Efficient evaluation of multi-column selection predicates in main-memory

David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler

Abstract—Efficient evaluation of selection predicates is a performance-critical task, for instance to reduce intermediate result sizes being 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 a large-scale 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 queries returning less than 10% of all tuples — a selectivity almost one order of magnitude larger than observed for related indexing approaches. For TPC-H queries with multi-column selection predicates, we achieve a speedup between factor five and two orders of magnitude, mainly depending on the selectivity of the predicates. Further scaling experiments reveal that for large data sets, these speedup factors are expected to increase, due to more densely populated data spaces. Finally, our results indicate that using a delta-store like concept to support periodic insertions results in virtually no performance penalty for reasonable sizes of a write-optimized Elf as delta store.

Index Terms—multi-column selection predicates, main-memory databases, hardware-sensitive indexing

1 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 filtering 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 systems. In a recent study, Das et al. propose to use an index structure for very low selectivities only, such as values smaller than 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.

Manuscript received November 24, 2017.
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. Furthermore, our approach features a fixed search path as each level belongs to one column and this 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) Our evaluation including a micro benchmark and multi-column selection predicates from the TPC-H benchmark shows 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%.

This is an extended version of [8] and, in addition to the original contributions, this paper also features:

1) A detailed description of the Elf build algorithm reducing building Elf to incremental sorting of the data.
2) We introduce a mechanism to support periodic insertions of new data, such as daily updates, using a read-optimized Elf and a write-optimized Elf. An in-depth evaluation reveals that the effect on query performance can be neglected in case the size of the write-optimized Elf does not exceed 0.1% of the overall data. Moreover, the results suggest lower and upper bounds for a periodic merging of the two Elfs.
3) All experiments from [8] are conducted using a larger scaling factor \( s = 200 \) instead of \( s = 100 \) of the TPC-H benchmark verifying all results.
4) Additional experiments indicate that Elf scales better than any competitor. This suggests that the observed performance increases are even higher in case the data size increases. The reason is that the data space is more densely populated and, thus, Elf can exploit more prefix-redundancy eliminations.

The remainder of the paper is organized as follows: In Section 2, we give a definition of the problem of evaluating multi-column selection predicates and a description of redundancy elimination. In Section 3, we explain details of the implementation and optimization of the Elf approach. Building and maintaining Elf is subject of Section 4. In Section 5, we evaluate Elf’s performance against well-known state-of-the-art competitors. In Section 6, we briefly discuss related approaches and summarize in Section 7.

## 2 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 the Elf index structure.

### 2.1 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 \( \text{col} \in C \), there is one of the following basic predicates given: =, <, >, \( \leq \), \( \geq \), BETWEEN. Column data and constants in the predicate are numeric integer values either by definition of the schema or due to order preserving dictionary encoding [9], [10]. For the remainder of the paper, we assume the latter case. As a result, the predicate defines a (different) window on each \( \text{col} \) and we can transform these predicates into one notation and treat them in a uniform manner as defined in Table 1. For example, \( \text{col} = x \), 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, \( \text{col} < x \) 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 \( R_{\text{mcsp}} \) is a position list containing the references of all tuples (Ref), which can be used for subsequent operations like joins.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = x )</td>
<td>([x, x])</td>
</tr>
<tr>
<td>( &lt; x )</td>
<td>([\min, x])</td>
</tr>
<tr>
<td>( \leq x )</td>
<td>([\min, x])</td>
</tr>
<tr>
<td>( &gt; x )</td>
<td>([x, \max])</td>
</tr>
<tr>
<td>( \geq x )</td>
<td>([x, \max])</td>
</tr>
<tr>
<td>( \leq x \land y \land x \leq y ) (BETWEEN)</td>
<td>([x, y])</td>
</tr>
</tbody>
</table>

**Table 1** Columnar selection predicate translation

**Definition 2.1** (Result position list: \( R_{\text{mcsp}} \)). Let \( \text{Ref}_i \) denote the tuple identifier of the \( i^{th} \) tuple \( t_i \) in the data set. Moreover, let \( \text{SAT}_{\text{mcsp}}(\text{Ref}_i) \) be a Boolean function that is true, if all attribute values of \( t_i \) for all columns are defined in the window by query \( \text{mcsp} \). Then, \( R_{\text{mcsp}} \) is a list of identifiers such that \( \text{Ref}_i \in R_{\text{mcsp}} \iff \text{SAT}_{\text{mcsp}}(\text{Ref}_i) = \text{true} \).

