International Journal Of Advancement In Engineering Technology, Management and Applied Science (IJAETMAS)
ISSN: 2349-3224 || www.ijaetmas.com || Volume 03 - Issue 12 || December - 2016 || PP. 96-105

Elf: A Main-Memory Index for Efficient Multi- Dimensional Range and Partial Match Queries

Veit Köppen[1], David Broneske[1], Martin Schäler[2], Gunter Saake[1]

[1]*University of Magdeburg, Germany*
[2]*Karlruhe Institute of Technology, Germany*

***Abstract*— Efficient evaluation of selection predicates (e.g., range predicates) defined on multiple columns of the same table is a difficult, but nevertheless important task. As we have seen an enormous increase of data within the last decade, efficient multi-dimensional selection predicate evaluation becomes more important. This is especially important for scientific data management tasks, where we often face data sets that need to be filtered based on several dimensions. So far, the state-of-the-art solution strategy is to apply highly optimized sequential scans. However, the intermediate results are often large, while the final query result often only contains a small fraction of the data set. This is due to the combined selectivity of all predicates. We propose Elf - a new tree-based approach to efficiently support such queries. Our structure indexes densely populated sub-spaces allowing for efficient pruning.**

Keywords*— *data analytics, indexing, main-memory databases, storage structures.

I. MOTIVATION

Efficient evaluation of range predicates on large database tables is ever since an important task. With the rise of main-memory systems for analytical and scientific processing, more and more multi-dimensional data sets now fit into main memory that need fast methods to evaluate range predicates efficiently. However, efficiency in the field of main-memory database systems does not only mean that we have to care about accesses to main-memory, but also need to take other factors, such as CPU utilization and cache misses, into consideration [Br14, LKN13].

Classical indexing schemes already allow for evaluating multi-dimensional range queries. For instance, the R-tree family allows for easy range query evaluation, due to their concept of minimum bounding rectangles [Gu84]. In a similar way, kd-trees naturally support range-query evaluation by their concept of partitioning the space using axis-parallel hyperplanes [Be75]. However, so far all tree-based approaches suffer from the curse of dimensionality [BBK98, GBS13]. This means in particular, that for a certain number of dimensions (16 according to [BBK98]), simple accelerated scans have a better performance than these approaches [BBS14].

There has been a large discussion whether tree-based indexes can somehow be modified to weaken the effects of the curse. However, as improvements only increase the threshold of the number of dimensions, where we observe the performance turning point between index and sequential scan, the overall problem is still unsolved. We argue that this is the reason why recent research mainly focuses on highly optimized sequential scans. Nevertheless, when taking a closer look at all indexing approaches, we make an interesting observation.

Independent of the specific concept they are designed on (e.g., bounding rectangles, separation of the space by hyperplanes), they are in that way similar, that they enclose a certain sub space with a certain geometric form. This, however, means for multi-dimensional spaces that most of the space within these forms is empty, due to the sparse population of multi-dimensional spaces. This negatively affects various query types including k-nearest neighbor queries and range queries. We argue that this is the inherent problem of all these approaches. Hence, we need to come up with a totally different concept that does not index empty space, but still allows for an efficient evaluation of selection predicates.

In this paper, we contribute a novel approach for efficient evaluation for multi-column selection predicate evaluation, which we name Elf. Moreover, we empirically show the superiority of our novel approach in comparison to a state-of-the-art sequential scan and the kd-tree [Be75] for two use cases. In particular:

1. We define design principals for our novel approach based on limitations of currently used approaches that in fact are the primary reason for the performance loss of tree-based approaches in multi-dimensional spaces.
2. Based on our design principals, we conceptually design the Elf.
3. We evaluate the space partitioning of our Elf and discuss its superiority to other index structure's partitioning.
4. We conduct an experimental evaluation of our approach indicating the superiority of our approach compared to a sequential scan for exact-match, range and partial match queries.

The remainder of this paper is structured as follows. Based on limitations of well-known tree-based indexes, we define our new approach in Section 3. The empirical evaluation is presented in Section 4 and we conclude our work in Section 5.

