Nr.: FIN-002-2015

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

Veit Köppen, David Broneske, Gunter Saake, Martin Schäler

FIN-ITI-DBSE

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-002-2015

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

Veit Köppen, David Broneske, Gunter Saake, Martin Schäler

FIN-ITI-DBSE

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Veit Köppen
Postfach 4120
39016 Magdeburg
E-Mail: veit.koeppen@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 22.12.2015

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

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

Veit Köppen
Otto-von-Guericke University
Magdeburg, Germany
veit.koeppen@ovgu.de

David Broneske
Otto-von-Guericke University
Magdeburg, Germany
david.broneske@ovgu.de

Gunter Saake
Otto-von-Guericke University
Magdeburg, Germany
gunter.saake@ovgu.de

Martin Schäler
Karlsruhe Institute of
Technology, Germany
martin.schaeler@kit.edu

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. Especially for subsequent join processing or aggregation, we need to reduce the amount of tuples to be processed. As we have seen an enormous increase of data with the last decade, this kind of selection predicate became more important. Especially in OLAP scenarios or scientific data management tasks, we often face multi-dimensional 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. In this report, we propose Elf - a new tree-based approach to efficiently support such queries. In contrast, to other tree-based approaches, we do not suffer from the curse of dimensionality. The main reason is that we do not apply space or data partitioning methods, like bounding boxes, but incrementally index sub-spaces. In addition, our Elf is cache sensitive, contains an optimized storage layout, fixed search paths, and even supports slight data compression rates. Our experimental results indicate a clear superiority of our approach compared to a highly optimized SIMD sequential scan as competitor. For TPC-H queries with multi-column selection predicates, we achieve a speed-up between factor five and two orders of magnitude, mainly depending on the selectivity of the predicates.

Keywords

data analytics, indexing, main-memory databases, storage structures

1. MOTIVATION

Efficient evaluation of range predicates on large database tables is ever since an important task. Especially for subsequent join-processing, reducing the input by applying the famous optimization rule "pushing down" selections is one important application. With the rise of main-memory systems for analytical and scientific processing, we encounter a fundamental change. For disk-based systems, mostly organized in rows, the main performance bottleneck is loading data into the main-memory. Now, with sufficient main-memory available at reasonable cost, we cannot only count the accesses to main-memory, but also need to take other factors, as CPU utilization and cache misses, into consideration [10, 23, 24, 29]. In concert with this technological change, we also encounter a change in data processing. Due to the increasing amount of data, we observe an increasing number of multi-dimensional data sets from various domains, such as OLAP and scientific computing.

When working with multi-dimensional data sets, we usually have to answer queries with selection predicates on *several* attributes of a table. This is also observable when looking at certain well-known benchmarks, such as TPC-H [34] or Star Schema Benchmark [27], which include such queries.

As an example, we may want to sum up the total amount of goods sold in last year in Europe for a products from European suppliers. This query can be formulated SQL as presented in Listing 1 for instance. We call such a selection a multi-column selection predicate. For the example, we have to evaluate three different selection predicates.

```
SELECT SUM(sold_goods)
FROM A_Table
WHERE supplier_region = 'Europe'
AND customer_region = 'Europe'
AND year(salesdate) = 2014;
```

Listing 1: Example of Multi-column Selection Predicate Query

By looking at the selectivity of all three predicates, they result in large fractions of the overall data set. However, their combined selectivity is often that low (e.g., around one percent of the data) that tree-based indexing schemes seem promising. Recent approaches, like BitWeaving [25] or Column Imprints [32], however, focus only on mono-column selection predicates. Consequently, we need to fully scan each column having a predicate and then compute the final result. We hypothesize that we could highly speed up query performance if we can exploit the combined selective power of all predicates in concert. To the best of our knowledge, there is currently no such approach that has been optimized for main-memory environments.

When looking at classical indexing schemes, we find a vast amount of approaches that allow for evaluating multi-dimensional range queries, which is highly related to evaluation of multi-column

Copyright is held by the owners/authors.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from the authors.

selection predicates. For instance, the R-tree family allows for easy range query evaluation, due to their concept of minimum bounding rectangles [17]. In a similar way, kd-trees naturally support range-query evaluation by their concept of partitioning the space using axis-parallel hyperplanes [5]. However, so far all tree-based approaches suffer from the curse-of-dimensionality [6, 7]. This means in particular, that for a certain number of dimensions (16 according to [6]), simple sequential scans have better performance than these approaches.

There has been a large discussion whether tree-based indexes can somehow be modified to weaken the effects of the curse (e.g., X-trees [7]). However, as undertaken countermeasures only increase the number of dimension 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 are empty, due to the sparse population of these spaces. We argue that this is the actual problem of all these approaches. Hence, we need to come up with a totally different concept that does not index empty spaces, but still allows for evaluation of selection predicates, for instance by separating the data space in an axis parallel manner. With *prefix-redundancy elimination*, we found such a concept.

In this report, we contribute a novel approach for efficient evaluation for multi-column selection predicate evaluation, which we name Elf[1]. Moreover, we empirically show the superiority of our novel approach in comparison to a state-of-the-art SIMD sequential scan 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 and introduce respective build and search algorithms.
3. We define a memory layout for Elf, effectively exploiting the targeted low selectivity scenario of multi-column predicate evaluation.
4. We introduce non-trivial optimizations of the memory layout to counter all negative effects of the curse of dimensionality on our approach.
5. We conduct an experimental evaluation of our approach indicating the superiority of our approach compared to a SIMD sequential scan for exact-match queries and multi-columns selection predicate evaluation queries based on the TPC-H benchmark.

The remainder of this report is structured as follows. In Section 2, we review related work to support our argumentation on limitations of currently used approaches. Based on that information, we define our new approach in Section 3. In this section, we also introduce all optimizations and required algorithms. The empirical evaluation is presented in Section 4.

[1]The underlying idea for Elf is derived from Dwarf, but the unique combination of additional carefully chosen optimizations renders it superior to state of the art approaches. Therefore, we see more magic in it.

2. RELATED WORK

Multi-dimensional data is in the focus of many application scenarios in databases, ranging from statistical, multimedia, or forensic data analysis to On-Line Analytical Processing (OLAP) or Big Data. The complexity behind querying these data sets is discussed on both levels, in application oriented approaches as well as on a more abstract level [16]. To handle data access, former research focused on multi-dimensional indexing (an overview on index structures can be found in [9, 15]) while currently, accelerated main-memory scans have become popular.

Multi-Dimensional Indexing

Weber et al. classify multi-dimensional index structures into space-partitioning and data-partitioning methods [36]. Space-partitioning methods (e.g., VA-File [36], kd-tree [5], p-stable LSH [13]) split the indexed space along predefined lines without considering the data distribution. Hence, we end up with several densely populated and some sparsely populated regions. Still, space-partition methods feature an easy definition of a partitioning of the space and, thus, such a partitioning strategy is a design decision included in our Elf.

In contrast, data-partitioning methods (e.g., R-tree [17] and its derivatives R^+-tree [31], R^*-tree [4], X-tree [7]) index only space, where data can be found. Although this leads to a more complex definition of regions and also overhead in traversing the index structure, data-partitioning methods are able to quickly prune the search space, because they do not index un-populated space. Although effected by the curse of dimensionality, pruning the search space is still an important feature to reach good query performance for high-dimensional queries and should be considered in our Elf as well. In conclusion, we aim at an index structure with a simple partitioning scheme that is also able to exclude regions from the search space.

