Accelerating Mono and Multi-Column Selection Predicates in Modern Main-Memory Database Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. David Broneske

geb. am 17.11.1989 in Burg

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Prof. Dr. Stefan Manegold

Eingereicht am:
Magdeburg, den 15.03.2019

Verteidigt am:
Magdeburg, den 16.05.2019

Broneske, David:
Accelerating Mono and Multi-Column Selection Predicates in Modern Main-Memory Database Systems
Dissertation, University of Magdeburg, 2019.

Abstract

Ever-since, database system engineers are striving for peak performance of their database operators. However, this goal is a major endeavor since database operators are influenced not only by the *hardware* (i.e., the executing processor or the memory hierarchy), but also by the *workload* (i.e., data distribution, selectivity, etc.). Especially in today's world of main-memory database systems, there are new processing capabilities (e.g., advanced vector instructions such as AVX-512), new storage devices (e.g., Intel Optane as non-volatile RAM), or new diverse applications for data management (e.g., in-database machine learning) frequently introduced that become essential impact factors. Hence, a once optimal operator has to be frequently adapted with these new arising hardware and workloads.

A typical database operator that is frequently tuned by researchers is the selection operator, because selections are essential to reduce the load of subsequent operators and are usually one of the first operators that are executed in a query plan. Hence, a selection is working on the full amount of data – a fact that emphasizes the importance of tuning this data-intensive operator to avoid a serious bottleneck.

Although the selection operator is frequently tuned for arbitrary use cases mentioned above, there is no comprehensive and holistic way to tune this operator automatically. Furthermore, considering multiple selections on the same table, straight-forward implementations use candidate scans for several selection predicates. However, exploiting the interdependence and, hence, high selectivity is not investigated so far. In this thesis, we tackle the aforementioned challenges of (1) creating hardware-sensitive operator implementations automatically and (2) exploiting the relation between multiple selection predicates.

For solving the first challenge, we investigate the commonalities of different optimizations for arbitrary hardware and workloads on the example of the selection operator. As a result, we introduce the abstraction of *code optimizations* as a means to generate hardware-sensitive code variants automatically. The solution is completed by the concept of a tuning framework for operators in main-memory database systems.

As a solution for the second challenge, we propose to revive multi-dimensional index structures as a means to exploit the relation between selection predicates on several columns in main-memory database management systems. In order to allow for hardware-sensitivity – especially cache consciousness – we propose our main-memory index structure Elf. Elf is a tree structure combining prefix-redundancy elimination with an optimized memory layout explicitly designed for efficient main-memory access. Our experiments show that Elf is able to outperform several highly-potent

baselines (including generated hardware-sensitive scans and state-of-the-art multi-dimensional index structures) by several orders of magnitude for reasonable selection predicates and queries from the standard OLAP benchmark TPC-H. However, our evaluation also identifies that an integration into the query engine of the main-memory database system MonetDB does not only show strengths but also limitations that any sort-based index structure is faced with.

Overall, the resulting approaches can be used in future query engines to form a Swiss army knife for arbitrary selection predicates. Hence, our contribution enriches a query engine far beyond current state-of-the-art-approaches by allowing for efficient execution of a single predicate (i.e., *mono-column selection predicates*) at *bare-metal speed* as well as exploiting the combined selectivity of several predicates (i.e., *multi-column selection predicates*).

Inhaltsangabe

Seit jeher streben Datenbankentwickler nach Höchstleistungen ihrer Datenbankoperatoren. Dieses Ziel ist jedoch ein umfängliches Unterfangen, da Datenbankoperatoren nicht nur von der Hardware (d.h. dem ausführenden Prozessor oder der Speicherhierarchie), sondern auch von den zu verarbeitenden Daten (d.h. Datenverteilung, Selektivität, etc.) beeinflusst werden. Mit dem Fortschreiten der Technik – vor allem in Hauptspeicherdatenbanken – gibt es ständig neue Verarbeitungsmöglichkeiten (z.B. erweiterte Vektorbefehle wie AVX-512), neue Speichergeräte (z.B. Intel Optane als nicht-flüchtiger RAM) oder neue vielfältige Anwendungen für das Datenmanagement (z.B. datenbankintegriertes maschinelles Lernen), die zu wesentlichen Einflussfaktoren werden. Daher muss ein einst optimaler Operator häufig an diese neu entstehende Hardware und Anwendungsszenarien angepasst werden.

Ein typischer Datenbankoperator, der von Forschern häufig getunt wird, ist der Selektionsoperator, denn Selektionen sind unerlässlich, um die Last nachfolgender Operatoren zu reduzieren. Des weiteren sind Selektionen in der Regel einer der ersten Operatoren, die in einem Abfrageplan ausgeführt werden. Daher arbeitet der Selektionsoperator auf der gesamten Datenmenge - eine Tatsache, die die Bedeutung des Tunings dieses datenintensiven Operator unterstreicht, um einen schwerwiegenden Engpass zu vermeiden. Obwohl der Selektionsoperator häufig auf eine Vielzahl der oben genannten Einflüsse abgestimmt ist, gibt es keine umfassende und ganzheitliche Möglichkeit, diesen Operator automatisch zu optimieren. Darüber hinaus verwenden naive Implementierungen bei mehreren Selektionen auf derselben Tabelle einen Kandidaten-Scan für mehrere Selektionsprädikate. Die Ausnutzung der Interdependenz und damit der hohen Selektivität wurde jedoch noch nicht untersucht. In dieser Arbeit beschäftigen wir uns mit den oben genannten Herausforderungen: (1) der automatischen Erstellung von hardwaresensitiven Operatorimplementierungen und (2) der Ausnutzung der Beziehung zwischen mehreren Selektionsprädikaten.

Zur Lösung der ersten Herausforderung untersuchen wir die Gemeinsamkeiten verschiedener Optimierungen für beliebige Hardware und Datencharacteristika am Beispiel des Selektionsoperators. Als Ergebnis führen wir die Abstraktion von Code-Optimierungen ein, um hardwaresensitive Codevarianten automatisch zu generieren. Abgerundet wird die Lösung durch das Konzept eines Tuning-Frameworks für beliebige Operatoren in Hauptspeicherdatenbanksystemen.

Als Lösung für die zweite Herausforderung schlagen wir vor, multidimensionale Indexstrukturen wiederzubeleben, um die Beziehung zwischen Selektionsprädikaten auf mehreren Spalten auszunutzen. Um die Hardware-Sensitivität – insbesondere die Optimierung auf CPU-Caches – zu berücksichtigen, schlagen wir unsere

Hauptspeicher-Indexstruktur Elf vor. Elf ist eine Baumstruktur, die die Eliminierung von Präfixredundanzen mit einem optimierten Speicherlayout kombiniert und dessen spezielle Speicherorganisation für einen effizienten Zugriff im Hauptspeicher optimiert ist. Unsere Experimente zeigen, dass Elf in der Lage ist, mehrere hochpotente Kontrahenten (einschließlich generierter hardwaresensitiver Scans und moderner multidimensionaler Indexstrukturen) um mehrere Magnituden für eine sinnvolle Auswahl von Abfragen aus dem Standard OLAP-Benchmark TPC-H zu übertreffen. Unsere Auswertung zeigt aber auch, dass eine Integration in die Query-Engine des Hauptspeicher-Datenbanksystems MonetDB nicht nur Stärken, sondern auch Grenzen aufweist, mit denen jede sortierbasierte Indexstruktur konfrontiert ist.

Insgesamt können die resultierenden Ansätze in zukünftigen Anfrageverarbeitungs-Engine genutzt werden, um eine eierlegende Wollmilchsau für beliebige Selektionsprädikate zu bilden. Daher bereichert unser Beitrag eine Anfrageverarbeitungs-Engine weit über den aktuellen Stand der Technik hinaus, indem er einerseits die Ausführung eines einzelnen Prädikats (d.h. einspaltiger Selektionsprädikate) an das obere Ende der technisch möglichen Performanz der CPU bringt und andererseits die kombinierte Selektivität mehrerer Prädikate (d.h. mehrspaltiger Selektionsprädikate) effizient ausnutzen kann.

Contents

Contents vii

List of Figures xi

List of Tables xv

List of Code Listings xvii

List of Algorithms xix

1 Introduction **1**
- 1.1 Goal of this Thesis . . . 2
- 1.2 Structure of the Thesis . . . 4

2 Selections in the Rear-View Mirror **7**
- 2.1 State of the Art in Selection Predicates . . . 7
  - 2.1.1 Relational Selection Basics . . . 8
  - 2.1.2 Predicate Characteristics . . . 9
  - 2.1.3 Selection Result Representation . . . 10
- 2.2 CPU Capabilities . . . 10
  - 2.2.1 Pipelining in CPUs – the RISC Pipeline . . . 11
  - 2.2.2 Hazards . . . 12
    - 2.2.2.1 Data Hazard . . . 12
    - 2.2.2.2 Control Hazard . . . 12
  - 2.2.3 Single Instruction Multiple Data . . . 13
- 2.3 Heterogenous Programming . . . 13
  - 2.3.1 Hardware-Sensitive Programming . . . 14
  - 2.3.2 Hardware-Oblivious Programming . . . 15
- 2.4 Summary . . . 16

3 Hardware Sensitive Full-Table Scans as Working Horse **17**
- 3.1 Code Optimizations for Hardware-Sensitive Full-Table Scans on Single Predicates . . . 17
  - 3.1.1 Running Example . . . 18
  - 3.1.2 Software Predication . . . 19
  - 3.1.3 Loop Unrolling . . . 20
  - 3.1.4 Single Instruction Multiple Data . . . 23
  - 3.1.5 Code Optimizations in Other Operators and Domains . . . 24

3.2 Multi-Predicate Code Optimizations . . . 25
3.2.1 Conditional AND . . . 25
3.2.2 Bitwise AND . . . 26
3.3 Exploiting Code Optimizations in Database Management Systems . . 27
3.3.1 Variant Generation . . . 29
3.3.2 Variant Selector & Feedback Loop . . . 29
3.3.3 Variant Management . . . 30
3.3.4 Usage of Adaptive Reprogramming in Recent Database Management Systems . . . 30
3.4 Summary . . . 31

4 Elf as Multi-Column Selection Predicate Index 33
4.1 Conceptual Design of Elf . . . 35
4.2 Improving Elf's Memory Layout . . . 36
4.2.1 Mapping `DimensionLists` to Arrays . . . 36
4.2.2 Implicit Length Control of Arrays . . . 37
4.2.3 Alternative Memory Layouts . . . 37
4.3 Storage Optimizations for Elf . . . 38
4.3.1 Hash Map to Deal With the First `DimensionList` . . . 38
4.3.2 `MonoList`: One-Element List Elimination . . . 39
4.3.3 Worst Case Storage Consumption . . . 40
4.4 Searching in Elfs . . . 41
4.4.1 Search Algorithm . . . 41
4.4.2 Selection of the Column Order . . . 44
4.5 Empirical Evaluation . . . 45
4.5.1 Experiment 1: `MonoList` Storage Consumption . . . 46
4.5.2 Experiment 2: TPC-H Predicates and Data . . . 46
4.5.2.1 Mono-Column Selection Predicate Queries . . . 49
4.5.2.2 Multi-Column Selection Predicate Queries . . . 51
4.5.3 Experiment 3: Selection Time Scaling . . . 53
4.5.4 Experiment 4: TPC-H Predicates in MonetDB . . . 54
4.5.4.1 Mono-Column Selection Predicates . . . 55
4.5.4.2 Multi-Column Selection Predicates . . . 56
4.5.5 Result Summary . . . 56
4.6 Summary . . . 58

5 Complex Selection Queries in Elf-Supported Main-Memory Database Systems 59
5.1 Complex Selection Predicates . . . 60
5.1.1 Column-Column Comparisons . . . 61
5.1.2 IN-Predicates . . . 64
5.1.3 Summary . . . 68
5.2 MonetDB Integration . . . 70
5.2.1 MAL Extensions . . . 71
5.2.2 Operator Interoperability . . . 72
5.3 Evaluation . . . 73
5.3.1 Microbenchmarks for Complex Predicates . . . 75

5.3.1.1 Experiment 1: Column-Column Comparison Microbenchmark . . . 75
5.3.1.2 Experiment 2: IN-Predicates Microbenchmark 77
5.3.1.3 Microbenchmark Summary 79
5.3.2 Experiment 3: Elf's Integration Test in MonetDB 80
5.3.2.1 TPC-H Query Runtimes 81
5.3.2.2 Result Summary 83
5.4 Summary 84

6 Elf Life Cycle 87
6.1 Initial Build: Elf Bulk Load 89
6.2 Maintaining an Elf 89
6.2.1 Insertions 90
6.2.2 Deletion 92
6.2.3 Updates 93
6.3 Evaluation 93
6.3.1 Experiment 1: Build Times 94
6.3.2 Experiment 2: Query Overhead of InsertElf 94
6.3.3 Experiment 3: Merge Threshold 95
6.4 Summary 96

7 Related Work 99
7.1 Competitors 99
7.1.1 BitWeaving 99
7.1.2 Column Imprint 101
7.1.3 Sorted Projection 102
7.1.4 BB-Tree 103
7.2 Data Redundancy Elimination 104
7.2.1 Prefix and Suffix-Redundancy Elimination 105
7.2.2 The Data Dwarf Structure 105
7.3 SIMD-Accelerated Main-Memory Indexing 107
7.3.1 Seg-Tree and Seg-Trie 107
7.3.2 Fast Architecture Sensitive Tree 108
7.3.3 Vector-Advanced and Compressed Structure Tree 110
7.3.4 Adaptive Radix Tree 110
7.3.5 Comparison to Elf 112
7.4 One-Dimensional Main-Memory Indexing 114
7.5 Multi-Dimensional Main-Memory Indexing 115

8 Conclusion 117

9 Future Work 121

Bibliography 125

List of Figures

2.1 RISC pipeline 11

2.2 Example of a data hazard 12

2.3 Example of a control dependency (`instr. i+1` was the wrong decision) 12

2.4 Single Instruction Single Data (SISD) vs. Single Instruction Multiple Data (SIMD) 13

2.5 Hardware-sensitive vs. hardware-oblivious programming 14

3.1 Response time of a branching scan on 30 million data items 19

3.2 Response time of branching and predicated scans on 30 million data items 20

3.3 Response time of unrolled branching scans for varying selectivities for 30 million data items (LUn = n-times loop-unrolled) 21

3.4 Response time of unrolled predicated scans for varying selectivities for 30 million data items (LUn = n-times loop-unrolled) 22

3.5 Response time of scan variants on 30 million data items 24

3.6 Response time of conditional `AND` scan for two predicates under varying selectivities 26

3.7 Response time of bitwise `AND` scan for two predicates under varying selectivities 27

3.8 Response time of bitwise `AND` scan under different numbers of predicates 28

3.9 Sketch of our adaptive reprogramming approach for reaching hardware-sensitive database operations on heterogeneous hardware 29

4.1 (a) `WHERE`-clause, (b) selectivity, and (c) response time of Elf and a scan generated by adaptive reprogramming (ARScan) on TPC-H query `Q6` and its predicates `Q6.1 - Q6.3` on `Lineitem` table $s = 100$. 34

4.2 Elf tree structure using prefix-redundancy elimination 35

4.3 Memory layout as an array of 64-bit integers 36

4.4 Hash-map property of the first `DimensionList` 39

4.5 Percentage of 1-element lists per dimension for the TPC-H `Lineitem` table with scale factor 100 . . . 40

4.6 `MonoList` (visualized as gray `DimensionLists`) for optimized cache performance and storage utilization . . . 40

4.7 Optimized memory layout . . . 40

4.8 Query response times of Elf and accelerated full-table scans for mono-column TPC-H queries ($s = 100$) . . . 49

4.9 Query response times of Elf and multi-dimensional index structures for mono-column TPC-H queries ($s = 100$) . . . 50

4.10 Query response times of Elf and accelerated full-table scans for multi-column TPC-H queries ($s = 100$) . . . 51

4.11 Query response times of Elf and multi-dimensional index structures for multi-column TPC-H queries ($s = 100$) . . . 52

4.12 Selection time scaling ratios for all approaches . . . 53

4.13 Query response times for mono-column TPC-H queries ($s = 100$) in MonetDB . . . 56

4.14 Query response times for multi-column TPC-H queries ($s = 100$) in MonetDB . . . 57

5.1 Prefix redundancy elimination in Elf for efficient evaluation of column-column comparisons . . . 61

5.2 Prefix redundancy elimination in Elf for efficient IN-predicate evaluation 65

5.3 Query details for mono and multi-column selections . . . 75

5.4 Microbenchmark runtime for different column-column comparison operators on two dates of the `Lineitem` table ($s = 100$) in MonetDB 76

5.5 Microbenchmark runtime for different column-column comparison operators between three dates of the `Lineitem` table ($s = 100$) in MonetDB . . . 77

5.6 Microbenchmark runtime for different IN-lists on the `p_container` attribute of the `Part` table ($s = 100$) in MonetDB . . . 78

5.7 Microbenchmark runtime for different IN-lists on the `p_size` attribute of the `Part` table ($s = 100$) in MonetDB . . . 79

5.8 Query execution times of Elf and MonetDB's full-table scans for our six selected TPC-H queries ($s = 100$) . . . 82

6.1 InsertElf for a 5-dimensional data set . . . 91

6.2 (a) Elf with marked tuple to be deleted, (b) reorganized Elf . . . 92

6.3 Build time for `Lineitem` table of $s = 200$ . . . 94

6.4 Normalized runtime overhead caused by different InsertElf sizes in the TPC-H queries on the `Lineitem` table ($s = 200$) 95

6.5 Accumulated runtimes for the four TPC-H `Lineitem` queries ($s = 200$) on a merged Elf (including merge time) and the sum of runtimes of a linearized Elf and InsertElf w.r.t. different InsertElf-to-linearized-Elf ratios 96

7.1 BitWeaving approach 100

7.2 Column Imprints 101

7.3 Sorted Projections 102

7.4 BB-Tree Structure 103

7.5 An exemplary table and an excerpt of its cube 104

7.6 The Data Dwarf of the cube from Figure 7.5 106

7.7 Inner node format of Seg-Tree 107

7.8 Index tree blocked in three-level hierarchy: First-level page blocking, second-level cache-line blocking, third-level SIMD blocking of FAST. . 109

7.9 Inner nodes of ART. The partial keys 0, 2, 3, and 255 are mapped to pointers of the subtrees. 111

List of Tables

2.1 Columnar selection predicate translation 8

4.1 Running example data 35
4.2 Upper bound storage overhead 41
4.3 Storage consumption for `Lineitem` table 46
4.4 Query details for mono and multi-column selections 49

5.1 Query details for our TPC-H queries 81

7.1 Comparison of the considered index structures based on extracted criteria 112

List of Code Listings

3.1 Scan loop 18

3.2 Scan loop with a predicated filter 19

3.3 k-times unrolled aggregation loop 21

3.4 Vectorized serial scan 23

List of Algorithms

1 Search multi-column selection predicate 42
2 Scan a `DimensionList` within an Elf 43
3 Scan a `MonoList` within an Elf 44

4 Column-column queries on a `DimensionList` 63
5 Column-column queries on a `MonoList` 64
6 IN-predicate evaluation on hash map 66
7 IN-predicate evaluation on a `DimensionList` 67
8 IN-predicate evaluation on a `MonoList` 69
9 Additional MAL operators for an integration of Elf 72
10 Excerpt of MAL plan of TPC-H `Q17` 73
11 Excerpt of MAL plan of TPC-H `Q17` using the integrated Elf 74

12 Building an Elf 88
13 Merge a linearized `DimensionList` within a `DimensionList` of the InsertElf 91

1. Introduction

One of the first questions that I came across on my first conference was:

What are the three most important things a databaser wants?

The answer is both, totally hilarious but also truly honest and insightful:

- Performance
- Performance
- Performance

Christopher Ré, EDBT 2014 Keynote.

Although not being the only feature that a customer expects from a database system, *performance* is one of the most important optimization criteria for a database system. Especially in the prevailing workload scenarios[1] *online transaction processing (OLTP)* and *online analytical processing (OLAP)*, there is a plethora of work optimizing the performance of these workloads [Ros04, KSC+09, PPI+14, BBS16, ABP+17, ACP+18, RBB+18]. With the increase in main-memory capacities, especially main-memory database systems, which usually store the entire database in main memory, became popular adding further tuning knobs [BMK99, BKM08]. For instance, due to the missing bottleneck of disk access, the actual processing cost of the CPU matters, which is essential for long-running OLAP operators. Hence, designing hardware-sensitive OLAP database operators has become a hot research topic [PRR15, PMZM16].

An important yet challenging operator to optimize for is the selection operator because of the following two reasons. First, a selection – or filter – is usually executed on the input data tables. Hence, it is confronted with the whole original table, which means that the volume of processed data is usually more than for subsequent operators, which work only on the filtered set of qualifying tuples. Second, the

[1] apart from the hot research topic of *hybrid transaction analytical processing (HTAP)* [AKPA17]

selectivity itself (i.e., the fraction of filtered out tuples) poses a big challenge for selections, because the best strategy for optimal performance is highly dependent on the selectivity. As we will see in this thesis, the selectivity is essential to choose the best operator for optimized performance and also to avoid performance degeneration by even several magnitudes. To summarize, both the high data volume processed by selections and the varying selectivity are important challenges that we want to tackle. Hence, we frame the overarching goal of this thesis as optimizing selections on a single column as well as combining the selective power of selections on multiple columns. In fact, this means that we explore how to optimize full-table scans for a single column beyond state of the art approaches and how to overcome their conceptual limitations for selections on multiple columns. In the following, we define this goal in more detail.

1.1 Goal of this Thesis

The goal of this thesis is to investigate how to optimize selection predicate evaluation of main-memory database systems beyond state-of-the-art approaches at two levels:

Level 1: At this level, we analyze full-table scans on a single column and optimize them for the underlying hardware using code optimizations. This means, we push the working horse to its physical limits in order to create a comprehensive baseline as our competitor.

Level 2: The second level advances the field in the direction of efficiently evaluating selections on multiple columns (i.e., multi-column selection predicates). To this end, we aim for a clever design of a multi-dimensional sort-based index structure to beat the baseline from Level 1 as well as other traditional index structures.

In the following, both levels are explained in detail.

Level 1: Hardware-Sensitive Scans

With the missing I/O bottleneck, the actual processing time of the CPU is the new tuning factor. In fact, exploiting processor-specific capabilities is an essential part to reach the best performance [BTAÖ13, BBHS14]. Since processing capabilities change every processor era, it is necessary to abstract general optimization concepts from their specific implementation in order to apply them when they are needed. However, so far, there is no comprehensive study about different hardware-sensitive optimizations (i.e., code optimizations) for selections on a single or multiple columns. More severely, not all code optimizations are beneficial for all selection workloads (i.e., selectivity and columns) and, hence, the right set of optimizations has to be chosen for each query individually. Especially the last argument calls for an automated approach to generate hardware-sensitive selection code for the actual use case.

In summary, we aim to answer the following research questions as the first goal of this thesis:

RQ 1: Which optimizations are reasonable for a full-table scan and under which circumstances do they outperform each other?

RQ 2: How can we ease the implementation effort of hardware-sensitive operations for heterogeneous hardware?

As a result of these research questions, we present the first full-table scan approach that is able to optimize itself during runtime until its performance is best optimized. This will act as a hardware-sensitive gold standard for selection performance. In addition to that, the optimizations are universally applicable to other operators beyond simple full-table scans.

Level 2: Accelerating Multi-Column Selection Predicates

While many queries include at least one selection predicates, there is a reasonable amount of queries involving a predicate on several columns of a table [Tra14]. Hence, optimizing queries for multiple column predicates is an important goal. An essential observation in Level 1 is that the execution of any selection with full-table scans (naturally) is dominated by the data size, not by the cardinality of the result. We argue that it should be the opposite: small results, as they are common for multi-column selection predicates, should be much faster computed than large results. Due to their usually combined low selectivity, it is obvious to improve their execution with a multi-dimensional index structure[2]. However, current research rather focuses on reaching bare-metal speed with accelerated scans or synopsis structures that are not able to exploit the combined selectivity as a multi-dimensional index structure. To achieve the goal of Level 2, we introduce the concept of a multi-dimensional sort-based index structure, which we call Elf, to support multi-column selection predicates. Of course, SQL offers more than comparing a column value to a constant [ISO99]. Hence, we aim to extend this index structure for more complex predicates that include IN-predicates and column-column comparisons to reach query support that is compatible with usual full-table scans in order to compare Elf to the state of the art beyond multi-column selection predicates. In summary, this will answer the following research questions:

RQ 3: What is an efficient data structure to exploit multi-column selection predicates beating hardware-sensitive scans?

RQ 4: How to support more complex selection predicates in Elf and what is the benefit against full-table scans?

As the goal is to provide a universal index structure, there are several further challenges to be solved beyond delivering good selection performance. To reach this goal, there are two challenges. First, although our index structure Elf delivers the same set of results, the order of our qualifying tuples usually differs to the one of

[2]Please mind that result combination of partial results from one-dimensional index structures like CSB-Trees [RR00] is often way more expensive than a full-table scan. This is due to the rather low filter rates of single predicates.

full-table scans due to the internal sorting of Elf. Hence, it is important to identify the impact of Elf's result order on subsequent operators. Second, an important property of universal index structures for OLAP is to support maintenance tasks such as frequent updates efficiently. Hence, we finalize the thesis by answering the following research questions:

RQ 5: What is the benefit and drawback of using Elf in main-memory database systems?

RQ 6: How to exploit Elf's design to efficiently execute maintenance tasks (i.e., updates and appends)?

As a result of both levels, we can equip a query engine with our combination of approaches as a Swiss army knife that efficiently evaluates arbitrary SQL predicates by selecting the best approach from the two levels. In summary, these are hardware-sensitive scans with peak performance for a limited set of selected columns or lower selectivities and Elf for the common pain point of main-memory database systems – highly-selective multi-column selection predicates.

1.2 Structure of the Thesis

In order to present the contributions of the thesis in understandable chunks, we divide the thesis into 8 chapters. In the following, we give a brief overview of the content chapters Chapter 2 to Chapter 6 as well as related work in Chapter 7 by including references to shared material from own previous publications. Of course, the thesis is rounded up by a conclusion in Chapter 8 and future work in Chapter 9 that has to be carried out to complete the Swiss army knife of this thesis.

Chapter 2 – Selections in the Rear-View Mirror

In Chapter 2, we present the basics that should be well understood before going deeper into the topic of hardware-sensitive selections and multi-dimensional indexing. Hence, we give basics about selections from the relational algebra and SQL. Furthermore, we review CPU capabilities and hardware-sensitive programming frameworks to give the background for our Level 2 of our contributions. The chapter shares content with:

> David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward Hardware-Sensitive Database Operations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 229–234, 2014

> David Broneske. Adaptive Reprogramming for Databases on Heterogeneous Processors. In *SIGMOD/PODS Ph.D. Symposium*, pages 51–55. ACM, 2015

Chapter 3 – Hardware Sensitive Full-Table Scans as Working Horse

The first content chapter, Chapter 3, focuses on the first goal of this thesis. In this chapter, we review important code optimizations and their combination in order to push the performance of full-table scans by optimizing them for the underlying hardware. As a result, we present concepts for answering **RQ 1** and also present a framework for automatically optimizing arbitrary database operators by applying code optimizations for **RQ 2**. This chapter is based on:

> David Broneske, Sebastian Breß, and Gunter Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, Lecture Notes in Computer Science (LNCS), pages 97–111. Springer, 2014
>
> David Broneske. Adaptive Reprogramming for Databases on Heterogeneous Processors. In *SIGMOD/PODS Ph.D. Symposium*, pages 51–55. ACM, 2015
>
> David Broneske and Gunter Saake. Exploiting Capabilities of Modern Processors in Data Intensive Applications. *it - Information Technology*, 59(3):133–140, 2017
>
> David Broneske, Andreas Meister, and Gunter Saake. Hardware-Sensitive Scan Operator Variants for Compiled Selection Pipelines. In *Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 403–412, 2017
>
> Lars-Christian Schulz, David Broneske, and Gunter Saake. An Eight-Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors. *Proceedings of the VLDB Endowment*, 11(11):1550–1562, 2018

Chapter 4 – Elf as Multi-Column Selection Predicate Index

In Chapter 4, we address the conceptual shortcoming of full-table scans of executing multi-column selection predicates. To this end, we design a multi-dimensional sort-based index structure, called Elf, whose design exploits the underlying characteristics of the data of multiple columns. As a consequence, we answer **RQ 3** by showing that index structures still have a wide use case for selections in main-memory database systems. The chapter shares content with:

> David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017
>
> David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *Transactions on Knowledge and Data Engineering (TKDE)*, 2018. Accepted in April 2018

Chapter 5 – Complex Selection Queries in Elf-Supported Main-Memory Database Systems

In Chapter 5, we extend Elf to support more complex selection predicates from SQL beyond the definition from the relational algebra. This endeavor answers **RQ 4** and compares the benefits of a multi-dimensional index structures compared to full-table scans. Furthermore, we answer **RQ 5** in this chapter by integrating Elf in MonetDB as a representative of a successful main-memory database management system. The content of this chapter is so far not published, but will be published as a next step.

Chapter 6 – Elf Life Cycle

In the last content chapter, Chapter 6, we introduce important maintenance tasks to our multi-dimensional index structure Elf. To answer **RQ 6**, we present Elf's build algorithm and a delta-stored-like insertion procedure to cope with frequent inserts. This chapter is based on and shares content with:

> David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017
>
> David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *Transactions on Knowledge and Data Engineering (TKDE)*, 2018. Accepted in April 2018

Chapter 7 – Related Work

In Chapter 7, we review related work for this thesis. This related work includes a description of our chosen competitors. Furthermore, we review related multi-dimensional index structures. This chapter shares content with:

> Veit Köppen, Martin Schäler, and David Broneske. *Emerging Perspectives in Big Data Warehousing*, chapter Index Structures for Data Warehousing & Big Data Analytics. IGI Global, 2019
>
> Marten Wallewein-Eising, David Broneske, and Gunter Saake. SIMD Acceleration for Main-Memory Index Structures – A Survey. In *Proceedings of the International Conference Beyond Databases, Architectures and Structures (BDAS)*, pages 105–119. Springer, 2018

2. Selections in the Rear-View Mirror

Selections are one of the main operations in the relational algebra, because an essential functionality of the database system is to provide only tuples to the user that qualify specific properties. This is especially import in analytical scenarios, where time-base, region-based, and product-based selections are executed before a set of subsequent joins and aggregations [Inm05]. Our goal of the thesis is on two different approaches for selections. Essentially, a selection can be executed using a full-table scan or an index-structure and we aim to contribute to both these parts.

In the following, we first review the basics about selections in database management systems including a small classification of selection predicates and favorable approaches. Since the first contribution is to optimize full-table scans to the used hardware, we additionally present the overall processing capabilities of CPUs and, afterwards, introduce state of the art for supporting heterogeneous hardware properties in generic programs.

These three parts introduce the fundamental concepts to understand our design decisions in the later chapters and also help to differentiate our contributions from other work.

2.1 State of the Art in Selection Predicates

A typical database operation is to restrict the tuples to a user-defined subset of tuples that qualify specific properties [LP13]. This operation is usually called a selection, filter, or SELECT operator [SSH18]. Due to their importance, selections have a high optimization potential. For example, there is no query in the TPC-H [Tra14], SSBM [OOC09], and also TPC-DS [Tra15] benchmark (the standard OLAP benchmarks) not having at least one selection predicate. Conceptually, the selection splits a relation into two horizontal partitions, where one partition contains all qualifying tuples and the other all non-qualifying tuples [EN15] – this observation

is used for example in database cracking [KM05, IKM07]. However, usually only the qualifying tuples matter, because they are the input for subsequent operators.

In the following, we first introduce the basic concepts from the relational algebra in order to create a common ground of a selection predicate for this thesis. Afterwards, we give a characterization of the underlying predicates (i.e., mono and multi-column selection predicates) that helps to understand the use cases of different selection operator implementations (i.e., full-table scans and index structures).

2.1.1 Relational Selection Basics

In the relational algebra [Cod70], the schema of a selection is as follows: $\sigma_{cond}(R)$. The operator σ represents a selection that filters non-qualifying tuples of relation R for the condition *cond*. The condition is usually defined on a set of columns C of relation R with $C \subseteq R$. For each column $col \in C$, there is usually one of the following basic predicates given: $\{=, <, >, \leq, \geq, \neq\}$ [EN15].

Given the definition of a selection predicate, we can transform these predicates into an interval-based notation that helps us to treat them in a uniform manner across different selection approaches. In Table 2.1, we present how to transform a given predicate to an interval-based notation including a specialization for integer value ranges. For example, $col = x$, where x is a scalar value within the domain of this column's type, is translated to the interval $[x, x]$, where x indicates the lower and upper boundaries and both are included in the interval. By contrast, $col < x$ defines an interval where the lower boundary is the domain minimum (min) of this column and x defines the first value that is not included in the interval. Notably, it is possible to express $\neq$ as two intervals.

Predicate	Interval	Interval (Integer)
$= x$	$[x, x]$	$[x, x]$
$< x$	$[\min, x)$	$[\min, x - 1]$
$\leq x$	$[\min, x]$	$[\min, x]$
$> x$	$(x, \max]$	$[x + 1, \max]$
$\geq x$	$[x, \max]$	$[x, \max]$
$\neq x$	$[\min, x) \cup (x, \max]$	$[\min, x - 1] \cup [x + 1, \max]$
$\geq x$ and $\leq y$ with $x \leq y$ (BETWEEN)	$[x, y]$	$[x, y]$

Table 2.1: Columnar selection predicate translation

Of course, these interval predicates work well for attributes from an *ordered domain*, i.e., numeric values or dates [EN15]. In addition, string values can be compared according to their orthographic order – please mind that numbers are differently ordered when using an orthographic ordering. However, a special property of database systems is to support enumerations as well, usually representing an unordered value domain. For example, car models of a car manufacturer do not contain an intrinsic order and, hence, can usually only be compared on equality or inequality.