II. LIMITATIONS OF WELL-KNOWN TREE-BASED INDEXES

It is commonly known that all tree-based multi-dimensional index structures are affected by the curse of dimensionality. Although best researched for k-nearest neighbors queries, it is also relevant for range or partial match queries. In consequence, this means that their performance in high-dimensional space is often worse than that of a simple sequential scan. As an example, Berchtold et al. state that 16 dimensions are the crucial turning point [BKK96]. The reasons therefore are mainly:

- the pure size of the multi-dimensional space which means that the whole space is sparsely populated, because there is not enough data to fill the whole space and
- ·the tendency of data points to be clustered in one of the corners of the resulting data space

However, if we take a closer look at the most well-known tree-based indexing approaches, namely R-tree [Gu84] and kd-tree [Be75], we make an interesting observation. The problem is not the curse of dimensionality itself but non-trivial design limitations are in fact the real problem. For the R-tree family (and several spherical improvements) the basic idea is using minimum bounding geometric forms (rectangles, e.g., [An10, Be90, JL98, SRF87], spheres, e.g., [vO90, WJ96], or mixtures, e.g., [KS97, KJS97]) to describe an area that totally encloses a sub-tree or the points in a leaf node. This design features two fundamental limitations. First, all nodes cover nearly the same number of entries. As a result, the deeper we descend in the tree, the worse is the ratio between indexed space and number of points covered in the respective leafs of the considered sub-tree. A related effect of this issue is that we often have large overlaps of sub-trees, which does not occur in low dimensional spaces [Be90]. Second, by concept,

bounding boxes also index space where there is no data. Finally, storing such bounding geometric forms requires a lot of storage space, e.g., we need to store two n-dimensional points and the pointers to the sub-trees for one MBRs. Due to these limitations, we observe the problem of overlapping nodes besides others.

We observe similar issues when looking at kd-trees. Based on these observations, we define two design principles that our Elf should feature:

1. No indexing of empty parts of the data space, which is conducted for instance by space-partitioning methods like kd-trees, but also by bounding boxes like MBRs.
2. The size of indexed space and the number of indexed points per tree level should decrease in the same order of magnitude.

To the best of our knowledge, there have been attempts solve some of these issues for disk-based indexing approaches resulting, for instance, in the multi-dimensional B-Tree [LBJ95,SC85, GK80]. However, these indexes are designed for disk-based systems and OLTP workloads, i.e. they also allow for efficient updates and deletes. Consequently, their nodes are generally half empty and the fixed maximal node size usually requires to fetch multiple nodes from disk, which is unsuitable in main-memory environments. Hence, we require for a new approach for main-memory systems. Such an approach avoiding the above mentioned issues would be the first that does not suffer from the curse of dimensionality. In particular, we could even benefit from this curse, as we take advantage of the sparse population of high-dimensional data spaces. As we are not aware of an easy concept that allows for incorporating these principles in R-tree-like or kd-tree-like structures, we need to look for a different concept. This concept is described in the following.

III. ELF STORAGE STRUCTURE

We present our new index structure for efficient multi-dimensional querying in this section. First, we discuss prefix-redundancy elimination and its benefits for efficient storage and querying of multi-dimensional data. Second, we present design concepts to incorporate prefix-redundancy elimination in our new index structure.

Table 1 Running example data

D_1	D_2	D_3	D_4	Ref
0	1	0	1	T_1
0	2	0	0	T_2
1	0	1	0	T_3

Prefix-redundancy elimination

A concept that allows us to compress the amount of data to be stored and queried is prefix-redundancy elimination. It is first proposed as part of the dwarf data structure, which materializes a cube operator [Si02]. Although, we are focusing on multi-dimensional data and not the whole cube, prefix-redundancy elimination still contributes a good benefit for our use case. For a given n-dimensional data set D n and some order of dimensions, we observe a prefix redundancy (for at least two points) if the following holds. There is a k defining an interval [1, k] over the range of dimensions, with $k \leq n$. For such a k, we find at least two points P1 and P2 in the data set, with $P_1 \neq P_2$ having the same values for all dimensions in the interval: $\forall d \in [1,k]\ P_1[d] = P2[d]$. Considering the running example table with four columns and a tuple identifier in Tab. 1, we observe a prefix redundancy of T_1 and T_2 for k = 1, because both points have the same value in the first dimension T_1 and, thus, $T_1[1] = T_2[1]$ holds for the whole interval [1,1].