Accelerated Main-Memory Scans

Due to the curse of dimensionality [6], index structure performance is said to deteriorate for data sets with a dimensionality above 16, which renders a scan faster than index structures. To improve the performance of scan algorithms, especially in main-memory, the exploitation of the hardware has become an important topic. This includes the usage of SIMD (*single instruction multiple data*), e.g., in the work of Zhou and Ross [40], Willhalm et al. [38, 39] and Polychroniou and Ross [28]. Furthermore, cache-consciousness and compression is an important property for accelerated scans as the results of the Column Imprints [32] and BitWeaving [25] show. To reach compatible results, our Elf has to feature a cache-friendly memory layout.

3. ELF STORAGE STRUCTURE

We present our new storage structure for efficient multi-dimensional querying in this section. Firstly, we discuss the design and aspects to overcome the curse of dimensionality. Secondly, we give some insights in constructing and searching the Elf.

3.1 Core Design Idea

It is commonly known that, so far, all tree-based index structures are affected by the curse of dimensionality. 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 [7]. The reasons therefore

are mainly the pure size of the multi-dimensional space and the tendency of data points to be 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-trees [17] and kd-trees [5], 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., [3, 4, 18, 31], spheres, e.g., [35, 37], or mixtures, e.g., [19, 22]) to describe a box that totally encloses a sub-tree or the points in a leaf node. This design features two fundamental limitations. First, all nodes cover 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. Second, by concept, bounding boxes also index space where there is no data. Finally, storing such bounding rectangles requires a lot of storage space, as we need to store two n-dimensional points and the pointers to the sub-trees. 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 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, our approach 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 needed to look for a different concept. This concept is described in the following.

3.2 Conceptual design

Now, we explain the basic design of Elf with the help of the example data in Table 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* can be discarded. This differentiation of data can also be seen as differentiation between hot and cold data [2], where hot data attributes are directly stored within the data structure and cold data attributes are not used within the query process, but can be referenced for other purposes.

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 build an approach in concert with the design principles is prefix redundancy elimination. It is first proposed as part of the dwarf data structure, which materializes a cube operator [33]. In contrast to the dwarf, our approach is designed for efficient multi-dimensional querying. This requires for additional concepts by omitting other features of the dwarf like suffix redundancy elimination. These optimizations go in hand with the critics for the evaluation of dwarfs [14, 26]. Therefore, we do not focus on a fact schema, but design our structure also for OLTP scenarios. However, the multi-dimensionality is more often addressed in the OLAP context. So, we use TPC-H benchmark data in our evaluation in Section 4. A direct comparison of the performance of the data dwarf is therefore not useful and we refer to the results for tailoring index and data structures for specific use cases [12, 20, 21].

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 P_1 and P_2 in the data set, with $P_1 \neq P_2$ having the same values for all dimensions in the interval: $\forall_{d \in [1,k]} P_1[d] = P_2[d]$. Considering the running example in Table 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 $[1, 1]$.

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 running example data from Table 1, before we introduce further optimizations.

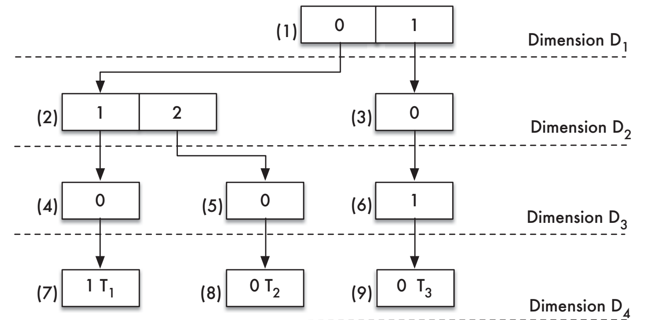

Figure 1: Prefix redundancy elimination within Elf

An example Elf

In Figure 1, we depict our structure Elf for the example data from Table 1. In the first dimension, there are two distinct values, 0 and 1. Thus, the first `dimension list`, $L_{(1)}$, contains two entries and one pointer for each element. The respective pointer points 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 more the probability decreases that we can eliminate a prefix redundancy, as there are less points that may share the same prefix. 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 take always take the smallest cardinality

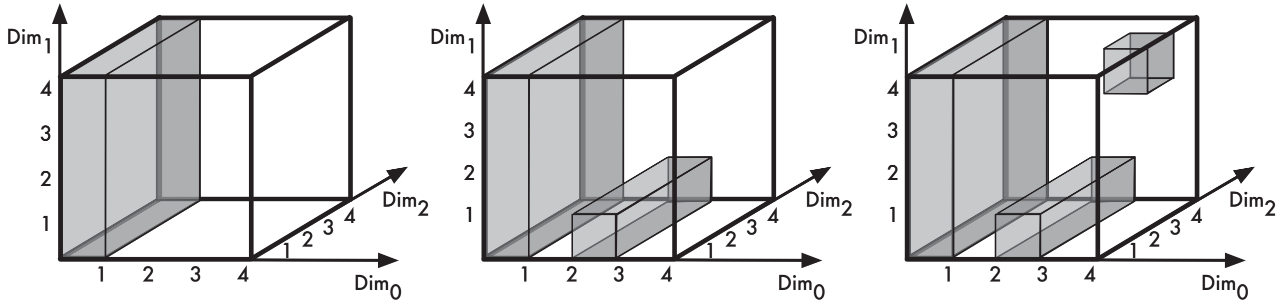

Figure 2: Partitioning of a 3-dimensional space Elf – every piece is indexed except the gray parts.

dimension at first. Note, in our improved design, the first dimension is an exception and should not be selected as the dimension with the smallest cardinality.

Considering the node structure of Elf, 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 reference pointer (*Ref*). In case no *Refs* are required, the entry consists of one value only.

In summary, our storage structure is a bushy tree structure of a fixed height resulting in stable search paths. 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. To further optimize such queries, we also need to optimize the memory layout of the Elf.

Relationship to the design principles

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 consider 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. We visualize this in Figure 2, where we show the indexed space of a 3-dimensional Elf (everything except the gray boxes is indexed). In the first dimension, there is no dimension value 1 and thus, we can exclude a whole slice. Furthermore, we can exclude 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 a small point of the data space. Hence, the earlier sparsely populated dimensions come in the Elf, the more space can be excluded leading to highly performant Elfs.

In summary, our Elf features the following properties:

- **Fixed depth**: the tree levels are directly defined by the number of involved columns (dimensions) that are used for our Elf.
- **Ordered in-node elements**: Within all nodes the values are ordered. This allows for stopping evaluation of a node in case the upper limit of the query window is smaller than the current node element.
- **Prefix redundancy elimination**: This property allows for efficient pruning of the search space and also introduces light compression rates.

However, as discussed, the space is getting sparse very quick so that deeper levels often contain only one element, leading to a linked-list-like data structure. Furthermore, the first dimension list is always a perfect hash map, which should be exploited for efficient queries. Both aspects are addressed in the following section.

3.3 Improving the design

An in-depth analysis of our first design reveals two limitations. Both of them are related to our concept of prefix-redundancy elimination and decrease the performance of our Elf. The first limitation are long lists in the first dimension, which need to be scanned and the second limitation are dimension lists in deeper levels of the tree containing only one key, due to the sparse population of high-dimensional spaces. In the following, we introduce optimizations that remove these limitations.

