University of Magdeburg

Faculty of Computer Science

Master's Thesis

Level Order Linearization for the Elf Approach

Author:

Huanqing Shang

November 20, 2020

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Department of Technical and Business Information Systems

Dr.-Ing. David Broneske
Department of Technical and Business Information Systems

Shang, Huanqing:
Level Order Linearization for the Elf Approach
Master’s Thesis, University of Magdeburg, 2020.

Abstract

With the progress of the times, the amount of data is getting bigger, the development of database systems is very rapid. Limited by the capacity of the main memory, in the past people used hard disks as the main storage medium of the database system. With the improvement of hardware technology, the main memory becomes larger, and the CPU becomes faster, which provides an excellent opportunity for the development of the database system - the Main Memory Database System. It uses the main memory to store data instead of hard disk storage. This greatly improves the speed of data access, but new challenges and bottlenecks have emerged.

In the era of big data and the Main Memory Database System, the processing requirements for massive data are getting higher, especially for multi-dimensional data query processing, as analytical queries become more and more complex, the number of evaluated selection predicates for each query and table also increases. This will result in a large number of multi-column selection predicates. Therefore, how to accelerate multi-column selection predicates in main memory has become the latest challenge. At this point, the advantages of tree-based index structures, such as B-tree and B^+-tree, in queries with multi-column selection predicates become limited. To change this situation, Broneske et al. proposed the Elf, an index structure that is able to exploit the relation between several selection predicates on multiple columns in a main memory database management system. Elf features cache sensitivity, an optimized storage layout, fixed search paths, and slight data compression. Their evaluation results show that their approach has significant advantages for multi-column selection predicate queries with low combined selectivity.

With the deepening of the research, especially after the introduction of the vertical linearized Elf variants and the cutoffs pointer, the vertical linearization approach has some defects instead. For example, the vertical linearization approach will reserve space for the cutoffs pointer when constructing a Elf variant. In addition, when the cutoffs pointer is added, a gap will be generated in the data structure. On the one hand, we hope to eliminate these defects. On the other hand, we also want to evaluate the impact of other linearization methods on the performance of Elf. Therefore, we have implemented level order linearization for the Elf approach. This is the main contribution of this thesis. In addition, we conducted a comprehensive evaluation of the level order linearized Elf. The evaluation results show that the level order linearization approach not only eliminates these defects for the Elf variants, but it can further improve query performance.

Acknowledgements

I would like to thank my supervisor, Dr. -ing. David Broneske, for offering me the opportunity to write this paper at a tough time. Thanks to his very patient guidance. His every suggestion was invaluable.

I also want to thank my family, because of their support and encouragement, I can continue to move forward in my study.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Listings

List of Acronyms

BFS	Breadth First Search
CPU	Central Processing Unit
DFS	Depth First Search
DL	Dimension List
FIFO	First In First Out
FTS	Full Table Scan
I/O	Input/Output
IDE	Integrated Development Environment
LIFO	Last In First Out
LP	Level Pointer
LRU	Least Recently Used
ML	Mono List
MMDS	The Main Memory Database System
MMIS	The Main Memory Index Structure
PM	Partial Match Query Algorithm
RQ	Research Question
SIMD	Single Instruction Multiple Data

1. Introduction

In the past, the bottleneck of database system with hard disk as storage medium was the access gap between permanent disk and volatile memory [Wol19]. To reduce the impact of this bottlenecks, databases are typically optimized to speed-up loading data into main memory. However, with the development of main-memory database systems, this optimization combination becomes irrelevant [FCP+12]. The new task of the database system engineer becomes to reduce the impact of the new bottleneck. From the Table 1.1, without considering other factors, the CPU is 167 times faster than the main memory, this means that CPUs will spend much of their time waiting for memory. Additionally, the gap between the CPU and memory speed will also increase with the development of hardware.

Component	Capacity	Latency
CPU	bytes	0.3 ns
L1 Cache	kilobytes	$\approx$ 1 ns
L2 Cache	kilobytes	3-10 ns
L3 Cache	megabytes	10-20 ns
Main Memory	gigabytes	50-100 ns
Disk	terabyte	$5 * 10^6$-10^7ns

Table 1.1: New Bottleneck-Memory Access (adapted from David's courseware)

In the process of reducing the impact of the new bottleneck, the database system engineer focus on achieving the best performance of the database operators. Database operators can be affected by hardware and workload, and the most typical database operator is the selection operator [Bro19]. Selectivity[1] is inseparable from the selection operator. In WHERE-Clause, the columns involved in different selection predicates have different selectivities. Das et al. propose to use an index structure for very low selectivities only, such as values under 2% [DYZ+15], hence, most OLAP (Online Analytical Processing) queries would never use an index structure to evaluate the selection predicates. However, this Approach neglected the possibility that

[1]Selectivity = Distinct Values / Total Number Rows

the accumulated selectivity of the multi-column selection predicates is below the 2% threshold [BKSS17]. Therefore, a new research question has arisen, how to use the combined selectivity of multi-column selection predicates to accelerate predicate evaluation. For this question, Broneske et al. proposed the main-memory index structure Elf, which implements an efficient query with multiple column selection predicates and an efficient evaluation of complex analytical queries [KBSS15].

The Elf approach not only shows high efficiency in the execution of mono-column selection predicates, but also the query performance for multi-column selection predicates is far beyond the current state-of-the-art-approaches [Bro19]. The standard build of Elf is a DFS[2] algorithm. Therefore, this linearization is vertical, which also means that the distance[3] between the data of the ancestor nodes and the child nodes is not as big as to sibling nodes.

Since the vertical linearization strategy is used when creating a standard Elf, hence, query algorithms of the standard Elf must also use recursion to exploit the distance in vertical linearization. Although selection predicate does not always start from the first column, the query algorithm for the standard Elf must still traverse from the root node. Additionally, the conceptual design of the standard Elf dictates that we have to traverse down to the leaf levels of the Elf to reach TIDs, mono-column selection predicates will suffer high-performance penalties from this case [Bro19]. Although the introduction of CUTOFFs optimizes these workloads, this optimization poses further challenges, because not all levels of the Elf need to be equipped with CUTOFFs (by incurring additional storage space consumption), and moreover, the Seg-Tree, FAST, ART, VAST use horizontal vectorization in search operation for a single (broadcasted) search the key in the index structure showing significant performance improvements [Bro19]. Therefore, it is necessary to adopt different linearization strategies (i.e., level order linearization) to expand Elf.

1.1 Goal of this Thesis

The goal of this thesis is to achieve level order linearization for all Elf variants and to evaluate them. The purpose of the evaluation is to verify whether the Elf approach can benefit from level order linearization. Combining the following Research Question (RQ), we introduce the key contributions we have made within the framework of this thesis:

RQ 1: How to implement level order linearization for the Elf Approach?

Based on the analysis of the build method of the vertical linearization, the level order linearization for Elf approach is designed and implemented. We will analyze the advantages of linearization for Elf in Chapter 2. Level order linearization will retain this advantage, but change the process of data processing.

[2]DFS is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

[3]The distance here is the distance between the positions of elements in the linearized ELF data space, or can also be considered as the distance between offsets.

RQ 2: What is the difference between the Partial Match algorithm adapted to level order linearization and the standard Partial Match algorithm?

According to the characteristics of level order linearization, we designed a Partial Match algorithm with new characteristics, redesigned and adjusted the relevant content of MonoLists. This will be analyzed and discussed in Chapter 3.

RQ 3: How will the level order linearization strategies affect Elf's performance?

We will evaluate level order linearization strategies through benchmark tests. In Chapter 4, We will analyze and summarize the evaluation results.

1.2 Structure of the Thesis

In order to better understand the work and contribution of this thesis. We divided the thesis into 6 chapters. In the following we brief summarize each chapter:

Chapter 2 - Background

In this chapter, we introduce the conceptual model and design model of standard Elf. Through the intuitive understanding of vertical linearization, it will be more helpful to understand the work of level order linearization.

Chapter 3 - Implementation

In this chapter, we introduce the implementation of level order linearization and the corresponding partial-match algorithm. Additionally, we will discuss the Elf variables and separation attributes mentioned in the second chapter.

Chapter 4 - Evaluation

In this chapter, we will benchmark different linearization strategies. Based on the evaluation results, we will summarize the impact of different linearization on the performance of Elf, in order to verify the primary goal of our paper.

Chapter 5 - Related Work

In this chapter, we will introduce the work related to horizontal linearization (or horizontal vectorization). The typical tree structure (e.g., CSB-Tree [RR00]) in the index structure is summarized and compared with Elf's horizontal linearization.

Chapter 6 - Conclusion

This chapter will summarize the work of the thesis. In addition, we will introduce some work that needs to be done in the future. The completion of these tasks will make the level order linearization strategy perfect so that Elf can benefit from it.

2. Background

In order to answer the presented questions and understand them fundamentally, this chapter aims to create background knowledge and requirements. Before we introduce the work of level order linearization for Elf approach, it is necessary to understand the background knowledge of the standard Elf, such as conceptual model and implementation method. For this purpose, We will first briefly introduce the index structure.

2.1 Index Structure

For the purpose of achieving high performance in a database management system, one of the approaches is to store the database in main memory rather than on hard disk [LC85]. Although the performance of the database management system has been greatly improved, there are still new bottlenecks (i.e., the gap between the CPU and memory speed). Therefore, design new data structures (i.e., index structures) and algorithms oriented towards making efficient to use CPU cycles and memory space, this approach is more effective than minimizing disk access and disk space usage [LC85].

To understand the principles and characteristics of the index structure, in this section, we first introduce the two ways for the database system to access data and some restrictions, so as to understand the principle of the index and the way to access the index. Then, this thesis will introduce selection predicates, which include mono-column selection predicates and multi-column selection predicates, and extends to four types of query. In addition, it also introduces the basic challenges posed by multiple column selection predicates.

2.1.1 Ways to Access Data

The first step to understand an index structure is to know how the database system accesses the data in the table. From a logical point of view, there are essentially two ways to access data in database tables: Full Table Scan and with the help of a new access tool - Index Structure [SSH18]. In the following, we will introduce these two access methods.

2.1.1.1 Full Table Scan

In order to achieve a full table scan, the table is stored in the memory of the database system, and then the database system sequentially traverses all the tuples in the table. If there is a selection predicate in the query statement (such as Listing 2.2), it will check whether each row meets the `WHERE` restrictions. Of course, for queries without selection predicate, this process is more obvious [Wol19]. For example, in the statement in Listing 2.1, the database will completely traverse the Lineitem table without performing any other operations.

```
SELECT *
  FROM Lineitem;
```

Listing 2.1: Lineitem-Query without Selection

```
SELECT *
  FROM Lineitem
 WHERE l_returnflag = 1;
```

Listing 2.2: Query with Selection Criterion

Before the emergence of the main storage database system, I/O was one of the important factors that affected the efficiency of full table scans. In Oracle, the evaluation of I/O is for blocks instead of rows. Because if each I/O operation processes a small amount of data, for a full table scan, the total number of I/O will increase significantly, which affects the performance of the system, especially for queries with selection predicates, the processing speed will be slower. If I/O can read multiple database blocks at once, instead of reading only one data block, that is, Multiple Block Reads, which will greatly reduce the total number of I/O and improves the system throughput. Therefore, the method of multi-block reading can efficiently achieve full table scan [Xu05]. In the past, multi-block read operations can only be used in the case of full table scans, but the index fast full scan (IFFS), which were introduced in Oracle 7.3 [Bur01], is also a type of multi-block reads.

In general, when query in a data table with a large amount of data, since a full table scan always traverses the entire data table, using index access seems to be a better way. From a cost perspective, the time complexity of a full table scan is $O(n)$, as the size of the table becomes larger, the cost of a full table scan increases linearly. In addition, the cost of an index scan is $O(Matches)$+index traversal cost. However, it is not intuitive to directly compare these two costs, hence, we used experiments from Das et al. [DYZ+15] and Lin [Cha01] to analyze the differences between them.

1. An Experiment in SQL Server

Different from Oracle, SQL Server not only has a full table scan and index scan, but it also has index seek. An index seek is performed when we search for the specific record through a column that has an non-clustered index defined on the table. Lin used a query plan with the same selection predicate to test the performance of the full table scan and index access in the SQL server. The clustered index[1] and non-

[1]A clustered index is a special type of index that reorders the way records in the table are physically stored.

clustered index[2] were used as experimental variables to achieve the purpose of use index scan and index seek respectively to compare with full table scan.

Experimental data shows that the percentage of rows returned has a great impact on retrieval performance. According to Lin's thesis, if there is a mass of data to be fetched, such as more than 5%-10% (full table scan compared with non-clustered index access in Table 2.1) or 95% (full table scan compared with clustered index access Table 2.2) of the total amount [Cha01], or the parallel query function is to be used, using a full table scan in a larger data table is more efficient than index access.

PMR	3%	4%	5%	6%	7%	8%	9%	10%
IDX Cost(ms)	5248	6704	8186	9888	11632	13360	14528	16272
FTS Cost(ms)	8432	8432	8432	8432	8432	8432	8432	8432
Method	IDX	IDX	IDX	FTS	FTS	FTS	FTS	FTS

PMR=The Percentage of Matched Rows; IDX=Index Access; FTS=Full Table Scan

Table 2.1: Full Table Scan and Non-Clustered Index Access cost estimation comparison table (adapted from [Cha01])

PMR	91%	92%	93%	94%	95%	96%	97%	98%
IDX Cost(ms)	8112	8208	8304	8400	8464	8560	8656	8752
FTS Cost(ms)	8432	8432	8432	8432	8432	8432	8432	8432
Method	IDX	IDX	IDX	IDX	FTS	FTS	FTS	FTS

Table 2.2: Full Table Scan and Clustered Index Access cost estimation comparison table (adapted from [Cha01])

2. An Experiment in Oracle

Das et al. introduced the experiment of Single Table Query in their thesis *Query Optimization in Oracle 12c Database In-Memory*. They created an on-disk table and an in-memory table respectively, and the structure and content of the two tables are exactly the same. Then they create a B-tree index on the same column of these two tables. Finally set the variables and tested all possible combinations. The results are shown in Figure 2.1, this figure shows the performance of the index access path, on-disk full FTS, and in-memory FTS plans for 5 projected columns at various selectivities [DYZ+15]. We can observe that when the selectivity is greater than 25%, compared with the full table scan on the on-disk table, the efficiency of the index method begins to decrease.

[2] A non-clustered index is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk. The leaf node of a non-clustered index does not consist of the data pages. Instead, the leaf nodes contain index rows.

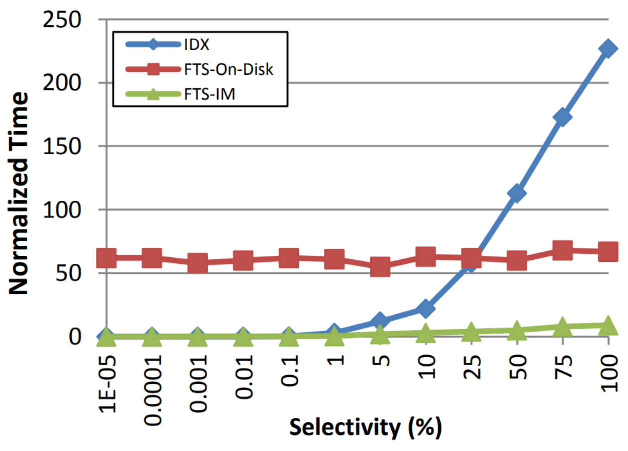


Figure 2.1: Single Table Query with 5 Projected Columns (from [DYZ+15])

With the help of these two experiments, we find that when the percentage of rows returned or selectivity exceeds a certain percentage, the efficiency of index access will be reduced. In this case, the performance of the full table scan is outstanding.

2.1.1.2 Index Access

The full table scan is oriented to the data in the table, while the index access is oriented to the index. In a relational database, an index is a separate, physically stored data structure that sorts the values of one or more columns in a database table. It is a collection of one or several column values in a table and a list of logical pointers. These pointers point to data pages that physically identify these values in the table. In simple terms, indexes are data structures that help database systems efficiently obtain data.

Common index types include bitmap index, dense index [GM08], sparse index, reverse index, primary index and secondary index. Here, we focus on the primary index and the secondary index, which helps to understand the content of this section. The index consists of the search key K, and at least one data record or reference to it ($K\uparrow$) is assigned to this search key. If this mapping is bijective, then K happens to point to a $K\uparrow$, which is called the primary index. If K points to one or more $K\uparrow$ is called the secondary index [SSH18]. The biggest difference between the primary index and the secondary index is that the leaf node of the primary index stores the entire row of data, and the leaf node of the secondary index stores the value of the primary key. Additionally, the primary index is also called clustered index[1] and the secondary index is also called non-clustered index[2].

All these indexes can be implemented using a variety of data structures. Classic data structures include Balanced trees, B$^+$-trees and Hashes [Pow06], such as both primary and secondary indexes can be organized into a B-tree or B$^+$-tree structure. These structures are designed to access data more efficiently, because the traversal cost of the tree-based index structure is usually $O(log(n))$, and the ways to access data with the help of these index structures are called index access. Index access includes index seek and index scan.

Index seek is a method that exists in SQL Server to access data by index. When search criterion matches an index well enough that the index can navigate directly to a particular point in data, that's called an index seek [Jav17]. When we specify a condition in `WHERE` clause like searching a `lineitem` by `l_returnflag` or `tid` if we have a respective index. For example, the query in Listing 2.2 will use an index seek when we run this on SQL server. In this case, the query optimizer can use the index to go directly to where `l_returnflag = 1` and retrieve the data. However, from Table 2.1, we can find that the index seek also has limitations, that is, the percentage of the number of rows returned cannot exceed 5%, and under some special conditions, it will reach 10%. Therefore, when the size of the matched data is large, a full table scan or index scan will be used. Additionally, according to Figure 2.1, in Oracle, the method of accessing data is selected based on the selectivity and the location of the table.

Index scan means that the database manager accesses the index before accessing the base table and shrinks the collection of qualified rows by scanning the rows in the specified range in the index. If the table has an index and we are firing a query which needs all or most of the rows i.e. query without `WHERE` or `HAVING` clause (e.g., Listing 2.1), then it uses an index scan [Jav17]. It works similar to the table scan, but it is slightly faster than a table scan. The main reason is that index scan look at sorted data and query optimizer know when to stop and look for another range. Excluding the effect of the amount of data, it can be seen from Figure 2.1 that index scans are suitable for a lower selectivity.

```
SELECT * FROM Lineitem
 WHERE l_shipdate >= [DATE] AND l_shipdate < [DATE] + "1Year"
   AND l_discount between [DISCOUNT] - 0.01 AND [DISCOUNT] + 0.01
   AND l_quantity < [QUANTITY]
```

Listing 2.3: Query with several Selection Criterion(TPC-H Query `Q6`)

However, in the experiment of Das et al., selectivity refers to the selectivity of a single predicate. For example, in Listing 2.3, `l_shipdate`, `l_discount`, and `l_quantity` are all in a query plan, but their selectivity is considered separately. What are the potential challenges and how to find opportunities from the challenges, we will explain the answers to these questions by introducing selection predicates in the following section.

2.1.2 Selection Predicate and Types of Query

To filter out the data that meets the requirement from a large amount of data, there are usually one or several filtering predicates in the query statement. These predicates involve a mono-column or multiple-columns. The combination of the number of columns involved in these selection predicates and the type of predicate constitute different types of query.

2.1.2.1 Query with the Selection Predicate

A query with the selection predicate always involve one column or multiple columns, for each of these columns, there is one of the following basic predicates given: $=$, $<$,

$\leq$, $>$, $\geq$, `BETWEEN` [BKSS17]. In Table 2.3, all possible integer attribute predicates and the range or interval they represent are listed (max and min are the maximum and minimum values of the attribute value range). Using this predicate translation can help to understand the upper and lower boundaries of a selection predicate, and can more intuitively understand the query algorithm. For example, $col = x$, where x is both the upper boundary and the lower boundary, we can convert it to $[x, x]$. Therefore, for column col,x is a scalar.

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 $x \leq y$	$[x, y]$

Table 2.3: Columnar selection predicate translation (from [BKSS17])

After a brief introduction to the basic knowledge about predicates, the following will specifically introduce the multi-column selection predicate and the basic challenges it brings.

1. Multi-column selection predicates

The number of columns involved in a query plan with the `WHERE`-clause determines the type of selection predicate. For example, the query in Listing 2.3 is a typical TPC-H query involving several column predicates. We name such a collection of predicates on several columns in the `WHERE`-clause a multi-column selection predicate. The execution of a mono-column selection predicate is relatively simple because it only involves one column or one dimension. Correspondingly, the database system will only a scan single column or one-dimensional index. However, multi-column selection predicates can be particularly complex, such as in Listing 2.4 is the part of a TPC-H query `Q19` given. Although both of `Q6` and `Q19` are multi-column selection predicates, this query `Q19` involve more columns than `Q6` (Listing 2.3), which is also a typical feature of complex analytical queries, that is, if the analytical query becomes more complex, the number of selection predicates to be evaluated for each query also increases, which will result in numerous multi-column selection predicates [BKSS17].

```
SELECT SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE
  FROM LINEITEM, PART
 WHERE (P_PARTKEY = L_PARTKEY
   AND  P_BRAND = "Brand#12"
   AND  P_CONTAINER IN ("SM CASE", "SM BOX", "SM PACK", "SM PKG")
   AND  L_QUANTITY >= 1 AND L_QUANTITY <= 1 + 10
   AND  P_SIZE BETWEEN 1 AND 5
   AND  L_SHIPMODE IN ("AIR", "AIR REG")
   AND  L_SHIPINSTRUCT = "DELIVER IN PERSON")
```

Listing 2.4: Discounted Revenue (Part of `Q19`)

2. The basic challenge of multi-column selection predicate

For a main memory database system, if the data table can be completely stored in the main memory, since the I/O bottleneck between the main memory and the disk is eliminated, the overhead of the full table scan will be greatly reduced. In addition, from Figure 2.1 we know that selectivity is an important factor that affects whether the database system uses a full table scan or an index structure. Therefore, in the main memory database system, the selectivity threshold required to use an index structure will be very low, or even lower than the selectivity threshold of the disk-based database system [BKSS17]. This also conforms to the point made by Das et al. in their thesis, that is, the index structure is used only for very low selectivity, such as values under 2% [DYZ+15]. This means that the chance of using an index structure is greatly reduced, and even most OLAP queries would never use an index structure to evaluate selection predicates.

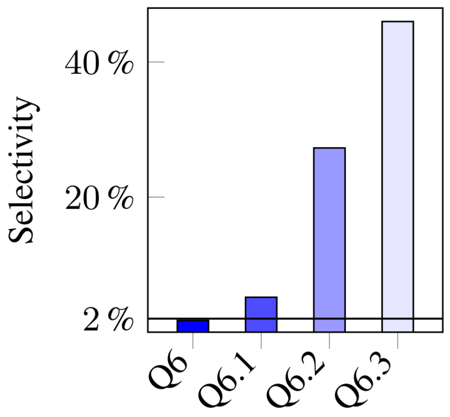


Figure 2.2: The Selectivity of TPC-H query Q6 and its predicates Q6.1 - Q6.3 on Lineitem table scale factor 100 (from [BKSS17])

David et al. analyzed the difference between the selectivity of single-predicate and accumulative selectivity of a query by TPC-H query `Q6` in their thesis *Accelerating multi-column selection predicates in main-memory – the Elf approach*. They marked the selection predicates of `l_shipdate`, `l_discount`, and `l_quantity` as Q6.1, Q6.2 and Q6.3 respectively. It can be observed from Figure 2.2 that the selectivity of each single predicate is greater than 2%, and the selectivity of Q6.3 even exceeds 40%. In this case, according to the point of view proposed in the thesis by Das et al., a full table scan will be used. However, if considering the accumulated selectivity of multi-column selection predicates, that is, accumulated selectivity of `Q6`, this value will be as low as 1.72% (less than 2%).

This also leads to the basic challenge of multi-column selection predicates. The selectivity of the overall query is often small, but the selectivity of the column involved in each single predicate is high enough to enable the database system to decide to use a scan for all columns. Therefore, we cannot use an index to complete the query. As a result, a common way is to use optimized column scans [LP13] [SK13]. Based on this situation, an index structure would be favored if it could exploit the relation between all selection predicates of the query [BKSS17]. The realization of this multi-dimensional index structure (i.e., Elf) is the main contribution of David et al.

and is also the basis of the work of this thesis. Before introducing it, we will briefly introduce how to classify queries to understand the principle of the partial-match query.

2.1.2.2 Query Type

Whether it is a mono-column selection predicate or a multi-column selection predicate, they are an essential part of the query. We can use their different combinations to form different query types. Understanding these query types is helpful to understand the query-related work of this thesis. According to different criteria, such as a number of search keys and type of predicate, the query can be divided into exact-match query, partial-match query, range query and partial range query. For example, when the number of search keys is equal to the number of columns, and the selection predicates are $col = x$, it is an exact query. In order to better understand these query types, we assume that there is a table with k columns, one or more selections can be used to specify qualification conditions for up to k search keys in a query. If key values meet these requirements, they can be included in the result set of Q. Hence, the types of these queries can be explained by the following numerical attributes [HR13]:

1. **Exact Match Query**: It specifies a value for each key.

   $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge ... \wedge (A_k = a_k)$

2. **Partial Match Query**: It specifies s key values, which $s < k$.

   $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge ... \wedge (A_s = a_s)$, with $s < k$

3. **Range Query**: It specifies a range $r_i = [l_i \leq a_i \leq u_i]$ for each key A_i

   $$\begin{aligned} Q &= (A_1 \in r_1) \wedge (A_2 \in r_2) \wedge ... \wedge (A_k \in r_k) \\ &\equiv (A_1 \geq l_1) \wedge (A_1 \leq u_1) \wedge ... \wedge (A_k \geq l_k) \wedge (A_k \leq u_k) \end{aligned}$$

4. **Partial Range Query**: It specifies a range for s keys, which $s < k$.

   $Q = (A_1 \in r_1) \wedge (A_2 \in r_2) \wedge ... \wedge (A_s \in r_s)$, with $s < k$

If we define an exact range $a_i \in [l_i, u_i]$ and an unlimited range $a_i \in (-\infty, \infty)$ in the query, all four types of queries can be regarded as general range queries [HR13]. Especially when $s = k$, all partial queries can be regarded as exact queries.

2.1.3 One-Dimensional Index and Multi-Dimensional Index

Generate an index on an attribute in the relationship and use it in the query, this index is a one-dimensional index. Similarly, the index generated for n attributes in the relationship is a multi-dimensional index. A multi-dimensional index can be a one-dimensional index ($n = 1$). The one-dimensional index can also be executed multiple times to complete the work of the multi-dimensional index. But the efficiency is far lower than the multi-dimensional index. We will introduce this in detail below. In this thesis, we refer to query statements involve one attribute such as Listing 2.2 as one-dimensional queries, and query statements involve at least two attributes are collectively referred to as multi-dimensional queries. In order to better explain, in Table 2.4 we adapted the `Lineitem` table, only set four attributes, `LID` as a unique identifier, the range is $[L_0, L_9]$, the range of `l_quantity` is still $[1, 10]$, the `l_shipdate` is $[2010, 2020]$, the range of `l_returnflag` is $[0, 1]$, K_{2dim} uses arrays to represent the simplified relationship between attribute `l_quantity` and attribute `l_shipdate`.

LID	l_quantity	l_shipdate	l_returnflag	K_{2dim}
L_3	1	2015	0	(1, 2015)
L_7	2	2011	0	(2, 2011)
L_0	2	2018	0	(2, 2018)
L_4	4	2016	0	(4, 2016)
L_1	4	2019	0	(4, 2019)
L_8	5	2011	1	(5, 2011)
L_5	5	2015	1	(5, 2015)
L_2	8	2018	1	(8, 2018)
L_9	9	2012	1	(9, 2012)
L_6	9	2015	1	(9, 2015)

Table 2.4: Table adapted from `Lineitem` table based on TPC-H benchmark

2.1.3.1 One-Dimensional Index

First, we introduce the one-dimensional index. As mentioned in the previous chapter, common data structures such as B-tree and B$^+$-tree can be used to implement a one-dimensional index, and the index is ordered. If we build an index for the flag attribute in Listing 2.2, all rows with `l_returnflag` $= 1$ in the table are arranged sequentially due to the order of the index. After a search for the first position of `l_returnflag` $= 1$ in the index, the `LID` can be obtained sequentially to the end, thereby avoid full table scans and improve query efficiency.

```
SELECT *
  FROM Lineitem
 WHERE l_shipdate = 2011;
```

Listing 2.5: Partial Match Query with One-Dimensional Index

```
SELECT *
  FROM Lineitem
 WHERE l_returnflag = 1
   AND l_shipdate = 2011;
```

Listing 2.6: Exact Match Query with Multi-Dimensional Index

Although a one-dimensional index is built for `l_returnflag`, the use of this one-dimensional index structure for other one-dimensional queries such as Listing 2.5 will not improve efficiency. Such queries will still use full table scans or index scans. The reason is that the `l_returnflag` is ordered in the index structure. If it is mapped to the table, it means that the rows will be reordered in the order of `l_returnflag`, so other attributes depend on the `l_returnflag`. As can be observed from Table 2.4, `l_shipdate` is out of order. Therefore, the advantage of the one-dimensional index is not reflected in the one-dimensional query such as Listing 2.5, unless the attribute `l_shipdate` also has its own index.

For multi-dimensional queries, use `l_returnflag` and other attributes to form a multi-dimensional query such as Listing 2.6, this index structure can be used to effectively implement such queries. However, for multi-dimensional queries without `l_returnflag` such as Listing 2.7, it is the same as Listing 2.5, all of them cannot benefit from a one-dimensional index structure (`l_returnflag`). This means that for any type of query that contains indexed attributes (i.e., Listing 2.2, Listing 2.6), its one-dimensional index structure can efficiently obtain results. In contrast, other query plans will still use full table scans. Therefore, whether the attribute in the query statement is indexed will affect the execution efficiency of the query.

```
SELECT *
  FROM Lineitem
 WHERE l_quantity >= 4 AND l_quantity <= 7
   AND l_shipdate >= 2014 AND l_shipdate <= 2017;
```

Listing 2.7: Range Query based on Table 2.4 (multi-dimensional index)

However, if we build their own one-dimensional index for each attribute involved in a query, the situation will be different for one-dimensional queries and multi-dimensional queries. For a one-dimensional query such as Listing 2.2 and Listing 2.5, results can be obtained from their respective index structures, thereby improve query efficiency. For multi-dimensional query (i.e., Listing 2.3, Listing 2.6, Listing 2.7), it needs to retrieve data from the index structure of each attribute. Then calculating the intersection of all returned result sets to get the final result set.

For example, for Listing 2.7, if we create separate one-dimensional indexes for `l_quantity` and `l_shipdate`, and assume that the sets returned by each attribute are $|S_1| = m$, $|S_2| = n$, then the matched data can be obtained by calculating their intersection, so their time complexity is $O(m * n)$. For a query with n dimensions, use $|S_n| = M_n$ to represent the set returned by each attribute. The time complexity can be expressed as $O(M_1 * M_2 * ... * M_n)$, as the dimension increases, this value will increase significantly, which also means that the efficiency of this method decreases. In addition, using the traditional tree structure (i.e., B-tree, B$^+$-tree) or hash table as a data structure for store two-dimensional or multi-dimensional indexes, compared with store one-dimensional indexes, its performance will be significantly reduced [LY10]. Therefore, it is necessary to introduce a data structure that can implement a multi-dimensional index.

2.1.3.2 Multi-Dimensional Index

An index generated for n attributes in the relationship is a multi-dimensional index. Common data structures used to implement multi-dimensional indexes include

Grid-Files (hash), K-d tree, k-dB tree, R-tree and UB-tree. The biggest difference between the multi-dimensional index and one-dimensional index is that the process of multi-dimensional data in multi-dimensional index structure spans multi-dimensional space. For example, based on the multi-dimensional query Listing 2.7 and Table 2.4, we can map `l_quantity` and `l_shipdate` to the two-dimensional space plane shown in Figure 2.3. In this two-dimensional plane, the abscissa is `l_quantity` $\in [0, 10]$, the ordinate is `l_shipdate` $\in [2010, 2020]$. Their coordinate combination contains all the rows in Table 2.4, for which we use grid points to represent and map to `LID` (i.e., L_3(1, 2015)).

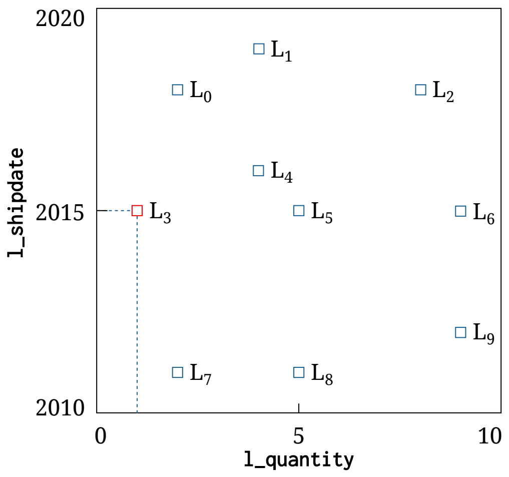


Figure 2.3: Two-dimensional plane on Table 2.4 and attributes contained in Listing 2.7

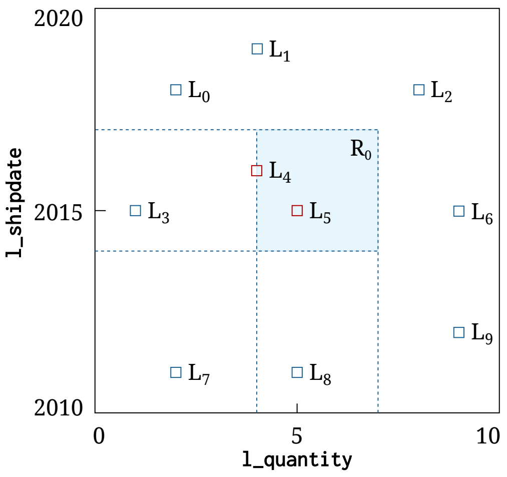


Figure 2.4: Range Query in Two Dimensional plane on Listing 2.7 and Figure 2.3

After converting the two attribute columns `l_quantity` and `l_shipdate` into a two-dimensional plane, we can display the result set of the range query Listing 2.7 in Figure 2.4 based on this two-dimensional plane. First, determine the upper and lower boundary values of each attribute in the query plan, that is, `l_quantity` $\in [4, 7]$ and `l_shipdate` $\in [2014, 2017]$, then draw a vertical line to the respective coordinate axis. All the grid points in the finally formed dark region R_0 are the result set of Listing 2.7, which contains two grid points L_4 and L_5. In Table 2.4, we use K_{2dim} to represent this array of coordinate forms, and they will also be stored in the corresponding index structure. For example, the Grid File will save each array in the bucket[3] of the corresponding area, and the tree structure such as K-d tree or k-dB tree will directly store array in child nodes or leaf nodes.

```
SELECT *
  FROM Lineitem
 WHERE l_shipdate >= 2014 AND l_shipdate <= 2017
```

Listing 2.8: One-Dimensional Range Query with Multi-Dimensional Index

[3]A bucket is a data structure that uses the key values as the indices of the buckets, and store items of the same key value in the corresponding bucket.