In summary, the redefinition of a selection predicate as an interval predicate allows to create a common interface of a selection for arbitrary predicates. It is only necessary

to encode a lower and an upper border for the comparisons. Of course, comparisons matter and, hence, specializations for these intervals can be made for full-table scans for example. However, for our traversal-based multi-dimensional index structure presented in Chapter 4, the interval-based representation is useful.

2.1.2 Predicate Characteristics

Since predicates are user defined, selection predicates usually differ in complexity. For determining efficient execution strategies, important properties are the predicate selectivities and the number of involved columns of a table.

Selectivity

The term *selectivity* is not consistently used in literature. Selectivity could be used to express how many tuples are filtered out or how many tuples qualify the selection. In this thesis, we stick to the following definition: selectivity represents how many tuples are filtered out. Hence, low-selectivity workloads filter out only a small fraction of tuples (i.e., a big fraction qualifies the predicate), while high-selectivity workloads filter out a big fraction of tuples (i.e., only a small number of tuples qualify the predicate). For simplicity, we also use the term *selectivity factor*, which represents the fraction of data that qualifies a predicate. Hence, the selectivity factor is invers to the selectivity.

The selectivity plays a vital role when deciding which technique to use for executing the selection. For low-selectivity workloads, a full-table scan is usually superior to indexing techniques [SSH18]. This is due to an introduced overhead per tuple (e.g., index traversal, random memory access), which does not pay off for a big fraction of qualifying tuples. On the other hand for high-selectivity workloads, using a special indexing technique is beneficial, because it only touches promising candidates in contrast to a full-table scan touching all data by definition.

Number of Columns

Depending on the number of columns in the condition *cond*, we can differentiate between mono and multi-column selection predicates.

Mono-Column Selection Predicates: For a mono-column selection predicate, there is only a predicate on a single column of the table (i.e., $|C| = 1$). Mono-column selection predicates are well supported by accelerated full-table scans (e.g., SIMD Scans [WBP+09, WOMF13], BitWeaving [LP13], Column Imprints [SK13]) and one-dimensional index structures (e.g., CSB-Tree [RR00]). However, recent studies suggest that especially in main-memory database systems, the threshold for using an index structure instead of a full-table scan has drastically moved towards full-table scans [DYZ+15].

Multi-Column Selection Predicates: A multi-column selection predicate is defined for a set of columns C, where $|C| > 1$. The basic challenge of multi-column selection predicates is that the selectivity of the overall query is often small, but the selectivity for each column is high enough that a query optimizer would

decide to use a scan for all columns [SSH11]. Thus, we cannot use only one column that dominates the query and use traditional indexes, such as B-Trees, and then perform index lookups for the remaining tuple identifiers on the other columns. As a result, most used approaches are optimized column scans that exploit the full speed of the processing unit [LP13, SK13].

2.1.3 Selection Result Representation

Especially in bulk-processing engines, the representation of an intermediate result of a selection predicate can be manifold, depending on the used main-memory database system. Common representations are position lists, bitmaps, or even materialized intermediate columns or column groups [AMDM07]. Without loss of generality, we focus in this work on position lists as intermediate results of the complete selection, which we define as R_{mcsp} as follows:

Definition 2.1 (Result position list: R_{mcsp})**.** *Let Ref_i denote the tuple identifier of the ith tuple (t_i) in the data set. Moreover, let $SAT_{mcsp}(Ref_i)$ be a Boolean function that is true, iff all attribute values of t_i for all columns are defined in the interval by query mcsp. Then, R_{mcsp} is a list of identifiers such that*
$Ref_i \in R_{mcsp} \Leftrightarrow SAT_{mcsp}(Ref_i) = true.$

The definition of our position list complies to the standards of main-memory database systems such as MonetDB [BK99] using equivalent representations. Notably, R_{mcsp} is the final result of a selection which is used in this thesis for interoperability with subsequent operators. Of course, we use bitmaps for synchronizing intermediate selection results of multi-column selection predicates on accelerated full-table scans (the standard way for BitWeaving). A final conversion step is then constructing the final position list R_{mcsp}.

2.2 CPU Capabilities

Since selections usually incur a branch per tuple depending on the outcome of the predicate, the *central processing unit (CPU)* is usually well suited for evaluating selection predicates. The CPU is the main processor of the computer and is also the standard processor for any database operation. With the rise of main-memory database systems, the bottleneck shifted from I/O to efficient processing. Hence, optimizing for cache-efficient access and also CPU-efficient algorithms is the new goal. Hence, we review important characteristics of CPUs in this section that are necessary to reach peak selection performance. We start by introducing the common processing paradigm of CPUs – *pipelining* – where processing steps of different instructions are executed in parallel to increase instruction throughput. A good example to explain pipelining is the RISC pipeline, which we introduce in the next section. However, due to the dependencies between the execution of different instructions, pipeline efficiency is reduced by so-called *hazards*, which we present in Section 2.2.2.

2.2.1 Pipelining in CPUs – the RISC Pipeline

CPUs are optimized for instruction throughput, which is measured in *instructions per cycle* [HP07]. In an ideal world, the CPU will execute one instruction per cycle. However, since instructions include different overhead, the latency of an instruction differs w.r.t. the work to be done (e.g., executing an addition on a register is faster than loading a value from main memory into a register) and a common clock rate is hard to define. To this end, tasks were split into a pipeline of subtasks, where each subtask has an equally long latency. Furthermore, subtasks of different instructions can be executed in parallel to allow for parallelism. A classical example of a pipeline is the RISC pipeline [HP07], which we show in Figure 2.1. The RISC pipeline consists of five stages:

Instruction Fetch (IF): In the instruction fetch cycle, the operation code (also: OP code – a number identifying the operation) pointed to by the program counter (PC) is fetched from memory and the program counter is incremented.

Instruction Decode / Register Fetch (ID): The main purpose of the ID stage is to decode the instruction and detect and finish a branch instruction, which may manipulate the PC. In parallel to that, the given register content is fetched.

Execution / Effective Address (EX): The EX stage performs the operations on the fetched operands. For a memory reference (load/store), the effective address is computed, else (e.g., for an addition) the result of the function is computed.

Memory Access (MEM): For a load or store instruction, the operand is read from or stored at the effective address from the last cycle.

Write Back (WB): For computations on registers or load instructions, the result is written into the register file.

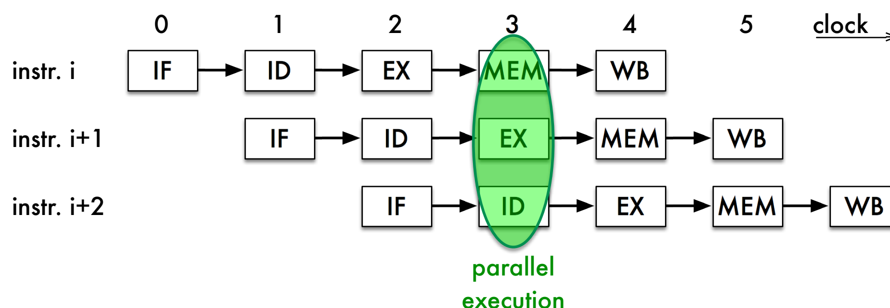

Figure 2.1: RISC pipeline

The RISC pipeline is a good example to show limitations of pipelining, which we do in the next section by introducing hazards. However, a five-stage pipeline is easy to implement, but will neither bring optimal performance, nor good power consumption. Hence, CPU vendors came up with numerous improvements of the classical pipeline. These include the introduction of branch-predictors, μops and an extension of the pipeline to up to 31 stages for an Intel Pentium 4. However, current CPUs feature pipelines of a length around 20, because longer pipelines bring a bigger impact of pipeline stall due to hazards.

2.2.2 Hazards

In this section, we present the two most important hazards that can be avoided with code optimizations. These two hazards are data and control hazards. In general, a hazard forces the CPU to include stalls into the pipeline, because two or more operations depend on each other and are serialized (or can only partially overlap).

2.2.2.1 Data Hazard

A data hazard exists if for two operations i_1 and i_2, the result of operation i_1 is used by operation i_2 directly or via transitivity [HP11]. In Figure 2.2, we show an example of a data hazard. The data that is loaded by the first instruction is needed as an operand for the subtraction and the logical `AND`. However, since the result of the load is only stable after the WB stage, the other operations have to be stalled in order to read the right result from the register[1].

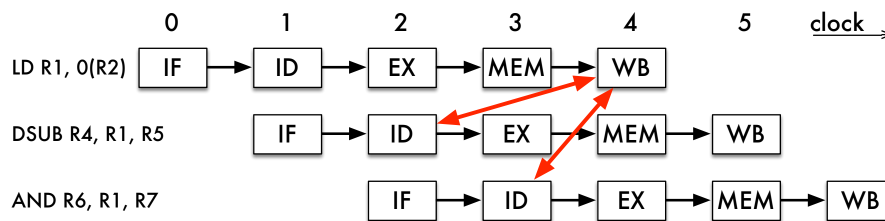

Figure 2.2: Example of a data hazard

To reduce the number of stalls due to data hazards, *instruction reordering* is an essential optimization of compilers. However, this is only possible when there are enough independent operations. For instance, tight loops with small amount of work inside the loop will be prone to data hazards.

2.2.2.2 Control Hazard

A control hazard exists if there is a branch instruction that may manipulate the program counter w.r.t. a given condition [HP11]. Branch instructions are mainly produced by `if`-clauses, but also `for`/`while` loops include a branch instruction. In these cases, instructions before the branch instruction cannot be scheduled after the branch and instructions depending on the branch outcome cannot be scheduled before the branch instruction. Since the outcome of a branch instruction is known only after the ID stage, stalls may have to be included.

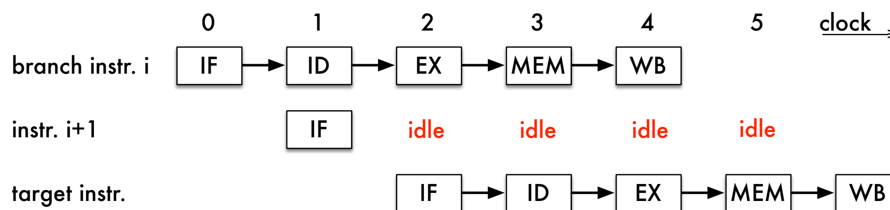

Figure 2.3: Example of a control dependency (`instr. i+1` was the wrong decision)

[1]Notably, the technique *bypassing* can forward the result after the EX stage [Blo59].

There are different optimizations that one could take. The obvious one is to pipeline the instruction that would come if the branch was not taken, because branches are rather included to prevent from rare exceptions. A more robust technique is branch prediction, where CPUs keep a history of the targets of a specific branch (a *branch target buffer*). In this case, the instruction is pipelined that is likely to be executed according to the branch target buffer. Apart from the used technique, whenever the CPU pipelines the wrong instruction, it has to flush the pipeline as shown in Figure 2.3.

2.2.3 Single Instruction Multiple Data

A lot of the design considerations for a CPU depend on the instructions and the number of used registers, but not on the size of the register. Hence, CPU vendors include special registers with wider register sizes (e.g., 128-bits XMM registers). Now, several values can be packed into this register and the same instruction is executed on all the values in a data-parallel fashion. This is called *Single Instruction Multiple Data (SIMD)*.

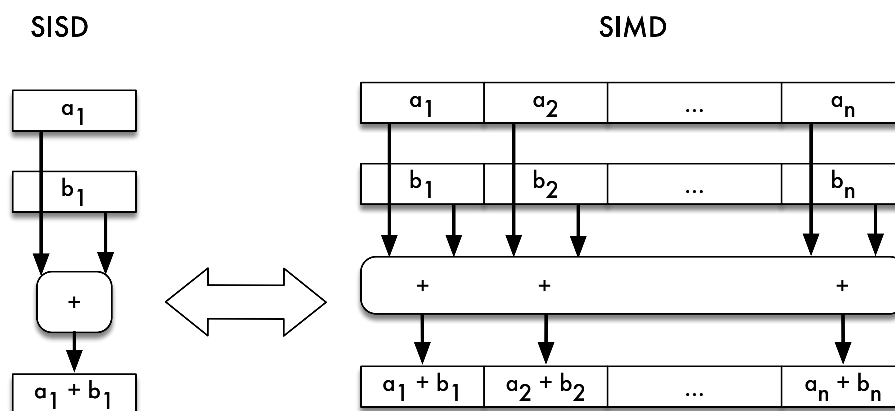

Figure 2.4: Single Instruction Single Data (SISD) vs. Single Instruction Multiple Data (SIMD)

In an ideal world, for a register that fits s values, a speedup of s is achieved. For example, using the Streaming SIMD Extensions 2 (SSE2) on the example in Figure 2.4, we can add four values in one clock cycle – exhibiting usually four times the speed. Hence, SIMD is an important capability to optimize for.

In practice, however, several limitations come into play. For example, the data should be aligned for reaching peak performance and values to process should fill a SIMD register fully [BS17b]. Furthermore, we cannot execute branching in SIMD, because we cannot branch for each packed value individually in a SIMD fashion [ZR02]. Thus, an algorithm has to be carefully tuned with SIMD instructions for having a benefit of SIMD.

2.3 Heterogenous Programming

Due to ever-changing hardware characteristics, a once optimized algorithm may have to be re-optimized with new arising capabilities of processors. Since our first goal of

the thesis is to generate hardware-sensitive scan code for current and future CPUs, we review the two main paradigms to achieve hardware sensitivity: the hardware-sensitive paradigm, in which the algorithms are tuned to one specific processor; and the hardware-oblivious paradigm, which means that the algorithms are abstractly defined and efficiently executed using a processor-dependent driver [HSP+13]. We depict a sketch of these two paradigms in Figure 2.5 and explain their characteristics in detail in the following.

2.3.1 Hardware-Sensitive Programming

The main idea behind hardware-sensitive programming is that the programmer knows the system that the algorithm is written for in detail. Hence, in a database, programmers would write a set of operators per processor and tailor the code to the underlying hardware by fully exploiting the hardware's properties.

With this approach, programmers are able to reach the best performance, because they know what hardware to program for [HSP+13]. However, this approach does not scale to a high amount of different processors. The reason is that with each new processor, another set of operators has to be implemented, although they may only differ slightly. Thus, the development and maintenance effort is too high in this approach, especially if we have in mind the increasing heterogeneity of future processors.

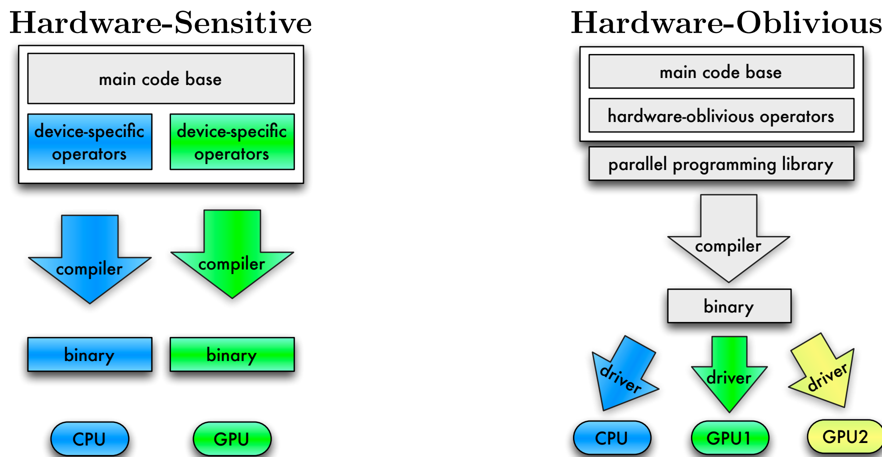

Figure 2.5: Hardware-sensitive vs. hardware-oblivious programming. Adapted from [HSP+13]

Hardware-Sensitive Database Operators

Despite the high development overhead, tuning operators to the underlying hardware has found much attention in research. First tuning focusses on optimizing main-memory database operators for different CPUs and the cache hierarchy.

CPU

Early work in this area includes to tune database selections using vectorization, e.g., the work of Zhou and Ross [ZR02], or predication in the work of Ross [Ros04].

Further optimizations for selections include vectorized scans on compressed data for single [WBP+09], and complex predicates [WOMF13], as well as using vectorized bloom filters for the scan [PR14].

Recent studies consider even more complex operators such as joins and aggregations. Here, Zukowski et al. optimize hash tables and functions to the underlying hardware [ZHB06]. Based on that, efficient vectorized aggregation functions are proposed by Polychroniou and Ross [PR13] to speed up aggregations in databases. Simultaneously, the debate about the best join algorithm has been revitalized, leading to even more specialized and tuned join algorithms. These include the *massively parallel sort-merge (MPSM) join* by Albutiu et al. [AKN12], the sort-merge join using SIMD-accelerated sorting networks by Kim et al. [KSC+09], and the radix join which has been initially proposed by Boncz et al. [BMK99] and further improved with vectorization and additional optimizations by Balkesen et al. [BATÖ13, BTAÖ13].

GPU

Early work considering database operations on GPUs has been published by He et al. [HLY+09], which uses highly optimized primitives on the GPU whose combination can compute any database operation. Furthermore, Sitaridi and Ross [SR13] present an efficient selection with GPU's missing branch-prediction capabilities. Furthermore, several papers outline how to adapt joins to GPUs [HYF+08, KLMV12].

Other Processors

Furthermore, there is work presenting how to design database operations for more specialized co-processors. He et al. [HLH13] show how to tailor the hash join to work efficiently on an APU (a CPU with an integrated GPU) and Jha et al. [JLC+15] investigate hash joins on an Intel Xeon Phi. Moreover, Mueller et al. [MTA11] extensively discuss how to design sorting algorithms to implement them efficiently on FPGAs (field-programmable gate arrays).

All these publications show that tuning algorithms to the underlying hardware can improve performance by orders of magnitude. Nevertheless, all of them are only tailored to a single (co-)processor and do not provide a comprehensive solution for the increasing heterogeneity of the hardware landscape.

2.3.2 Hardware-Oblivious Programming

In contrast to hardware-sensitive programming, hardware-oblivious programming includes an additional abstraction layer: a parallel programming library. With this library, database operations are implemented without explicit knowledge of the hardware based on the parallel programming library (e.g., OpenCL), which then compiles a binary for each processor [HSP+13]. This binary is executed using a specialized driver for each processor which should exploit special hardware capabilities of the (co-)processor.

The advantage of hardware-oblivious programming is, that code for each operator is written only once and hardware-related properties (e.g., parallelization possibilities)

are included by the driver. As a result, development and maintenance overhead is reduced to a minimum. However, the compiler and driver optimize algorithms for the average use case and cannot take the workload into account. Furthermore, an efficient execution and exploitation of hardware capabilities always relies on a good implementation of the driver. Thus, it is not guaranteed that the hardware-oblivious approach always provides the best performance. Differently phrased, the hardware-oblivious programming approach allows for code portability but not for performance portability [RHVM15]. Additionally, the driver is mainly designed to optimize for the general use case. Hence, we are not able to fully exploit the domain knowledge that we have in database systems about the workload.

2.4 Summary

In this section, we reviewed important basics to understand this work. We introduced selection predicates and classified them to show their sweet spots for different operator implementations (full-table scans vs. index structures). As a result, we clarified for which use cases which of our next approaches is best suited. Furthermore, we presented important properties of the CPU that are important to optimize a full-table scan for. In concert with the definition of the two heterogeneous programming paradigms, we are ready to introduce a hybrid approach of hardware-sensitive and hardware-oblivious programming in the next chapter. The approach, which we call adaptive reprogramming, is able to optimize selection operator implementations to modern hardware capabilities during runtime.

3. Hardware Sensitive Full-Table Scans as Working Horse

As introduced in the previous chapter, accelerated full-table scans are the straightforward method to execute a selection. However, with an increasing diversity of hardware [HSP+13, BBHS14], the number of possible tuning opportunities increases as well [BBS14]. In this chapter, we introduce our definition of the term *code optimizations* as means to exploit diverse hardware features for the full-table scan [BS17a]. With code optimized full-table scans, we create a first baseline that other approaches have to compete against. Furthermore, differently tuned operators perform best depending on the use case (data and processor characteristics). Hence, it is important to automate the generation of variants. To this end, we outline our idea of a framework to automatically apply and exploit code optimizations for database systems. This resulting framework serves as a solution for the problems of Level 1, optimizing full-table scans to reach *bare-metal speed*.

In summary, we make the following contributions in order to answer the research questions RQ 1 and RQ 2:

- Definition of *code optimizations* as a means to reach hardware-sensitivity
- Pointers to applications of code optimizations in other database operators
- Evaluations of *baseline implementations* for accelerated full-table scans
- Proposal of a *code generation framework* for hardware-sensitive database operators

3.1 Code Optimizations for Hardware-Sensitive Full-Table Scans on Single Predicates

In this section, we look at important code optimizations that allow for hardware sensitivity. We define the term code optimizations as [BS17a]:

```
1  for(int i = 0; i < column_size; ++i) {
2    if(column[i] < predicate) {
3      result[pos++]=i;
4    }
5  }
```

Listing 3.1: Scan loop

"A code optimization changes the code in order to improve its performance for a specific processor and/or workload without changing the code's external behavior"

In this regard, code optimizations are similar to a classical refactoring [FBBO99] in the way that both change the code without loss of its behavior, while code optimizations improve for performance instead of for maintainability/code comprehension.

Furthermore, code optimizations are optimizations that a compiler could detect and automatically apply. However, it does usually not apply such optimizations, because they are only useful for a specific combination of workload and processor. In contrast, the workload and system characteristics are well known in a specific database system and, thus, an execution engine can decide when to use a code-optimized variant.

Code optimizations, although no explicitly named, are already used in several systems. For instance, *flavors* in VectorWise [RBZ13] are primitives that are tuned with different code optimizations. In Section 3.3.4, we outline systems that currently exploit code-optimized operator variants. As a consequence, code optimizations usually have to be applied by hand and then exploited explicitly by the system.

In the following, we present a selection of code optimizations that are applicable to full-table scans. These include loop unrolling, software predication, single instruction multiple data (SIMD). In this regard, we also exemplify their performance impact as well as their usage in other database operators based on our literature review [BS17a].

3.1.1 Running Example

For illustrating the impact of different code optimizations, we start with a small example that we will use throughout this chapter. In Listing 3.1, we show the code of a branching scan with a single `less than` predicate on an arbitrary column. Please note, the `less than` predicate can be exchanged with any other predicate using template expansion, such that implementation effort for query engines is minimal.

The usual way of implementing a scan in a column-oriented operator-at-a-time engine is to have a `for`-loop iterating over the column values, in which we check with an `if`-statement whether the predicate is fulfilled. If it is fulfilled, we add the current position of the value to the result – in this case a position list as defined in Definition 2.1.

We show the baseline performance for the branching variant from Listing 3.1 in Figure 3.1.[1] The branching variant shows an increasing response time for increasing

[1]The experiments in this section are executed on an Intel Xeon E5-2609 v2 with 2.5 GHz frequency and 256 Kb of L1 cache, 256 Kb of L2 cache per core, and 10 Mb of L3 cache per core. Furthermore, all variants have been compiled with the GNU C++ compiler 4.6.4. with the same flags as used by Răducanu et al. [RBZ13]. Each scan evaluates the predicate on 60 million data items, which is around 240 Mb of data.

```
1  for(int i = 0; i < column_size; i++) {
2    result[pos]=i;
3    pos += (column[i] < predicate);
4  }
```

Listing 3.2: Scan loop with a predicated filter

selectivity factors (*Sel*) until reaching its peak at around *Sel*=50 %. Afterwards, its performance improves again. This behavior is caused by the branch prediction unit of the CPU to avoid control hazards by speculating on the outcome of the branch in Line 2 of Listing 3.1. However, especially at *Sel*=50 %, the branch predictor is wrong half of the time and has to flush the CPU pipeline which has already been loaded with the wrongly predicted instructions. Hence, several cycles are lost until the pipeline is filled up again and can work at an *Instructions per Cycle (IPC)* of 1.

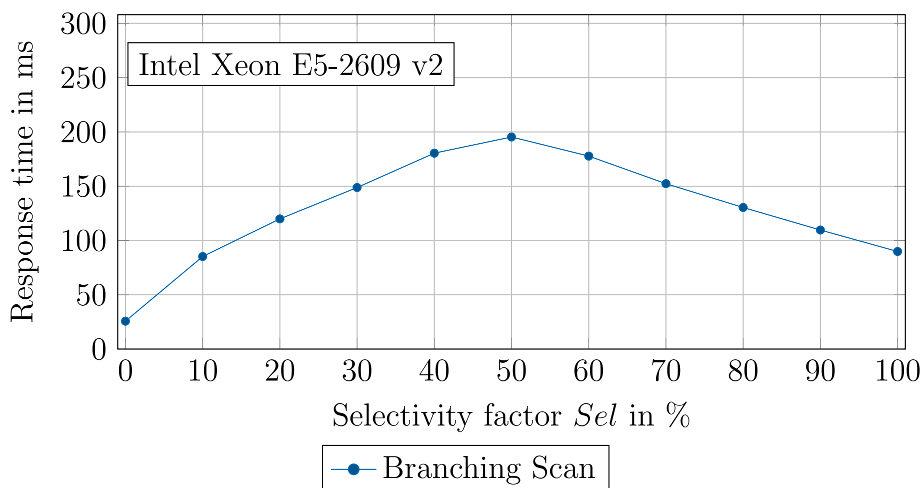

Figure 3.1: Response time of a branching scan on 30 million data items

3.1.2 Software Predication

Since branches inside the code hinder performance when their outcome is hard to predict, *predication* is commonly used to eliminate branches and turn them into data flow. It can be implemented in hardware, e.g., by the Intel Itanium 2. Here, both branches are executed in parallel in different pipelines, but only the result of the correct pipeline succeeds at the end. This leads even to better performance than a predicated version in software [BZN05]. In contrast, *software predication* does neither execute both branches, nor does it speculate on the result of the condition. In our example, the result of the condition is used to manipulate the write position in the array. For example in Listing 3.2, we show the code for a predicated scan. However, instead of using an if-clause, the result of the condition (cf. Line 3) is used as the increment of the current write position. Since the result of the predicate is either 0 (false) or 1 (true), it can be directly used to increment the position.

Performance Impact of Predication

Predication is useful if the CPU cannot easily decide about the outcome of a branch, which is the case for selectivities other than around 0 or 100 %. In Figure 3.2, we

show the runtimes for the branching loop and the predicated loop from Listing 3.2. Especially for selectivity factors around *Sel*=50 %, the branching variant incurs big penalties due to many mispredictions that cause pipeline flushes. Here, predication in these tight loops also causes write hazards for creating the position list that further diminish performance [BBS14]. Another important observation is that the branching version is superior for selectivity factors below 2 % and above 98 %. While these thresholds seem to indicate only a small range of use cases, such selectivity factors are often enough part of analytical queries. Considering the commonly used TPC-H Benchmark [Tra14], the queries Q1 ($\sigma_{\texttt{Sel}}$(L_Shipdate) $>$ 98 %), Q2 ($\sigma_{\texttt{Sel}}$(P_Size) $<$ 2 %), Q8 ($\sigma_{\texttt{Sel}}$(P_Type) $<$ 1 %), Q14 ($\sigma_{\texttt{Sel}}$(L_Shipdate) $<$ 2 %), Q16 ($\sigma_{\texttt{Sel}}$(PS_SupplyCost) $>$ 99 %), and Q20 ($\sigma_{\texttt{Sel}}$(P_Name) $<$ 2 %) feature predicates that fulfill these boundaries. Hence, both variants should have their existence in query engines.

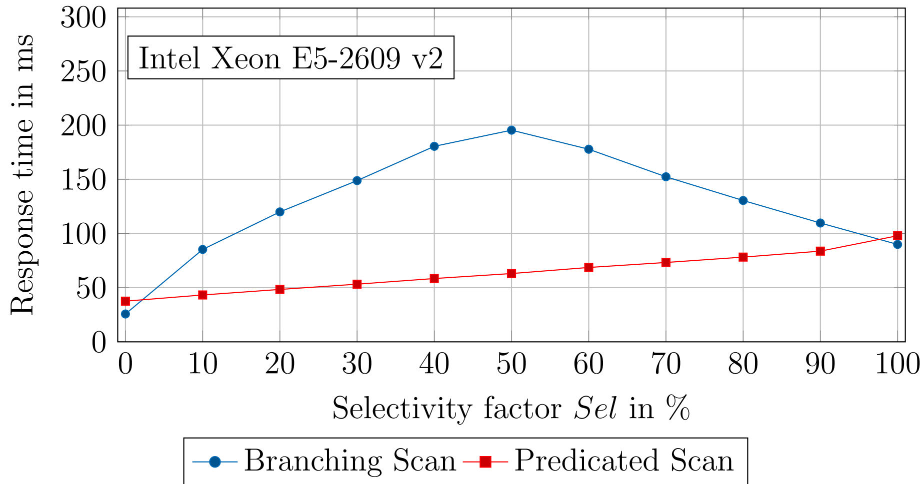

Figure 3.2: Response time of branching and predicated scans on 30 million data items

Software Predication in Database Operator Implementation

Since predication trades branches for writes, it essentially applies only to operators that include branches. Obviously, software predication has been used to eliminate branches in selections [RHVM15, BBS14, ZF15, Ros02, Ros04]. Furthermore, selections combined with aggregations or hash builds are a big application when using query compilation [Neu11]. In this regard, Broneske et al. show the impact of predication for filtered aggregation pipelines [BMS17] and Polychroniou and Ross for filtered in-place hash aggregations [PR13].

Furthermore, Zukowski et al. [ZHB06] use predication to remove branches in the hash table build of cuckoo hashing. This also applies for hash table probes, because scanning a hash bucket also includes a branch. Moreover, Kim et al. [KSC+09] use predication with conditional moves to implement sorting. Additionally, Pirk et al. [PPI+14] show predication's applicability for database cracking.

3.1.3 Loop Unrolling

Loop unrolling is a simple optimization that is already done by compilers. They replicate the instructions inside the body of a loop in order to allow for more

```
1  for(int i = 0; i+k < column_size; i+=k) {
2    //code for one iteration
3    if(column[i] < predicate) {
4      result[pos++]=i;
5    }
6    //unrolled iterations
7    if(column[i+1] < predicate) {
8      result[pos++]=i+1;
9    }
10   ...
11   if(column[i+k-1] < predicate) {
12     result[pos++]=i+k-1;
13   }
14 }
15 ... //process remaining tuples in a normal loop
```

Listing 3.3: k-times unrolled aggregation loop

possibilities in instruction rescheduling, which helps to reduce the impact of data hazards. However, compilers are often unable to find the best unrolling depth, e.g., for loops with a dynamic number of iterations. In these cases, loop unrolling has to be done by hand, which we show in Listing 3.3 where the loop body is unrolled by hand k times.

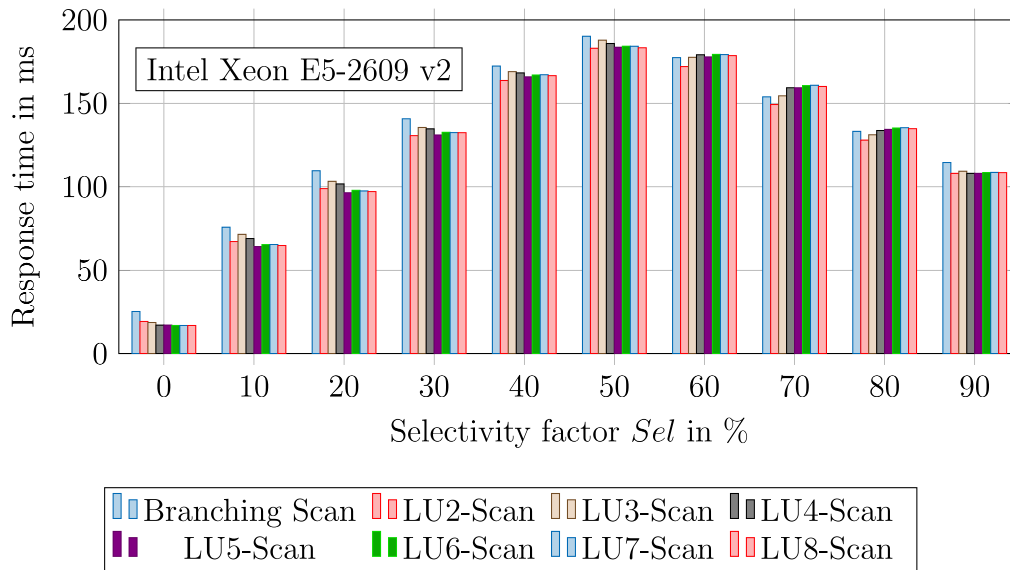

Figure 3.3: Response time of unrolled branching scans for varying selectivities for 30 million data items (LUn = n-times loop-unrolled)

We have already shown that the best unrolling depth depends on the used processor [BBS14]. However, the benefit of loop unrolling depends on two further criteria: (1) whether there are branches inside the loop and (2) whether the number of loop iterations is known.

Branches Inside the Loop

Branches inside the loop change the number of executed instructions, which makes it hard to find the right unrolling depth. Notably, the best depth will even depend

on the current iteration of the branch [BBS14]. Hence, operators that include a branch have a limited applicability. Those operators include selections and hash table probes.

In Figure 3.3, we show the response time for a scan with varying selectivities whose body was unrolled several times. We can see that for small selectivities a higher unrolling number is better, while for selectivity factors around 50 %, a 2-times unrolling is the best. Since the best unrolling number is hard to define, a modern compiler avoids an unrolling in this case and we have to apply this optimization by hand. In contrast to that, the behavior of the predicated version under different unrolling depths is rather homogeneous (cf. Figure 3.4). Here, an unrolling depth of eight is the best.