The hash-map property of the first dimension

The first dimension list includes all distinct values of the first dimension. This results in the limitation that we have to scan a large amount of values in order to find the desired paths. However, due to our design, the values in the first dimension are (already) arranged in a way that allow us to directly use them as a hash map. Thus, we totally remove the overhead for scanning a potentially large number of values. We obtain this hash-map property, because the values in the first dimension are *ordered and dense* (i.e., all values between the smallest and largest value exist) due to the fact of the dictionary encoding. Using these properties of a perfect hash map, we can introduce a second optimization reducing the size required to store the first dimension by Factor 2. So far, we store for every value also the pointer to the next level (dimension). In the first dimension, the value's offset from the start of the dimension list implicitly encodes the value. Thus, we only need to store the pointers. In Figure 3, we depict the first dimension list for six values with pointers P_0 to P_5 illustrating this optimization.

MonoLists to counter the sparsity in high dimensions

When descending a path in an Elf, we reduce the dimensionality of the remaining sub-space by one for each level we descend. These sub-spaces are usually sparsely populated, which is a well-known

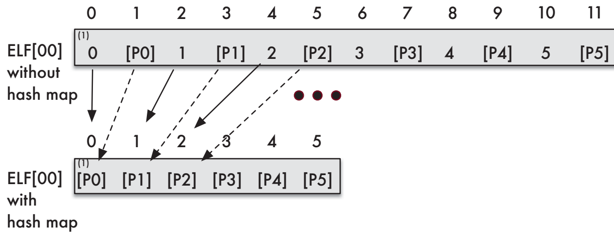

Figure 3: Hash map property of the first `dimension list`

property of high-dimensional spaces. As a result, we often encounter a level where there is only one path and this path belongs to only one point. So, we cannot exploit any more prefix-redundancy eliminations, but create dimension lists containing only value and pointer. For better illustration of this limitation, let us consider the 15-dimensional `Lineitem` table of the 10GB TPC-H benchmark. An analysis of this table reveals, we can only exploit prefix-redundancy elimination until Dimension 7. For any subsequent path, we only find dimension lists belonging only to one point. When searching in this list, we expect to observe a significant performance decrease due to unnecessary jumps to the next dimension lists etc. Thus, removing this limitation is desirable.

Our solution is the introduction of *MonoLists*. The basic idea is to change the memory layout in case we encounter that a certain path becomes unique. That means it belongs only to one point[2]. The purpose of this change is to allow for efficient evaluation of the remaining attribute values, by placing them adjacently in memory. Consequently, we do not need to jump to the sub-seeding dimension lists and save the storage space for the pointer to the next dimension list. Differently speaking that means that we switch from a columnar layout [1] to a row-wise layout. At the end of each MonoList, we find the reference uniquely identifying this point. Using this concept removes the aforementioned drawback.

Resulting improved memory layout

As a result of both improvements we receive the conceptual design of the Elf as depicted in Figure 4. This structure has a more efficient memory design and can make use of further technology regarding in-memory. The different levels for the dimension lists within the Elf are depicted by the different gray-scales. Note, the first dimension exploits the hash map property and consists of two elements for pointing to the next level dimension lists (2 and 12). The end of the list is marked by the negative number of the value. A negative pointer depicts that the next list is a `MonoList`.

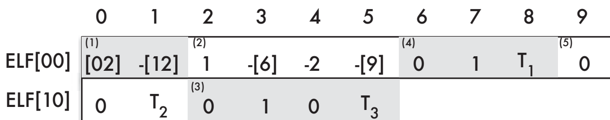

Figure 4: Optimized memory layout

Cold-data avoidance

Another observation can be introduced by analyzing the workload. We often observe several dimensions that are rarely used. These dimensions only add little or even no selective power to our Elf. On the contrary storing this dimension requires storage space, which in final consequence leads to worse caching performance due to missing local adjacency of the hot data. Our Elf offers the possibility to build on a sub-set of the set of dimensions. The remaining attributes of a point can be accessed by its reference that is stored in the Elf. The cold data is stored in columnar table layout. In the remainder of this report, we denote such a cold data avoidance by Elf_n to indicate how many dimensions are indexed. Moreover, we evaluate the benefit of this optimization.

[2]In case we allow for duplicates; this optimization is extended to all points with the same attribute values.

3.4 Implementation

For the implementation of Elf, we use the programming language C++. Due to the good control for memory and a fine granular control of time measuring within C++, we decided for this programming language. Nevertheless, in the following section, we present the construction and the search within the Elf in a more general description.

3.4.1 Construction algorithm

Since we assume that the incoming data are not sorted according to the dimension order, we build a data structure called `InsertElf` at first. The `InsertElf` is an intermediate data structure, where insertions, deletions, and updates can be executed. Although it already features prefix-redundancy elimination, it is not optimized for query execution, because its `DimListElements` are not necessarily stored in an efficient accessible sequence, but may be spread across memory. Thus, the construction of the Elf is split into two phases, where we build the `InsertElf` and then linearize it to have adjacent `DimListElements`, include the HashMap and also build `MonoLists`.

Building the `InsertElf`

The `InsertElf` is stored as an array, where new elements are appended to the end of this array. For being able to traverse the elements of a `dimension list` and to go to the child `dimension list` of a `DimListElement`, each element has a pointer to the array position, where the next `dimension list` is stored, and to the position, where the next element in the current `dimension list` is stored. We present the construction algorithm of the `InsertElf` in Algorithm 1.

```
1  insert (point, tid, INSERT_ELF) {
2      currentDimList ← FIRST_LIST;
3      for (dim ← 0 to NUM_DIM) do
4          dimListElement = getElement (currentDimList, point [dim ],
           INSERT_ELF);
5          if (isValid (dimListElement)) then
6              currentDimList ← getNextDimList (dimListElement,
               INSERT_ELF);
7          else
8              dimListElement ← createRemainingDimLists (dim,
               currentDimList, point, tid, INSERT_ELF);
9              return;
10         end if
11     end for
       // handle duplicate entries
12     dimListElement ← newElement (currentDimList, point [dim − 1],
       INSERT_ELF) ;
13     setTID (dimListElement, tid) ;
14 }
```

Algorithm 1: Build the *InsertElf*

To insert a point in the `InsertElf`, we traverse the `dimension lists` of the `InsertElf` along the coordinates of the point to insert until we find a list that has no element corresponding to our current dimension value. In other words, we descend the `InsertElf`

till the prefix of the point diverges from already stored prefixes. The traversal is implemented as a for-loop (cf. Line 3), that retrieves the element with the dimension value specified by the point (Line 4) and checks whether this element exists (Line 5). On existence, we follow the pointer to the element's child `dimension lists`. Otherwise, we found the maximum prefix and have to create the `dimension lists` for the last dimensions (Line 8) as well as setting the TID in the last dimension. If we can traverse the `InsertElf` till the last dimension, we have found a duplicate point and just have to add the TID to the last `dimension list` (Line 12-13).