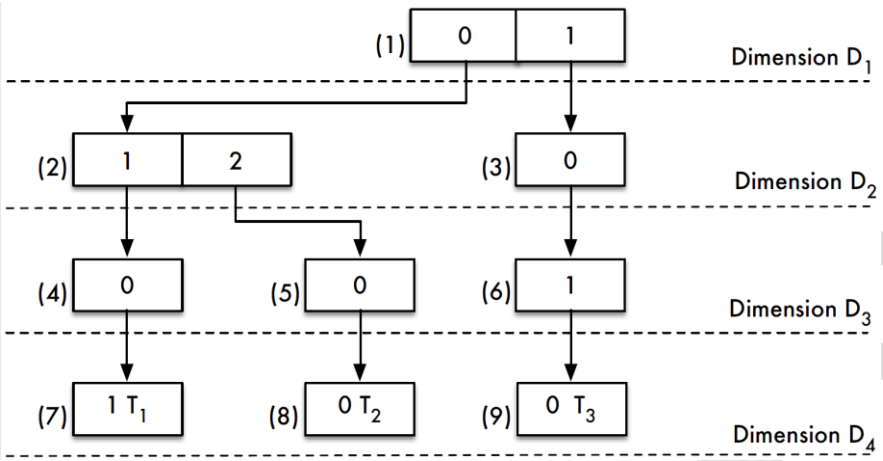

Figure 1 Prefix redundancy elimination within Elf

The main idea of prefix-redundancy elimination is to store such redundant occurrences of values, named a path, only once and not for every point separately. The resulting data structure containing all paths, is a tree of height n, where each level refers to the corresponding dimension. This tree naturally supports efficient execution of multi-column selection predicate queries, as we point out in the remainder. For explanatory reasons, we now illustrate the basic conceptual design of that tree with the help of the data from Table 1.

Conceptual design

Now, we explain the basic design of Elf with the help of the example data in Tab. 1. The data sets consist of a four-dimensional key with an additional reference (Ref) that points to additional data such as BLOB, CLOBS, or image data, as well as tuple identifiers. In case there is no additional data, the Ref is the tuple identifier only.

In Fig. 1, we depict our structure Elf for the example data from Tab. 1. In the first dimension, there are two distinct values, 0 and 1. Thus, the first so-called dimension list, L(1) , contains two entries and one pointer for each element. The respective pointers point to the beginning of the respective dimension lists of the second dimension, L(2) and L(3) . Note, as the first two points share the same value in the first dimension, we observe a prefix redundancy elimination. In the second dimension, we cannot eliminate any prefix redundancy as all attribute combinations in this dimension are unique. As a result, the third dimension contains three dimension lists: L(4) , L(5) , and L(6) . This is the same number as points. Note, the further we go through the dimensions, the smaller the probability that we can eliminate a prefix redundancy, as there are less points that may share the same prefix for a given range of values. However, this depends on the dimension level as well as on the cardinalities of the current and prior dimensions. Therefore, we assume as a good heuristic for the order of dimensions to always take the dimension with the smallest value range at first.

Considering the node structure of Elf seen so far, the structure of the entries changes in the last dimension. In an intermediate dimension, an entry consists of a value and a pointer. In the last dimension, the pointer is interpreted as a tuple identifier or reference (Ref).

In summary, our storage structure is a bushy tree structure of a fixed height resulting in stable search

paths. Furthermore, it features the following properties:

- **Prefix redundancy elimination**: This property allows for efficient pruning of the search space and also introduces light compression rates. ·
- **Fixed depth**: The tree levels are directly defined by the number of involved dimensions that are used for our Elf. ·
- **Ordered in-node elements**: We order all the values inside the nodes. This allows us to stop the evaluation of a node in case the upper limit of the query window is smaller than the current node element (similar to B-trees).

Moreover, our storage structure is not restricted to OLAP-cube scenarios, but generally applicable for multi-dimensional data. Note that our primary design goal for Elf is not compressing the data, but to allow for efficient multi-dimensional querying. For a detailed description of the index structure Elf, including design issues and implementation details see [Kö15]. Furthermore, we argue for a linearized storage layout for the Elf due to performance benefits at hand [KSS14].