This way of process with two-dimensional planes is a kind of visualization, which shows how the multi-dimensional index structure associates data of different dimensions and how to query data that meet the predicate conditions. In terms of cost, for example, a k-dB tree is used to save the multi-dimensional index, for exact match, the cost is $O(logn)$, and partial-match is better than $O(n)$ [KSS14], both of them are much smaller than $O(m * n)$[4]. Therefore, the use of multi-dimensional index structure can effectively achieve multi-dimensional query, in the same way, it can also effectively implement one-dimensional query, in the plane make a vertical line on the boundary value of the corresponding attribute coordinate, the data on the vertical line (i.e., Listing 2.5) or within the range of two vertical lines (Listing 2.8) are result sets of the one-dimensional query.

2.2 Elf as Multi-Dimensional Index Structure

After understanding the advantages of index structure (especially multi-dimensional index structure), we introduced a novel multi-dimensional index structure - Elf, which is used to solve the challenge brought by multi-column selection predicates. This chapter mainly introduces the background of Elf, such as conceptual design, vertical linearization and Elf variables. On this basis, we finally introduce the core work of this thesis - level order linearization and briefly explain it.

2.2.1 Conceptual Design of Elf

Before introducing the background of Elf, it is necessary to explain the source of inspiration for Elf (i.e., Dwarf). Hence, in this chapter, we will first describe the background, construction methods and advantages of Dwarf. Then on this basis, the concept design of Elf will be introduced.

2.2.1.1 Dwarf

The Dwarf is a highly compressed data structure that is commonly used for computing, storing, and querying data cubes [SDRK02]. The main contribution of Dwarf is to reduce the storage requirements by dramatically compressing the cube without reducing the precision of the cube query.

This requirement mainly comes from the cube operator. Whether it is computing or storage, the size of the cube operator is an inevitable challenge, because the operator performs the computation of one or more aggregation functions for all possible combinations of grouping attributes [SDRK02]. The number of grouping combinations will increases exponentially as the dimension increases. Assume that there are only two dimensions a, b in a data set, its groupings are a, b, ab. If we add a dimension c, the grouping combination will be a, b, c, ab, ac, bc, abc, hence, for an n-dimensional cube, there are $2^n - 1$ grouping combinations. This also causes the size of the cube operator to increases exponentially as the dimension increases. Therefore, Simen et al. designed this data structure (i.e., Dwarf) to store a highly compressed version of the cube operator [BKSS17].

[4]This time complexity comes from the cost of using a one-dimensional index to process a multi-dimensional query in Section 2.1.3.1

The way that Dwarf highly compressing the cube is to avoid the redundancy of the cube. To be specific, it identifies prefix redundancies and suffix redundancies in the cube entries and factoring them out of the store, so as to achieve the purpose of high compression.

1. Prefix Redundancy

In a sample cube with three dimensions a, b, c, we listed all the grouping combinations in the previous paragraph. We can observe that each value of dimension a will appear in four group-bys (a, ab, ac, abc). Here, dimension a is the prefix of dimensions b and c, and each value of dimension a may appear multiple times in different groupings, then these repeated values are prefix redundancy. It can also happen when the prefix size is larger than one node, in which case the prefixes will appear in pairs. Such as (a, b) will appear in group-bys ab and abc. In simple terms, prefix redundancy occurs when two or more dimension keys share a common prefix [BKSS17].

2. Suffix Redundancy

Use the same sample cubes, the suffix redundancy is like abc and bc. Suppose there is a value b_i in dimension b, which appears in the fact table only with the value a_i in dimension a, that is, (a_i, b_i). Then for any value m in dimension c, groups $\langle a_i, b_i, m\rangle$ and $\langle b_i, m\rangle$ always have the same value. The reason is that for $\langle b_i, m\rangle$, all values of dimension a are aggregated. Since b_i only appears with a_i, the value in dimension a here is just the value a_i. In simple terms, suffix redundancy occurs when two or more group-bys share a common suffix [SDRK02].

Sismanis et al. used this way to compress a Petabyte cube with up to 25 dimensions, and finally they got a Dwarf Cube with only 2.3 GB, the high compression rates reached 1: 400000. However, even with such dramatically compression, Dwarf still maintains 100% precision of the cube query [SDRK02]. To introduce Dwarf Cube, in the following we refer to the example fact table (Table 2.5) from the thesis of Simen et al. to show how to compress this fact table into a Dwarf cube.

Filiale	Customer	Product	Price
F1	C2	P2	\$ 70
F1	C3	P1	\$ 40
F2	C1	P1	\$ 90
F2	C1	P2	\$ 50

Table 2.5: Fact Table for cube Sales (adapted from [SDRK02])

Table 2.5 contains four attributes. We will grouping attributes Filiale, Customer, and Product, then perform the aggregation operation `SUM` on the fact Price in the corresponding group-by. For example, with the value *F1* in the attribute Filiale as a prefix, the corresponding group-bys are $\langle F1, C2, P2\rangle$, $\langle F1, C3, P1\rangle$, $\langle F1, C2\rangle$, $\langle F1, C3\rangle$, $\langle F1, P2\rangle$, $\langle F1, P1\rangle$, $\langle F1\rangle$. For $\langle F1, C2, P2\rangle$ and $\langle F1, C2\rangle$, their combination can also be regarded as group-bys, which prefixed by the dimension pair ($F1$, $C2$). Hence, for $\langle F1, C2\rangle$ mean that all values of the dimension Product is aggregated, which is reflected in the `SUM` computation of the Price. Here we use a special value `All` to replace the aggregated attributes after grouping, hence, $\langle F1, C2\rangle$ is equivalent to $\langle F1, C2, All\rangle$. We display all group-bys prefixed with F1 in Table 2.6 in this form. Since the value pair $(F1,\ C2)$ only appears once in Table 2.5, hence, in Table 2.6, $\langle F1, C2, P2\rangle$ and $\langle F1, C2, All\rangle$ have the same value \$70 in the fact Price. Otherwise, the aggregation operation `SUM` will be performed. For example, for $\langle F1, All, All\rangle$, all tuples with F1 in Table 2.5 will be aggregated, and the fact Price will be added to get \$110 (i.e., \$70 + \$40).

Filiale	Customer	Product	Price
F1	C2	P2	\$ 70
F1	C3	P1	\$ 40
F1	C2	`All`	\$ 70
F1	C3	`All`	\$ 40
F1	`All`	P2	\$ 70
F1	`All`	P1	\$ 40
F1	`All`	`All`	\$ 110

Table 2.6: Cube Operator in Dwarf base on value *F1* in Table 2.5 as the prefix

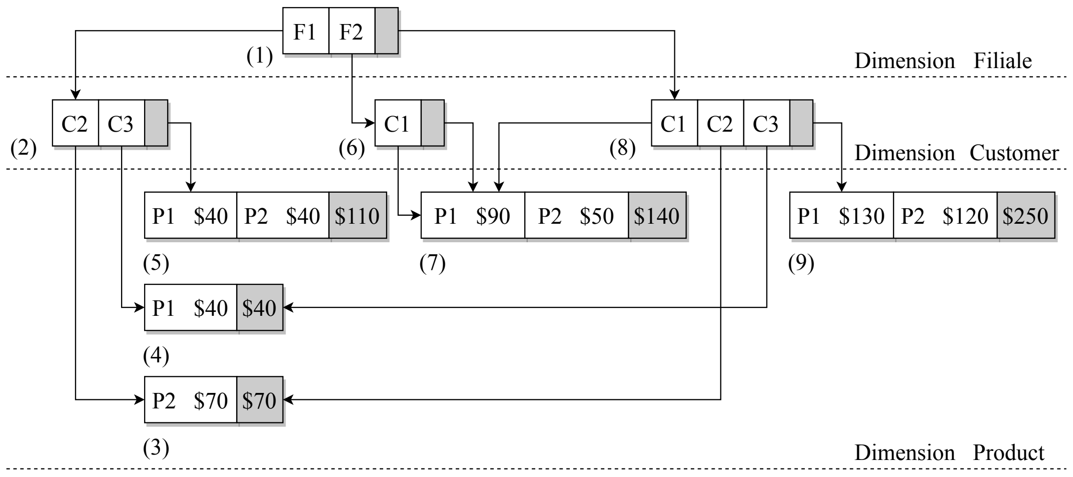


Figure 2.5: Dwarf Operator base on Table 2.5

Figure 2.5 shows the Dwarf Cube for the table shown in Table 2.5. It can be observed from Figure 2.5 that the height of Dwarf is the number of dimensions in Table 2.5, and the value of Price is stored in each leaf node. Each node consists of several cells. For non-leaf nodes, in addition to the cell representing `All` (small and dark rectangles in Figure 2.5), each cell of each node contains one for each distinct value of corresponding dimension and a pointer. We use the form $[key, pointer]$ to represent a cell. The pointer of the cell points to a node, which contains all the distinct values of the next dimension associated with the key of this cell. For example, the node(2) pointed to by the cell with $key = F1$ contains two key values $(C2, C3)$, both of which are related to $F1$. The cell `All` contains only one pointer, and the node pointed to by this pointer contains all the different values of the next dimension (e.g., Node(8)). For the cell of the leaf node, we use the form $[key, aggregate]$ to represent. This means that each cell of the leaf node holds a distinct value of the last dimension and a corresponding aggregate value. For example, $\langle F1, C2, P2\rangle$ in Table 2.6 corresponds to the path $\langle(1),(2),(3)\rangle$ in Figure 2.5, and $\langle F1, C2, All\rangle$ also corresponds to the path $\langle(1),(2),(3)\rangle$, but the cells in node(3) they access are different.

Dwarf associates values of different dimensions through pointers, and eliminates prefix redundancy at non-leaf nodes, especially at the root node. In addition, suffix redundancy is recognized and processed during the construction of Dwarf [SDRK02]. After analyzing the Dwarf cube, we can summarize the advantages of Dwarf into four points, three of which are the main factors for Elf to choose Dwarf as the basis.

1. **Prefix redundancy elimination**

   Each node contains different values in the corresponding dimension. Especially the first dimension contains all the different values in the first dimension, and each value appears only once in this node. Nodes in other dimensions only contain the dimension values associated with the prefix *key* value, and there are also no duplicate values in the nodes. This also means that along the path from the root node to the leaf node, the dimension value of the prefix is only saved once.

2. **Suffix redundancy elimination**

   In Dwarf, when two or more nodes of a dimension share a common suffix, the pointers of these nodes will point to the same node in the next dimension. This suffix node appears only once in Dwarf.

3. **Ordered node elements**

   In a node of Dwarf, all elements are arranged in order. Using this order can improve the efficiency of the query. And this also corresponds to the order of the index.

4. **Fixed search depth**

   The height of a completed Dwarf is equal to the number of dimensions, which means that the path length from the root node to the leaf node is fixed. For a query, a dimension can be better exploited instead of a tuple.

2.2.1.2 Elf

In order to accelerate the multi-column selection predicate in the main memory, the index structure can be used on the multi-column selection predicate with low cumulative selectivity. Inspired by Data Dwarf, Broneske et al. implemented a novel multi-dimensional index structure for order-preserving dictionary-compressed data or numeric data, that is, Elf, which is a tree structure, and the index structure it constructs can effectively use the relationship between several selection predicates in a query [BKSS17], this means that for the main memory database system, optimized accelerated scan is no longer the only option. In addition, the emergence of Elf has greatly improved the selectivity threshold for using index structures, with 18% instead of 2% (proposed by Das et al.), which greatly increases the opportunity of using index structures in a query [DYZ$^+$15, BKSS17].

In the following, we first convert the Dwarf table (Table 2.5) and cube (Figure 2.5) into an Elf form to compare the differences between Dwarf and Elf and introduce the conceptual design of Elf. Then, we introduce the specific construction process of Elf through a simple example.

1. Compare Dwarf and conceptual design of Elf

Due to one of the advantages of Dwarf, that is, the elimination of suffix redundancy, it allows the suffix node in Dwarf to have multiple direct precursor pointers, so Dwarf is a graph structure. The `All` cell is the main factor that causes multiple direct pointers to a node.

For Elf, as a multi-dimensional index structure, it will not store any facts (that is, column Price in Table 2.5), and because it is a tree structure, its non-root node will only have a precursor pointer. Therefore, compared to Dwarf, Elf as the index structure does not retain the characteristics of All cell and suffix redundancy elimination, and because Elf stores a reference to the tuple, the fact will not appear in Elf, but a unique identifier `TID`. In Table 2.7, we will replace the fact with `TID`, and in the Elf structure, the `TID` does not appear in the form of a dimension but is stored in the leaf node as the fact.

Filiale	Customer	Product	TID
F1	C2	P2	T1
F1	C3	P1	T2
F2	C1	P1	T3
F2	C1	P2	T4

Table 2.7: Data table adapted to Elf based on Table 2.5

In Figure 2.6, we use the data in Table 2.7 to construct an Elf structure. Compared with Figure 2.5, we can observe that Elf retains three of Dwarf's advantages. The height of Elf is equal to the number of dimensions, which also means that the path length from the root node to the leaf node is fixed. In Elf, we named all non-leaf nodes `DimensionLists`, these nodes are consist of values and a pointer. These values in a single node are distinct, and in the node, these values are arranged in order. The leaf node stores the data of the last dimension and the `TID` of the tuple. For the general Elf, the final saved `TIDs` are unordered. The 1 of T1 has no practical significance and is only used as a reference for the unique identifier. In addition, since there is no `All`-cell in Elf, each level in Elf is a column of data.

2. Construction of Elf

After the introduction of the conceptual design of Elf, we show the complete process of constructing an Elf through a simple and typical example. Table 2.8 shows the four columns being indexed and a tuple identifier to uniquely identify each row. Table 2.9 is the final table after building Elf. During the construction of Elf, the order of the tuples in Table 2.8 will also change due to the order of the elements within the nodes in Elf. Based on Table 2.8, we show the complete process of constructing Elf in Figure 2.7.

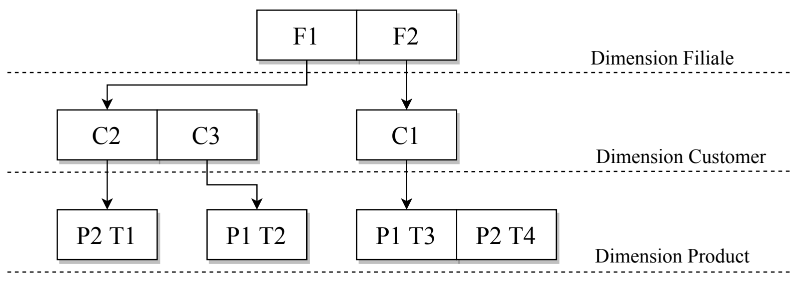


Figure 2.6: Elf Structure base on Table 2.7

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

Table 2.8: Example Table ([BKSS17])

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

Table 2.9: Sorted Example Table

First, sort the column C_i in Table 2.8, and then create node entry in the DimensionList for each distinct value of this column. When $i = 1$, this step corresponds to step I in Figure 2.7, and column C_1 contains two distinct values $(0, 1)$. Then perform logical partitioning in the sorted Table 2.9, which is prefixed with distinct values of C_1 $(0, 1)$. Then sort $C_2(C_{i+1})$, and repeat these steps recursively.

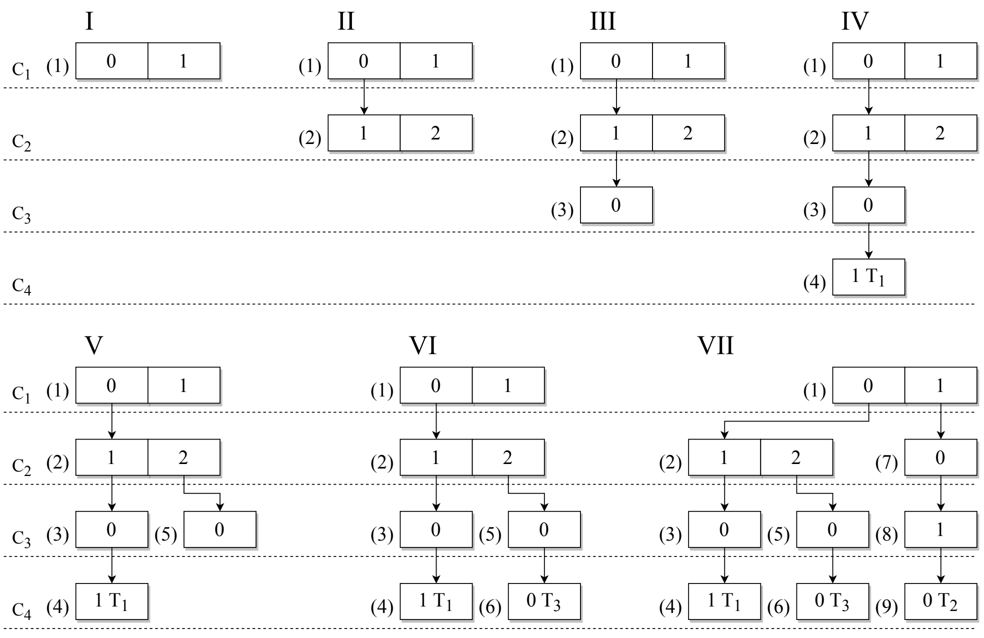


Figure 2.7: Construction Process Diagram of Elf base on Table 2.9

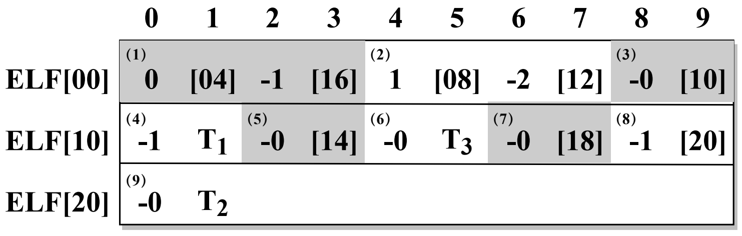
In Figure 2.7, we number each node according to the construction order. It can be observed that this order is a standard depth-first algorithm. Its characteristic is that each recursion will start from the root node (1), first construct the left subtree, first reach the left leaf node (4) in step IV, and then to the right leaf node (6) in step VI, until return to the root node (1). Then the right subtree is processed until the last leaf node (9) in VII is constructed, and every time the path from the root node to the leaf node is a complete tuple.

2.2.2 Linearization of Elf

The principle of linearization for Elf approach is an important foundation of this thesis. In this chapter, we introduce the vertical linearization for the standard Elf. The use of linearization is to enhance the query performance of OLAP, so that Elf is linearized into an array of integer values in order to use an explicit memory layout [BKSS17]. Based on Elf in Figure 2.7, we briefly introduce the memory layout of Elf, the optimization method (i.e., the introduction of MonoList) and the linearized memory layout structure after optimization.

2.2.2.1 Memory Layout based on Elf

In Figure 2.7 step VII is a complete Elf, which is constructed based on Table 2.8. Assume that column values and pointers within this Elf are 64-bit integer values [BKSS17]. We can show the linearized Elf from step VII in Figure 2.8. In the previous chapter, we collectively referred to nodes in Elf as DimensionList. In order to store nodes, we need to map each DimensionList here to an array.



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

Figure 2.8: Memory layout as an array of 64-bit integers (adapted from [BKSS17])

For example, for the element of the node (1) in Figure 2.7, which is the first DimensionList $L_{(1)}$, we map it to two integers. In the memory layout of Figure 2.8, the node (1) as the root node is stored at position 0 and contains two elements: $E_{(1)}$, the value is 0, the pointer is 04, and use the format of [04] in the figure to distinguish it from the value of the element. The second element is $E_{(2)}$, the value is 1, and the pointer is 16. In the figure, the minus sign is used to identify the value of $E_{(2)}$ to indicate that this element is the last element of the DimensionList $L_{(1)}$. The elements in a DimensionList are adjacent in the memory layout. The purpose of this design is because if a DimensionList is long enough, the number of distinct elements in it is large, then scanning this list will have a significant improvement in performance. The reason is that the elements in Elf's nodes are ordered. In the memory layout, the order and adjacent characteristics of elements can be used to speed up the scan.

Pointers at position 1 and position 3, respectively, point to the next DimensionList associated with the current element, that is, position 4 and position 16. From Figure 2.7, we can observe that 0 in $L_{(1)}$ points to the left subtree and 1 points to the right subtree. In the memory layout, it is mean that from position 4 ($L_{(2)}$) to position 16 ($L_{(7)}$), the entire left subtree is stored, and from position 16, the entire right subtree is stored. This also conforms to the construction order of Elf, that is, the order of the numbers (1 - 9) in Figure 2.7 completely corresponds to the order of the DimensionList in the memory layout ($L_{(1)}$ - $L_{(9)}$). In addition, for the last column of data (C_4), since there is no next dimension, the pointer will not be saved, but the `TID` will be stored.

2.2.2.2 Optimization Methods

Optimization 1: Use a hash map on the first DimensionList

In a standard Elf, all values in the first DimensionList must be traversed until the upper boundary of the interval defined on the first column is found. In order to solve this bottleneck and ensure that Elf is always sensitive to the column. Broneske et al. found that the three characteristics of the first DimensionList can be used to store pointers in the form of a hash map, and the dimension value is used as the hash graph value.

Denseness. Since dictionary compression of data is sequentially retained, there exist all integer values between 0 and the maximum value of the column [Bro19].

Ordering. In the node of Elf, which is DimensionList, all the element values are arranged in order.

Uniqueness. One of the advantages of the Elf structure is the elimination of prefix redundancy, which means that nodes only contain distinct values of corresponding dimensions. Only the first DimensionList contains all the distinct values of the first column.

This optimization method can be intuitively shown in Figure 2.9. The upper part is the Elf memory layout without hash map, and the lower part is the optimized Elf with hash map. The position of the dimension value is marked with red. These dimension values are used as hash map values. In other words, the position of the layout below is equal to the corresponding dimension value. If the dimension value does not exist, a null pointer can be stored at this position, indicating that the dimension value does not exist. This optimization eliminates the bottleneck of the first dimension column and only requires the original DimensionList half the space [BKSS17, Bro19]. Since other DimensionList do not save all the distinct values of the corresponding dimension, this method is only used for the first DimensionList.

Optimization 2: The introduction of MonoList

Broneske et al. found in experiments that along the path of the Elf structure, the deeper the search path, the shorter the DimensionList [BKSS17]. That is, the closer to the leaf node in Elf, the greater the probability that a DimensionList with only one element appears. They display this issue for the TPC-H Lineitem table of scale factor 100 with all 15 attributes resulting in a 15-level Elf in Figure 2.10 [BKSS17]. It can be easily observed from the Figure 2.10 that start from the dimension 11, all the DimensionList (100%) contain only one entry, and they are connected by pointers, which is more like a linked list. The prefix of the one-element DimensionList

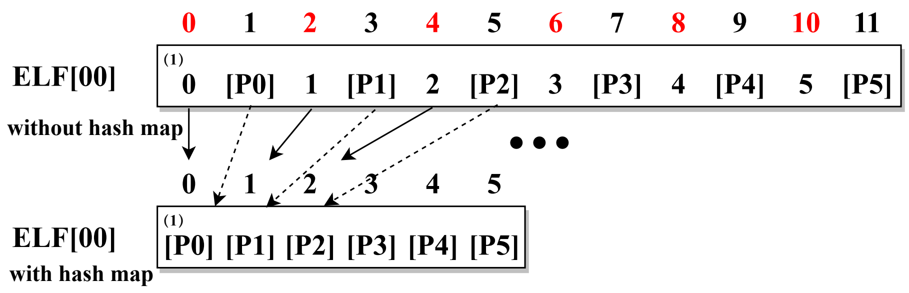


Figure 2.9: Hash-map property of the first DimensionList (adapted from [Bro19])

becomes unique, so there will be no more branches. In addition, each one-element DimensionList still needs to store a pointer, which also increases the storage space requirements of Elf.

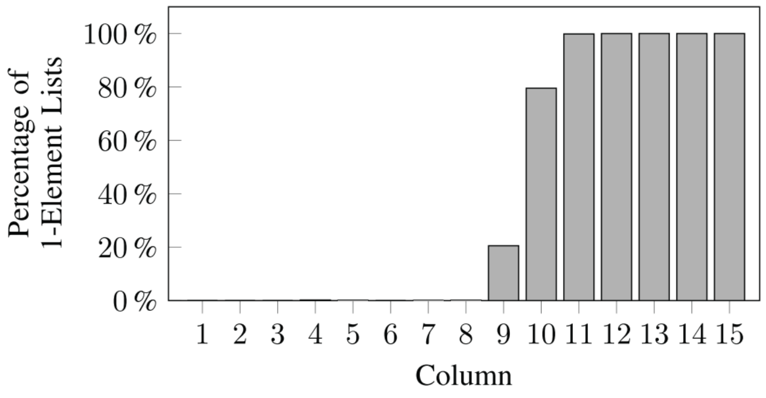


Figure 2.10: Percentage of 1-element lists per dimension for the TPC-H Lineitem table with scale factor 100 (from [BKSS17])

In order to overcome this deterioration, Broneske et al. introduced MonoList. In the absence of prefix redundancy, the column values in the tuple will be stored adjacent to each other, which similar to a row storage manner. Figure 2.11 is an Elf structure optimized based on Figure 2.7. The gray DimensionList represents MonoList. It can be observed from Figure 2.11 that the start dimension of MonoList is not fixed, for example, $L_{(3)}$ starts from C_2, which covers C_2, C_3 and C_4. $L_{(4)}$ and $L_{(5)}$ start from C_3, they cover C_3 and C_4. Compared to 9 DimensionLists in Figure 2.7, the sum of the number of DimensionLists and MonoLists in Figure 2.11 is only 5. From this perspective, the introduction of MonoList reduces the unnecessary storage overhead of Elf. To better illustrate this, we use the optimized memory layout as shown in Figure 2.12 to explain.

Figure 2.12 shows the final memory layout of the Elf approach, which combines these two optimization methods. In Figure 2.12, the first DimensionList only holds the pointer, and the dimension value is used as the hash map value, which is equal to the position number. The pointer with a minus sign in the figure indicates that the next dimension is a MonoList. In the implementation process, we represent this minus sign by setting the most significant bit for the pointer of the dimension element. Compared with Figure 2.8, the optimized memory layout for Elf approach has a

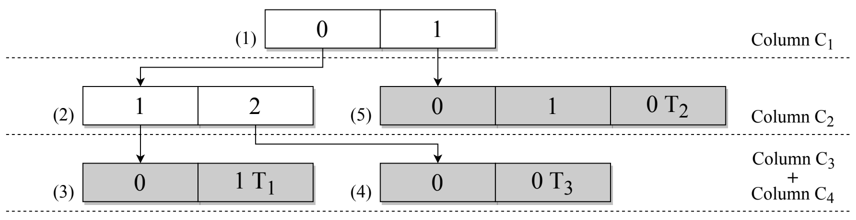


Figure 2.11: Elf with MonoLists (from [BKSS17])

significantly reduction in storage consumption, and the values in the later columns have a better adjacency.

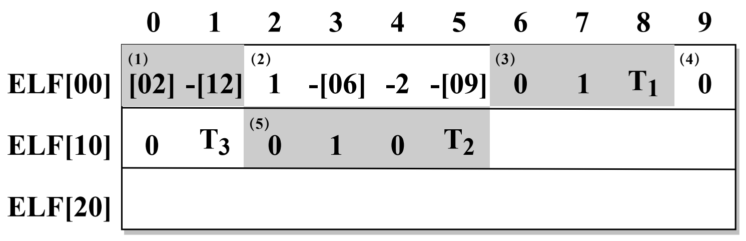


Figure 2.12: Final memory layout of the Elf approach (adapted from [BKSS17])

2.2.3 Partial Match Algorithm of Elf

In Section 2.1.2.2 four types of queries are briefly introduced. This section mainly introduces partial match algorithm for standard Elf. The partial match query can be regarded as part of the exact match query. The only difference is that the number of columns involved in the partial match query is less than the total number of columns in the table. Here, we separately discuss a single partial matching query and a group of partial matching queries.

For a single query, used the standard Elf as the index structure. Due to vertical linearization and prefix redundancy, all queries must start from the root node regardless of whether the first dimension is in the query condition.

If the query condition contains the first dimension, because the first dimension in Elf uses a hash graph, it can be directly mapped to the position of the first dimension in Elf, so as to obtain the pointer of the next dimension. If the pointer is not a null pointer, it means that the first dimension is matched successfully. If the query value is not within the hash value range of the first dimension or the pointer is a null pointer, the query does not match. If the value in the first dimension matches successfully, then use the pointer to find the position of the next dimension and repeat the above steps. Perform recursion until all conditions are matched, then return `TID`.

If the first dimension is not involved in the query condition, this means that all elements of the first dimension need to be recursively one by one until a path matches the condition is found. Or until no path matches the query condition at the end.

For a set of queries, according to the characteristics of Elf, we can find the upper and lower boundaries of the query values of the same dimension involved in these query conditions, that is, the maximum and minimum values. Use the window form to indicate that it is $[R_{min}, R_{max}]$. Because of the order of the elements in the node, after determining the upper and lower boundaries, it is possible to avoid traversing the unnecessary paths.

2.3 Elf Variants

This chapter briefly introduces several variants of Elf. The main purpose of using the variants is to support the final SIMD scanning algorithm. Before introducing the Elf variant, a new variable needs to be introduced, namely Cutoffs. Cutoffs are an additional data structure used for direct access to `TID` at an earlier time, it only saves the cutoff-pointer. These pointers will not point to the data structure of Elf but will point to a new data structure named Elf_TIDs. In this data structure, the `TIDs` of all tuples are saved. The order of `TIDs` is not the natural order in Table 2.8, but the order of `TIDs` in the new table formed after Elf is constructed, as in Table 2.9. This also means that if there is only one query condition for a partial matching query, such as Listing 2.2, it can directly obtain the `TID` through Cutoffs instead of continuing down until it reaches the leaf node to obtain the `TID`. The Cutoffs has two sides, using Cutoffs will improve query performance, but in storage consumption, Cutoffs also has some drawbacks, which will be explained in subsequent chapters.

2.3.1 Elf64

Elf64 provides a standard implementation of Elf. If it is an Elf64 without Cutoffs, its memory layout is the same as the figure, they only contain the value and the pointer of the next dimension. If it is Elf64 with Cutoffs, we need to reserve the space for cutoff-pointers for each element in the memory layout, we can illustrate this with a picture. In Figure 2.13, we used the new data structure ELF_TIDs for Elf in Figure 2.11 and added TIDs.

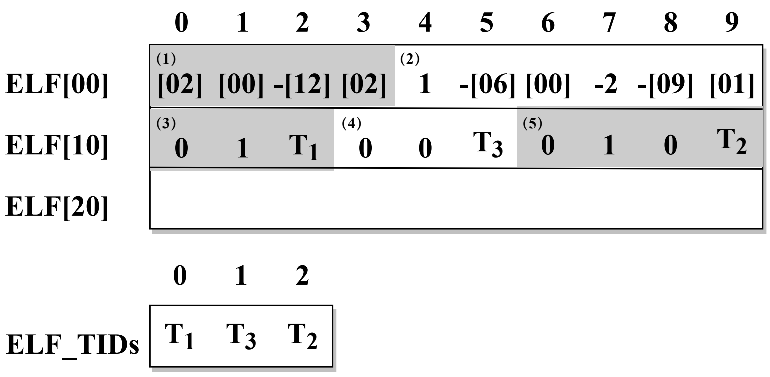


Figure 2.13: Memory layout of the Elf approach with Cutoffs

Elf64 does not use a separate CUTOFFs data structure to store the cutoff-pointers but reserves a place for it in the memory layout. As show in Figure 2.13, the positions 1, 3, 6, and 9 in the memory layout store cutoff-pointers, which point to the positions in ELF_TID. In ELF_TID, it saves all `TIDs` in the order of `TID` in Table 2.9. And it can be observed that because MonoList contains `TID`, only the DimensionLists holds the cutoff-pointers. Each element in DimensionList only saves the cutoff pointer of its corresponding leftmost leaf node. For example, the DimensionList $L_{(1)}$ in Figure 2.7 contains two elements $E_{(1)}$ and $E_{(2)}$, where the dimension below element $E_{(1)}$ has branches, which means that there are at least two `TIDs` with an $E_{(1)}$ prefix, here are T1 and T3 respectively. They are adjacent in ELF_TID, so store the position of T1 means store the start position of the `TIDs` prefixed by $E_{(1)}$. Its end position is the position of the cutoff-pointer stored by $E_{(2)}$.