Linearization

After building the `InsertElf`, we can descend this insert-optimized data structure and linearize its content into an integer array. The corresponding pseudo code is shown in Algorithm 2. We linearize the Elf using a preorder traversal of the `InsertElf`. This means, we store a full `dimension list` and after that store the `dimension list` of the first child. Apart from the hash map in the first dimension and the `MonoLists`, a `dimension list` of size m consists of m value-pointer pairs stored consecutively in the array.

```
   Result: writePointer pointing to the next dimList in ELF
1  int linearize (ELF, writePointer, dimList, dim, INSERT_ELF) {
2      dimListElement ← getFirstElement (dimList, INSERT_ELF) ;
3      listSize ← getSize (dimList, INSERT_ELF) ;
4      nextListPositions ← new Array[listSize ];
       // Write dimension values and pointers; last value
       treated differently
5      for (i ← 0 to listSize-1)do
6          ELF [writePointer ] ← getValue (dimListElement) ;
7          writePointer ← writePointer +1;
8          nextListPositions [i] ← writePointer;
           // remember where to write the offset of the
           next dimList
9          writePointer ← writePointer +1;
           // point to next location where to add a value;
           not a pointer
10         dimListElement ← getNextElem (dimListElement,
           INSERT_ELF) ;
11     end for
       // Create end of list by setting MSB
12     ELF [writePointer ] ← setMSB (getValue (dimListElement) );
13     writePointer ← writePointer +1;
14     nextListPositions [listSize-1] ← writePointer;
15     writePointer ← writePointer +1;
       // Call remaining lists to linearize themselves
16     dimListElement ← getFirstElement (dimList, INSERT_ELF) ;
17     for (i ← 0 to listSize)do
18         nextList ← getNextDimList (dimListElement,
           INSERT_ELF) ;
19         if (!isMonoList (nextList, dim +1, INSERT_ELF) )then
20             ELF [nextListPositions [i]] ← writePointer;
21             writePointer ← linearize (ELF, writePointer, nextList,
               dim +1, INSERT_ELF);
22         else
23             ELF [nextListPositions [i]] ← setMSB (writePointer) ;
24             writePointer ← linearizeMonoList (ELF, writePointer,
               nextList, dim +1, INSERT_ELF);
25         end if
26         dimListElement ← getNextElem (dimListElement,
           INSERT_ELF) ;
27     end for
28     return writePointer;
29 }
```

Algorithm 2: Linearize the `InsertElf` to build the final Elf

The first step in the linearization is to write the content of the current `dimension list` (Line 5-11). However, at the moment of the linearization of the current list, we do not know, where, e.g., the second child `dimension list` will be written, because all children of the first child `dimension list` are written before the second one. Hence, we cannot write this pointer, but store its position in the array `nextListPositions` (Line 8). With this, we can write the pointers after the linearization of the child `dimension lists`. When writing the elements of a `dimension list`, we always have to treat the last element specifically, because we use the indirect length control by setting the most significant bit (MSB) of the stored value (Line 12-15).

After writing the current `dimension list` by iterating over the current `dimension list`, we have to reiterate over it to follow the paths to the child `dimension lists` (Line 16-26). If the child `dimension list` is a `MonoList`, we have to use the function `linearizeMonoList`, which writes all dimension values of a point consecutively including the point's TID. Otherwise, we can call the function `linearize` recursively. Furthermore, we write the pointers to the child `dimension lists` in this loop, as we know now where they will start (Line 20). Notably, we do the same in case of a `MonoList` except that we set the MSB of the pointer to 1 to indicate that at next, we find a `MonoList` (Line 23).

3.4.2 Search algorithm

In this section, we briefly present the search functionality of our new structure. We restrict ourselves to search for multi-column selection predicates. This can be easily used for exact matches as well as partial match queries.

```
   Result: L Resultlist
1  SearchElf (lower, upper) {
2      L ← ∅;
3      if (lower[0] ≤ upper[0])then
           // predicate on first column defined
           // exploit hash-map
4          start ← lower[0]; stop ← upper[0];
5      else
6          start ← 0; stop ← max {C_1};
7      end if
8      for (offset ← start to stop)do
9          pointer ← Elf[offset];
10         if (noMonoList (pointer))then
11             SearchDimList (lower, upper, pointer, dim ← 1, L);
12         else
13             L ← L+SearchML
               (lower, upper, unsetMSB(offset), dim ← 1, L);
14         end if
15     end for
16     return L;
17 }
```

Algorithm 3: Search in the Elf

The first part for searching in an Elf deals with the evaluation of the query for the first dimension. This has to be differentially handled due to the hash map within the first dimension list. We present this part in Algorithm 3. As parameters the lower and upper bounds for all dimensions are required. Note, if no definition for a single dimension boundary is given, this query is a partial match query and all elements in this dimension have to be retrieved.

The result of the search algorithm is a list L with all references to points that fulfill the search criteria. Note, in the case that there are no references used, we include the points themselves.

In Line 3 the hash map property is exploited for the case that the query hyper rectangle contains a predicate on the first dimension. Otherwise, we propagate the search for each value in the first list to the next level. The next level is either a `MonoList` referring only to one point with its values adjacently located in memory or a dimension list. Due to this differentiation, we also have to check whether or not the next level dimension list is of this type. In this case, all remaining elements are checked for matching to the query constraints and in a positive outcome included into L. Due to the

simplicity of this method, we do not depict it as an algorithm.

The search in a dimension list that is no MonoList is depicted in Algorithm 4. Besides the lower and upper query ranges, this algorithm needs as input the start offset of the current dimension as well as the current dimension level. Additionally, we also need the result list L.

```
1  SearchDimList (lower, upper, startlist, dim, L) {
2      if (lower[dim] ≤ upper[dim])then
3          position ← startList;
4          do
5              if (isIn (lower[dim], upper[dim], Elf[position]))then
6                  pointer ← Elf[position + 1];
                   // start of next list in dim+1
7                  if (noMonoList (pointer))then
8                      SearchDimList
                       (lower, upper, pointer, dim + 1, L);
9                  else
10                     L ← L+SearchML (lower, upper,
                       unsetMSB(pointer), dim + 1, L);
11                 end if
12             else
13                 if (Elf[position] > upper[dim])then
14                     return;// abort
15             end if
16             position ← position + 2;
17         while (notEndOfList (Elf[position]));
18     else
           // call SearchDimList or SearchML with dim + 1
              for all elements
19     end if
20 }
```

Algorithm 4: Scan one dimension list within an Elf

The start offset marks the first element value in the dimension list (Line 3). If the search predicate is defined for this dimension, we check every value in this dimension list and propagate the search to the next level or a MonoList in case of matching. We stop checking further values in this list either if the next higher value is larger than the defined search attribute value (Line 13) or the list end is reached (Line 17). This is possible as we store attribute values in an ordered fashion. Our search algorithm is a depth-first search, because we firstly go into the next child level for identification of results (Line 5) before checking the next element value in this level. The reason therefore is that we assume that the curse of dimensionality results in fewer values matches, which require to visit deeper levels. More generally the idea is that we want to quickly reach a level where we have a low selectivity, so that we do not need to jump to the next level. The claim is that we achieve maximum performance by optimizing the algorithm to be able to exclude all values of a dimension list from further consideration as fast as possible. That's why attribute values are located next to each other in memory. This claim is supported by the tendency of multi-dimensional spaces of being sparsely populated and the application scenario for low selectivity. Nevertheless, our Elf is not limited to such an algorithm. In case future research indicates that a different algorithm may result in better performance, for instance for special data distributions, we could also change the algorithm. However, initial test with different algorithms and slightly modified data layouts did not show a superiority of these adaptions.

4. EMPIRICAL EVALUATION

This section presents small micro-benchmarks for the properties of our presented index structure. We differentiate between build times and storage consumption for creating our Elf. The query performance is evaluated in two different scenarios. In the first scenario, we investigate exact-match queries with a multi-dimensional query predicate. In the second scenario, we execute selected queries from the TPC-H benchmark to investigate partial match query performance.

