2017 IEEE 33rd International Conference on Data Engineering

Accelerating multi-column selection predicates in main-memory – the Elf approach

David Broneske
University of Magdeburg
david.broneske@ovgu.de

Veit Köppen
University of Magdeburg
veit.koeppen@ovgu.de

Gunter Saake
University of Magdeburg
gunter.saake@ovgu.de

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

***Abstract*—Evaluating selection predicates is a data-intensive task that reduces intermediate results, which are the input for further operations. With analytical queries getting more and more complex, the number of evaluated selection predicates per query and table rises, too. This leads to numerous multi-column selection predicates. Recent approaches to increase the performance of main-memory databases for selection-predicate evaluation aim at optimally exploiting the speed of the CPU by using accelerated scans. However, scanning each column one by one leaves tuning opportunities open that arise if all predicates are considered together. To this end, we introduce Elf, an index structure that is able to exploit the relation between several selection predicates. Elf features cache sensitivity, an optimized storage layout, fixed search paths, and slight data compression. In our evaluation, we compare its query performance to state-of-the-art approaches and a sequential scan using SIMD capabilities. Our results indicate a clear superiority of our approach for multi-column selection predicate queries with a low combined selectivity. For TPC-H queries with multi-column selection predicates, we achieve a speed-up between a factor of five and two orders of magnitude, mainly depending on the selectivity of the predicates.**

(a)
Q6.1 l_shipdate >= [DATE] and l_shipdate < [DATE] + '1 year'
Q6.2 and l_discount between [DISCOUNT] $-$ 0.01 and [DISCOUNT] + 0.01
Q6.3 and l_quantity < [QUANTITY]

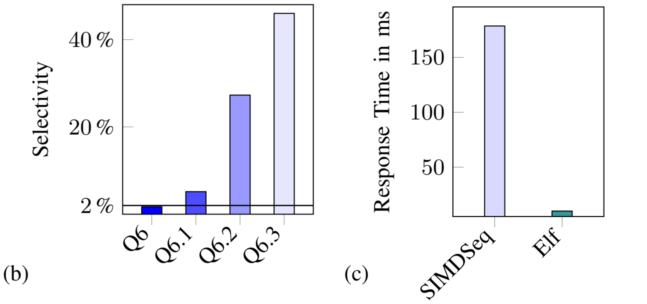


Fig. 1. (a) WHERE-clause, (b) selectivity, and (c) response time of TPC-H query Q6 and its predicates Q6.1 - Q6.3 on Lineitem table scale factor 100

I. Introduction

Predicate evaluation is an important task in current OLAP (Online Analytical Processing) scenarios [1]. To extract necessary data for reports, fact and dimension tables are passed through several filter predicates involving several columns. For example, a typical TPC-H query involving several column predicates is Q6, whose WHERE-clause is visualized in Fig. 1(a). We name such a collection of predicates on several columns in the WHERE-clause a *multi-column selection predicate*. Multi-column selection predicate evaluation is performed as early as possible in the query plan, because it shrinks the intermediate results to a more manageable size. This task has become even more important, when all data fits into main memory, because the I/O bottleneck is eliminated and, hence, a full table scan becomes less expensive.

In case all data sets are available in main memory (e.g., in a main-memory database system [2], [3], [4]), the selectivity threshold for using an index structure instead of an optimized full table scan is even smaller than for disk-based database systems. In a recent study, Das et al. propose to use an index structure for very low selectivities only, such as values under 2 % [5]. Hence, most OLAP queries would never use an index structure to evaluate the selection predicates. To illustrate this, we visualize the selectivity of each selection predicate for the TPC-H Query Q6 in Fig. 1(b). All of its single predicate selectivities are above the threshold of 2 % and, thus, would prefer an accelerated scan per predicate. However, an interesting fact neglected by this approach is that the accumulated selectivity of the multi-column selection predicates (1.72 % for Q6) is below the 2 % threshold. Hence, an index structure would be favored if it could exploit the relation between all selection predicates of the query. Consequently, when considering multi-column selection predicates, we achieve the selectivity required to use an index structure instead of an accelerated scan.

In this paper, we examine the question: *How can we exploit the combined selectivity of multi-column selection predicates in order to speed up predicate evaluation?* As a solution for efficient multi-column selection predicate evaluation, we propose Elf, an index structure that is able to exploit the relation between data of several columns. Using Elf results in performance benefits from several factors up to two orders of magnitude in comparison to accelerated scans, e.g., a scan using *single instruction multiple data* (SIMD). About factor 18 can be achieved for Q6 on a Lineitem table of scale factor $s = 100$, as visible in Fig. 1(c).

Elf is a tree structure combining prefix-redundancy elimination with an optimized memory layout explicitly designed for efficient main-memory access. Since the upper levels represent the paths to a lot of data, we use a memory layout that resembles that of a column store. This layout allows to prune the search space efficiently in the upper layers. Following the paths deeper to the leaves of the tree, the node entries are representing lesser and lesser data. Thus, it makes sense to switch to a memory layout that resembles a row store, because a row store is more efficient when accessing several columns of one tuple.

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.118

Furthermore, our approach features a fixed search path as each level belongs to one column and leads to a compression of the original data due to the prefix-redundancy elimination.

In particular, we make the following contributions:

1) We introduce Elf, a novel main-memory index structure for efficient multi-column selection predicate evaluation.
2) We develop improvements for our conceptual design to address deteriorations of our tree-based structure additionally enhancing its performance.
3) We conduct a comprehensive evaluation including a micro benchmark and multi-column selection predicate queries from the TPC-H benchmark showing the benefits and limitations of our approach in comparison to state-of-the-art approaches (e.g., BitWeaving [6] or Sorted Projection [7]) and a sequential scan using SIMD.
4) We show that the assumed selectivity threshold from Das et al. [5] does not hold for Elf – instead we can beat accelerated scans for mono-column selection predicates and for selectivities of up to 18% instead of 2%.

The remainder of the paper is organized as follows. Section II starts with a definition of the problem of evaluating multi-column selection predicates as well as a description of redundancy elimination. In Section III, we explain details of the implementation and optimization of our novel approach named Elf for main-memory environments. In Section IV, we extensively evaluate Elf's performance against well-known state-of-the-art competitors. In Section V, we briefly discuss related approaches and, finally, we summarize our insights in Section VI.

II. Preliminaries

In this section, we explain our use case, which is the evaluation of multi-column selection predicates. Furthermore, we present the concept of prefix-redundancy elimination, a key optimization of Elf.

A. Multi-column selection predicates

A multi-column selection predicate is defined for a set of columns C of a table T with $C \subseteq T$ and $|C| > 1$. For each column $col \in C$, there is one of the following basic predicates given: $=, <, >, \leq, \geq,$ BETWEEN. Column data and constants in the predicate are numeric values either by definition of the schema or due to order preserving dictionary encoding [8], [9]. For the remainder of the paper, we assume the latter case. As a result, the predicate defines a (different) window on each col and we can transform these predicates into one notation and treat them in a uniform manner as defined in Table I. For example, $col = x$, where x is a scalar value within this column, is translated to the window $[x, x]$, where x indicates the lower and upper boundaries and both are included in the window. By contrast, $col < x$ defines a window where the lower boundary is the domain minimum $(\min)$ of this column and x defines the first value that is not included in the window. Notably, it is possible to express $\neq$ as two windows.

A multi-column selection predicate result R_{mcsp} is a position list containing the references of all tuples (*Refs*), which can be used for subsequent operations like joins.

Definition II.1 (Result position list: R_{mcsp})**.** *Let Ref_i denote the tuple identifier of the i^{th} tuple (t_i) in the data set. Moreover, let $SAT_{mcsp}(Ref_i)$ be a Boolean function that is true, iff all attribute values of t_i for all columns are defined in the window by query mcsp. Then, R_{mcsp} is a list of identifiers such that*

$$Ref_i \in R_{mcsp} \Leftrightarrow SAT_{mcsp}(Ref_i) = true.$$

Predicate	Window
$= x$	$[x, x]$
$< x$	$[\min, x) \equiv [\min, x-1]$
$\leq x$	$[\min, x]$
$> x$	$(x, \max] \equiv [x+1, \max]$
$\geq x$	$[x, \max]$
$\leq x$ and $\geq y$ with $x \leq y$ (BETWEEN)	$[x, y]$

TABLE I. Columnar selection predicate translation

The basic challenge of multi-column selection predicates is that the selectivity of the overall query is often small, but the selectivity for each column is high enough that a database system would decide to use a scan for all columns. Thus, we cannot use only one column that dominates the query and use traditional indexes, like B-Trees, and then perform index lookups for the remaining tuple identifiers on the other columns. As a result, most used approaches are optimized column scans that exploit the full speed of the processing unit due to the cache-conscious columnar table layout [6], [10].