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 [6], [11].

### 2.2 Prefix-redundancy elimination

An interesting concept that has been observed by Sismanis et al. is prefix-redundancy elimination [12]. Prefix redundancies occur whenever two or more dimension keys share a common prefix. This is visible in the example data of Table 2, where tuple \( t_1 \) and \( t_2 \) share the same value in the first dimension. We formalize this as:

**Definition 2.2** (Prefix-redundancy). Let \( t_a \) and \( t_b \) be two tuples over the same schema having \( n \) columns. Let \( \Pi \) denote an ordering of all \( n \) columns and let \( t[i] \) be the first and \( t[i] \) be the \( i^{th} \) attribute value of some tuple \( t \) according to \( \Pi \). Then, we observe a prefix-redundancy regarding \( \Pi \) 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 \( k_{\text{max}} \) for that we observe a prefix redundancy between two tuples.

## 3 Elf index structure

Based on the insights from Section 2, we design a novel index structure for order-preserving dictionary-compressed data or nu-
meric 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.

### 3.1 Conceptual design

In the following, we explain the basic design with the help of the example data in Table 2. 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).

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.3</td>
<td>1</td>
<td>T1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0.3</td>
<td>0</td>
<td>T2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5.2</td>
<td>0</td>
<td>T3</td>
</tr>
</tbody>
</table>

**TABLE 2 Running example data**

In Fig. 2, we depict the resulting Elf for the four indexed columns of the example data\(^1\) from Table 2. 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 pointer points to 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).

**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 with 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.

### 3.2 Improving Elf’s memory layout

The straightforward implementation of Elf resembles 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.

#### 3.2.1 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.

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

The conceptual Elf structure is designed from the idea of prefix-redundancy elimination in Section 2.2 and the properties of multi-column selection predicates. To this end, it features the following properties 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.

1. Note, we assume that all values within one column have a fixed-length data type, such as integer or double. In order to store variable length types, we conduct a dictionary encoding known to work well e.g., for strings.

**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.
3.2.2 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\(^2\), 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 TID list by setting the most significant bit, allowing to store duplicates as well.

3.3 Storage optimizations
Considering the structure of Elf depicted in Fig. 2, we can 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.

3.3.1 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. Moreover, 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\(^3\). 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 values in a DimensionList are unique.

Denseness. Due to the order preserving dictionary compression of the data, all integer values between 0 and the maximum value max\(_D\) 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 \([0, \text{max}_D]\), 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. 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. This way, we remove the deterioration of the first DimensionList and 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.

\(^2\) 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. Furthermore, we manipulate the sign bit for floating-point values.

\(^3\) With perfect hash map, we mean that we can represent the hash map as a dense array, where the keys represent the array positions.

3.3.2 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. We display this issue for the TPC-H Lineitem table 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 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.

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, thus, the deterioration of one-element lists is resolved. 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 \(T_3\).
3.4 Search algorithm

In the following, we present the algorithm to evaluate a multi-column selection predicate within Elf, based on Definition 2.1. The algorithm mainly consists of two functions.

Algorithm 1: Search multi-column selection predicate

```
Result: L ResultList
1 SearchMCSP (lower, upper) {
2   L ← ∅;
3   if (lower[0] ≤ upper[0]) then // predicate on list column - exploit hash-map
4     start ← lower[0]; stop ← upper[0];
5   else
6     start ← 0; stop ← max (C1);
7   end if
8   for (offset ← start to stop) do
9     pointer ← Elf[offset];
10    if (notMonoList (pointer)) then
11       SearchDimList (lower, upper, pointer, col ← 1, L);
12    else
13      L ← L+SearchML (lower, upper, unsetMSB (pointer), col ← 1, L);
14    end if
15   end for
16   return L;
17 }
```

Algorithm 2: Scan a DimensionList within an Elf

```
1 SearchDimList (lower, upper, startlist, col, L) {
2   if (lower[col] ≤ upper[col]) then
3     position ← startList;
4     while (notEndOfList (Elf[position]) do
5       L ← L+SearchML (lower, upper, position, col, L);
6       if (position > upper[col]) then
7         return; // abort
8       else
9         position ← position + 2;
10      end if
11   end while
12   // call SearchDimList or SearchML with col+1 for all elements
13 }
```

3.5 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 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. 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.

4. 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.
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 3). It is computed as follows: \( \text{overhead}(n) = (n + 1)/n \).