We evaluate two different implementations of our proposed index structure. On the one hand, we evaluate a 32-bit implementation of our Elf, where all dictionary compressed values as well as pointers to the next dimension are stored in a 32-bit Integer. On the other hand, we implemented a 64-bit Elf where the pointers are represented by a 64-bit Integer. For a fair comparison, we use 32-bit data values for this implementation, too.

Additionally, we use two competitors for our evaluation. On the one side, we use the kd-tree [5], due to the promising results from [30]. On the other side, we also want to compare our approach against a non-index structure, which is optimized for new technologies: a columnar SIMD sequential scan [11]. Such an optimized scan is a good competitor, because it is said to be the fastest access method for high-dimensional data [8].

As evaluation data and benchmark, we decide for the well-known TPC-H benchmark and data [34]. This benchmark is quite close to our motivation for developing Elf and it is available for a sound evaluation including repeatability and confirmability. However, due to several parameters (e.g. scaling factors) as well as queries, we restrict our evaluation to the following evaluation design decisions:

- Scaling factors: $s = 10$, $s = 20$, $s = 30$, $s = 50$, and $s = 100$ are used.
- Queries: Q1, Q6, Q10, Q14, Q17, and Q19 are selected, because they provide a reasonable set of predicates on the biggest TPC-H tables.

We do not use directly the TPC-H data, but the dictionary compressed data, where we only use integer values. Note, for a fair comparison we use these data as input for all investigated data and index structures. This goes along with a mapping of the queries to the dictionary compressed values, too. Therefore, we also omit the dictionary look-ups.

4.1 Experiment 1: Storage consumption

One important influence factor for performance evaluations is the storage consumption. For index and storage structures we differentiate between data information and index information. The consumption for a sequential scan just requires the data and therefore, it is a benchmark for all structures. A typical index structure for using storage for data and index information is the kd-tree.

Comparing the storage consumption in our scenario, we select the scaling factor $s = 10$ in Figure 5. Furthermore, we present the results for the sequential data, the kd-tree and four versions of the Elf. Two include all data information and two use only seven dimensions, whereas the others are addressed via a reference. We name the later ones Elf-reduced. In the reduced version only the first seven attributes are included in the Elf and the others are stored in a row-wise structure as the original data elements. As order of dimensions we use the following attributes:

1. Shipdate
2. Discount
3. Quantity
4. Linestatus
5. Returnflag
6. Shipinstruct
7. Shipmode
8. Linenumber
9. Tax
10. Commitdate
11. Receiptdate
12. Suppkey
13. Partkey
14. Extendedprice
15. Orderkey

Two remarkable points can be seen in Figure 5. Firstly, the Elf achieves a slight compression due to its exploitation of the prefix

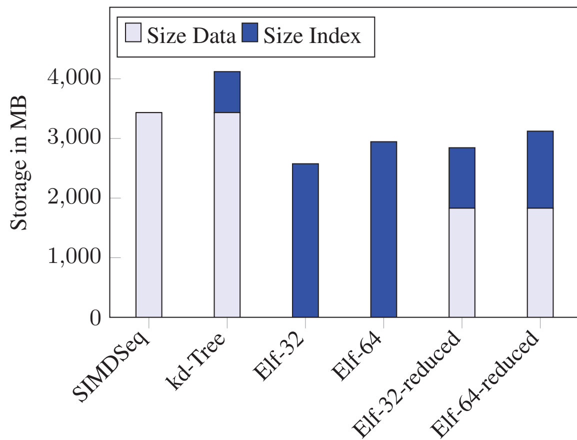

Figure 5: Storage consumption for `Lineitem` table

redundancies. Note, this is additional to the dictionary compression. Secondly, of course, the 64-bit version requires more storage. Nevertheless, this increase does not lead to a higher consumption than the columnar stored data.

For a cold-data example, we present a reduced version of the Elf that has a smaller amount of dimensions and all remaining dimensions are stored via a reference. In our evaluation we use seven dimensions for the Elf. Although the references consume additional storage and only seven dimensions are used within the Elf, a slight compression compared to the columnar stored data is also achieved.

We also evaluated other scaling factors, but besides the increase of storage consumption, there is no real difference of ratios compared to $s = 10$. Therefore, we omit the other scaling factors.

4.2 Experiment 2: Build Times

As second performance evaluation, we focus on build times for our data structure. In this evaluation, we use our construction algorithm as presented in Section 3.4. Again, we use the `Lineitem` table with a scaling factor $s = 10$. Due to the fact that the original dictionary encoded data is stored in a row-wise way, our SIMD implementation also requires build times for the columnar design of the data.

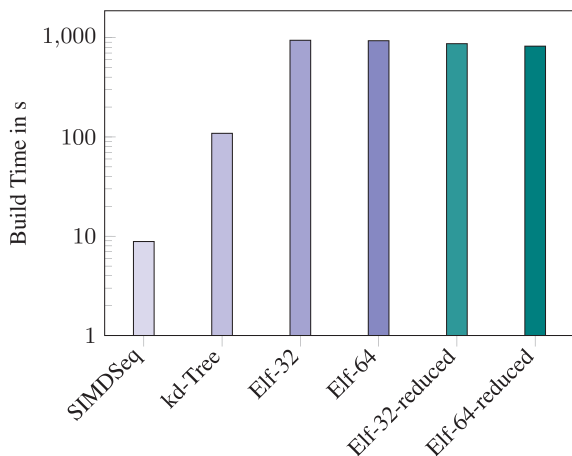

Figure 6: Build time for `Lineitem` table of 10GB TPC-H

Again, we use the same candidates for our comparison. We present our results in Figure 6. The rearrangement from a row-wise structure to a columnar store takes approximately 9 s and is necessary for our SIMD sequential scan. The kd-tree implementation requires 109 s and is quite slow. Our 32- and 64-bit implementations do not really differ and therefore, we can state that the pointer implementation does not influence the construction times. Due to the fact that we do not have to change the cold data for our reduced implementation, the corresponding build times are slower than the full dimensional Elf with 931.8 s for Elf-64. The difference is about 100 s. As a result, we can state that the reduced-64-bit Elf requires 821 s for the construction and is the fastest of our designs. However, the construction time is the weakest point of our structure. Therefore, we see the need for additional research in the construction algorithm.

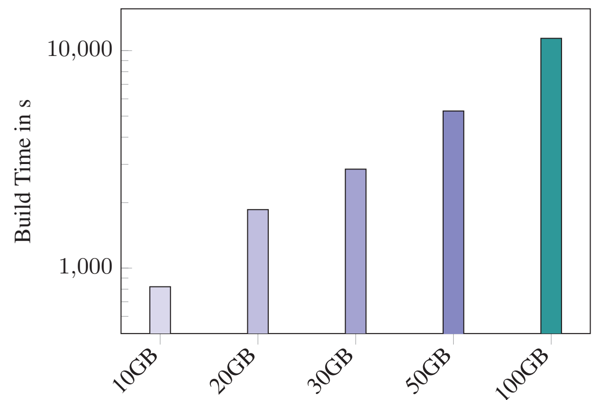

Figure 7: Elf-64-reduced build times for `Lineitem` table

We also use different scaling factors ranging from $s = 20$ to $s = 100$. However, each Elf variant shows linear increase in the build times. Therefore, we only present here one candidate, Elf-64-reduced, in Figure 7. Note, we use a logarithmic scale for the build times.