Column order selection

First results indicate that the column order in Elf is one important performance factor. That is particular the case for partial-match queries where there are several dimensions not having a selection predicate (or a wildcard). Elf is most efficient in case the dimensions having a selection predicate are in the upper levels. In addition, cardinality of the dimensions are also important, as we observe more prefix-redundancy eliminations in case the upper levels have a low cardinality. Low cardinality also means that we have to scan less dimension list entries. Thus, we use a heuristic in the remainder of this paper to determine a good column order. Note that we build one Elf per data set, not multiple ones. For our heuristic, we analyze a sample of the workload and count the frequency of occurrence of the dimensions. Then, the most frequent dimension is the first level etc. In case there are two dimensions having the same usage frequency, we take the one with the lowest cardinality first. Currently, we work on a fully-fledged cost model and first results are highly promising [Sc16]. They also back-up this heuristic

IV. QUANTITATIVE AND QUALITATIVE EVALUATION

To show the benefits of our Elf index structure, we look at two directions. First, we present the space partition of our Elf and discuss its benefits w.r.t. other traditional index structures. Second, we evaluate the query performance of Elf compared to a sequential scan and a kd-tree implementation.

Space-efficient indexing

Due to the concept of prefix redundancy elimination, we state that we do not index parts of the data space where there is no data. Starting from the root node of an Elf (referring to the first dimension) to an arbitrary level, we consider a sub space of the overall data space. For each path, that means a combination of values for each dimension, that can be constructed descending this Elf, we find at least one point in the data set that has these values. Prefix redundancy elimination also ensures that the nodes get smaller the deeper we descend an Elf. In particular, we consider only the remaining sub space and store all distinct values of points that are contained in the current path. Differently speaking that means the following: Per tree level, we do not only reduce the volume of the data space, but decrease the dimensionality of the remaining sub-space to be considered by one dimension.

Vice versa, if the data set does not contain a specific value in the first dimension of an n-dimensional data set, we can exclude an n − 1 dimensional space from our index. Considering the example data set from Tab. 1, we can immediately answer any query asking for a point having any value in the first dimension, which is larger than 1 and return an empty result. Similarly, we can do that every sub-sequent dimension based on the combination of values of query by incrementally increasing the considered sub- space (i.e., descending a path) indexed by Elf. As the volume of the considered space increases exponentially based on the number of considered dimensions we efficiently prune. Hence, the earlier sparsely populated dimensions come in the Elf, the more space can be excluded resulting into a highly efficient query execution of Elfs.

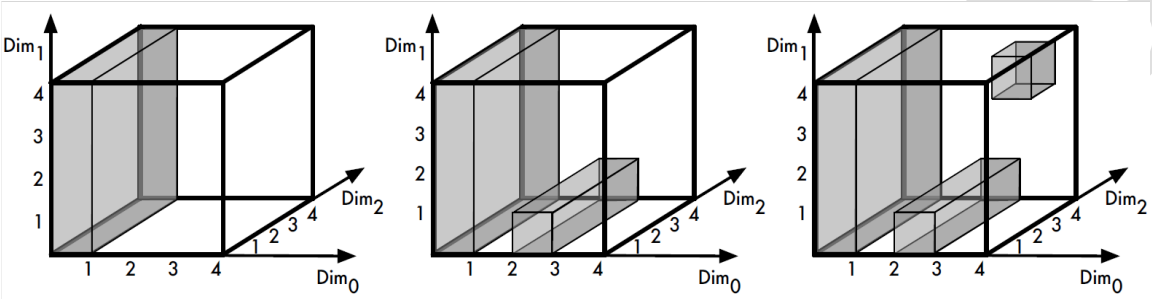

Figure 2 Partitioning of a 3-dimensional space Elf – every piece is indexed except the gray parts.

We visualize the indexed space of a 3-dimensional Elf (everything except the gray boxes is indexed) in Fig. 2 to show the influence of none existing values in the data set. In the first dimension, there is no dimension value 1 and thus, we can exclude a whole three dimensional slice. Furthermore, we can exclude a two-dimensional sub-space (i.e., a line), because there is no point with dim_0=3 and there is no dimension value 1 in the second dimension. Finally, there is no point P[4,4,3], so that we can exclude this point of the data space. Hence, the earlier sparsely populated dimensions come in the Elf, the more space can be excluded leading to an Elf with good performance.

Comparison to other index structures. To compare the space partitioning of Elf with the other index structures, we divide competing approaches into space-partitioning methods (e.g., kd-tree, VA-File) and data partitioning methods (e.g., R-Tree family).

Space-partitioning methods inherently partition the whole data space. That allows them to easily define borders of their partitioning, but also indexes empty space. However, we define the indexing of empty space as a problem that renders index structures to deteriorate due the curse of dimensionality. In contrast, the Elf does only index space that is populated and, thus, is superior to space-partitioning methods in this regard, while still having a simple border definition (the borders are defined by the dimension of the dimension list and the values inside).