<table>
<thead>
<tr>
<th>Number of columns</th>
<th>Storage overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
</tr>
<tr>
<td>3</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>5</td>
<td>1.20</td>
</tr>
<tr>
<td>6</td>
<td>1.17</td>
</tr>
</tbody>
</table>

**Table 3** 
Upper bound storage overhead

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

### 3.6 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 [13].

Due to the design of Elf, the first column should be the most often 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.

### 4 THE ELF LIFECYCLE

The primary application field of Elf are data warehousing scenarios having read-mostly workloads with periodic insertions. To this end, we need to support initial build and periodic insertions efficiently. In this section, we give technical details on how Elf supports initial building by means of multi-dimensional sorting (cf. Section 4.1). In addition, we explain how Elf supports periodic updates (cf. Section 4.2) reducing this task to merging of pre-sorted lists. We evaluate both solutions in Experiment 5 & 6 in Section 5. Finally, in Section 4.2, we outline how Elf handles updates and deletions.

#### 4.1 Initial Build: Elf Bulk Load

The initial build of Elf is executed as a bulk load, where all data of the table is read to create the Elf with its explicit memory layout (cf. Algorithm 3). The build procedure consists of a step-wise multi-dimensional sort paired with a build of all DimensionLists of the currently sorted dimension using a preorder linearization.

The build is invoked as: **BuildDimList**\((data[dim]=0, start=0, num=data.size)\), where data is a two-dimensional array. First, the whole data set is sorted according to the first dimension and the build algorithm is executed for each DimensionList. For each call, we know that all points between \(data[start]\) and \(data[start+num]\) refer to the current DimensionList and are already sorted according to the prefix until dimension \(dim-1\). The algorithm then additionally sorts these points according to the current dimension \(dim\). Next, all existing values within the current DimensionList are linearized starting with the smallest one. Note, so far we do not know where the corresponding sub tree (i.e., the next Elf level) will start. Thus, we store the position where this pointer is located and count how many points refer to this sub tree in an auxiliary structure \(pf\) (cf. Line 3-15). Finally, the algorithm linearizes the corresponding sub tree of the first DimensionElement entirely, before it moves on to the next. In case more than one point refers to that sub tree, a recursive call is executed (Line 21), otherwise a MonoList is created. As build times are an important factor of the practicality of Elf, we compare the build times of all competitors in Section 5.5.

#### Algorithm 3: Building an Elf

```
1 BuildDimList (data[dim], dim, start, num, writePointer) {
2     // (1) incremental sort w.r.t. a given dimension
3     sort (data[start], num, dim);
4     // (2) determine all stores and value position of child pointers
5     pf ← new list(); // of 2-tuples (position, frequency)
6     cur ← data[start][dim]; // smallest value in this dim
7     for (i ← start + 1 to start + num) do
8         if (cur ! = data[i][dim]) then
9             Elf[writePointer]← cur; // write this value
10            pf.add((writePointer, 1)); // (position, frequency)
11            writePointer++ = 2; // DIM_Element size
12            cur ← data[i][dim];
13        end if
14        pf[last.freq]++; // in final dimension: write all TIDs
15    end for
16    setMSB (Elf[writePointer-3].); // End of DimList
17    cur ← final dimension
18    offset ← start;
19    for (i ← 0 to pf.size) do
20       Elf[pf[i].pos] ← writePointer; // pointer to begin set
21       if (pf[i].size > 1) then
22            Elf[pf[i].pos] ← BuildDimList (data, dim+1, offset, pf[i].size);
23        else
24            setMSB(Elf[pf[i].pos]); // mark as monolist
25            write all remaining dim values, then
26            all TIDs, set MSB of last TID
27        end if
28        offset ← offset + pf[i].freq;
29    end for
30 } // in final dimension: write all TIDs
```

#### 4.2 Maintaining an Elf: Insert, Update, and Delete

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.

Insertions: Our solution for periodic inserts consists of two parts. First, new data is collected in an auxiliary data structure named InsertElf. It has the same conceptual design as a normal Elf without the explicit memory layout and MonoLists (i.e., we use the design from Fig.2). That is, DimensionLists are lists in that we can insert easily. The idea is similar to delta stores in columnar databases [14]: there is one write-optimized InsertElf and one linearized read-optimized Elf. When a specific threshold of insertions is reached, data should be transferred from the InsertElf to Elf, being a merge of both structures.