4.3 Experiment 3: Exact Match Queries

In the third evaluation, we analyze the response behavior for a fully specified query. For this experiment, we query the `Lineitem` table and specify every dimensional attribute. This can be also seen as a primary key constraint in the case, a typical OLAP fact table is queried in a star schema approach. Note, we use all 15 attributes and leave the reduced versions in this scenario out, because they are not designed for this evaluation.

We measure 100 randomly selected exact match queries in our evaluation. The SIMD sequential scan requires more than 1.320 s, whereas the kd-tree does not even require 0.75 ms. Note, this is more than three orders of magnitude. However, our implementation with a 32-bit point requires about 0.11 ms which is approximately 7 times faster than the kd-tree. Additionally, we assume that the 64-bit pointer implementation of our Elf is preferred in a 64-bit hardware environment. Therefore, the execution time of less than 0.089 ms can be explained.

For other scaling factors (20, 30, 50, and 100), we obtain similar results. In Figure 9, we present the results for our Elf-64 implementation. Due to the very fast response times, we executed 1000 exact match queries in this part of the experiment. A linear trend, as in the scenario of build times is observable for scaling factors $s = 20$ and more. However, the difference in the response times for $s = 10$

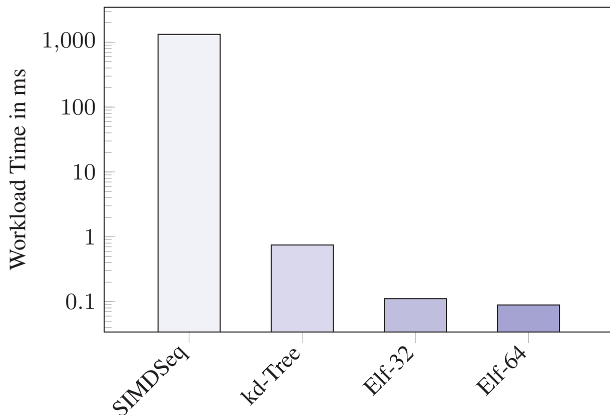

Figure 8: Workload time for exact-match queries on `Lineitem` table (s = 10)

and $s = 20$ do not differ significantly. We assume that the first dimension (in our Elf implemented as hash map) plays a vital role due to the fact that increasing scaling factors do not influence the first dimension and therefore, we obtain a stable first dimension list. Due to the filling in lower scaling factors, we have a very similar computational effort with our implementation which leads to similar response times.

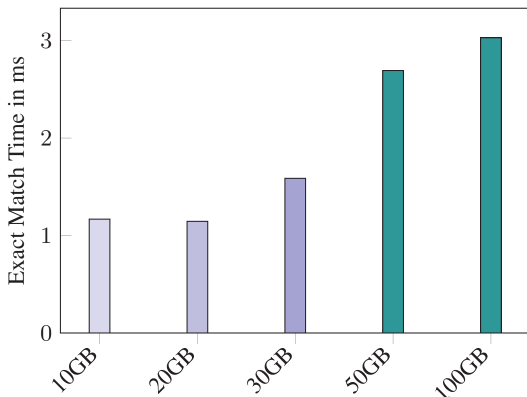

Figure 9: Elf-64-reduced exact match response times for `Lineitem` table at different scaling factors

With the increase of data, a less than linear relation for execution times for exact match queries can be seen. We interpret this as a good indicator for scalability of our Elf design and good scaling implementation.

The results from Figure 8 as well as Figure 9 underline the superiority for exact match queries in a high-dimensional context of our implementation. However, we assume that this type of query is not really significant for practical applications, such as OLAP. Therefore, we evaluate our approach in the following with the help of the TPC-H benchmark [34].

4.4 Experiment 4: TPC-H evaluation

In our last experiment setting, we are interested in typical queries from the domain of decision making. Therefore, we use the TPC-H data and queries. Due to the complexity of the underlying schema and possible queries, we select only an excerpt from this benchmark. We differentiate two main query types from the TPC-H queries: (I) Queries on one predicate column and (II) multi-dimensional queries. Although we focus on high-dimensional queries, we also want to evaluate query scenarios, where our structure is not directly designed for.

In Table 2, we present our selected queries. $Q1$, $Q10$, and $Q14$ use only a single selection attribute. However, very different selectivities (σ) are addressed in these queries. Furthermore, we use the same design and order of dimensions as presented earlier. Queries $Q6$, $Q17$, and $Q19$ query multiple predicates. Note, $Q19$ uses two tables (`Lineitem` and `Part`) and we also use two structures for both tables.

	example σ in %	predicate columns	Col_{Elf}
Q1	98.0	l_date	0
Q10	1.72	l_returnflag	4
Q14	1.3	l_date	0
Q6	1.72	l_date, l_discount, l_quantity	{0,1,2}
Q17	0.099	p_brand, p_container	{1,2}
LQ19	1.4	l_quantity, l_shipinstr, l_shipmode	{2,5,6}
PQ19	0.083	p_brand, p_container, p_quantity	{1,2,3}

Table 2: Query details for mono and multi-column selections

We executed 1000 randomly selected queries and present the measured response times in Figure 10. Due to the fact that our kd-tree implementation always requires at least a magnitude more than the SIMD sequential scan, we leave this competitor out of the result presentation. Otherwise, the graphical interpretation would be more complicate.

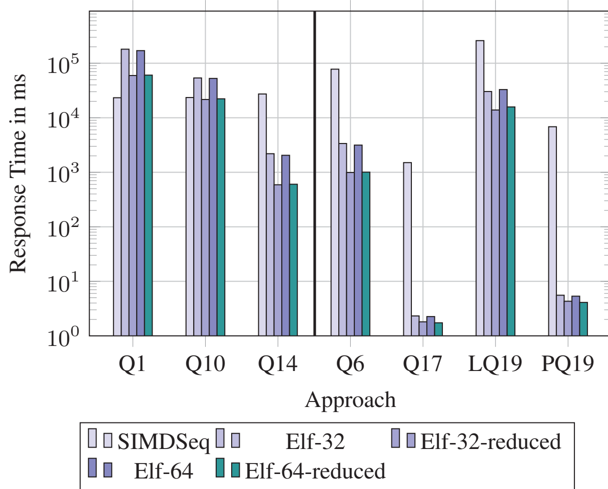

Figure 10: Response Times for TPC-H queries (s = 10)

For the single predicate queries the selectivity plays an important role. In the case of a very big result (compared to the complete table) ($Q1$), our Elf has a quite bad response rate and the SIMD sequential scan outperforms it. However, applying the cold data scenario (again with seven dimensions) reduces this gap significantly. For these queries, the 32- or 64-bit implementation of Elf does not influence the response times. For a lower selectivity ($Q10$), our full Elf requires only some additional time compared to the SIMD scan, if the queried attribute is not one of the first ordered dimensions. However, the order of dimensions also plays an important role, as it can be directly seen in Query $Q14$, where the queried attribute is the first dimension of our Elf. In this scenario, our Elf outperforms