2.3.2 Elf_Separated

Elf_separated is the basis of all Elf variables. In this chapter, we introduce how to separate the standard Elf items. For a better explanation, we use an Elf which is more complicated than in Figure 2.7. In Figure 2.14, we mark all MonoLists in gray.

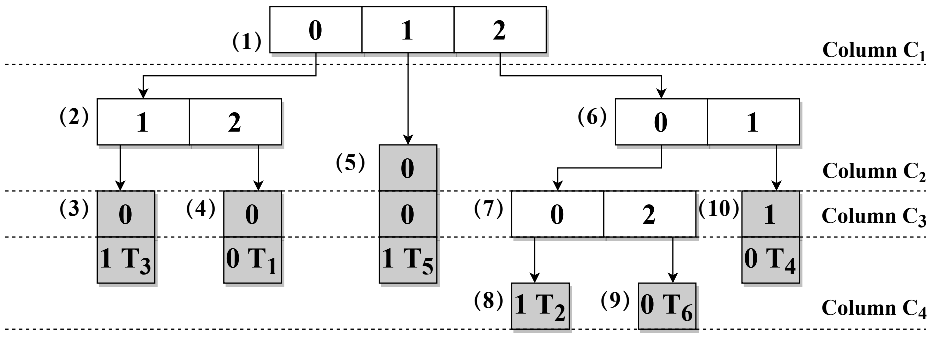


Figure 2.14: More complex Elf structure

The idea of separation is to divide the entries in Elf into five types, and then use a separate data structure to store them. In the Elf memory layout, there are two types of data, one is a value and the other is a pointer. The value saved by the node can be divided into the values in DimensionList and MonoList. There are also two types of pointers, one is a pointer to the next dimension, and the other is a cutoff-pointer. According to these classifications, we can store these values in different data structures.

1. `Elf` stores the dimension values of DimensionList.

2. `MonoLists` only stores the dimension value and TID of MonoList.

3. `Elf_TIDs` only stores TIDs.

4. `Child_Pointers` only stores the pointer of the DimensionList element.

5. `CUTOFF_Pointers` only stores the cutoff-pointer corresponding to the elements of DimensionList.

In Figure 2.15, we show the Elf memory layout after the linearization and separation for the Elf of Figure 2.14. Due to the use of a hash map, the dimension value of the first dimension will not be stored in `Elf`, but from the second dimension. Mark the dimension value with a minus sign to indicate the end of the DimensionList.

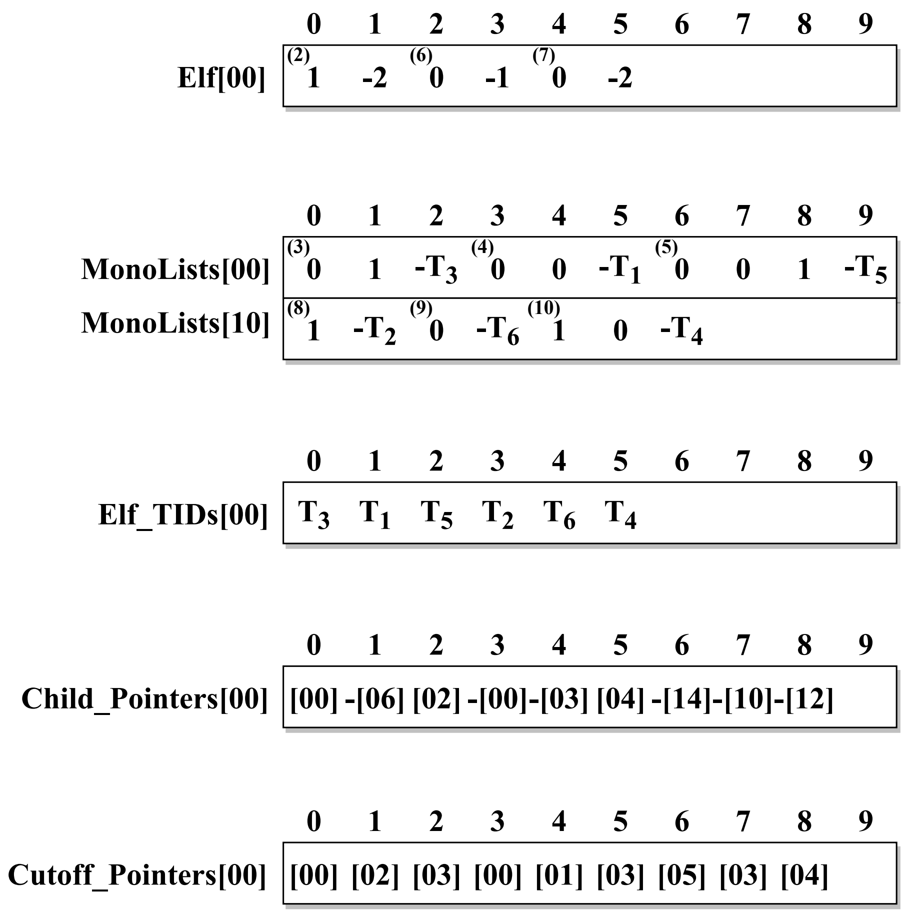


Figure 2.15: Elf_Separated layout

`MonoLists` store all MonoLists individually, marking the end of MonoList with a negative sign. For tuples with the same value and different `TIDs`, we only need to store the value once, then store multiple `TIDs`, and set the most significant bit of the last `TID`. `Elf_TID` stores `TIDs`, and its order is the `TIDs` order after building Elf. `Child_Pointers` stores the pointer of DimensionList but does not include the cutoff-pointers. In addition, because `Child_Pointers` contains the first DimensionList, its storage space requirement is always greater than `Elf`. If we subtract the size of `Elf` from the size of `Child_Pointers`, the result we get will be exactly the size of the first dimension. Because `Child_Pointers` stores the pointer of the DimensionList, the pointer with the minus sign means that the corresponding dimension value points to a MonoList stored in `MonoLists`. The cutoff-pointer corresponding to the dimension value in each DimensionList is stored in the `CUTOFF_Pointers`.

2.3.3 Elf_Separated_Length

In order to use the size of the DimensionList (except the first DimensionList), we explicit store it at the front of each DimensionList in `Elf` instead of set the most significant bit for the last dimension element of the DimensionList. For example, DimensionList $L_{(2)}$ in Figure 2.15 contains two dimension elements, so its length is 2. Mapping the most significant bit of the last element to the `Elf` is equivalent to using a negative sign for the last dimension element to indicate the end of the DimensionList.

The purpose of this design is to adapt to the final variable of Elf - SIMD (Single Instruction Multiple Data). The reason for using explicit lengths for SIMD will be explained in the next section. This design needs to adjust the corresponding data structure. Although only Elf has been redesigned here, the data structures associated with Elf include `Child_Pointers` and `CUTOFF_Pointers`, and their data structures also need to be adjusted. We show this design in the new data structure in Figure 2.16.

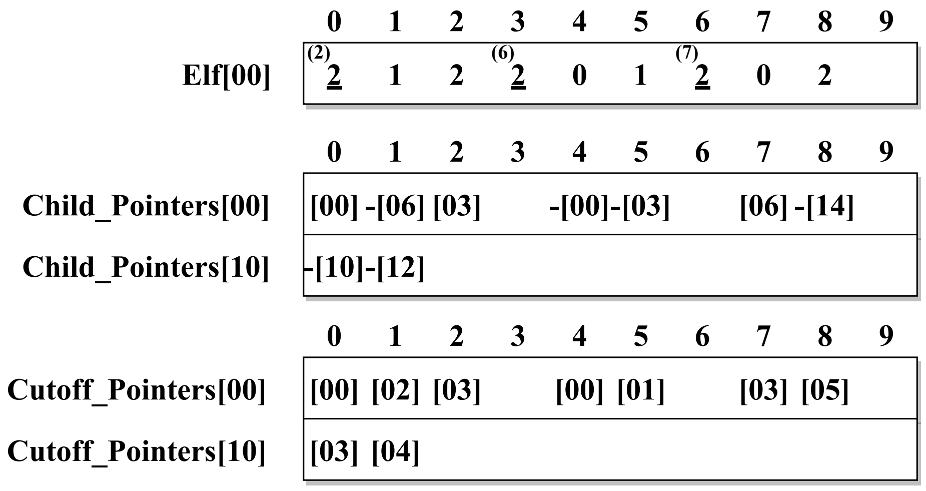


Figure 2.16: Elf_Separated_length layout based on Figure 2.15

In Figure 2.16, the start position of each DimensionList in `Elf` stores the length of DimensionList. This changes the position of all dimensional elements. Because we use the position of the dimension element as the offset to get the position of the pointer in `Child_Pointers`. Therefore, not only the position information of DimensionList must be modified in `Child_Pointers`, but also a gap must be reserved for the length of DimensionList.

2.3.4 Elf_SIMD

SIMD (Single Instruction, Multiple Data) can copy multiple operands and pack them into a set of instructions in large registers. In a synchronous manner, the same instruction is executed at the same time. Take the addition instruction as an example. After the SISD (Single instruction, Single data) CPU decodes the addition instruction, the execution component first accesses the memory and obtains the first operand; then accesses the memory again to obtain the second operand; then the sum operation can be performed. However, in the SIMD CPU, several execution components access the memory at the same time after the instruction is decoded [ZR02], and all the operands are obtained at one time for calculation. This feature makes SIMD particularly suitable for data-intensive operations.

Elf_SIMD uses SIMD for the scanning algorithms in the Elf to take advantage of data parallelism and accelerate their execution if necessary. This variant only provides an improvement on the query algorithm. It uses the Elf structure constructed by the Elf variant Elf_Separated_Length_Offset. The content of this part is comprehensively introduced in Kai Wolf's master's thesis *Datenparallele Selektionen auf der multidimensionalen Indexstruktur Elf*. At the same time, this part occupies a small proportion in our task, so we only make a brief introduction.

2.4 Summary

This chapter introduces the basic principles of this work. We introduce the background of our thesis from two aspects. The first is the index structure. We introduced the index structure by introducing the way of accessing data, and then respectively explained the query type, the one-dimensional index structure and the multi-dimensional index structure in detail. The second is Elf, the basis of our thesis. We first introduce the origin and thoughts of Elf. On this basis, we describe the concept of Elf based on the tree structure. Then we introduced the basic memory layout of Elf and its optimization methods, especially the optimization of linearization. Since our contribution contains some improvements to the partial match query algorithm, we give a brief introduction to the partial match query algorithm for the Elf approach. Finally, based on these backgrounds, we introduced variants of the Elf.

3. Implementation

After we introduced Elf's background, we talked about our work and contributions. In the previous chapters, we briefly mentioned the necessity of level order linearization for Elf, especially the impact on Cutoffs. For the vertically linearized Elf, even if the dimensions do not require Cutoffs, the Elf will still reserve space for Cutoffs for all dimensions during the construction process. Secondly, it was found in the evaluation work that it is beneficial to use level order linearization for some other index structures (e.g., the Seg-Tree, FAST, ART, VAST) [Bro19]. In addition, due to the characteristics of DFS, the Elf approach of vertical linearization cannot skip unnecessary dimensions when executing queries (e.g., PartialMatch Query, ColumnColumn Query). Therefore, for Elf, whether it can benefit from using level order linearization is a new research direction. In this regard, our task is to modify the vertical linearization of the standard Elf to a level order linearization through a new algorithm. In this chapter, we will introduce the implementation of level order linearization in detail.

This chapter is divided into three parts. We introduced the conceptual model of horizontal linearization in the first part, which demonstrated the concept and theory of level order linearization for the Elf approach. The second part is the implementation. In this part, we introduce the algorithm and its code for constructing level order linearization Elf. In the last part, we will introduce the new PartialMatch query algorithm. This algorithm takes advantage of the characteristics of level order linearization. In addition, in the next chapter, we will use this query algorithm for evaluation.

3.1 Conceptual Model

In order to understand the work of level order linearization, we use a simple example (i.e., Figure 2.11) to explain. In Section 2.2.2 we introduced the vertical linearization of standard Elf, and we simplified this process into a way of sorting DimensionsLists and MonoLists. After vertical linearization of Figure 2.11, the node sequence we obtain is $L_{(1)}$, $L_{(2)}$, $L_{(3)}$, $L_{(4)}$, $L_{(5)}$. If we use level order linearization for the same

Elf, the order we expect to obtain is $L_{(1)}$, $L_{(2)}$, $L_{(5)}$, $L_{(3)}$, $L_{(4)}$. We can observe that the node order of Elf using level order linearization has changed. Our task is to achieve this change.

The process of constructing Elf is also the process of linearization. Due to the difference between the level order linearization algorithm (BFS) and the vertical linearization algorithm (DFS), we cannot use recursion as the main algorithm in level order linearization. So in simple terms, our task is to rewrite the standard Elf construction algorithm to achieve level order linearization.

Before introducing the algorithm we implemented, we first introduce and compare the DFS and BFS algorithms in the first section. Then we will introduce a conceptual model based on the BFS algorithm. Finally, we will briefly introduce the advantages and disadvantages of level order linearization that can be predicted in advance during the design process. In addition, in order to better explain the details of level order linearization, we will use a more complex Elf in the following chapters, that is, the Elf shown in Figure 2.14.

3.1.1 Depth First Search and Breadth First Search

Both of Depth-First Search (DFS) and Breadth-First Search (BFS) are graph-based search algorithms [VAQLMJ20]. Their purpose is to traverse all the vertices in the graph. However, due to their different principles and implementation methods, they are often used in different fields and purposes. Because Elf is a tree structure, in this chapter we use the binary tree in Figure 3.1 to introduce the principles of DFS and BFS, and the differences between them.

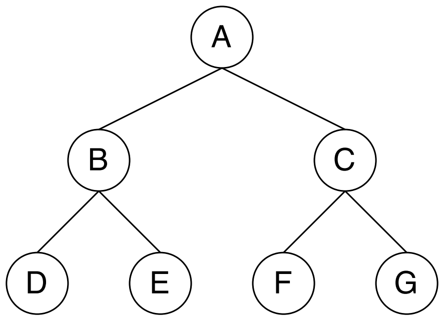


Figure 3.1: A simple binary tree

3.1.1.1 Depth First Search

The DFS algorithm will start from the root node of the tree and traverse longitudinally along the direction of the left subtree until the leaf node is found. Then backtrack to the previous node, and traverse the right subtree node until all reachable nodes are traversed. If we use DFS to traverse the tree in Figure 3.1, the order of the nodes we get is A, B, D, E, C, F, G. DFS always traverses the nodes of the tree along with the depth of the tree, searching the branches of the tree as deep as possible, so we can also call this algorithm vertical linearization.

The key to implementing the DFS algorithm lies in backtracking (i.e., back to front, tracing the way it has traveled). For this, we can use the Last In First Out (LIFO) feature of the stack or use recursion to implement DFS. What we should know is that the implementation of standard Elf linearization uses a recursive method.

1. Stack and Last-In-First-Out

The stack is a data structure in which data items are arranged in order, and data items can only be inserted and deleted at one end (called the top of the stack). We use push and pop to represent these two main principal operations. Push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. The order in which elements come off a stack gives rise to its alternative name, LIFO. Additionally, a peek operation may give access to the top without modifying the stack [KT07]. The name ”stack” for this type of structure comes from the analogy to a set of physical items stacked on top of each other. This structure makes it easy to take an item off the top of the stack, while getting to an item deeper in the stack may require taking off multiple other items first [CLRS09].

2. Recursion

Recursion is a way to solve problems. It usually converts a large and complex problem layer by layer into a smaller problem similar to the original problem. Then it iteratively calculates these small questions, and finally obtains the answer to the original question. In programs, recursion solves such recursive problems by using functions that call themselves from within their own code [Epp10]. Among the many problems that people use recursion to solve, the Fibonacci sequence is one of the most classic. Each item in this sequence starting from item 3 is equal to the sum of the first two items. Mathematically, the Fibonacci sequence is defined by recursion as follows [BG10, Bón02]:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \ (n > 1, n \in N*)$$

Algorithm 3.1: Fibonacci sequence implemented using recursion

```
Input: the $n$ th term of the Fibonacci sequence
Output: the $n$ th Fibonacci number
Function fib($n$):
    if $n$ equals 0 then
        return the first Fibonacci number 0;
    else if $n$ equals 1 then
        return the second Fibonacci number 1;
    else
        return recursively call $fib(n-1) + fib(n-2)$ and finally returns
        the $n$ th Fibonacci number;
End Function
```

We use recursion to implement the Fibonacci sequence in Algorithm 3.1. Suppose we need to get the nth value in the Fibonacci sequence (the sequence counts from 0). If $n < 2$, the value of the sequence is fixed, they are $0, 1$. If $n > 1$, because the value of each item is equal to the sum of the previous two items, it calls itself within the function to get the value of the first two items. If the value of n is large enough, it will iteratively call itself until $fib(2)$ is called. Then all the results are added retrospectively until the value of $fib(n)$ is obtained. This process is a complete recursion. When we use the DFS algorithm to traverse the tree in Figure 3.1, we can simplify this process into three steps:

1. Visit root node A. Saved nodes:$[A]$.
2. Visit the leftmost child node B, and perform a depth-first traversal of the subtree with B as the root node until the rightmost leaf node of the B subtree is visited. Saved nodes:$[A \rightarrow B] \rightarrow [A \rightarrow B \rightarrow D \rightarrow E]$.
3. Visit the child node C on the right, and perform a depth-first traversal of the subtree with C as the root node until the rightmost leaf node of the C subtree is visited. If there is still a subtree on the right side of node C, repeat step 3. Saved nodes:$[A \rightarrow B \rightarrow D \rightarrow E \rightarrow C] \rightarrow [A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G]$.

Algorithm 3.2: A recursive implementation of DFS

```
Input: A vertex $v$ of a graph $G$
Output: All vertices reachable from $v$ labeled as visited
Procedure dfs($v$):
    label $v$ as visited;
    foreach vertex $w$ adjacent to $v$ do
        if vertex $w$ has not been visited then
            recursively call dfs($w$);
        end
    end
End Procedure
```

Recursion can be used to implement the process of DFS traversing the tree, and the generated pseudo code is shown in Algorithm 3.2 (adapted from [GT06]). This algorithm first visits vertex v and prints the node. Then it traverses the child node w of v, if the node w exists and has not been visited, it processes the node w recursively. This process continues until there are no unvisited nodes. Now we discuss the space complexity and time complexity of DFS. Assume that the graph or tree to be traversed by DFS has V vertices in total. Because the goal of the DFS algorithm is to visit all nodes in the graph or tree and save each node once, its space complexity is $O(V)$. The time complexity of DFS is relatively complicated. The process of DFS traversing the graph or tree is essentially the process of finding the neighboring points of each vertex. The time it takes depends on the structure of the storage node used. If we use the form of adjacency list for storage, because each vertex needs to be searched once, $T_1 = O(V)$. In the search process, each edge of the graph (E) or tree is accessed at least once, so $T_2 = O(E)$. The total time complexity is $O(|V| + |E|)$ [CLRS01a]. If we use the adjacency matrix for storage, the time complexity required to find the adjacent point of each vertex is $O(V)$, so the total time complexity is $O(|V|^2)$.

3.1.1.2 Breadth First Search

Compared with DFS, BFS is easier to understand. BFS starts from the root node and traverses the nodes of the tree (graph) along the width of the tree (graph). If all nodes are visited, the algorithm stops. It uses the opposite strategy of DFS, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes [CLRS09]. The key for implementing the BFS algorithm is replay. It is the opposite of backtracking. For example, we traverse Figure 3.1 according to the BFS idea. We first traverse the root node A. Then traverse all the child nodes B and C of A. In order to continue traversing the child nodes of B and C, we need to review the vertices B and C that we have traversed just now in order. So we can find the neighboring vertices D and E from the vertex B; find the neighboring vertices F and G from the vertex C. The final vertex order is A, B, C, D, E, F, G. The process of reviewing the traversed vertices in the order of traversal is called replay.

We often use queues to implement BFS. This is similar to DFS, which implemented using a stack (non-recursive way). A queue is a collection of entities that are maintained in a sequence. The operation of adding an element to the rear of the queue is known as enqueue, and the operation of removing an element from the front is known as dequeue. The operations of a queue make it a First In First Out (FIFO) data structure [Dro12]. In a FIFO data structure, the first element added to the queue will be the first one to be removed [Knu97].

Algorithm 3.3: BFS algorithm

```
   Input: A vertex $root$ of a graph $G$
   Output: All vertices reachable from $root$ labeled as visited
1  Procedure bfs($root$):
2      let $Q$ be a queue;
3      label $root$ as visited;
       // add $root$ to $Q$;
4      $Q.enqueue(root)$;
5      while $Q$ is not empty do
           // take the vertex $v$ in $Q$ and delete it from the $Q$;
6          $v := Q.dequeue()$;
7          forall vertex $w$ adjacent to $v$ do
8              if $w$ is not labeled as visited then
9                  label $w$ as visited;
                   // add $w$ to $Q$;
10                 $Q.enqueue(w)$;
11             end
12         end
13     end
14 End Procedure
```

We show the pseudo-code that implements BFS in Algorithm 3.3 (adapted from [CLRS01b]). We know from the code that the traversal starts from the root node. First, we store the root node in the queue. Then we use a loop to process the nodes in the queue. Because the queue currently only contains the root node, in the while loop, we first take out the root node and traverse all its adjacent nodes (that is, all child nodes of the root node). These vertices are added to the queue in turn until all the child nodes of the root node have been visited. Then continue to loop through the newly added nodes in the queue until all vertices are marked as visited. The time complexity can be expressed as $O(|V| + |E|)$, since every vertex and every edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is [CLRS01a]. The BFS algorithm also traverses all vertices and stores them, so its space complexity is $O(|V|)$.

3.1.1.3 Comparison and Summary

Through the analysis of these two algorithms, we will find that DFS and BFS are the same in time complexity and space complexity. The difference between them lies in the order of access to vertices [YW02]. Because the order of accessing vertices is different, the data structures used to implement them are also different. Backtracking is the main idea of the DFS algorithm, so we can use stack and recursion to implement it. The Replay is the core concept of BFS, so we use queues and loops to implement it. Understanding the background of these two algorithms is helpful to understand our tasks and contributions that will be introduced next.

3.1.2 Concept Design

After introducing the background of the BFS algorithm, we introduce the conceptual design of level order linearization in this chapter. It explains how to construct a level order linearized Elf through theoretical analysis.

TID	C1	C2	C3	C4
T1	0	2	0	0
T2	2	0	0	1
T3	0	1	0	1
T4	2	1	1	0
T5	1	0	0	1
T6	2	0	2	0

Table 3.1: Complex example table

TID	C1	C2	C3	C4
T3	0	1	0	1
T1	0	2	0	0
T5	1	0	0	1
T2	2	0	0	0
T6	2	0	2	0
T4	2	1	1	0

Table 3.2: Sorted complex example table

The Elf structure in Figure 2.14 is constructed by linearizing the data table in Table 3.1. After we construct the Elf, the original data table has been sorted. Finally, a new data table as shown in Table 3.2 is formed, which conforms to the ordered node mentioned in Section 2.2.1.1. In the following chapters, we will use these two tables for analysis. First, we will introduce the algorithms used in our task, then conduct theoretical analysis and discuss the derived problems.

The implementation of level order linearization for Elf is based on the breadth-first search algorithm. After introducing the DFS and BFS algorithms, we know that the recursive method is no longer applicable. The Elf structure with width as the search level will be constructed and traversed in units of dimensions (i.e., columns). Therefore, the process of constructing a level order linearized Elf can also be regarded as a process of sequentially scanning each column of the data table. As we introduced in the second chapter, the standard Elf uses a hash map for the first dimension, which not only improves the performance of the Elf but also reduces the storage space of the first dimension. In addition, whether we use vertical linearization or level order linearization to construct an Elf, scanning the first dimension (i.e., first column) is the first step. Therefore, as long as we maintain the vertical linearization processing method for the first dimension, we do not need to pay attention to the first dimension.

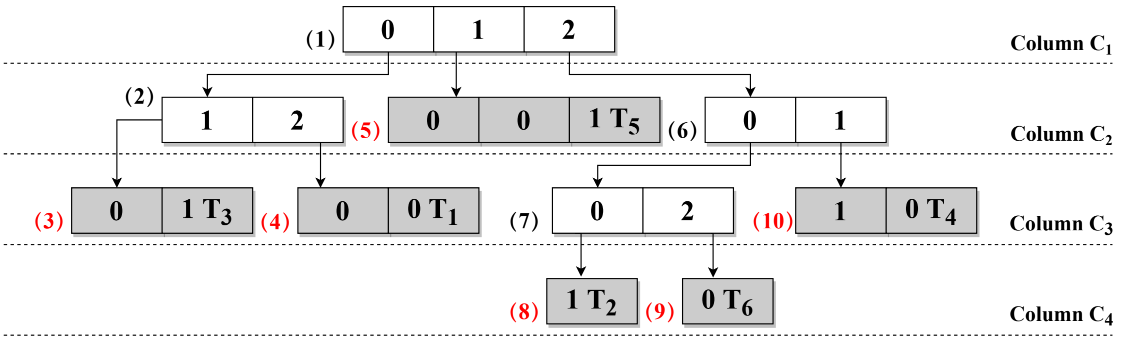
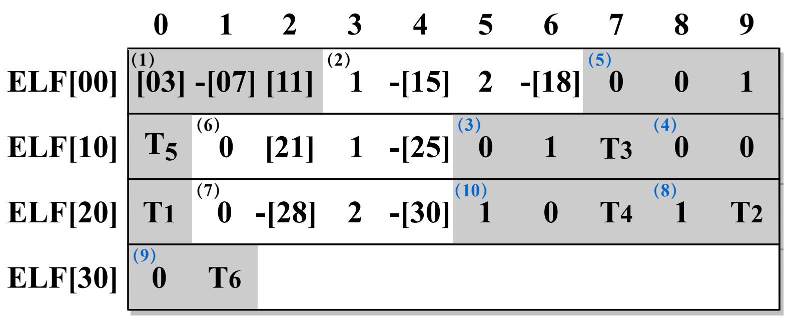


Figure 3.2: Elf structure adjusted based on Figure 2.14

For the second dimension, since we cannot use recursive methods to implement level order linearization, the process of constructing or traversing Elf does not require backtracking. In addition, level order linearization retains the processing method of vertical linearization for the first dimension. For all dimensions from the second dimension to the last dimension, they can all use the same linearization method. Therefore, for all dimensions except the first dimension, we use a separate loop. The entire first dimension is treated as a dimension column, so there is no special case. However, due to the introduction of MonoList, starting from the second dimension, there are multiple possibilities for the composition of each dimension. We can use combination pairs to represent these possibilities: $\langle DimensionLists\rangle$, $\langle MonoLists\rangle$, $\langle DimensionLists, MonoLists\rangle$. For example, in Figure 3.2, there is only one DimensionList L_1 in the first dimension C_1. The second dimension C_2 and third dimension C_3 have both DimensionLists (L_2, L_6, L_7) and MonoLists (L_5, L_3, L_4, L_{10}). The fourth dimension C_4 has only MonoLists (L_8, L_9).

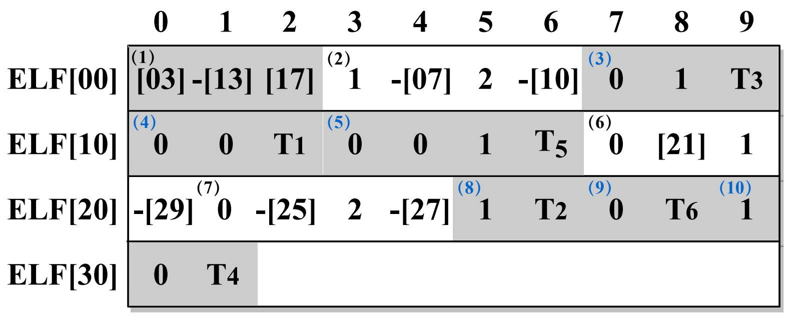
According to the definition of MonoList, when a MonoList is retrieved during the construction of Elf, we store the entire MonoList in the corresponding data structure. We use vertical linearization and level order linearization to construct Elf for Table 3.1, and show the memory layout of the Elf approach formed by these linearizations in Figure 3.3 and Figure 3.4. In Figure 3.2, L_5 will be the first MonoList stored when the Elf is constructed in a level order linearized manner. The constructed Elf stores MonoLists in the order of dimensions. For MonoLists in the same dimension, Elf stores them in order from left to right. Therefore, the order of the MonoList stored in the Elf constructed by level order linearization in Figure 3.3

is $L_5, L_3, L_4, L_{10}, L_8, L_9$. However, in Figure 3.4, the order of MonoList stored in Elf constructed by vertical linearization is $L_3, L_4, L_5, L_8, L_9, L_{10}$.



	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) [03]	-[07]	[11]	(2) 1	-[15]	2	-[18]	(5) 0	0	1
ELF[10]	T_5	(6) 0	[21]	1	-[25]	(3) 0	1	T3	(4) 0	0
ELF[20]	T1	(7) 0	-[28]	2	-[30]	(10) 1	0	T4	(8) 1	T2
ELF[30]	(9) 0	T6								

Figure 3.3: The memory layout of the Elf constructed by level order linearization of Table 3.1 (the first dimension and the MonoLists are marked in gray)



	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) [03]	-[13]	[17]	(2) 1	-[07]	2	-[10]	(3) 0	1	T3
ELF[10]	(4) 0	0	T1	(5) 0	0	1	T_5	(6) 0	[21]	1
ELF[20]	-[29]	(7) 0	-[25]	2	-[27]	(8) 1	T2	(9) 0	T6	(10) 1
ELF[30]	0	T4								

Figure 3.4: The memory layout of the Elf constructed by vertical linearization of Table 3.1 (the first dimension and the MonoLists are marked in gray)

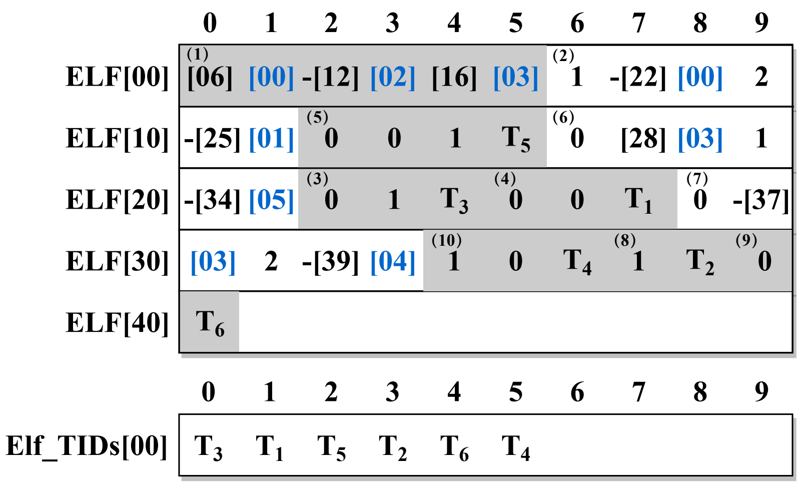
We can observe that the order of the MonoLists stored in standard Elf is consistent with the order of all leaf nodes in Figure 5 from left to right. In contrast, the MonoLists stored in the Elf constructed by level order linearization are out of order. However, it still has an observable law. In Figure 3.2, the MonoList L_5 in the second dimension C_2 contains 3 elements and a `TID`. All MonoLists in the third dimension C_3 contains 2 elements and a `TID`. In addition, all MonoLists in the fourth dimension C_4 contain 1 Element and a `TID`. Therefore, for an N-dimensional Elf, if there is at least one MonoList in each dimension except the first dimension, then the length of the MonoList on the M-th dimension is $N - M + 1$ $(M > 1)$. Its value range is $[N - 1, 1]$. This rule has no special meaning for vertical linearization. However, it is very important for level order linearization, because the length of the MonoList is an important indicator. From Figure 3.3, we observe that the length of the MonoLists decreases in the order of storage (the MonoLists in the same dimension have the same length). When we use other variants of the Elf to save the MonoLists, we can more easily observe this rule. In addition, because the MonoLists are stored in the specified data structure, if we can use the rule that the number of elements in the MonoLists decreases in the order of storage, it will make our processing of MonoList easier.

3.1.3 Elf Variants

In this chapter, we will introduce the different Elf variants constructed by level order linearization (e.g., Elf64_Level, Elf_Level_Separated). We have already introduced the background of Elf variants in Section 2.3. Therefore, we will directly introduce and analyze their memory layout.

3.1.3.1 Elf64_Level

Figure 3.3 shows the memory layout of Elf64_Level without Cutoffs. Figure 3.5 shows the memory layout of Elf64_Level with Cutoffs. In Figure 3.5 we mark the entire first dimension and all the MonoLists with a gray background. In addition, the blue font indicates the cutoff-pointer. By comparing the memory layout of Elf64 in Figure 3.4 and the memory layout of Elf64_Level in Figure 3.3, We can deduce several features about the level order linearization.



	0	1	2	3	4	5	6	7	8	9
ELF[00]	(1) [06]	[00]	-[12]	[02]	[16]	[03]	(2) 1	-[22]	[00]	2
ELF[10]	-[25]	[01]	(5) 0	0	1	T_5	(6) 0	[28]	[03]	1
ELF[20]	-[34]	[05]	(3) 0	1	T_3	(4) 0	0	T_1	(7) 0	-[37]
ELF[30]	[03]	2	-[39]	[04]	(10) 1	0	T_4	(8) 1	T_2	(9) 0
ELF[40]	T_6									

	0	1	2	3	4	5	6	7	8	9
Elf_TIDs[00]	T_3	T_1	T_5	T_2	T_6	T_4				

Figure 3.5: Memory layout of the Elf64_Level with Cutoffs based on Figure 3.2

Feature 1: The pointers in the first dimension of the level order linearized Elf are more densely distributed than those of the vertical linearized Elf. For example, the Elf[00] in Figure 3.3 and Figure 3.4 both point to Elf[03], the Elf[01] in Figure 3.3 (level order linearization) points to Elf[07], but the same element in Figure 3.4 (vertical linearization) points to Elf[13]. The reason for this situation is that Elf64 first stores the entire subtree, but Elf64_Level first stores the entire dimension.