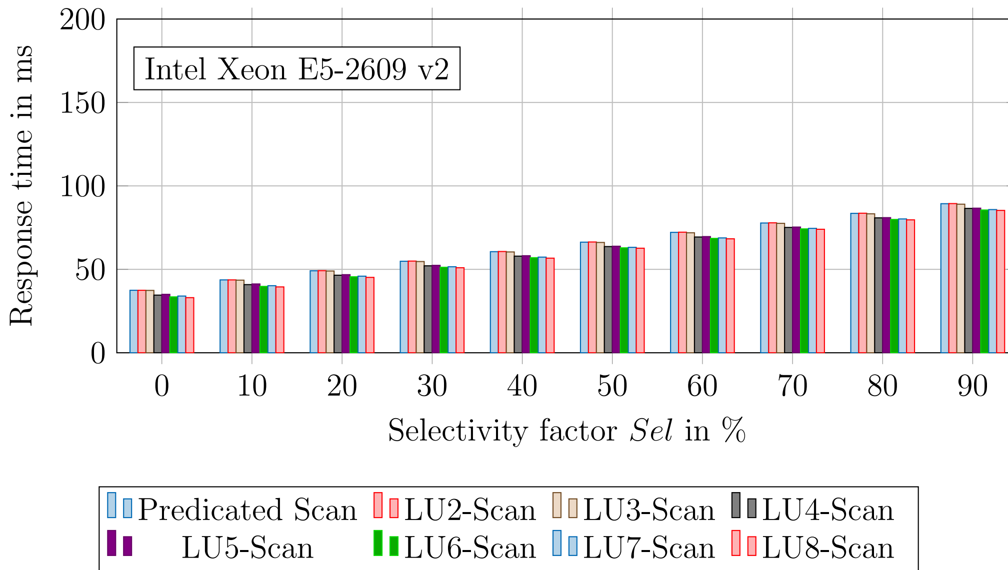

Figure 3.4: Response time of unrolled predicated scans for varying selectivities for 30 million data items (LUn = n-times loop-unrolled)

Number of Loop Iterations

Loop unrolling trades a reduction of loop iterations for an increased number of CPU registers to be used. Hence, the number of available CPU registers sets an upper bound. In the best case, the loop can be fully eliminated if the number of needed registers for an unrolling of the whole iteration space does not exceed available registers. Furthermore, if the iteration space is smaller than the unrolling depth, the unrolling is not used efficiently. Hence, only if the number of loop iterations is known beforehand (i.e., at runtime), it can easily be decided to which extent the loop should be unrolled.

Loop Unrolling in Database Operator Implementation

Loop unrolling can be used in every database operator of a vectorized or bulk-processing engine as they always loop over the content of a column. However, knowledge about the number of iterations is essential for the applicability of loop

```
 1 ... // Code for unaligned tuples
 2 for(int i=0;i < simd_array_size;++i)
 3 {
 4  mask=SIMD_COMPARISON(SIMD_array[i],comp_val);
 5  if(mask){
 6   for (int j=0;j < SIMD_Length;++j)
 7   {
 8    if((mask >> j) & 1)
 9     result_array[pos++]=j+offsets;
10   }
11  }
12 }
13 ... // Code for remaining tuples
```

Listing 3.4: Vectorized serial scan

unrolling. Consequently, only systems that employ a query compiler can fully benefit from loop unrolling as they can include meta data about table sizes [HKHL15]. Still hash table probes and loop unrolling have a limited applicability, because the sizes of hash buckets change and a specialization of the probe code is hardly practical.

Overall, loop unrolling has been applied in various work. Several researcher already apply loop unrolling for simple selections [RHVM15, BBS14] or for bloom filters [PR14] and it is used to optimize aggregations [RHVM15] or hashing [MSL+15]. Furthermore, loop unrolling is also used to apply SIMD (via *unroll-and-jam* [LA00]), which may cause further loops. For instance, Polychroniou et al. [PR15] use loop unrolling to remove those loops that iterate over SIMD register elements.

3.1.4 Single Instruction Multiple Data

Due to the advances in hardware, adapting algorithms to SIMD has become essential [PR14]. To exploit this optimization, we can implement the scan in Listing 3.4 based on the SIMD scan by Zhou and Ross [ZR02]. Since SIMD operations work best on aligned memory, we first have to process tuples that are not aligned. For this, we use the branching variant, since only a few tuples have to be processed. The same procedure is executed for the remaining tuples that do not completely fill one SIMD register. The presented code snippet evaluates the elements of a SIMD array and retrieves a bit mask for each comparison (cf. Line 4). After that, the mask is evaluated for the four data items and if there is a match, the corresponding position is inserted into the position list (cf. Line 6-10). Notably, similar to the algorithm by Zhou and Ross, we also use an `if`-statement for evaluating whether there has been a match at all, which could reduce executed instructions if the selectivity is high. Furthermore, we implemented a predicated version of the SIMD-accelerated scan by unrolling the loop which evaluates the branch.

Performance Impact of SIMD

Due to the usage of 128-bit registers for `int` values, we can compare 4 values at a time and can expect a performance improvement of a factor of around 4 in Figure 3.5. However, the final code shows only a run-time improvement if the selectivity factor is low because the probability of excluding several data items in one step is high and beneficial for performance. In the case of Sel=0 %, the vectorized branching scan

takes only 35 % of the branching scan. Its performance loss at higher selectivity factors is caused by the inefficient result extraction from the bit mask. Also, a predicated mask evaluation of the vectorized scan does not yield performance improvements and is thus not preferable to the branching version.

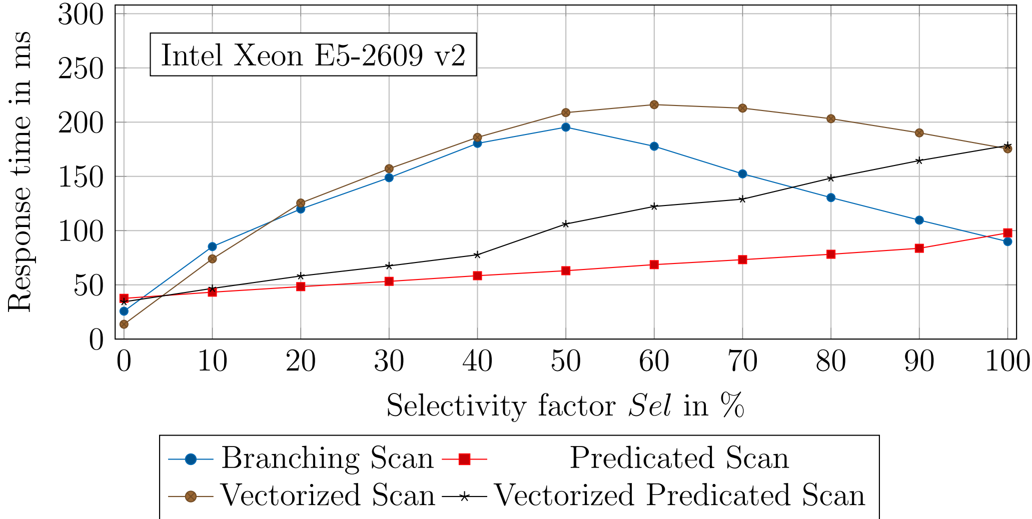

Figure 3.5: Response time of scan variants on 30 million data items

SIMD Acceleration in Database Operator Implementation

SIMD is applied to various operator implementations. Especially selections [RHVM15, BBS14, ZR02] and joins [KSC+09, ZR02, AKN12, BATÖ13] are accelerated with SIMD in several publications. Still in SSE, it is hard to accelerate hash table builds. However, due to the introduction of scatter and gather in AVX2, operations such as hash builds using radix sort become vectorizable as shown by Polychroniou et al. [PR13, PRR15]. Gurumurthy et al. investigate the applicability of AVX2 for different hashing techniques [GBP+18].

Additionally, SIMD is proposed for index structure acceleration [ZR02, KCS+10, ZHF14], for compression [WOMF13], and for database cracking [PPI+14].

3.1.5 Code Optimizations in Other Operators and Domains

Code optimizations are a big and evolving field of research. Hence over the years, there are more and more code optimizations introduced. In the following, we introduce code optimizations that have been introduced for other operators or in other domains:

AVX-512: With the introduction of AVX-512, a plethora of different intrinsics has been introduced. Especially intrinsics for masked execution allows for the *full computation* code optimization introduced by Răducanu et al. [RBZ13]. Also, AVX-512 allows to skip the mask computation of the SIMD-accelerated scan by an intrinsic that allows to compresses the result mask to a position list. Furthermore, in combination with *just-in-time compilation*, AVX-512 is able to efficiently accelerate multi-column scans [DKF+18].

Prefetching: The CPU starts loading cache lines not yet accessed by the program, expecting them to be accessed later due to the principle of locality [HP11]. This is called prefetching. Whenever the CPU detects a sequential access pattern of the program, its hardware prefetcher loads data in advance that is needed next. Prefetching can also be triggered by the hardware itself, or by using special instructions in software [Int16]. Software-controlled prefetching can accelerate pre-determined non-sequential access patterns (e.g., in a binary search). However, due to limited prefetching units and cache sizes, prefetching intrinsics should be cautiously used [SBS18].

Coalesced Access: With the rise of GPGPU computing, special optimizations for GPUs have been introduced. Since GPUs use light-weight threads that are executed in a SIMD fashion, they are sensitive to how tasks are performed. While parallel CPU implementations rely on sequential access where data is split into contiguous blocks that are processed on separate cores, GPUs use a coalesced access pattern. In a coalesced access pattern, adjacent data items are processed by adjacent threads leading to a rather interleaved execution [RHVM15].

3.2 Multi-Predicate Code Optimizations

To evaluate multiple predicates, we have to extend the above variants to evaluate not one, but several predicates in the `for`-loop. However, how to combine the predicates in an `if`-clause is an open question that is answered in this section, because we can use a conditional `AND` or a bitwise `AND`. Notably, these two variations only apply to the branching scan, because (1) the predicated omits every branch and, thus, will only use the bitwise variant and (2) the SIMD variants only support a bitwise `AND` as intrinsic.

3.2.1 Conditional `AND`

The conditional `AND` (e.g., P_1 && P_2) between predicates will start to evaluate the first predicate at first and only if it evaluates to true, the second predicate will be evaluated. This is also often called a short-circuit evaluation, because the computation of further predicates can be skipped, when the first predicate is already evaluated to false. Hence, it yields a speedup, if the first predicate is highly selective [Ros04]. However, each conditional `AND` will produce a conditional branch in the execution and may lead to heavy branch misprediction penalties.

Performance Impact of Conditional `AND`

In Figure 3.6, we show the performance of a scan using a conditional `AND` for evaluating different selectivities of two predicates. Although we evaluated the whole space of the selectivity combination of both predicates, only half of the space would be considered (the space where $\mathrm{P}_1 < \mathrm{P}_2$) because the query optimizer will choose the most selective feature as P_1.

Looking at the variant behavior depending on the selectivity of the first predicate P_1, it is visible that it causes the known branch-misprediction penalty if the selectivity

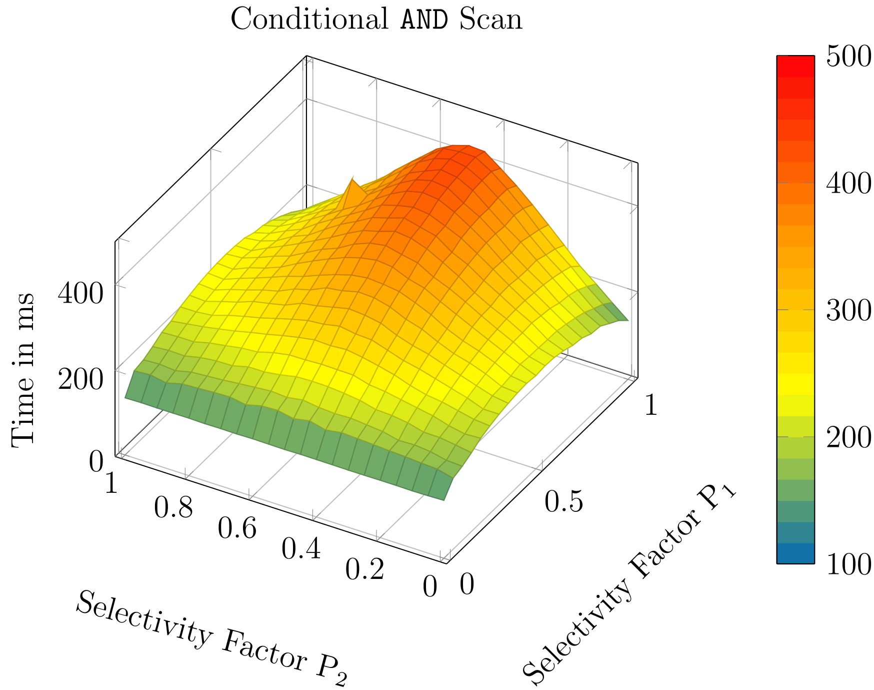

Figure 3.6: Response time of conditional AND scan for two predicates under varying selectivities

factor of the second predicate is high or low. In the other case, the penalty even adds up. For instance at $Sel_{P_1} = Sel_{P_2} = 0.5$, the conditional AND creates an overhead of 135 % compared to a scan that retrieves all data ($Sel_{P_1} = Sel_{P_2} = 1$). The peak of the diagram is located at $Sel_{P_1} = 0.9$ and $Sel_{P_2} = 0.5$, because it is the worst case: a filter on the first predicate does not give much benefit from early pruning and the final outcome depends on the unpredictable branch of the second predicate. In contrast, $Sel_{P_1} = Sel_{P_2} = 0.5$ at least reduces the branch misprediction penalty for those tuples where P_1 is not satisfied.

3.2.2 Bitwise AND

The bitwise AND (e.g., P_1 & P_2) inside an `if`-clause evaluates all predicates, forms the final result, and then executes the branch according to the result of the predicate evaluation. In this way, branch misprediction penalties are reduced (except the last one for the `if`-clause), but it misses the possibility to skip irrelevant predicates as the conditional AND can do.

Performance Impact of Bitwise AND

In Figure 3.7, we show the same experiment as for the conditional AND. At first, we can see that the impact of branch misprediction is reduced compared to the conditional AND. Especially points that are close to the plane with $Sel_{P_1} = Sel_{P_2}$ have a smaller run time compared to the conditional AND variant. Here, we can achieve speedups of up to 16 %. The peak of the diagram is at the same position as for the conditional AND. However, the gradient is much steeper and the peak is 10 % higher.

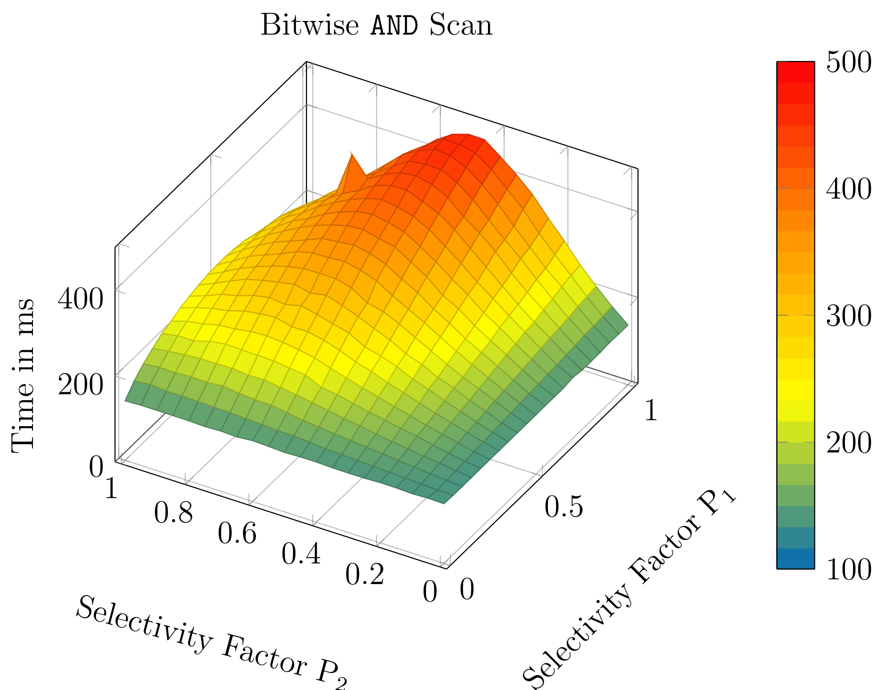

Figure 3.7: Response time of bitwise `AND` scan for two predicates under varying selectivities

Performance Impact of Number of Predicates on Bitwise `AND`

In Figure 3.8, we show the performance of a bitwise `AND` with varying selectivities of the first predicate `P1` and an increasing number of predicates (1-10 predicates)[2]. In this experiment, we set the selectivity of all the other predicates to 1, because we wanted to see the overall impact of an increasing number of predicates on the variant's performance. Especially for this variant, the selectivity of all other predicates plays no role since the bitwise variant checks the outcome of the predicates at last.

Overall, we can see that the response time increases linearly with an increasing amount of predicates. However, since the branching overhead for $Sel_{P_1} = 0.5$ compared to $Sel_{P_1} = 1$ is an almost constant value (in these experiments around 70ms), its impact for an increasing number of predicates diminishes for this variant. Hence, for an increasing number of predicates, this variant outperforms the conditional `AND` for highly similar selectivities of all predicates, since it only pays the branching overhead once.

3.3 Exploiting Code Optimizations in Database Management Systems

In the previous sections, we made the case for the parallel existence of different variants due to different device characteristics and data characteristics. However, there are several inherent problems caused:

[2] For a single predicate the code and performance resembles the one of the branching scan.

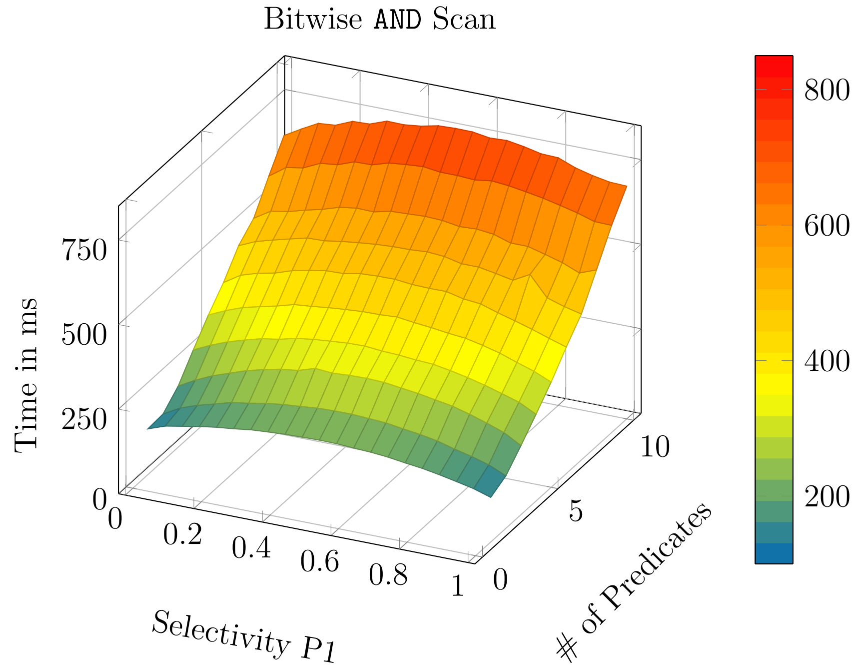

Figure 3.8: Response time of bitwise AND scan under different numbers of predicates

Variant Space Explosion: Due to the amount of available optimizations, the possibility to combine them, and the number of different operators in a DBMS (e.g., around 30 in MonetDB), there is a zoo of all variants to be maintained. Especially, the *implementation* of these variants cannot be done by hand, but needs a clear generation mechanism. The variant explosion opens up two sub-ordinate problems:

Unclear Impact of Data and Hardware Characteristics: With the diversity of hardware and their features as well as different use cases that imply different data characteristics (selectivity, data distribution, data size, and types), it is tedious to test *all* variants against the current workload. Hence, there needs to be a clever way to asses the costs of a variant for different workloads.

Unclear Impact of Code Optimization Combinations: Looking at single optimizations alone is not the key for best performing variants. Often, *a combination* of optimizations improves performance beyond the sum of the benefit of the single optimizations. However, an optimization can also harm performance of a specific variant (cf. branched loop-unrolled scan).

To tackle the aforementioned problems, we propose the *adaptive reprogramming* approach. This approach is tightly integrated with the query engine to be able to create several hardware-sensitive operators out of one abstract operator description while monitoring the variant performance. The overall structure of our approach is visualized in Figure 3.9, which is based on the hardware-sensitive and hardware-oblivious approach described in Section 2.3. In the following, we explain the components of the adaptive reprogramming approach.

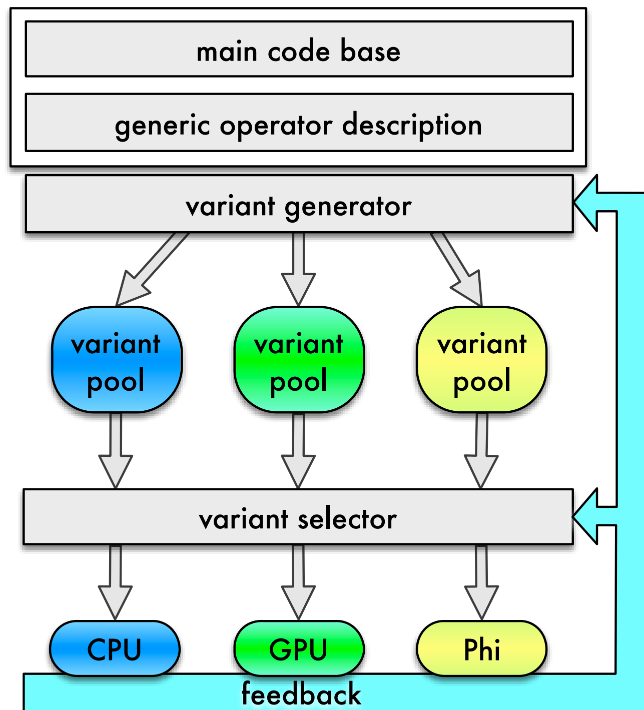

Figure 3.9: Sketch of our adaptive reprogramming approach for reaching hardware-sensitive database operations on heterogeneous hardware

3.3.1 Variant Generation

To tackle the problem of the *variant space explosion*, we propose to use a domain-specific language (DSL) to define an operator [BBHS14]. As a result, we add an abstraction level to the operator implementation, which helps us to flexibly adapt the resulting operator code. Given a generic operator description, we are able to apply different sets of code optimizations to produce different variants. For example, we could vectorize the tight loop in a selection and then unroll the vectorized code.

The resulting variants are specific to one processor and have variations in code depending on different possible workloads. These variants are grouped per available processor type in a dedicated *variant pool*. On execution, the variant selector chooses the optimal variant for the selected processor and workload.

3.3.2 Variant Selector & Feedback Loop

Since it is not an easy task to estimate how a given algorithm performs on the given hardware – especially for parallel algorithms [BATÖ13] – we argue to use an unsupervised learning algorithm instead of a static cost model. Especially new advances in deep learning gives rise to techniques that may be applied to the operator selection problem [CPP+18].

The selector learns the execution behavior of variants under given hardware properties and workload characteristics. In order to learn the execution behavior, we need a feedback loop, which informs the selector about the runtime of the chosen variant to refine the selector's learned cost model. Also, the feedback loop has to inform

the variant generator about the performance impact of used code optimizations in order to generate more efficient code. This procedure resembles the idea of adaptive query processing [DIR07, BLC+16], where the query plan is refined during run-time to achieve better performance.

3.3.3 Variant Management

In literature, there are numerous code optimizations proposed [BBS14, DMV+08, RBZ13] that improve the code for different use cases. Hence, if we assume n independent code optimizations, we can create 2^n variants. Consequently, the number of possible variants increases exponentially with an increasing number of code optimizations. This causes the variant pool as well as the learned cost model of the selector to grow enormously. As a consequence, we argue to limit the variant pool and to keep only promising variants in the pool. However, if the workload of the database system changes, better variants could be generated and included in the variant pool, while others are evicted.

3.3.4 Usage of Adaptive Reprogramming in Recent Database Management Systems

Adaptive reprogramming is a combination of already proposed concepts and can be used to implement and optimize hardware-sensitive database management systems. In this section, we outline parts of our approach that are implemented in current systems and also point at parts that need to be implemented in these systems. For this comparison, we reviewed available research prototypes that deal with hardware-sensitive operator implementations (especially those managing different operator variants) and finally come to the following systems (in chronological order): VectorWise, CoGaDB, LegoBase, Voodoo.

VectorWise

VectorWise and its proposal of micro adaptivity [RBZ13] in vectorized execution engines is the origin of variant management. In their work, Răducanu et al. define *flavors* as different variants and propose variant selection as solving a multi-armed bandit problem. However, their work does not cover an efficient variant generation approach. They simply use template expansion for generating different flavors, which is limited in its abstraction potential [KKRC14].

CoGaDB

CoGaDB [Bre14], as a GPU-accelerated system, inherently depends on hardware-sensitive operator implementations, because operators have to be efficiently executable across different devices. Also, its query execution engine HyPE [Bre13] is designed to choose an efficient implementation for different devices. In this sense, it implements *variant pools* by having different code for each device and a *variant selector* (HyPE) that adapts its cost models. However, CoGaDB does not feature a *variant generator* that can generate new hardware-sensitive relational operator implementations.

A recent advancement in CoGaDB is an impactful implementation of our adaptive reprogramming architecture. Hawk [BKF+18], which is the new hardware-tailored

code generator of CoGaDB, allows to manage variants of query pipelines efficiently. It has an abstract representation of a query pipeline, a so-called *pipeline program*, defining an own DSL. Furthermore, Hawk applies different code optimizations in a staged manner while exploring the operator space using a variant optimizer that combines our *variant selector* and *feedback loop*.

LegoBase

LegoBase [KKRC14] and the DBLAB/LB query compiler [SKP+16] use light-weight modular staging [Rom12] to stepwise lower and optimize a query plan. Both implement a DSL to allow for abstract operator descriptions and then use transformers to adapt code and data structures. Although they are able to generate code for different (co-)processors, they do not include hardware-sensitive code optimizations and a selection of different variants.

Voodoo

Voodoo [PMZM16] is an algebra and compiler for vector programs on hybrid CPU/GPU systems. Voodoo can be seen as a hardware-sensitive code generator for different variants of pipelines of vector operations. Thus, it combines a DSL for vector programs and a variant generator. However, variant selection and the selection of variants to be generated is not automatically decided and can be done by the engine of LegoBase for example.

Conclusion

Overall, we can see that there are many systems that have to cope with different implementation variants of the same operator or primitive [GBD+18]. Furthermore, some of the systems implement several components of adaptive reprogramming. In fact, CoGaDB's Hawk implements all components of adaptive reprogramming, but currently only focuses on compiled query pipelines and not whole relational operators or primitives. Hence, we argue for the usefulness and effectiveness of adaptive reprogramming.

3.4 Summary

In this chapter, we defined the term *code optimizations* and looked at the impact of different code optimizations for full-table scans that implement the selection operator. The result is the answer to research question RQ 1, since we looked at optimizations of a single predicate and also at variants for combined selection predicates providing guidelines when to use which variant. This led to a baseline that has to be beaten by more elaborate techniques for single as well as multi-predicate selections in the following chapters.

In addition, we presented arising challenges of variant management and introduced an approach to deal with these challenges. Our approach of adaptive reprogramming allows to generate, maintain and select different hardware-sensitive variants, which answers research question RQ 2. Furthermore, we reviewed several systems that were the basis of adaptive reprogramming and also outlined approaches that follow the same design principles, but lack some important components to tackle the defined challenges.

4. Elf as Multi-Column Selection Predicate Index

With analytical queries getting more and more complex, the number of evaluated selection predicates per query and table rises, too. For example, a typical TPC-H query involving several column predicates is `Q6`, whose `WHERE`-clause is visualized in Figure 4.1(a). We name such a collection of predicates on several columns in the `WHERE`-clause a *multi-column selection predicate*. Multi-column selection predicate evaluation is performed as early as possible in the query plan, because it shrinks the intermediate results to a more manageable size. This filtering has become even more important in main-memory database systems due to the limited amount of available RAM, which calls for condensed intermediate representations (cf. late materialization [AMDM07]).

In case all data sets are available in main memory (e.g., in a main-memory database system [BKM08, KN11, Pla09]), the selectivity threshold for using an index structure instead of an optimized full table scan is even smaller than for disk-based systems. This phenomenon is due to the missing I/O bottleneck between disk and main memory, giving full-table scans an advantage. In their study, Das et al. [DYZ+15] propose to use an index structure for very low selectivities only, such as values smaller than 2 %. Hence, most OLAP queries would never use an index structure to evaluate the selection predicates. To illustrate this, we visualize the selectivity of each selection predicate for the TPC-H Query `Q6` in Figure 4.1(b). All of its single predicate selectivities are above the threshold of 2 % and, thus, would prefer an accelerated full-table scan per predicate. However, an interesting fact neglected by this approach is that the accumulated selectivity of the multi-column selection predicate (1.72 % for `Q6`) is below the 2 % threshold. Hence, an index structure would be favored if it could exploit the relation between all selection predicates of the query. Consequently, when considering multi-column selection predicates, we achieve the selectivity required to use an index structure instead of an accelerated full-table scan.

In this chapter, we present the design of an index structure for order-preserving dictionary-compressed data or numeric data. The new index structure, called Elf, is

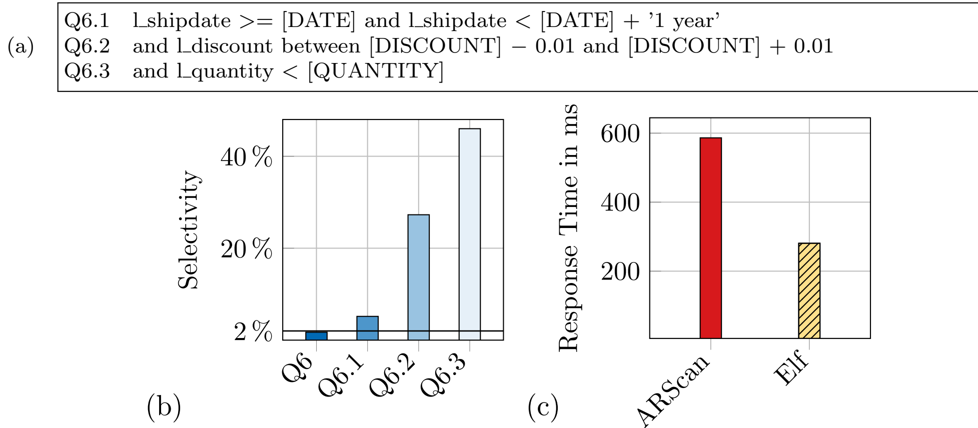

Figure 4.1: (a) WHERE-clause, (b) selectivity, and (c) response time of Elf and a scan generated by adaptive reprogramming (ARScan) on TPC-H query `Q6` and its predicates `Q6.1` - `Q6.3` on `Lineitem` table $s = 100$

optimized for executing multi-column selection predicates in main-memory systems. Elf is a novel combination of concepts and optimizations behind existing indexing approaches in concert with a memory layout explicitly designed to exploit the capabilities of modern CPUs. Due to its design, Elf can outperform full-table scans on multiple columns by several factors (cf. Figure 4.1(c)). To this end, we contribute to the goals of Level 2 of our thesis (i.e., eliminating the pain point of full-table scans – multi-column selection predicate evaluation) and answer the research question RQ 3.

In the following, we first explain the Elf's basic design and the underlying memory layout. Then, we introduce additional optimizations to counter deteriorations due to sparsely populated subspaces and present how to search in Elf. Finally, we determine a theoretical upper bound for its storage size and talk about our heuristic for the column order. The chapter ends with an extensive evaluation of Elf's selection predicate evaluation performance against highly-potent competitors.

In summary, we make the following contributions:

- Introduction of a main-memory index structure for multi-column selection predicates, called *Elf*
- Proposal of improvements for Elf's conceptual design to address deteriorations of its tree-based structure to reach peak performance in main-memory database systems
- Evaluation of Elf's selection performance against stand-alone indexes:
  - Full-table scans (generated using adaptive reprogramming)
  - State-of-the-art main-memory indexes (BitWeaving [LP13], Column Imprints [SK13])
  - Multi-dimensional index structures (BB-Tree [SSL18a, SSL19], sorted projections [ABH^{+}13])
- Evaluation of Elf integrated into MonetDB against MonetDB's highly optimized full-table scans

4.1 Conceptual Design of Elf

In the following, we explain the basic design with the help of the example data in Table 4.1. The data set shows four columns that we want to index and a tuple identifier (TID) that uniquely identifies each row (e.g., the row id in a column store).

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

Table 4.1: Running example data

In Figure 4.2, we depict the resulting Elf for the four indexed columns of the example data from Table 4.1. The Elf tree structure maps distinct values of one column to `DimensionList`s at a specific level in the tree. In the first column, there are two distinct values, 0 and 1. Thus, the first `DimensionList`, $L_{(1)}$, contains two entries and one pointer for each entry. The respective pointer points to the beginning of the respective `DimensionList` of the second column, $L_{(2)}$ and $L_{(3)}$. Note, as the first two points share the same value in the first column, we observe a prefix redundancy elimination. In the second column, we cannot eliminate any prefix redundancy, as all attribute combinations in this column are unique. As a result, the third column contains three `DimensionLists`: $L_{(4)}$, $L_{(5)}$, and $L_{(6)}$. In the final column, the structure of the entries changes. In an intermediate column, an entry consists of a value and a pointer. Now, the pointer is interpreted as a tuple identifier (TID).