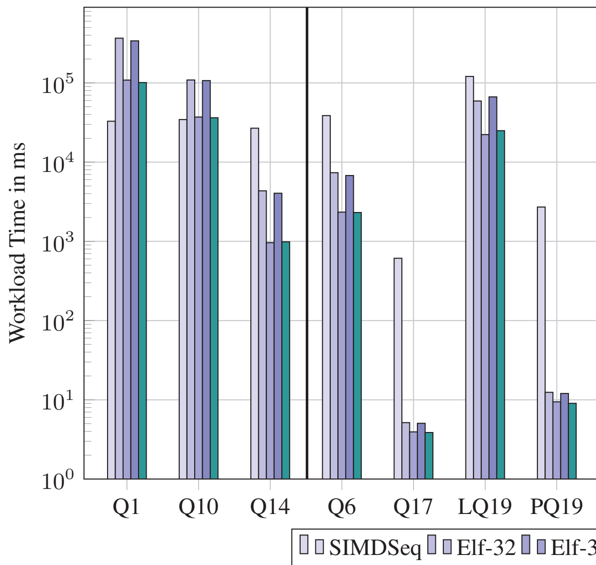

Figure 11: Workload times for TPC-H queries (s = 20)

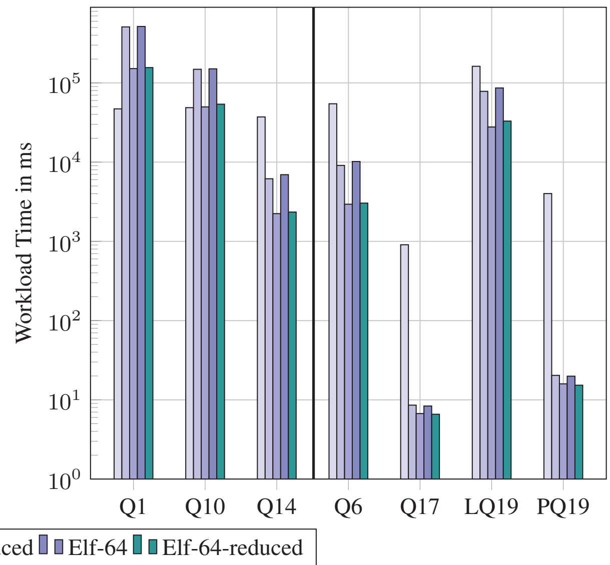

Figure 12: Workload times for TPC-H queries (s = 30)

the SIMD scan by more than one order of magnitude. Note, the selectivity is comparable to $Q10$.

For multi-column selection predicates, the performance of the Elf is much better than the SIMD scan. The reduced versions of our implementation always have the best response times, but we have to state that the queried attributes are always in the Elf stored and therefore, these data are marked as hot data. Nevertheless, the performance of the Elf is especially outstanding for $Q17$ and the `Part` table in $Q19$. We explain this with the low selectivity of the query and the occurrence of the queried attributes in the first dimensions.

We also executed the experiment series for different scaling factors. We present the corresponding results for $s = 20$ in Figure 11 and for $s = 30$ in Figure 12.

As it can be easily seen, the measured workload times are quite close and do not show a significant difference. An increase for the workload times of all measured values is inherent. Our Elf cannot compete with the SIMD sequential scan in $Q1$, However, as in the scenario $s = 10$, the reduced versions are comparable to the SIMD scan for $Q10$. In $Q14$, the measured results for all implementation versions outperform the SIMD scan, which proves the applicability for one column selection predicates with a low selectivity.

The multi-dimensional query types ($Q6$, $Q17$, and $Q19$) are again similar to the presented results for $s = 10$.

For scaling factors $s = 50$ and $s = 100$, we present the response times in Figure 13 and in Figure 14 respectively. In these scenarios, the addressed data within our Elf cannot be handled in the 32-bit implementation anymore. Therefore, we present only the SIMD sequential scan and both versions of the Elf-64. However, the result interpretation is very similar to the other scaling factors. Furthermore, we can derive a linear relationship of scaling factors and response times for our approach. This drives us to the conclusion that our approach is scalable and can handle big amounts of data very efficiently.

5. CONCLUSION AND FUTURE WORK

Predicate evaluation in large OLAP applications usually involves scanning multiple columns. In this scenario, we need an index structure that is able to exploit the relation between data of several columns. In this report, we present Elf, an index structure that exploits prefix redundancies between data of several columns. Additionally, Elf features a fixed search path, a cache-friendly memory layout, and comes with an additional slight compression of the data. In our evaluation, we have shown that different our Elf data structure outperforms a SIMD sequential scan and the kd-tree for exact match as well as partial match queries on the TPC-H database by several orders of magnitude. Furthermore, Elf shows a sub linear scaling with increasing data sizes showing its potential for huge data sets.

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. Furthermore, an extension of Elf to also store aggregates or support grouping operations seems possible given the structure of our index structure. Also, we have to investigate to which extent the Elf is able to support join processing.

Acknowledgments

This work was supported by many people who gave their comments, ideas and suggestions for improving the paper and Elf. We would especially thank Wolfram Fenske, Reimar Schröter, Sebastian Breßfor their valuable comments.

6. REFERENCES

[1] ABADI, D., BONCZ, P., AND HARIZOPOULOS, S. Column oriented database systems. *PVLDB 2*, 2 (2009), 1664–1665.

[2] ALEXIOU, K., KOSSMANN, D., AND LARSON, P.-A. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB 6*, 14 (Sept. 2013), 1714–1725.

[3] ANANDHAKUMAR, P., PRIYADARSHINI, J., MONISHA, C., SUGIRTHA, K., AND RAGHAVAN, S. Location based hybrid indexing structure - R k-d Tree. In *Proc. Int'l Conf. on Integrated Intelligent Computing (ICIIC)* (2010), IEEE, pp. 140–145.

[4] BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. The R*-tree: An efficient and robust access

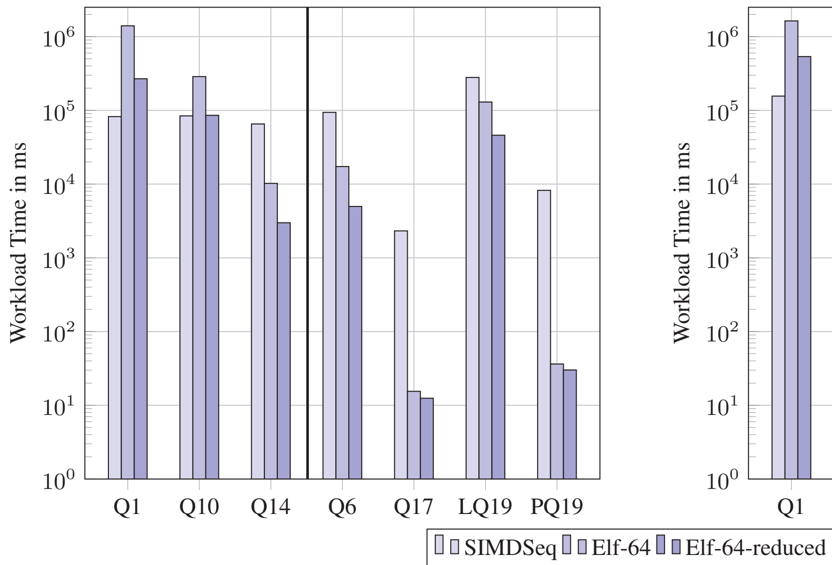

Figure 13: Workload times for TPC-H queries (s = 50)

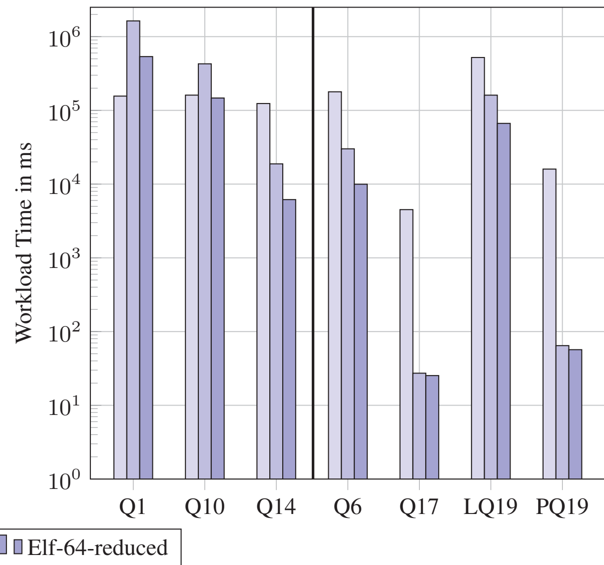

Figure 14: Workload times for TPC-H queries (s = 100)

method for points and rectangles. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (1990), ACM, pp. 322–331.