Feature 2: We can quickly determine the start pointer and end pointer of a certain dimension. For example, in Figure 3.5, the start position of the first dimension is Elf[00]. It points to Elf[06], which is the start position of the second dimension. Therefore, the position range of the first dimension can be simplified to [00, 06). Similarly, the range of the second dimension is [06, 22). This also means that the pointers stored in the first element in the first DimensionList of each dimension point to the start position of the next dimension. With this feature, we can also quickly determine the coordinate range of a certain dimension.

```
SELECT * FROM Table WHERE C1 < 2;
```

Listing 3.1: Partial range query based on Table 3.1

Feature 3: The order of `TIDs` in Elf_TIDs is different from the order of traversing MonoLists. The Elf_TIDs in Figure 3.5 are exactly the same as the Elf_TIDs generated by standard Elf. Because the order of such `TIDs` conforms to the order of the leaf nodes from left to right in the Elf structure. Many queries can benefit from this order of `TIDs`. For example, we perform the partial range query in Listing 3.1 in Table 3.1. We use the interval to express $C_1 < 2$, which is $[0, 2)$. Then we use the closed interval $[0, 1]$ instead of $[0, 2)$. From Table 3.2, we can directly find the qualified `TIDs`: T_3, T_1, T_5. If we perform this partial range query on the Elf64_Level with Cutoffs in Figure 3.5. We only need to traverse its first dimension to find its upper boundary Elf[00] and lower boundary Elf[02]. Then we determine the range of the corresponding cutoff-pointers, which is ([00], [02]). Finally, we take out all `TIDs` in this range from the data structure Elf_TIDs and store them in the result vector. From the Elf_TIDs in Figure 3.5, we get the `TIDs` in the range of ([00], [02]): T_3, T_1, T_5. This result is consistent with the expected result.

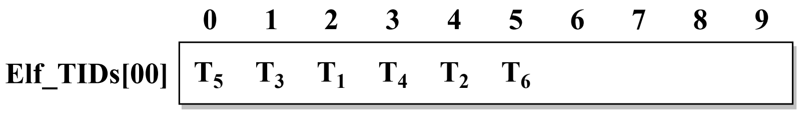


Figure 3.6: Elf_TIDs that stores `TIDs` in the order of MonoLists in Figure 3.5

When we use the data structure Elf_TIDs shown in Figure 3.6, the order of `TIDs` in this data structure is equivalent to the order of traversing the MonoLists in the level order linearization process. Then, the cutoff-pointer of Elf[00] is Elf_TIDs[01], and the cutoff-pointer of Elf[02] is Elf_TIDs[00]. Their corresponding `TIDs` in this range only contain T_5 and T_3, so we cannot use these two cutoff-pointers to simplify the query process. Therefore, when we implement the level order linearization algorithm, we will preserve the order of `TIDs` in Figure 3.5 in a special way.

Feature 4: It can be observed from the memory layout of Elf64_Level that the position coordinates (pointer) stored in its DimensionLists are increasing, rather than out of order as shown in Figure 3.4. For example, Elf64_Level without Cutoffs in Figure 3.3, where the position coordinates stored in all DimensionLists are arranged in order [03], [07], [11], [15], [18], [21], [25], [28], [30]. The reason for this feature is the same as for feature 1. They are all because level order linearization constructs Elf in the order of dimensions.

3.1.3.2 Elf_Level_Separated

We have introduced the first variant of the standard Elf in Section 2.3.2. In this chapter, we will introduce Elf_Level_Separated, which is the first variant of level order linearization Elf. In Figure 3.7 we show the memory layout of Elf_Level_Separated constructed by level order linearization and its cutoffs=N (N is the total number of dimensions).

Compared with Figure 2.15, we can more intuitively observe that the order of storage nodes has changed. At the same time, in the data structure called MonoLists, we have a deeper understanding of the length of the MonoLists mentioned in Section 3.1.2. In Figure 3.7, we use red lines to identify the elements in each MonoList. We can find that those MonoLists with the same number of elements belong to the same dimension. For example, L_3, L_4 and L_{10} belong to the third dimension C_3 in Figure 3.2. Therefore, this storage order in MonoLists implies information about dimensions. Reasonable use of this information is very helpful for us to skip unselected predicates.

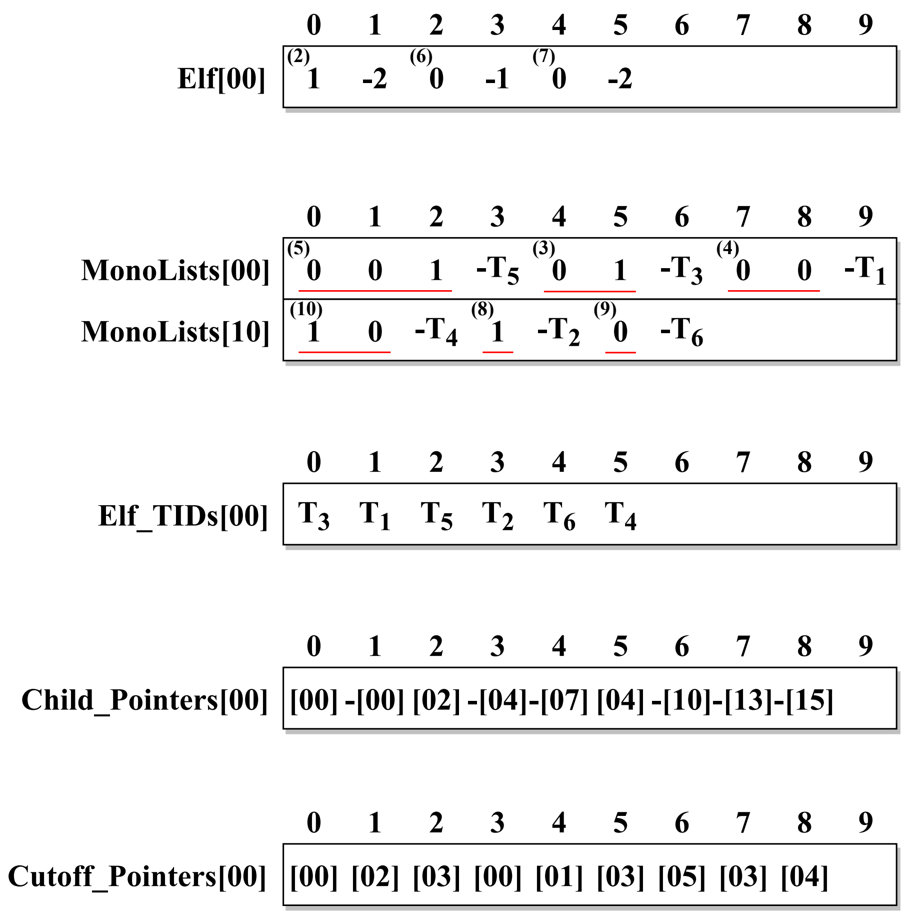


Figure 3.7: Memory layout of Elf64_Level_Separated based on Table 3.1

Feature: Unselected dimensions will not reserve space for Cutoffs. This feature only applies to Elf_Separated variants. We provide a new Elf structure on the left side of Figure 3.8. According to this structure, we show the memory layout of Cutoff_pointers of Elf_Level_Separated and Elf_Separated on the right side of Figure 3.8. In these memory layouts, we only add cutoff-pointers for the first two dimensions. We can observe from the figure that there are gaps (for L_3) in Elf_Separated. This means that it needs to reserve space of Cutoffs pointers for all DimensionLists.

Elf_Level_Separated eliminates this gap. It only provides space for the dimensions that need to add cutoff-pointers.

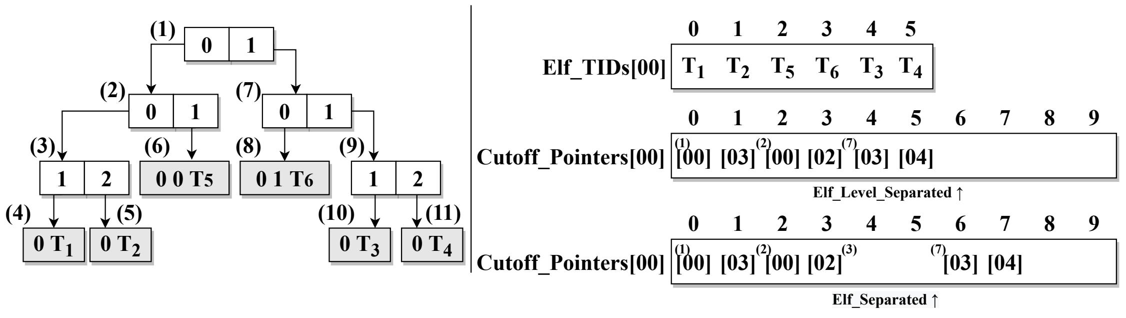


Figure 3.8: Cutoff_Pointer of Elf_Level_Separated and Elf_Separated with cutoffs=2

3.1.3.3 Elf_Level_Separated_Length

Elf_Level_Separated_Length has not changed much in the memory layout compared to the standard Elf variant. The only difference is still the different traversal order caused by different algorithms. We show in Figure 3.9 the memory layout of Elf_Level_Separated_Length constructed by level order linearization. Because the memory layout of `Elf_TID` and `MonoLists` has not changed, we no longer show them in the figure.

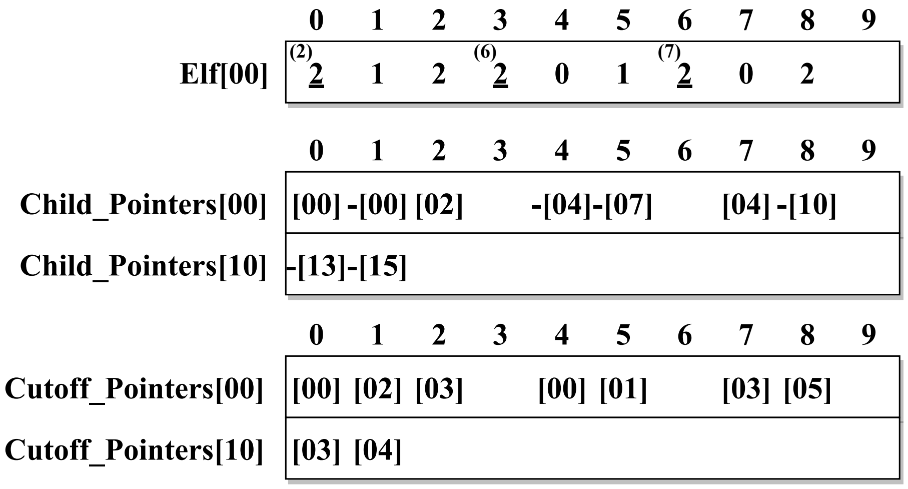


Figure 3.9: Memory layout of Elf64_Level_Separated_Length based on Table 3.1

3.1.4 Theoretical Advantages and Disadvantages

According to the conceptual design and the research on the variants, we put forward the possible advantages and disadvantages of the Elf constructed with horizontal linearization in theory. Then in the evaluation stage, we verify these theoretical conjectures in practice.

Advantages

1. Space advantage for Elf_Separated. This involves the space issue of Cutoffs that we have been talking about. In the Elf constructed by level order linearization, since the tree structure is traversed in the BFS manner, we can only reserve space for Cutoffs in the DimensionLists before the n-th dimension (cutoffs=n, n≤N, N is the total number of dimensions) instead of reserving space for all DimensionList. This not only avoids the waste of space but also eliminates the gap problem of Cutoffs.

2. Retrieve directly from the dimension corresponding to the first selection predicate. Due to the characteristics of horizontal linearization, we can easily find the start and end positions of a certain dimension. Therefore, it is different from vertical linearization in that not all query operations must start from the first dimension. For example, when querying, if our first selection predicate corresponds to the third dimension, then in the Elf constructed by Level order linearization, it can be retrieved directly from the third dimension. For the dimensions before the third dimension, we only need to deal with the possible MonoLists.

Disadvantages

1. In terms of construction time, although the theoretical space-time complexity of DFS and BFS are the same, because the algorithm and data structure may not be implemented completely in accordance with the standard algorithm of BFS, there are still some differences in the specific implementation. For level order linearization, we need to traverse and process each dimension (column). Perhaps this process is still weaker than the recursive method of vertical linearization.

2. In terms of query time, level order linearization may not perform as well as vertical linearization due to differences in data tables and selection predicates. For example, if the mono-column selection predicate only involves the first dimension, then the characteristics of level order linearization are not particularly prominent. However, these are only speculations. In fact, even if there are some special cases, the performance of level order linearization is similar to that of vertical linearization.

These advantages and disadvantages are only based on theory, they are some points that we can consider in the design stage. We will perform reasonable evaluations to verify these points after implementing the level order linearization algorithm.

3.2 Algorithm of Level Order Linearization

In the previous section, we have introduced the conceptual design and theoretical background of level order linearization. In this section, we will introduce the implementation and code of level order linearization.

3.2.1 Implementation

In this section, we first introduce the construction algorithm of level order linearization by dividing Table 3.1. Then we introduce how to preserve the same storage order of TIDs as vertical linearization during the level order linearization process. Finally, we briefly introduce some special methods we used in the process of implementing other Elf variants.

3.2.1.1 Algorithm Design

When we process a data table, the data table will be cached in a temporary two-dimensional vector in units of columns. Then, we scan each column. We first sort the scanned columns until all the column elements are in order. For example, after we sort the first column, a new table will be formed. All the same values in the first column can be regarded as a common prefix, which also means that the number of unique values in the first column is equal to the number of prefixes. If the second dimension of the same prefix has different unique values, then the prefix points to a DimensionList. The existence of different values in the second dimension with the same prefix can also be called existence branch. If there are no branches in all dimensions with the same prefix, they are MonoLists. After understanding these backgrounds, we began to analyze Table 3.1. First, we complete the first step of the algorithm, which is to sort the first column C_1 of the table. We show in Table 3.3 the new table formed after sorting the first column. In order to show the arrangement of DimensionLists and MonoLists on each column more clearly, we use bold red to identify DimensionLists from Table 3.3 to Table 3.6 and use black bold to identify MonoLists.

TID	C1	C2	C3	C4
T1	**0**	2	0	1
T3	**0**	1	0	0
T5	**1**	0	0	1
T2	**2**	0	0	0
T4	**2**	0	2	0
T6	**2**	1	1	0

Table 3.3: The table formed after sorting the first column based on Table 3.1

TID	C1	C2	C3	C4
T3	0	**1**	0	1
T1	0	**2**	0	0
T5	1	**0**	**0**	**1**
T2	2	**0**	0	0
T6	2	**0**	2	0
T4	2	**1**	1	0

Table 3.4: The table formed after sorting the second column based on Table 3.3

1. Processes the First Dimension

After we temporarily store the table in a two-dimensional vector, we can initialize the Elf size according to the total size of the data volume. Because for Elf64_Level, all data must be stored in the same data structure Elf, so the initial size of Elf can be preset to the size of the entire data table containing `TIDs`. Then we process the data in the first column (first dimension) after sorting. From Table 3.3, we can obtain three distinct values, which are 0, 1, and 2. Since we do not change the processing method for the first dimension, we still use the hash map to store the relevant data of the first dimension. Therefore, in this step, we determine the spatial size of the first dimension. This means that we have also determined the starting position of the second dimension (Elf[03] in Figure 3.10) and store this position information under the first element of the first dimension (Elf[00] in Figure 3.10). In Figure 3.10, what we have marked with red boxes are the three dimension values of the first dimension. The gray background is used to fill the space of the first dimension in the Elf.

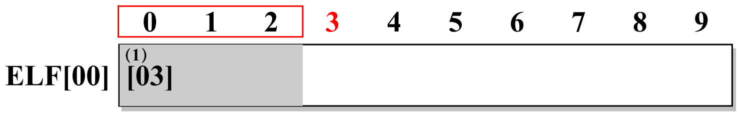


Figure 3.10: The process of constructing the first dimension of Elf64_Level (1)

We know that in the first dimension, the hash map value is equal to the corresponding dimension value. We need to store pointers in the first dimension. Because these pointers are based on the construction of the second dimension, while we deal with the first dimension, we need to process the dimension value of the second dimension. Sorting is still the first step when we deal with new dimensions. We sorted all the dimension values of the first dimension as a whole. However, for other dimensions, we first need to partition according to its prefix. Then we sort the dimension values in each partition separately. Table 3.4 is the new table formed after we sort C_2 in this way. In addition, when we use the dimension value in C_1 as a prefix, we can divide the second column C_2 into three partitions: prefix 0 - $\langle T_3, T_1 \rangle$, prefix 1 - $\langle T_5 \rangle$ and prefix 2 - $\langle T_2, T_6 T_4 \rangle$. According to the definition of the DimensionList and the MonoList, we know that the partitions prefixed with 0 and 2 have different dimension values in C_2. Therefore, the dimension values 0 and 2 in the first dimension point to DimensionLists. Since there is no branch in the partition prefixed by the dimension value 1 of the first dimension, the dimension value 1 points to a MonoList. Their relationship is clearly shown in column C_1 and column C_2 in Figure 3.2.

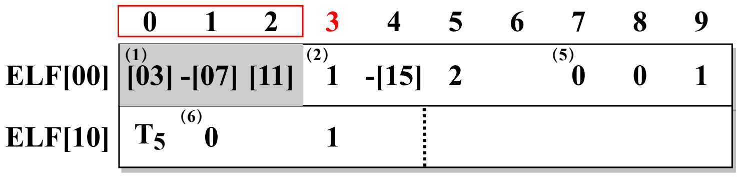


Figure 3.11: The process of constructing the first dimension of Elf64_Level (2)

After the sorting is complete, our next step is to add the unique values of all partitions in the second dimension to the Elf in the order of the prefix. We show in Figure 3.11 the memory layout after adding the dimension value of the second dimension. Because all the DimensionLists (L_2, L_6) and MonoLists (L_5) in the second dimension have been stored in the Elf, we can obtain the corresponding pointer (-[07], [11]) to store in the first dimension. At the same time, we can also get the space size of the second dimension and add the start pointer of the third dimension to Elf[04]. We can find that if we want to add the complete data (i.e., space size and pointer) of the first dimension to the Elf, we need to process the data in the first and second columns of the data table at the same time.

Whether we deal with the first dimension or the second dimension, their common feature is that we must traverse each of their rows. This is a key point to realize the level order linearization algorithm. Because our processing method for the first column of all data tables is fixed, we can observe the law when it adds the dimension value of the second dimension to the Elf. For example, from C_1 we can get three dimension values: 0, 1, and 2. We need to partition C_2 based on these prefixes. The number of rows in these partitions is 0 (2 rows), 1 (1 row) and 2 (3 rows). Their sum is also the total number of rows in the data table.

If we can save the information about the number of these rows in the order of prefixes, we can deal with them according to the number of rows when processing the second dimension, because these rows have their own prefixes. For example, we use a temporary vector $Temp$ to store the number of rows: $Temp = [2, 1, 3]$. When we process C_2, we take the data in Temp in order, we can know that the first two rows in Table 3.3 have the same prefix 0. Then, the second data of $Temp$ is 1, we use a variable $rowSum$ to save the calculated number of rows, $rowSum = 2 + 1$, which is the third row. Therefore, $rowSum[n] = Temp[0] + Temp[1] + ... + Temp[n]$ ($n <$ total number of rows). With this vector, we can directly obtain the range of rows under the same prefix without comparing the prefix information of C_1.

2. Processes the Second Dimension

When we deal with the second dimension, the biggest difference between it and the first dimension is that the second dimension is composed of several DimensionLists and several MonoLists. We partitioned the second dimension by the dimension values of the first dimension. Each partition is either DimensionList or MonoList. The criterion for determining the type of each partition is that we sort each partition of C_2. If there are different dimension values in the partition, then the partition is DimensionList. If there is only one dimension value in the partition, there are two cases. If the partition contains only one row, then the partition must be a MonoList. The second case is that if there are multiple rows in the partition, then we perform branch determination. If there is a branch in the later dimension, the partition is a DimensionList. Otherwise, there are multiple MonoLists. We store related row information in the temporary vector $Temp\_C_2$ according to whether the partition is DimensionList or MonoList. For example, we sort the partitions of C_2:

1. $Temp[0] = 2$, we directly sort the second column of the first two rows T_3 and T_5 of Table 3.3. We found that this partition meets the criteria of the DimensionList. So we store the number of rows (1 and 1) contained in these two dimension values, but because this partition is a DimensionList, we need to mark the last entry of it. We set the last 1 through the bitwise OR operation. Therefore, the data of the first partition we save in the $Temp\_C_2$ is $[1, 1(L)]$, which means that the first row of the data table is the starting point of the first DimensionList in the second dimension, and the second row is the end point of this DimensionList, L represents the last entry mask.

2. $Temp[1] = 1$, which means that this partition has only one row, which meets the standard of MonoList. Because we use the bitwise OR operation identifier as a MonoList, and then store it in $Temp\_C_2$ as $[1, 1(L), 1(M)]$, and M represents the Mono Mask.

3. $Temp[2] = 3$, the new partition contains three rows. As shown in Table 3.4, after we sort them, the partition meets the standard of DimensionList. It has two dimension values 0 and 1. The dimension value 0 involves two rows, and 1 involves only one row, so it is stored in $Temp\_C_2$ as $[1, 1(L), 1(M), 2, 1(L)]$.

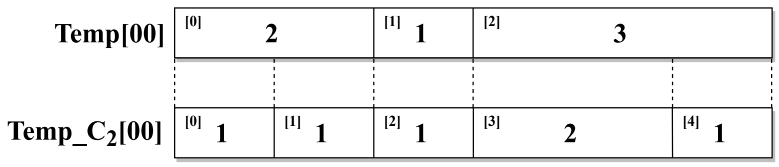


Figure 3.12: The memory layout of the vector $Temp$ and $Temp\_C_2$ generated based on Table 3.4

In the implementation, we only use one temporary vector to store information about the number of these rows. We will clean up the vector in each loop, and then add the data about the number of rows in the next column to the vector. However, for the convenience of explanation, we use its own temporary vector for each column in this chapter. Next, we analyze the existing two temporary vectors $Temp$ and $Temp\_C_2$. We show the memory layout of these two vectors in Figure 3.12. From the figure, we can observe that the value of $Temp[0]$ is 2, which represents two rows in the data table. When we deal with the second dimension, we need the value of Temp to determine the range of each partition on C_2, that is, we need to split the elements of $Temp[n]$ when $Temp[n] > 1$, so $Temp[0]$ will be split into two vector elements $Temp\_C_2[0]$ and $Temp\_C_2[1]$ with both values 1. We can simplify this relationship to:

- $Temp[0] = Temp\_C_2[0] + Temp\_C_2[1]$
- $Temp[1] = Temp\_C_2[2]$
- $Temp[2] = Temp\_C_2[3] + Temp\_C_2[4]$

In Figure 3.11, we add the dimension values in the partitions into the Elf according to the order of the partitions in the second column. Whenever a complete partition is added to the Elf, we can set the pointer for the first dimension until all partitions in the second dimension are added to the Elf. Similarly, if we want to set pointers for the second dimension in Elf, we need to use the vector value of $Temp\_C_2$ to process the third dimension, and add the dimension value in the third dimension to the Elf. Next, we begin to deal with the third and fourth dimensions.

3. Processes the Third and Fourth Dimensions

From the size of $Temp\_C_2$, we know that the third column C_3 is divided into 5 partitions. $Temp\_C_2 = [1, 1(L), 1(M), 2, 1(L)]$. Since we use the number of rows as the vector value, the value range of all vector values is the closed interval [1, N] (N is the total number of rows in the data table). The minimum value is 1, which represents a row in the data table. The vector values not marked with M in C_2 are all part of the DimensionList. For example, $Temp\_C_2[0]$ and $Temp\_C_2[1]$ together form a DimensionList. However, because their respective partitions in C_3 only contain one

TID	C1	C2	C3	C4
T3	0	1	**0**	**1**
T1	0	2	**0**	**0**
T5	1	**0**	**0**	**1**
T2	2	0	0	0
T6	2	0	2	0
T4	2	1	**1**	**0**

Table 3.5: The table formed after sorting the third column based on Table 3.4

TID	C1	C2	C3	C4
T3	0	1	**0**	**1**
T1	0	2	**0**	**0**
T5	1	**0**	**0**	**1**
T2	2	0	0	**0**
T6	2	0	2	**0**
T4	2	1	**1**	**0**

Table 3.6: The table formed after sorting the fourth column based on Table 3.5

row, starting from C_3, these two rows are MonoLists. We add the corresponding data in $Temp\_C_3$, which is $[1(M), 1(M)]$. In addition, $Temp\_C_2[2]$ $(1(M))$ already has the identifier M, which means that when we are processing the second dimension, $Temp\_C_2[2]$ has been processed in the first dimension as a MonoList, so we can ignore it. In order to be able to distinguish it from the new MonoList in the latter dimension, we use the bitwise OR operation to make a new set of this vector value, which is $1(M\_EX)$. When we encounter the M_EX mark again, we can skip it directly. Therefore, we save it in $Temp\_C_3$, which is $[1(M), 1(M), 1(M\_EX)]$. We can know from Table 3.5 that the fourth partition of C_3 is a DimensionList. Then, we split $Temp\_C_2[3]$ (2) into $Temp\_C_3[3]$ and $Temp\_C_3[4]$. Since $Temp\_C_2[4]$ has the same state as $Temp\_C_2[0]$, it is a MonoList. Therefore, the vector $Temp\_C_3[3]$ after we added is $[1(M), 1(M), 1(M\_EX), 1, 1(L), 1(M)]$.

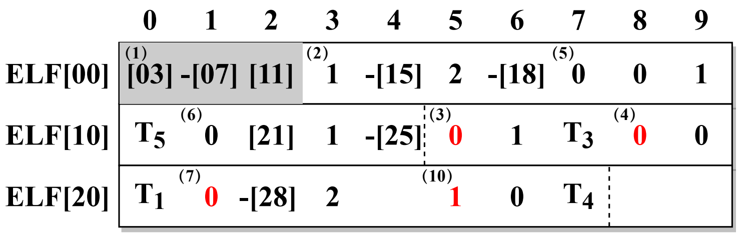

0 1 2 3 4 5 6 7 8 9
ELF[00] (1) [03] -[07] [11] (2) 1 -[15] 2 -[18] (5) 0 0 1
ELF[10] T5 (6) 0 [21] 1 -[25] (3) 0 1 T3 (4) 0 0
ELF[20] T1 (7) 0 -[28] 2 (10) 1 0 T4

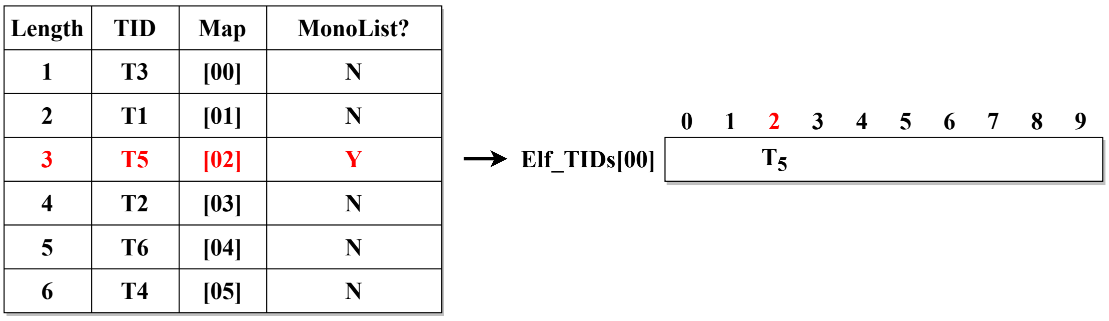
Figure 3.13: The process of constructing the first dimension of Elf64_Level (3)

We showed in Figure 3.13 the memory layout of Elf after we processed the third dimension. Compared with Figure 3.11, the second dimension has been constructed (the pointer insertion is completed), and all the DimensionLists and MonoLists of the third dimension have been added, only the pointers of its DimensionLists have not been set. In order to add pointers to the third dimension, we use the same method to process the fourth dimension. $Temp\_C_3$ divides the fourth column in Table 3.6 into 6 partitions, and each partition contains only one row. For the MonoList that has been processed, we use M_EX to set. Therefore, the $Temp\_C_4$ we get is $[1(M\_EX), 1(M\_EX), 1(M\_EX), 1(M), 1(M), 1(M\_EX)]$. However, since C_4 is already the last dimension, the vector value with the mark M can also be set to the mark M_EX through the bitwise OR operation. Therefore, the final saved $Temp\_C_4$ is: $[1(M\_EX), 1(M\_EX), 1(M\_EX), 1(M\_EX), 1(M\_EX), 1(M\_EX)]$. At the same time, the final Elf memory layout is shown in Figure 3.3. At this point, the construction of Elf64_Level is complete. The sign of its completion is when all the values in our temporary vector are $1(M\_EX)$.

3.2.1.2 Algorithms related to Cutoffs

We have introduced the construction algorithm of level order linearization for Elf without Cutoffs. When Elf introduces Cutoffs, in addition to reserving space for Cutoffs in the data structure, we also need to store `TIDs` in Elf_TIDs. We have already discussed the different storage order of `TIDs` in Elf_TIDs in Section 3.1.3.1 with Figure 3.6. Now we will introduce how to preserve the order of `TIDs` in vertical linearization during the construction of level order linearization.

When we introduced the construction algorithm in Section 3.2.1.1, we used temporary vectors to store information about the number of rows spanned by the dimension value. If we add up all the items in the vector, its sum is equal to the total number of rows in the data table. From this feature, we can introduce *Length*. Except for the first dimension, every time it loops, all rows in the next column will be traversed. We use *Length* in each loop to record the number of rows we are processing. We know that whenever we find a MonoList, the position of this row in the table will no longer change. As in Table 3.4, we find the first MonoList whose TID is T_5. Then we compare the position of T_5 in Table 3.5 and Table 3.6, we will find that this position has been fixed. Because the size of Elf_TIDs is equal to the total number of rows in the data table. Therefore, when we find a MonoList, we can map the number of rows recorded in *Length* to Elf_TIDs, and save the TID of the MonoList in the corresponding position in Elf_TIDs.



Length	TID	Map	MonoList?
1	T3	[00]	N
2	T1	[01]	N
3	T5	[02]	Y
4	T2	[03]	N
5	T6	[04]	N
6	T4	[05]	N

Figure 3.14: The process of adding `TIDs` to Elf_TIDs

We show the process of adding T_5 to Elf_TIDs in Figure 3.14. This process is based on Table 3.4. Correspondingly, $Temp\_C_2 = [1, 1(L), 1(M), 2, 1(L)]$. From the term of this temporary vector, we can observe that in the process of processing the second dimension, we have found the first MonoList ($1(M)$). The length of this item is $1 + 1(L) + 1(M)$, which is equal to 3. We map it to Elf_TIDs, and its position is Elf_TIDs[02]. Then we only need to put T_5 in this position. Whenever we find MonoList, we use its *Length* to determine the position in Elf_TIDs and add TID in it. However, even if Elf_TIDs is empty, we can still set the cutoff-pointers for the dimension values of each DimensionList.

As shown in Figure 3.15, we add the cutoff-pointers to the Elf structure and use the red font to identify it. Taking the first dimension as an example, when we process the first dimension, we will get a temporary vector $Temp = [2, 1, 3]$. The size of $Temp$ is 3, which is the number of dimension values in the first dimension. We use the $Length$ in Figure 3.14 to represent the three partitions divided by $Temp$: 2 - $\langle 1, 2 \rangle$, 1 - $\langle 3 \rangle$, 3 - $\langle 4, 5, 6 \rangle$. We take the first $Length$ of each partition, which is 1, 3 and 4. These $Lengths$ correspond to positions 0, 2 and 3 on Elf_TIDs in Figure 3.15. Therefore, we can directly use the $Length$ to set the cutoff-pointers for the DimensionList without having to wait until all `TIDs` are added to Elf_TIDs. The $Length$ can also be used to set a pointer for the dimension column, skip the MonoList that has been processed, and serve as an indicator of the end of the construction.

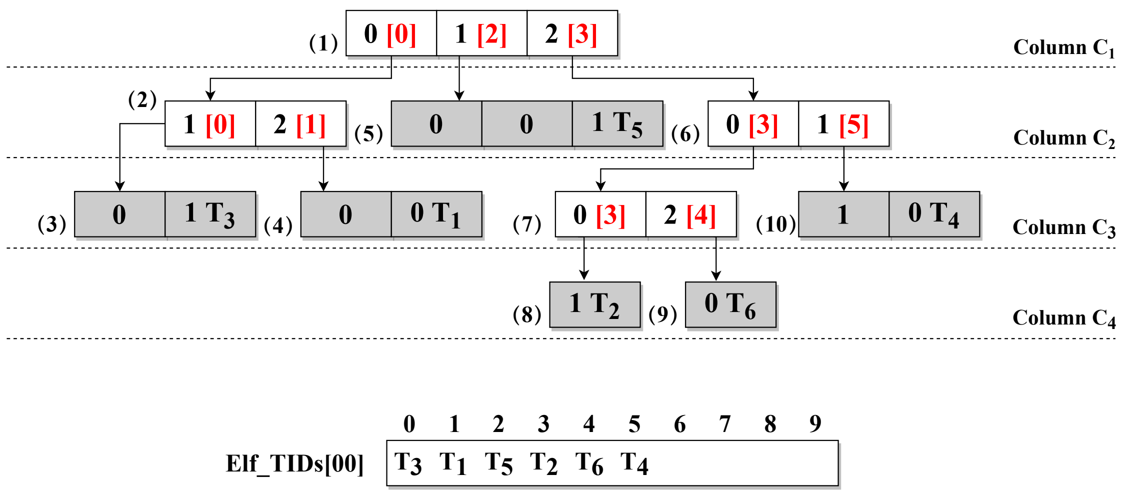


Figure 3.15: Elf structure with Cutoffs and memory layout of Elf_TIDs

3.2.1.3 Examples of extreme scenarios

After introducing the construction algorithm of level order linearization, we propose two possible extreme scenarios based on this algorithm.