B. Prefix-redundancy elimination

An interesting concept that has been observed by Sismanis et al. is prefix-redundancy elimination [11]. Prefix redundancies occur whenever two or more dimension keys share a common prefix. This is visible in the example data of Table II, where tuple $\mathtt{T}_1$ and $\mathtt{T}_2$ share the same value in the first dimension. We formalize this term in Definition II.2.

Definition II.2 (Prefix-redundancy)**.** *Let t_a and t_b be two tuples over the same schema having n columns. Let Π denote an ordering of all n columns and let $t[1]$ be the first and $t[i]$ be the i^{th} attribute value of some tuple t according to Π. Then, we observe a prefix-redundancy regarding Π in case $\exists k$ such that $\forall i \leq k$ the attribute values of both tuples are equal, i.e. $t_a[i] = t_b[i]$ holds, for some k with $1 \leq k \leq n$. In this context, the longest common path is the largest value k_{max} for that we observe a prefix redundancy between two tuples.*

In [11], the authors use this observation to design a data structure that stores a highly compressed version of the cube operator. These high compression rates come mainly from avoiding redundancies of the cube entries. Nevertheless, these redundancies can also be found when considering multiple columns in a multi-column selection predicate evaluation scenario.

III. Elf index structure

Based on the insights from Section II, we design a novel index structure for order-preserving dictionary-compressed data or numeric data. The new index structure, called Elf, is optimized for executing multi-column selection predicates in main-memory systems. In the following, we first explain the Elf's basic design and the underlying memory layout. Then, we introduce additional optimizations to counter deteriorations due to sparsely populated subspaces and provide algorithms for searching, building, and maintenance. Finally, we determine a theoretical upper bound for its storage size and introduce our heuristic for the column order.

A. Conceptual design

In the following, we explain the basic design with the help of the example data in Table II. The data set shows four columns to be indexed and a tuple identifier (TID) that uniquely identifies each row (e.g., the row id in a column store).

C_1	C_2	C_3	C_4	...	*TID*
0	1	0	1	...	T_1
0	2	0	0	...	T_2
1	0	1	0	...	T_3

TABLE II. RUNNING EXAMPLE DATA

In Fig. 2, we depict the resulting Elf for the four indexed columns of the example data from Table II. The Elf tree structure maps distinct values of one column to DimensionLists at a specific level in the tree. In the first column, there are two distinct values, 0 and 1. Thus, the first DimensionList, $L_{(1)}$, contains two entries and one pointer for each entry. The respective pointer points to the beginning of the respective DimensionList of the second column, $L_{(2)}$ and $L_{(3)}$. Note, as the first two points share the same value in the first column, we observe a prefix redundancy elimination. In the second column, we cannot eliminate any prefix redundancy, as all attribute combinations in this column are unique. As a result, the third column contains three DimensionLists: $L_{(4)}$, $L_{(5)}$, and $L_{(6)}$. In the final DimensionList, the structure of the entries changes. While in an intermediate DimensionList, an entry consists of a value and a pointer, the pointer in the final dimension is interpreted as a tuple identifier (TID).

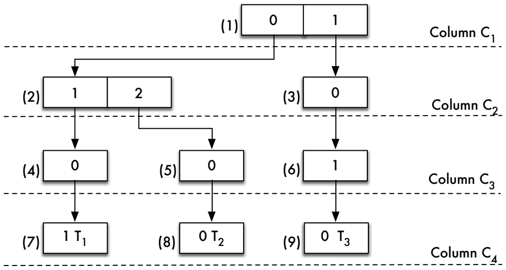


Fig. 2. Elf tree structure using prefix-redundancy elimination.

The conceptual Elf structure is designed from the idea of prefix-redundancy elimination in Section II-B and the properties of multi-column selection predicates. To this end, it features the following properties to accelerate multi-column selection predicates on the conceptual level:

Prefix-redundancy elimination: Attribute values are mainly clustered, appear repeatedly, and share the same prefix. Thus, Elf exploits this redundancy as each distinct value per prefix exists only once in a DimensionList to reduce the amount of stored and queried data.

Ordered node elements: Each DimensionList is an ordered list of entries. This property is beneficial for equality or range predicates, because we can stop the search in a list if the current value is bigger than the searched constant/range.

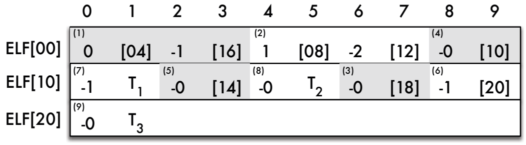
Fixed depth: Since, a column of a table corresponds to a level in the Elf, for a table with n columns, we have to descend at most n nodes to find the corresponding *TID*. This sets an upper bound on the search cost that does not depend on the amount of stored tuples, but mostly on the amount of used columns.

In summary, our index structure is a bushy tree structure of a fixed height resulting in stable search paths that allows for efficient multi-column selection predicate evaluation on a conceptual level. To further optimize such queries, we also need to optimize the memory layout of the Elf approach.

B. Improving Elf's memory layout

The straight-forward implementation of Elf is similar to data structures used in other tree-based index structures. However, this creates an OLTP-optimized version of the Elf, which we call InsertElf. To enhance OLAP query performance, we use an explicit memory layout, meaning that Elf is linearized into an array of integer values. For simplicity of explanation, we assume that column values and pointers within Elf are 64-bit integer values. However, our approach is not restricted to this data type. Thus, we can also use 64 bits for pointers and 32 bits for values, which is the most common case.

1) Mapping DimensionLists to arrays: To store the node entries – in the following named DimensionElements – of Elf, we use two integers. Since we expect the largest performance impact for scanning these potentially long DimensionLists, our first design principle is *adjacency* of the DimensionElements of one DimensionList, which leads to a preorder traversal during linearization. To illustrate this, we depict the linearized Elf from Fig. 2 in Fig. 3. The first DimensionList, $L_{(1)}$, starts at position 0 and has two DimensionElements: $E_{(1)}$, with the value 0 and the pointer 04 (depicted with brackets around it), and $E_{(2)}$, with the value 1 and the pointer 16 (the negativity of the value 1 marks the end of the list and is explained in the next subsection). For explanatory reasons, we highlight DimensionLists with alternating colors.



	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) 0	[04]	-1	[16]	(2) 1	[08]	-2	[12]	(4) -0	[10]
ELF[10]	(7) -1	T_1	(5) -0	[14]	(8) -0	T_2	(3) -0	[18]	(6) -1	[20]
ELF[20]	(9) -0	T_3								

Fig. 3. Memory layout as an array of 64-bit integers

The pointers in the first list indicate that the DimensionLists in the second column, $L_{(2)}$ and $L_{(3)}$ (cf. Fig. 2), start at offset 04 and 16, respectively. This mechanism works for any subsequent DimensionList analogously, except for those in the final column (C_4). In the final column, the second part of a DimensionElement is not a pointer within the Elf array, but a *TID*, which we encode as an integer as well. The order of DimensionLists is defined to support a depth-first search with expected low hit rates within the DimensionLists. To this end, we first store a complete DimensionList and then recursively store the remaining lists starting at the first element. We repeat this procedure until we reach the final column.

2) Implicit length control of arrays: The second design principle is size reduction. To this end, we store only values and pointers, but not the size of the DimensionLists. To indicate the end of such a list, we utilize the most significant bit (MSB) of the value. Thus, whenever we encounter a negative value[1], we know we reached the end of a list (e.g., the DimensionElement at offset 2). Note, in the final column, we also mark the end of the list by setting the most significant bit, allowing to store duplicates as well.