Data-partitioning methods partition the space that is populated mainly using special geometric forms. Although, they exclude unpopulated space, the geometric forms still enclose space, where no data points exist. Thus, they suffer due to many false-positives to be evaluated during querying. Furthermore, it is hard to exclude such single non-existing data points from the indexed space, because we would have to align one geometric form per dimension next to this non-existing point to index all neighboring points, but exclude the specific non-existing point. In contrast, we can easily see, that the Elf is able to exclude single non-existing data points from the search space (e.g., P[4,4,3] in Fig. 2).

Towards breaking the curse of dimensionality. Our new data structure deals with the curse of dimensionality in such a way that we do not consider the whole space at ones, but a sequence of exponentially growing (populated) sub-spaces. The first dimension is divided according to all possible values and due to the ordered node values, the search within nodes in a linear or binary search is straight forward. In the next dimension, only these data entries are included in the corresponding lists that occur in the data set. Each list can again be seen as a starting point (of many smaller Elf sub-trees). Hence, problems can be delegated to subordinate sub-trees that work on a highly reduced part of the data space. For equally distributed data, the subordinate indexed space holds only n_i/c_i elements, when n_i is the number of points in the current Elf's sub-tree in dimension i and c_i the cardinality of dimension i. This leads to efficient performance for different types of queries, such as exact match, partial match, as well as range queries.

Towards breaking the curse of dimensionality

In this section, we present query time measurements from a 51-dimensional well-established scientific data set about physical activity monitoring [RS12, Sc13]. It consists of 3,850,505 points, and every data point consists of normalized integer values in the range between 0 and 1023. To evaluate the query performance, we perform three different types of queries: exact match queries, where we identify one data point, partial match queries defined on several dimensions that return all matches that fit to the given search data, and range queries, where all dimensions are restricted by a given parameter range. Note, there exist other queries that could be derived from these types. Due to space limitations, we omit results from other query types.

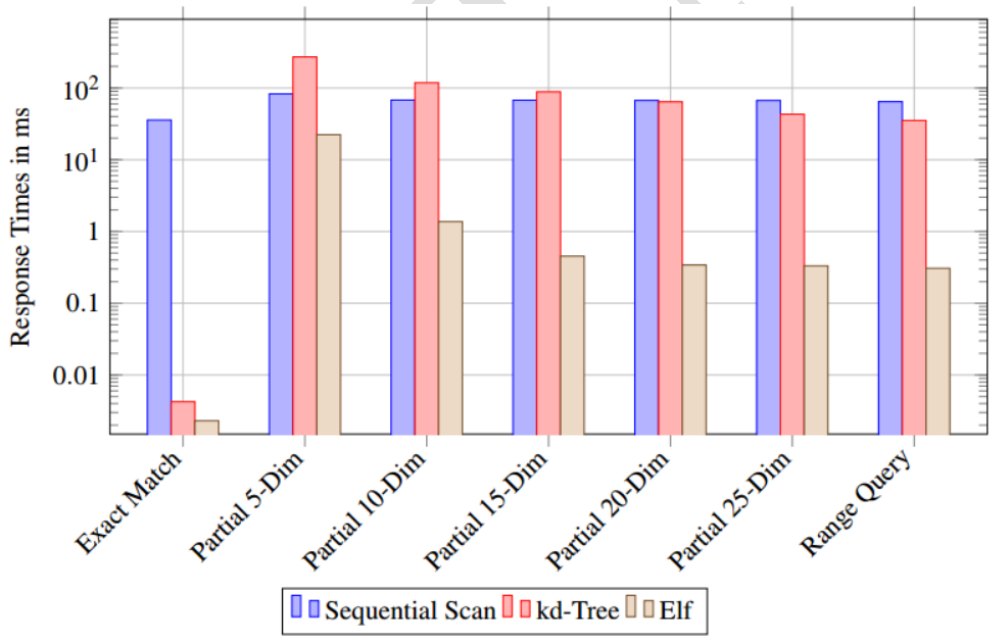

Figure 3 Response Times for different queries and Query Types

For partial match queries, we present different defined numbers of dimensions to show impacts on the performance of Elf and the other approaches. We select the first 5, 10, 15, 20, and 25 dimensions from our data set to vary the impact of the columns in the Elf. For exact-match queries, we take a randomly selected point from the data set and search for it; for range queries, we take two points from the data set and define the lower and upper bounds according to the selected point data in every dimension. For statistical

soundness, we executed each experiment 100 times and present the average response time in our evaluation in Fig. 3.