1. There is no DimensionList from the nth dimension

When the nth dimension is processed, the construction is complete ($0 < n < N$, N is the total number of dimensions). Because the level order linearization algorithm we implemented is not like vertical linearization, it does not need to go from the root node to the leaf node every time. This provides the possibility to complete the construction after processing certain dimensions near the front. For example, the data table of lineitem.tbl has 15 columns. After sorting, starting from the 12th dimension, all are MonoLists, which means that the construction of the level order linearization ends when the processing reaches the 13th dimension. Because the 12th dimension is not the last dimension, the value in the temporary vector still contains $1(M)$ at this time, so we need to perform the bitwise OR operation for all $1(M)$ in the 13th dimension to set the bit to $1(M\_EX)$. When the values in the vector are all $1(M\_EX)$, we don't have to scan the remaining dimensions.

2. There is a DimensionList in the $N - 1$th dimension

In the construction phase, for level order linearization, the other extreme scenario is that there are DimensionLists in the $N - 1$ th dimension (N is the total number of dimensions, $n \in [1, N]$). In addition, if there is only one DimensionList in this dimension, a lot of time will be used to scan the temporary vector from the second dimension to the $N-1$th dimension. Because the level order linearization algorithm is different from the recursive algorithm, each time it loops through the entire column of data in the data table. Although the theoretical time complexity of DFS and BFS algorithms is the same, due to the influence of MonoList and DimensionList, the construction time of level order linearization is slightly inferior in theory.

3.2.2 Introduction of Pseudo-Code

After introducing the algorithm we designed, in this chapter, we introduce our implementation code. We use pseudo-code to illustrate the construction process of level order linearization in detail. For existing algorithms, such as reading data tables, storing the data of the data table in a multi-dimensional vector, etc., we will not introduce them. For the level order linearization algorithm, we split it into three parts. The first part is the linearization of the first dimension, the second part is the linearization of the remaining dimensions ($1 < Dim \leq N$, N is the total number of dimensions), and the third part is the linearization of the DimensionList. In addition, we will also selectively introduce the implementation code of Elf variants.

3.2.2.1 Linearize the First Dimension

After we read the data table and store the compressed and encoded data in the `Store`. We can start the work of linearizing the data table. In Algorithm 3.4 `linearizeFirstDim`, we use pseudo-code to describe the process of linearizing the first dimension.

First, we pass the vector member `store` of the object `s` into the function as a parameter. Then we initialize the variables (Lines 1-2). In order to keep the pseudo code concise, we did not list all the variables. Here, we introduce the key variables that appear in the code. For example, `nextDimTemp` (see line 22) is a vector whose size is equal to the total number of rows in the data table. We use it to store the number of rows associated with each dimension value in each column. It is also the temporary vector mentioned in Section 3.2.1.1. Then, `data_array` is a two-dimensional vector, and the data items it stores can be expressed as `data_array[TID][value]`. We dump the data in the vector `store` into `data_array` row by row. After it saves all the data, we sort `data_array[TID][0]`, that is, sort the first value of each row, which is equivalent to sorting the first dimension (Lines 3-7). We take the first value of the first dimension after sorting. Then, we reset the size of the Elf, because we use the hash map for the first dimension, so we only need to get the maximum value of the first dimension to set the size of the Elf (Lines 8-11).

We start to retrieve the rows involved in each dimension value in the first dimension. This method compares the first data in each row with the data stored in `cur` until it finds the first different dimension value. Therefore, the first value of all rows involved during this period is the same dimension value. We store these rows in a two-dimensional vector `temp`, and analyze these rows in vector `temp`, using the dimension value of the first dimension as the prefix to determine whether it has branches in other dimensions (Lines 13-15). If there is a branch, it means that there

Algorithm 3.4: Linearize the first dimension

Input: The vector *store* saves the data of the data table in a row-store manner

Output: An Elf containing the complete first dimension and dimension values of the second dimension.

```
Procedure linearizeFirstDim(&store[]):
    initialization;
    vector⟨TID⟨tid_type, value_type⟩⟩ data_array;
    for i ← 0 to NUM_POINTS do
        // fetch a row of data from the store each time the loop;
        data_array[i] ← &store[i*NUM_DIM] to &store[(i+1)*NUM_DIM];
    end
    sort(data_array[].begin().value[0], data_array[].end().value[0]);
    cur ← data_array.front().value[0];
    pos ← cur;
    Elf.resize(max_dims[0] + 1)*getStepSize(0);
    begin ← 0;
    for i ← 1 to NUM_POINTS do
        // get the dimension value of the first dimension;
        if cur != data_array[i].value[0]) then
            temp[] ← &data_array.at(begin) to i - begin;
            if hasBranchOut(temp[], 0) then
                setPointer(pos, 0, Elf.size());
                if Cutoffs > 0 then
                    setCutoffPointer(pos, 0, Elf_TID.size());
                end
                linearizeDimList(pos, temp[], 1);
            end
            else
                nextDimTemp[] ← ((i-begin) | VECTOR_MONO_MASK);
                setPointer(pos, 0, Elf.size() | MONO_LIST_MASK);
                if Cutoffs > 0 then
                    setCutoffPointer(pos, 0, Elf_TID.size());
                end
                linearizeMonoList(temp[], 1);
            end
        end
        begin ← i;
        cur ← data_array[i].value[0];
        pos ← cur;
    end
    // linearize remaining dimension;
    linearizeRemainDim(data_array[]);
    Elf_TID ← NUM_POINTS;
End Procedure
```

is a DimensionList in the following dimension. If there is no branch, then these rows are MonoLists.

If there is a branch, because the first dimension uses a hash map, we can use the function `setPointer` to directly set the pointer for it. Then, if the use of Cutoffs is allowed, we call the function `setCutoffPointer` to set the cutoffs pointer for the first dimension. Since we have set the size for the first dimension in Elf, we can add the pointer of the start position of the second dimension to the first dimension. In order to continue setting the pointer for the first dimension, we need to deal with the DimensionList. This means that we need to call `linearizeDimList` to add the dimension value of the DimensionList in the second dimension to the Elf (Lines 15-21). We will analyze `linearizeDimList` in the next section. If there is no branch, we process these rows as MonoList. First, we store the number of rows in the temporary vector `nextDimTemp`, and perform a bitwise OR operation on the data and the bit mask `VECTOR_MONO_MASK` to identify the type of these rows as MonoList. Then we set the pointers of MonoList and cutoff-pointers for the first dimension. Finally, we use the function `linearizeMonoList` to add MonoList to the second dimension in Elf. After processing this dimension value, we use a loop to process the remaining dimension values (Lines 22-34). After the first dimension is processed, we get the Elf as shown in Figure 3.11. Since the construction method of level order linearization for the first dimension is not applicable to other dimensions, we put the processing of other dimensions in another function `linearizeRemainDim`.

3.2.2.2 Linearize the Remaining Dimensions

Starting from the second dimension, the level order linearization process for all dimensions is the same. We demonstrated the linearization algorithm for the remaining dimensions in Algorithm 3.5 `linearizeRemainDim`. We pass the vector *data_array* as a parameter to the function `linearizeRemainDim`. This vector stores the data of the data table and the first dimension and the second dimension of these data have been sorted.

First of all, we are still initializing variables, such as the `dimension`, the number of rows to be processed `beginPos`, and the starting number of rows for the next prefix `realPos`. A nested loop is the main implementation algorithm, the outer loop is the loop of the dimensions, and the inner loop is the traversal processing for each dimension. Because each time through the loop, the data table will be rearranged, and a new temporary vector `nextDimTemp` will be generated, which stores the latest information about the number of rows. Each time the loop ends, the data table will be rearranged and a new temporary vector will be generated, which stores the latest information about the number of rows. Therefore, before we deal with the new dimension, we use a new local vector `dimTemp` to store the data in vector `nextDimTemp`. Then we use vector `dimTemp` to process the new dimensions and vector `nextDimTemp` to store new data about the number of rows (Lines 1-6). In the inner loop, we use iterators to get the data from `dimTemp`. Since in the temporary vector we use several different bit masks to identify the MonoList(M), the last dimension value of the DimensionList(L) and the processed MonoList (M_EX), we need to remove all the masks to get real data (Lines 6-10).

Algorithm 3.5: Linearize the remaining dimensions

```
Input: A vector that stores rows with the common prefix data_array
Output: A complete Elf constructed by level order linearization
Procedure linearizeRemainDim(&data_array[]):
    initialization;
    for dimension ← 1 to NUM_DIM do
        dimTemp[].clear();
        dimTemp[].swap(nextDimTemp[]);
        // Traverse each column by number of rows;
        for subSpan ← dimTemp[].begin() to dimTemp[].end() do
            pos = *subSpan;
            if isLastEntry(pos) || isMono(pos) || isMonoEx(pos) then
                pos ← pos & ALL_MASK_RECOVER;
            end
            realPos ← beginPos + pos;
            temp[] ← &data_array[beginPos] to &data_array[pos];
            // Processing DimensionLists;
            if pos!=1 && hasBranchOut(temp[], dimension) then
                linearizeDimList(*subSpan, temp[], dimension+1);
            end
            else
                // Skip processed MonoList;
                if *subSpan & MONO_MASK_EX then
                    ++count;
                    nextDimTemp[] ← *subSpan;
                end
                // Process the MonoList processed in the previous dimension
                   and set it to MONO_MASK_EX;
                else if *subSpan & MONO_MASK then
                    linearizeDimList(*subSpan, temp[], dimension+1);
                end
                // Process the MonoList;
                else
                    setPointer(curPointer, dimensionm, Elf.size());
                    nextDimTemp[] ← *subspa | MONO_MASK;
                    linearizeMonoList(temp[], dimension+1);
                end
            end
            beginPos ← realPos;
        end
        // Construction completion indicators;
        if count == dimTemp[].size() then
            break;
        end
    end
End Procedure
```

Then, we perform the corresponding level order linearization according to the types of these lines. If the number of rows is greater than 1 and there are branches, it means that there are unprocessed DimensionLists in these rows. We execute function `linearizeDimList` to add DimensionLists (Lines 12-15). However, if the number of rows is 1, there are several states, such as the Monolist that has been processed before the previous dimension, that is, the number of rows marked with M_EX, we just need to skip it (Lines 17-20). If it is the MonoList processed in the previous dimension, that is, the number of rows marked with M, we need to process it in linearizeDimList so that it is marked as M_EX and provides position information for setting the pointer (Lines 21-22). If the number of rows is 1, but there is no mask related to MonoList, it also means that the row is a MonoList starting from the current dimension (Lines 25-28). We determine whether the construction is complete by counting the number of rows marked with M_EX (see Line 18 and Line 32-34). When the value of count is equal to the total number of rows in the data table, it means that all columns have been processed. There are no longer any unprocessed DimensionLists and MonoLists. At the same time, this represents that the construction of level order linearization has been completed.

3.2.2.3 Linearize the DimensionList

In the process of constructing Elf with level order linearization, it is very important to deal with DimensionLists reasonably. It involves the problem that for the same dimension the time of adding dimension value and adding pointer is not synchronized. When we add the dimension value of the n-th dimension ($n \in [2, N]$, N is the total number of dimensions), we can only set the pointer for the DimensionLists of the n-1 th dimension. In addition, the process of adding the dimension value of the n+1 th dimension to Elf belongs to work of the next loop. Therefore, we introduce how to implement this algorithm in this chapter and show its pseudo-code in Algorithm 3.6.

The three parameters we need to pass in the function are *position*, *data_array* and *dim*. The parameter *position* is the value obtained directly from the temporary vector `nextDimTemp`, and this value has not been processed by removing the mask. We use this raw data to determine whether these rows are MonoLists or DimensionLists. The parameter *data_array* stores the row data of those rows corresponding to the *position*. The parameter *dim* represents the dimension we will deal with. We divide the pseudo-code in Algorithm 3.6 into three parts according to their functions. The first part is the initialization, in this part, we mainly pre-process the necessary and qualified data. The second part is to add the dimension value of the second dimension into the Elf to set the pointer for the first dimension. The third part is to set the pointer for the DimensionList of the previous dimension. Next, we will analyze this algorithm comprehensively.

If we want to set a pointer for the n-th dimension, then we need to add all the DimensionLists and MonoLists in the $n + 1$ dimension to the Elf, and since it is not possible to set pointers for these DimensionLists at the same time, we only add the dimension values of all DimensionLists in the $n + 1$ dimension to the Elf. Therefore, when we initialize variables, we need a vector `dimList` to store the starting row number of each dimension value of the DimensionList (see Lines 10).

Algorithm 3.6: Linearize the DimensionList

```
Input: Number of rows (with mask) position,
       A vector that stores rows with the common prefix data_array,
       Number of the dimension to be processed dim
Output: A complete Elf constructed by level order linearization
Procedure linearizeDimList(position, &data_array[], dim):
    initialization;
    if !isMonoMask(position) || dim - 1 == FIRST_DIM then
        sort(data_array[].begin().value[dim], data_array[].end().value[dim]);
        begin ← 0 ;
        cur ← data_array[0].value[dim];
        for i ← 1u to data_array[].size() do
            if cur != data_array[i].value[dim] then
                dimList[] ← begin;
                begin ← i;
                cur ← data_array[i].value[dim];
            end
        end
        dimList[] ← begin;
    end
    if dim - 1 == FIRST_DIM then
        Elf.resize(ElfStartSize + dimList.size() * getStepSize(dim));
        for j ← 0u to dimList.size() do
            listLength ← dimList[j + 1] - dimList[j];
            nextDimTemp[] ← listLength;
            setValue(curValuePos, data_array[dimList[j]].value[dim]);
            if cutoffs > dim then
                setCutoffPointer(curValue, dim, tidSize);
        end
    else
        if position & MONO_MASK then
            ++count;
            nextDimTemp[] ← position | MONO_MASK_EX;
        else
            setPointer(curPointerPos, dim - 1, position);
            Elf.resize(ElfStartSize + dimList.size() * getStepSize(dim));
            for j ← 0u to dimList.size() do
                nextDimTemp[] ← listLength;
                setValue(curValuePos, data_array[dimList[j].value[dim]]);
                if cutoffs > dim then
                    setCutoffPointer(curValuePos, dim, tidLength);
            end
        end
    end
End Procedure
```

After initializing the variables, we first retrieve the dimension values of the `dim`-th dimension in the rows stored in the `data_array`. This process is similar to how we deal with the dimension values of the first dimension. First, we sort the values of the `dim`-th column in `data_array`. After sorting, the rows in `data_array` have been rearranged in the order of the `dim`-th column. We use the variable `cur` to store the first dimension value of the `dim`-th column after sorting. In addition, we use the variable `begin` to record the start position of these dimension values (row number in `data_array`). Whenever we find a new dimension value, we store the information of its starting position and use `cur` to save this new dimension value in order to find the next dimension value. After we find all the dimension values, the start position of all dimension values is saved in `dimList`. These position data are also the main reference data when we add data to the temporary vector `nextDimTemp` (Lines 3-16).

After we get all the dimension values of a certain DimensionList in the `dim`-th column, we need to expand the size of the Elf first (see Line 17), and then add these dimension values to the Elf (see Line 21). Due to the special situation of the first dimension, we set the pointer for the first dimension in the function `linearize-FirstDim`. In the function `linearizeDimList`, we only add the dimension value of the second dimension to Elf and save the corresponding number of rows to the temporary vector `NextDimTemp` (see Line 19-20). In addition, if we need to set the cutoff-pointers for the current dimension `dim`, we only need to set the value of `cutoffs` to be greater than `dim`. In line 19, we show the calculation method of the data stored in the temporary vector `NextDimTemp`, which is the calculation method of *Length* mentioned in Section 3.2.1.2. It is the number of rows involved in a dimension value. The result obtained by subtracting the starting position of the current dimension value from the starting position of the next dimension value is the number of rows.

When `dim`$>$ 1, we need to perform a set operation for the MonoList processed in the previous dimension (`dim`-1), so that the row of the MonoList is marked with M_EX (Lines 26-29). Then, when we process the next dimension (`dim`$+1$), we can skip this line directly by identifying the M_EX identifier. In line 27, we use the global variable `count` to count the number of new M_EX identifiers. The advantage of this is that in the temporary vector `nextDimTemp`, once all the values become $1(M\_EX)$, the construction is completed immediately instead of scanning the vector again. For the DimensionList, we first set the pointer for the DimensionList of the (`dim`-1)-th dimension in the Elf, and then add the dimension value of the `dim`-th dimension to the Elf to set the pointer for the next DimensionList of the (`dim`-1)-th dimension (Lines 30-40).

3.2.2.4 Construction Algorithm in Elf Variant

The main difference between Elf's variant and Elf is that instead of using only one data structure to store all the data, the Elf variant store the data in different data structures. For example, **`Elfs`**, **`MonoLists`**, **`Elf_TIDs`** store numeric data, **`Child_Pointers`**, **`Cutoff_Pointers`** store pointer data. We only need to replace the data structure in the implementation code of Elf64 to complete the implementation of Elf variants. Since the algorithm is the same, we only show the part that replaced the data structure and some key statements in Algorithm 3.7.

Algorithm 3.7: Linearize the first dimension for the Elf variants

```
Input: The vector store saves the data of the data table in a row-store
       manner
Output: Elf the DimensionLists of the second dimension,
        MonoLists all the monoLists of the second dimension,
        Child_Pointers all the pointer of the first dimension,
        Cutoff_Pointers cutoff-pointers of the first dimension,
        Elf_TIDs TIDs of MonoLists in the second dimension
Procedure linearizeFirstDim(&store[]):
    initialization;
    for i ← 1 to NUM_POINTS do
        if cur != data_array[i].value[0]) then
            if hasBranchOut(temp[], 0) then
                Child_Pointers[pos] ← Elf.size();
                if Cutoffs > 0 then
                    Cutoff_Pointers[pos] ← Elf_TIDs.size();
                end
                linearizeDimList(pos, temp[], 1);
            end
            else
                nextDimTemp[] ← ((i-begin) | VECTOR_MONO_MASK);
                Child_Pointers[pos] ← MonoLists.size() |
                  MONO_LIST_MASK;
                if Cutoffs > 0 then
                    Cutoff_Pointers[pos] ← Elf_TIDs.size();
                end
                linearizeMonoList(temp[], 1);
            end
        end
    end
End Procedure
```

Compared to Algorithm 3.4, we can observe that for the Elf variant, we do not call the function to add the corresponding data to the Elf, but directly store the data in the corresponding data structure. For example, when we store the cutoff-pointers and the DimensionList pointer, we directly use the corresponding data structure `Cutoff_Pointers` and `Child_Pointers` to store the data (see Line 6 and 8). For algorithm `linearizeRemainDim` and algorithm `linearizeDimList`, we also store the data in the corresponding data structure to complete the construction of the Elf variant.

3.3 Partial Match Query Algorithm

After we analyze the construction algorithm, we introduce our other contribution in this chapter, that is, Partial Match Query Algorithm (PM). We redesigned the PM algorithm based on the features of Elf variants constructed by level order linearization. In addition, we have implemented PM algorithms for Elf variants (e.g. Separated, Separated_Length). The new PM algorithm does not apply to Elf64_Level, the reason we will explain in the following section. Before we introduce the pseudo code, we first introduce the algorithm design and compare it with the PM algorithm adapted to the Elf constructed by vertical linearization. Then, we present an important problem that we found during the design process and discuss its solution.

3.3.1 Partial Match Query Algorithm Design

In Section 3.1.3.1, we analyzed and summarized the features of Elf64_Level. Combined with our analysis of the Elf constructed by level order linearization, we expect that the new PM algorithm can effectively utilize the horizontal traversal characteristics of BFS. In addition, we have briefly introduced the four query types in Section 2.1.2.2, from which we can summarize the scope of the selection predicate of PM query. For a data table with N columns, if the total number of selection predicates given in the query statement is less than N, the query statement is a PM query. The selection predicates appearing in a query statement are often randomly combined and it is possible that there is no selection predicate on the first column. What we expect to achieve by using the characteristics of horizontal traversal is that we can directly jump to the first predicate for PM queries. Therefore, we can first determine that the two functions provided by the new PM algorithm are:

1. When there is a selection predicate in the first column, the PM algorithm starts to match from the first dimension;
2. When there is no selection predicate in the first column, the PM algorithm directly jumps to the first selection predicate for the query.

We use these two functions as the start point for designing a new PM algorithm. When there is a selection predicate in the first column, we will retrieve matches from the first dimension. This is consistent with the PM algorithm of vertical linearization. The query statement is divided into mono-column select predicate query and multi-column select predicate query. Therefore, if the predicate involves the first column and the query is a one-dimensional query, we only traverse the first column. At this point, we can use Cutoffs to quickly obtain query results. If we don't use Cutoffs, we perform normal traversal retrieval, which also applies to the case of multi-column selection predicate queries. When there is no predicate in the first column, whether it is a mono-column selection predicate query or a multi-column selection predicate query, we directly jump to the dimension where the first predicate is located and start searching. However, for vertical linearization, due to the algorithm limitation of DFS, it still needs to traverse and retrieve from the first dimension.

The new PM algorithm does not apply to Elf64_Level

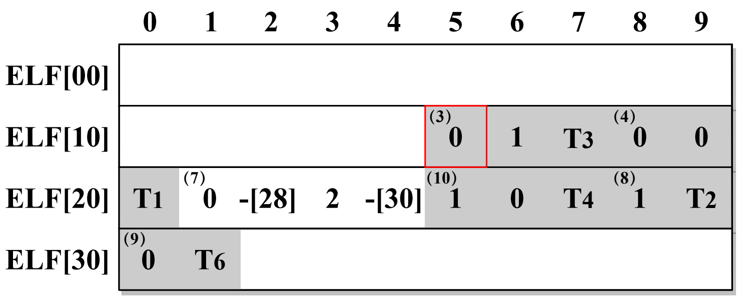


Figure 3.16: The memory layout of Elf64_Level starting from C_3

Combined with the analysis of the Elf and Elf variants constructed by level order linearization in the Section 3.1.2, we know that for the MonoList L_5 of the second dimension in Figure 3.2, we directly store it in the second dimension of Elf (see Figure 3.3). If the first predicate starts from the third dimension (C_3) in Figure 3.2, for Elf64_Level, according to the new PM algorithm, we jump directly to the Elf[15] in Figure 3.16, so we will not be able to match the MonoList L_5 stored in the second dimension. If we want to determine the position of L_5, we must retrieve the first dimension to get the pointer. Therefore, Elf64_Level will use the general PM algorithm, that is, it always starts retrieval from the first dimension.

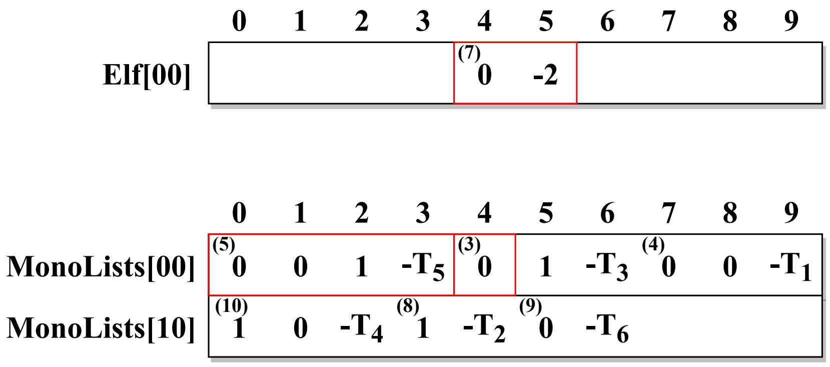


Figure 3.17: The memory layout of Elf_Level_Separated starting from C_3

For variants of Elf64_level, such as Elf_level_Separated, it stores MonoLists in a separate data structure. The above situation becomes solvable. We can handle MonoLists and DimensionLists separately. As shown in Figure 3.17, we only need to deal with L_5 in the data structure `MonoLists`, without having to traverse C_1 and C_2 completely. If the dimension of the first predicate contains several DimensionLists and MonoLists or only DimensionLists, we first process all the MonoLists before this dimension, and then directly jump to this dimension. If this dimension only contains MonoLists, we only need to retrieve all MonoLists. Therefore, effective retrieval of the data structure `MonoLists` has become one of the focuses of our design.

3.3.2 MonoLists in Elf_level_separated

In Section 3.1.3.2, we analyze the data structure `MonoLists` of Elf_Level_Separated. We can quickly determine the start position of each dimension, but we cannot quickly determine the start position of each dimension in MonoLists, so we need to design a method to quickly confirm the dimension in the process of retrieving MonoLists. For example, when the first predicate is C_3, we need to process all the MonoLists of the second dimension in the data structure. The challenge at this time is that we have to confirm when this process ends. Obviously, when it traverses to the first MonoList of the third dimension, this process can end. However, how to effectively obtain this indicator is of great significance for us to realize the new PM algorithm. We have proposed three possible solutions to discuss which one is more efficient.

3.3.2.1 Completely Traverse

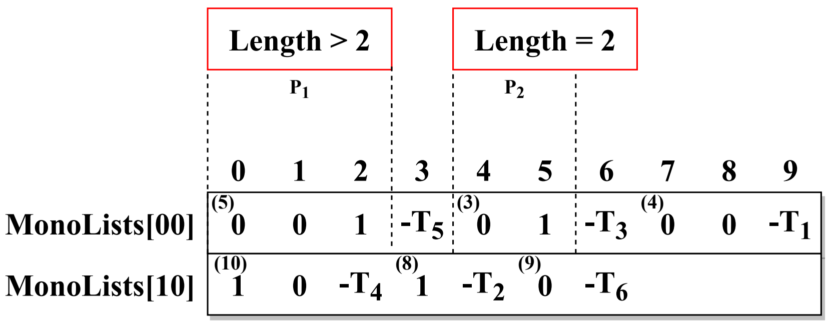

Length > 2
Length = 2
P1
P2
0 1 2 3 4 5 6 7 8 9
MonoLists[00]
(5) 0 0 1 -T5 (3) 0 1 -T3 (4) 0 0 -T1
MonoLists[10]
(10) 1 0 -T4 (8) 1 -T2 (9) 0 -T6

Figure 3.18: Method of processing MonoLists (1) - Completely Traverse

This method requires us to traverse all MonoLists from the front. In this process, we traverse each MonoList and record the number of data in each MonoList. The way to get this value is that we traverse a MonoList and start counting when we read the first value. When we read the first TID, the count ends. The data we get is the length of this MonoList. We apply the same method for other MonoLists. In the process of traversal, if we find that the MonoList currently traversed is shorter than the length of MonoLists in the first predicate, we can stop the traversal process. It means that we have reached the next dimension. The length of the MonoList in the next dimension is always shorter than the length of the MonoList in the predicate. In addition, the MonoLists of the next dimension does not need to be processed in advance.

As shown in Figure 3.18, we still use C_3 as the first predicate of the PM query. According to the algorithm in Section 3.2.1.1, the length of the MonoList on the M-th dimension is $N - M + 1 (M > 1)$. Therefore, the length of the MonoList on

C_3 is 2 (value of $4 - 3 + 1$). When we traverse the MonoLists, if we encounter the first MonoList whose length is not greater than 2 (P_2 in Figure 3.18), we can stop. The last position we traversed in Figure 3.18 is MonoLists[06].

3.3.2.2 First visited MonoList

In the process of level order linearization construction, we use a temporary vector to store the pointer of the first MonoList we encounter in each dimension. We can use this method to indirectly get the start pointer of each dimension in `MonoLists`.

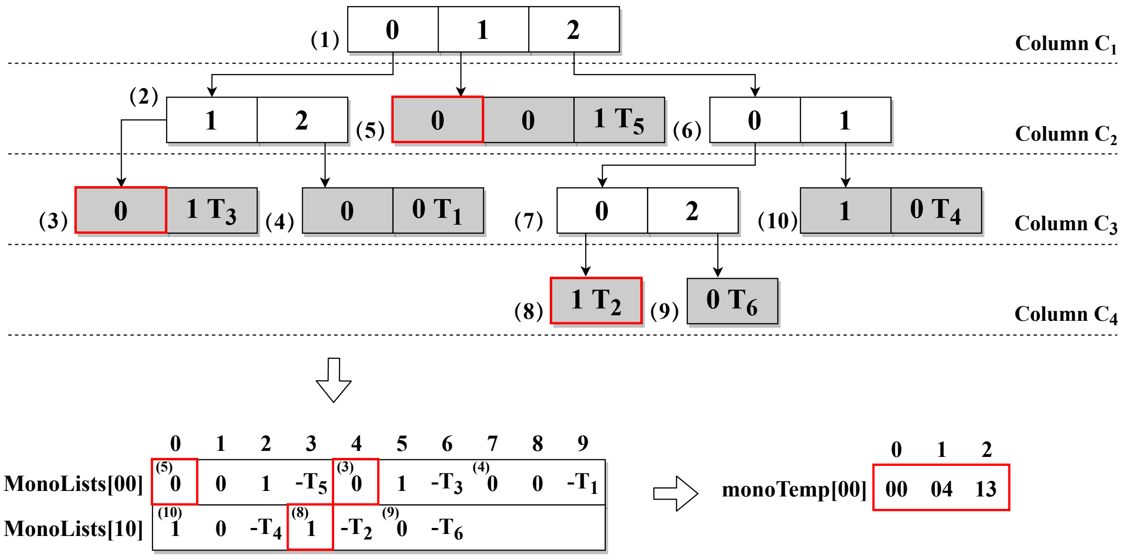


Figure 3.19: Method of processing MonoLists (2) - First visited MonoList

As shown in Figure 3.19, we store the position of the first MonoList in each dimension in a temporary vector `monoTemp` during the construction process. Since the entire first dimension constitutes a large DimensionList, there is no MonoList in the first dimension. Therefore, `monoTemp`[01] in the temporary vector is the start pointer of MonoLists on C_3. In the process of matching the MonoLists of the second dimension, the last position we visit in `MonoLists` is `MonoLists`[04] and the first two values in the `monoTemp`.

3.3.2.3 MonoList with Length

This method is similar to the way of Elf_Level_Separated_Length stores DimensionLists. We add sizes of the MonoList to the front of each MonoList. This will tell us until which MonoList we have to check. As shown in Figure 3.20, we added the size of each MonoList to the memory layout of `MonoLists` based on Figure 3.7. In this way, we can determine where we terminate the search directly by comparing the first value of each MonoList. For example, we find that `MonoLists`[00]>2, so the position of this `MonoLists` is before C_3, and then we compare with `MonoLists`[05], we can find that its value is 2, which means we have reached C_3. This process can end. In addition, we can also directly determine the position of the data we want to match in the DimensionList through the Length. For example, we know that the length of MonoList on C_3 is 2. When we traverse to `MonoLists`[00], the length of L_5 we get is 3, so the position of the data we need to match in this MonoList (L_5) is `MonoLists`[1+(3-2)], that is, `MonoLists`[02].

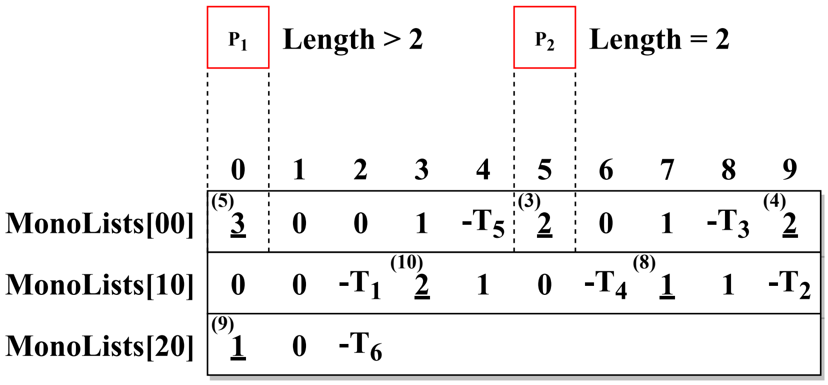


Figure 3.20: Method of processing MonoLists (3) - MonoList with Length

3.3.2.4 Compare and Summary

For the first method, there is no need for extra auxiliary methods. We only need to compare the length of the MonoList during the traversal process, but we must traverse one more MonoList each time to ensure that it meets the termination condition. For the second method, we need an additional vector to store the start pointer of `MonoLists`. In addition, during the construction process, we need to determine whether it is the first MonoList in the dimension when we add the MonoList to `MonoLists`. In terms of function, it only provides a start pointer. For the third method, we only need to add the size of the MonoList in front of each MonoList, which is equivalent to simplifying the first method. It can also quickly determine the start pointer of the dimension in `MonoLists` without traversing the entire `MonoLists` because we can skip MonoList according to the length. In addition, we can also use this length to directly process a value in the MonoList. Therefore, after we compared these three methods, we chose MonoList with length as the solution.

3.3.3 Introduction of Pseudo-Code

In this chapter, we introduce the Pseudo-Code of the new PM algorithm based on the query type. For a query, its predicate either contains the first dimension or does not contain it.

1. If the query predicate contains the first dimension, there are two cases: mono-column select predicate query (one-dimensional query) and multi-column select predicate query (multi-dimensional query). We respectively introduce the algorithms for handling these two types of queries.

2. If the query predicate does not contain the first dimension, there are also two cases. The first case is that the first selection predicate in the Elf variant contains DimensionLists. The second case is that the first selection predicate only contains MonoLists. Both of these cases involve the processing of MonoLists. Therefore, we will first introduce the algorithm for processing MonoLists with different cases, and then introduce the algorithm for processing DimensionLists with the first case.