C. Storage optimizations

Considering the structure of Elf depicted in Fig. 2, we can further optimize two conceptual inefficiencies: (1) since the first list contains all possible values of the first column, this list can become very large, resulting in an unnecessary performance overhead and (2) the deeper we descend in Elf, the sparser the nodes get, which results in a linked-list-like structure in contrast to the preferred bushy tree structure. For both inefficiencies, we introduce as solutions: a hash map for the first column and MonoLists for single-element lists.

1) Hash map to deal with the first DimensionList: The first DimensionList contains all distinct values of the first column, including pointers that indicate where the next list starts. As a result, we have to sequentially scan all these values until we find the upper boundary of the window defined on the first column. This, however, results in a major bottleneck and renders the approach sensitive to the number of inserted tuples instead of the number of columns. However, due to the applied compression scheme and prefix redundancy elimination, the first DimensionList has three properties that allow us to store only the pointers in the form of a perfect *hash map*[2]. As keys of the *hash map*, the dimension values are used and as the *hash map* values, the pointer to the referenced DimensionList of the second column is used. We now discuss the three properties of the values in the first DimensionList that lead to a perfect hash-map property.

Uniqueness. Due to prefix redundancy elimination within Elf, all dimension values in every list are unique.

Denseness. Due to the order preserving dictionary compression of the data, all integer values between 0 and the maximum value $\max_0$ of that column exist.

Ordering. By definition, all values within a DimensionList are ordered.

As a result, the first DimensionList contains *every* integer value of $[0, \max_0]$, which are stored in an ordered manner. We depict the resulting Elf for the first column with the value range $[0, 7]$ in Fig. 4 (upper part). The primary observation is that we can compute the position of the pointer to the next list by simply multiplying the value by 2 and incrementing the result. Consequently, we could also omit the values and only store the pointers, as shown in the lower part of the figure. Hence, we can directly use the values as keys to the pointers of the first column like in a hash map. In order to determine the start and end points of a query window on the first column, we take the query window and identify the respected pointers. This way, we remove the deterioration of the first DimensionList and furthermore, require only half of the storage space for it. This also works in case the data is not dense. Then, we use a special pointer directly indicating that for this value there is no data, effectively being a null pointer.

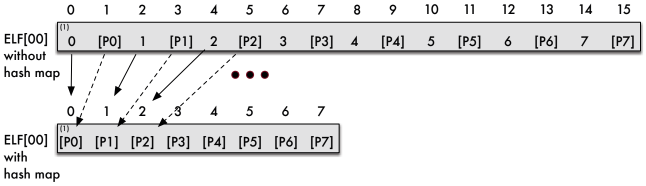


Fig. 4. Hash-map property of the first DimensionList

2) MonoList: One-element list elimination: A main challenge of our data structure is that the lists get shorter the further the search descends into an Elf. Notably, there is a level where only one-element lists exist. That means, there are no more prefix redundancies that can be exploited. We display this issue for the TPC-H Lineitem table of scale factor 100 with all 15 attributes resulting in a 15-level Elf in Fig. 5. The plot shows that at dimension 11 the prefix of each data item has become unique and each data item is now represented with its own path in the Elf. This leads to one linked list per data item, where each entry is a DimensionList with only one entry. The result of those one-element lists is that the remaining column values of each data item are scattered over the main memory. Additionally, we need to store pointers to these values, although branching is not necessary anymore. This phenomenon destroys the caching performance and unnecessarily increases the overall size of Elf. To overcome this deterioration, we introduce MonoLists. The basic idea of MonoLists is that, if there is no prefix redundancy, the remaining column values of this tuple are stored adjacent to each other (similar to a row-store) to avoid jumps across the main memory.

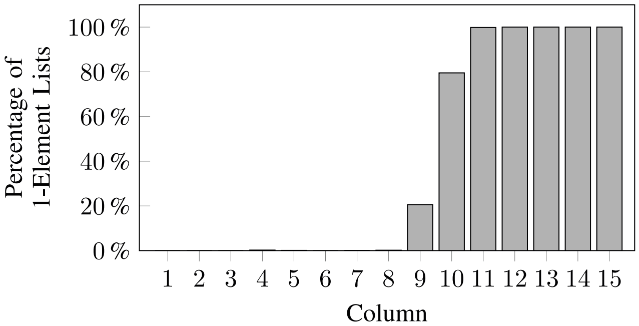


Fig. 5. Percentage of 1-element lists per dimension for the TPC-H Lineitem table with scale factor 100

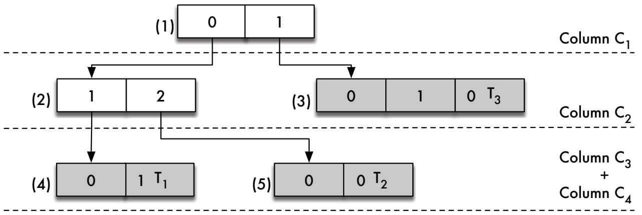


Fig. 6. Elf with MonoLists (visualized as gray DimensionLists)

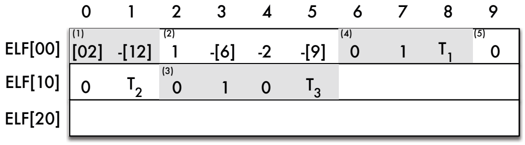
In Fig. 6, we depict the resulting Elf with MonoLists shown in gray and in Fig. 7 the respective memory layout. Note that a MonoList can start at different dimensions and

[1] This visualization is not correct according to the definition of the two's complement, but allows us to visualize the end of the list while displaying the original value. In our implementation, we use bit masks to set, unset, and test the most significant bit to determine whether we reached the end of a list.

[2] With *perfect hash map*, we mean that we can represent the *hash map* as a *dense* array, where the keys represent the array positions.

thus, this optimization totally removes the deterioration of one-element lists. To indicate that there is a `MonoList` in the next column, we utilize the most significant bit of the pointer of the respective `DimensionElement` in the same way as we mark the end of a `DimensionList`. Thus, we depict such a pointer in the same way, by using a minus in front of the pointer in Fig. 7. In the example, there are two `MonoLists` for C_3 and C_4 and a third one covering C_2, C_3, and C_4 for T_3.

In Fig. 7, we depict the optimized storage layout of the Elf for the example data from Table II using the `MonoList` and the hash map optimization. In comparison to the initial layout in Fig. 3, we observe a decrease in storage consumption and better adjacency of values in later columns. We give more insights into worst case storage consumption in the Section III-F.



	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) [02]	-[12]	(2) 1	-[6]	-2	-[9]	(4) 0	1	T_1	(5) 0
ELF[10]	0	T_2	(3) 0	1	0	T_3				
ELF[20]										

Fig. 7. Final memory layout of the Elf approach

D. Search algorithm

In the following, we present the algorithm to evaluate a multi-column selection predicate within Elf, based on our definition from Section II-A. The algorithm mainly consists of two functions.

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

Algorithm 1: Search multi-column selection predicate

The first function `SearchMCSP`, depicted in Algorithm 1, is executed once in order to evaluate a multi-column selection predicate. It returns a list `L` of *TIDs* for each tuple in accordance with the multi-column selection predicate. Two arrays define the multi-column selection predicate containing the lower and upper boundaries of the (hyper) rectangle. Moreover, this function evaluates the first `DimensionList` exploiting its hash-map property (Line 3-5), in case C_1 is part of the multi-column selection predicate. Otherwise, in case of a wildcard for this column, the boundaries for evaluation are set to 0 and maximum of Column C_1. We have to check for each value whether the next `DimensionList` is a `MonoList`. Based on this check, we either call the function to evaluate a `MonoList` or a normal `DimensionList` (Line 10-14). Note, the evaluation of a `MonoList` (`SearchML`) is straightforward, as the remaining values are located beside each other in main memory including a *TID* after the values. Hence, we do not depict this function.