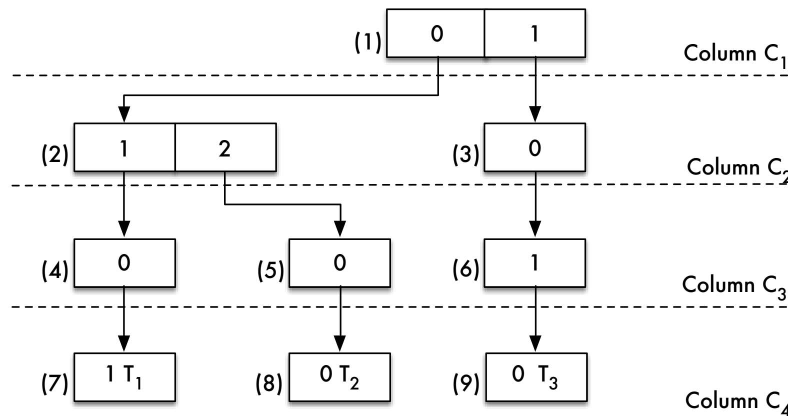

Figure 4.2: Elf tree structure using prefix-redundancy elimination

The conceptual Elf structure is designed from the idea of prefix-redundancy elimination [SDRK02] and the properties of multi-column selection predicates. To this end, it features the following properties to accelerate multi-column selection predicates on the conceptual level:

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

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

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

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

4.2 Improving Elf's Memory Layout

The straight-forward implementation of Elf is similar to data structures used in other tree-based index structures. However, this creates an OLTP-optimized version of the Elf, which we call InsertElf. To enhance OLAP query performance, we use an explicit memory layout, meaning that Elf is linearized into an array of integer values (a classical way to reduce read and memory effects by compromising update performance [AKM+16]). For simplicity of explanation, we assume that column values and pointers within Elf are 64-bit integer values. However, our approach is not restricted to this data type. Thus, we can also use 64 bits for pointers and 32 bits for values, which is the most common case.

4.2.1 Mapping `DimensionLists` to Arrays

To store the node entries – in the following named `DimensionElements` – of Elf, we use two 64-bit integers. Since we expect the largest performance impact for scanning these potentially long `DimensionLists`, our first design principle is *adjacency* of the `DimensionElements` of one `DimensionList`, which leads to a preorder traversal during linearization. To illustrate this, we depict the linearized Elf from Figure 4.2 in Figure 4.3. The first `DimensionList`, $L_{(1)}$, starts at position 0 and has two `DimensionElements`: $E_{(1)}$, with the value 0 and the pointer 04 (depicted with brackets around it), and $E_{(2)}$, with the value 1 and the pointer 16 (the negativity of the value 1 marks the end of the list and is explained in the next subsection). For explanatory reasons, we highlight `DimensionLists` with alternating colors.

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

Figure 4.3: Memory layout as an array of 64-bit integers

The pointers in the first list indicate that the `DimensionLists` in the second column, $L_{(2)}$ and $L_{(3)}$ (cf. Figure 4.2), start at offset 04 and 16, respectively. This mechanism works for any subsequent `DimensionList` analogously, except for those in the final column (C_4). In the final column, the second part of a `DimensionElement` is not a pointer within the Elf array, but a *TID*, which we encode as a 64-bit integer as well. The order of `DimensionLists` is defined to support a depth-first search with expected low hit rates within the `DimensionLists`. To this end, we first store a complete `DimensionList` and then recursively store the remaining lists starting at the first element. We repeat this procedure until we reach the last column. Note, in the last `DimensionList`, we mark the end of the list as well by setting the most significant bit, allowing to store non-unique multi-dimensional data sets as well.

4.2.2 Implicit Length Control of Arrays

The second design principle is size reduction. To this end, we store only values and pointers, but not the size of the `DimensionLists`. To indicate the end of such a list, we utilize the most significant bit (MSB) of the value. Thus, whenever we encounter a negative value[1], we know we reached the end of a list (e.g., the `DimensionElement` at offset 2). Note, in the final column, we also mark the end of the list by setting the most significant bit, allowing to store duplicates as well. Finally, we use direct pointers. This means that the pointers directly point to the offset in the array where the respective list in the next dimension starts without any use of additional structures (e.g., hash maps) or computation overhead (e.g., pointer arithmetics).

4.2.3 Alternative Memory Layouts

The memory layout of Elf plays a vital role in ensuring cache-friendly access patterns. In the linearization, we rely on two principles: (1) we store values and keys together, so that they are likely located at the same cache line, and (2) we store all entries of a node together, even though our search is a depth-first search (cf. Section 4.4). Arising from this, we implemented two different linearization strategies, which are (1) separate arrays for pointers and values, and (2) storing each path of Elf from root to a node adjacent to each other.

Separate Value and Pointer Arrays

An intuitive idea to accelerate processing is to store values and pointers in separate arrays, because if a value does not match, we also do not need the corresponding pointer in the caches. This layout also supports the usage of SIMD, as we could compare several values with the given bounds in one step. Unfortunately, this layout gives only comparable or slightly worse performance – even when used with SIMD. The problem is that we incur an additional memory reference if a value matches. Since multi-column selection predicates mostly use range predicates matching several values instead of only one, this impact even worsens.

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

Adjacent Path Storage

Our current implementation puts all elements of a `DimensionList` in consecutive storage. As a result, the scan on a `DimensionList` is accelerated, but following the path up to the leaves still causes a possible cache miss per `DimensionList`. The alternative to store whole paths (or sub-paths) adjacently causes one cache miss per walk over another path. Thus, only for highly selective workloads this linearization makes sense. However, in-depth evaluations show no significant benefit over our described linearization strategy.

4.3 Storage Optimizations for Elf

Considering the structure of Elf depicted in Figure 4.2, we can further optimize two conceptual inefficiencies. First, since the first list contains all possible values of the first column, this list can become very large, resulting in an unnecessary performance overhead. Second, the deeper we descend in Elf, the sparser the nodes get, which results in a linked-list-like structure in contrast to the preferred bushy tree structure. For both inefficiencies, we introduce as solutions: a hash map for the first column and `MonoList` for single-element lists.

4.3.1 Hash Map to Deal With the First `DimensionList`

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

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

Denseness. Due to the order preserving dictionary compression of the data, all integer values between 0 and the maximum value of that column exist[3].

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

As a result, the first `DimensionList` contains available integer values between $[0, \max]$, which are stored in an ordered manner. We depict the resulting Elf for the first column

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

[3]Our hash map also works in case the data is not dense. Then, we use a special pointer directly indicating that for this value there is no data, effectively being a null pointer.

with the value range [0, 7] in Figure 4.4 (upper part). The primary observation is that we can compute the position of the pointer to the next list by simply multiplying the value by 2 and incrementing the result. Consequently, we could also omit the values and only store the pointers, as shown in the lower part of the figure. Hence, we can directly use the values as keys to the pointers of the first column like in a hash map. In order to determine the start and end points of a query interval on the first column, we take the query interval and identify the respected pointers. This way, we remove the deterioration of the first `DimensionList` and furthermore, require only half of the storage space for it.

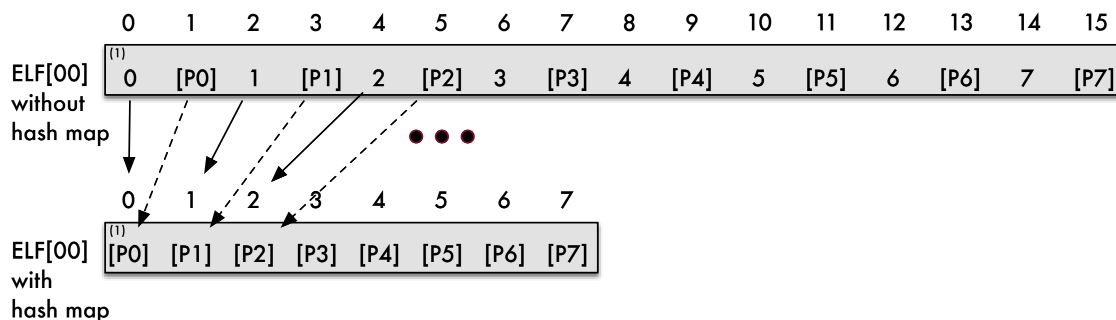

Figure 4.4: Hash-map property of the first `DimensionList`

4.3.2 `MonoList`: One-Element List Elimination

A main challenge of our data structure is that the lists get shorter the further the search descends into an Elf. Notably, there is a level where only one-element lists exist. That means, there are no more prefix redundancies that can be exploited. We display this issue for the TPC-H `Lineitem` table of scale factor 200 with all 16 attributes resulting in a 16-level Elf in Figure 4.5. The plot shows that at dimension 11 the prefix of each data item has become unique and each data item is now represented with its own path in the Elf. This leads to one linked list per data item, where each entry is a `DimensionList` with only one entry. The result of those one-element lists is that the remaining column values of each data item are scattered over the main memory and we need to additionally store pointers to these values, although branching is not necessary anymore. This phenomenon destroys the caching performance and unnecessarily increases the overall size of Elf. To overcome this deterioration, we introduce `MonoLists`. The basic idea of these `MonoLists` is that, if there is no prefix redundancy, the remaining column values of this tuple are stored adjacent to each other, similar to a row-store, to avoid jumps across the main memory.

In Figure 4.6, we depict the resulting Elf with `MonoLists` shown in gray and in Figure 4.7 the respective memory layout. Note that the `MonoList` can start at different dimensions and thus, it totally removes the deterioration of one-element lists. To indicate that there is a `MonoList` in the next column, we utilize the most significant bit of the pointer of the respective `DimensionElement` in the same way as we mark the end of a `DimensionList`. Thus, we depict such a pointer in the same way, by using a minus in front of the pointer in Figure 4.7. In the example, there are two `MonoLists` for C_3 and C_4 and a third one covering C_2, C_3, and C_4 for T_3.

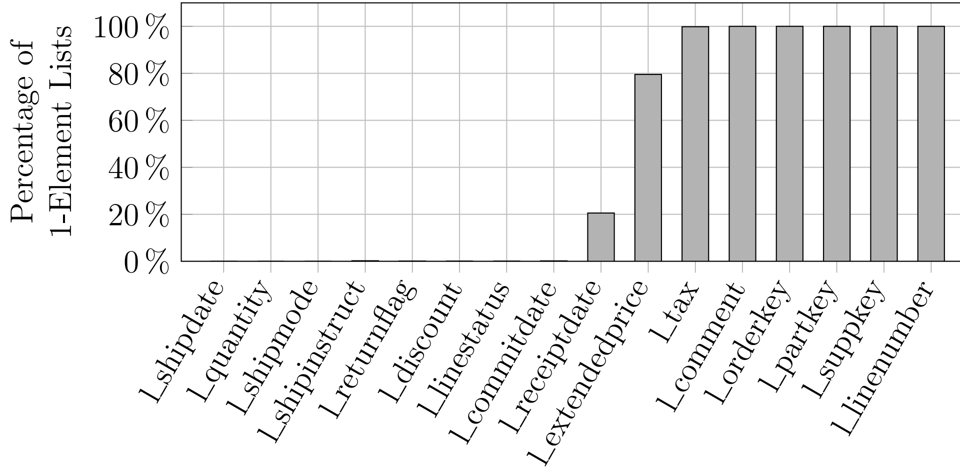

Figure 4.5: Percentage of 1-element lists per dimension for the TPC-H `Lineitem` table with scale factor 100

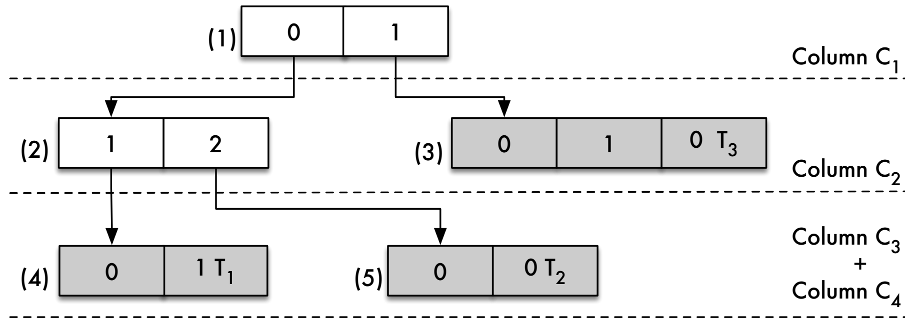

Figure 4.6: `MonoList` (visualized as gray `DimensionLists`) for optimized cache performance and storage utilization

In Figure 4.7, we depict the optimized storage layout of the Elf for the example data from Table 4.1 using the `MonoList` and the hash map optimization. In comparison to the initial layout in Figure 4.3, we observe a decrease in storage consumption and a better adjacency of values in later columns. Even for this toy example, we decrease storage consumption by almost 30 %. We give more insights into worst case storage consumption in the Section 4.3.3.

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

Figure 4.7: Optimized memory layout

4.3.3 Worst Case Storage Consumption

Storage consumption remains an important issue due to limited main-memory capacities and better cache utilization for smaller storage and index structures. We examine

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

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

Worst case means that for each point, we immediately start a `MonoList` after the first column, because with a prefix reduction, we achieve a better storage consumption[4]. The worst case leads to storage of *one* additional value per data item. The additional value is the *TID*, which would not be stored in the original row or column store representation as it is encoded implicitly based on the offset from the beginning of the array.

As a result, the maximum storage overhead per data item depends on the number of indexed columns n of the data set and decreases with an increasing amount of columns (cf. Table 4.2). It is computed as follows: $\text{overhead}(n) = (n + 1)/n$.

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

Table 4.2: Upper bound storage overhead

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

4.4 Searching in Elfs

After introducing the conceptual design and some optimizations of Elf, we now present our search algorithms to execute multi-column selection predicates in Elf. The search algorithm consists of several functions, each one depending on the node that is searched in (*hash map*, `DimensionList`, `MonoList`). Afterwards, we outline a simple heuristic that will be used in the evaluation to find a good column ordering.

4.4.1 Search Algorithm

In the following, we present the algorithm to evaluate a multi-column selection predicate within Elf, based on our definition from Section 2.1.2. The algorithm mainly consists of three functions: `SearchMCSP`, `SearchDimList` and `SearchML`.

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

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

Algorithm 1: Search multi-column selection predicate

`SearchMCSP` Algorithm

The first function `SearchMCSP`, depicted in Algorithm 1, is executed once in order to evaluate a multi-column selection predicate. It returns a list `L` of *TIDs* for each tuple in accordance with the multi-column selection predicate. Two arrays define the multi-column selection predicate containing the lower and upper boundaries of the (hyper) rectangle. Moreover, this function evaluates the first `DimensionList` exploiting its hash-map property (Line 3-5), in case C_1 is part of the multi-column selection predicate. Otherwise, in case of a wildcard for this column, the boundaries for evaluation are set to 0 and maximum of Column C_1. We have to check for each value whether the next `DimensionList` is a `MonoList`. Based on this check, we either call the function to evaluate a `MonoList` or a normal `DimensionList` (Line 10-14).

`SearchDimList` Algorithm

The second function `SearchDimList`, depicted in Algorithm 2, evaluates a predicate on a single `DimensionList`. The function has two more input parameters besides the lower and upper boundaries. This function also needs the start offset of the current `DimensionList` within the Elf (`startlist`) and the current column (`col`). The start offset directly marks the position of the first (and smallest) value in that `DimensionList` (Line 3). In case there is a predicate defined on this column, we start scanning the single values until we either reach the end of the list (Line 17) or we find a first value that is larger than the upper boundary of the query interval. Remember that the values are ordered, which allows us to abort the evaluation of that particular list. Whenever we find a value within the predicate boundaries,

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

Algorithm 2: Scan a `DimensionList` within an Elf

we propagate the evaluation of the multi-column selection predicate to the child `DimensionList` (Line 5-12). This results in a depth-first search, because we evaluate the child `DimensionList` before evaluating the next value by incrementing the `position` (Line 16). We decided on a depth-first search to make use of the program stack and, as the first column is handled by `SearchMCSP`, we benefit from the curse of dimensionality as the sparsity of the created spaces results in relatively low hit rates. Thus, on average, we are able to scan large parts of a `DimensionList` located in a small cache window without propagation to the next column. Consequently, the Elf search algorithm is optimized for low selectivity-rate workloads, as it is common for tree-based structures. For other query workloads, returning large parts of the data set, we would favor optimized full-table scans anyway.

`SearchML` Algorithm

The last function depicted in Algorithm 3 is `SearchML`, which evaluates a `MonoList`. Since `MonoList`s store the data of several columns in an array-like structure, we iterate through the array comparing each value with its corresponding lower and upper bound (cf. Line 4). If the value is not inside the query boundaries, the tuples of the `MonoList` do not satisfy the query. Hence, we can directly return.

If all values of the `MonoList` lie inside the query boundaries, the TID(s) of the `MonoList` have to be added to the result (cf. Line 8-12). Note that we use the same trick as for `DimensionList`s to mark the end of a TID list.

```
1  SearchML(lower, upper, startlist, col, L){
2      for (curCol ← col to NUM_COLs)do
3          if (lower[curCol] ≤ upper[curCol])then
4              if (!isIn(lower[curCol], upper[curCol], Elf[startlist + curCol −
                 col]))then
5                  return;// tuple does not qualify
6      end for
7      position ← NUM_COLs − col;
8      while (notEndOfList(Elf[position]))do
           // Handle duplicates
9          L ← L + Elf[position];
10         position ← position + 1;
11     end while
12     L ← L+unsetMSB(Elf[position]);
13 }
```

Algorithm 3: Scan a `MonoList` within an Elf

4.4.2 Selection of the Column Order

One important aspect of building an Elf is the order of columns, because it influences search time as well as storage consumption. To this end, we propose a simple heuristic that is used to determine a column order.

First results indicate that the pruning power is the most important parameter. In fact, it defines how many lists are traversed which has the highest performance impact (similar to number of cache lines read). Due to this fact, Elf's columns should be ordered according to their usage in queries, because the whole workload will benefit from it. Hence, the first column should be the most commonly used in the queries, e.g., a time dimension. The following columns are sorted in ascending order of their usage in queries. In the case that two columns are used in the same amount of queries, they are ordered by their selectivity. Due to this heuristic and the prefix reduction in the first columns, the data space is fast divided into sparse regions. Hence, we benefit from an early pruning of the search space.

4.5 Empirical Evaluation

We conduct several experiments to gain insights into the benefits and drawbacks of Elf. In total, we execute four different experiments ranging from micro-benchmarking Elf, to a comparison of stand-alone selection approaches against Elf, up to a comparison in a real main-memory database system (MonetDB):

Experiment 1: We start with evaluating the impact of our `MonoList` optimization. Here, we are interested how much storage space is saved when using our `MonoList`s and how it compares to storing the data in a tabular fashion.

Experiment 2: In Experiment 2, we evaluate the query performance of Elf on real-world selection predicates, such as those from the TPC-H benchmark queries. For this experiment, we select highly potent competitors from literature that did our evaluation in a similar fashion [LP13, SK13].

Experiment 3: In Experiment 3, we are interested in the scalability of our approach compared to the other competitors. To this end, we execute the TPC-H queries on different scale factors of data and show the distribution of speedups across these scale factors.

Experiment 4: Our last experiment, deals with the question whether our results also apply to a whole query engine that uses Elf. Hence, we integrate Elf into the well known main-memory database system MonetDB [BKM08] and execute our queries once with Elf and once without Elf in MonetDB.

To ensure a valid comparison, all approaches are implemented in C++ and tuned to an equal extent. The code of our evaluation is provided on the project website[5]. The result of a multi-column selection predicate evaluation is a position list complying to Definition 2.1. All experiments are single threaded to support an inter-operator parallelism concept, which we deem best for OLAP workloads, except for the multi-threaded BB-Tree variant (which we name BB-Tree-MT) using 32 threads. We perform our experiments on an Intel Xeon E5-2630 v3 (Haswell architecture) with max. 3.2 GHz clock frequency, 20 MB L3 cache, and 1 TB RAM. Our SIMD optimizations are implemented using AVX 2. In our evaluation, we present the response time for

[5] www.elf.ovgu.de

the selection predicates of each considered TPC-H query (scale factor $s = 100$). For statistical soundness, we repeated every measurement 100 times and present the median as robust averages.

4.5.1 Experiment 1: `MonoList` Storage Consumption

Although main-memory capacities increase rapidly, efficient memory utilization remains important, because it is shared between all data structures (e.g., hash tables) of the database system. In this micro-benchmark, we want to examine, first, whether our worst-case storage boundaries for Elf from Section 4.3.3 hold. This upper bound, however, is quite pessimistic. Thus, we are interested in empirical numbers of the storage overhead for the TPC-H `Lineitem` table ($s = 100$). Second, we are interested in how far this result is influenced by the usage of `MonoLists`, because they are an essential optimization for our Elf for multi-dimensional data.

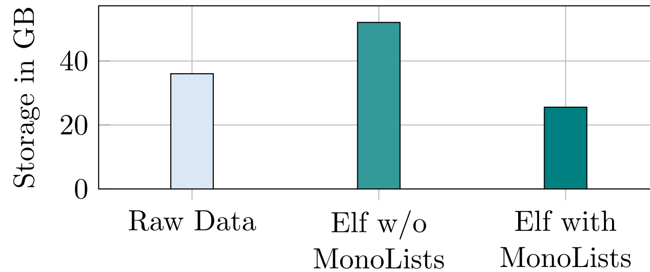

Table 4.3: Storage consumption for `Lineitem` table

In Table 4.3, we display the storage consumption of the raw data, an Elf without using `MonoLists` and an Elf with `MonoList` optimization. As visualized, the raw data consumes about 36 GB, while the Elf without `MonoList` optimization consumes 52.03 GB and the fully optimized Elf consumes 25.55 GB of RAM. This is a remarkable result, because the Elf is not only taking only 70 % of the raw data storage space, but we could also clearly improve a severe deterioration of the conceptual Elf. In fact, the optimized Elf consumes only half of the memory that the Elf without `MonoLists` consumes. This can be explained by the high number of `MonoLists` especially in deeper tree levels (cf. Figure 4.5). For instance, at level 9, we encounter around 114 million `MonoLists`, that save around 6 GB of pointers. Hence, the `MonoList` optimization is worth using for sparsely populated spaces, because it does not only save space, but also reduces the amount of cache lines that have to be fetched to visit the *TIDs*. Notably, Elf stores the whole data set, which means that we do not need to store the data additionally. Thus, we can even save space when using Elf as a storage structure (one of the scenarios for our next evaluation), because all information is directly available within the Elf.

4.5.2 Experiment 2: TPC-H Predicates and Data

In the following, we conduct an experiment using selection predicates from queries of the TPC-H benchmark [Tra14], which our competitors performed in a similar

fashion [LP13, SK13][6]. As competitors, we rely on accelerated full-table scans and multi-dimensional index structures. As accelerated full-table scans, we chose BitWeaving [LP13], as a technique working with a compressed data representation, Column Imprints [SK13], a filter power with a bloom filter, and scans that are generated using our adaptive reprogramming approach. As multi-dimensional index structures, we use Sorted Projections, a multi-dimensional sorting of the data being very similar to the Elf, and the BB-Tree, a kd-Tree with a fixed fanout and depth with underlying bubble buckets that are sequentially scanned. For a detailed description of the competitors, we refer to Section 7.1.

Elf Use Cases

This experiment answers the questions in how far our index structure Elf can compete against hardware-sensitive scans that are compiled according to the adaptive reprogramming approach and also against multi-dimensional competitors for real-world queries of the TPC-H benchmark. For the comparison, we create Elfs for three different use cases, which are called Elf, Elf_{red}, and Elf_{min}.

Elf: In the first use case, we build Elfs over the full tables. Essentially, it represents a scenario where Elf is used as a storage structure (i.e., storing the whole table), which has the advantage that we do not need to store data in a redundant storage format like row-wise or columnar storage and we even save storage consumption (cf. Experiment 1).

Elf_{red}: In the second use case, we built an Elf over the reduced set of columns that are necessary for the whole query workload on the table. As a consequence, it represents the scenario that we want to reuse one Elf for several queries and build the Elf over the full set of needed columns in the queries. This scenario needs all original data stored redundantly in their raw table format.

Elf_{min}: In the third use case, Elfs are built over the minimal set of columns *per query*. In fact, it is the best case for query performance, but here we store columns redundantly in different Elfs and also needs the underlying tables to be stored redundantly.

Query Selection

We select queries having a multi-column selection predicate and additional ones having a mono-column selection predicate, as summarized in Table 4.4. Notably, the last two columns states where the columns with a predicate are located within the full or reduced Elf and the minimal Elf, respectively. The first column number is 0 to emphasize that we can exploit the hash-map property for this column. The column $\text{Col}_{\text{Elf}_{min}}$ is also important for our multi-dimensional competitors (BB-Tree and Sorted Projection), because they will be built on the same column combinations.

[6] Other benchmarks, e.g., Starschema [OOC09] or TPC-DS [Tra15], could be used in a similar fashion. However, due to the common use of the TPC-H benchmark, we restrict our evaluation to this one.

Hence, both multi-dimensional index structures have the same potential as Elf_{min} by only indexing the minimal amount of columns needed for a specific query.

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

Since the accelerated full-table scans are sensitive to the number of queried columns, we also include several multi-column selection predicate queries on different tables. For `Q19`, we have two multi-column selection predicates on two different tables. The first is defined on the `Lineitem` table (as indicated by the *L* prefix) and the second is defined on the `Part` table. We refer to them as `LQ19` and `PQ19`, respectively. The column order for executing `LQ19` starts from the second column in the full and reduced Elf. Thus, we cannot exploit the hash-map property here and we are interested to see the impact of it.

Query `Q6` works on the `Lineitem` table and the predicates are defined on the first two columns and the 6th column. Please mind that in the minimal column order, the position of l_quantity and l_discount is swapped. This is due to the fact that in the column order of the full and reduced Elfs, the l_quantity is more often used in queries and, thus, has a higher overall impact giving it a higher rank. In contrast for query `Q6` alone, l_discount has a better selectivity than l_quantity (i.e., $Sel_{l_discount} < Sel_{l_quantity}$) and, hence, should be at a higher position in the column order.

Furthermore, `Q17` addresses the `Part` table and the predicate is defined on the first and second column with a rather low selectivity factor. Hence, we expect good results for all of these queries using Elf, because only a fraction of data is retrieved. Moreover, we expect to verify the superiority of state-of-the-art approaches for theses queries regarding the baseline of a full-table scan.

While the last three queries contain real multi-column selection predicates, Query `Q1` only contains a single column selection predicate. Although our work mainly focuses on accelerating multi-column selection predicates, we are still interested in the performance of our Elf in comparison to the competitors. However, we do not expect to outperform the state-of-the-art approaches for `Q1` as it represents a query that our approach is not designed for.

In the following, we depict the values for the selection predicates of the TPC-H benchmark with its order-preserving dictionary-compressed data. We generate 100 random predicates by varying selection predicate parameters according to the TPC-H specification and compute the median response time to assure robust measurements.

	example *Sel* in %	predicate columns	Col_{Elf}/ $\text{Col}_{\text{Elf}_{red}}$	$\text{Col}_{\text{Elf}_{min}}$
Q1	98.0	l_shipdate	0	0
Q10	24.68	l_returnflag	4	0
Q14	1.3	l_shipdate	0	0
Q6	1.72	l_shipdate, l_quantity, l_discount	{0,1,5}	{0,2,1}
LQ19	1.4	l_quantity, l_shipmode, l_shipinst.	{1,2,3}	{2,0,1}
Q17	0.099	p_brand, p_container	{0,1}	{1,0}
PQ19	0.083	p_brand, p_container, p_size	{0,1,2}	{0,1,2}

Table 4.4: Query details for mono and multi-column selections

4.5.2.1 Mono-Column Selection Predicate Queries

In this section, we discuss the response times for the for mono-column selection predicates Elf and its competitors. We start by comparing our Elf scenarios against accelerated full-table scans including BitWeaving, Column Imprints and a scan generated for each query using our adaptive reprogramming approach. Afterwards, we discuss our result for Elf's performance in comparison with the multi-dimensional competitors BB-Tree and Sorted Projection.

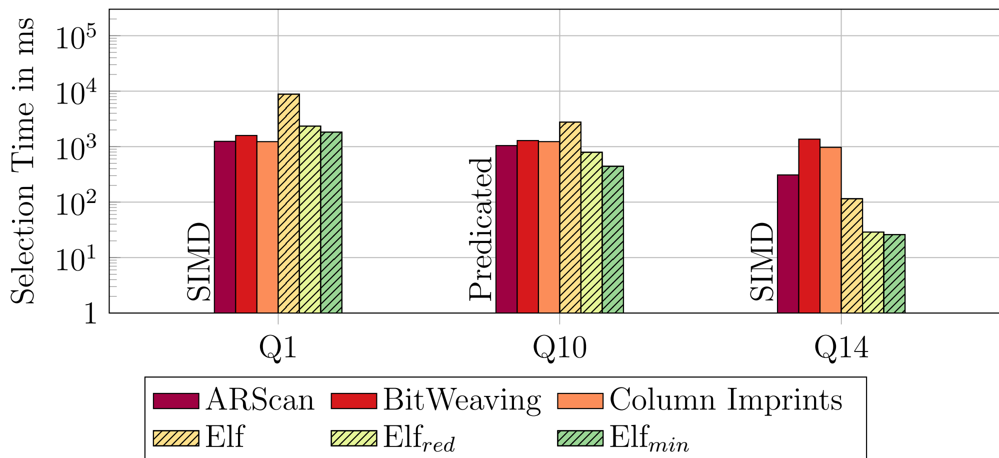

Figure 4.8: Query response times of Elf and accelerated full-table scans for mono-column TPC-H queries ($s = 100$)

Elf vs. Accelerated Full-Table Scans

In Figure 4.8, we depict the results for the mono-column selection predicates in a logarithmic plot. For query `Q1` and `Q10`, all accelerated full-table scans perform similar, however for `Q14`, adaptive reprogramming generates a SIMD scan that outperforms the other accelerated full-table scans by a factor of 3-4. The SIMD scan performs best for `Q1` and `Q14`, while a predicated version is best for `Q10` due to its medium selectivity.

Considering the performance of Elf, we observe high differences regarding the three queries in comparison to the competitors. For `Q1` returning 98 % of the tuples of the `Lineitem` table, Elf is clearly outperformed by all accelerated full-table scans.

Even Elf_{red} and Elf_{min} are slower than the accelerated full-table scans. By contrast, for `Q10`, where the selection column is the fifth column, using the Elf_{red} results in a better response time than the accelerated full-table scans. However, the Elf containing all columns is by a factor of 2.7 slower than the predicated scan. Reasons for this behavior are the high selectivity of `Q1`, the moderate selectivity of `Q10` and the fact that the selection predicate in `Q10` is at the fifth dimension. This forces Elf to follow a majority of paths. Therefore, we cannot and do not intend to compete with optimized full-table scans in this scenario.

Considering Q14, all Elf variants outperform the accelerated full-table scans. In fact, we reach a performance improvement by a factor of 2.7, 10, and 12 against the SIMD scan for this highly selective query when using Elf, Elf_{red}, and Elf_{min} respectively. From our point of view, this is a remarkable result, because our approach is designed and optimized for multi-column selection predicates. However, in Query `Q14`, we benefit from the hash-map property, a low selectivity factor, and the fact that the selection column is at the first instead of the fifth level, as in `Q10`.

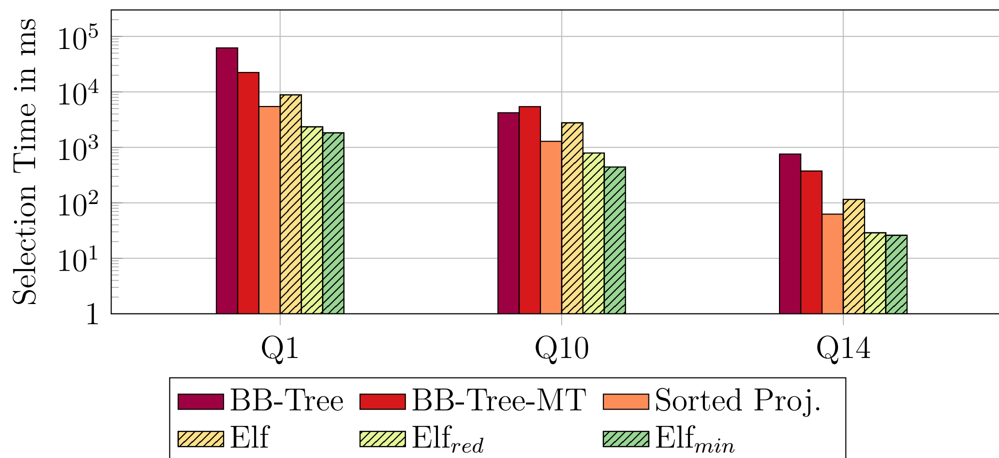

Figure 4.9: Query response times of Elf and multi-dimensional index structures for mono-column TPC-H queries ($s = 100$)

Elf vs. Multi-Dimensional Index Structures

In Figure 4.9, we depict the results for the mono-column selection predicates for the multi-dimensional competitors in comparison to Elf. Notably, we build the BB-Tree (including its multi-threaded version BB-Tree-MT using 32 threads) and Sorted Projection over the minimal set of columns with the same column order as Elf_{red}.

In general, the BB-Tree is outperformed by Sorted Projection and the Elf variants for all mono-column selection predicates. Interestingly, the multi-threaded version using 32 threads has an improvement of a factor of 2-3, while it performs even 30 % worse for `Q10`.

For all mono-column selection predicate queries, Sorted Projection performs better than Elf but worse than Elf_{red} and Elf_{min}. The benefit of Elf is that the prefix redundancy elimination allows to touch less memory locations than Sorted Projection, but skipping over the cold data diminishes this benefit. Thus, only the smaller Elfs constantly outperform Sorted Projection by a factor of 2 (Elf_{red}) and 2.5 (Elf_{min}).