In addition, we use each type of query as the title of the algorithm. For example, for a multi-dimensional query whose predicate contains the first dimension, we named the section title as the first dimension as the first selection predicate. Correspondingly, there is only one predicate for a one-dimensional query whose predicate contains the first dimension. In order to indicate that the algorithm is for a one-dimensional query, we named its section name as the first dimension as the last dimension. Our introduction to each query type algorithm is based on the order from the first dimension to the last dimension. For example, we process the first dimension first, then the nth dimension, and finally the n+1th dimension, etc.

3.3.3.1 The First Dimension as Last Selected Predicate

In Algorithm 3.8 we showed the algorithm `firstDimAsLastSelected`. When the first dimension is used as the predicate in a mono-column selection predicate, we use this algorithm to traverse the data on the first dimension.

Algorithm 3.8: PartialMatch algorithm for the first dimension (1)

```
Input: A class, which contain information about the query data query,
       Scope of cutoffs cutoffs
Output: A vector that stores all TIDs that meet the query conditions
        resultTIDs
1  Function firstDimAsLastSelected(cutoffs, &query[]):
2      if cutoffs > FIRST_DIM then
3          if lastSelected == FIRST_DIM then
4              lower ← max(query.lowerBound[0], minDims[0]);
5              upper ← min(query.upperBound[0], maxDims[0]);
6              while !getPointer(lower, 0) && lower ≤ upper do
7                  lower ← lower + 1;
8              end
9              if lower > upper then
10                 return resultTIDs[];
11             end
12             do
13                 upper ← upper + 1;
14             while !getPointer(upper, 0) && upper ≤ maxDims[0]
15             if upper > maxDims[0] then
16                 endTID ← NumPoints;
17                 resultTIDs[] ← vector(getTIDs(lower), getTIDs(endTID));
18             else
19                 endTID ← upper;
20                 resultTIDs[] ← vector(getTIDs(lower), getTIDs(endTID));
21             end
22             return resultTIDs[];
23         end
24     end
25 End Procedure
```

The two parameters we pass into the function are `query` and `cutoffs`. The `query` is an object of the Query class. This class stores all the information about the query statement, such as which selection predicates are selected and the conditions of the selected predicates. `cutoffs` represent the range of dimensions with cutoff-pointers. When the first dimension is used as the predicate of a single-column selection predicate query, it will be more conveniently and quickly if we use Cutoffs to obtain the result set. This is because we can find the corresponding TIDs as long as we find the upper and lower boundaries of the query predicate.

We first find the intersection of the upper and lower bounds of the predicate and the dimension value range of the first dimension `[minDims[0], maxDims[0]]` (Lines 4-5). Because we use a hash map to store pointers in the first dimension, not all values in the first dimension have pointers. Therefore, we need to retrieve the range of intersection to find the first dimension value that stores a valid pointer. We use the same way to find the real upper boundary (Lines 6-14). Once we have determined the real upper and lower boundaries, we can take all TIDs in the range of the two cutoff-pointers and store them in the `resultTIDs`(Lines 15-22). If there is no cutoff-pointer, we will call the traversal algorithm. The algorithm starts from the first dimension and traverses until it finds all TIDs that meet the conditions.

3.3.3.2 The First Dimension as First Selected Predicate

Algorithm 3.9: PartialMatch algorithm for the first dimension (2)

```
   Input: query, lower, upper
   Output: resultTIDs
1  Function firstDimAsFirstSelected(lower, upper, &query[]):
2      if firstSelected == FIRST_DIM then
3          while lower ≤ upper do
4              nextDimPointer ← getPointer(lower, FIRST_DIM);
5              if nextDimPointer == NOT_FOUND then
6                  lower ← lower + 1 ;
7                  continue;
8              nextEntry ← lower + 1;
9              while getPointer(nextEntry, FIRST_DIM)==NOT_FOUND do
10                 nextEntry ← nextEntry + 1;
11             end
12             if !isMonoListPointer(nextDimPointer) then
13                 partialMatch(1, nextDimPointer, resultTIDs, query);
14             else
15                 partialMatchMonoList(1, nextDimPointer, resultTIDs, query);
16             end
17             lower ← nextEntry;
18         end
19     end
20 End Procedure
```

In Algorithm 3.9 we showed the algorithm `firstDimAsFirstSelected`. When the first dimension is used as the predicate in a multi-column selection predicate, we use this algorithm for queries.

There are three parameters in this algorithm, which are `query`, `lower` and `upper`. We have already introduced the object `query` in the previous algorithm. `lower` is the maximum value between the lower boundary of the predicate and the minimum value of the first dimension. `upper` is the minimum value between the upper boundary of the predicate and the maximum value of the first dimension. We still need to confirm that our query boundary is valid, that is, a pointer is stored under the corresponding dimension value (Lines 4-8). After confirmation, we sequentially follow the dimension values of the first dimension until the upper boundary is reached (Line 9-17).

3.3.3.3 MonoLists before the First Selected Dimension

Algorithm 3.10 is suitable for queries that use a non-first dimension as the first selection predicate, and this dimension contains the DimensionLists. It and Algorithm 3.12 form a complete algorithm for processing such queries. This algorithm only handles all MonoLists that existed before that dimension. We need four parameters in this algorithm 3.10. `dimStartPointer` stores the start pointer of each dimension. `MONO_LISTs` store all MonoLists. `firstSelected` represents the first selected dimension. `query` is an instance object of class Query.

Algorithm 3.10: PartialMatch algorithm for MonoLists (1)

```
Input: dimStartPointer[], MONO_LISTS[], firstSelected, query
Output: resultTIDs[]
Function monoListsBeforeSelectedDim(dimStartPointer[],
 MONO_LISTS[], firstSelected, query):
    if firstSelected > FIRST_DIM then
        pos ← 0;
        monoLength ← getMonoLength(pos);
        startDim ← NUM_DIM - monoLength;
        while startDim ≤ firstSelected && pos < MONO_LISTS.size() do
            partialMatchMonoList(startDim, pos, resultTIDs, query);
            pos ← pos + monoLength;
            while !isLastEntryMaskMono(getTID(pos)) do
                pos ← pos + divideRoundNext();
            end
            pos ← pos + 2;
            if pos < MONO_LIST.size() then
                monoLength ← getMonoLength(pos);
                startDim ← NUM_DIM - monoLength;
            end
        end
    end
End Procedure
```

In this algorithm, we find the length of a MonoList stored in `MONO_LISTs`, and then use it to calculate the dimension information (Lines 2-4). We can use dimension and offset to control the end condition of the loop. Every time we finish processing a MonoList, we reset the dimension information and offset so that the next MonoList can be processed (5-14). These new PM algorithms are all adapted to MonoList with length.

3.3.3.4 Algorithm for MonoLists only

Algorithm 3.11 is suitable for queries that use a non-first dimension as the first selection predicate, and there are only MonoLists in this dimension (such as C_4 in Figure 3.2). In this case, we only need to deal with the `MonoLists`. We show pseudocode in Algorithm 3.11. It can be observed that the logic of the algorithm is the same as that of the algorithm 3.10, only minor adjustments are made in the judgment conditions. Algorithm 3.10 is to process the MonoLists before the specified dimension, while algorithm 3.11 is to process all MonoLists. After processing, all TIDs that meet the query conditions are returned.

Algorithm 3.11: PartialMatch algorithm for MonoLists (2)

```
Input: dimStartPointer[], MONO_LISTS[], firstSelected, query
Output: resultTIDs[]
1  Function monoListsInSelectedDim(dimStartPointer[], MONO_LISTS[],
    firstSelected, query):
2      if firstSelected > dimStartPointer.size() - 1 then
3          pos ← 0;
4          monoLength ← getMonoLength(pos);
5          startDim ← NUM_DIM - monoLength;
6          while pos < MONO_LISTS.size() do
7              partialMatchMonoList(startDim, pos, resultTIDs, query);
8              pos ← pos + monoLength;
9              while !isLastEntryMaskMono(getTID(pos)) do
10                 pos ← pos + divideRoundNext();
11             end
12             pos ← pos + 2;
13             if pos < MONO_LIST.size() then
14                 monoLength ← getMonoLength(pos);
15                 startDim ← NUM_DIM - monoLength;
16             end
17         end
18         return resultTIDs[];
19     end
20 End Procedure
```

3.3.3.5 Non-first Dimension as the First Selected Predicate

Algorithm 3.12 is suitable for queries that use a non-first dimension as the first selection predicate, and this dimension contains the DimensionLists. It implements

Algorithm 3.12: PartialMatch algorithm for the n-th Dimension

```
   Input: firstSelected, lastSelected, lower, upper, query,
          dimStartPointer[]
   Output: resultTIDs[]
1  Function nDimAsFirstSelectedDim(firstSelected, lastSelected, lower,
    upper, query, dimStartPointer[]):
2      if firstSelected != FIRST_DIM then
3          if firstSelected + 1 < dimStartPointer.size() then
4              dimEnd ← dimStartPointer[firstSelected + 1] - 1;
5          else
6              dimEnd ← Child_Pointer.size() - 1;
7          end
8          dimStart ← dimStart - firstDimEndPointer;
9          dimEnd ← dimEnd - firstDimEndPointer;
10         while dimStart ≤ dimEnd do
11             nextDimPointer ← getPointer(dimStart, firstSelected);
12             dimValue ← getValue(dimStart, false);
13             if !isIn(lower, upper, dimValue) then
14                 dimStart ← dimStart + 1;
15                 continue;
16             end
17             nextEntry ← dimStart + 1;
18             if isLastEntryPointer(nextDimPointer) then
19                 if !isMonoListPointer(nextDimPointer) then
20                     temp ← getDimTID(firstSelected + 1, nexDimPointer);
21                 end
22             else
23                 temp ← getCutoffPointer(nextEntry, firstSelected);
24             end
25             if !isMonoListPointer(nextDimPointer) then
26                 if ENABLE_Cutoff then
27                     partialMatch(firstSelected + 1, nextDimPointer,
                        resultTIDs, query, lastSelected, temp);
28                 else
29                     partialMatch(firstSelected + 1, nextDimPointer,
                        resultTIDs, query);
30                 end
31             else
32                 partialMatchMonoList(firstSelected + 1, nextDimPointer,
                    resultTIDs, query);
33             end
34             dimStart ← nextEntry;
35         end
36     end
37 End Procedure
```

the main function of the new PM algorithm, that is, directly jumps to the first predicate for PM queries. We use algorithm `monoListsBeforeSelectedDim` in the algorithm 3.10 to match all MonoLists before the first predicate.

After processing the MonoLists of the first $n-1$ dimensions, we jump directly to the n-th dimension. First, because the first selected dimension contains DimensionLists and MonoLists, we need to determine the end pointer of this dimension (Lines 3-7). Since the new PM algorithm is only applicable to Elf variants other than Elf64_Level, if we want to obtain the dimension value of the dimension value, we need to subtract the size of the first dimension from the pointer data (Lines 8-9). This is because the data structure used to store the DimensionLists does not contain the dimension value of the first dimension.

Next, we traverse this dimension. The most important point is that this algorithm only processes all DimensionLists in that dimension. Therefore, we only need to traverse the data structure used to store the DimensionLists. The dimension value of each DimensionList is ordered, but when the dimension value of the DimensionList of the entire dimension is placed in the same data structure, it is out of order. We need to obtain each dimension value one by one to determine whether it is in the upper and lower boundaries of the predicate (Lines 11-16). If we use Cutoffs, we need to get the cutoff-pointer in advance to get more accurate results (Lines 18-24). Because in this process, we will provide a function `getDimTID` that can query the last cutoff-pointer of each DimensionList. The reason for using this function is that when we use the cutoff-pointers of the two DimensionLists on the n-th dimension to form an interval, some TIDs contained in this interval may come from the processed MonoLists in the first $n-1$ dimensions. After obtaining the relevant cutoff-pointers, we select the corresponding function according to the requirements (Lines 25-33). Finally, we matched all the dimension values of the dimension.

3.4 Summary

In this chapter, we give a comprehensive introduction to our tasks and contributions. We first introduced the BFS and DFS algorithms. By comparing these two well-known graph algorithms, we can initially understand our task requirements, that is, which aspects need to be adjusted and changed compared to the standard Elf. Subsequently, we introduced the conceptual design of the level order linearization for the Elf approach. It simplifies the process of level order linearization, then displays it in the form of graphs and analyzes them briefly. Then, based on the conceptual model, we introduced the algorithm for level order linearization in detail in the implementation part. These contents can be regarded as the answer to the first scientific question RQ1 raised in Section 1.1. Finally, we introduce the partial Match Query algorithm adapted to the level order linearization algorithm. In this part, we answer the second research question RQ2. In addition, We complete the introduction of these two main contributing algorithms by analyzing pseudo-code.

4. Evaluation

In Chapter 3, we introduced level order linearization for Elf Approach, which is the contribution of our thesis. In the next step, we check whether this linearization is meaningful in terms of performance of the Elf Approach. In addition, we also verify the theoretical advantages and disadvantages of level order linearization, we introduced in Chapter 3. Before conducting evaluation experiments, we explain the experimental environment, experimental settings and procedures. Then we introduce our evaluation results, analyze the data and discuss the subjective and objective factors of these results.

4.1 Framework of the Experiment

Before we conduct the experiment, we introduce the framework of our experiment in this section. It mainly consists of three parts. The first is the experimental environment. We introduce the hardware facilities to perform the evaluation. Then there is the data for evaluation. The last is the subject of our evaluation, including evaluation objects and evaluation types.

4.1.1 Experimental Environment

The experiment will be performed on a machine with CentOS Linux release 7.3.1677 (Core). This machine is equipped with an Intel(R) Xeon(R) CPU E5-2630 v3 with 2.4 GHz base clock. This CPU has 8 cores and 16 threads. In addition, the memory size of this machine is 1TB.

4.1.2 Data for Evaluation

The evaluation experiment uses the TPC-H benchmark test. TPC-H is a decision support, transaction processing and database benchmark. It consists of business oriented ad-hoc queries and concurrent data modification. In addition, it defines 8 tables, 22 queries and follows SQL92. The database model of the TPC-H benchmark follows the third normal form. In this experiment, we do not perform a cross-table query, so we only choose one data table `Lineitem` as the data table for evaluation. In order to make a horizontal comparison, we evaluate the Lineitem tables of different sizes separately. The size of these data tables is 1GB, 10GB, 50GB and 100GB.

4.1.3 Evaluation Objects

The Elf constructed by level order linearization is the main contribution of this thesis, so we mainly evaluate it. Because our main purpose is to verify whether the level order linearization approach is meaningful for the performance of Elf, we need to evaluate the vertical linearization at the same time under the same experimental environment. By comparing the evaluation results of vertical linearization and level order linearization, we can more intuitively observe the impact of the level order linearization approach on the performance of Elf. We select four representative Elf variants, which are the standard Elf, Elf_Separated, Elf_Separated_Length, Elf_SIMD. Similarly, for level order linearization we select the corresponding four Elf variants.

4.1.4 Evaluation Type

The impact of the level order linearization approach on the performance of Elf is mainly reflected in the construction and query. Therefore, we mainly evaluate the construction of Elf and queries that use Elf as an index.

4.1.4.1 Construction Evaluation

The level order linearization approach is a new linearization type for the Elf approach, and it is necessary to conduct a comprehensive evaluation for its construction. We mainly evaluate the two aspects, memory consumption and build time. The main way of our evaluation is to construct the same data file in the same environment with the level order linearization method and the vertical linearization method. Then we analyze the acquired data to show some advantages of level order linearization.

4.1.4.2 Query Evaluation

Regarding query evaluation, we separately evaluate mono-column select predicate queries and multi-column select predicate queries. Because the main purpose of our evaluation is to verify whether the Elf approach can benefit from the level order linearization, we designed several new TPC-H benchmark query statement for query evaluation. These query statements are adapted from `Q1`, `Q6`, `Q10` and `Q19`. They may not have practical significance, but it can support our evaluation work well.

4.2 Experiment

After introducing the experimental framework, in this chapter, we introduce the specific evaluation work, analyze the evaluation data and summarize the evaluation results. According to the evaluation type, we divide this chapter into two parts, construct evaluation and query evaluation.

4.2.1 Construction Evaluation

In this chapter, we evaluate the construction of vertical linearization and level order linearization for Elf approach in the same environment. The Elf variants we mainly evaluated are the construction of Elf and Elf_Separated, and Elf_Separated_Length. The construction algorithm of Elf_Level_SIMD is based on Elf_Level_Separated_Length. Their construction process is almost the same. In terms of construction, the performance of Elf_SIMD is almost the same as Elf_Separated_Length, so we only show the evaluation data of Elf_Separated_Length. For the construction of Elf, we evaluate them from two aspects, storage consumption and construction time.

4.2.1.1 Storage Consumption

We use the TPC-H data file *Lineitem* with a size of 10GB as experimental data to evaluate storage consumption. The reason we chose this data file is that for the constructed Elf and Elf variants, there are some observable rules between them. For example, if we observe the data structure ELF in Figure 3.5 and the Separated data structures in Figure 3.7, we can find that it has the following rules: Size(`ELF`)=Size(`Elf` + `Child_Pointers` + `Cutoff_Pointers` + `MonoLists`). This means that if the level order linearization approach can optimize storage consumption, it will certainly feedback this optimization through the size of these data structures. Regardless of the size of the data file we choose, as long as the amount of data in the data file is sufficient to support feedback such changes, the impact of the level order linearization approach can be well confirmed. Therefore, when we evaluate storage consumption, it is reasonable for us to use a 10GB data file for evaluation. In the experiment, we will separately analyze the case of without Cutoffs and with Cutoffs.

1. Without Cutoffs

In Figure 4.1 and Figure 4.2, we both show the storage consumption of Elf64, Elf_Separated and Elf_Separated_Length constructed from a 10GB TPC-H data file using the vertical linearization method, and the storage consumption of Elf64_Level, Elf_Level_Separated and Elf_Level_Separated_Length constructed using the level order linearization approach.

We use the red line in the figures to mark the Elf variant with the greatest storage consumption. In addition, we introduced *Lengths* for MonoLists in the Elf variant of Level order linearization (e.g. Elf_Level_Separated) and explained the *Lengths* in Section 3.3.2.3. In order to compare the difference in storage consumption before and after the *Lengths* is introduced. We show the storage consumption of the Elf variant without Lengths in Figure 4.1. The storage consumption of the Elf variant with the Lengths is shown in Figure 4.2. Since we only set Lengths for the level order linearized variants, the data for the vertically linearized Elf variants are the same in these two figures.

When we index the same data file, whether we use the vertical linearization method or the level order linearization method to construct the corresponding Elf, the number of DimensionLists and MonoLists in this data file must be fixed. Therefore,

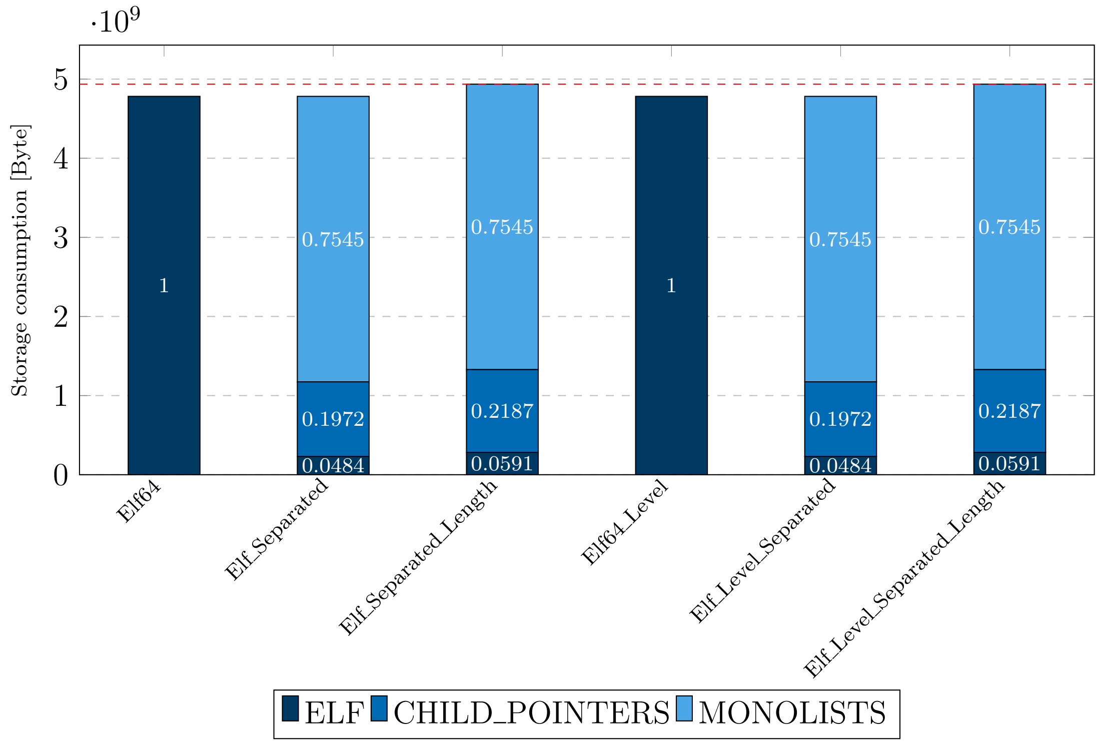


Figure 4.1: The storage consumption required for the data structure of all Elf variants when indexing a 10GB TPC-H table `Lineitem` (without Cutoffs and without *Lengths* of `MonoLists`)

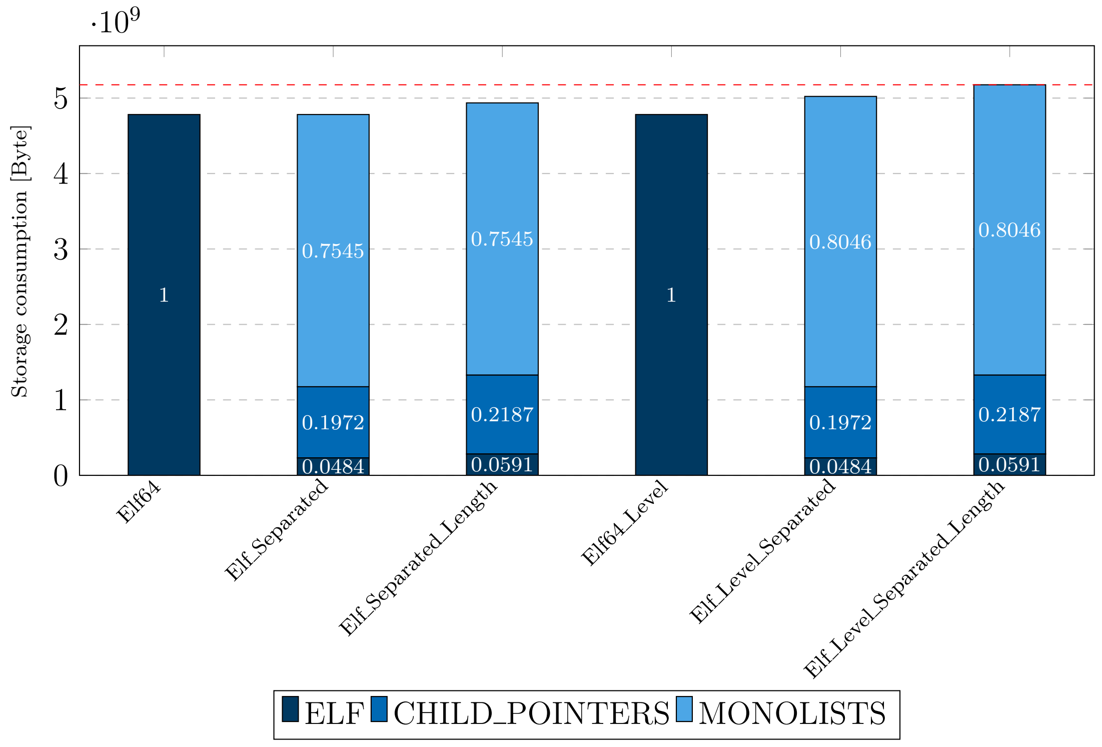


Figure 4.2: The storage consumption required for the data structure of all Elf variants when indexing a 10GB TPC-H table `Lineitem` (without Cutoffs and with *Lengths* of `MonoLists`))

without Cutoffs, the storage consumption of the Elf variant constructed by the level order linearization method should be the same as the storage consumption of the corresponding Elf variant constructed by the vertical linearization method. We can observe in Figure 4.1 that the storage consumption of the corresponding variants of the vertical linearization method and the level order linearization approach is the same. For example, the ratio of ELF in Elf_Separated and Elf_Level_Separated is 0.484. This ratio is obtained by dividing the size of a single data structure in Separate by the size of ELF in Elf64 or Elf64_Level (e.g., ratio(Separated.`MONOLISTS`) = size(Separated.`MONOLISTS`)/size(Elf64.`ELF`)).

However, when we use the level order linearization approach to construct an Elf Separated variant, we will add the size of the MonoList in front of each MonoList. We have introduced this in Section 3.3.2. Then, the storage requirement of `MONOLISTS` in Elf_Level_Separated and Elf_Level_Separated_Length should be greater than Elf_Separated and Elf_Separated_Length. This is consistent with the performance of `MONOLISTS` in Elf_Level_Separated and Elf_Level_Separated_Length in Figure 4.2 (i.e., $0.8046 > 0.7545$). The number of *Lengths* stored in MonoLists in Elf_Level_Separated is equal to the number of MonoLists. In addition, the number of MonoLists will not exceed the number of rows, that is, the range of the number of *Lengths* is (0, M] (M is the total number of rows or the number of TIDs).

Therefore, without Cutoffs, the storage requirements of the standard Elf (Elf64_Level) constructed by the level order linearization approach are the same as those of the standard Elf (Elf64) constructed by the vertical linearization method. This is because the number of MonoLists and DimensionLists in the same data file is equal. For other Elf variants without Cutoffs, since `MONOLISTS` adds Lengths, the storage requirement of the level order linearization approach is slightly higher than that of the vertical linearization approach.

2. With Cutoffs

In Section 3.1.3.2, we have discussed the theoretical advantages of the level order linearization approach in terms of Cutoffs compared to the vertical linearization approach. Next, we set the cutoff-pointers for the specified dimensions (from the first dimension to the n-th dimension, $n \in [1, N]$, N is the total number of dimensions) under the same experimental environment. We set the variable *cutoffs* to 1, which means we add the cutoff-pointers for the first dimension.

In Figure 4.3, we can observe that the value of Cutoffs in the Elf variant constructed by vertical linearization is much greater than the value of Cutoffs in the Elf variant constructed by level order linearization. For example, the ratio of `CUTOFF_POINTERS` in Elf_Separated is 0.158, and the ratio of `CUTOFF_POINTERS` in Elf_Level_Separated is 0.0805. The numbers in the corresponding data structure are 117,842,192 (0.158) and 60,000,002 (0.0805). The absolute value obtained by subtracting these two values is 57,842,190 (converted into a ratio, it is 0.0388). From Figure 4.3, we can find two data structures with a ratio of 0.0388, that is, the ratio of `ELF` in Elf_Separated and Elf_Level_Separated. For the Elf variant, we know that `ELF` only stores all DimensionLists except the first dimension, while `CUTOFF_POINTERS` stores cutoff-pointers for all DimensionLists. Therefore, the value of |`CUTOFF_POINTERS - ELF`| is the size of the first dimension (60,000,002). From this, we can know that

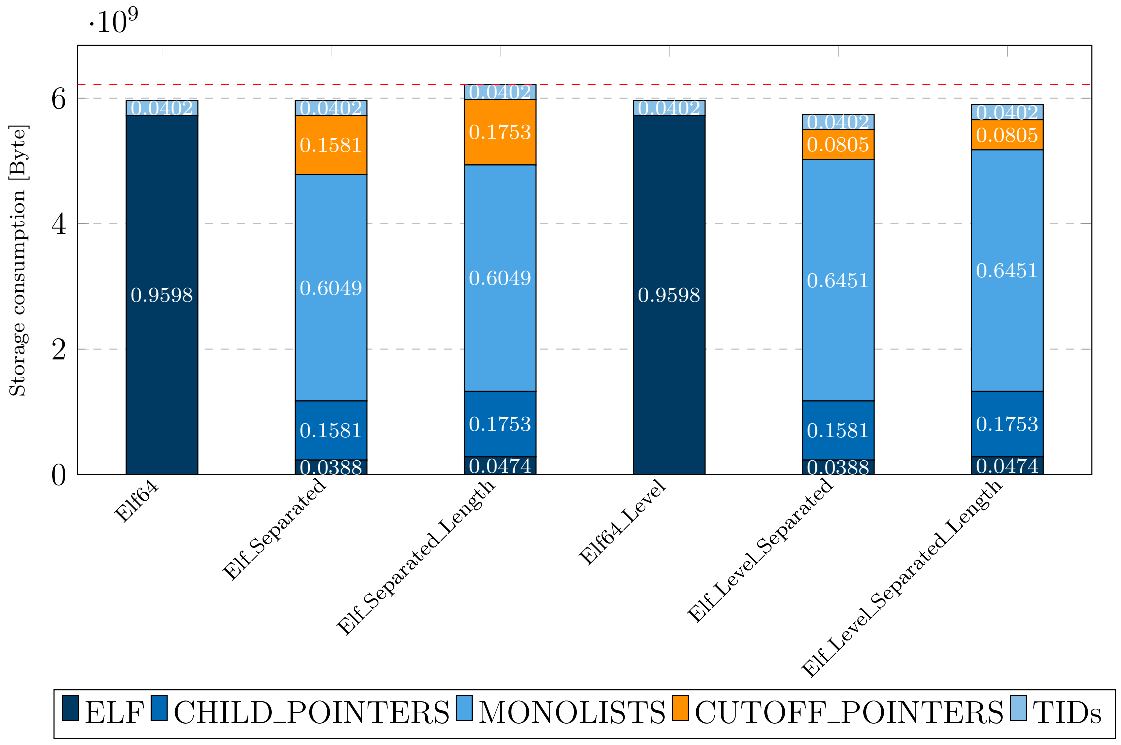


Figure 4.3: The storage consumption required for the data structure of all Elf variants when indexing a 10GB TPC-H table `Lineitem` (with cutoffs=1)

the `CUTOFF_POINTER` in Elf_Level_Separated and Elf_Level_Separated_Length only stores the cutoff-pointers of the first dimension. In addition, because the data structure `ELF` and the data structure `CUTOFF_POINTERS` store different data types, the ratio of the corresponding storage space size cannot be directly used for calculation.

However, the data types stored in `TIDs` and `MONOLISTS` are the same, and we can use ratios to directly perform calculations to express the relationship between data structures. For example, `MONOLISTS` (0.6451) in Elf_Level_Separated has stored *Lengths*. As we have introduced, the number of *Lengths* is not bigger than the number of `TIDs`. We add the ratio of `TIDs` (0.0402) and `MONOLISTS` (0.6049) in Elf_Separated to get 0.6451, which is the ratio of MONOLISTS (0.6451) in Elf_Level_Separated.

cutoffs = 2

We carried out the experiment with *cutoffs* = 2 in the same environment. We show the obtained evaluation data in Figure 4.4. We compared it with the evaluation data in Figure 4.3 and found that, except for the size of `CUTOFF_POINTERS` in Elf_Level_Separated and Elf_Level_Separated_Length, the size of the data structure of other Elf variants has not changed. In addition, when cutoffs=2, we set the cutoff-pointers for the first dimension and the second dimension. Therefore, the number of extra pointers in `CUTOFF_POINTERS` in Elf_Level_Separated and Elf_Level_Separated_Length is the number of cutoff-pointers in the second dimension. Although in Figure 4.4, the ratio of `CUTOFF_POINTERS` in the corresponding Elf variants is the same, but these data are not the same in the size of the specific data structure. They are just very close. For example, the size of `CUTOFF_POINTERS` of Elf_Separated is 130,698,330, and the size of `CUTOFF_POINTERS` of

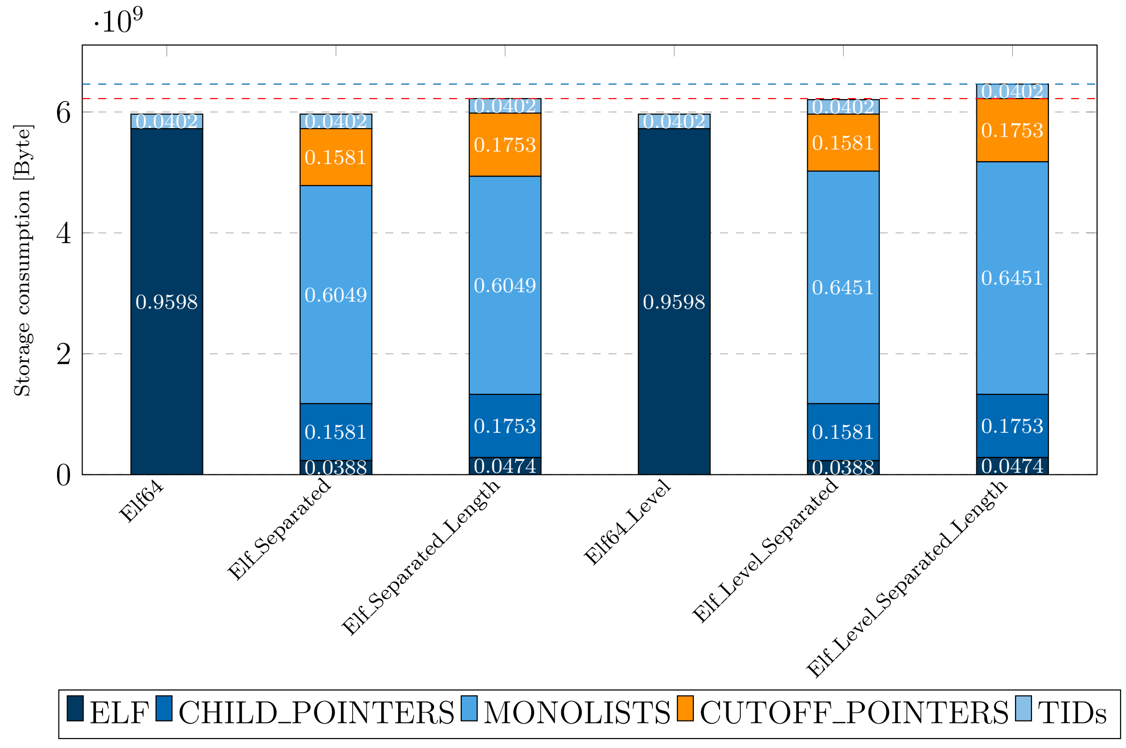


Figure 4.4: The storage consumption required for the data structure of all Elf variants when indexing a 10GB TPC-H table `Lineitem` (with cutoffs=2)

Elf_Level_Separated is 130698160. The difference between them is only 170 values. This value is very insignificant compared to their base (10^8). However, the existence of this value has a very important meaning. It means that there are still dimension columns in the third dimension. Since the size of Cutoffs only affects the size of `CUTOFF_POINTERS` in the Elf variant constructed by the level order linearization approach, we show in Figure 4.5 the size of `CUTOFF_POINTERS` from cutoffs=0 to cutoffs=16 for all Elf variants.