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

Algorithm 2: Scan a `DimensionList` within an Elf

The second function `SearchDimList`, depicted in Algorithm 2, evaluates a predicate on a single `DimensionList`. The function has two more input parameters besides the lower and upper boundaries: It also needs the start offset of the current `DimensionList` within the Elf (`startlist`) and the current column (`col`). The start offset directly marks the position of the first (and smallest) value in that `DimensionList` (Line 3). In case there is a predicate defined on this column, we start scanning the single values until we either reach the end of the list (Line 17) or we find a first value that is larger than the upper boundary of the query window. Remember that the values are ordered, which allows us to abort the evaluation of that particular list. Whenever we find a value within the predicate boundaries, we propagate the evaluation of the multi-column selection predicate to the child `DimensionList` (Line 5-12). This results in a depth-first search, because we evaluate the child `DimensionList` before evaluating the next value by incrementing the `position` (Line 16). We decided on a depth-first search to make use of the program stack and, as the first column is handled by `SearchMCSP`, we benefit from the curse of dimensionality as the sparsity of the created spaces results in relatively low hit rates. Thus, on average, we are able to scan large parts of a `DimensionList` located in a small cache window without propagation to the next column. Consequently, the Elf search algorithm is optimized for low selectivity-rate workloads, as it is common for tree-based structures.

E. Building and maintaining an Elf

Building and maintaining an Elf is done in two different ways. Due to space limitation, we do not explain the full algorithms, but outline the general ideas of the build algorithm and the insert, update, and delete operations on Elf. For further details we refer to [12].

The initial build of Elf is executed as a *bulk load*, where all data of the table is read to create the initial Elf with its explicit memory layout. The build algorithm consists of a step-wise multi-dimensional sort which is paired with a build of all `DimensionLists` of the currently sorted dimension using a preorder linearization. Notably, after sorting the first dimension we can divide the index creation for each sub tree into independent tasks allowing for parallel execution.

Due to Elf's explicit memory layout, maintenance (i.e., insert, update, and delete) is not trivial, but still manageable. Since Elf is designed for analytical scenarios, supporting periodic inserts of new data, such as weekly or daily inserts, are most important for us. Our solution consists of two parts. First, new data is collected in the auxiliary data structure InsertElf, which has the same conceptual design as a normal Elf without the explicit memory layout and `MonoLists`. That is, the `DimensionLists` are arrays in that we can insert easily. The general idea is similar to delta stores in column-oriented databases or to the ideas by Zhang et al. [13]: there is one write-optimized InsertElf and one read-optimized Elf. Whenever we have reached a specific threshold of changes, it will become necessary to transfer the data from the InsertElf to the Elf, which is a merge of both structures. The merge algorithm works similar to merging two sorted lists of elements as we can exploit the (same) total order in both, InsertElf and Elf. Hence, if as much data has been inserted into the InsertElf that a merge becomes necessary, we can efficiently combine both structures with a complexity of O(Elf$_{size}$+InsertElf$_{size}$).

For deletion, we perform a lookup for the data item we want to delete and store for each level the pointers when jumping to a new `DimensionList` or a marker in case of a `MonoList`. In case, we delete a duplicate data item, we just remove the *TID* in the list of *TID*s. Otherwise, we need to invalidate the path that only belongs to the data item we want to delete. Assume, we want to delete data item T_2 from Fig. 6. We know that in `DimensionList` (2) a `MonoList` starts and thus invalidate the pointer to that `MonoList` using a pre-defined error value.

Finally, updates are rare for analytical workloads, but nevertheless possible within Elf. Generally, there is a large amount of `MonoLists` (cf. Section III-C2). Updating a value within a `MonoList` does not result in any problem, as we just have to write the new value to the correct position. This is possible as all values have the same size due to the applied dictionary compression. Otherwise, an update is composed of a delete and an insert as described above.

F. Worst case storage consumption

Storage consumption remains an important issue due to limited main-memory capacities and better cache utilization for smaller storage and index structures. We examine worst case storage consumption to give an upper limit for our novel structure to show its potential. For Elf, we can construct a worst case scenario analytically. In the first `DimensionList`, worst case means that there are only unique keys. Thus, there is no prefix redundancy elimination resulting in k pointers to be stored, where k is the number of points in the data set. Notably, this does not cause any overhead compared to the normal storage of values, because of the hash map property. For the other columns, we have two cases:

1) We can perform a prefix reduction of the column value: Then, we store the pointer to the next level and one value representing m values, reducing the storage consumption to $2/m$.
2) We find a `MonoList`: Then, we need to store the attribute values and the *TID* of the data item.

Worst case means that for each point, we immediately start a `MonoList` after the first column, because with a prefix reduction, we achieve a better storage consumption[3]. The worst case leads to storage of *one* additional value per data item. The additional value is the *TID*, which would not be stored in the original row or column store representation as it is encoded implicitly based on the offset from the beginning of the array.

As a result, the maximum storage overhead per data item depends on the number of indexed columns n of the data set and decreases with an increasing amount of columns (cf. Table III). It is computed as follows: overhead$(n) = (n+1)/n$.

Number of columns	1	2	3	4	5	6
Storage overhead	2.00	1.50	1.33	1.25	1.20	1.17

TABLE III. UPPER BOUND STORAGE OVERHEAD

As this worst case is very unlikely, we expect even light compression rates for most data sets. Hence, the actual storage size of Elf is an analysis target in our evaluation section.

G. Selection of the column order

One important aspect of building an Elf is the order of columns, because it influences search time as well as storage consumption. To this end, we propose a simple heuristic that is used to determine a column order. Currently, we work on a fully-fledged cost model and first results are highly promising [14].

Due to the design of Elf, the first column should be the most commonly used in the queries, e.g., a time dimension. The following columns are sorted in ascending order of their usage in queries and cardinality. Due to this heuristic and the prefix reduction in the first columns, the data space is fast divided into sparse regions. Hence, we benefit from an early pruning of the search space.

IV. EMPIRICAL EVALUATION

We now conduct several experiments to gain insights into the benefits and drawbacks of Elf. We start with a micro benchmark that systematically evaluates the influence of parameters such as query selectivity and queried columns on the response time. In this evaluation, we are interested in the break-even points regarding selectivity that indicate when a SIMD-sequential scan becomes faster than our approach. To this end, we use an artificial query load defined on the TPC-H schema with scale factor 100. Another micro benchmark considers our `MonoList` optimization and shows its benefits considering the storage consumption of the resulting Elf.

In further experiments, we evaluate how far our artificial results of the first experiments can be transferred to real-world selection predicates, such as those from the TPC-H benchmark queries. As competitors, we select three state-of-the-art approaches, BitWeaving/V [6], Column Imprint [10],

[3] Nodes with two elements lead to the same storage consumption as a `MonoList` due to the pointers. Both cases are equivalent for our worst case consideration.

and Sorted Projection [7]. In addition, we compare our approach to a columnar scan and an optimized SIMD-accelerated version as a good baseline. We select the kd-Tree [15] as a well-known classical multi-dimensional index structure with axis-parallel splits natively supporting multi-column selection predicates. As every indexing technique, we trade query performance for initial build time [16]. Hence, we evaluate the tradeoffs of Elf and its competitors regarding build times.

To ensure a valid comparison, all approaches are implemented in C++ and tuned to an equal extent to allow for a fair comparison. The code of our evaluation is provided on the project website[4]. The result of a multi-column selection predicate evaluation, is a position list as defined in Definition II.1. We perform our experiments using a single-threaded execution for all index structures on an Intel Xeon E5 2609 v2 (Ivy-Bridge architecture) with 2.6 GHz clock frequency, 10 MB L3 cache, and 256 GB RAM. Our SIMD optimizations are implemented using SSE 4.2. In our evaluation, we present the response time for the selection predicates of each considered TPC-H query. For statistical soundness, we repeated every measurement 1,000 times and present the median as robust averages.

A. Experiment 1: Micro benchmark

In this experiment, we are interested in how well our approach scales for different selectivities and for different amounts of queried columns in Elf. Moreover, we are interested in the break-even point when an optimized scan becomes faster than our novel approach. So far, the break-even point for most tree-based indexes in main-memory environments is stated to be around 1 or 2 percent [5]. Finally, we want to gain insights on the limiting factor: whether it is the overall selectivity or the column order in an Elf.

To this end, we conduct several experiments on the `Lineitem`[5] table with scale factor 100. We select this table to avoid biasing our results by, for instance, using uniformly distributed synthetic data instead. To have a fair comparison between the SIMD scan and Elf, we assume that the whole table has to be indexed by Elf (e.g., because our workload includes selections on all columns). Notably, smaller Elfs on a reduced set of columns would further boost the performance, but for this experiment we want to show a worst case scenario.