By concept, Elf introduces a total order into the multi-dimensional data space. As the read-optimized Elf and its write-optimized counterpart imply the same order, we can exploit this for reducing the problem of merging two Elfs to the problem of merging pre-sorted lists. Therefore, the merge algorithm works at DimensionList level (cf. Algorithm 4) and is highly similar to merging two sorted lists of elements. The algorithm starts at the first element of the root DimensionList of both Elfs in order to merge both roots (i.e., DimensionList). To merge two DimensionLists, the algorithm first compares the values of the first DimensionElements differenciating three cases:
1) If the value of the linearized Elf is smaller, the common prefix ends here. Hence, the sub tree of the linearized Elf is copied into the new Elf without changes (Line 5).
2) If the value of the InsertElf is smaller, there is an insertion of new data to be done. In this case, the whole sub tree of the InsertElf is linearized into the new Elf (Line 8).
3) If the value in the InsertElf and the linearized Elf is the same, the prefix redundancy is further exploited. This leads to a subsequent merge of the underlying DimensionList of the InsertElf and the linearized Elf (Line 12).

After comparing the first two elements, the algorithm increments the smaller position in the two DimensionLists in order to compare and merge the next values, until the end of one of the lists is reached. Due to the sorting criteria of both structures, we can efficiently combine both structures with a complexity of $O(Elf_{size} + InsertElf_{size})$. However, even if the InsertElf is by several orders of magnitude smaller than the read-optimized Elf, there is still some performance loss on query execution to be expected. We quantify this performance loss in Section 5.6 considering different sizes of the InsertElf. Moreover, for any approach relying on a combination of read-optimized and write-optimized structures, such as delta stores, solving the problem when to merge both structures is important. To this end, we also conduct experiments in Section 5.6 to answer this question.

```
MergeDimLists(toBeInsertedList, position, newElf, writePointer) {
  ifPosition = 0;
  while (notEndOfList(toBeInsertedList[ElfPosition]) ∧ notEndOfList(toBeInsertedList[ElfPosition]))
    if (ElfPosition < toInsertDimList[ElfPosition]) then
      writePointer ←
      copySubTree(ElfPosition+1, newElf, writePointer);
      position ← position + 2;
    else if (ElfPosition > toInsertDimList[ElfPosition]) then
      // Process remaining entries of longer DimList
      return
    else
      writePointer ←
      MergeDimLists(toBeInsertedList[position].child(), Elf[NewElf, writePointer]);
      position ← position + 2;
      writePointer ←
      MergeDimLists(toBeInsertedList[position].child(), Elf[NewElf, writePointer]);
      position ← position + 2;
  end if
}
```

Algorithm 4: Merge a linearized DimensionList within a DimensionList of the InsertElf

Deletion: 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 TIDs. 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.

Updates: Finally, updates are rare for analytical workloads, but possible within Elf. Generally, there is a large amount of MonoLists (cf. Section 3.3.2). Updating a value in 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.

5 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 200. 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. Moreover, we investigate how the response times scale for different data set sizes. As competitors, we select three state-of-the-art approaches, BitWeaving/V [6], Column Imprint [11], and Sorted Projection [7]. In addition, we compare our approach to a columnar scan and an optimized SIMD-accelerated version (both with bit maps as intermediate results) as a good baseline. We also 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, Elf trades 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. The code of our evaluation is provided on the project website\(^5\). The result of a multi-column selection predicate evaluation is a position list complying to Definition 2.1. All experiments are single threaded to support an inter-operator parallelism concept, which we deem best for OLAP workloads. We perform our experiments on an Intel Xeon E5-2630 v3 (Haswell architecture) with max. 3.2 GHz clock frequency, 20 MB L3 cache, and 1 TB RAM. Our SIMD optimizations are implemented using AVX2. 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.

5.1 Experiment 1: Micro benchmark

In this experiment, we examine how well our approach scales for different selectivities and for different amounts of queried columns. 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].

To this end, we conduct experiments on the Lineitem\(^6\) table with $s = 200$. 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:

\(^5\) www.elf.ovgu.de
\(^6\) This table takes about 144 GB of memory. Note, for a fair comparison, we only use the order-preserving dictionary encoded data (with a size of ca. 72 GB) for all experiments and all competitors.
$l \in \{0,1,2,3,4\}$. 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 $l_{\text{shipdate}}$ and the second one $l_{\text{discount}}$. The associated SQL query is:

```
SELECT l_{\text{shipdate}}, l_{\text{discount}} FROM lineitem  
WHERE l_{\text{shipdate}} BETWEEN c1 AND c2  
AND l_{\text{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 several times to achieve reliable results.

![Graph](image.png)

**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 quite stable for varying selectivities. For example, for the queries on two columns, the SIMD scan requires 2.252 $s$ for 1\% selectivity and 2.797 $s$ for the highest selectivity, this is only 24\% more. 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 2.528 $s$ for one column and 4.276 $s$ for five columns, leading to 69\% increased time.

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

**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 bound 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 has 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. An Elf, representing the `Lineitem` table, has 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 compare it to the total number of tuples in the table, which is about 1.2 billion.

**Break-even point at higher selectivity than expected:**

The break-even point is by one order of magnitude larger (10\% to 20\% in contrast to 1\% to 2\%) than postulated in the literature. In fact, Elf does not only outperform the baseline, but this also holds for a selectivity exceeding 10\%; 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 10\% selectivity, it is at 17\% selectivity for five queried columns.

The results of our micro benchmark indicate Elf is superior to the SIMD sequential scan for predicates with a low selectivity (i.e., small result sets) on several columns. However, 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 workloads from the TPC-H benchmark.

### 5.2 Experiment 2: MonoList 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 3.5 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 ($s = 200$). 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.

<table>
<thead>
<tr>
<th>Storage Consumption in GB</th>
<th>Raw Data</th>
<th>Elf w/o MonoLists</th>
<th>Elf with MonoLists</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>72.00</td>
<td>95.84</td>
<td>49.32</td>
</tr>
</tbody>
</table>

**TABLE 4: Storage consumption for Lineitem table**

In Table 4, 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 72 GB, while the Elf without MonoList optimization consumes 95.84 GB and the fully optimized Elf consumes 49.32 GB of RAM. This is a remarkable result, because the Elf is not only taking only 68\% of the 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 (cf. Fig. 5). For instance, at level 9, we encounter around 87 million MonoLists, that save around 4 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.
5.3 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], [11]. 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. We select queries having a multi-column selection predicate and additional ones having a mono-column selection predicate, as summarized in Table 5. 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 \( \text{Col}_{\text{Elf}} \) is also important for Sorted Projections, because we create one Sorted Projection per distinct prefix.

The mono-column selection predicate queries are selected to explore the general applicability (and limitations) of Elf for real-world workloads. To this end, 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 queried the whole 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.

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 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.

<table>
<thead>
<tr>
<th>Query</th>
<th>( \sigma ) in %</th>
<th>Predicate Columns</th>
<th>( \text{Col}_{\text{Elf}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>98.0</td>
<td>( l_{\text{shipdate}} )</td>
<td>0</td>
</tr>
<tr>
<td>Q10</td>
<td>24.68</td>
<td>( l_{\text{returnflag}} )</td>
<td>4</td>
</tr>
<tr>
<td>Q14</td>
<td>1.3</td>
<td>( l_{\text{shipdate}} )</td>
<td>0</td>
</tr>
<tr>
<td>Q6</td>
<td>1.72</td>
<td>( l_{\text{shipdate}}, l_{\text{discount}}, l_{\text{quantity}} )</td>
<td>{0.1,2}</td>
</tr>
<tr>
<td>LQ19</td>
<td>1.4</td>
<td>( l_{\text{quantity}}, l_{\text{shipmode}}, l_{\text{shipdate}} )</td>
<td>{2.5,6}</td>
</tr>
<tr>
<td>Q17</td>
<td>0.099</td>
<td>( p_{\text{brand}}, p_{\text{container}} )</td>
<td>{1}</td>
</tr>
<tr>
<td>PQ19</td>
<td>0.083</td>
<td>( p_{\text{brand}}, p_{\text{container}}, p_{\text{size}} )</td>
<td>{1.2}</td>
</tr>
</tbody>
</table>

**Table 5** Query details for mono and multi-column selections

In the following, we depict the values for the selection predicates of the TPC-H benchmark with its order-preserving dictionary-compressed data. We generate 1,000 random predicates by varying selection predicate parameters 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 5 is 6 (i.e., the seventh column), we also evaluate an Elf that only indexes the first seven columns. We refer to the reduced Elf as Elf\(_7\), named by the number of columns (incl. \( TID \)) it contains. Note, we omit a detailed evaluation of the update-optimized InsertElf, 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.

5.3.1 Mono-column selection predicate queries

In Fig. 9, 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 the fifth column, using the Elf\(_7\) results in a response time comparable to both state-of-the-art approaches. However, the Elf containing all columns is 80% 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 3,415 ms whereas the Column Imprint requires 1,580 ms. Thus, the performance gains for accelerated scans are around 45%. 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.