4.5.2.2 Multi-Column Selection Predicate Queries

Since Elf has shown remarkable performance benefits even for some mono-column selection predicates, we assume an even better behavior for multi-column selection predicates. In the following, we again first compare Elf with the accelerated full-table scans and afterwards with its multi-dimensional competitors.

Elf vs. Accelerated Full-Table Scans

For the adaptive reprogramming approach, all multi-column selection predicate queries work best with a SIMD scan using bitwise `AND` for the predicates (cf. Figure 4.10). Overall, SIMD benefits here from reusing the comparator elements and also combining intermediate results in SIMD. This has already shown beneficial for compiled aggregation pipelines [BMS17] and also applies to selections. The superiority of the bitwise `AND` is due to our SIMD acceleration whose branching `AND` is only helpful in case the first predicate is highly selective. However, this is not the case for these predicates, as only their combination is highly selective. Hence, in this case, a bitwise `AND` is the best option.

In contrast to mono-column selection predicates, we observe Elf's superiority for all multi-column selection predicate queries. In particular, all Elf variants deliver the fastest response times for every query compared to the full-table scans. Moreover, we observe a stable performance increase between a factor of 3 and 4 when using Elf_{red} as compared to a full Elf. An in-depth analysis reveals that this correlates to the difference in size of both variants.

However, the performance gain of our approach over the accelerated full-table scans varies widely. For the `Lineitem` selection predicates of Query `Q19(LQ19)` and `Q6`, we observe the *smallest* performance gain compared to the fastest accelerated full-table scan (adaptive reprogramming), which is between a factor of 2 and 4. By contrast, the largest performance gain is measured for the queries with the smallest result sizes: `Q17` and `PQ19` (cf. Table 4.4). It is in the order of almost *two orders of magnitude* compared to the adaptive reprogramming scan.

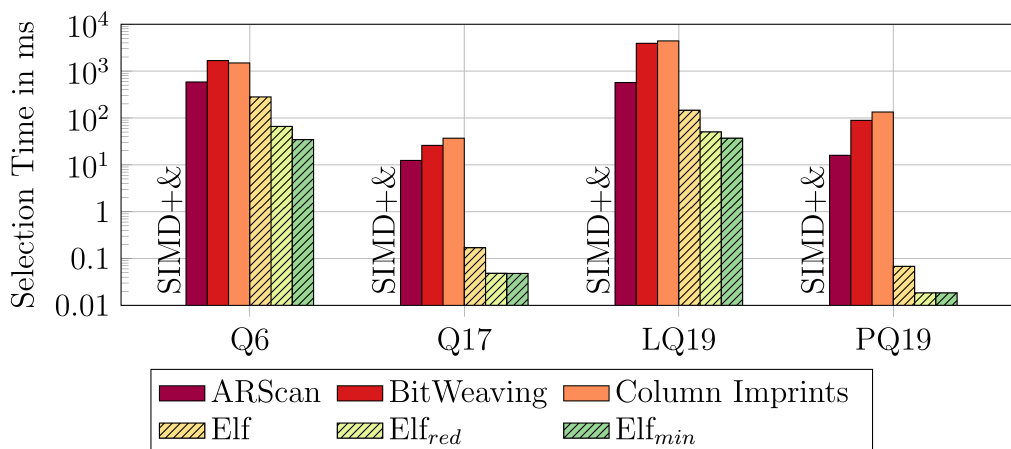

Figure 4.10: Query response times of Elf and accelerated full-table scans for multi-column TPC-H queries ($s = 100$)

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

According to our results, the performance gain also depends on the column order. This is especially observable for `Q6` and `LQ19`, which have a similar selectivity, but the selection predicates are defined on different columns. In fact, to evaluate `Q6`, we have to traverse the Elf until the sixth column (the last column with a predicate) in order to exclude last parts of the tree, while we pruned the remaining Elf to the final set of paths after the fourth column for `LQ19`. This explains the different speedups of both queries. Interestingly, the response times of `LQ19` (multi-column) and `Q14` (mono-column), whose selectivities are similar, are comparable, indicating the consistency and stability of our approach and cost model.

Elf vs. Multi-Dimensional Index Structures

When comparing the performance of the multi-dimensional competitors with the accelerated full-table scans, we can see that especially for query `Q17` and `PQ19` the BB-Tree and Sorted Projection perform by a magnitude better than BitWeaving, Column Imprint and SIMD scans. However, for the other two queries, the SIMD scan generated by adaptive reprogramming can even outperform the BB-Tree and Sorted Projection. Notably, the multi-threaded version of the BB-Tree does only give a performance improvement of a factor of 2-3 for `Q6` and `LQ19` and only 30 - 70 % for `Q17` and `PQ19`. The small benefit for the part table queries comes probably from the small size of the table. Hence, we cannot recommend to use the multi-threaded version for this little amount of data. Comparing Sorted Projection and BB-Tree, for the queries on the `Lineitem` table, Sorted Projection can outperform the single-threaded BB-Tree but Sorted Projection is outperformed by the multi-threaded version. For the queries on the `Part` table, both BB-Tree versions outperform Sorted Projection.

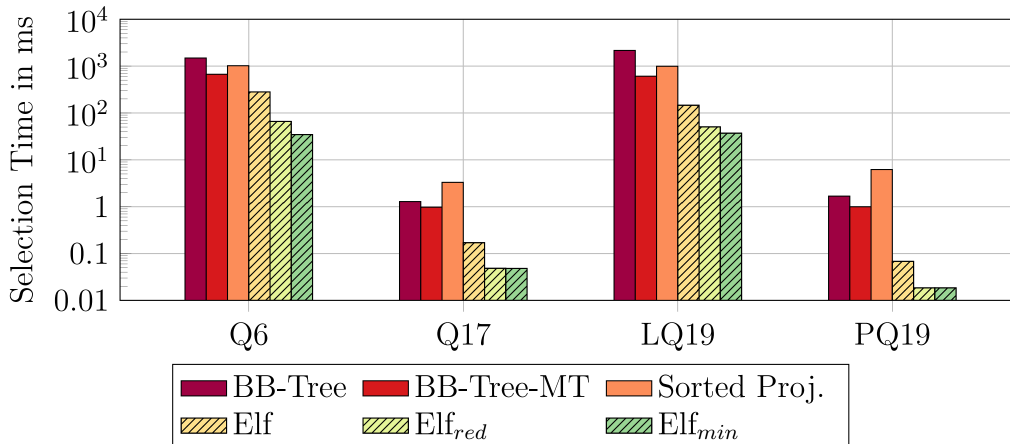

Figure 4.11: Query response times of Elf and multi-dimensional index structures for multi-column TPC-H queries ($s = 100$)

As a side note, we expected BB-Tree to perform by an order of magnitude better compared to its competitors (especially accelerated full-table scans). While its former evaluation setup (cf. Sprenger et al. [SSL19]) included around the same dimensionality, it however had a much higher cardinality of different column values. As stated in their paper, the BB-Tree has some deficiencies with low cardinality data. Especially for the sampling-based column ordering, low cardinality columns represent a threat. Hence, there is a need for further research to optimize the BB-Tree for our workload of TPC-H data and queries.

Similar to the results against accelerated full-table scans, Elf can outperform its multi-dimensional competitors by at least a factor of 3. In the best case, the full Elf gives a performance benefit by a factor of 14 (`PQ19`) compared to the multi-threaded BB-Tree. Considering the minimal Elfs, the performance improvement for `PQ19` adds up to a factor of 55. Notably, this is a remarkable result as Elf does outperform its multi-dimensional competitors even with a higher factor than for the mono-column selection predicate queries.

4.5.3 Experiment 3: Selection Time Scaling

In this experiment, we investigate how the selection time of Elf scales. Our hypothesis is that Elf scales with a smaller linear factor, e.g., in case one doubles the amount of data, the selection time increase is less than factor two. The rationale is that, when the data size increases while keeping column cardinalities the same, the data space is more densely populated. Therefore, we observe more beneficial prefix redundancy eliminations in Elf. This would be a valuable property of Elf, because full-table scans (including optimized ones), by concept, scale with a factor of 1. In addition, other tree-based approaches, such as the BB-Tree, probably face issues reaching a linear scaling [SGS+13].

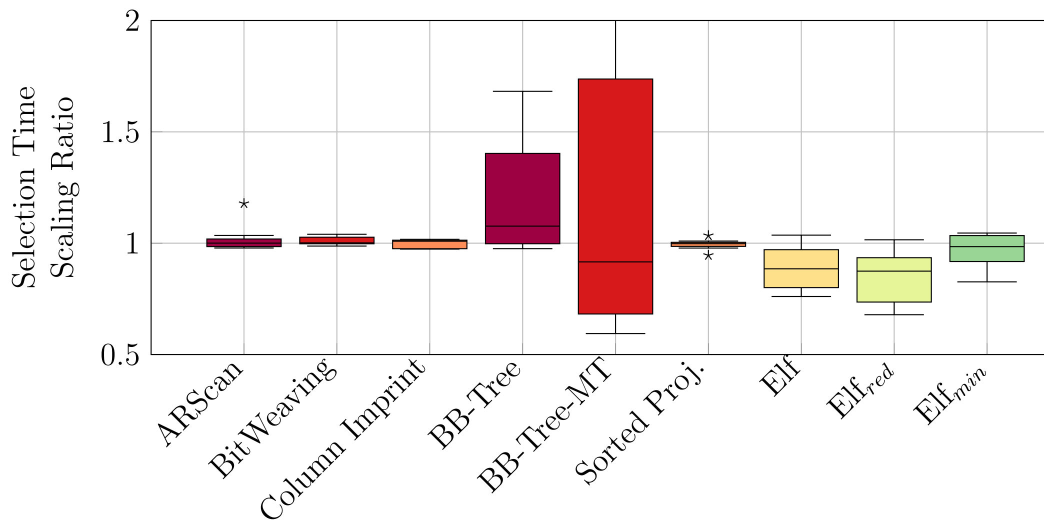

Figure 4.12: Selection time scaling ratios for all approaches

To investigate the validity of this hypothesis, we conduct the following experiment: We select two pairs of TPC-H scaling factors $(s_{\text{small}}, s_{\text{large}})$ such that $2 \times s_{\text{small}} = s_{\text{large}}$ holds. That is, the data size is doubled. In particular, we use $(50, 100)$ and $(100, 200)$. For each pair, we determine the selection time t_s^q for both scaling factors and all three mono-column selection predicates as well as the four multi-column selection predicate queries having seven queries in total: q with $1 \leq q \leq 7$ (cf. Section 4.5.2). Then, we compute the *selection time scaling ratio* per query as division of $t^q_{s_{\text{large}}}$ and $t^q_{s_{\text{small}}}$ normalizing by the data size increase, i.e., factor 2. As a result, a selection time scaling ratio of 1.0 indicates that the approach scales linearly for this query and scaling factor pair. In turn, a value observably smaller than 1.0 suggests a lower scaling. To confirm our hypothesis, we require that using Elf, on average considering all 14 measurements, results in a selection time scaling ratio below 1.0.

In Figure 4.12, we visualize the results for each approach. We observe that all full-table scans (i.e., ARScan, BitWeaving. and Column Imprint) scale linearly as expected. The overall result pattern is another indicator stating the validity of our experimental results in general. This is because all (optimized) full-table scans scale in a linear way with almost no deviation. Considering the BB-Tree, the serial variant performs worse when increasing the data size. This is probably due to an insufficient splitting due to the small cardinality columns and hence, overfull bubble buckets. The multi-threaded version shows a wide range of scaling factors. This is due to the difference in our queries. In fact, queries with a small selectivity lead to a small number of bubble buckets that are searched in parallel. Hence, parallelism does not pay out in this case.

Interestingly, we also observe, on average, a linear scaling for Sorted Projections being conceptually the most similar approach to Elf. However, the scaling of Elf is different. First, with 1.012 there is only one measurement where the selection time scaling ratio is slightly larger than 1.0. On average the selection time scaling ratio is 0.899 with $\sigma = 0.091$, i.e., the scaling ratio is observably smaller than 1.0. To this end, we confirm that Elf scales with a lower linear scaling factor than 1. Looking at the scaling factor of Elf_{red}, we see that the median is slightly lower than for the full Elf. However, scaling factors also have a higher deviation of $\sigma = 0.11$. The difference between Elf and Elf_{red} can be explained by two facts. First, the difference in depth (i.e., indexed columns) between both Elfs varies drastically and, hence, the possibility of exploiting a prefix-redundancy increases (cf. Section 4.3.3). Our experiments show that, although we double the number of tuples, Elf's storage consumption is only increased by a factor of 1.91. This is also visible for Elf_{min}, where the number of inserted tuples has the highest impact on performance scaling. However, due to the reduced set of indexed columns, Elf_{min} exploits less prefix redundancies compared to the length of paths. For example, the minimal Elfs for the mono-column selection predicates have a similar size and structure as the corresponding Sorted Projections (which scale by a factor of 1). Hence, Elf_{min} also scales with a factor closer to 1.

4.5.4 Experiment 4: TPC-H Predicates in MonetDB

In this experiment, we want to find out whether a full-fledged system with hardware-sensitive full-table scans is able to outperform Elf. For this experiment, we integrated Elf into MonetDB and ran the same queries once with the built-in full-table scans

and once with the built Elfs for the selections. In the latter case, MonetDB maps queries to Elf traversals and Elf results are expressed as a BAT[7]. A more detailed description of the integration into MonetDB and the impact on query execution and intermediate result combination is given in Chapter 5.

Experimental Setup

For this experiment, we load the dictionary-compressed TPC-H data (s=100) into MonetDB[8]. Our queries follow the TPC-H specification in the sense that they use the same predicates as their original queries. However, we only use one table per query (thus excluding joins) and our result is not the materialized table but the count of qualifying tuples. As a consequence, our results are free of intermediate result combination and interference with other operators. We also checked the final projection of the resulting count value and both MonetDBs scan queries and our queries with integrated Elfs incur comparable overhead for creating the final result. The impact of Elf on the full-fledged query engine is examined in Chapter 5.

In the following, we first discuss the results for all mono-column selection predicates and afterwards for all multi-column selection predicates that we already used in Experiment 2 (cf. Section 4.5.2). Again, we create Elfs on the full table (called Elf), Elfs for the minimal amount of columns for the table's whole query workload (called Elf_{red}) and a minimal Elf for each query (called Elf_{min}). Column orders are the same as in Experiment 2 (cf. Table 4.4).

4.5.4.1 Mono-Column Selection Predicates

In Figure 4.13, we show the performance of the three mono-column selection predicates (`Q1,Q10,Q14`) comparing the response time of full-table scans in MonetDB with the response times of different Elfs integrated in MonetDB. In general, MonetDB's scans perform similar for all mono-column selection predicates. The difference between the response times of the queries is caused the by costly result materialization which is the same for both, Elf and scans.

Overall, the scans outperform Elfs that are built over the whole `Lineitem` table if the selectivity factor is moderately or high (i.e., Sel_{Q1}=98 % & Sel_{Q10}=24.68 %). In contrast, for queries with a low selectivity factor (i.e., Sel_{Q14}=1.3 %), even the full Elf outperforms MonetDB's scans. The reason is the combination of good pruning capabilities with prefix-redundancy elimination. Hence, there is only a reduced set of nodes in the full Elf touched, which is even less costly than scanning a single column.

When comparing the scans with the reduced and minimal Elfs, we see that all Elfs outperform the scans by a factor between 2 and 5, which is in line with our previous experiments. Interestingly, reducing the number of indexed columns in Elf gives a big benefit (by a factor of 5-11), but cutting the indexed columns down to a minimum adds only a marginal benefit on top (around 10 %).

The results in MonetDB are in accordance with Experiment 3, which shows the validity of our experiments. MonetDB's scans perform similar to accelerated full-table

[7] BAT: Binary Association Table – MonetDBs internal column and intermediate data structure.
[8] We integrated Elf into MonetDB 5 server of version 11.8.0. with 64 bit, 128 bit integer.

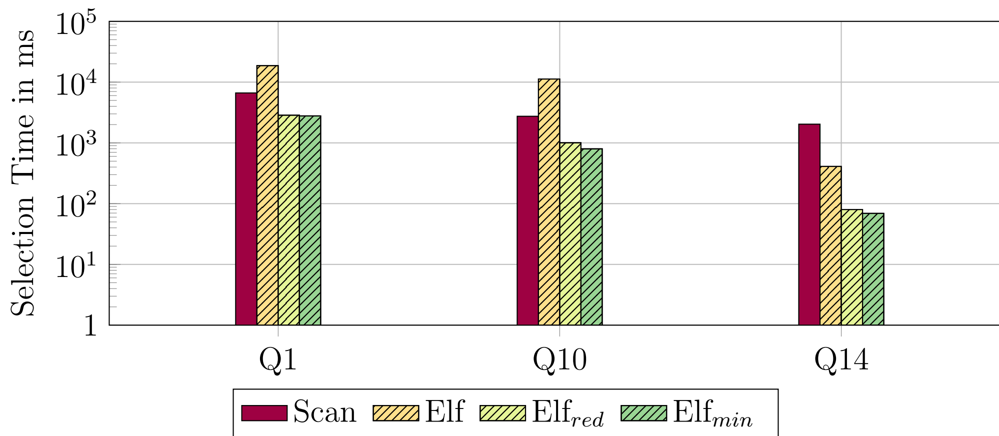

Figure 4.13: Query response times for mono-column TPC-H queries ($s = 100$) in MonetDB

scans and the integrated Elf performs similar to the stand-alone implementation. The difference is due to the interoperability between Elf and MonetDB's query engine. For example, our intermediate result, the list of TIDs, is transformed into a BAT, which MonetDB uses for further processing.

4.5.4.2 Multi-Column Selection Predicates

In this experiment, we want to see how well MonetDB's scans execute multi-column selection predicates. MonetDB uses the intermediate results (a BAT of matching entries) of subsequent scans in two ways. On the one hand, it is used as an upper bound to estimate the size of the result vector of the next scan. Obviously, it is never bigger than the result of the preceding scan. On the other hand, the intermediate result is used to only touch and evaluate results that qualified the preceding scan. With this property, MonetDB's scans are superior to BitWeaving whose interface is currently focusing on independent full-table scans and result merging using `OR` and `AND` operations on bit maps.

Nevertheless, MonetDB's scans cannot overcome the overhead of memory access when compared to a multi-dimensional index structure such as Elf. For all queries, the integrated full Elf reaches performance improvements by a factor of 3 up to a factor above 300. Elf_{min} adds up to this with a maximum performance improvement by a factor above 1500.

4.5.5 Result Summary

Based on the results of all our experiments, we have empirically shown the superiority of our approach compared to several strong competitors.

Our `MonoList` optimization reduces the storage overhead by 50 % compared to an Elf without `MonoList` and by 30 % compared to the raw data. In our performance evaluation on queries of the TPC-H benchmark, we first compared the full Elf, Elf_{red} indexing all columns needed for the set of queries, and Elf_{min} indexing only the necessary columns for a specific query against state-of-the-art accelerated full-table scans and a scan generated using our adaptive reprogramming approach.

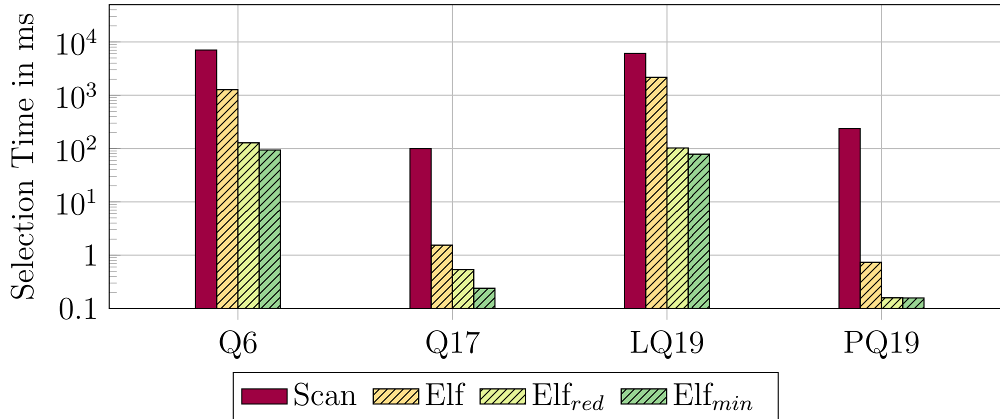

Figure 4.14: Query response times for multi-column TPC-H queries ($s = 100$) in MonetDB

Overall, adaptive reprogramming improves selection times against state-of-the-art scans by several factors. Furthermore, all Elf variants show remarkable performance improvements across a wide range of queries. Especially for highly selective queries, the Elf variants outperform accelerated full-table scans up to more than an order of magnitude. Only for queries with a single predicate with a medium selectivity, the Elf variants are outperformed by accelerated full-table scans.

In our performance evaluation, we also compare Elf against Sorted Projections and BB-Trees. Although both approaches are a reasonable competitor built on the minimal set of columns for each query, they cannot reach the low response times of Elf. Even the full Elf outperforms them on all evaluated multi-column selection predicates. Only Sorted Projections outperform Elf for single predicate queries with a medium selectivity. However, both Elf$_{red}$ and Elf$_{min}$ outperform all its multi-dimensional competitors by at least a factor of 1.6 up to a factor of 53 for $s = 100$ and our tested TPC-H queries.

In our third experiment, we invested the impact of the data size on our index structures and competitors. While the accelerated full-table scans double their runtime when doubling the data, we have seen a different picture for Elf. The full Elf slightly improves its runtime when increasing the data size due to a better exploitation of prefix-redundancies. The scaling factor can even be improved by Elf$_{red}$ due to its good ratio between columns used in the selections and its total size. However, due to the small size (i.e., length of paths) of Elf$_{min}$, it does not scale better than the other Elf scenarios. In fact, Elf$_{min}$ scales similar to Sorted Projections with a factor of 0.97.

At last, we compared Elf's query performance for the same data and queries of Experiment 2 in MonetDB. Overall, the results of this test confirm our previous evaluation results. The Elf integrated in MonetDB can reach similar performance improvements as in Experiment 2 when compared to MonetDB's built-in highly-optimized scans, which emphasizes the validity of our results.

4.6 Summary

In this chapter, we argue for multi-dimensional index structures to efficiently support multi-column selection predicates as the answer to research question RQ 3. To this end, we present the conceptual design of our multi-dimensional main-memory index structure Elf that uses prefix-redundancy elimination, ordered node entries putting a multi-dimensional ordering on the data, and a main-memory-optimized storage layout for efficient traversal. After presenting the conceptual design and some optimizations for Elf, we explain its search algorithm and a simple heuristic for ordering its columns. We extensively evaluate Elf's performance against our baseline, scans generated using our proposed adaptive reprogramming approach, against state-of-the-art accelerated full-table scans (BitWeaving and Column Imprints), against recent multi-dimensional competitors (BB-Tree and Sorted Projections), and against MonetDB's scans. In all experiments, adaptive reprogramming is the best accelerated full-table scan and Elf outperformed all competitors by several factors for highly-selective workloads. We end this chapter with a detailed description of our competitors and related work.

5. Complex Selection Queries in Elf-Supported Main-Memory Database Systems

The overarching goal of this thesis is to provide an acceleration for a comprehensive set of selections in relational main-memory database systems. This goal can be split into two parts: accelerating arbitrary selections, and accelerating the runtime of the whole SQL query due to improved selection performance.

So far, we investigated the suitability of Elf for dictionary-compressed values executing mono and multi-column selection predicates, which provides the basic set that an SQL engine should support. However, SQL does not only support selections on single values (i.e., constants), but also with a list of constants (IN-predicates) and also comparing sets of values (i.e., a comparison of values of two columns)[1] [ISO99]. To reach a comprehensive selection support in Elf, we have to extend its capability to execute column-column comparison and IN-predicates. As a positive side result, we also support the majority of TPC-H queries. As a consequence, our Elf is now able to accelerate a similar set of selection predicates as usual scans, which allows to comprehensively reach our goal of Level 2 by answering research question RQ 4.

Reaching reasonable performance boosts with a sort-based index structure for the whole SQL query does not only concern the selection itself. Especially the interoperability between selection results and subsequent operators is an important issue when considering query runtimes in a holistic manner. As a proof-of-concept for an easy interoperability, we extended the well-known main-memory database system MonetDB [BK99] to also support Elf as an additional index structure. To this end, Elf can use MonetDB's query facilities and dictionary compression to execute whole TPC-H queries using Elf for selections. As a result of the this proof-of-concept, we can answer the following research questions:

[1]Another important operation is a `like`-predicate. However, by exploiting a dictionary encoding and supporting IN-predicates, we are able to efficiently support also `like`-predicates.

- Is it possible to integrate a multi-column index into a column store query engine?
- What speedups can be expected when using a sort-based multi-column index?
- How can the incurred overhead of a sort-based structure be resolved?

Especially the last question arises, because Elf puts a multi-dimensional ordering depending on the data, forcing TIDs to be possibly scattered. Hence, it also impacts the runtime of subordinate operators that may incur random access due to the selection result. Hence, the ultimate question to resolve this issue is whether overheads are acceptable or a complete rebuild of the query engine is inevitable. To this end, we investigate the improvement of scan performance and also performance drawbacks for downstream operators in our evaluation section of this chapter. In fact, the holistic evaluation of Elf in MonetDB allows to assess the whole contribution of Elf to the field of main-memory database systems and to the goal of Level 2. It also answers RQ 5.

In summary, we make the following contributions in this chapter:

- Concept and implementation of efficient algorithms for executing column-column comparisons and IN-predicates
- Evaluation of performance impacts of different factors for column-column comparisons and IN-predicates in several microbenchmarks
- Integration concepts for a multi-dimensional index structure into a main-memory column store on the example of Elf and MonetDB
- Evaluation of performance benefits and drawbacks of using Elf in MonetDB for a reasonable set of TPC-H queries

5.1 Complex Selection Predicates

The SQL standard allows for a variety of possibility to limit the tuples that qualify for the result set. Apart from the possibility to connect different selection operators with logical `AND` and `OR`, a single selection predicate can highly differ in its complexity. Simple selection predicates include our multi-column selection predicates where the column values are compared against a given constant. More complex selections extend a selection to be checked against a list of constants (i.e., IN-predicates) and comparing values of different columns for the same tuple (i.e., column-column comparisons). Both complex selection predicates pose a big challenge for full-table scans. Similar to multi-column selection predicates, a column-column comparison needs to touch all tuples, but this time cache misses are incurred for all involved columns. Hence, the more columns are involved, the higher the overhead. Even more severe, an IN-predicate usually incurs one scan per value in the list of values of the IN-predicate. To overcome these issues, we investigate how to exploit Elf's properties of prefix-redundancy elimination and sorted node entries to accelerate complex selection predicates beyond the potential of full-table scans.

To this end, in this section, we explain how to exploit the crafty design of Elf to execute column-column comparisons and IN-predicates. First, we give a broad overview of the general idea of executing complex selection predicates in Elf. Afterwards, we elaborate implementation details for each complex selection predicate and each special type of nodes in Elf (i.e., hash map, `DimensionList`, `MonoList`).

5.1.1 Column-Column Comparisons

In comparison to usual selection predicate, where a constant is compared to all values of a column, a column-column comparison is more complex. A column-column comparison is defined by a comparison operator θ between two columns with $\theta \in \{<, >, <=, >=, =, <>\}$. A necessary prerequisite is that the columns have the same cardinality. Hence, columns of the same table can be compared early in the query plan, while conceptually a comparison across tables is executed after joining both tables.

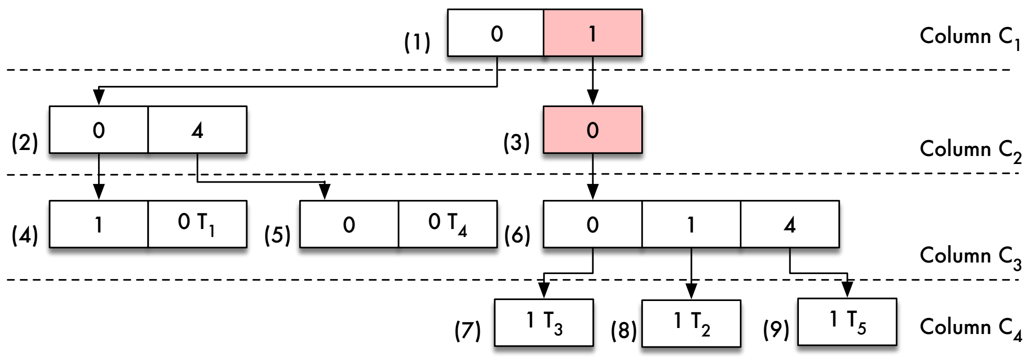

Figure 5.1: Prefix redundancy elimination in Elf for efficient evaluation of column-column comparisons

Column-Column Comparisons in Elf

As an example to illustrate column-column comparisons in Elf, we visualize an Elf built on five tuples with four columns in Figure 5.1. To accelerate column-column comparisons, we can exploit the prefix-redundancy elimination property of Elf. Due to the prefix-redundancy elimination, all values within a path down the tree have the same values as prefix. Hence, if there is a column-column comparison predicate between two columns, e.g., $C_1 <= C_3$, we can use the prefix of the nodes in level C_3 on deciding which values qualify the predicate. To allow for column-column comparisons, the prefix of `DimensionList` $L_{(6)}$, marked in red, has to be saved while traversing down the tree[2]. Upon reaching the deeper column in the tree on which a column-column predicate is defined, in our case C_3, we can compare its values to the value of C_1 in the prefix (which is stored at position 0 in the prefix). For `DimensionList` $L_{(6)}$, the first and second `DimensionElement` qualify the given predicate and we can even use early pruning if the current `DimensionElement` has a bigger value than the prefix value it is compared against (of course, this works only for the comparators $<, <=, or =$).

[2]Notably, storing the prefix of the traversal is also the straight-forward way to allow for an efficient materialization of the query result.

Regarding the column order of Elf, a column-column comparison adds another predicate type to be considered while optimizing the column order. In general, it adds another a comparison with a usual selectivity to the heuristic. However, when the column-column comparison does not comply with the column order (e.g., consider the comparison $C_4 <= C_3 <= C_1$), the predicates have to be transformed to follow the given column order of the Elf (e.g., to a comparison of $C_1 > C_3 > C_4$).

Notably, a column-column comparison is only executed when reaching the second column that is part of the comparison. Hence, the first possibility to execute a column-column comparison is on the second column and, thus, there is no special code for the hash map.

Column-Column Comparison on a `DimensionList`

Since the execution of a column-column comparison is on the second column that is part of the comparison, the code for traversing the hash map is the same as the code in Algorithm 1. Hence, an actual processing of the column-column comparison is only possible on a `DimensionList` or `MonoList`. We present the code that is executed on a `DimensionList` in Algorithm 4.

The column-column comparison is encoded with three input parameters of the function: `colColSel`, `compOPs`, `compCols`. The boolean array `colColSel` encodes whether the current column is part of the comparison. The arrays `compOPs` and `compCols` encode the comparison operator θ and the (previous) column that our values have to be compared against, respectively. Another additional input parameter of this function is the current path encoding the prefix of the current `DimensionList` used for comparison.

The algorithm for executing a column-column comparison (cf. Algorithm 4) works as follows. At first, the current column is checked upon the existence of a column-column comparison (Line 2). If there is no column-column comparison on the current column, we just follow the pointers of all `DimensionElement`s. Otherwise, we have to investigate the comparison operator of the column-column comparison (Line 3 and 18ff). In the shown case of equality, we check for each `DimensionElement`, whether its value is equal to the value of the comparison column in the prefix. In case of a match, we store the current value in the prefix for a prospective further column-column comparison (Line 7) and call the respective function for the next `DimensionList` (Line 10) or `MonoList` (Line 12). If there is no match and the current value is bigger than its prefix, we can skip the remaining entries due to the property of sorted node entries in Elf. Please mind that an early pruning is only possible for the comparison operators $\theta \in \{=, <, <=\}$. The other operators iterate till the end of the list.

Column-Column Comparison on a `MonoList`

We show the pseudo code for evaluating a column-column comparison on a `MonoList` in Algorithm 5. It is very similar to the code for executing a multi-column selection predicate (cf. Algorithm 3). Since there is only a consecutive list of column values in the `MonoList`, we iterate through them without any necessary jumps (Line 2). For each `DimensionElement`, we append its value to the prefix (Line 3) and check

```
1  ColColDimList(startlist, col, path, colColSel, compOPs, compCols, L){
2      if (colColSel[col])then
3          if (compOPs[col] = EQUALS)then
4              position ← startList;
5              do
6                  if (Elf[position] = path[compCols[col]])then
7                      path[col] = Elf[position];
                       // remember current path
8                      pointer ← Elf[position + 1];
                       // start of next list in col+1
9                      if (noMonoList(pointer))then
10                         ColColDimList(pointer, col +
                             1, path, colColSel, compOPs, compCols, L);
11                     else
12                         L ← L+ColColML(unsetMSB(pointer), col +
                             1, path, colColSel, compOPs, compCols, L);
13                     end if
14                 else if (Elf[position] > path[compCols[col]])then
15                     return;// abort
16                 position ← position + 2;
17             while (notEndOfList(Elf[position]));
18         else if ( /* check other comparators                */)then
               // execute code from the If case above with the needed
                  comparison operator
               // Please mind:  early pruning is only possible for
                  EQUALS, LESS_THAN, LESSER_EQUALS
19     else
           // call ColColDimList or ColColML with col + 1 for all
              elements
20     end if
21 }
```

Algorithm 4: Column-column queries on a `DimensionList`

whether it is part of a column-column comparison (Line 4). If it is for example an equality comparison, we check whether the current value and the column in the prefix do not match (Line 6). In this case, we can return since the tuple(s) do not qualify the predicate. Please mind for this execution, we have to negate the comparison operator to stop iterating in case the predicate is not fulfilled. Otherwise, if all predicates match, we extract the TIDs and add them to the result (Line 11-15).

```
1  ColColML(startlist, col, path, colColSel, compOPs, compCols, L){
2      for (curCol ← col to NUM_COLs)do
3          path[curCol] = Elf[startlist + curCol − col];
           // remember current path
4          if (colColSel[curCol])then
5              if (compOPs[curCol] = EQUALS)then
6                  if (Elf[startlist + curCol − col] != path[compCols[curCol]])then
7                      return;// tuple does not qualify
8              else if ( /* check other comparators                */)then
                   // adapted code with the needed comparison operator
9      end for
10     position ← NUM_COLs − col;
11     while (notEndOfList(Elf[position]))do
           // Handle duplicates
12         L ← L + Elf[position];
13         position ← position + 1;
14     end while
15     L ← L+unsetMSB(Elf[position]);
16 }
```

Algorithm 5: Column-column queries on a MonoList

5.1.2 IN-Predicates

Another important complex selection predicate of SQL are IN-predicates. An IN-predicate consists in our case of a list of attribute values and an equality or inequality predicate. In case of an equality IN-predicate, we have to output those column values that are contained in the list of attribute values. In case of an inequality IN-predicate, all values that are not in the list of attribute values qualify the predicate.