Altogether, we measure the response times for the combinations of selectivity $\sigma \in [0.0003\,\%, 50\,\%]$ and last queried column: $l \in \{0, 1, 2, 3, 4\}$. In this context, a selectivity of $0.5\,\%$ means that $0.5\,\%$ of the tuples of the `Lineitem` table are retrieved. If last queried column is 3, then there is a predicate defined on the columns 0, 1, 2, 3. For instance, we conduct one measurement for the parameter combination $(\sigma = 1\,\%, l = 1)$. In this case, the selection predicate is defined on the first column `l_shipdate` and the second one `l_discount`. The associated SQL query is:

```
SELECT l_shipdate, l_discount FROM lineitem
WHERE l_shipdate BETWEEN c1 AND c2
AND l_discount BETWEEN c3 AND c4
```

We substitute the values for constants like $c1$ with appropriate values to achieve the desired combined selectivity, meaning that the result contains $1\,\%$ of the data. Note, we define multiple windows having the same selectivity and repeat each of the parameter configurations 100 times to achieve reliable results.

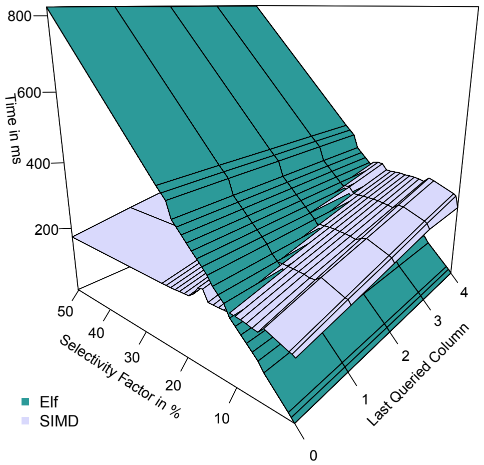


Fig. 8. Comparison of Elf (green) and SIMD scan (blue)

Result interpretation: In Fig. 8, we depict the mean response time for every evaluated parameter combination of Elf (green plane) and the corresponding response time of a SIMD sequential scan (blue plane).

As expected, we observe that the response times of the SIMD sequential scan are very stable for varying selectivities. For example, for the queries on two columns, the SIMD scan requires $177\,ms$ for the lowest selectivity and $202\,ms$ for the highest selectivity, which takes only $14\,\%$ more time. The small differences result from the overhead for managing larger results. In contrast, the number of searched columns has a bigger impact on the runtime of the SIMD sequential scan. For a selectivity of $10\,\%$, the SIMD sequential scan takes $182\,ms$ for one column and $289\,ms$ for five columns, leading to $58\,\%$ increased response time.

For Elf, we observe a strong dependency of response time and selectivity. As this is expected, the interesting insights are the following:

Linear correlation of response time and selectivity:
The important point about this insight is that we have a predictable increase in response time. This property allows to define an upper border for the response time of a query.

Minor performance impact of column position:
This insight is to some extent surprising, as we expected that the number of queried columns have a stronger influence. However, a deeper investigation of the worst case scenario strengthens this phenomenon: for instance, assume there is a query that *only* defines a selection predicate on the third column. In Elf, representing the `Lineitem` table, we have a cardinality of 2,526 values for the first dimension and 11 for the second one. This means that, in the worst case, assuming that all value combinations exist, we have to execute 27,786 additional jumps in main memory due to wild cards in the first two columns. Nevertheless, this number is negligible, if we

[4] www.elf.ovgu.de

[5] This table takes about 72 GB of memory. Note, for a fair comparison, we only use the order-preserving dictionary encoded data (with a size of ca. 33.5 GB) for all experiments and all competitors.

compare it to the total number of tuples in the table, which is about 600 million.

Break-even point at higher selectivity than expected: The break-even point is by one magnitude larger (10 % to 20 % in contrast to 1 % to 2 %) than postulated in the related literature. This fact reveals that using Elf does not only result in performance gains over the baseline, but this also holds for a selectivity exceeding 10 %, which is an enormous value, that to the best of our knowledge has not been observed so far. Furthermore, the more columns we query, the higher the acceptable selectivity. While for one queried column the break-even point is at 11 % selectivity, it is at 18 % selectivity for five queried columns.

The results of our micro benchmark indicate that Elf is superior to the SIMD sequential scan for predicates on several columns and a low selectivity (i.e., small result sets). However, the comparison does not take into account more elaborate index structures such as BitWeaving with its ability to skip a search early. Furthermore, our predicates are artificially generated with selections on a prefix of all columns in an Elf. A deeper insight into this brings Experiment 3 with real-world workloads from the TPC-H benchmark.

B. Experiment 2: Impact of MonoLists on Storage Consumption

Although main memory capacities increase rapidly, efficient memory utilization remains important, because it is shared between all data structures (e.g., hash tables) of the database system. In this micro-benchmark, we want to examine, first, whether our worst case storage boundaries for Elf from Section III-F hold. This upper bound, however, is quite pessimistic. Thus, we are interested in empirical numbers of the storage overhead for the TPC-H `Lineitem` table with scale factor 100. Second, we are interested in how far this result is influenced by the usage of `MonoLists`, because they are an essential optimization for our Elf for multi-dimensional data.

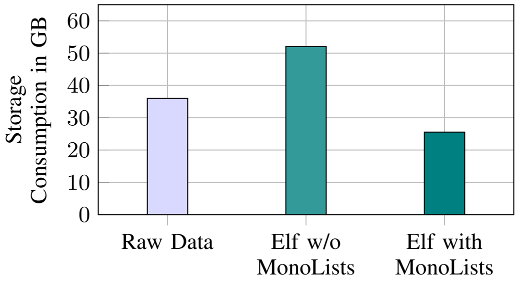


Fig. 9. Storage consumption for `Lineitem` table

In Fig. 9, we display the storage consumption of the raw data, an Elf without using `MonoLists` and an Elf with `MonoList` optimization. As visualized, the raw data consumes about 36 GB, while the Elf without `MonoList` optimization consumes 52.03 GB and the fully-optimized Elf consumes 25.55 GB of RAM. This is a remarkable result, because the Elf is not only taking only 70 % of the original raw data storage space, but we could also clearly improve a severe deterioration of the conceptual Elf. In fact, the optimized Elf consumes only half of the memory that the Elf without `MonoLists` consumes. This can be explained by the high number of `MonoLists` especially in deeper tree levels (c.f. Fig. 5). For instance, at level 9, we encounter around 114 million `MonoLists`, that save around 6 GB of pointers. Hence, the `MonoList` optimization is worth using for sparsely populated spaces, because it does not only save space, but also reduces the amount of cache lines that have to be fetched to visit the *TIDs*. Notably, Elf stores the whole data set, which means that we do not need to store the data additionally. Thus, we can even save space when using Elf as a storage structure, because all information is directly available within the Elf.

C. Experiment 3: TPC-H queries and data

In the following, we conduct an experiment using selection predicates from queries of the TPC-H benchmark [17], which our competitors performed in a similar fashion [6], [10][6]. The main difference to our first experiment is that we do not use synthetic query predicates, but predicates that reflect real-world information needs common for analytical databases. Consequently, the results of this experiment are better suited to draw conclusions regarding real-world workloads. We select queries having a multi-column selection predicate and additional ones having a mono-column selection predicate, as summarized in Table IV. Notably, the last column states where the columns with a predicate are located within the Elf. The first column number is 0 to emphasize that we can exploit the hash-map property for this column. The column Col_{Elf} is also important for Sorted Projections, because we create one Sorted Projection per distinct prefix.