![Fig. 9. Query response times for mono-column TPC-H queries (\( s = 200 \))](image-url)

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 1,885 ms. By contrast, the response time of the Elf is 318 ms and the Elf\(_7\) requires 72 ms. Consequently, our results show a performance gain of almost a factor of 6 for the Elf and of more than a factor of 26 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 Query Q14, we benefit from the hash-map property and the fact that the selection column is at the first instead of the fourth level, as in Q10.

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 among
the two fastest approaches and for the third query the fastest full-
table scan. 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 [18] 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.

5.3.2 Multi-column selection predicate queries
In contrast to mono-column selection predicates, we observe Elf’s
superiority for all multi-column selection predicate queries in
Fig. 10. 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 Elf7 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 7,839 ms and 8,404 ms, respectively. The speedup factor
of the Elf (1,743 ms) is 4.5 and considering the Elf7 (625 ms) it is
factor 12. By contrast, the largest performance gain is measured
for the queries with the smallest result sizes: Q17 and PQ19 (cf.
Table 5). It is in the order of almost two orders of magnitude.

![Graph](image_url)

Fig. 10. Query response times for multi-column TPC-H queries (s = 200)

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 outper-
forms 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 [19]), because the selectivity for one column is still
much higher than 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 speedups 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 speedups
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 Elf7 also
seem plausible as they directly correlate to the difference in size
between Elf and Elf7. 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 speedups 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 speeds of factor five to even one order of magnitude.

On a more abstract level, we observe that all approaches outper-
form 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 [20, [21].
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 speedups 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.

5.4 Experiment 4: Selection time scaling
We now investigate how the selection time of Elf scales. Our
hypothesis is that Elf scales with a smaller linear factor, e.g., in
case one doubles the amount of data, the selection time increase
is less than factor two. The rational is that, when the data size
increases, the data space is more densely populated and therefore,
we observe more prefix redundancy eliminations in Elf. This would
be a valuable property of Elf, because sequential scans (including
optimized ones), by concept, scale with factor 1. In addition, other
tree-based approaches, such as the kd-tree, face issues reaching a
linear scaling [16].

To investigate the validity of this hypothesis, we conduct the
following experiment: We select two pairs of TPC-H scaling factors
(s\text{small}, s\text{large}) such that 2 × s\text{small} = s\text{large} holds. That is, the data
size is doubled. In particular, we use (50, 100) and (100, 200).
For each pair, we determine the selection time t_{\text{q}} for both scaling
factors and all three mono-column selection predicate as well as
the four multi-column selection predicate queries having seven
queries in total: q with 1 ≤ q ≤ 7 (cf. Section 5.3). Then, we
compute the selection time scaling ratio per query as division of
t_{\text{q}}^{s\text{large}} and t_{\text{q}}^{s\text{small}} normalizing by the data size increase, i.e., factor 2.
As a result, a selection time scaling ratio of 1.0 indicates that
the approach scales linearly for this query and scaling factor pair.
In turn, a value observably smaller than 1.0 suggests a lower scaling.
To confirm our hypothesis, we require that using Elf and Elf7, on
average considering all 14 measurements, results in a selection
time scaling ratio below 1.0.
we confirm that Elf scales with a lower linear scaling factor than 1. where the selection time scaling ratio is slightly larger than way with almost no deviation. Interestingly, we also observe, on
The purpose of the build time examination is to evaluate whether a
This indicates a clear smaller linear scaling.

The linear scaling below 1 is more evident for the reduced Elf
This is because all (optimized) sequential scans scale in a linear
is the time to allocate aligned memory and copy the respective
maximum and the overall selection time would be the
meaning that the selection time is the sum of individual selection times. Generally, one can execute the query on both Elfs in parallel and the overall selection time would be the maximum of the
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 three-quarters of an hour for finishing the build. The next fastest approach is BitWeaving, with a build time of 443.71 s. Building the Sorted Projection approach requires 530.93 s. Thus, BitWeaving is only about two times faster than building the Elf (907.03 s). Therefore, we argue that build times are no counterargument for the applicability of our approach, especially as we reach a speedup of one order of magnitude for the query performance. For instance, in analytic environments, such an additional build time is acceptable. Interestingly, the Elf7 is the third fastest approach. Note, Column Imprint and kd-Tree require more build time than Elf.