IN-Predicate Evaluation in Elf

A straight-forward way of executing an IN-predicate in Elf is to issue one query per IN-predicate and then merge (i.e., unite) the results. However, this would lead to v traversals for a list of attributes with v entries. Furthermore, single traversals do not exploit the Elf's properties of prefix-redundancy elimination and ordered node entries.

A more sophisticated execution of an IN-predicate is to execute a merge join between the list of attribute values and the values of the DimensionList. To this end, we first sort the values of the list and then step-wise compare the values with the values

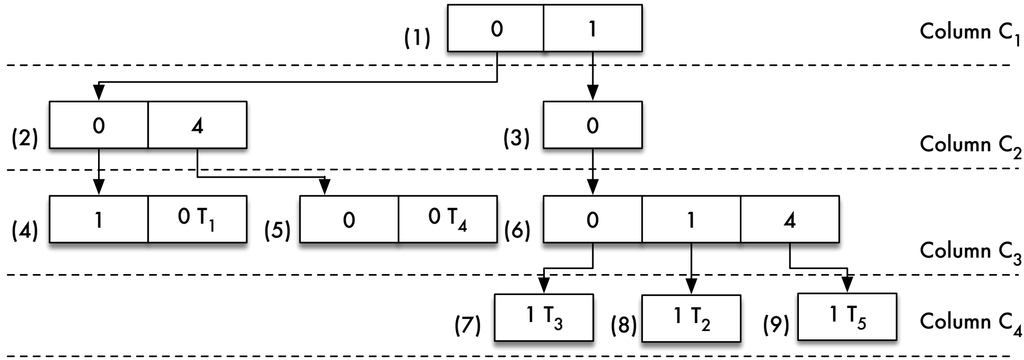

Figure 5.2: Prefix redundancy elimination in Elf for efficient IN-predicate evaluation

of the `DimensionList`. This procedure results in a linear time complexity instead of the naiive quadratic time complexity.

For example, consider an equality IN-predicate with $[0, 1]$ as the list of values on column C_3 from the Elf in Figure 5.2. In the case of `DimensionList` $L_{(6)}$, the first entry of the list of the IN-predicate (i.e., 0) is compared with the first value of $L_{(6)}$ (i.e., 0 as well). In case of a match, we can proceed both pointers; in case of a mismatch, we proceed the smaller value's pointer similar to a merge join. In case, we reach the end of the list of the IN-predicate, we can directly skip any remaining sub trees of Elf, which essentially exploits the opportunity for early pruning.

An inequality IN-predicate is similar to the equality version. In this case, however, we follow the path down the Elf in case the value of the `DimensionElement` is smaller than the current IN-value. Furthermore, the last `DimensionElement`s are a match in case we iterated to the end of the IN-list. For example, take the inequality predicate with $[1, 2]$ as the list of values on column C_3. When evaluating `DimensionList` $L_{(6)}$, the first `DimensionElement` matches since its value is smaller than the first IN-list value. Furthermore, after checking all IN-list values, we follow the path down the last `DimensionElement` with the value 4.

IN-Predicate Algorithm

The function to execute an IN-predicate is shown in Algorithm 6. Due to the traversal-based processing, we use an input array that encodes whether the current column has an IN-predicate. This array, called `inSel`, also encodes whether it is an equality or inequality predicate. Another input parameter is a multi-dimensional array containing the IN-list values, called `inValues`.

The algorithm first checks whether an IN-predicate is defined on the hash map (Line 3). In this case, we determine whether the IN-predicate is an equality or an inequality predicate (Line 4 and 13)[3]. In the case of an equality predicate, we can directly jump to the according values in the IN-predicate due to the properties of our hash map (Line 7). For each match, we recursively call `INDimList` (Line 9) or `INML` (Line 11) if the next node is a `DimensionList` or a `MonoList`, respectively.

[3]For ease of implementation, the `inSel` array is a char array with the following encodings. A value of 0 represents no IN-predicate, a value of 1 represents an equality IN-predicate, and a value of 2 represents an inequality IN-predicate.

```
1  INQuery(inSel, inValues){
2      L ← ∅;
3      if (inSel[0])then
4          if (isEq(inSel[0]))then
               // equality predicate
5              curINValue ← inValues.begin();
6              while
                (curValue <= max{C_1} && curINValue < inValues.end())do
7                  pointer ← Elf[inValues[curINValue]];
8                  if (noMonoList(pointer))then
9                      INDimList(pointer, col + 1, inSel, inValues, L);
10                 else
11                     L ← L+INML(unsetMSB(pointer), col+1, inSel, inValues, L);
12                 end if
13             end while
14         else
               // inequality predicate
15             curValue ← 0;
16             curINValue ← inValues.begin();
17             while
                (curValue <= max{C_1} && curINValue < inValues.end())do
18                 if (curValue != inValues[curINValue])then
19                     pointer ← Elf[curValue];
20                     if (noMonoList(pointer))then
21                         INDimList(pointer, col + 1, inSel, inValues, L);
22                     else
23                         L ←
                             L+INML(unsetMSB(pointer), col+1, inSel, inValues, L);
24                     end if
25                 else
26                     curINValue ← curINValue + 1;
27                 end if
28                 curValue ← curValue + 1;
29             end while
30         end if
31     else
           // call INDimList or INML with col + 1 for all elements
32     end if
33     return L;
34 }
```

Algorithm 6: IN-predicate evaluation on hash map

```
1  INDimList( startlist, col, inSel, inValues, L){
2      if (inSel[col])then
3          if (isEq(inSel[col]))then
4              curINValue ← inValues.begin();
5              do
6                  if (Elf[position] < inValues[curINValue])then
7                      position ← position + 2;
8                  else if (Elf[position] > inValues[curINValue])then
9                      curINValue ← curINValue + 1;
10                 else
11                     pointer ← Elf[position + 1];
12                     if (noMonoList(pointer))then
13                         INDimList(pointer, col + 1, inSel, inValues, L);
14                     else
15                         L ←
                             L+INML(unsetMSB(pointer), col+1, inSel, inValues, L);
16                     end if
17                     position ← position + 2;
18                     curINValue ← curINValue + 1;
19                 end if
20             while
                (notEndOfList(Elf[position]) && curINValue < inValues.end());
21         else
               // NOT IN predicate
               // Evaluate next column if
                  Elf[position] < inValues[curINValue]
               // In the other cases increment position or curINValue
22         end if
23     else
           // call INDimList or INML with col + 1 for all elements
24     end if
25 }
```

Algorithm 7: IN-predicate evaluation on a `DimensionList`

If the current IN-predicate is an inequality IN-predicate, we have to iterate through all pointers of the hash map and check whether the currently represented value in the hash map does not match the current value in the IN-list. If the inequality predicate is satisfied, we also follow the pointer recursively by differentiating a `DimensionList` (Line 21) or `MonoList` (Line 23). Afterwards, we proceed with the next entry in the hash map. In the case the current value of the hash map and the IN-list are equal (Line 25-27), we do not follow the pointer down the Elf, but increment the position in the IN-value in addition to the increment of the hash map. Please note that the case where the IN-value is smaller than the hash map value does not exist. This is due to the denseness criteria of the hash map. Hence, all IN-values will eventually be found in the traversal of the hash map (except it is bigger than the column maximum).

IN-Predicate Evaluation on a `DimensionList`

We present the pseudo code for evaluating an IN-predicate on a `DimensionList` in Algorithm 7. When evaluating an IN-predicate on a `DimensionList`, we have to follow a merge-join-like execution, because we do not have guaranteed denseness in values of a `DimensionList`. Hence after checking for an IN-predicate (Line 2) and differentiating between an equality (Line 3ff) or inequality (Line 19ff) IN-predicate, we stepwise iterate through `DimensionList` and IN-list. In case of an equality IN-predicate, we increment the pointer of the smaller value (Line 6-10) and on a match (Line 10-19), we recursively call `INDimList` or `INML`. On a match, we also increment both pointers, since neither in the `DimensionList` nor in the IN-list are duplicate values allowed.

For the inequality case, we switch cases. Here, we call `INDimList` or `INML` if the `DimensionList` value is smaller than the current IN-list value and in the other cases, we advance either pointer.

IN-Predicate Evaluation on a `MonoList`

In a `MonoList`, there is only a single path represented down to the tuple identifiers. As a consequence, there is only a single value in the `MonoList` that is to be found in the IN-list. Hence, after checking for an IN-predicate (Line 3), we search the current value of the `MonoList` at the specified column in the IN-list. Currently, we use a vectorized predicated search since we assume only a limited amount of IN-list values. For bigger IN-lists, a binary or k-ary search is useful [SBS18]. This, however, seems to be implausible for real SQL use cases, since usually the data to be analyzed exceeds RAM and not the query.

Depending on the type of IN-predicate, we either continue the evaluation when the value has been found in the IN-list (Line 7) and otherwise skip this `MonoList` (Line 8), or we switch those cases for an inequality IN-predicate. The remaining code from Line 15 to 20 represent the TID extraction as already used in Algorithm 3 and 5.

5.1.3 Summary

In this section, we investigated how to exploit Elf's design for selection predicates beyond multi-column selection predicates. To reach a comprehensive set of predicates,

```
1  INDimList( startlist, col, inSel, inValues, L){
2      for (curCol ← col to NUM_COLs) do
3          if (inSel[col]) then
4              result ← search(inValues,Elf[startlist + curCol − col]);
5              if (isEq(inSel[col])) then
6                  if (isFound(result)) then
7                      continue;
8                  return;
9              else
10                 if (isFound(result)) then
11                     return;
12                 continue;
13             end if
14     end for
15     position ← NUM_COLs − col;
16     while (notEndOfList(Elf[position])) do
           // Handle duplicates
17         L ← L + Elf[position];
18         position ← position + 1;
19     end while
20     L ← L+unsetMSB(Elf[position]);
21 }
```

Algorithm 8: IN-predicate evaluation on a `MonoList`

we extended Elf for column-column comparisons and IN-predicates. To this end, we explain a conceptual idea how to exploit Elf's design for accelerating both complex selection predicates and also present algorithms for ease of reproducibility. An evaluation of the performance for these algorithms is presented in Section 5.3.1.

5.2 MonetDB Integration

With the conceptual design of executing complex selection predicates from the previous section, we are able to execute a comprehensive set of SQL queries. Especially most of the queries of the TPC-H are now supported. However, accelerating only the selection predicates of a query with Elf does not necessarily mean that the whole query time can be reduced. In fact, there are two possible reasons that an Elf-based query execution is outperformed:

1. A query engine could be able to exploit data and query characteristics using additional data structures. For instance, MonetDB features Column Imprints, candidate scans (i.e., previous positive selection results are only considered for the next selection), and even a hash lookup (i.e., values are hashed and only the matching bucket(s) are scanned sequentially) for accelerating selections. Using these additional structures gives a non-linear performance improvement.
2. Due to the sorting of Elf, the resulting tuples are usually not in insertion order. Hence, when a subsequent operator works with the results of an Elf selection, it usually incurs additional cache misses due to the unordered result adding a considerable overhead.

For a comprehensive comparison between a system's clever query processing engine supporting different ways of executing a complex predicate, we aim to integrate Elf into MonetDB [BK99]. This serves as a proof-of-concept for the usability of Elf for accelerating scans in a set of reasonable queries. However, the overhead for using Elf leads to the ultimate goal of building a query engine to exploit Elf's properties instead of interfacing Elf to an arbitrary query engine.

In fact, there are several systems that could be used for integrating Elf. Still, we choose MonetDB for the following reasons: First, Elf is a main-memory structure and its aim is to accelerate in-memory query processing. Hence, for disk-based systems we would have to investigate suitable buffering techniques [JLR+94, ADHW99, SSH11] since paging of the operating system would not be efficient enough. Second, the operator-at-a-time bulk-processing engine of MonetDB is well suited for integrating Elf, because we currently return all matches at once. Of course, an adaption for tuple-at-a-time processing is possible but needs additional implementation overhead. Considering compilation-based engines, such as Hyper [KN11], would need a different design, because recursive traversals would have to be transformed to a tuple-at-a-time, kernel-like processing. Furthermore, due to its closed-source development, we are currently unable to integrate Elf. Third, the competing storage scheme of a column store in MonetDB is reasonable for analytical scenarios. Still, it would be interesting to compare Elf against hybrid storage schemes between column and row stores, such as in Peloton [APM16], Aqua [Lüb17], H2O [AIA14]. Last but not least, MonetDB is open source which allows for an accountable integration.

Integration Overview

The integration of Elf into MonetDB has touched several layers. To give a short overview, we adapt the following components:

- MonetDB's *SQL parser* is extended to allow for an index creation of the type Elf.
- the *MAL (MonetDB Assembly Language) interface*, which is the internal mapping layer between MonetDB's front-end languages (e.g., SQL) and the operator of the query processing back-end. Here, we have to allow to not only map a selection predicate to a selection operator, but also to a selection in Elf.
- the query optimizer of MonetDB is adapted to merge single predicates that can be executed in concert on a single Elf. Furthermore, optimizations of the query plan should favor the execution using Elf and should not hinder its applicability.
- MonetDB's storage engine is extended to store the built Elf upon table indexing in the heap.

In the following, we discuss the extensions to the MAL layer and the impact on interoperability of other operators in detail, because these extensions are important to weigh the benefits and drawbacks of Elf. For a more comprehensive presentation on all touched parts in MonetDB, we refer to the master thesis of Florian Bethe [Bet18][4].

5.2.1 MAL Extensions

In order to use our Elf during query execution, we extend MonetDB's internal representation of a query plan – the MAL plan [BK99] – with suitable back-end operator calls that construct and execute the selections on the Elf. We show the necessary extensions to the MAL primitives in Algorithm 9.

The first two operators (`create_query_idx`, `drop_query`) are required for housekeeping. Creating a query allocates memory for the necessary input parameters for a selection of the Elf. These are the input arrays with the lower and upper bounds of multi-column selection predicates (cf. Section 4.4), the IN-lists for IN-predicates (cf. Section 5.1.2), and the columns and comparators for column-column comparisons (cf. Section 5.1.1). To convert comparison values into the same type that Elf uses for indexing (unsigned 32-bit integer values), the functions `num2elfval` (Line 3) and `resolve_bat_index_map` (Line 4) are needed. While the former converts integer values of different types (bit, bte (i.e., byte), short, int, lng (i.e., long), any), the latter operator converts string values. String values are represented as an identifier in an order-preserving dictionary that MonetDB holds in addition to the general

[4] Florian Bethe implements two variants of string resolution: (1) resolve based, where an identifier is stored in the Elf and during selection execution, the string comparison is done and (2) index-based, where an order-preserving dictionary is used to resolve identifiers before running the final selection. Since the second alternative has proven to be mostly superior, we only consider an index-based Elf in this thesis.

```
   // Create/destroy a selection query
1  command create_query_idx(elf:ptr):idxquery;
2  command destroy_query(query:idxquery):void;
   // Value conversion
3  command num2elfval(v:type_t, cmp:int):elfval;
4  command resolve_bat_index_map(b:bat[:any], map:bat[:elfval], val:any_1,
     cmp:int):elfval;
   // Operator combination and selection execution
5  command add_window_query(query:idxquery, dim:int, cmp:int,
     val:elfval):idxquery;
6  command add_in_query(query:idxquery, dim:int, cmp:int, val:elfval):idxquery;
7  command add_col_col_query(query:idxquery, dim:int, target_dim:int,
     cmp:int):idxquery;
8  command select(query:idxquery):bat[:oid];
```

Algorithm 9: Additional MAL operators for an integration of Elf

columnar storage as a *BAT (Binary Association Table)* [BMK99]. As a consequence, `resolve_bat_index_map` scans the dictionary and determines the corresponding identifiers for the comparison operators.

The last set of operators fill and execute a selection on the Elf. A converted value (`val`) can be added to a multi-column selection predicate or to an IN-list in a specific column (`dim`) by using `add_window_query` or `add_in_query`, respectively. Similarly, `add_col_col_query` is used to put a comparison on two columns (`dim` and `target_dim`). At last, the operator `select` executes a prepared selection query on the given Elf and returns an unordered BAT. How to use this BAT and how to execute subsequent operators after Elf is explained in the following section.

5.2.2 Operator Interoperability

The integration of Elf into MonetDB does not significantly change the MAL plan for a given query. The reason for the minimal invasiveness is two fold:

1. Selections on base tables are usually executed early in the query plan – mostly as the leaves of the query tree. Hence, we only have to care about subsequent operators. Please mind, that we do not support selections on intermediate result table since those would have to be linearized into an Elf at runtime.

2. MonetDB's scalar scans usually create a candidate list as a result, which is similar to the result of our Elf from Definition 2.1, because MonetDB delays result materialization as long as possible [IKM09]. Hence, the overall query execution of MonetDB only has to be extended, not fundamentally changed.

To exemplify both changes to the query plan, we show an excerpt of the Q17 query plan in Algorithm 10 and the new one when using the Elf for selections in Algorithm 11. On the left side of the assignment operator (":="), the result variable is presented, which is either a single BAT or a pair of BATs in case of a join (cf. Line 9). On

the right side of the assignment operator, the executed function and its parameters are listed. One of the first operations is to bind the input columns to a variable. In Algorithm 10 Line 2 and 3, the two columns for the selections are bound to variable `X_77` and `X_70`. In Line 5 and 6, the selections are executed storing their results in a BAT of OIDs (MonetDB's term for TIDs). In the last selection, MonetDB executes a candidate scan with the result of the previous scan as input. The final result of both selections (`C_88`) is then used for projecting the qualifying `l_partkeys` (Line 8), which are then used for the subsequent join (Line 9).

```
1  ...
2  X_77:bat[:str] := sql.bind(X_12:int, "sys":str, "part":str, "p_container":str, 0:int);
3  X_70:bat[:str] := sql.bind(X_12:int, "sys":str, "part":str, "p_brand":str, 0:int);
4  ...
5  C_85:bat[:oid] := algebra.thetaselect(X_70:bat[:str], C_61:bat[:oid],
     "Brand#23":str, "==":str);
6  C_88:bat[:oid] := algebra.thetaselect(X_77:bat[:str], C_85:bat[:oid], "MED
     BOX":str, "==":str);
7  X_63:bat[:int] := sql.bind(X_12=0:int, "sys":str, "part":str, "p_partkey":str,
     0:int);
8  X_89:bat[:int] := algebra.projection(C_88:bat[:oid], X_63:bat[:int]);
9  (X_113:bat[:oid], X_114:bat[:oid]) := algebra.join(X_89:bat[:int], X_25:bat[:int],
     nil:BAT, nil:BAT, false:bit, nil:lng);
10 ...
```

Algorithm 10: Excerpt of MAL plan of TPC-H `Q17`

Compared to the MAL plan in Algorithm 10, the resulting MAL plan when using Elf in Algorithm 11 comprises several additional instructions. At first, we have to call `create_query_idx` to allocate the necessary space. For the first predicate (`p_brand`="*Brand*#23"), the column is bound to the Elf (Line 2) and the value is resolved. To resolve the string, its dictionary is scanned and the identifier is retrieved in `X_101`. Afterwards, the selection value is added as an interval predicate on the first level of the Elf (Line 5). The same procedure is executed for the predicate on `p_container`, but on the column at position 1. In Line 9, the selection predicate is finally executed on the Elf. The result is an *unordered* BAT with matching TIDs. Hence, the next step is to project the `l_partkeys` that belong to the matching tuples. This time, however, `projectionpath` is called since MonetDB cannot rely on ordered TIDs. The `l_partkeys` are then passed to the join operator, which also cannot rely on keys that are sorted by TID.

In summary, due to the unordered output of TIDs, our Elf approach incurs additional overhead for subsequent operators like projections and joins. The real impact of this overhead for real-world queries is investigated in our evaluation.

5.3 Evaluation

The evaluation section consists, similar to the content of this chapter, of two parts: first, we evaluate the efficiency and impact of different parameters on Elf's query

```
1  ...
2  X_97:idxquery := elf.create_query_idx(9:ptr);
3  X_83:bat[:elfval] := elf.bind_index_map(X_12:int, "sys":str, "part":str,
    "p_brand":str);
4  X_101:elfval := elf.resolve_bat_index_map(X_85:bat[:str], X_83:bat[:elfval],
    "Brand#23":str, 4:int);
5  X_102:idxquery := elf.add_window_query(X_97:idxquery, 0:int, 4:int,
    X_101:elfval);
6  X_93:bat[:elfval] := elf.bind_index_map(X_12:int, "sys":str, "part":str,
    "p_container":str);
7  X_106:elfval := elf.resolve_bat_index_map(X_95:bat[:str], X_93:bat[:elfval],
    "MED BOX":str, 4:int);
8  X_107:idxquery := elf.add_window_query(X_102:idxquery, 1:int, 4:int,
    X_106:elfval);
9  X_109:bat[:oid] := elf.select(X_107:idxquery);
10 X_69:bat[:int] := sql.bind(X_12:int, "sys":str, "part":str, "p_partkey":str, 0:int);
11 X_110:bat[:int] := algebra.projectionpath(X_109:bat[:oid], C_67:bat[:oid],
    X_69:bat[:int]);
12 (X_137:bat[:oid], X_138:bat[:oid]) := algebra.join(X_110:bat[:int], X_25:bat[:int],
    nil:BAT, nil:BAT, false:bit, nil:lng);
13 ...
```

Algorithm 11: Excerpt of MAL plan of TPC-H `Q17` using the integrated Elf

performance for complex predicates (Experiment 1 & 2) and, second, we evaluate the impact of including Elf in MonetDB for queries with complex predicates (Experiment 3). While the first part is represented as microbenchmarks with artificial queries that are inspired by TPC-H queries, the second part evaluates a set of real TPC-H queries. In this second part, we investigate what optimization potential can be gained when using Elf and also how the impact to subsequent operators is. This impact probably results in a performance decrease for the other operators due to Elf's unordered output leading to many additional cache misses. The observed limitations of Elf lead to the ultimate need for an adaption of query engines to suit Elf's processing model in future work.

Evaluation Setup

Throughout this section, we evaluate the performance of Elf on complex predicates with the same Elf use cases (i.e., Elf, Elf_{red}, Elf_{min}) and the same machine that we used in Section 4.5. Furthermore, we adapted the column order of Elf, because we now use several additional queries from the TPC-H ($s = 100$), which is possible due to Elfs extension for column-column comparisons and IN-predicates. The resulting column order for the `Lineitem` and `Part` table is shown in Figure 5.3. We also use this column order for our microbenchmarks to have a realistic column order even though each microbenchmark operates only on a limited set of columns (these are, however, reflected in the results of Elf_{min}). A detailed description of the used queries and how they impact the resulting column order is given in Section 5.3.2.

<u>Lineitem</u>					
Elf	:=	(0) l_quantity	(1) l_shipmode	(2) l_receiptdate	(3) l_commitdate
		(4) l_shipdate	(5) l_discount	(6) l_shipinstruct	(7) l_returnflag
		(8) l_linestatus	(9) l_extendedprice	(10) l_tax	(11) l_orderkey
		(12) l_partkey	(13) l_suppkey	(14) l_linenumber	
Elf_{red}	:=	(0) l_quantity	(1) l_shipmode	(2) l_receiptdate	(3) l_commitdate
		(4) l_shipdate	(5) l_discount	(6) l_shipinstruct	
<u>Part</u>					
Elf	:=	(0) p_brand	(1) p_container	(2) p_size	(3) p_type
		(4) p_name	(5) p_mfgr	(6) p_retailprice	
		(7) p_comment	(8) p_partkey		
Elf_{red}	:=	(0) p_brand	(1) p_container	(2) p_size	(3) p_type

Figure 5.3: Query details for mono and multi-column selections

5.3.1 Microbenchmarks for Complex Predicates

In the microbenchmarks, we want to evaluate how well Elf performs when executing different column-column comparison or IN-predicates. To this end, we adapt promising TPC-H queries and vary parameters that influence Elf's execution performance for the specific predicates. We start by analyzing column-column comparisons in Experiment 1 and afterwards show the impact for IN-predicates in Experiment 2.

5.3.1.1 Experiment 1: Column-Column Comparison Microbenchmark

In this microbenchmark, we are interested in how well Elf can exploit prefix-redundancy elimination in column-column comparisons. Since we want to execute meaningful experiments that also represent real-world use cases, we choose column-column comparison from the TPC-H benchmarks. In several queries, the three date columns of the `Lineitem` table (`l_shipdate, l_receiptdate, l_commitdate`) are compared and, hence, these columns are suitable candidates for our evaluation. The TPC-H query `Q12` defines a column-column comparison predicate as `l_shipdate` < `l_commitdate` < `l_receiptdate` and we use transformations of this predicate for our microbenchmark. An example query for the microbenchmark is the following, with $\theta \in \{<, >, <=, >=, =, <>\}$:

select	count(*)
from	lineitem
where	l_shipdate θ l_receiptdate

In the following, we first take two of the date columns and different comparison operators to emulate different selectivities for each query. Afterwards, we examine the impact of having three date columns that are compared against each other resulting in two column-column comparisons being executed.

Single Column-Column Comparison

For a single predicate, we can vary the predicates between `l_shipdate` & `l_commitdate`, `l_receiptdate` & `l_commitdate`, and `l_receiptdate` & `l_shipdate`. Considering all 6 comparison operators, this results in 18 different queries. However, the resulting selectivities only have a distinct number of 11 selectivity factors. In Figure 5.4, we show the selection time for MonetDB's full-table scans and the three Elf scenarios for the resulting 11 queries with distinct selectivity factors.

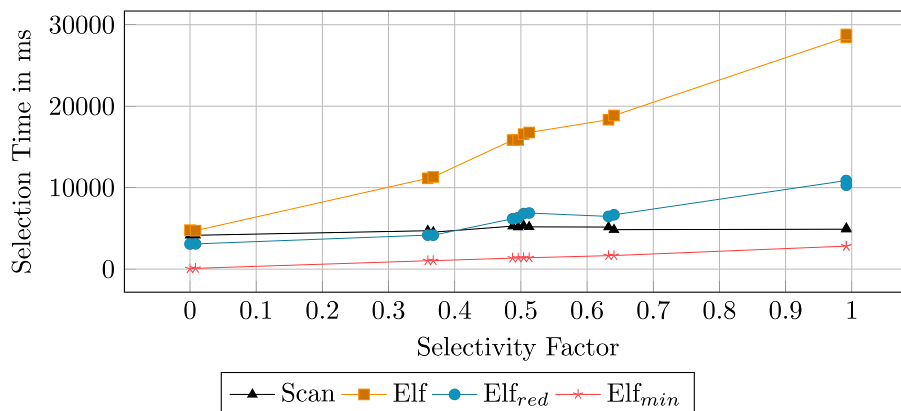

Figure 5.4: Microbenchmark runtime for different column-column comparison operators on two dates of the `Lineitem` table ($s = 100$) in MonetDB

Overall, we can see that an increasing selectivity factor results in an increase in runtime for all approaches. However, while the full-table scan has a rather stable selection time independent of the selectivity, the Elf approaches are more sensitive to the selectivity showing an increasing selection time for an increasing selectivity factor. This is due to the increasing number of traversed paths of the Elf. Furthermore, the baseline of full-table scans incur only a single traversal and, hence, we can expect a similar performance behavior as for mono-column selection predicates. To this end, Elf cannot outperform full-table scans in this experiment. Only Elf$_{red}$ and Elf$_{min}$ are able to beat a full-table scan. While Elf$_{red}$ outperforms a full-table scan up to a selectivity factor of 0.35, Elf$_{min}$ can outperform the full-table scan across all selectivities – even for a selectivity factor of around 1. This is a remarkable result, which is only possible due to the medium sized cardinalities of the date attributes of around 2500 distinct dates each. Hence, the minimal Elf consists of around 2500 pointers in the hash map and 6,250,000 values and pointers in C_1. In fact, the whole Elf$_{min}$ has only 4 % of the size that a single date column of the `Lineitem` table of $s = 100$ has. As a consequence, Elf$_{min}$ is up to a factor of 9 faster than the full-table scan for a single column-column comparison. The impact of several column-column comparisons on the performance is investigated in the second set of queries.

Two Column-Column Comparisons

Compared to the previous set of queries, we extend the column-column comparisons to two comparisons. With the available three date columns and the six date columns, it results in 36 different queries. Executing these 36 queries yield 13 different selectivity factors. In Figure 5.5, we show the performance of the four approaches for these 13 queries.

In general, the performance behavior of the Elf approaches is similar to the experiment with a single column-column comparison. However, MonetDB's full-table scan shows an increasing selection time for an increasing selectivity factor compared to the rather constant performance for a single column-column comparison. This is due to the candidate scans of the second selection that incurs a different overhead depending on the number of matches from the first column-column comparison. Due to this, the full Elf can even outperform full-table scans for up to a selectivity factor around 0.2.

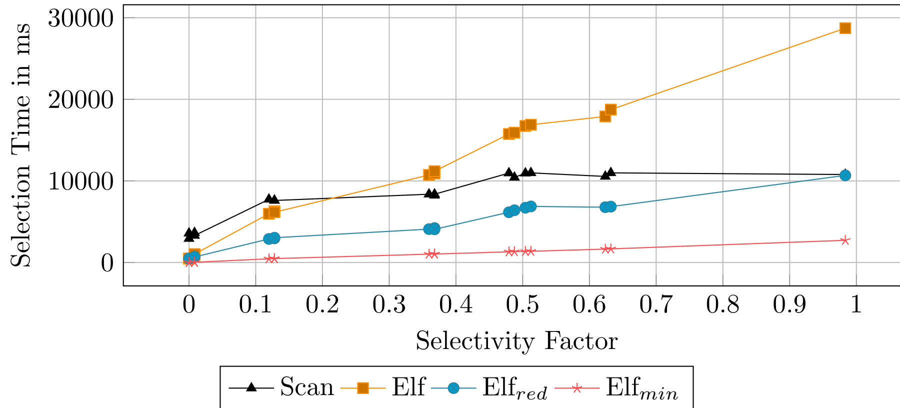

Figure 5.5: Microbenchmark runtime for different column-column comparison operators between three dates of the `Lineitem` table ($s = 100$) in MonetDB

In this experiment, even Elf$_{red}$ can outperform full-table scans for all tested selectivity factors. Up to a selectivity factor of 0.63, the performance difference between these two is around a factor of 2-3. Interestingly, Elf$_{min}$ performs very similar to the experiment with a single column-column comparison. This can be explained, because the main impact for Elf$_{min}$ is filling the result vector and traversals are only a minor overhead. Furthermore, this shows that the proposed algorithm scales well for an increased number of column-column comparisons.

5.3.1.2 Experiment 2: IN-Predicates Microbenchmark

In this section, we are interested in the impact of the parameters for different IN-predicates. There are two important parameters impacting the performance of an IN-predicate:

Size of IN-List: The more values are present in the IN-list, the more comparisons need to be done. Especially full-table scans incur one scan per IN-list, which means that our Elf can probably outperform them in this case.

Attribute Cardinality: The bigger the value domain of an attribute, the bigger are the possible `DimensionLists`. Hence, this parameter influences the runtime of the traversal, but usually not the runtime of scans.

Of course, the parameter of the selectivity and column order play another vital role. However, selectivity is defined by the size of the IN-list and the attribute cardinality. Hence, we more specifically look at the cause than the effect. Also, the impact on the best column order can be mapped to the impact of the selectivity on the column order. Thus, we treat an IN-predicate the same as a multi-column selection predicate in our column order heuristic.

To vary the parameters of the IN-predicate, we choose possible IN-predicates from TPC-H query `Q16` and `Q19`. While `Q16` defines an IN-predicate on the attribute `p_size`, the query `Q19` has an IN-predicate on the attribute `p_container`. Similar to the evaluation of column-column comparisons, we generate different queries of the form:

select	count(*)
from	lineitem
where	p_size IN [15,16]

We choose corresponding IN-list values in the middle of the value range such that we do not exploit early pruning too extensively. In the following, we first look at queries on the attribute `p_container` and afterwards on those using an IN-predicate on the `p_size` attribute.

Experiment on `p_container`

For the IN-predicates on the attribute `p_container`, we vary the number of the containers that are within the 40 possible containers. We choose a maximum size of 10 containers, because this already reaches a selectivity factor of $Sel = 0.25$, which is beyond the typical use case of an index structure. In Figure 5.6, we show the different selection times (i.e., only the selection on the Elf or necessary selection operations of MonetDB, but not the whole query runtime) for an increasing number of containers in the IN-predicate. Each additional container adds 0.025 to the selectivity factor.