The queries having a mono-column selection predicate are selected to explore the general applicability (and limitations) of Elf for real-world workloads. To find answers regarding this question, we select Query `Q1, Q10,` and `Q14`. The predicates for `Q1` and `Q14` are defined on the first column. This means that the main cost factor for this query is traversing cold data of Elf in order to determine the respective *TIDs*. We choose these two queries, because their selectivity differs significantly. By contrast, the predicate for `Q10` is defined on the fifth column, which is a different scenario than in our micro benchmark, where we only queried a prefix of the column order. In general, we expect Elf performance to vary significantly across the three queries, as they represent cases Elf is not designed for.

	example σ in %	predicate columns	Col_{Elf}
Q1	98.0	l_date	0
Q10	24.68	l_returnflag	4
Q14	1.3	l_date	0
Q6	1.72	l_date, l_discount, l_quantity	{0,1,2}
LQ19	1.4	l_quantity, l_shipinstr, l_shipmode	{2,5,6}
Q17	0.099	p_brand, p_container	{1,2}
PQ19	0.083	p_brand, p_container, p_quantity	{1,2,3}

TABLE IV. QUERY DETAILS FOR MONO AND MULTI-COLUMN SELECTIONS

Since the accelerated scans are sensitive to the number of queried columns, we also include several multi-column selection predicate queries on different tables. For `Q19`, we have two multi-column selection predicates on two different tables. The first is defined on the `Lineitem` table (as indicated by the *L* prefix) and the second is defined on the `Part` table. We refer

[6] Other benchmarks, e.g., Starschema [18] or TPC-DS [19], could be used in a similar fashion. However, due to the common use of the TPC-H benchmark, we restrict our evaluation to this one.

to them as LQ19 and PQ19, respectively. Query Q6 works on the Lineitem table and the predicates are defined on the first three columns. By contrast, Q17 addresses the Part table and the predicate is defined on the second and third column. Thus, we cannot exploit the hash-map property here. In general, we expect good results for all of these queries using Elf.

In the following, we depict the values for the selection predicates of the TPC-H benchmark with its order-preserving dictionary-compressed data of scale factor 100. We generate 1,000 random predicates according to the TPC-H specification and compute the median response time to assure robust measurements. Similar to our micro benchmarks, we include an Elf that indexes the whole TPC-H table as a baseline. However, since the maximum column index for all queries in Table IV is 6 (i.e., the seventh column), we also evaluate an Elf that only indexes the first 7 columns. We refer to the reduced Elf as Elf$_7$ in the following, named by the number of columns (incl. *TID*) it contains. Note, we omit a detailed evaluation of the update-optimized InsertElf, because its storage consumption would exceed our available RAM (cf. Experiment 2) as it omits MonoLists for better update performance. Moreover, downgraded experiments show that its run times are higher by a factor of 50 when indexing all columns and by a factor of 3 when indexing the first seven columns only.

1) Mono-column selection predicate queries: In Fig. 10, we depict the results for the mono-column selection predicates in a logarithmic plot. Overall, we observe high differences in performance of Elf regarding the three queries in comparison to the competitors. For Q1 returning 98 % of the tuples of the Lineitem table, Elf is clearly outperformed by all accelerated scans. Even Elf$_7$ is slower than a columnar sequential scan, although it can outperform Sorted Projection and kd-Tree. By contrast, for Q10, where the selection column is only the fifth column (remember that we start counting columns from 0), using the Elf$_7$ results in a response time comparable to both state-of-the-art approaches. However, the Elf containing all columns is 79 % slower than a columnar sequential scan, while the difference between the state-of-the-art approaches and the baseline (the columnar sequential scan) is quite small. For instance, the columnar sequential scan requires 238 ms whereas the Column Imprint requires 161 ms. Thus, the performance gains for accelerated scans are around one third. Reasons for this behavior are the high selectivity of Q1, the moderate selectivity of Q10 and the fact that the selection predicate in Q10 is at the fifth dimension. This forces Elf to follow a majority of paths. Therefore, we cannot and do not intend to compete with optimized full-table scans in this scenario. Notably, Elf$_7$ can even slightly outperform all other approaches for query Q10.

In contrast to Query Q1 and Q10, our results for Query Q14 indicate that our approach results in a clear performance gain for both variants of Elf. The response time of the SIMD sequential scan is 124 ms. By contrast, the response time of the Elf is 19 ms and the Elf$_7$ requires 6.1 ms. Consequently, our results show a performance gain of more than a factor of 6 for the Elf and about a factor of 20 for the Elf$_7$. From our point of view, this is a remarkable result, because our approach is designed and optimized for multi-column selection predicates. However, in the case of Query Q14, we benefit from the hash-map property of Elf and the fact that the selection column is the first level instead of the fourth level, as in Query Q10.

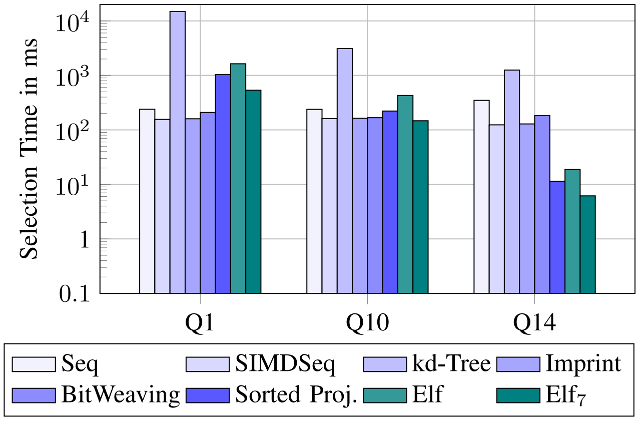


Fig. 10. Query response times for mono-column TPC-H queries (s = 100)

Notably, Sorted Projection performs better than Elf but worse than Elf$_7$ for all mono-column selection predicates. The benefit of Elf is that the prefix redundancy elimination allows to touch less memory locations than the Sorted Projections, but skipping over the cold data diminishes this benefit. Thus, only the Elf$_7$ outperforms the Sorted Projections. Furthermore, we observe that the SIMD sequential scan implementation performs well, even beyond our expectations. For the first two queries, it is the fastest approach and for the third query the third fastest behind the Elf variants. Moreover, our results show that the SIMD sequential scan slightly outperforms state-of-the-art approaches. However, recent results show that BitWeaving is able to reach at least the same performance when using SIMD [20] and we expect Column Imprints to behave in a similar fashion. Finally, the two state-of-the-art approaches *always* outperform the baseline and the kd-Tree by far.

2) Multi-column selection predicate queries: In contrast to mono-column predicates, we observe Elf's superiority for all multi-column selection predicate queries in Fig. 11. In particular, Elf delivers the fastest response times for every query. Moreover, we observe a stable performance increase between a factor of 2 and 4 when using Elf$_7$ as compared to a full Elf. An in-depth analysis reveals that this correlates to the difference in size of both variants. However, the performance gain of our approach over the competitors varies widely. For the Lineitem selection predicates of Query Q19(LQ19), we observe the *smallest* performance gain compared to the next fastest accelerated scan. These are BitWeaving and the SIMD scan, requiring 526 ms and 521 ms, respectively. The speed-up factor of the Elf (161 ms) is 3.2 and considering the Elf$_7$ (66.5 ms) it is factor 7.9. By contrast, the largest performance gain is measured for the queries with the smallest result sizes, Q17 and PQ19 (cf. Table IV). It is in the order of *two magnitudes*.

For these queries, our results reveal that we can fully benefit from the properties of Elf. All multi-column selection predicates have in common that they have a low *accumulated* selectivity (< 2 %). This is one important reason why our approach outperforms all other competitors in this experiment. Furthermore, the accelerated scans are scanning each column separately, which does not scale for an increasing number of involved columns. Notably, it would not scale to use an optimized index structure per column (e.g., CSB-tree [21]), because the selectivity for one column is still much higher than

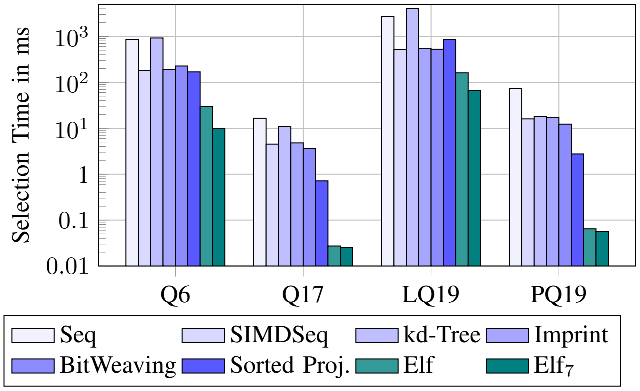


Fig. 11. Query response times for multi-column TPC-H queries (s = 100)

the accumulated selectivity of the whole query (cf. Fig. 1). In these selections, the only competitor that shows an improved performance compared to accelerated scans is Sorted Projection although it cannot beat Elf.