5.6 Experiment 6: Periodic insert mechanism
In Section 4, we propose a mechanism to handle periodic inserts consisting of a read-optimized Elf containing the majority of the tuples and a write-optimized InsertElf collecting newly inserted tuples. Furthermore, the mechanism contains a merge algorithm to merge both Elfs into one read-optimized Elf. We now investigate its usefulness by answering the following questions:
1) What is the selection time overhead caused by querying both Elfs based on the fraction of tuples stored in the InsertElf?
2) When is a merge more cost-efficient than querying both structures considering the number of executed queries and the number of tuples in the InsertElf?

In summary, in addition to the already observed high speedups for Elf and Elf7, our results suggest that the more data is indexed, the higher are the expectable selection time speedups. We deem this a remarkable result, because Elf is the only approach showing such behavior considering state-of-the-art competitors.

5.5 Experiment 5: 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.

In Fig. 12, we visualize the build times for the Lineitem table. The build time of the SIMD sequential scan (182.36 s) is the time to allocate aligned memory and copy the respective

![Fig. 11. Selection time scaling ratios for all approaches](image1)

![Fig. 12. Build time for Lineitem table of s = 200](image2)

![Fig. 13. Normalized runtime overhead caused by different InsertElf sizes in the TPC-H queries on the Lineitem table (s = 200)](image3)
To determine when to merge the InsertElf into the linearized Elf becomes more efficient than querying both structures.

latest executed for a ratio of \( r \) as a lower bound. For frequently added data, a merge should be \( r < 0.01 \) for most queries (for instance, 16 minutes of executing queries). In fact, postponing the merge can lead to a performance loss of \( r \) factor 8 for a query workload 10 times, a merge would have been more beneficial. However, for small ratios \( (1 - r) \% \) into the read-optimized Elf. In Fig. 13, we observe that the overhead is scaling similar to the ratio of indexed tuples by the InsertElf. Thus, for reasonable amounts of data, the overhead can be neglected. That is for \( r \leq 0.01 \) the overhead is hardly measurable. Furthermore, we consider an overhead of about \( 5 \% \) for \( r = 0.1 \% \) as acceptable stating the efficiency of the periodic insert mechanism. Nevertheless, the high overhead for \( r \geq 1 \% \) makes a merge inevitable.

5.6.2 Merge threshold

To determine when to merge the InsertElf into the linearized Elf best, we have to examine when the threshold of all executed queries is high enough that querying both structures is less efficient than merging them and querying the resulting Elf. In Fig. 14, we show the runtime of executing a merge and a multiple of the queries (Q1, Q6, Q10, LQ19) compared to executing these queries on both, the linearized Elf and InsertElf, w.r.t. different ratios \( r \).

We can observe the exponential increase of the runtime for querying both data structures. While for small ratios \( (r < 0.01 \%) \), a merge is more costly than querying the InsertElf and linearized Elf, this behavior changes for \( r = 0.5 \% \). Here, when executing the query workload 10 times, a merge would have been more beneficial. In fact, postponing the merge can lead to a performance loss of factor 8 for \( r = 2 \% \) and 20 executed queries. Notably, executing more than one query would lead to a performance loss for \( r = 2 \% \) and, hence, a merge is inevitable.

These results indicate that for workloads with less frequent updates, a merge is necessary right away, because after a short time of executing queries (for instance, 16 minutes of executing queries for \( r = 0.02 \%) \) a merge would have paid off, which can be seen as a lower bound. For frequently added data, a merge should be latest executed for a ratio of \( r \geq 2 \% \), because in this case merging becomes more efficient than querying both structures.

5.7 Result summary

Based on the results of all our experiments, we have empirically shown the superiority of our approach compared to several strong competitors. Although initial build times are a challenging factor for Elf, it well supports periodic insertions of new data of reasonable size, which is highly relevant for analytic scenarios.

In comparison to a SIMD sequential scan, Elf performs better for small selectivities (11–18 %) and the more columns are queried, the higher the acceptable selectivity. Moreover, our MonoList optimization reduces the storage overhead by 50 % compared to an Elf without MonoList and by 30 % compared to the raw data. Furthermore, our approach performs best for low-selectivity workloads from non-synthetic queries, such as those of 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. Finally, our results indicate that Elf provides best scaling behavior of all considered approaches.

6 Related Work

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

6.1 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 and has shown significant performance increases.

The first ideas of using SIMD in database selections by Zhou and Ross aim at accelerating full-table scans [22]. Polychroniou and Ross extend the work to AVX by using bloom filters [23] and Sitardi and Ross to GPUs [24]. Willhalm et al. use compression and SIMD acceleration to speedup the scan for single [20] and complex [21] predicates. The results and insights of these publications are used in this paper to implement our SIMD scan. Moreover, we use SIMD to accelerate database operations has gained much attention and has shown significant performance increases.