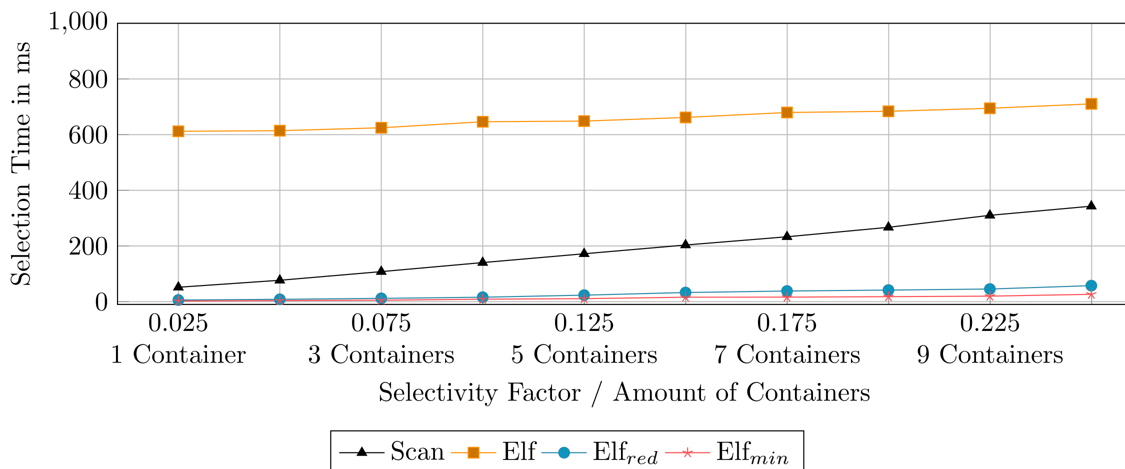

Figure 5.6: Microbenchmark runtime for different IN-lists on the `p_container` attribute of the `Part` table ($s = 100$) in MonetDB

The experiment shows that for an increasing number of IN-list values, the selection time of all approaches (Elf or MonetDB's full-table scan) increases as well. However, the factor is significantly higher for MonetDB's full-table scans than the integrated Elfs. However, especially the full Elf cannot outperform MonetDB's scans since traversing down to the TIDs is very costly. Overall, the impact of retrieving tuples with one or ten containers is only 16 %.

In contrast, when limiting the indexed columns to the necessary ones for the workload (Elf$_{red}$) or the query (Elf$_{min}$), the Elfs can outperform full-table scans by a factor between 9 and 13. Interestingly, the overhead for the selection on the Elf$_{red}$ and Elf$_{min}$ is for both circa a factor of 10 comparing the selection time of one and ten containers. This, however, does not only depend on the size of the IN-list, but also on the cardinality of the attribute, as we can see in the next experiment on `p_size`.

Experiment on p_size

Compared to the experiment before, the attribute p_size has 50 possible values. Hence, its value domain is increased by 25 % compared to p_container. In this experiment, we extended the range to 20 different sizes which results in a maximum selectivity factor of 0.4.

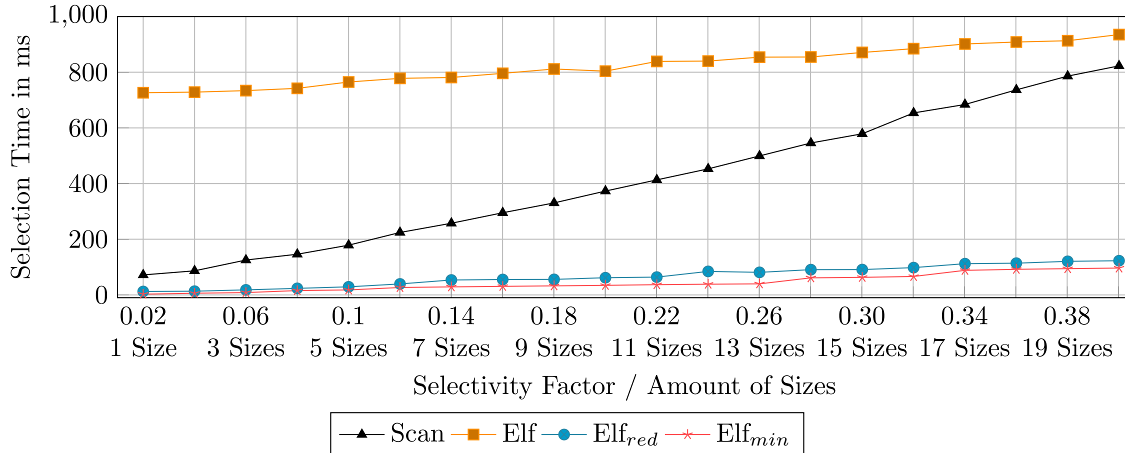

Figure 5.7: Microbenchmark runtime for different IN-lists on the p_size attribute of the Part table ($s = 100$) in MonetDB

In Figure 5.7, we show the selection times for all four approaches for IN-predicates on the p_size attribute. The overall selection time behavior is the same as for the selection on p_container. Again, the full Elf has the worst performance, although for 20 sizes, the difference between Elf and MonetDB's full-table scans is only 13 %. This is due to the increasing number of scans that MonetDB has to execute.

Compared to the experiment on p_container, the Elf_{red} needs 20 sizes to reach an increased selection time of a factor of 10. This is almost double the amount of IN-list values. The explanation for this is the different cardinality of the p_size attribute. Since the initial effort for a selection is a little bit higher due to bigger DimensionLists but there is only a small overhead for the merge-like execution of the IN-predicate, more IN-list values can be checked before the overhead is equally big as for a smaller attribute domain. However, the bigger value domain has the impact that for a selectivity factor of 0.24, the performance difference between a full-table scan and Elf_{red} is only a factor of 5.4 compared to a factor of 6 when selecting 10 containers.

5.3.1.3 Microbenchmark Summary

In this benchmark, we investigated the impact of the selectivity, the number of predicates in a column-column comparison, the number in IN-list values, and the attribute cardinality of IN-predicates on the performance of our Elf. Hence, positive results indicate a well performed design of the algorithms for these complex selection predicates.

Our column-column comparison microbenchmark shows that Elf_{red} can partially (up to a selectivity factor of 0.35) and Elf_{min} can fully outperform a full-table scan for a

single predicate. When increasing the number of column-column comparisons, the baseline worsens its performance due to expensive result materialization and merging such that even the full Elf can outperform full-table scans for a selectivity factor of up to 0.2. This underlines the well chosen design of our column-column comparison efficiently exploiting Elfs design.

Considering IN-predicates, we have seen that especially Elf$_{red}$ and Elf$_{min}$ can outperform MonetDB's full-table scans even for a single IN-predicate by several factors. However, the value domain has a big impact on Elf's selection time due to the impact on the size of the resulting `DimensionLists`. As a result, Elf's IN-predicate algorithm scales well for an increasing number of IN-list values.

As a consequence of these positive results, we assume that also the whole query performance should increase similar to the performance increase for their selections. Whether this hypothesis is true is investigated in the next experiment.

5.3.2 Experiment 3: Elf's Integration Test in MonetDB

With the ability of efficiently executing complex selection predicates in Elf, we extended the usability of Elf to a large variety of SQL queries. This opens the door for executing a variety of real-world queries such as the TPC-H. Since powerful query engines could outperform the processing of Elf for these real-world queries, we want to assess the usefulness of the integrated Elf compared to the query engine of MonetDB in this experiment. In fact, this experiment is a proof-of-concept of the integration of Elf in a columnar main-memory bulk-processing engine on the example of MonetDB.

We expect from this experiment that Elf can outperform MonetDB's full-table scans with respect to selection performance[5]. This expectations arises from our previous experiments and the microbenchmarks. Usually, this means that also the runtime of the whole query is improved in the same manner. However, we are aware of the drawback of using a sort-based structure in a column store (cf. Section 5.2), which means that we have to investigate whether our integrated Elf can beat full-table scans w.r.t. the whole query time.

To assess the usability of Elf and the overhead that is caused for usual queries, we select several queries from the TPC-H benchmark ($s = 100$) that fit the overall use case of Elf. Since single-column selection predicates and small tables probably do not benefit enough from using the Elf, we decided for queries involving at least two columns in a predicates and those should be on the `Part` or `Lineitem` . In Table 5.1, we show the details of the chosen queries, which we can classify into three categories:

Multi-Column Selection Predicate Queries: The usual case of Elf are still multi-column selection predicate queries and, hence, we choose the two queries `Q6` and `Q17` to assess the impact of using Elf for subsequent operators. Query `Q6` only features a multi-column selection predicate with a final sum and, thus, can be seen as a baseline with two projections. In query `Q17`, two joins plus two selections are executed. Hence, the impact on join processing can be assessed with `Q17`.

[5] We used MonetDB 5 server of version 11.8.0. with 64 bit, 128 bit integer.

IN-Predicate Queries: More complex queries represent those that include a column-column comparison, e.g., `Q16` and `Q19`. While `Q16` defines a multi-column selection predicate and an IN-predicate on the `Part` table, `Q19` has two selection queries – one on the `Part` and one on the `Lineitem` table. Each selection query of `Q19` has a multi-column selection predicate on two columns (one using an equality predicate and another one using a between-predicate) and an IN-predicate on one column. Both queries include one join, however the join between the `Part` and `Lineitem` table (`Q19`) is far more costly than joining the `Part` and `Partsupp` table (`Q16`).

Column-Column Comparison Queries: The last group of queries are those that mix column-column comparisons, IN-predicates and multi-column selection predicates. These queries include query `Q4` and `Q12`. While `Q4`'s predicate is a single column-column comparison, in `Q12` two column-column comparisons and a multi-column selection predicate on two columns are executed. Both queries do a subsequent join between the `Lineitem` and `Orders` table. Hence, their overhead on using Elf is to a certain extent comparable – only the effort for the selections should differ.

Query	Number of			example *Sel*	Number of relevant	
	MCSPs	CCs	INs	in %	Joins	Projections
Q6	3	-	-	1.9	-	2
Q17	2	-	-	0.10	2	2
Q16	1	-	1	15.37	1	7[5]
Q19	4	-	2	0.04 & 0.79	1	4
Q4	-	1	-	63.22	2	3[6]
Q12	1	2	1	0.52	1	3

Table 5.1: Query details for our TPC-H queries. MCSPs = involved columns in a multi-column selection predicates, CCs = involved column-column comparisons, IN = involved IN-predicates.

Considering the selectivities, there is a big range of selectivity factors covered. However, we expect higher selectivity factors (e.g., for query `Q4` and `Q16`) to have a higher impact on subsequent operators. The reason is that the more tuples are returned, the more random access is caused. This is even more severe for `Q4` since it operates on the biggest table, the `Lineitem` table.

5.3.2.1 TPC-H Query Runtimes

In Figure 5.8, we show the execution time of our selected six queries. For each query, we show the response time for MonetDB using full-table scans, for the full Elf, Elf_{red} and Elf_{min}. The execution time is further split into four parts:

- the overall time taken for the selections,

[6]This includes an intersection of the join result.

- the overhead for initializing Elf (i.e., the summed up runtime for all commands from Algorithm 9 except `Elf.select`),
- the overhead for the unsorted output of Elf on other operators such as projections and joins (Elf InterOp),
- and the runtime of the remaining operators that are not influenced by the Elf selection.

Of course, Elf InterOp and Elf Initialization is only applicable for the Elf. A first insight is that the general initialization time is not a problem for compute intensive queries.

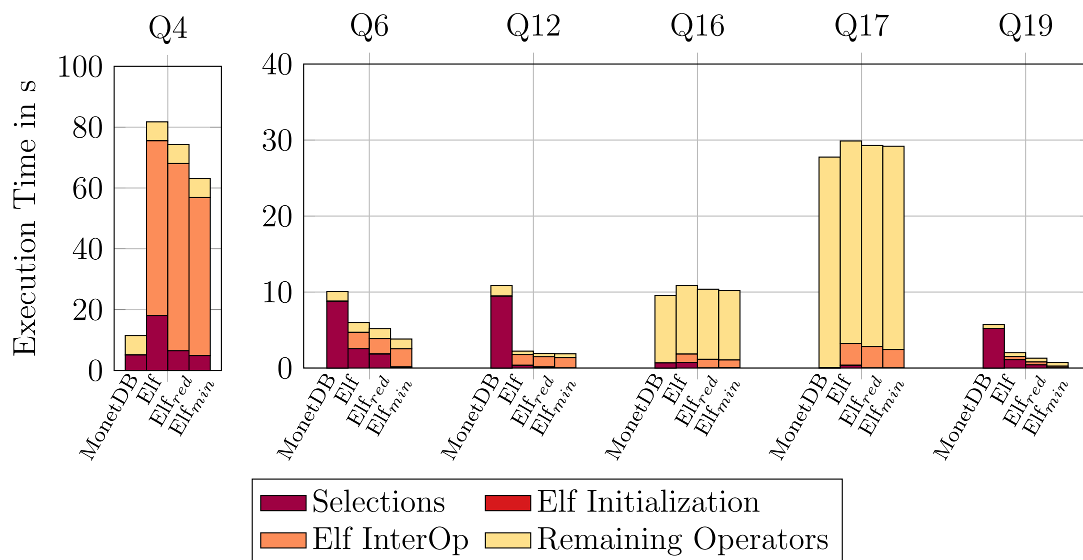

Figure 5.8: Query execution times of Elf and MonetDB's full-table scans for our six selected TPC-H queries ($s = 100$)

The overall results for the queries are mixed. While all Elf use cases outperform MonetDB for the queries `Q6`, `Q12`, and `Q19`, they cannot clearly win for the other queries. The reason for the good performance is the small selectivity factor on the big table `Lineitem` and the small number of subsequent joins and projections. In fact for these queries, using Elf gives a performance improvement for the whole query execution time between a factor of 1.68 (`Q6`) and 5.64 (`Q19`), using Elf_{red} improves by a factor between 1.95 and 5.65, and using Elf_{min} improves by a factor between 2.64 and 5.8. Especially for `Q19`, the overhead for the remaining and affected operators is minimal, which leads to performance improvements similar to the improvement in selection time. However, this does not apply for `Q6` and `Q12`, where the introduced overhead is significantly higher due to the usage of Elf.

For the remaining three queries, we get mixed results compared to the other queries. Hence, we discuss them in detail in the following.

Result for `Q4`

Although `Q4` has a rather expensive column-column comparison, its selectivity factor of 0.63 is far too expensive for using Elf. Especially the full Elf worsens selection

times by a factor of 3.54, while the Elf_{red} and Elf_{min} show comparable selection times. However, the resulting TID list is huge and due to it being unsorted, we face high overheads for the following joins and projections. In fact, 90 % of the execution time is spent due to the random memory access. Hence, this query would have been only beneficial with a reasonable selectivity and, thus, shows the limitation of Elf.

Result for `Q16`

The selectivity factor of `Q16` is with 0.15 in a reasonable range. Hence, we can see that the full Elf has a similar performance to MonetDB's full-table scans and Elf_{red} and Elf_{min} outperform full-table scans by a factor of 35. However, the saved execution speed is not as significant to hide the overhead for Elf's interoperability. Here, especially an additional intersection operation to interoperate with the like-predicate, which is currently evaluated by a full-table scan, is a big problem. However, in the future, we can support a like-predicate by looking up the dictionary and execute an IN-predicate.

Result for `Q17`

The query `Q17` has a similar problem as `Q16`. Although the selectivity factor of 0.001 is very small, full-table scans outperform Elf, because MonetDB can use a hash lookup for the equality predicates including a candidate hash lookup, which is extraordinary fast. Still, Elf_{red} and Elf_{min} can outperform MonetDB's scans by a factor of 377. However, the overhead for using Elf is 420 times higher than its performance benefit.

5.3.2.2 Result Summary

From our integration test, we identified benefits and drawbacks of using Elf in a main-memory column-oriented bulk-processing engine on the example of MonetDB. The positive points are:

Consistent Selection Boost: The improvement in selection performance is consistent to our previous experiments for reasonable selectivity factors. Especially the included selection capability of executing column-column comparisons and IN-predicates in Elf is of utmost importance.

Selectivity-Interoperability Dependency: From the experiment, we can observe that the overhead for subsequent operators is generally related to the selectivity of the selection. When a selection returns only a small amount of tuples, the overhead for materializing these small amount of tuples is minimal. For bigger result sets (cf. `Q4`), the overhead is extra-ordinary big. However, we deem such an approach not suitable for Elf.

Apart from the great results, there are also some serious drawbacks for some queries – especially `Q4, Q16` and `Q17`. These drawbacks are:

Additional Result Merge: Especially when we had to offload selections to MonetDB's full-table scans (e.g., for `Q16`), we had to merge candidates of both selections using a costly intersection. This, however, can be avoided in the future when more operators are implemented in MonetDB's integrated Elf (e.g., like-predicates).

Unordered Output: The result TID list is sorted according to the data, not according to their insertion order. However, most of the subsequent operators favor sequential access, especially for materialization. This is a conceptual problem that has to be addressed either by a reorganization of the underlying table (especially when using Elf as a storage structure), or by a clever declustering approach [MBNK04]. Please mind that it is possible to sort the resulting TID list. However, the sprting overhead on large results is usually more severe than unordered access [Bet18].

Another solution is to incorporate more operators into the Elf, especially projections and joins. On the one hand, to allow for efficient projections, we can change the result of an Elf to one (or more) materialized column(s) that are created during traversal of their values with additional TIDs. These materialized columns can then directly passed to the next operator. On the other hand, a join simply results in a merge-join between sorted `DimensionList`s from two Elfs, which outputs a list of pairs of TIDs.

5.4 Summary

In this chapter, we extended Elf in two important directions. First, we extend Elf to support column-column comparisons and IN-predicates in order to allow for more SQL queries being accelerated by Elf. After a short definition of these complex selection predicate types, we discussed the conceptual idea of evaluating these predicates using Elf and afterwards discuss implementation details to execute these predicates on Elf's hash map, `DimensionList`s, and `MonoList`s.

Second, we investigate whether a holistic query execution would benefit from Elf. This means that not only selection times are improved, but also downstream operators are not hindered too much by Elf. To assess the usefulness in a real system, we integrated Elf into MonetDB as a proof-of-concept. We discussed Elf's interface to MonetDB introducing a set of additional MAL commands. The result of Elf is an unordered BAT of TIDs, which is compatible to MonetDB's current full-table scan results (though the resulting TIDs are usually sorted).

In our evaluation we identified benefits and drawbacks of Elf's column-column comparison and IN-predicate approaches in two microbenchmark. In summary, Elf$_{red}$ and Elf$_{min}$ can outperform full-table scans on a wide range of selectivity factors. Furthermore, the more column-column comparisons the better does Elf scale. Furthermore, Elf$_{min}$ shows almost no impact on the number of executed column-column comparison, which strengthens the efficiency of our designed algorithm. For IN-predicates, we observe that a reasonable selectivity factor for using Elf increases the more IN-predicates are used. In our experiment, the full Elf is still outperformed by MonetDB's full-table scans, but Elf$_{red}$ and Elf$_{min}$ win by several factors. Hence,

we deem our support of additional selection operators successful (cf. RQ 4), because we exploited Elf's design for a scalable execution of complex predicates.

In our last experiment, we then examined the overhead for using Elf for *full queries* in a full-fledged main-memory DBMS – MonetDB. Usually, one would expect that improving the selection times also improves query performance in the same manner. To validate this expectation, we selected six representative queries from the TPC-H benchmark that should challenge Elf, but also benefit from Elf. In fact, most of the selections got a reasonable boost by Elf, especially when using Elf_{red} and Elf_{min}. However, the interoperability between Elf's selections and subsequent operators is still an open question, because the introduced overhead overshadows the good selection performance for some queries. From our experiments, we identified that the underlying reason is the unordered output of Elf, which causes cache misses for subsequent operators. Still, we deem this goal as successful, because the performance of TPC-H queries is in the same order of magnitude and, hence, we answered research question RQ 5 with the outcome of this experiment.

The performance penalties of subsequent operators lead to the ultimate goal of equipping Elf with more operator implementations to exploit prefix-redundancies also for joins and projections and to minimize the interoperability overhead. Furthermore, by storing tuples in accordance to Elf's tuple order would minimize this overhead as well, calling for an (own) adapted storage and query engine for Elf.

6. Elf Life Cycle

The primary application field of Elf are data warehousing scenarios and decision support systems. Although hybrid transaction/analytical processing has earned much of attention recently [PFRE14, KN11, APM16, PBDS17], the usual use case of Elf are still read-mostly workloads with periodic insertions of new tuples (e.g., over night). To this end, we need to support *initial build* and *periodic insertions* efficiently.

In this chapter, we give technical details on how Elf supports initial building by means of multi-dimensional sorting (cf. Section 6.1). In addition, we explain how Elf supports periodic insertions by means of a Elf-like delta store (cf. Section 6.2), which reduces this task to merging of pre-sorted lists. Finally, we outline how Elf handles updates and deletions.

In our evaluation of this chapter, we first compare the initial build time of Elf against its competitors from Chapter 4. This experiment gives insights whether high build times are a counter argument for using Elf. Afterwards, we evaluate the efficiency of our second important use case – supporting periodic insertions – and also determine the best point to consolidate the Elfs. As a consequence, we complement the good query performance of Elf as the goal of Level 2 of this thesis with an ability to support OLAP in data-warehouses-like scenarios. As a consequence, we answer research question RQ 6 in this chapter.

In summary, we make the following contributions:

- Concept for efficient maintenance of Elfs including initial builds, insertions, updates, deletions
- Build time comparison between Elf and competitors
- Evaluation of the overhead for delta-store-based insertion handling
- Determination of a reasonable threshold for handling insertions in a delta-store-like Elf

```
1  BuildDimList(data[][], col, start, num, writePointer){
     // (1) incremental sort w.r.t. a given column
2    sort(data[start], num, col);
     // (2) determine all values and store position of their
        pointers
3    pf ← new list();              // of 2-tuples (position, frequency)
4    cur ← data[start][col];              // smallest value in this col
5    for (i ← start + 1 to start + num)do
6      if (cur != data[i][col])then
7        Elf[writePointer] ← cur;                  // write this value
8        pf.add(writePointer,1);            // (position, frequency)
9        writePointer+=2;                          // DIM_Element size
10       cur ← data[i][col];
11     else
12       pf.last.freq++;
13     end if
14   end for
15   setMSB(Elf[writePointer-3]);                  // End of DimList
     // (3) write pointers and interate to next col
16   if (col + 1 < NUM_COL)then
       // not in last column
17     offset ← start;
18     for (i ← 0 to pf.size)do
19       Elf[pf[j].pos] ← writePointer;        // pointer to begin of next
          DimList
20       if (hasFanOut(data[offset],col,pf[j].freq))then
21         writePointer ← BuildDimList(data, col+1, offset,
            pf[j].size,writePointer);
22       else
23         setMSB(Elf[pf[j].pos]);                 // mark as monolist
24         for (curCol ← col to NUM_COL)do
25           Elf[writePointer] ← data[offset][curCol];   // write values
26           writePointer++;
27         end for
28         for (curTuple ← 0 to pf[j].freq)do
29           Elf[writePointer] ← data[offset][NUM_COL]; // write TIDs
30           writePointer++;
31         end for
32         setMSB(Elf[writePointer-1]);            // mark as last TID
33       end if
34       offset ← offset+pf[j].freq;
35     end for
36   else
       // in last column:  write all TIDs (cf. Line 22-33)
37   end if
38 }
```

Algorithm 12: Building an Elf

6.1 Initial Build: Elf Bulk Load

The initial build of Elf is executed as a *bulk load*, where all data of the table is read to create the Elf with its explicit memory layout as shown in Algorithm 12. The build procedure consists of a step-wise multi-dimensional sort paired with a build of all `DimensionLists` of the currently sorted column.

The build is invoked as: `BuildDimList`$(data,\ col = 0,\ start = 0,\ num = data.size,\ writePointer = 0)$, where data is a two-dimensional array containing all tuples plus the TID at the end (at position `NUM_COL`). The general approach consists of sorting[1] the data for the current dimension, bringing all values with the same prefix next to each other, and then finding all partitions that belong to a single `DimensionList`. For each resulting partition, we call the build function recursively.

For each call of `BuildDimList` from Algorithm 12, we know that all points between $data[start]$ and $data[start + num]$ (i.e., the current partition) refer to the current `DimensionList` and are already sorted according to the prefix until column $col - 1$. The algorithm then additionally sorts these points according to the current column col (Line 2). Next, we iterate over the current partition twice: first for finding all distinct values (the common prefixes) and, second, for building the `DimensionLists` of each sub-partition. Note that we need two steps because we do not know the exact number of distinct values in the current sub-partition.

In the first iteration, all existing values within the current `DimensionList` are linearized starting with the smallest one. Since we do not know where the corresponding sub tree (i.e., the next Elf level) will start, we store the position where this pointer is located and count how many points refer to this sub tree in an auxiliary structure pf (cf. Line 3-15).

In the second iteration, the algorithm linearizes the corresponding sub tree of the first `DimensionElement` entirely, before it moves on to the next. In case the current partition can be split further, a recursive call is executed (Line 21). Otherwise, we create a `MonoList` by storing first all remaining values of the next columns (cf. Line 24-27) and afterwards storing a number of TIDs (cf. Line 28-31). Notably, this case also holds for duplicates in the data which leads to a list of TIDs being stored. In this case, only the last TID's MSB is set.

This sort-based algorithm proves to be superior to a hash-based alternative (cf. our technical report [KBSS15]). As build times are an important factor of the practicality of Elf, we compare the build times of all competitors in detail in Section 6.3.1.

6.2 Maintaining an Elf

Due to Elf's explicit memory layout, maintenance (i.e., insert, update, and delete) after its initial built is not trivial (a typical RUM tradeoff [AKM+16]), but it is still possible. Since Elf is designed for analytical scenarios, supporting periodic inserts of new data, such as weekly or daily inserts, are most important and are explained in the following.

[1] Our sorting routine is adapted from the MPSM join [AKN12].

6.2.1 Insertions

Our solution for periodic inserts consists of two parts. First, new data is collected in an auxiliary data structure named InsertElf. It has the same conceptual design as a normal Elf without the explicit memory layout and `MonoLists`. In Figure 6.1, we show an exemplary InsertElf for a table with 5 columns. When inserting new tuples, we can simply extend the corresponding `DimensionLists`, which would not be possible with the explicit memory layout due to its tight packing. Also, we do not create `MonoLists`, although `DimensionList` 4, 5, 9, and 10 would benefit from this w.r.t. query performance. However, since the main objective is insertion performance, a non-`MonoList` design is preferable since they will probably be split anyway when more data is added.

Our described idea is similar to delta stores in columnar databases [ZAP+16]: there is one write-optimized InsertElf and one linearized read-optimized Elf. Usually, the write-optimized InsertElf is by several factors bigger than an analogous read-optimized Elf. This is similar to row stores having worse compression capabilities and, hence, cause an overhead. Hence, when a specific threshold of insertions is reached, data is transferred from the InsertElf to Elf. This transfer is simply a merge of both structures, which is conducted as follows.

Merging InsertElf and Linearized Elf

By concept, Elf introduces a total order into the multi-dimensional data space. As the read-optimized Elf and its write-optimized counterpart imply the same order, we can exploit this for reducing the problem of merging two Elfs to the problem of merging pre-sorted lists. Therefore, the merge algorithm works at `DimensionList` level (cf. Algorithm 13) and is highly similar to merging two sorted lists of elements. The algorithm starts at the first element of the root `DimensionList` of both Elfs in order to merge both roots (i.e., `DimensionList`). To merge two `DimensionLists`, the algorithm first compares the values of the first `DimensionElements` differentiating three cases:

1. If the value of the linearized Elf is smaller, the common prefix ends here. Hence, the sub tree of the linearized Elf is copied into the new Elf without changes (Line 5).
2. If the value of the InsertElf is smaller, there is an insertion of new data to be done. In this case, the whole sub tree of the InsertElf is linearized into the new Elf (Line 8).
3. If the value in the InsertElf and the linearized Elf is the same, the prefix redundancy is further exploited. This leads to a subsequent merge of the underlying `DimensionList` of the InsertElf and the linearized Elf (Line 12).

After comparing the first two elements, the algorithm increments the smaller position in the two `DimensionLists` in order to compare and merge the next values, until the end of one of the lists is reached. Due to the sorting criteria of both structures, we can efficiently combine both structures with a complexity of $O(\text{Elf}_{size}+\text{InsertElf}_{size})$.

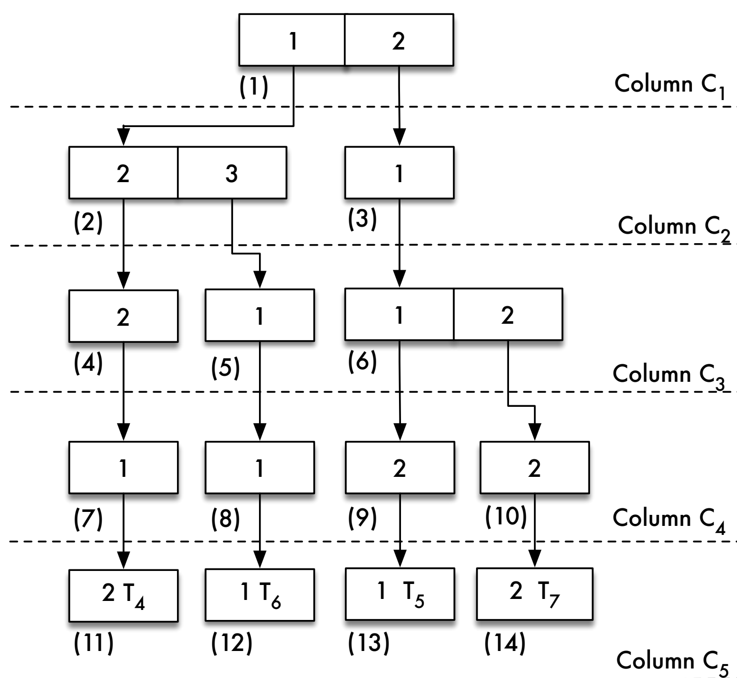

Figure 6.1: InsertElf for a 5-dimensional data set

```
1  MergeDimLists(toInsertDimList, position, newElf, writePointer){
2      iElfPos ← 0;
3      while (notEndOfList (Elf[position]) ∧ notEndOfList
        (toInsertDimList[iElfPos]))do
4          if (Elf[position] < toInsertDimList[iElfPos])then
5              writePointer ← copySubTree(Elf[position+1],newElf,writePointer);
6              position ← position + 2;
7          else if (Elf[position] > toInsertDimList[iElfPos])then
8              writePointer ←
                linearizeDimLists(toInsertDimList[position].child(),
                newElf, writePointer);
9              iElfPos ← iElfPos + 1;
10         end if
11         else
12             writePointer ← MergeDimLists(toInsertDimList[position].child(),
                Elf[position + 1], newElf, writePointer);
13             position ← position + 2;
14             iElfPos ← iElfPos + 1;
15         end if
16     end while
       // Process remaining entries of longer DimList
17 }
```

Algorithm 13: Merge a linearized DimensionList within a DimensionList of the InsertElf

However, even if the InsertElf is by several orders of magnitude smaller than the read-optimized Elf, there is still some performance loss on query execution to be expected. We quantify this performance loss in Section 6.3.2 considering different sizes of the InsertElf. Moreover, for any approach relying on a combination of read-optimized and write-optimized structures, such as delta stores, solving the problem when to merge both structures is important. To this end, we also conduct experiments in Section 6.3.3 to answer the question when to merge both Elfs.

6.2.2 Deletion

For deletion, we first perform a lookup for the data item we want to delete. Due to the prefix-redundancy elimination and our `MonoList` optimization, we have to traverse down until we find the corresponding `MonoList` for the tuple. Notably, deletions only impact the path starting from the last value before the corresponding `MonoList`, because its prefix still represents values of several data items and has to persist. Now, there are two cases: the tuple can be a duplicate or not.

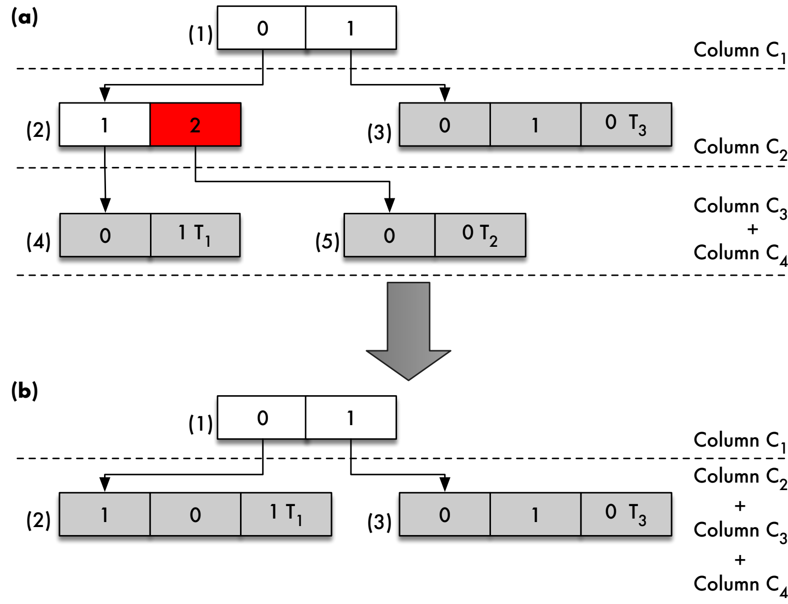

Figure 6.2: (a) Elf with marked tuple to be deleted, (b) reorganized Elf

In case, we delete a duplicate data item, we just remove the *TID* in the list of *TID*s. Otherwise, we have to delete the `MonoList` and update the parent `DimensionList`[2]. Assume, we want to delete data item T_2 from Figure 6.2. We know that in `DimensionList` (2) its `MonoList` starts. Its parent `DimensionList`, however, has only two values before the deletion. Hence, we use the freed space to transform `DimensionList` (2) to a `MonoList`, because after deletion there is no branch out anymore in `DimensionList` (2). Please mind that this is only possible due to the preorder linearization of our Elf. Otherwise, in the case there is more than one value left in the last `DimensionList` and we delete any value but the last, we just have

[2] It is also possible to just invalidate the pointer to that `MonoList` using a pre-defined tombstone [RAD15], but this is neither space efficient nor efficient for traversals.

to shift all values and pointers of the `DimensionList` to the left. In case the last value of the `DimensionList` has to be deleted, we just have to set the MSB of the before-last value to indicate the end of the list.

