufficient for our purposes as discussed before. To this end, we propose to apply novel programming techniques used in software product lines. Software product lines are a well-known approach to handle multiple variants of a program on conceptual and implementation level [14]. All variants share a common code base and thus, this approach has been shown to simplify the effort of keeping large code bases consistent. These techniques are manifold and contain language extensions, such as aspect-oriented programing (AOP) and feature-oriented programming (FOP) [15], or use tool-based approaches, such as advanced pre-processor approaches [16]. Each technique having its own benefits and drawbacks exemplary regarding introduced performance overhead.

Currently, most approaches focus on one of these programming techniques as they have only one primary objective, which is to support variability management. However, our experiences reveal – we require for different solutions due to:

- the granularity of the intended changes,
- its call frequency,[1]
- and the amount of code redundancies (clones).

For instance, right now we hardly see alternatives to preprocessor techniques avoiding unnecessary if-statements in hot loops of index implementations. However, this does not solve problems regarding code clones. We are currently experimenting with applying templates and macros and first results are promising. By contrast, for changes that affect large parts of the infrastructure (including *QuEval*), we need different implementation techniques such as AOP or even using components in a plug-in infrastructure.

Multi Software Product Lines: In addition to the aforementioned challenges, allowing a large variety of evaluations requires a rather complex infrastructure. For instance, our *QuEval* framework consists of a *Scheduler*, *Benchmark Generator* creating *Benchmarks*, *Indexes*, and *Metrics*. Simple changes on conceptual level impose major changes to the underlying implementations. Consider for instance a data-type change from integer to float or even to string types. Although many algorithms and the general concept remain the same, the underlying implementation varies and has to be consistent throughout the whole infrastructure. As a result, all parts of our infrastructure (not only the indexes) have to be variable. In the same sense, *QuEval* provides an abstract class that must be extended by each index structure to guarantee the compatibility and the access from the framework. If we change the abstract class because of a new application scenario (e.g., a new query type), each index structure has to be changed as well to realize the compatibility. This also affects the whole infrastructure. Consequently, we use a set of interdependent software product lines. We refer to this as a *multi software product line* [17], which imposes new interesting challenges, such as variability within APIs and ensuring their type safety.

V. Conclusions

Identification of an optimal high dimensional index structure is itself a multi dimensional challenge. Besides the comparison of different index structures, parameter optimization and implementation details have to be considered. In prior work, we focus on parameter optimization [3]. In this paper, we emphasize the importance of implementation details. In particular, we address the challenge of variability management of different index structure implementations. An intuitive approach, such as cloning, is not feasible for a large number of index variants. To this end, we consider software product lines that contain code artifacts to tailor each index structure to a use case. We consider single modules of QuEval as product lines themselves. This imposes the consistency challenge among the modules, being an important issue of current multi software product line research.

For a more general point of view, the proposed solution to handle variability on source code level contributes is consistent to as well as it contributes to the current trend of optimizing data processing to specialized hardware and use cases seen as one of the major future challenges [18], [19]. To this end, we argue that solving this problem, and required future research on this topic, results in a significant contribution to address future challenges in data processing (e.g., Big Data). Nevertheless, our ideas are in contrast to past and recent approaches to provide hardware oblivious solutions. However, we argue the basic challenge is not knowing the hardware but efficiently managing the large amount of tailored variants for this hardware and different use cases.

[1]The more often a code is called and the shorter its run time, the more severe is the performance penalty introduced by the programming technique.

Acknowledgments

This work has been partly funded by the German BMBF under Contract No. 13N10818 and by the DFG SA 465/34-2.

References

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," SIGMOD Rec., vol. 14, no. 2, pp. 47–57, 1984.

[2] 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 GvD. GI, 2012, pp. 77–82.

[3] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake, "QuEval: Beyond high-dimensional indexing à la carte," PVLDB, vol. 6, no. 14, pp. 1654–1665, 2013.

[4] V. Gaede and O. Günther, "Multidimensional access methods," ACM Comput. Surv., vol. 30, pp. 170–231, 1998.

[5] H. Samet, Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.

[6] M. Jürgens and H.-J. Lenz, "Tree based indexes vs. bitmap indexes - a performance study," in DMDW, 1999, pp. 1.1 – 1.10.

[7] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," ACM Comput. Surv., vol. 33, no. 3, pp. 322–373, 2001.

[8] E. Achtert, H.-P. Kriegel, and A. Zimek, "ELKI: A software system for evaluation of subspace clustering algorithms," in SSDBM, ser. LNCS (5069). Springer, 2008, pp. 580–585.

[9] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, "Dwarf: Shrinking the petacube," in SIGMOD. ACM, 2002, pp. 464–475.

[10] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The Pyramid-technique: Towards breaking the curse of dimensionality," SIGMOD Rec., vol. 27, no. 2, pp. 142–153, 1998.

[11] D.-H. Lee and H.-J. Kim, "An efficient technique for nearest-neighbor query processing on the SPY-TEC," TKDE, vol. 15, no. 6, pp. 1472–1486, 2003.

[12] A. Grebhahn, M. Schäler, V. Köppen, and G. Saake, "Privacy-aware multidimensional indexing," in BTW, 2013, pp. 133–147.

[13] S. Schulze, E. Jürgens, and J. Feigenspan, "Analyzing the effect of preprocessor annotations on code clones," in SCAM. IEEE, 2011, pp. 115–124.

[14] K. Czarnecki and U. Eisenecker, Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley, 2000.

[15] S. Apel, D. Batory, C. Kästner, and G. Saake, Feature-Oriented Software Product Lines. Springer, 2013.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in ICSE. ACM, 2008, pp. 311–320.

[17] R. Schröter, N. Siegmund, and T. Thüm, "Towards modular analysis of multi product lines," in SPLC-Workshops. ACM, 2013, pp. 96–99.

[18] D. Broneske, S. Breß, M. Heimel, and G. Saake, "Toward hardware-sensitive database operations." in EDBT, 2014, pp. 229–234.

[19] B. Răducanu, P. Boncz, and M. Zukowski, "Micro adaptivity in vector-wise," in SIGMOD. ACM, 2013, pp. 1231–1242.