As the results from Fig. 3 show, our new data structure Elf performs very well on all different query types. In contrast, for exact match queries, a sequential scan requires a lot of unnecessary data accesses. kd-tree as well as Elf outperform the sequential scan in such a scenario, because they can easily prune the search space given a query in that all 51 dimensions are defined for the searched point. Note, in our evaluation, the sequential scan is faster for exact matches than for full range queries. That is because it can skip the point evaluation earlier for a point search than for a range predicate, where some dimensions fit easier.

For partial matches, the sequential scan shows a quite constant behavior. With an increasing number of defined dimensions, the query response times only slightly decrease due to the increasing selectivity of the queries. The kd-tree outperforms the sequential scan, as discussed in the literature, only for higher dimensions, due to its cyclic space partitioning though all dimensions. Hence, whenever we have a wildcard for a dimension and the kd-tree does a split in this dimension, we have to follow two paths instead of pruning the search space. In contrast, Elf copes better with the curse of dimensionality for partial match queries. The effect that an increasing number of dimensions is beneficial for Elf is related to an increasing selectivity due to less masked dimensions by the partial match query.

Range queries are, in fact, a special case of partial match queries with range predicates on all dimensions. Still, for the sequential scan, we can see a similar behavior as for partial match queries. The kd-tree can improve its performance to partial match queries, because it can use every dimension to prune the search space. Nevertheless, Elf performs again much better than both competitors as it outperforms them by at least two magnitudes.

Summary. To summarize, our Elf data structure performs better for our presented query types. This underlines our assumption, that the curse of dimensionality is not applicable to our tree-based data structure due to the fact of the divide-and-conquer method bringing together a sequential linear scan on the one hand and a tree-based approach for all underlying dimensions.

V. CONCLUSION AND FUTURE WORK

Predicate evaluation in large multi-dimensional applications usually involves predicates on multiple dimensions of the data. In this scenario, we need an index structure that is able to exploit the relation between data of several dimensions. In this report, we present Elf, an index structure that exploits prefix redundancies between data of several dimensions. Additionally, Elf features a fixed search path and ordered node entries that allow for a fast pruning inside dimensions. In our evaluation, we have shown that our Elf data structure outperforms a sequential scan and the kd-tree for exact match, partial match, and range queries.

For future work, we have to focus on a better understanding of the impact factors of the Elf. This includes to create a cost model for the Elf w.r.t. the column order and executed queries. We also have to investigate to which extent the Elf is able to support other query types, such as nearest neighbor searches.

VI. REFERENCES

[An10] AnandhaKumar, P.; Priyadarshini, J.; Monisha, C.; Sugirtha, K.; Raghavan, Sandhya: Location Based Hybrid Indexing Structure - R k-d Tree. In: Proc. Int'l Conf. on Integrated Intelligent Computing (ICIIC). IEEE, pp. 140–145, 2010.

[BBK98] Berchtold, Stefan; Böhm, Christian; Kriegel, Hans-Peter: The Pyramid Technique: Towards Breaking the Curse of Dimensionality. In: Proc. Int'l Conf. on Management of Data (SIGMOD). ACM, pp. 142–153, 1998.

[BBS14] Broneske, D.; Breß, S; Saake,G.; Database Scan Variants on Modern CPUs: A Performance Study. In Proc. Internat'l Workshop on In-Memory Data Management and Analytics (IMDM), LNCS, pp 97–111. Springer, 2014.

[Be75] Bentley, Jon: Multidimensional Binary Search Trees Used for Associative Searching. Commun. ACM, 18(9):509–517, 1975.

[Be90] Beckmann, Norbert; Kriegel, Hans-Peter; Schneider, Ralf; Seeger, Bernhard: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In: Proc. Int'l Conf. on Management of Data (SIGMOD). ACM, pp. 322–331, 1990.

[BKK96] Berchtold, Stefan; Keim, Daniel; Kriegel, Hans-Peter: The X-tree: An Index Structure for High-Dimensional Data. In: Proc. Int'l Conf. on Very Large Data Bases (VLDB). Morgan Kaufmann, pp. 28–39, 1996.