According to our results, the performance gain also depends on the column order. This is especially observable for `Q6` and `LQ19`, which have a similar selectivity, but the selection predicates are defined on different columns. In fact, to evaluate `LQ19`, we have to traverse the Elf until the third column (the first column with a predicate) in order to exclude further parts of the tree. This explains the different speed ups of both queries. Interestingly, the response times of `Q6` (multi-column) and `Q14` (mono-column), whose selectivities are similar, are comparable, indicating the consistency and stability of our approach.

The results of this experiment reveal that the major cost factor is the accumulated selectivity, as we achieve the largest speed ups for the queries with the lowest selectivity. This is consistent with the results from the mono-column selection predicates and our micro benchmark. The additional improvements using the Elf_7 also seem plausible as they directly correlate to the difference in size between Elf and Elf_7. Hence, determining the required columns is an important factor to fully exploit the potential of Elf.

For the other approaches, we observe that the state-of-the-art approaches result in large speed ups compared to the baseline. Moreover, BitWeaving clearly outperforms the SIMD scan for `Q17` and `PQ19` and delivers comparable results for `LQ19`. Only for `Q6`, we measure faster response times for the SIMD scan. Nevertheless, Sorted Projection outperforms all accelerated scans for the multi-column selection predicate queries, reaching speed ups of factor five to even one magnitude.

On a more abstract level, we observe that all approaches outperform the baseline, usually by several factors. This is consistent with results from the literature. The only exception is the remarkable performance of the SIMD sequential scan, which we explain by additional optimizations from SIMD-related publications [22], [23]. Moreover, this demonstrates that an efficient implementation for accelerated scans is vital and it also emphasizes the necessity for such approaches in main-memory systems in general. In addition to that, we observe a wide range of speed ups of most approaches compared to the baseline, suggesting that different application scenarios require different approaches as there does not seem to be a one-size-fits-all solution.

D. Experiment 4: Build times

The purpose of the build time examination is to evaluate whether a large build time is a counterargument for the applicability of Elf.

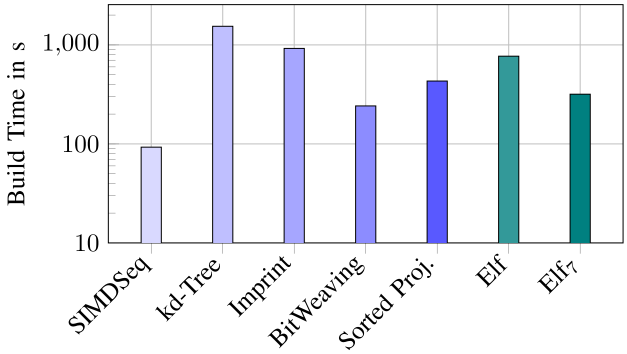


Fig. 12. Build time for `Lineitem` table of s = 100 TPC-H

In Fig. 12, we visualize the build times for the `Lineitem` table with scale factor 100. The build time of the SIMD sequential scan (92.72 s) is the time to allocate aligned memory and copy the respective values from a row-based representation into an aligned columnar layout in order to fully exploit SIMD benefits. In comparison to the other approaches this build time is negligible. We included this value to highlight that using SIMD always entails a certain overhead compared to normal sequential scans. In contrast to the build time of the SIMD sequential scan, all other approaches need at least several minutes up to half an hour for finishing the build. The next fastest approach is BitWeaving, with a build time of 241.91 s. Building the Sorted Projection approach requires 431.47 s. Thus, it is about two times faster than building the Elf (769.09 s). Therefore, we argue that build times are no counterargument for the applicability of our approach, especially as we reach a speed up of one order of magnitude for the query performance. For instance, in analytic environments and other scenarios that have low update frequencies, such an additional build time is acceptable. Interestingly, the Elf_7 is the third fastest approach. Note, Column Imprint and kd-Tree require more build time than Elf.

E. Result summary

Based on the results of all our experiments, we have empirically shown the superiority of our approach compared to several strong competitors. However, the build times are still a challenging factor for Elf, which currently makes our approach sensitive for applications with high update and insert rates.

In comparison to a SIMD sequential scan, Elf performs better for selectivities smaller than 11–18 % and the more columns are queried, the higher the acceptable selectivity. Moreover, we can show that our `MonoList` optimization reduces the storage overhead by 50 % compared to an Elf without `MonoList` and by 30 % compared to the raw data. Furthermore, we show that our approach performs best for low-selectivity workloads from non-synthetic queries such as those from the TPC-H benchmark regardless of the location of the queried column in the Elf. For those queries, we reach a performance improvement of up to two orders of magnitude.

V. Related Work

Accelerating the evaluation of selection predicates in main-memory databases has become a vital task. We now review different types of related approaches and point out the difference regarding our approach.

A. *SIMD scans*

Current trends in main-memory databases show that optimized full-table scans (e.g., using SIMD) are able to compete with specialized index structures, because of the fast access in main memory compared to traditional disk-based approaches. Moreover, the sequential access pattern leads to cache consciousness. Therefore, using SIMD to accelerate database operations has gained much attention. In particular, accelerating selection conditions using SIMD scans result in high performance increases [22], [23], [24], [25], [26].

The first ideas of using SIMD in database selections by Zhou and Ross aim at accelerating full-table scans [26]. Polychroniou and Ross extend the work to AVX by using bloom filters [24] and Sitaridi and Ross to GPUs [25]. Willhalm et al. use compression and SIMD acceleration to speed up the scan for single [22] and complex [23] predicates. The results and insights of these publications are used in this paper to implement our SIMD scan. Moreover, we use SIMD to improve the Column Imprint competitor. Similarly, a recent publication by Polychroniou et al. improves BitWeaving's compression, decompression, and scan algorithm using AVX2 [20]. Although, all of these improvements result in an observable performance increase, they all still face the same conceptual issue. For multi-column selection predicate evaluation, we have to fully scan multiple columns and cannot apply an early pruning as performed by tree-based indexes.

A hybrid approach between scanning and indexing is database cracking by Idreos et al., which creates an index adaptively by sorting the data iteratively for each executed selection [27], [28]. Recently, database cracking has been improved using SIMD by Pirk et al. [29] and implemented as a parallelized background job by Petraki et al. [30]. However, this approach is also limited to single columns. Thus, each column has to be indexed and evaluated separately.

B. *Indexing approaches*

In the area of main-memory indexing, making tree structures optimized for caches has become essential. An early approach is the T-tree by Lehman and Carey [31]. According to their results this approach outperforms B-tree and AVL-tree. Currently, cache-conscious B-trees by Rao and Ross [21] and adaptive radix trees by Leis et al. [32] are promising index structures to improve cache utilization for index searches. Furthermore, the utilization of SIMD has become a topic in main-memory tree structures, e.g., in the k-ary search tree by Schlegel et al. [33], the Fast Architecture Sensitive Tree by Kim et al. [34], or the Segmented Tree by Zeuch et al. [35].

All these index structures support efficient mono-column predicate evaluation, but for each additional predicate on another column, an additional index structure has to be built and also searched. As a consequence, we argue that our approach is more space and time efficient for multi-column selection predicate queries.

The Data Dwarf [11] aims at storing a highly compressed version of the cube operator by eliminating the artificially created prefix and suffix redundancies of cube entries (i.e., grouping keys). However, the desired compression rates are hardly reached in practice [36]. By contrast, we use existing prefix-redundancies to exploit the full combined pruning power of a multi-column selection predicate in order to speed-up query evaluation.

C. *Elf competitors*

BitWeaving: BitWeaving is a bit-packing technique originally proposed by Li and Patel [6]. The idea of BitWeaving is to store the necessary bits (w.r.t. the given value range) of several values into one processor word (with a typical size of 64 bit). Hence, BitWeaving adapts the idea of SIMD even for scalar registers to exploit data parallelism in computation. Recent improvements are to use SIMD [20] and to add an encoding to the data that exploits skew in the data and predicate distribution [37]. For our evaluation, we execute the queries for both variants, BitWeaving/H and BitWeaving/V, but only take the fastest version as competitor for Elf (BitWeaving/V).