From Figure 4.5, we can observe that after adding the cutoff-pointers for the fourth dimension, the size of the `CUTOFF_POINTER` of Elf_Level_Separated and Elf_Level_Separated_Length is completely equal to that of Elf_Separated and Elf_Separated_Length. When cutoffs>4, the size of all data structures of all Elf variants is fixed. At the same time, this also means that from the fifth dimension there is no DimensionList for this data file (`Lineitem` table with a size of 10GB). Because if there are DimensionLists in the fifth dimension, we can add cutoff-pointers for them. Then, the size of `CUTOFF_POINTER` will be changed. However, we observe from the obtained evaluation data that this size no longer changes from the fourth dimension. This confirms that there are only Monolists from the fifth dimension to the last dimension. In the first three dimensions, we can observe the positive impact of the level order linearization approach on Cutoffs compared to the vertical linearization approach. Especially when the cutoffs $\in [0, 2]$, we can intuitively observe that the level order linearization approach only reserves space for the specified dimensions to add the cutoff-pointers. For vertical linearization, space must be reserved for all

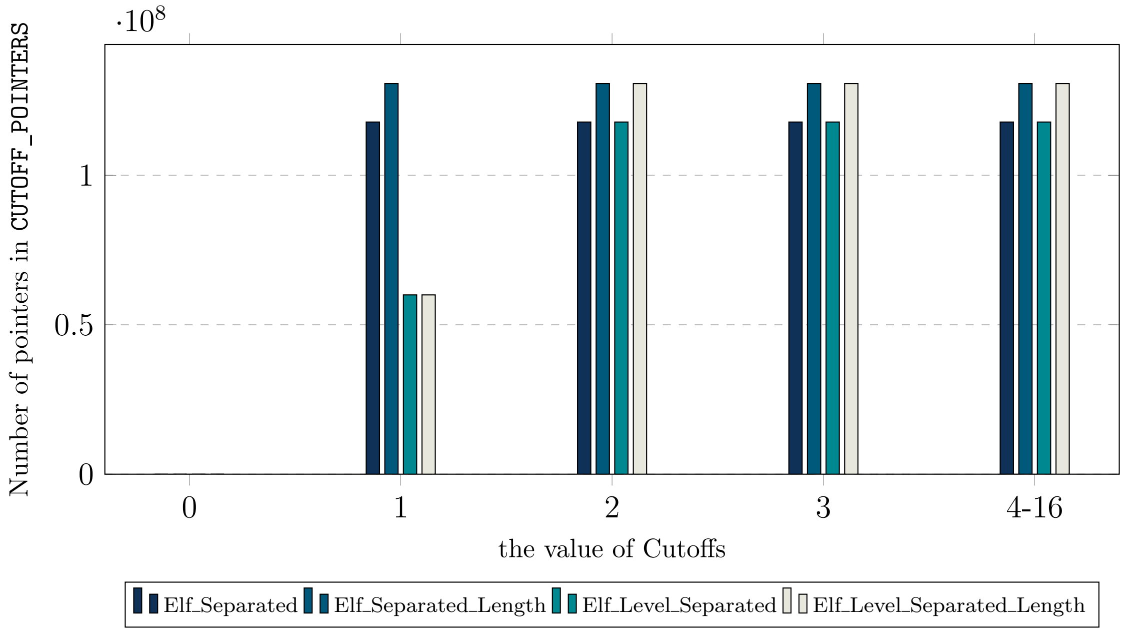


Figure 4.5: The size of `CUTOFF_POINTERS` from cutoffs=0 to cutoffs=16 for all Elf variants

dimensions due to the limitation of the depth-first algorithm, so gaps are generated. Level order linearization approach does not have this problem.

Therefore, when we add cutoff-pointers to data structure `MONOLISTS`, the Elf variant constructed by the level order linearization approach does not reserve space for cutoff-pointers of unselected dimensions in advance. In addition, when all the cutoff-pointers are added, due to the addition of *Lengths* in `MONOLISTS`, the storage space required for the Elf64_Level variant is slightly higher than that of the Elf64 variant, but this size (size of Lengths) does not exceed the number of TIDs. If the *Lengths* in `MONOLISTS` are not considered and after adding all the cutoff-pointers, the storage consumption of all the data structures of the Elf variant constructed by the level order linearization approach is exactly the same as that of the vertical linearization.

4.2.1.2 Construction Time

The construction time is the second evaluation indicator of the construction, and we introduce it in this chapter. We first evaluate the construction time of Elf variants without Cutoffs and then evaluate the construction time of Elf variants with Cutoffs. Since we will index `Lineitem` tables with different sizes, and the data of each table is not exactly the same, we cannot determine which dimension of each table is the dimension that contains the last few DimensionLists (e.g., the 4th dimension of the `Lineitem` table with a size of 10GB). In order to ensure the accuracy of the experiment, when we perform the construction experiment of the Elf variant with Cutoffs, we uniformly add the cutoff-pointers for all dimensions, that is, set cutoffs=16.

1. Without Cutoffs

We use readTSV to read `Lineitem` tables with sizes of 1GB, 10GB and 100GB respectively, and use the vertical linearization approach and the level order lin-

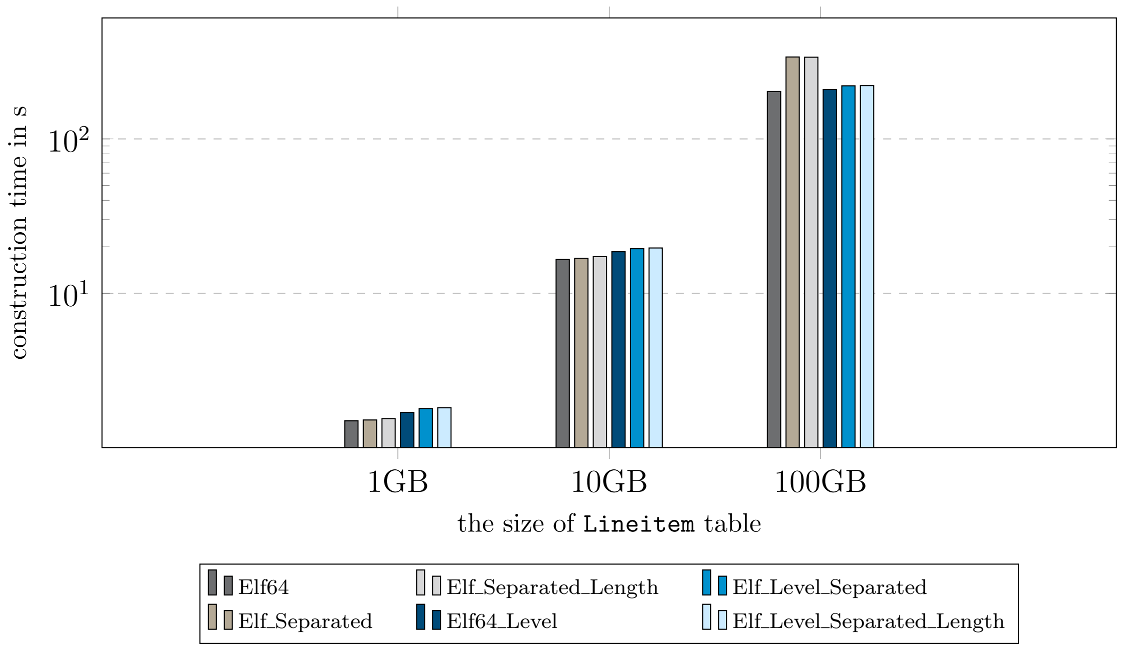


Figure 4.6: Construction time of all Elf variants without Cutoffs (readTSV)

earization approach to construct the Elf index structure for these tables in the same experimental environment. We show the evaluation data in Figure 4.6 and Table 4.1.

Elf variants	1GB		10GB		100GB	
`Elf64`	1.49	11.8%	16.57	10.8%	202.74	2.8%
`Elf_Separated`	1.51	15.6%	16.85	13.3%	339.51	-
`Elf_Separated_Length`	1.54	14.9%	17.26	12.2%	338.26	-
`Elf64_Level`	1.69	-	18.58	-	208.71	-
`Elf_Level_Separate`	1.79	-	19.43	-	220.85	34.9%
`Elf_Level_Separated_Length`	1.82	-	19.66	-	221.41	34.5%

Table 4.1: The construction time (s) of Elf and Elf variants for `Lineitem` tables with different sizes (readTSV and without Cutoffs)

In Table 4.1, we also provide a percentage of speedup for variants that have a faster build time under the same conditions. For example, in the same environment to build Elf64 and Elf64_Level for a 1GB `Lineitem` table at the same time, building Elf64 is 11.8% faster than building Elf64_Level. We can observe from Table 4.1 that when building an Elf index for a 1GB data table, the construction speed of the vertical linearization approach is 14.1% faster than the level order linearization approach on average. Then, when we construct Elf and Elf variants for the 10GB `Lineitem` table, the vertical linearization approach is still better than the level order linearization approach in terms of construction speed. The construction speed of the vertical linearization approach is 12.1% faster than that of the level order linearization approach on average. However, compared with its advantage of 14.1% faster than the level order linearization approach when indexing 1GB `Lineitem` tables, 12.1% indicates that its speedup trend is downward when indexing larger data tables. When we built Elf and Elf variants for the 100GB `Lineitem` table, we found interesting changes in the construction time. For Elf64 constructed by vertical linearization approach, its construction speed is only 2.86% faster than the construction speed of Elf64_Level constructed by level order linearization approach. However, when constructing Elf variants for the same data table, the level order linearization approach shows its great advantages. Its construction speed is 34.7% faster than the vertical linearization approach to construct Elf variants on average.

Verify the authenticity of the evaluation data for the 100GB data file

From Figure 4.6, we can observe that the time for the vertical linearization approach to construct the standard Elf (202.74s) and the time to construct the Elf variant (average 338.88s) deviates too much, while the deviation of the construction time between the different variants constructed by level order linearization is not large. In order to verify the validity of the evaluation data of the level order linearization approach, we evaluated the storage consumption of the Elf and Elf variants constructed by level order linearization approach. The storage consumption of the Elf64 and Elf64_Level we constructed for the 100GB data table are both 47,831,355,824 Byte. Both of their data structures contain 11,957,838,956 elements. In addition, the Elf variant constructed by the level order linearization approach also satisfies the formula mentioned in Section 4.2.1.1. We pass the sizes of `ELF`, `TIDs`, `Child_Pointers`, `Cutoff_Pointers` and `MonoLists` in Elf_Level_Separated in Table 4.2 into the following formula: `ELF`+`Child_Pointers`+`Cutoff_Pointers`+`MonoLists`-`TIDs`. We can get the value 11,957,838,955. This value is equal to the size of the data structure `ELF` in Elf64_Level. This also proves that construction evaluation for the 100GB `Lineitem` table is effective. The "-" in Table 4.2 means that this Elf variant does not have this data structure. To facilitate calculations, we added the number of cutoff-pointers. It is equal to the number of pointers in `Child_Pointers`.

	Elf64_Level	Level_Separate	Level_Separated_Length
`ELF`	11,957,838,956	578,616,305	707,194,708
`TIDs`	600,037,903	600,037,903	600,037,903
`Child_Pointers`	-	1,178,616,306	1,307,194,710
`Cutoff_Pointers`	-	1,178,616,306	1,307,194,710
`MonoLists`	-	9,622,027,941	9,622,027,941

Table 4.2: The size of data structure in the Elf and Elf variants constructed by level order linearization for 100GB `Lineitem` tables (readTSV and without Cutoffs)

Therefore, in the construction evaluation of Elf without Cutoffs, when the Elf index is constructed for smaller data files, the vertical linearization approach is better than the level order linearization approach in terms of construction time. However, as the size of the data table grows, the advantage of vertical linearization in construction time decreases, while the performance of the level order linearization approach becomes excellent. Especially for the data table with a size of 100GB, the level order linearization method is about 30% faster than the vertical linearization method in terms of construction time.

Use readTBL

To ensure the accuracy and completeness of the experiment, we also used readTBL to read data files of different sizes to construct Elf and Elf variants. We show the evaluation data in Table 4.3. From Table 4.3, we can intuitively observe that in the experimental environment where readTBL is used to read data files, the construction performance of the level order linearization approach is not as good as that of the vertical linearization approach. In Table 4.3, we can observe that for the evaluation data with normal performance, regardless of the size of the data file, the construction speed of the vertical linearization approach is always about 26.7% (average) faster than the construction speed of the level order linearization approach. This is consistent with our evaluation results using readTSV.

Elf variants	1GB		50GB	
`Elf64`	5.07	32%	583	26%
`Elf_Separated`	5.47	26%	572	27%
`Elf_Separated_Length`	5.50	27%	572	27%
`Elf64_Level`	7.49	-	785	-
`Elf_Level_Separate`	7.37	-	785	-
`Elf_Level_Separated_Length`	7.50	-	786	-

Table 4.3: The construction time (s) of Elf and Elf variants for `Lineitem` tables with different sizes (readTBL and without Cutoffs)

2. With Cutoffs

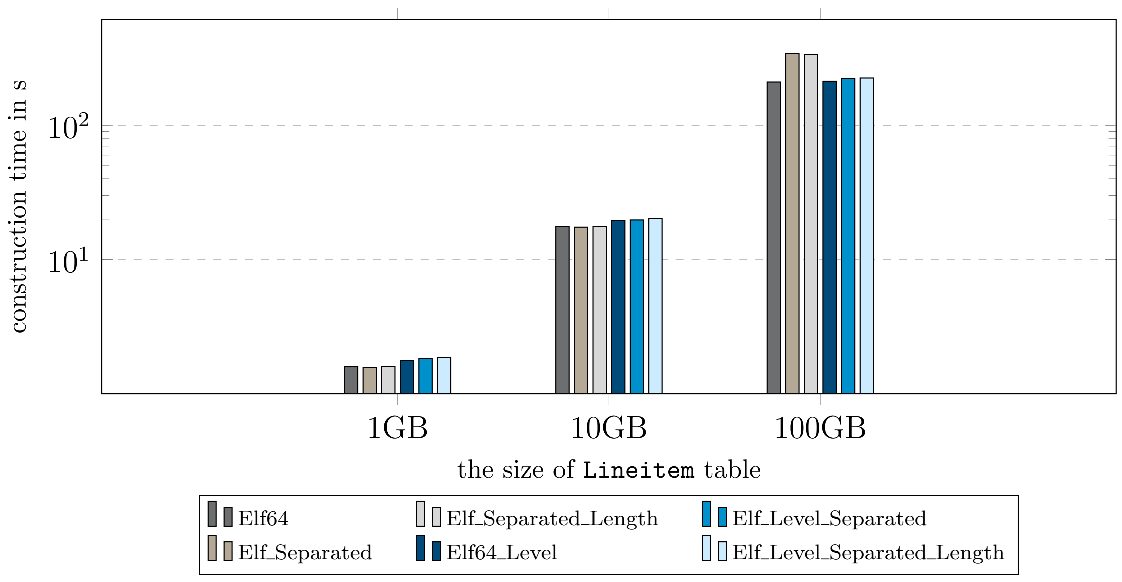


Figure 4.7: Construction time of all Elf variants with cutoffs=16 (readTSV)

We set Cutoffs to 16 to add cutoff-pointers for all dimensions. Then, we conduct construction evaluation. We show the evaluation data in Figure 4.7 and Table 4.4. By comparing with Figure 4.6 and Table 4.1, we can find that the time to construct Elf and Elf variants with Cutoffs has not changed in the overall trend. The larger the data file to be indexed, the better the performance of the level order linearization approach in construction time. When the data file is large enough, such as a 100GB data file, the construction speed of level order linearization approach is about 34.7% faster than that of vertical linearization approach.

Elf variants	1GB		10GB		100GB	
`Elf64`	1.59	10%	17.56	10%	210.16	1.3%
`Elf_Separated`	1.57	14.1%	17.43	11.6%	343.44	-
`Elf_Separated_Length`	1.60	16.4%	17.58	13%	344.68	-
`Elf64_Level`	1.77	-	19.53	-	212.98	-
`Elf_Level_Separate`	1.83	-	19.73	-	223.46	34.9%
`Elf_Level_Separated_Length`	1.86	-	20.21	-	225.16	34.6%

Table 4.4: The construction time [s] of Elf and Elf variants for `Lineitem` tables with different sizes (readTSV and with cutoffs=16)

4.2.1.3 Evaluation Result and Summary

In this chapter, we have mainly introduced the construction evaluation. By evaluating the storage consumption and construction time of the Elf and Elf variants constructed by the level order linearization, we found some advantages of the level order linearization for the Elf approach. First, because the level order linearization approach can save cutoff-pointers sequentially, it can indeed eliminate the gaps in the data structure `Cutoff_Pointers` in the Elf variant. Then, the Elf variant will not reserve space of cutoff-pointers for unselected dimensions. Finally, when we construct Elf variants (Separated) for the data table read by readTSV, if the size of the data table is large enough (100GB), the construction speed of the level order linearization approach is about 30% faster than that of the vertical linearization approach.

4.2.2 Query Evaluation

In this chapter, we introduce query evaluation. Because we only implemented the new partial match query algorithm for level order linearization, we use the PM algorithm to complete the query evaluation work. In addition, we evaluate the mono-column selection predicate query (one-dimensional) and the multi-column selection predicate query (multi-dimensional) respectively. For each type of query, we set three different query statements according to different conditions. All statements are adapted from the queries `Q1`, `Q6`, `Q10` and `Q19` defined in the TPC-H benchmark test. The standard for designing query statements is to cover the features of Elf constructed by level order linearization as much as possible, so as to verify whether the Elf approach can benefit from the level order linearization strategy. For example, we evaluate a query whose selection predicate is not the first dimension, so that we can not only test whether the new PM query algorithm we have implemented can actually jump to the first selected predicate for the query, but also verify the efficiency of this jump by comparing the performance of the vertical linearization approach.

4.2.2.1 Query Statement used for Evaluation

Since the new PM algorithm is implemented based on the features of the level order linearization approach, the design idea of the PM algorithm in the Section 3.3.3 can be used as the condition for designing query statements. We summarize it into the following three conditions:

1. The first dimension as the last selection predicate;
2. The first dimension as the first selection predicate;
3. Non-first dimension as the first selection predicate.

Mono-column Selection Predicate Query

We first introduce how to design specific `SQL` statements for mono-column select predicate queries. Mono-column selection predicate query is also a one-dimensional query. For one-dimensional queries, the three conditions enumerated above can be regarded as two major conditions. Because there is only one selection predicate in a one-dimensional query, condition 1 and condition 2 can be regarded as one condition. Therefore, the first `SQL` statement we set for a one-dimensional query is a query with the first dimension as the selection predicate.

The first dimension in the `Lineitem` table is `L_ORDERKEY`, which is often associated with other tables (e.g. `Order` table) in the standard TPC-H Benchmark schema. However, in order to directly evaluate the characteristics of the level order linearization method, we do not consider associated queries with other tables in this evaluation. Therefore, we can adapt `Q1` to generate the `SQL` statements we need. We show the adapted mono-column selection predicate query statement in Listing 4.1. In the following, we use `SQ1` to represent this query statement. `SQ` is an abbreviation for Single(mono)-column selection predicate Query.

```
SELECT *
  FROM Lineitem
 WHERE l_orderkey >= 1 and l_orderkey <= 1000;
```

Listing 4.1: Mono-column selection predicate query statement `SQ1` adapted from `Q1`

We will use Algorithm 3.8 `firstDimAsLastSelected` to process `SQ1`. Next, we consider the second main condition. When the predicate is not the first dimension, we can exploit the characteristics of the level order linearization approach to directly jump to this predicate for the query. In the new PM algorithm, we divide this situation into two independent scenarios. The first scenario is that the dimension where the predicate is located contains a DimensionList. Because we always call

Algorithm 3.10 `monoListsBeforeSelectedDim` to process the MonoLists that exists before this dimension, the MonoLists has no effect on this scenario. The second scenario is that there are only MonoLists in the dimension where the predicate is located. In this case, we call 3.11 `monoListsInSelectedDim` to traverse all the MonoLists directly.

Because we analyzed the `Lineitem` table with a size of 10GB in the construction evaluation, we know that in this table the fourth dimension is the last dimension that contains the DimensionLists. Therefore, we only need to arbitrarily select a non-first dimension before this dimension as a predicate to satisfy the first scenario. For the other scene, we can choose any dimension after the fourth dimension. For this, we have selected the third dimension `L_SUPPKEY` (`SQ2`) and the eleventh dimension `L_SHIPDATE` (`SQ3, Q1`) to adapt the query statement respectively. In Listing 4.2 we show the mono-column selection predicate query `SQ2` adapted from `Q1` for the first scenario. We directly use `Q1` as the query statement that meets the conditions of the second scenario. In order to facilitate the presentation of the evaluation data, we use `SQ3` to represent `Q1`. In addition, `Q10` also meets the second scenario.

```
SELECT *
  FROM Lineitem
 WHERE l_suppkey >= 100 and l_suppkey <= 500;
```

Listing 4.2: Mono-column selection predicate query statement `SQ2` adapted from `Q1`

Multi-column Selection Predicate Query

Multi-column selection predicate queries are also multi-dimensional queries. It can treat the above three conditions as two main conditions just like a one-dimensional query. The reason is that a multi-dimensional query that satisfies the first condition is also a one-dimensional query. A multi-dimensional query containing the first dimension only meets condition 2. Because the start point of this query type must be the first dimension, we can choose any other dimension and the first dimension as the predicates of the query statement. For this, we chose the first dimension `L_ORDERKEY` and the eleventh dimension `L_SHIPDATE` as the predicates of the new query, and we show an example of this query in Listing 4.3. In the following, we will use `MQ1` to represent this query. `MQ` is an abbreviation for Multi-dimensional selection predicate Query. These new query statements we designed are just to meet our evaluation requirements. Because we only consider the dimensions they represent, they have no practical significance.

```
SELECT *
  FROM Lineitem
 WHERE l_orderkey >= 100 AND l_orderkey <= 500;
   AND l_shipdate >= [DATE]
   AND l_shipdate <  [DATE] + "1Year"
```

Listing 4.3: Multi-column selection predicate query statement `MQ1`

Then, we design the query statement for the non-first dimension as the first selection predicate. It is also divided into two scenarios. The first scenario is that the dimension where the first selection predicate is located contains the DimensionLists. The second scenario is that there are only MonoLists in the dimension where the first selection predicate is located. We adapted `Q6` and `Q19` to meet the requirements of these two scenarios. We use `MQ2` to represent a query statement that uses a non-first dimension as the first selection predicate and this dimension contains the DimensionLists. We use `MQ3` to represent a query statement that uses a non-first dimension as the first selection predicate and there are only MonoLists in this dimension.

Query	Adapted	Predicate columns	Dimension
`SQ1`	`Q1`	`l_orderkey`	$\{0\}$
`SQ2`	`Q10`	`l_suppkey`	$\{2\}$
`SQ3`	`Q10`	`l_shipdate`	$\{10\}$
`MQ1`	`Q6`	`l_orderkey`, `l_shipdate`	$\{0, 10\}$
`MQ2`	`Q19`	`l_linenumber`, `l_shipinstruct`, `l_shipmode`	$\{3, 13, 14\}$
`MQ3`	`Q6`	`l_quantity`, `l_discount`, `l_shipdate`	$\{4, 6, 10\}$

Table 4.5: Details of the TPC-H query used for evaluation

We summarize the new query mentioned above in Table 4.5, which contains the name of the new query, the dimensions involved and the column names. `SQ1`, `SQ2` and `SQ3` are one-dimensional queries. `MQ1`, `MQ2` and `MQ3` are multi-dimensional queries. The function of each query in the evaluation is as follows:

1. `SQ1` and `MQ1` evaluate the query performance of Elf and its variants constructed by the level order linearization approach in the first dimension

2. `SQ2` and `MQ2` evaluate the query performance of Elf and its variants constructed by the level order linearization approach in non-first dimensions. This dimension where the first selected predicate is located must contain DimensionLists to ensure that the PM algorithm can scan the data structure Elf that stores the DimensionLists in the Elf variants (e.g., Separated, Separated_Length)

3. `SQ3` and `MQ3` are extensions of `SQ2` and `MQ2`. They also evaluate the query performance of Elf and its variants constructed by the level order linearization approach in non-first dimensions. However, This dimension where the first selected predicate is located only contain MonoLists, which ensures that the Elf variant only scans the data structure MonoLists that stores MonoList.

Due to the structural limitations of the standard Elf, all data is stored in a data structure ELF. The standard Elf constructed by level order linearization is only different from the standard Elf constructed by vertical linearization in the data storage order. In order to ensure the correctness of the query results, for any query, we must start traversing from the first dimension of the standard Elf. We have explained the reason in Section 3.3 through the analysis of Figure 3.16. Therefore, the above conditions have no effect on the standard Elf constructed by level order linearization. We only need to observe its performance for one-dimensional and multi-dimensional queries.

4.2.2.2 Without Cutoffs

In this chapter, we perform query evaluation on the Elf variant without Cutoffs. We use the six queries introduced in the previous chapter for partial match queries. In addition, we generate 100 query cases for each query, and each case is repeated 10 times. Then, we count the query time to get the average query time. We show the obtained evaluation data in Figure 4.8.

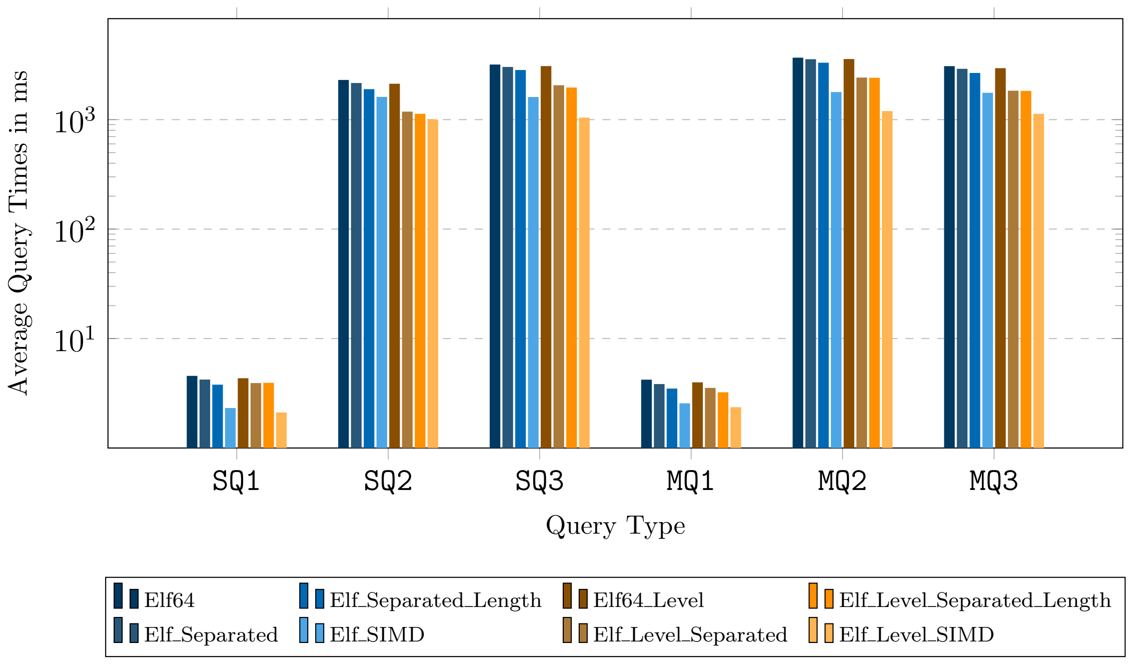


Figure 4.8: Average Query Times of all Elf variants for the `Lineitem` table with a size of 10GB (without Cutoffs)

From Figure 4.8, we can observe that whether it is a one-dimensional query or a multi-dimensional query, the query performance of Elf64_Level is almost the same as that of Elf64. The other Elf variants constructed by the level order linearization approach perform well in queries where the non-first dimension is used as the first selection predicate (`SQ2`, `SQ3`, `MQ2`, `MQ3`). They all perform better than the corresponding vertical linearized Elf variant. We show in Figure 4.9 the percentage of speedup for the same query by the level order linearization Elf relative to the vertical linearization Elf.

From Figure 4.9, we can intuitively observe that the level order linearization approach improves the query performance of Elf. However, this improvement still has certain limitations. For example, for Elf64_Level, we cannot directly implement predicate jumps in its data structure, so it does not benefit from level order linearization. This also leads to its poor performance in all queries (compare to other level order linearized Elf variants). From Figure 4.9, we can observe that the query

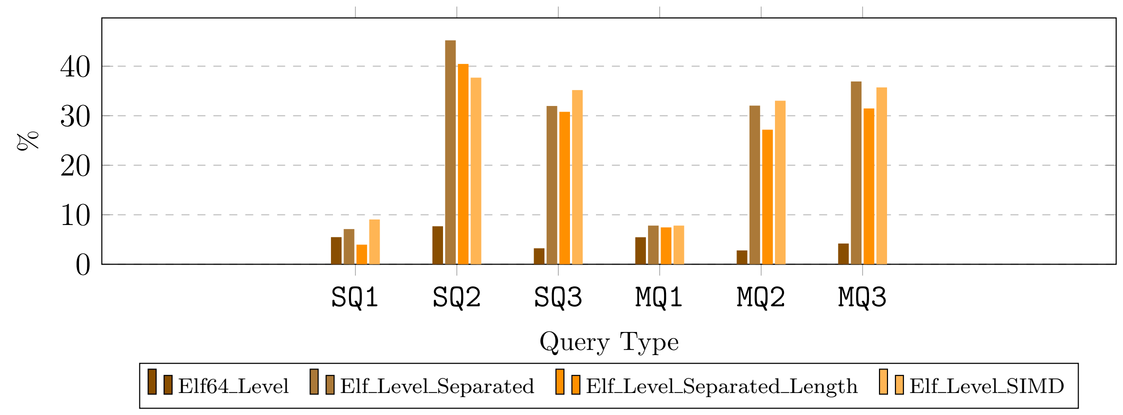


Figure 4.9: The Speedup percentage of all level order linearized Elf variants for the same query based on Figure 4.8 (without Cutoffs)

acceleration percentage of Elf64_Level fluctuates around 5%. This improvement is meaningless.

For other Elf variants, such as Elf_Level_Separated, ELf_Level_Separated_Length and Elf_SIMD, they do not have much speedup (less than 10%) in queries whose the predicate contains the first dimension (`SQ1`, `MQ1`). This means that their query performance in this scenario is similar to the vertical linearization approach. However, since we can directly jump to the first predicates in these Elf variants for queries, these Elf variants perform best in queries that use non-first-dimension as the first selection predicate (`SQ2`, `SQ3`, `MQ2`, `MQ3`). These level-order linearized Elf variants have an average speedup of 34.8% for the same query, 36.8% for one-dimensional queries, and 32.7% for multi-dimensional queries. Next, we perform query evaluation on the Elf variant with Cutoffs to observe if it has a higher improvement.

4.2.2.3 With Cutoffs

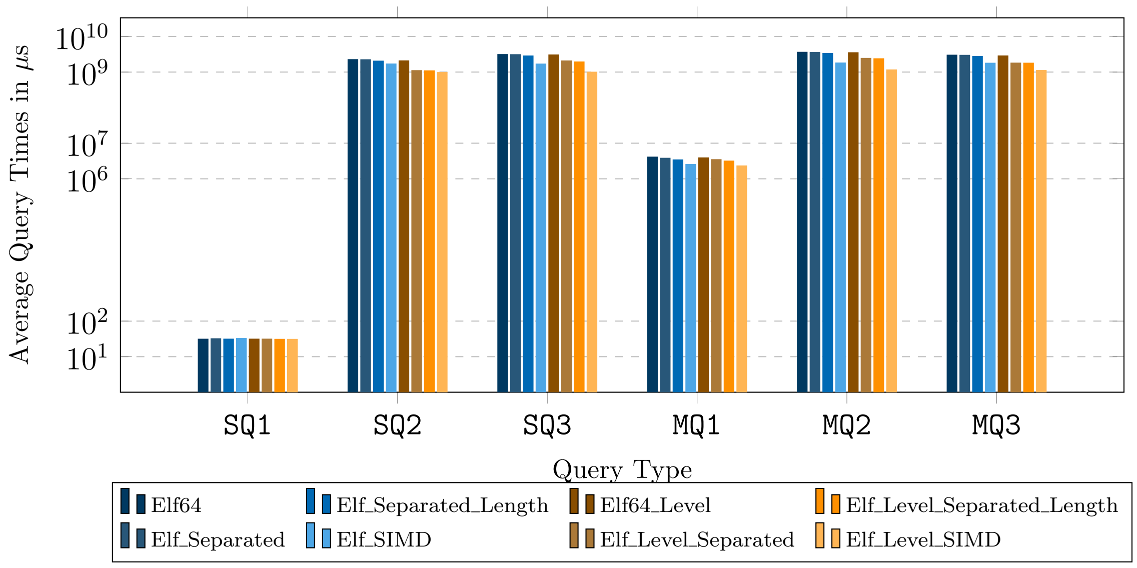


Figure 4.10: Average Query Times of all Elf variants for the `Lineitem` table with a size of 10GB (with cutoffs=16)

In this chapter, we introduce the query evaluation of Elf variant with cutoff. We show the evaluation data in Figure 4.10. From this figure, we can clearly observe that after using the cutoff-pointers, there is a huge performance improvement for one-dimensional queries involving the first dimension (i.e., `SQ1`). To show the query time of `SQ1`, we changed its unit from milliseconds to microseconds. Due to the order of magnitude, we cannot intuitively observe the acceleration effect of level order linearized Elf from the figure. Therefore, we still show its acceleration percentage in a separate table (i.e., Figure 4.11).