[5] BENTLEY, J. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[6] BERCHTOLD, S., BÖHM, C., AND KRIEGEL, H.-P. The Pyramid Technique: Towards breaking the curse of dimensionality. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (1998), ACM, pp. 142–153.

[7] BERCHTOLD, S., KEIM, D., AND KRIEGEL, H.-P. The X-tree: An index structure for high-dimensional data. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (1996), Morgan Kaufmann, pp. 28–39.

[8] BEYER, K., GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. When is "nearest neighbor" meaningful? In *Proc. Int'l Conf. on Database Theory (ICDT)* (1999), Springer, pp. 217–235.

[9] BÖHM, C., BERCHTOLD, S., AND KEIM, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3 (2001), 322–373.

[10] BRONESKE, D., BRESS, S., HEIMEL, M., AND SAAKE, G. Toward hardware-sensitive database operations. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)* (2014), OpenProceedings.org, pp. 229–234.

[11] BRONESKE, D., BRESS, S., AND SAAKE, G. Database scan variants on modern CPUs: A performance study. In *Int'l Workshop on In-Memory Data Management* (2014), vol. 8921 of *LNCS*, Springer, pp. 97–111.

[12] BRONESKE, D., DOROK, S., KÖPPEN, V., AND MEISTER, A. Software design approaches for mastering variability in database systems. In *German Nat'l Workshop on Foundations of Databases* (Oct 2014), vol. 1313 of *CEUR Workshop Proceedings*, pp. 47–52.

[13] DATAR, M., IMMORLICA, N., INDYK, P., AND MIRROKNI, V. S. Locality-sensitive hashing scheme based on P-stable distributions. In *Proc. Annual Symp. on Computational Geometry (SoCG)* (2004), ACM, pp. 253–262.

[14] DITTRICH, J., BLUNSCHI, L., AND SALLES, M. Dwarfs in the rearview mirror: How big are they really? *PVLDB 1*, 2 (2008), 1586–1597.

[15] GAEDE, V., AND GÜNTHER, O. Multidimensional access methods. *ACM Comput. Surv. 30* (1998), 170–231.

[16] GREBHAHN, A., BRONESKE, D., SCHÄLER, M., SCHRÖTER, R., KÖPPEN, V., AND SAAKE, G. Challenges in finding an appropriate multi-dimensional index structure with respect to specific use cases. In *German Nat'l Workshop on Foundations of Databases* (2012), pp. 77–82.

[17] GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec. 14*, 2 (1984), 47–57.

[18] JÜRGENS, M., AND LENZ, H.-J. 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* (1998), pp. 186–191.

[19] KATAYAMA, N., AND SATOH, S. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (1997), ACM, pp. 369–380.

[20] KÖPPEN, V., HILDEBRANDT, M., AND SCHÄLER, M. On performance optimization potentials regarding data classification in forensics. In *Proc. German Nat'l Conf. on Business, Technology, and Web (BTW) - Workshops* (2015), vol. 242 of *LNI*, Köllen Verlag, pp. 21–36.

[21] KÖPPEN, V., SCHÄLER, M., AND SCHRÖTER, R. Toward variability management to tailor high dimensional index implementations. In *Proc. Int'l Conf. on Research Challenges in Information Science (RCIS)* (2014), IEEE, pp. 452–457.

[22] KURNIAWATI, R., JIN, J., AND 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* (1997), pp. 110–120.

[23] LEHMAN, T., AND CAREY, M. A study of index structures for main memory database management systems. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (1986), pp. 294–303.

[24] LEIS, V., KEMPER, A., AND NEUMANN, T. The adaptive radix tree: Artful indexing for main-memory databases. In *Proc. Int'l Conf. on Data Engineering (ICDE)* (2013), IEEE, pp. 38–49.

[25] LI, Y., AND PATEL, J. Bitweaving: Fast scans for main memory data processing. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (2013), ACM, pp. 289–300.

[26] MICHALARIAS, I., OMELCHENKO, A., AND LENZ, H.-J. FCLOS: A client-server architecture for mobile OLAP. *Data Knowl. Eng. 68*, 2 (2009), 192 – 220.

[27] O'NEIL, P., O'NEIL, E., CHEN, X., AND REVILAK, S. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking* (2009), Springer, pp. 237–252.

[28] POLYCHRONIOU, O., AND ROSS, K. Vectorized bloom filters for advanced SIMD processors. In *SIGMOD Workshop DaMoN* (2014), ACM.

[29] RAO, J., AND ROSS, K. Making B^+-Trees cache conscious in main memory. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (2000), ACM, pp. 475–486.

[30] SCHÄLER, M., GREBHAHN, A., SCHRÖTER, R., SCHULZE, S., KÖPPEN, V., AND SAAKE, G. QuEval: Beyond high-dimensional indexing à la carte. *PVLDB 6*, 14 (2013), 1654–1665.

[31] SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (1987), Morgan Kaufmann, pp. 507–518.

[32] SIDIROURGOS, L., AND KERSTEN, M. Column imprints: A secondary index structure. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (2013), ACM, pp. 893–904.

[33] SISMANIS, Y., DELIGIANNAKIS, A., ROUSSOPOULOS, N., AND KOTIDIS, Y. Dwarf: Shrinking the PetaCube. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (2002), ACM, pp. 464–475.

[34] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC benchmark H (decision support). Tech. Rep. 2.17.1, 2014.

[35] VAN OOSTEROM, P. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Rijksuniversiteit te Leiden, 1990.

[36] WEBER, R., SCHEK, H.-J., AND BLOTT, S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (1998), Morgan Kaufmann Publishers Inc., pp. 194–205.

[37] WHITE, D., AND JAIN, R. Similarity indexing with the SS-tree. In *Proc. Int'l Conf. on Data Engineering (ICDE)* (1996), IEEE, pp. 516–523.

[38] WILLHALM, T., BOSHMAF, Y., PLATTNER, H., POPOVICI, N., ZEIER, A., AND SCHAFFNER, J. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB 2*, 1 (2009), 385–394.

[39] WILLHALM, T., OUKID, I., MÜLLER, I., AND FAERBER, F. Vectorizing database column scans with complex predicates. In *VLDB Workshop ADMS* (2013), pp. 1–12.

[40] ZHOU, J., AND ROSS, K. Implementing database operations using SIMD instructions. In *Proc. Int'l Conf. on Management of Data (SIGMOD)* (2002), ACM, pp. 145–156.