Column Imprint: A Column Imprint is a cache-conscious secondary index structure for range queries [10]. The idea is to apply a coarse-grained filter (similar to a bloom filter) indicating whether we can exclude a complete cache line for a given query. To this end, the Column Imprint builds a histogram over all values of a cache line and stores it in a 64-bit integer. The histogram is an equi-width histogram with 64 bins where a bit $b = 1$ means that at least one value of the corresponding cache line is in the range of the given bin. For evaluating selection predicates, we can use the histograms as a pre-filter such that we only have to evaluate the values of cache lines that definitely have a candidate w.r.t. the selection predicate. As an improvement, Polychroniou et al. propose novel vectorized designs based on advanced SIMD operations for database operators, explicitly including scans on Column Imprints [38]. We leverage their ideas into the variant pool of Column Imprint implementations selecting the best variant as competitor.

Sorted Projection: Sorted Projections are first introduced in C-Store [7], but also available in Vertica [39]. They are a simple yet powerful index that accelerates selections. For a frequently used set of queried columns, we sort the columns according to one attribute and add a column for the TIDs for tuple reconstruction purposes. As a consequence, we can now use binary search on the sorted column or compression techniques to accelerate query processing. For our implementation, we implement a sorted projection for each unique prefix of the queried columns from Table IV.

VI. Conclusions and future work

For domains like data warehousing or scientific computing it is essential to efficiently reduce large data sets according to a multi-column selection predicate. The result of such queries, is then used as input for further operations like join-processing or in-depth analysis (e.g., classifications). In contrast to other state-of-the-art approaches, we aim at fully exploiting the combined selective power of the predicate to efficiently compute the query result. So far, most approaches aim at exploiting the capabilities of modern CPUs for optimized full table scans. Using the combined selective power, tree-based indexing approaches seem promising even for main-memory scenarios where such approaches are mostly not considered today. To this end, we propose Elf, a tree-based index structure for multi-column selection predicate queries featuring prefix-redundancy elimination and a memory layout tailored to exploit modern in-memory technology. In empirical studies, involving synthetic queries as well as TPC-H queries and data, the results indicate that our approach outperforms state-of-the approaches up to a magnitude. We even reveal that our approach has competitive performance for mono-predicate selections, in case of low selectivity. On a more abstract level, the results reveal that the query selectivity is the dominant cost factor. Our approach is able to outperform state-of-the-art accelerated full-table scans up to a selectivity of 18 %. So far, values around 2 % have been reported. This emphasizes the significance of our contribution.

For future work, we examine the applicability of our approach for additional database operations, especially joins. Since the usage of Elf for groupings and aggregates are intuitive, an efficient join processing using Elfs is currently unknown. However, executing a partitioned join on several Elfs or applying wide table approach seems promising in this area [40], [41].

Acknowledgment

We like to thank Sebastian Breß, Sebastian Dorok, Wolfram Fenske, Reimar Schröter, and the anonymous reviewers for their constructive feedback.

References

[1] R. Johnson, V. Raman, R. Sidle, and G. Swart, "Row-wise parallel predicate evaluation," *PVLDB*, vol. 1, no. 1, pp. 622–634, Aug. 2008.

[2] P. Boncz, M. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008.

[3] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *ICDE*, April 2011, pp. 195–206.

[4] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *SIGMOD Keynote*. ACM, 2009, pp. 1–2.

[5] D. Das, J. Yan, M. Zait, S. R. Valluri, N. Vyas, R. Krishnamachari, P. Gaharwar, J. Kamp, and N. Mukherjee, "Query optimization in Oracle 12c database in-memory," *VLDB*, vol. 8, no. 12, pp. 1770 – 1781, 2015.

[6] Y. Li and J. Patel, "Bitweaving: Fast scans for main memory data processing," in *SIGMOD*. ACM, 2013, pp. 289–300.

[7] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," Foundations and Trends in Databases, 5(3): 197-280, pp. 197–280, 2013.

[8] G. Antoshenkov, "Dictionary-based order-preserving string compression," *The VLDB Journal*, vol. 6, no. 1, pp. 26–39, 1997.

[9] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*. ACM, 2009, pp. 283–296.

[10] L. Sidirourgos and M. Kersten, "Column imprints: A secondary index structure," in *SIGMOD*. ACM, 2013, pp. 893–904.

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

[12] V. Köppen, D. Broneske, G. Saake, and M. Schäler, "Elf: A main-memory structure for efficient multi-dimensional range and partial match queries," Otto-von-Guericke-University Magdeburg, Tech. Rep. 002-2015, 2015.

[13] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *SIGMOD*. ACM, 2016, pp. 1567–1581.

[14] J. Schneider, "Analytic performance model of a main-memory index structure," Bachelor Thesis, Karlsruhe Institute of Technology, 2015.

[15] J. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, 1975.

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

[17] Transaction Processing Performance Council, "TPC benchmark H (decision support)," Tech. Rep. 2.17.1, 2014.

[18] P. O'Neil, B. O'Neil, and X. Chen, "Star schema benchmark - Revision 3," 2009.

[19] Transaction Processing Performance Council, "TPC benchmark DS," Tech. Rep. 2.1.0, 2015.

[20] O. Polychroniou and K. Ross, "Efficient lightweight compression alongside fast scans," in *SIGMOD Workshop DaMoN*. ACM, 2015.

[21] J. Rao and K. Ross, "Making B^+-Trees cache conscious in main memory," in *SIGMOD*. ACM, 2000, pp. 475–486.

[22] T. Willhalm, Y. Boshmaf, H. Plattner, N. Popovici, A. Zeier, and J. Schaffner, "SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units," *PVLDB*, vol. 2, no. 1, pp. 385–394, 2009.

[23] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates," in *VLDB Workshop ADMS*, 2013, pp. 1–12.

[24] O. Polychroniou and K. Ross, "Vectorized bloom filters for advanced SIMD processors," in *SIGMOD Workshop DaMoN*. ACM, 2014.

[25] E. Sitaridi and K. Ross, "Optimizing select conditions on GPUs," in *SIGMOD Workshop DaMoN*. ACM, 2013, pp. 4:1–4:8.

[26] J. Zhou and K. Ross, "Implementing database operations using SIMD instructions," in *SIGMOD*. ACM, 2002, pp. 145–156.

[27] S. Idreos, M. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007, pp. 68–78.

[28] M. Kersten and S. Manegold, "Cracking the database store," in *CIDR*, 2005, pp. 213–224.

[29] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten, "Database cracking: Fancy scan, not poor man's sort!" in *SIGMOD Workshop DaMoN*. ACM, 2014, pp. 4:1–4:8.

[30] E. Petraki, S. Idreos, and S. Manegold, "Holistic indexing in main-memory column-stores," in *SIGMOD*. ACM, 2015, pp. 1153–1166.

[31] T. Lehman and M. Carey, "A study of index structures for main memory database management systems," in *VLDB*, 1986, pp. 294–303.

[32] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *ICDE*. IEEE, 2013, pp. 38–49.

[33] B. Schlegel, R. Gemulla, and W. Lehner, "K-ary search on modern processors," in *SIGMOD Workshop DaMoN*. ACM, 2009, pp. 52–60.

[34] C. Kim *et al.*, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *SIGMOD*. ACM, 2010, pp. 339–350.

[35] S. Zeuch, J.-C. Freytag, and F. Huber, "Adapting tree structures for processing with SIMD instructions." in *EDBT*, 2014, pp. 97–108.

[36] J. Dittrich, L. Blunschi, and M. Salles, "Dwarfs in the rearview mirror: How big are they really?" *PVLDB*, vol. 1, no. 2, pp. 1586–1597, 2008.

[37] Y. Li, C. Chasseur, and J. M. Patel, "A padded encoding scheme to accelerate scans by leveraging skew," in *SIGMOD*. ACM, 2015, pp. 1509–1524.

[38] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *SIGMOD*. ACM, 2015, pp. 1493–1508.

[39] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The Vertica Analytic Database: C-store 7 years later," *PVLDB*, vol. 5, no. 12, pp. 1790–1801, 2012.

[40] S. Baumann, P. Boncz, and K.-U. Sattler, "Bitwise dimensional co-clustering for analytical workloads," *The VLDB Journal*, pp. 1–26, 2016.

[41] Y. Li and J. M. Patel, "Widetable: An accelerator for analytical data processing," *PVLDB*, vol. 7, no. 10, pp. 907–918, Jun. 2014.