The first ideas of using SIMD in database selections by Zhou and Ross aim at accelerating full-table scans [22], Polychroniou and Ross extend the work to AVX by using bloom filters [23] and Sitardi and Ross to GPUs [24]. Willhalm et al. use compression and SIMD acceleration to speed up the scan for single [20] and complex [21] 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 [18]. 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 [25], [26]. Recently, Pirk et al. [27] improved database cracking using SIMD and Petraki et al. [28] implemented it as a parallelized background job. However, this approach is also limited to single columns. Thus, each column has to be indexed and evaluated separately.

6.2 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 [29]. According to their results,
this approach outperforms B-tree and AVL-tree. Currently, cache-conscious B-trees by Rao and Ross [19] and adaptive radix trees by Leis et al. [30] are promising index structures to improve cache utilization for searches. Considering B-trees, prefix B-Trees [31] and composite-key B-Trees with prefix compression [32] are the most similar approaches to our Elf. However, keys inside a B-tree are distributed over the whole tree prohibiting an early pruning.

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 tree FAST by Kim et al. [34], or the Segmented Tree by Zeuch et al. [35]. The log-structured merge-tree maintains data in two or more separate structures and is efficient for high inserts in hot and cold data scenarios with a usage on main-memory and flash disk [36], [37]. 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 used. Therefore, we argue that our approach is more space and time efficient for multi-column selection predicate queries.

The Data Dwarf [12] 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 [38]. By contrast, we use existing prefix-redundancies to exploit the full combined pruning power of a multi-column selection predicate in order to speedup query evaluation.

There is various work about indexing multiple columns on disk-based systems to minimize the number of accessed pages such as [39], [40]. However, they assume that data is split in a set of (potentially overlapping) horizontal or vertical fragments. In contrast, Elf flexibly adapts to the data distribution. That is, without having to specify when to switch, Elf continuously changes from one large list containing all distinct values (columnar storage) in the first column, to a MonoList when a prefix becomes unique (row-wise storage). Such MonoLists can start at any Elf level.

There is also work on when indexes should be preferred over sequential scans in main-memory, such as [41]. They refer to single columns or translate a multi-column selection predicate with \( n \) predicates to \( n \) mono-column queries including a result computation, e.g., by merging bit vectors. Selectivity thresholds for switching from index to full-table scans reported by such studies are about a magnitude smaller than we observe for Elf. The reason is that Elf renders materialization of intermediate results redundant.

6.3 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 [18] and to add an encoding to the data that exploits skewness in the data and predicate distribution [42]. For our evaluation, we execute the queries on BitWeaving/H and BitWeaving/V, but only take the fastest version, BitWeaving/V.

Column Imprint: A Column Imprint is a cache-conscious secondary index structure for range queries [11]. 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 this 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, explicitly including scans on Column Imprints [43]. 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 [44]. 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. Therefore, we can use binary search on the sorted column or compression techniques to accelerate query processing. For our evaluation, we implement a sorted projection for each unique prefix of the queried columns from Table 5.

7 Conclusion 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 technologies. 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 an order of magnitude. We even reveal that our approach has competitive performance for mono-column selection predicates, 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 17%. So far, values around 2% have been reported. This emphasizes the significance of our contribution. Furthermore, we have shown that Elf’s performance scales with a better factor than its competitors. Additionally, reasonable amounts of newly inserted data are well supported by our write-optimized Elf as delta store.

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 is 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 [45], [46].

References


[10] IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. XX, NO. XX, MONTH YEAR 14


David Broneske studied Computer Science at the University of Magdeburg (Bachelor of Science 2012, Master of Science 2013). Currently, he is a PhD student at the Databases and Software Engineering Group of Gunter Saake. His research interests include hardware-sensitive database operations and data structures.

Veit Köppen received his PhD in 2008 from Freie Universität Berlin. 2008-2015 he was scientific coordinator at the Otto-von-Guericke University Magdeburg. Since 2016, he is a head of the department for IT solutions at the library of the Otto-von-Guericke University. His research interests include business intelligence, data warehousing, and information systems.

Gunter Saake is a full professor of Computer Science. He received his PhD in 1988 from University of Braunschweig. Currently, he is the head of the Databases and Software Engineering Group at the University of Magdeburg. His research interests include database integration, tailor-made data management, database management on new hardware, and feature-oriented software product lines.