[Br14] Broneske, David; Breß, Sebastian; Heimel, Max; Saake, Gunter: Toward Hardware Sensitive Database Operations. In: Proc. Int'l Conf. on Extending Database Technology (EDBT). OpenProceedings.org, pp. 229–234, 2014.

[GBS13] Grebhahn, A.; Broneske, D; Schäler, M.; Schröter, R; Köppen, V.; Saake, G. Challenges in finding an appropriate multi-dimensional index structure with respect to specific use cases. In Foundation of Databases (GvD), pp. 77–82. 2012.

[GK80] Güting, H., & Kriegel, H. P. Multidimensional B-tree: An efficient dynamic file structure for exact match queries. In GI-10. Jahrestagung (pp. 375-388), Springer, 1980.

[Gu84] Guttman, Antonin: R-trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Rec., 14(2):47–57, 1984.

[JL98] Jürgens, Markus; Lenz, Hans-Joachim: The Ra*-tree: an improved R*-tree with materialized data for supporting range queries on OLAP-data. In: Int'l Workshop on Database and Expert Systems Applications. pp. 186–191, 1998.

[KJS97] Kurniawati, R.; Jin, J.; Shepherd, J.: The SS+-tree: An Improved Index Structure for Similarity Searches in a High-Dimensional Feature Space. In: Proc. of SPIE Conf. on
Storage and Retrieval for Image and Video Databases. pp. 110–120, 1997.

[Kö15] Köppen, Veit; Broneske, David; Saake, Gunter; Schäler, Martin: Elf: A Main-Memory Structure for Efficient Multi-Dimensional Range and Partial Match Queries. Technical Report 002-2015, Otto-

von-Guericke-University Magdeburg, 2015.

[KS97] Katayama, Norio; Satoh, Shin'ichi: The SR-tree: An Index Structure for High-dimensional Nearest Neighbor Queries. In: Proc. Int'l Conf. on Management of Data (SIGMOD). ACM, pp. 369–380, 1997.

[KSS14] Köppen, Veit; Schäler, Martin; Schröter, Reimar. Toward Variability Management to Tailor High Dimensional Index Implementations. In IEEE Int'l Conf. on Research Challenges in Information Science (RCIS), pp. 452–457. IEEE, 2014.

[LJB95] Leslie, H., Jain, R., Birdsall, D., & Yaghmai, H. Efficient Search of Multi Dimensional B-Trees. In VLDB (Vol. 95, pp. 11-15), 1995.

[LKN13] Leis, Viktor; Kemper, Alfons; Neumann, Thomas: The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In: Proc. Int'l Conf. on Data Engineering (ICDE). IEEE, pp. 38–49, 2013.

[RS12] Reiss, Attila; Stricker, Didier: Creating and benchmarking a new dataset for physical activity monitoring. In: PETRA. pp. 40:1–40:8, 2012.

[Sc13] Schäler, Martin; Grebhahn, Alexander; Schröter, Reimar; Schulze, Sandro; Köppen, Veit; Saake, Gunter: QuEval: Beyond high-dimensional indexing à la carte. PVLDB, 6(14):1654–1665, 2013.

[Sc16] Schneider, Jonas: Analytic Performance Model of a Main-Memory Index Structure. Bachelor thesis, Karlsruhe Institute of Technology (KIT), 2016.

[Si02] Sismanis, Yannis; Deligiannakis, Antonios; Roussopoulos, Nick; Kotidis, Yannis: Dwarf: Shrinking the PetaCube. In: Proc. Int'l Conf. on Management of Data (SIGMOD). ACM, pp. 464–475, 2002.

[SO82] Scheuermann, P.; Ouksel, M. Multidimensional B-trees for associative searching in database systems. Information systems, 7(2), 123-137, (1982).

[SRF87] Sellis, Timos; Roussopoulos, Nick; Faloutsos, Christos: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In: Proc. Int'l Conf. on Very Large Data Bases (VLDB). Morgan Kaufmann, pp. 507–518, 1987.

[vO90] van Oosterom, Petrus: Reactive Data Structures for Geographic Information Systems. PhD thesis, Rijksuniversiteit te Leiden, 1990.

[WJ96] White, David; Jain, Ramesh: Similarity Indexing with the SS-tree. In: Proc. Int'l Conf. on Data Engineering (ICDE). IEEE, pp. 516–523, 1996.