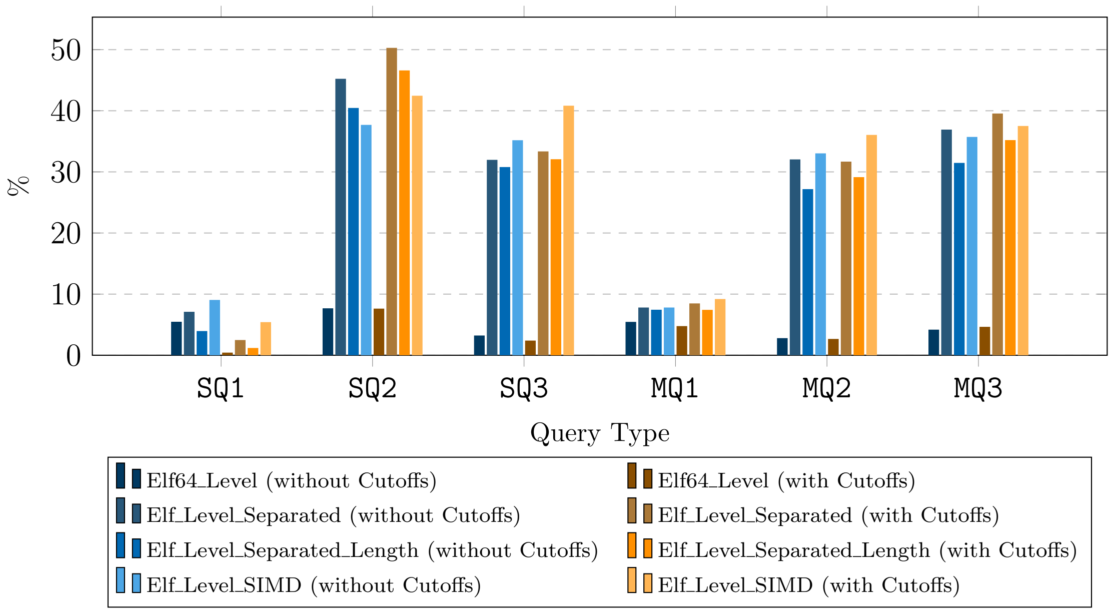


Figure 4.11: The Speedup percentage of all level order linearized Elf variants for the same query based on Figure 4.9 and Figure 4.10 (with cutoffs=16)

For a more intuitive comparison, we have also added the data in Figure 4.10 to Figure 4.11. The blue bars in Figure 412 represent that in case of without Cutoffs the acceleration percentage of the level order linearized Elf variant for different queries. The orange Bar is the acceleration percentage of the Elf variant with Cutoffs for different queries. Through comparison, we can find that the reference of the cutoff-pointers can make the level order linearization Elf variants (e.g., Separated, Separated_Length) further accelerate the query that does not involve the first dimension (`SQ2`, `SQ3`, `MQ2`, `MQ3`). These level-order linearized Elf variants have an average accelerate of 37.92% for the same query, 40.98% for one-dimensional queries (`SQ2`, `SQ3`), and 34.85% for multi-dimensional queries (`MQ2`, `MQ3`). In addition, the performance improvement of the level order linearization approach on the standard Elf (Elf64_Level) and queries involving the first dimension is still limited.

4.3 Summary

In this chapter, we mainly introduce our evaluation work. The purpose of our evaluation is to verify whether the Elf approach can benefit from level order linearization. At the same time, the results of the evaluation can also answer the research questions (RQ 3) we put forward in Chapter 1. First, we introduced the experimental framework. It provides all the basic information about the experiment, such as the experimental environment, experimental data, the object to be evaluated and the type of evaluation. Then, we conducted experiments based on the two evaluation types.

1. Construction Evaluation

We evaluate the construction from two aspects, which are storage consumption and build time. Below is an overview of our evaluation results:

1. Eliminate gaps in `Cutoff_Pointers`

   Through the evaluation of storage consumption, we proved that the level order linearization approach can indeed eliminate the gap problem in the data structure `Cutoff_Pointers`. This problem is caused by the vertical linearized Elf variant when the cutoff-pointers is added. However, the level order linearization approach can only eliminate gaps for Elf variants other than the standard Elf (e.g. Separated, Separated_Length).

2. Elf variants does not reserve storage space for cutoff-pointers

   The evaluation result is only applicable to all Elf variants except the standard Elf, such as Separate, Separate_Length. Due to the characteristics of the level order linearization approach, it adds elements in a level-order manner based on the order of dimensions. Then, when the cutoff-pointers is added for the specified dimension, the cutoff-pointers is also added in a level-order manner. Therefore, the Elf variants (e.g. Separated, Separated_Length) does not have to reserve space for dimensions that are not specified.

3. Accelerate the construction of Elf variants for large data files

   After we analyzed the construction time of Elf variants constructed from different data files, we found that although the vertical linearization approach is faster than the level order linearization approach when constructing ELf variants for small and medium-sized data files. However, as the size of the data file increases, the construction efficiency of vertical linearization decreases compared to level order linearization. When the data file is large enough, the level order linearization approach can construct ELf variants (e.g. Separated, Separated_Length) 30% faster than the vertical linearization approach.

Although we have summarized these advantages, there is also a disadvantage of storage space for the level order linearization approach, that is, the data structure `MonoLists` in the Elf variant (e.g. Separated, Separated_Length) stores the length of each MonoList. The number of these lengths does not exceed the number of TIDs, that is, does not exceed the total number of rows in the data table. However, when the data file is very large, the total number of rows may also reach a higher order of magnitude. At this time, these storage spaces may have some negative effects.

2. Query Evaluation

We use the partial match algorithm for query evaluation. In order to reasonably evaluate the query performance of the level linearization approach, we designed six new TPC-H queries. These queries cover many conditions, such as one-dimensional queries, multi-dimensional queries, the first dimension as a selection predicate, the non-first dimension as a selection predicate, the first selection predicate contains DimensionLists the first selection predicate only contains MonoLists. We perform query evaluation on Elf variants with Cutoffs and without Cutoffs, and finally, we can summarize the evaluation results as follows:

1. Level linearization approach can improve query efficiency

   We analyzed the evaluation data and found that the Elf variant (e.g. Separated, Separated_Length) constructed by level order linearization can increase the query speed by 35% on average for queries that do not involve the first dimension in the predicate.

2. Jumps directly to the first selection predicate for queries

   By analyzing the query statements involved in the previous evaluation result, we found that the first selection predicate of these query statements is not the first dimension. This means that the advantage of the level order linearization approach in query performance comes from the jump.

3. Elf64_Level

   Elf64_Level is the standard Elf constructed by level order linearization construction. However, although we have listed so many advantages of level order linearization, none of them is related to Elf64_Level. This is mainly because of the limitation of the data structure used to store the elements. All data are stored in a data structure, which makes it unable to benefit from the level order linearization approach. Therefore, in the query evaluation, the query performance of Elf64_Level is similar to that of the vertical linearized Elf.

4. Queries involving the first dimension

   In query evaluation, for all queries involving the first dimension, the level order linearization approach has not achieved obvious advantages. This is because the start point of these queries is always the first dimension, and the predicate jump is no longer applicable here. The normal traversal query makes the performance of level order linearization and vertical linearization not much different.

Combining the evaluation results of these two evaluations, we can confirm that the Elf approach can benefit from level linearization. However, because the standard Elf constructed by level order linearization cannot benefit from it, this benefit is somewhat restrictive. For Elf variants such as Separated and Separated_Length, the level linearization approach can bring many positive effects on their performance.

5. Related Work

This chapter aims to provide an overview of related work. Since the level order linearization approach is the core contribution of this thesis, we will introduce some tree-based index structures that involve level-order manner to store data. Specifically, CSS-Tree and CSB$^+$-tree should be considered. The reason why we introduce these trees is that their generated intermediate nodes are stored in level order. This is similar to the way of our level order linearization approach stores data. We can get some inspiration from it. In addition, before we introduce the CSS-Tree and CSB$^+$-tree, we review several common index structures, such as B-tree, B$^+$tree and T-tree.

5.1 Common index structure

In this chapter, we review several common index structures. These index structures are often mentioned in this thesis, but we did not systematically introduce them in the previous chapters. In particular, the B-tree, B$^+$tree are the basis of the CSS tree, and we need to review them before introducing the CSS tree.

5.1.1 B-tree

B-tree is a self-balancing tree that can keep data in order. This data structure enables the actions of searching data, sequential access, inserting data, and deleting all in logarithmic time. B-tree, in summary, is a generalized binary search tree, and a node can have more than 2 child nodes [Com79]. A B-tree of order m is a tree which satisfies the following properties [Knu98]:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $[m/2]$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains k - 1 keys.

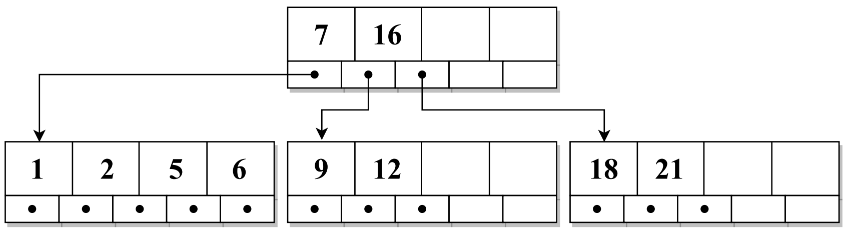


Figure 5.1: A B-tree of order 5 [BM72, Knu98]

5. All leaves appear at the same level and carry no information.

Combining the above properties, we can summarize the B-tree of order 5 in Figure 5.1. The number of keywords in root nodes n satisfies $1 \leq n \leq 4$. The number of keywords in non-root nodes n satisfies $2 \leq n \leq 4$ (e.g., 1, 2, 5, 6 in the left sub-tree), and each node contains up to 5 children (black solid dots). Except for the root node and leaf nodes, other nodes have at least 3 children. Here we only give a brief introduction to these properties, and we will not elaborate on how to search, insert, and delete. As an index structure, B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems [BM72]. Therefore, many database systems use B-tree as the standard index structure. It is a reasonable step to tuning B-tree to obtain better cache performance in the main memory database system [Bro19].

5.1.2 B$^+$-tree

A B$^+$-tree is an n-ary tree. It consists of a root, internal nodes (index node) and leaves [EN10]. In disk database systems and relational database management systems, B+ tree is the most widely used indexing technology [RJ00]. Its purpose is to reduce the number of data disk Input/Output (I/O) operations. This is achieved by passing in the disk page where the index is located and retrieving it to get the exact number of pages where the entry is located, then read and write. The CSS-tree is also based on this idea.

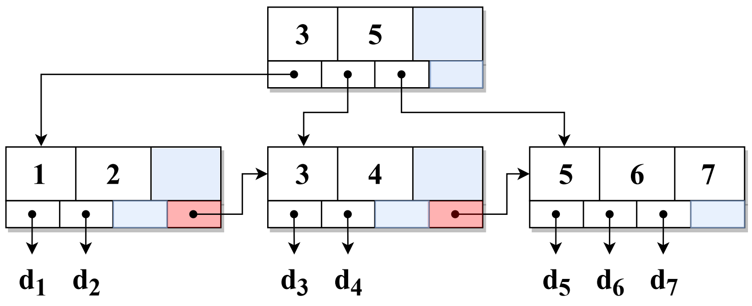


Figure 5.2: A simple B$^+$-tree example

A simple B$^+$-tree example in Figure 5.2 linking the keys 1–7 to data values d_1-d_7. The linked list (red rectangle) allows rapid in-order traversal. This particular tree's branching factor is b = 4 (order 4) [Wik20]. We can observe that the B$^+$-tree is similar to the B-tree. For example, their root node has at least one element, and the range of their non-root node elements is $[m/2] \leq k \leq [m-1]$. Compared with B-trees, B$^+$-tree also have many unique properties. B$^+$-tree have two types of nodes: internal nodes (non-leaf nodes) and leaf nodes. The internal node does not store data but only stores the index, and the data is stored in the leaf nodes. The keys in the internal nodes are sorted in ascending order. For a key in the internal node, all keys in the left tree are less than it, and keys in the right sub-tree are greater than or equal to it. The records in the leaf nodes are also arranged according to the size of the key. Each leaf node stores pointers of adjacent leaf nodes and the leaf nodes themselves are linked in order of the size of the key from small to large.

Because of these properties, the B$^+$-tree has its own advantages. For example, a single node stores more elements, which makes the number of query I/O operations less. In addition, its query performance is stable because all queries must reach the leaf nodes, and all leaf nodes form an ordered linked list, which is more convenient for querying.

5.1.3 T-tree

A T-tree is an indexing technology optimized for main memory access. it is a balanced index tree data structure containing many keywords in one node [CK96]. T-tree also inherits the characteristics of B-tree in update and memory aspect, but in terms of size and algorithm, index items of T-tree are much more streamlined than B-tree [Zhu11].

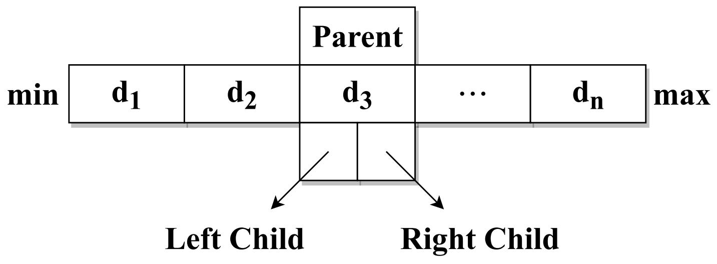


Figure 5.3: T-tree node

Figure 5.3 shows T-node, a node of a T-tree. A T-node usually consists of pointers to the parent node, the left and right child node, an ordered array of data pointers ([min, max]) and some extra control data [LNT00]. Multiple key values can be stored in a storage node. Its leftmost and rightmost key values are the minima and maximum key values of this node respectively (i.e., d_1, d_n). Its left subtree only contains a record whose key value is less than or equal to the minimum key value. Similarly, the right subtree only includes those records whose key value is greater than or equal to the maximum key value. Nodes with left and right subtrees are called internal nodes, nodes with only one subtree are called half-leaf nodes, and nodes without subtrees are called leaves.

T-tree as an index can mainly complete three tasks: search, insert and delete. Insert and delete are based on search. Therefore, to realize the T-tree index is to realize the T-tree search. In addition, the T-tree is maintained by rotating the T-tree. When the T-tree is unbalanced due to the insertion or deletion of key values, the T-tree must be rotated to rebalance it. Although T-trees were once widely used for main memory databases due to their performance advantages, the latest trends in large main memory databases have paid more attention to deployment costs.

5.2 Cache Sensitive Index Structure

The mainstream of today's database is the main memory database, because the data of the database is stored in memory, so the performance of the main memory database does not need to consider the time of disk I/O operations. We are more concerned about the mechanism of CPU access to memory. However, CPU does not directly access the memory, but first searches in the cache. If the required data exists in the cache (cache hit), it directly transfers the data to the CPU. If the required data does not exist in the cache (cache miss), the data needs to be found from the memory and written to the cache, then read to the CPU. The time for the CPU to access the cache is much shorter than the time to access the memory. Therefore, in a scene where the main memory database is very frequently read, the probability of direct cache hit directly affects its performance. This is also the reason for the emergence of cache sensitive technology.

5.2.1 CSS-tree

The full name of CSS-tree is cache sensitive search tree. Cache sensitive, that is, try to consider the data in the cache to ensure that they can be accessed frequently while reducing the number of times the CPU accesses the memory. Although for database tables, there are many techniques to accelerate queries, such as from simple sequential search, binary search to index structure B-tree, B^+-tree and T tree, etc., but not every method considers caching problem. Although the T-tree seems to be cache-sensitive, its utilization of the cache is actually very low [RR98]. Since B-tree is the standard index structure of many database systems, in the beginning, Rao and Ross used special linearization techniques to adjust the B-tree structure [RR98, Bro19]. Thereby a CSS-tree is generated.

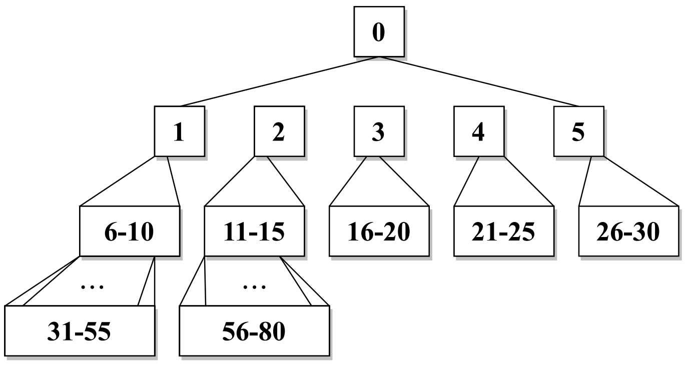


Figure 5.4: Layout of a full CSS-tree (m=4) [RR98]

In general, the idea of implementing the CSS-tree is to generate an index array b based on the existing ordered array a. This array b represents a complete m+1 binary tree (just as a complete binary tree can be stored in a one-dimensional array), and m is the number of index items in each non-leaf node. The m index items naturally generate $m + 1$ gaps (including both sides). A full CSS-tree is shown in Figure 5.4

(the numbers in the boxes are node numbers and each node has four keys) [RR98]. In this figure, nodes 0-15 are called internal nodes, which store index items, and nodes 16-80 are leaf nodes, which store data (that is, data in array a). So we can observe that each node is numbered, and each internal node has m index items, and each leaf node has m data items. After determining m, we can use this to calculate the number of the child node of the node numbered n (if it exists) and the index of the index item owned by the child node in the index array b. Take the above picture as an example, the effective data (array a) that can be stored in such a CSS-tree is (80-16+1)*4 = 260. In addition, the nodes of the CSS-tree can be stored in an array, and the intermediate nodes of the generated CSS-Tree are stored as a continuous array in level-order [RR98, Bro19]. In the end, after sorting the array, the order of the array is arranged in the order of the leaf nodes. However, the CSS-tree performs mediocre in the insertion and deletion of data, but it performs well in the case of more query operations. This is because the CSS-tree is static. Whenever we insert new data, we have to rebuild a CSS-tree. Although the construction time is not long, it is already significantly longer than the query time. In order to eliminate the reconstruction of CSS-Tree, while taking into account the efficiency of add and delete operations, Rao and Ross proposed CSB$^+$-tree [RR00].

5.2.2 CSB$^+$-tree

Although CSS-Tree performs well in query performance, it is static. For this reason, Rao and Ross proposed the CSB$^+$-tree. It is a variant of the B$^+$-tree that continuously stores the child nodes of a given node, and only stores the address of the first child node of the node. The addresses of other child nodes can be obtained by calculating the offset relative to this child node. Since only the pointer of one child node is stored, it has high utilization of the cache [RR00]. This can not only improve the cache sensitivity but also perform incremental updates like the B$^+$-tree [Bro19].

There are two variants of CSB$^+$-tree, segmented CSB$^+$-tree and full CSB$^+$-tree. Segmented CSB$^+$-tree divides the child nodes into segments, and the child nodes in the same segment are continuous Storage, in each node, only the starting address of each segment will be stored. When there is a split, the segmented CSB$^+$-tree can reduce the replication overhead, because only one segment needs to be moved, and the full CSB$^+$-tree redistributes space for the entire node, thus reducing the splitting overhead.

6. Conclusion

In this chapter, we summarize the work and contributions of the entire thesis. Then, we propose possible topics for future work based on the deficiencies mentioned in the conceptual model and algorithm design of the level order linearization approach, as well as the results of the evaluation.

6.1 Summary

In this section, we summarize the work of this thesis and answer the three research questions raised in the first chapter. In this thesis, we first give a brief introduction to the basic background. Besides, we also put forward three research questions based on the contribution of this thesis. Then, we mainly introduce all the background knowledge involved in this thesis. We explained the index structure and query, which helps us understand the basis of our task. Then we explained the content related to Elf. From its conceptual design, linearization method to query algorithm, we have carried out a detailed analysis. In addition, we introduce the Elf variant that will be used in our implementation. We also introduced the implementation of the level order linearization approach. First, we introduced the conceptual model and the conceptual design of the level order linearization approach in order to understand what the level order linearization does. Then we give a detailed introduction to our algorithm from design to implementation. Finally, we also introduced our partial match query algorithm for level order linearization. we evaluate the Elf variant constructed by the level order linearization approach. After implementation, we evaluate the performance of level order linearization through two aspects. One is construction evaluation and the other is query evaluation. We conducted a detailed analysis of the evaluation data in the corresponding evaluation. Finally, we list the evaluation results in the summary based on our analysis. After evaluation, we introduce the related works involved in the paper. We mainly introduced the cache-sensitive index structure. To be able to understand the relevant content, we first gave a brief introduction to the common index structure. Finally, we summarize the content of the paper and propose some future work.

Next, we answer the three research questions raised in Chapter 1 based on the content of this thesis.

RQ 1: How to implement level order linearization for the Elf Approach?

We divide the process of implementing level order linearization for the Elf approach into three stages, which are the analysis stage, the design stage, and the realization stage. In the analysis stage, we first studied the content related to the multi-dimensional index structure, and on this basis, we analyzed the conceptual design of Elf. This extends to the linearization of Elf. In the design stage, we first analyze the structure of level order linearization, which is the conceptual model. Then, we carry out a conceptual design based on this model. By analyzing a sample data table, we list our design step by step. In the realization phase, we complete the realization of level order linearization based on these design steps.

RQ 2: What is the difference between the Partial Match algorithm adapted to level order linearization and the standard Partial Match algorithm?

We also implemented a new PM algorithm in the realization stage of. In this process, we analyzed their differences. They are all designed based on the characteristics of their linearization. So the biggest difference is the idea of the algorithm. The standard partial matching algorithm follows the idea of the DFS algorithm, and the horizontal linearized matching algorithm uses the idea of BFS. The order in which they traverse the data is vertical and level-order manner respectively.

RQ 3: How will the level order linearization strategies affect Elf's performance?

This research question is also the second goal of our thesis work, which is to verify whether the Elf method can benefit from level order linearization. In order to answer this question, we evaluated the level order linearization approach we achieved. Through evaluation, we found that the level order linearized Elf variant (e.g., Separated, Separated_Length) saves the cutoffs pointer space, improves the construction speed of massive data files and query performance.

6.2 Future Work

In this chapter, we introduce possible topics for future work. They are the inspiration we got at different stages of completing the thesis. At the same time, they are also several topics that we have selected for further research after combining the results of the evaluation. In addition, we propose some optimization work for the existing level order linearization approach. Because these optimizations need to adjust the overall structure of the existing code, we put them in the future work based on the time of our task and personal factors.

6.2.1 Adjust the data structure of the standard Elf

Based on our evaluation results, we found that the standard Elf constructed by level order linearization only changed the data storage order. However, because it is limited by the data structure of the standard Elf, it cannot benefit from the level order linearization approach. Therefore, under the premise that all data is still stored in one data structure, how to adjust this data structure so that the standard Elf can benefit from the level linearization will be an interesting topic.

6.2.2 Queue as a New Implementation Method

The vertical linearization approach uses the standard DFS algorithm, which reasonably uses recursion. For the level order linearization approach, the standard idea of the BFS algorithm is to use queues to achieve a breadth-first search. However, we only did a quick screening in the initial design phase. The queue is restrictive, for example, it can only be stored and retrieved sequentially and cannot be accessed arbitrarily. Therefore, we did not choose to use the queue as the data structure for storing data in the linearization approach.

We analyzed the conceptual model of the level order linearization method and decided to use a general vector to store data about the number of rows. This avoids the use of queues. However, our theoretical inferences indicate that the construction time using this model will be slightly longer than the construction time of the vertical linearization method. Our evaluation data on the construction for small and medium-sized data files also confirmed this point. This does not conform to the theory that DFS and BFS have the same time complexity. Although there might be other factors that affect the construction time, we still attribute the main factor to the insufficient structure of the algorithm. Therefore, the queue as the main implementation method for the level order linearization approach is worth considering. In addition, if we choose a queue to achieve level order linearization, we must consider a new model, because the existing level order linearization approach cannot provide a valuable reference for it.

6.2.3 Further Optimization

In this chapter, we propose some optimization work. These works are based on the evaluation results and the theoretical part of our Thesis. When we carry out related work in the future, we can consider starting from these optimization work. Although these optimization works are based on theoretical speculation, they should still receive sufficient attention in future work.

6.2.3.1 First Visited MonoList

We discussed and analyzed the three methods of processing MonoLists in Section 3.3.2. Then, we chose the same way as Elf store DimensionLists, that is, save the size of a MonoList in the front of each MonoList. However, we found in the evaluation results that this method does not have an advantage in storage space. At the same time, if there are more MonoLists in the construction process, the time for us to calculate the size of the MonoLists will increase, and the greater the impact on the construction time. Among the remaining two methods, we focus on the second method, which is First visited MonoList in Section 3.3.2.2. Although this method lacks flexibility when we deal with MonoLists, it has a relatively large advantage in terms of storage space. This is because this method only needs to add one more vector. We use this vector to save the start position of a MonoList in each dimension. Therefore, the size of this vector will not exceed the number of dimensions. However, the most important part of achieving this optimization is to use a reasonable method to obtain the necessary data.

6.2.3.2 Optimization for Partial Match Queries

Although we have implemented the function for the most important feature of the level order linearization approach in the new PM algorithm, which jumps directly to the first predicate for the partial match query, we still use depth-first match when dealing with specific dimension values. Therefore, in future optimization work, we can consider using the breadth-first feature when we use partial match query algorithm processes DimensionLists.

6.2.4 Other Query Algorithms

In our contribution, we only implemented the PM algorithm based on the characteristics of the level order linearization approach. In our contribution, we only implemented the PM algorithm based on the characteristics of the level order linearization approach. However, for the normal Elf method, it has many different query types, such as column-column comparison query, parallel query. Therefore, in future work, it is an important theme to design and implement algorithms for multiple query types based on the characteristics of level order linearization.

Bibliography

[BG10] Matthias Beck and Ross Geoghegan. *The art of proof: basic training for deeper mathematics*. Springer Science & Business Media, 2010. (cited on Page 33)

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating multi-column selection predicates in main-memory-the elf approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017. (cited on Page xi, xv, 2, 10, 11, 16, 17, 19, 21, 22, 23, 24, and 25)

[BM72] R Beyer and EM McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972. (cited on Page xiii and 92)

[Bón02] Miklós Bóna. *A walk through combinatorics: an introduction to enumeration and graph theory*. World Scientific, 2002. ISBN-13: 978-9813237452. (cited on Page 33)

[Bro19] David Broneske. *Accelerating Mono and Multi-Column Selection Predicates in Modern Main-Memory Database Systems*. PhD thesis, University of Magdeburg, Germany, 2019. (cited on Page xi, 1, 2, 23, 24, 31, 92, 94, and 95)

[Bur01] Donald K Burleson. *Oracle high-performance SQL tuning, 1. Edition.* McGraw-Hill Education, Inc., 2001. ISBN-13 : 978-0072190588. (cited on Page 6)

[Cha01] Lin Chao. Which is faster table scan or index access. *Journal of Chinese Computer Systems*, 22(9):31, 2001. ISSN: 1000-1220. (cited on Page xv, 6, and 7)

[CK96] Kong-Rim Choi and Kyung-Chang Kim. T*-tree: a main memory database index structure for real time applications. In *Proceedings of 3rd International Workshop on Real-Time Computing Systems and Applications*, pages 81–88. IEEE, 1996. (cited on Page 93)

[CLRS01a] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms second edition. *The Knuth-Morris-Pratt Algorithm, year*, 2001. (cited on Page 34 and 36)

[CLRS01b] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Section 22.2: Breadth-first search. *Introduction to Algorithms*, pages 531–539, 2001. (cited on Page 35)

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009. (cited on Page 33 and 35)

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. (cited on Page 91)

[Dro12] Adam Drozdek. *Data Structures and algorithms in C++, Fourth Edition*. Cengage Learning, 2012. ISBN-13 : 978-1133608424. (cited on Page 35)

[DYZ+15] Dinesh Das, Jiaqi Yan, Mohamed Zait, SR Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. Query optimization in oracle 12c database in-memory. *Proceedings of the VLDB Endowment*, 8(12):1770–1781, 2015. (cited on Page xi, 1, 6, 7, 8, 11, and 19)

[EN10] R. Elmasri and S. Navathe. *Fundamentals of Database Systems, 6th Edition*. Pearson Education, 2010. ISBN-13: 978-0-136-08620-8. (cited on Page 92)

[Epp10] Susanna S Epp. *Discrete mathematics with applications, 5th Edition*. Cengage Learning, Inc, 2010. ISBN-13 : 978-0357114087. (cited on Page 33)

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012. (cited on Page 1)

[GM08] Hector Garcia-Molina. *Database systems: the complete book, Second Edition*. Pearson Education India, 2008. ISBN-13 : 978-0131873254. (cited on Page 8)

[GT06] Michael T Goodrich and Roberto Tamassia. *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006. (cited on Page 34)

[HR13] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, 2013. (cited on Page 12)

[Jav17] Paul Javin. Difference between table scan, index scan, and index seek in sql server database, November 2017. https://www.java67.com/2017/12/difference-between-table-scan-index.html Accessed Juni 26, 2020. (cited on Page 9)

[KBSS15] Veit Köppen, David Broneske, Gunter Saake, and Martin Schäler. Elf: A main-memory structure for efficient multi-dimensional range and partial match queries. *Otto-von-Guericke-University Magdeburg, Tech. Rep.*, pages 002–2015, 2015. (cited on Page 2)

[Knu97] Donald Ervin Knuth. *The art of computer programming, Subsequent Edition*, volume 3. Pearson Education, 1997. ISBN-13 : 978-0201896831. (cited on Page 35)

[Knu98] Donald E Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998. (cited on Page xiii, 91, and 92)

[KSS14] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien*. mitp Verlags GmbH & Co. KG, 2014. ISBN-13 : 978-3826694851. (cited on Page 16)

[KT07] Robert Kruse and CL Tondo. *Data structures and program design in C, 1st Edition*. Pearson Education India, 2007. ISBN-13 : 978-8177584233. (cited on Page 33)

[LC85] Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1985. (cited on Page 5)

[LNT00] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. In *Proceedings 11th Australasian Database Conference. ADC 2000 (Cat. No. PR00528)*, pages 65–73. IEEE, 2000. (cited on Page 93)

[LP13] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013. (cited on Page 11)

[LY10] Yong Liu and Xinquan Yang. Data retrieval performance analysis of multidimensional indexes is stored in b+-tree and kd tree structures, chinese edition. *Science & Technology Information*, 2010(19):86–87, 2010. ISSN:1001-9960. (cited on Page 14)

[Pow06] Gavin Powell. *Beginning database design, 1st Edition*. John Wiley & Sons, 2006. ISBN-13 : 978-0764574900. (cited on Page 8)

[RJ00] Ramakrishnan Raghu and Gehrke Johannes. Database management systems, 2nd edition, 2000. ISBN-13 : 978-0072322064. (cited on Page 92)

[RR98] Jun Rao and Kenneth A Ross. Cache conscious indexing for decision-support in main memory, 1998. (cited on Page xiii, 94, and 95)

[RR00] Jun Rao and Kenneth A Ross. Making b+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486, 2000. (cited on Page 3 and 95)

[SDRK02] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 464–475, 2002. (cited on Page xv, 16, 17, and 19)

[SK13] Lefteris Sidirourgos and Martin Kersten. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904, 2013. (cited on Page 11)

[SSH18] Gunther Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Konzepte und Sprachen*. MITP-Verlags GmbH & Co. KG, 2018. (cited on Page 5 and 8)

[VAQLMJ20] Leonardo V Arias, Christian Q López, Alexandra Martínez, and Marcelo Jenkins. Assessing two graph-based algorithms in a modelbased testing platform for java applications. In *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2020. (cited on Page 32)

[Wik20] Wikipedia contributors. B+ tree. https://en.wikipedia.org/w/index.php?title=B%2B_tree&oldid=973377564, 2020. [Online; accessed 5-November-2020]. (cited on Page 92)

[Wol19] Kai Wolf. Datenparallele Selektionen auf der multidimensionalen Indexstruktur Elf. Masterarbeit, University of Magdeburg, Germany, December 2019. (cited on Page 1 and 6)

[Xu05] Yujin Xu. Optimize sql by analyzing the execution plan of sql statements, chinese edition, November 2005. http://www.itpub.net/forum.php?mod=viewthread&tid=478999, [Online; accessed 6-June-2020]. (cited on Page 6)

[YW02] Weimin Yan and Weimin Wu. *Data structure (C language), Chinese Edition*. Tsinghua University Press, Beijing, 2002. ISBN: 978-7302023685. (cited on Page 36)

[Zhu11] Zhi Lin Zhu. Optimization of t-tree index of main memory database in critical application. In *Applied Mechanics and Materials*, volume 40, pages 206–211. Trans Tech Publ, 2011. (cited on Page 93)

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156, 2002. (cited on Page 29)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 20. November 2020