Overall, the delete procedure creates unused gaps in our tightly packed Elf for each deleted `MonoList`. Notably, the earlier the corresponding `MonoList` starts, the more space is left unused. Hence, cleaning up the resulting structure becomes an essential part to assure competitiveness. Such a clean up should be coupled with the merge in Algorithm 13, as it traverses and rearranges the whole Elf anyway.

6.2.3 Updates

Finally, updates are rare for analytical workloads, but possible within Elf. Generally, there is a large amount of `MonoLists` (cf. Section 4.3.2). Updating a value in a `MonoList` does not result in any problem, as we just have to write the new value to the correct position. This is possible since only changes on `DimensionLists` will change the position (or path) of the tuple in the Elf. In the case the update is located in a `DimensionList`, an update is composed of a delete and an insert as described above.

6.3 Evaluation

In this section, we evaluate the performance of the aforementioned algorithms. At first, we investigate the initial build time in Experiment 1. Furthermore, as already mentioned, the explicit memory layout of Elf is an accelerator for query performance, but a burden for update intensive workloads. This is because an easy enlargement of `DimensionLists` is hardly possible in the tightly-packed cache-efficient memory layout. Hence, we adapt the delta-store idea of Zhang et al. [ZAP+16] by building a separate InsertElf that indexes newly inserted data. However, this approach causes an overhead every time we execute a query. Hence, we also investigate its usefulness in Experiment 2 and 3. Overall, we answer the following research questions in this evaluation:

1. What is the overhead of our bulk-load algorithm compared to the build times of our competitors? Since too high build times would limit the usefulness of Elf even for analytical-only scenarios, we require that building has a significantly smaller overhead than Elf's query performance benefits (which are up to several magnitudes).

2. What is the selection time overhead caused by querying both Elfs based on the fraction of tuples stored in the InsertElf (Experiment 2)?
   Answering this question indicates the efficiency of the overall solution, suggests reasonable sizes for the InsertElf, and is a first indicator when to merge both Elfs.

3. When is a merge more cost-efficient than querying both structures considering the number of executed queries and the number of tuples in the InsertElf (Experiment 3)?
   In fact, we have observed performance differences between the InsertElf and linearized Elf of around a magnitude. Hence, determining the point in time when a merge is necessary is an important task.

6.3.1 Experiment 1: Build Times

The purpose of the build time examination is to evaluate whether a large build time is a counterargument for the applicability of Elf. In this experiment, we built all index structures and accelerated scans over the whole `Lineitem` table of scale factor $s = 200$ including all its columns. For each measurement, we build the corresponding index structures 10 times and compute the mean value.

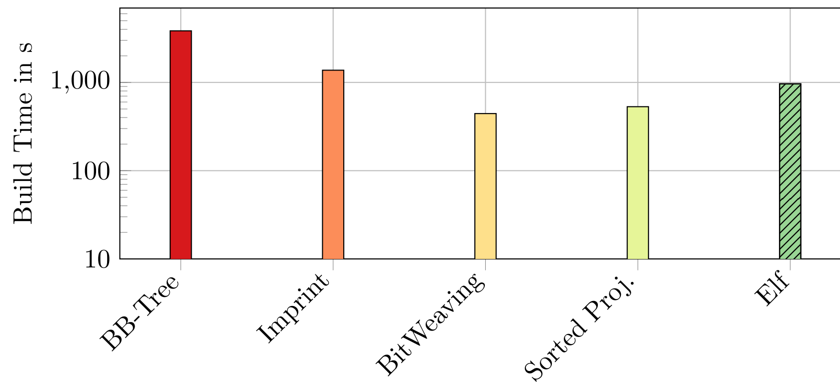

Figure 6.3: Build time for `Lineitem` table of $s = 200$

In Figure 6.3, we visualize the build times for the `Lineitem` table of $s = 200$. Overall, all approaches need at least several minutes up to 50 minutes for finishing the build. The fastest approach is BitWeaving, with a build time of 443.71 s. This is needed to compress all columns using its bit packing technique. The next fastest appraoch is Sorted Projection requiring 530.93 s, which is quite similar to the build time of BitWeaving. Here, Sorted Projection has the advantage to execute a single sort for the first dimension. In contrast, Elf executing a full multi-dimensional sort has even less than double of the time of Sorted Projection. Elf's build only takes 964.17 s. The reason for its good build times is that in the deeper levels of the Elf, we often just linearize `MonoLists`, which is less expensive than linearizing a hierarchy of `DimensionLists`. Therefore, we argue that build times are no counterargument for the applicability of our approach, especially as we reach a speedup of one order of magnitude for query performance. For instance, in analytic environments, such an additional build time is acceptable. Note, Column Imprints and BB-Trees require more build time than Elf. Here, especially the BB-Tree suffers from its sampling-based column ordering and high overhead in data shuffling [SSL18a, SSL19].

6.3.2 Experiment 2: Query Overhead of InsertElf

Considering the design of the InsertElf, we assume that for small data the overhead can hardly be measured. Hence, our approach would be well suited for read-intensive OLAP scenarios with (relatively) small periodic insertions. We deem our solution efficient in case a fraction of up to 0.1 % introduces an overhead of less than 10% selection time increase. To put these numbers into context, consider that with scale factor $s = 200$, the `Lineitem` table has 1,200 million tuples. Thus, 1 % means 12 million newly inserted tuples. To examine efficiency, we distribute the tuples of the `Lineitem` table between the read-optimized Elf and InsertElf using different tuple

distribution factors r. We execute the queries on both structures meaning that the selection time is the *sum* of individual selection times. Generally, one can execute the query on both Elfs in parallel and the overall selection time would be the *maximum* of the individual response times. However, further experiments suggest that an inter-query parallelism concept should be preferred.

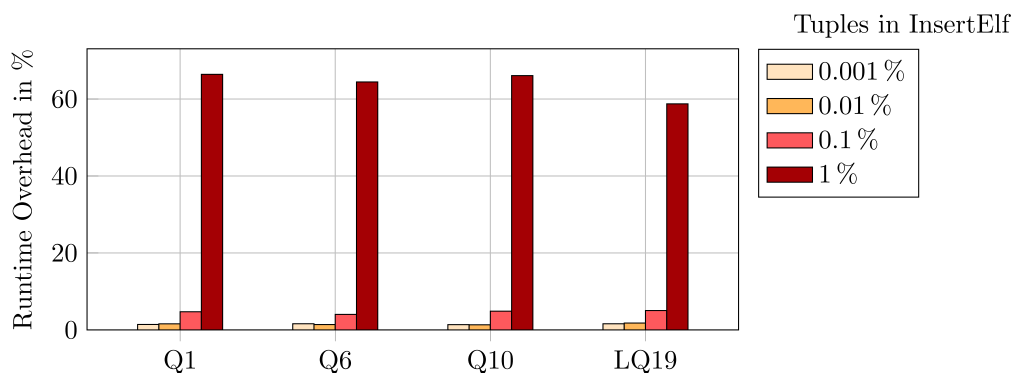

Figure 6.4: Normalized runtime overhead caused by different InsertElf sizes in the TPC-H queries on the `Lineitem` table ($s = 200$)

In Figure 6.4, we show the overhead for querying both structures depicting the average of 1,000 measurements. We load $r\,\%$ (with $r \in 0.001, 0.01, 0.1, 1$) of the data into the InsertElf and the remaining $1-r\,\%$ into the read-optimized Elf. In Figure 6.4, we observe that the overhead is scaling similar to the ratio of indexed tuples by the InsertElf. Thus, for reasonable amounts of data, the overhead can be neglected. That is for $r \leq 0.01$ the overhead is hardly measurable. Furthermore, we consider an overhead of about $5\,\%$ per query for $r = 0.1\,\%$ as acceptable stating the efficiency of the periodic insert mechanism. Nevertheless, the high overhead for $r \geq 1\,\%$ makes a merge inevitable.

In summary, the overhead for querying both structures is remarkably small for a small amount of appended data. Hence, we argue for the usefulness of the InsertElf in OLAP scenarios with a high frequency of updates.

6.3.3 Experiment 3: Merge Threshold

To determine when to best merge the InsertElf into the linearized Elf, we have to examine when the threshold of all executed queries is high enough that querying both structures is less efficient than merging them and querying the resulting Elf. In Figure 6.5, we show the runtime of executing a merge and a multiple of the queries (Q1, Q6, Q10, LQ19) compared to executing these queries on both, the linearized Elf and InsertElf, w.r.t. different ratios r.

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

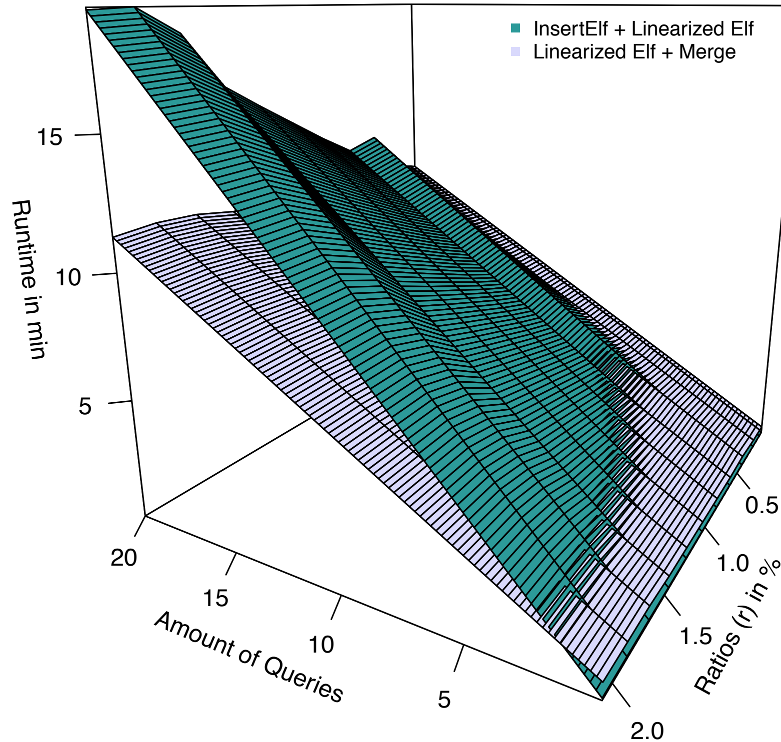

Figure 6.5: Accumulated runtimes for the four TPC-H `Lineitem` queries ($s = 200$) on a merged Elf (including merge time) and the sum of runtimes of a linearized Elf and InsertElf w.r.t. different InsertElf-to-linearized-Elf ratios

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

6.4 Summary

In this chapter, we extended the applicability of Elf for workloads that include maintenance tasks such as inserts, updates and deletes of data. To this end, we first explained the build of Elf implementing a bulk insert. Afterwards, we proposed our approach for handling periodic inserts to the initially built Elf. Our approach for handling periodic inserts is similar to a delta store in modern main-memory database systems, where there is one read-optimized and one write-optimized Elf version. Furthermore, we outlined how to implement updates and deletions. With the algorithms in this chapter, we took a big step towards using Elf as the only storage structure of the system.

In our evaluation, we analyzed the performance for the main use case of Elf – initial build times and the performance implications of our write-optimized InsertElf compared to standalone linearized Elf. In these experiments, we have shown that the Elf is built within factor 2-3 compared to its fastest competitor (BitWeaving). However, given that Elf outperforms its competitors by the order of up to one

magnitude, we argue for its usefulness. Furthermore, for reasonable ratios of inserted data, the overhead of our write-optimized InsertElf is manageable (5 % performance overhead for indexing additional 0.1 % of data). However, our last experiment shows that a merge is inevitable when more than 2 % of the data is stored in the write-optimized InsertElf due to its high impact on query performance. With these considerations, we argue that using Elf for data-warehouse-like workloads with frequent appends is reasonable, which is the answer to our research question RQ 6.

7. Related Work

In this section, we highlight important related work in the field of main-memory indexing and multi-dimensional indexes. At first, we explain the chosen competitors that are compared against Elf in our evaluation, because their characteristics are important to assess their strength and weaknesses compared to Elf. Afterwards, we introduce the Data Dwarf, because its concept of prefix-redundancy elimination and its tree structure are the key ideas for our Elf. In Section 7.3, we present SIMD-accelerated one-dimensional index structures for main-memory database systems, because they also focus on cache-efficiency and adaptations for modern hardware. These adaptations are important optimizations that could also be applied to Elf in the future and, hence, we give a more detailed explanation of these related structures compared to the following ones. In the last two sections, we review recent one-dimensional and multi-dimensional indexing approaches for main-memory databases.

7.1 Competitors

In the following, we introduce our selected state-of-the-art competitors to speed up column scans and range queries in OLAP environments. In particular, we explain BitWeaving [LP13] as an approach to utilize bit-parallelism for bit-packed data. Additionally, we describe Column Imprints [SK13], a secondary index structure, which uses a 64-bit representation of a histogram to exclude whole cache-lines of data from the search space that do not qualify for the query. For both approaches it has been shown that they outperform prior state-of-the-art approaches. As multi-dimensional competitors, we describe Sorted Projections [SAB$^+$05] and the recently proposed BB-Tree [SSL18a, SSL19].

7.1.1 BitWeaving

BitWeaving is a bit-packing technique proposed by Li and Patel [LP13]. The idea of BitWeaving is to store the necessary bits (w.r.t. the given value range) of several values into one processor word (with a typical size of 64 bit). Therefore, BitWeaving adapts

the idea of SIMD even for scalar registers to exploit data parallelism in computation. BitWeaving can either be used as a secondary index structure or a compression technique with improved scan performance. However, when using BitWeaving as the physical data layout, we have to deal with heavy-weight packing and unpacking of data, especially when accessing concrete data. Thus, in the context of this thesis, we look at BitWeaving as an index structure. BitWeaving offers two different methods to pack values into processor *words*: BitWeaving/H and BitWeaving/V. To facilitate the understanding, we present a small comparison between both layouts in Figure 7.1, where each value can be expressed with a 3-bit code.

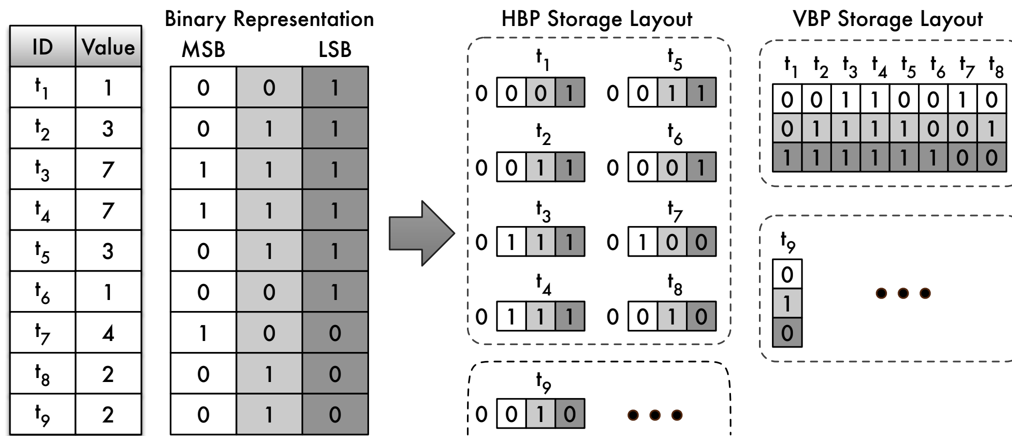

Figure 7.1: BitWeaving approach – adapted from [LP13]

Horizontal Bit-Parallel

Assuming that we can describe each value with k bits, the horizontal bit-parallel storage layout constructs a *code* of length $(k+1)$ bits, where the k bits of the word have to be extended with a leading 0. Several of these codes, exactly $\lfloor \frac{w}{k+1} \rfloor$ with a processor word length of w, are then consecutively packed together starting at the right word boundary (remaining bits are filled with zeros) [LP13]. In Figure 7.1, two words including the leading zero fill an 8-bit word. However, the physical data layout is rearranged such that the first four values are stored in the upper part of the words and the second four values are stored in the lower part of the words. This rearrangement is essential to reach peak scan performance. The idea of a scan over the horizontal bit-parallel (HBP) storage layout is that we use simple arithmetic functions to create a bit mask, which indicates whether the values satisfy the condition. The result bits of the comparison are stored in the added zero of each word (for the corresponding arithmetic functions for any comparison operator, we refer to the original sources [Lam75, LP13]). Now, by shifting each word an additional time to its predecessor and by doing an AND operation of the words, we can construct the resulting bit mask. Although the HBP storage layout allows to process several values in parallel instead of just one, it still has to process the whole word. To overcome this, the vertical bit-parallel (VBP) storage layout packs the bits of several words vertically.

Vertical Bit-Parallel

The vertical bit-parallel storage layout partitions one value across several processor words. For illustration, the binary representation can be seen as a set of columns of a table, which are then stored in a column store [LP13]. As a consequence, the ith bit of w values are stored consecutively in a processor word of length w (e.g., the most significant bit of 8 values in one word in Figure 7.1). So, we construct k words that can be independently processed. Although, packing and unpacking of values for this storage layout is expensive [PR15], scanning values in a vertical bit-parallel storage layout offers several benefits. First, it is possible to do an early pruning of the search. Since we start with comparing the most significant bits of the codes, we are able to deduce the result of the comparison from the first bits that differ in the comparison. If the results of all codes of a processor word are determined, we can stop to evaluate these codes and, thus, save computational effort.

Overall, the results of Li and Patel [LP13] and our own initial experiments suggest that BitWeaving/V is by far the superior approach. Thus, we only report performance results for BitWeaving/V in this thesis.

ID	Value	Bitmap 0	1	2	3	4	5	6	7	Column Imprints
t_1	1	0	1	0	0	0	0	0	0	
t_2	3	0	0	0	1	0	0	0	0	01010001
t_3	7	0	0	0	0	0	0	0	1	
t_4	7	0	0	0	0	0	0	0	1	
t_5	3	0	0	0	1	0	0	0	0	01010001
t_6	1	0	1	0	0	0	0	0	0	
t_7	4	0	0	0	0	1	0	0	0	
t_8	2	0	0	1	0	0	0	0	0	00101000
t_9	2	0	0	1	0	0	0	0	0	

Cache Line Dictionary

Counter	Repeat
2	1
1	0

Figure 7.2: Column Imprints – adapted from [SK13]

7.1.2 Column Imprint

A Column Imprint is a cache-conscious secondary index structure for range queries implemented in MonetDB [SK13]. The idea is to apply a coarse-grained filter (similar to bloom filters [Blo70]) indicating whether we can exclude a complete cache line for a given query. To this end, the Column Imprint builds a histogram over all values of a cache line and stores it in a 64-bit integer. The histogram is an equi-width histogram with 64 bins where a bit $b = 1$ means that at least one value of the corresponding cache line is in the range of the given bin. Notably, the first and the last bin hold values from $-\infty$ to the current smallest value and from the highest value to ∞, respectively.

As an example, we show the bitmap and the Column Imprint for our running example in Figure 7.2. Assume that our values range from 0 to 7 and we use an 8-bit value to

represent the histogram. In this case, we have a direct mapping between the values and bins of the histogram (although in practice, each bin corresponds to a range of values). Furthermore, if a cache line could only hold 3 values, then our imprint indexes the first three values and is constructed by applying the logical `AND` of the bitmaps of the corresponding values.

A further optimization that Sidirourgos and Kersten apply is that if two succeeding imprints are the same, they are compressed using a cache line dictionary [SK13]. Consequently, we only store and scan these repeated cache lines once.

To evaluate a selection predicate on a Column Imprint, a bit mask is created where all the bits are set that match the predicate (probably several bits for a range query). If the result of the logical `and` between the bit mask and the current imprint leaves any bit set, then there is at least one qualifying tuple and the cache line has to be consulted for filtering out false positives. However, we can skip this filtering step, if the range predicate and the bin borders totally align, which implies that all values of the cache line are included in the selection.

In our evaluation, we use a scalar version of Column Imprint that we extracted from MonetDB. Recently, Sidirourgos and Mühleisen extended Column Imprint to bigger imprints and encoded data blocks by using SIMD [SM17]. They show that when increasing the size of the imprint vectors, the number of encoded values (formerly the size of a cache line) should increase as well such that the ratio between imprint and number of encoded values is 8:1.

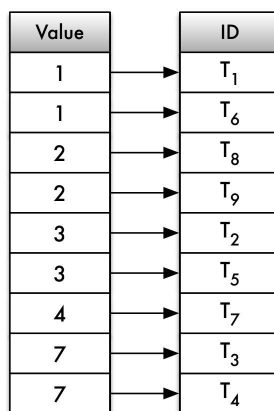

Figure 7.3: Sorted Projections

7.1.3 Sorted Projection

C-Store [SAB+05] proposes the concept of projections as an additional index structure. A projection is defined on a set of columns with a predefined order. With respect to this order, column data are replicated and sorted according to the first column in this order. This sorting enables an efficient binary search on the first column. However, it also alters the position of a tuple in the column that creates the need for an additional TID column (cf. Figure 7.3) which is implicitly encoded by the tuple position before. Hence, it creates additional storage overhead. The overhead can be reduced by using run-length encoding on the first column being effective due to the

sorting criteria. Note, in our example in Figure 7.3 the run-length encoding does not gain any compression. This is due to the fact of the very small data size.

Sorted Projections are very efficient if the query workload is known. Then, the minimal set of necessary projections can be determined to accelerate query execution. However, for frequent updates the more projections are created, the more update propagation has to be done.

7.1.4 BB-Tree

Due to the increasing importance of accelerated scans, employing them in a multi-dimensional index structure has become a reasonable idea. To this end, BB-Tree has been proposed, which is a combination of a kd-tree (*inner search tree*) that partitions the space into buckets (*bubble buckets*) that are subsequently scanned by a full-table scan (cf. Figure 7.4). In that sense, the BB-Tree uses similar concepts like X-Trees, where degenerating nodes form so-called *super nodes* that are scanned linearly [BKK96].

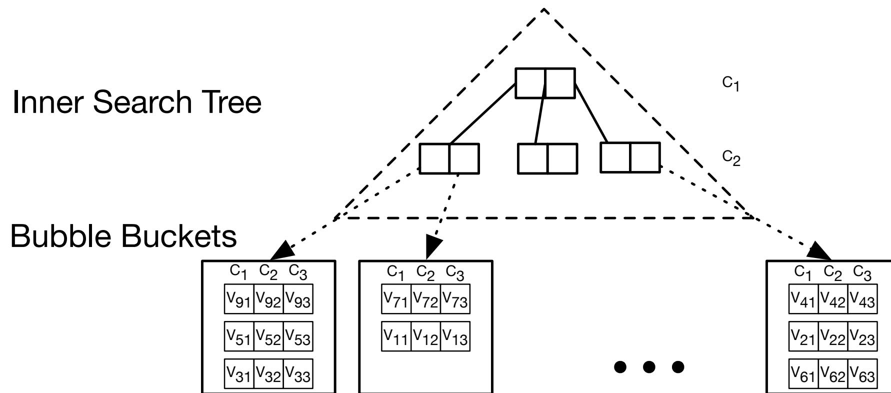

Figure 7.4: BB-Tree Structure

Inner Search Tree

The inner search tree is an array that linearizes the k-ary nodes of the kd-tree. Node entries are stored in a sorted manner to allow for binary search. In fact, the k node entries split the space on the given column C_1 into $k+1$ sub spaces. Each of the subspaces is then partitioned in the next level C_2 into $k+1$ further subspaces. This procedure is (as for usual kd-trees) executed in a cyclic manner until a certain depth of the tree. The columns for each level are chosen according to their cardinality of different values as presumably, higher cardinality columns lead to a more efficient pruning. From our Elf, we have seen that the pruning power depends on the usage of the columns in the query (e.g., comparing Elf and Elf_{min}) and the same also applies for the column order of the BB-Tree. Nevertheless, this procedure has shown effective for the GMRQ benchmark [SSL18b].

Bubble Buckets

Due to the incomplete splitting of the space into subspaces up to a certain depth of the inner search tree, there are several data points left in a subspace. Each of

these subspaces is stored in a so-called bubble bucket, which is traversed sequentially during querying. Although, this effectively reduces the number of possible candidates, all columns C_1–C_3 of the candidates have to be compared to the query again due to the incomplete splitting (in contrast to Elf's complete space partitioning).

An important optimization is that if buckets overflow, due to further insertions, a bubble bucket can be replaced with another inner search tree with subsequent bubble buckets. Hence, even on insertion, the BB-Tree performs well. However, if a bucket of the second level overflows, the tree is rebuilt in order to not degenerate further.

7.2 Data Redundancy Elimination

In this section, we introduce the basic concepts of the Data Dwarf because some of its concepts are the basis for our Elf. Hence, we give a detailed view on the use case and also structure of the Data Dwarf. For a better understandability, we use the dataset from Figure 7.5 to build and explain the Data Dwarf approach.

Example Table

In our example table (left side), the first dimension consists of two distinct values, the second dimension's value domain has three distinct values, and the third dimension consists again of only two different values. Note, we use a dense value domain for each dimension in this example. This could be easily obtained from any data set by applying an order-preserving dictionary encoding (for further details we refer to the work of Gennady Antoshenkov [Ant97]).

Table

	Dim_1	Dim_2	Dim_3	Fact
T_1	0	1	1	1
T_2	0	2	0	1
T_3	1	0	1	2

Cube

Dim_1	Dim_2	Dim_3	SUM(Fact)
0	1	1	1
0	2	0	1
1	0	1	2
ALL	1	1	1
ALL	2	0	1
ALL	0	1	2
⋮			
0	ALL	ALL	2
1	ALL	ALL	2
ALL	ALL	ALL	4

Figure 7.5: An exemplary table and an excerpt of its cube

Data Cube

Besides computation of aggregates on the fly, a pre-computation is possible in case sufficient storage is available. In a data warehouse, the corresponding data structure is

called cube, which is a materialization of the cube operator [KSS14]. In Figure 7.5, we show the result of the example relation for the cube operator. Note, all intermediate aggregations are stored and we use the ALL representation in our depiction in case a dimension is summarized to the top node.

Since the data cube causes a lot of redundancy due to the aggregation and denormalization, its efficient storage is an important problem. To solve this problem, Sismanis et al. introduced the Data Dwarf. In the following, we first explain the special properties of the Data Dwarf that we also exploit for our Elf and, afterwards, we introduce the conceptual tree structure of the Data Dwarf.

7.2.1 Prefix and Suffix-Redundancy Elimination

An interesting concept that has been introduced by the Data Dwarf is prefix-redundancy and suffix-redundancy elimination [SDRK02]. It is an essential part of the Data Dwarf, which is designed to compress the data of a materialized cube operator. Hence, besides actual dimension values, it also stores the aggregates of all dimension key combinations (cf. Figure 7.5), allowing for efficient aggregate queries.

Prefix Redundancy: Prefix redundancies occur whenever two or more dimension keys share a common prefix. This is visible in the left table in Figure 7.5, where tuple $\texttt{T}_1$ and $\texttt{T}_2$ share the same value in the first dimension.

Suffix Redundancy: Suffix redundancies are caused by creating the cube. If an aggregate has only one value to take into consideration (e.g., the aggregate ALL, ALL, 0, which only consists of T_2), then it is the same aggregate as T_2 itself. Hence, there is a suffix redundancy, which could be avoided for this aggregate.

In the examples of Sismanis et al. [SDRK02], they could shrink a 1 PB cube to a 2.3 GB Data Dwarf. These high compression rates come mainly from avoiding redundancies of the cube entries (cf. Figure 7.6). Since the Data Dwarf avoids both of these redundancies, it has, on the one hand, improved storage consumption. On the other hand, also the scan performance is increased, because compared to a full-table scan, the Data Dwarf only has to descend d nodes in a d-dimensional Data Dwarf until it finds a matching tuple.

In the following, we present the overall structure of the Data Dwarf, because its prefix-redundancy elimination and the overall Data Dwarf structure is an interesting starting point for our Elf structure.

7.2.2 The Data Dwarf Structure

The Data Dwarf has one root node containing all distinct values of the first dimension in an ordered fashion. Moreover, it contains pointers to the nodes of the next level, indicating a *path*. A path is the implementation of prefix-redundancy elimination. It defines a way from the root node to the leaf nodes representing the dimension value of an existing point of the data set. To store a d-dimensional cube, we need to create a Data Dwarf with d levels: one for each dimension. On the final level, a leaf node

stores entries that contain the last dimension value, as well as the aggregate for the given tuple. All entries in a Data Dwarf node are ordered according to the value in the dimension (similar to B-Tree nodes). Furthermore, the Data Dwarf introduces one additional entry per node, which is the "ALL" node that corresponds to the aggregate of all values of this node. We visualize an example of the constructed Data Dwarf in Figure 7.6 for the cube of Figure 7.5. Note, "ALL" nodes may reference parts in a different path implementing the suffix-redundancy elimination. To this end, the Data Dwarf conceptually is a *directed acyclic graph (DAG)* but not a tree.

Due to this structure, it is particularly hard to optimize Data Dwarf for efficient main-memory access. In addition, it has been shown that the storage reduction is rarely reached in practice [DBS08]. To this end, we focus in the remainder only on some properties of the Data Dwarf that are essential for the design of our Elf. Therefore, we cannot simply improve the Data Dwarf. Instead, we need to combine concepts behind it with new ideas and an optimized main-memory layout in order to significantly speed-up selection predicate evaluation on multiple columns in Elf.

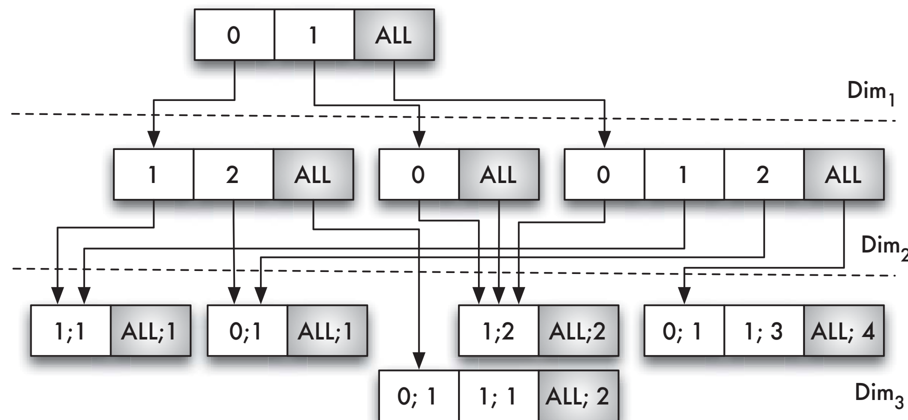

Figure 7.6: The Data Dwarf of the cube from Figure 7.5

Avoiding Prefix and Suffix Redundancies

The Data Dwarf avoids prefix redundancies by having only one entry per value per dimension in each list. For example, the value 1 in dimension Dim_1 is only one entry in the first dimension, although the cube uses two rows to store it. Thus, the graph-based representation of the Data Dwarf is avoiding this redundancy. For avoiding suffix redundancies, the graph property of the Data Dwarf is exploited. Whenever there is a common suffix for different entries of the cube, the pointers of all these entries will just point to the same node in the Data Dwarf.

Known Limitations

The resulting Data Dwarf layout is highly dependent on the order of the dimensions during construction. The best setup for the Data Dwarf is to start with the dimension with the highest cardinality, because it creates the best fanout so that the underlying sub-graphs get smaller. This is also a reasonable criteria for the heuristic of the column ordering in Elf. However, this is just a heuristic, which depends highly on the data distribution.

Another limitation for the Data Dwarf is that for an efficient insert, the data has to be inserted in a sorted order. Only under this circumstance, the construction could be done in one pass over the input data, because it facilitates the decision when to close a node with the "ALL" entry [SDRK02]. Sorting the data is also important for Elf's build algorithm and is the most promising way to construct our Elf.

Further Use Cases

Due to the good properties of the Data Dwarf, there are several more advanced applications where it has been used. Longgang and Feng use the Data Dwarf for storing iceberg cubes, i.e., cubes with a minimal support within each aggregation [LY04]. Sismanis et al. extend the Data Dwarf's capabilities to be applicable to different aggregation steps of the rollup operator [SDKR03]. Additionally, due to its good compression characteristics, Michalarias et al. use the Data Dwarf for mobile OLAP [MOL09].

7.3 SIMD-Accelerated Main-Memory Indexing

An important part of this work is to reach hardware-sensitivity for the proposed approaches. Apart from our work, there are recent advances in the same direction for other (one-dimensional) index structures that employ SIMD to reach hardware-sensitivity. In this section, we review previously proposed one-dimensional index structures Seg-Tree/Trie, FAST, ART, and VAST from which we want to extract important optimizations that could be applied to our Elf. We consider the adaptations made compared to the base index structure, the usage of SIMD, and the performance gain presented by the authors of the selected index structures.

7.3.1 Seg-Tree and Seg-Trie

Zeuch et al. adapted the B^+-Tree by having a k-ary search tree as each inner node, called segment, and perform a k-ary search on each segment. In Figure 2, we show the adaptation of nodes made by Zeuch et al. for Seg-Tree. The k-ary search bases on the binary search but divides the search space into k partitions with k-1 separators. Compared to binary search, the k-ary search reduces the complexity from $O(\log_2 n)$ to $O(\log_k n)$. Considering m as the maximum number of bits to represent a data type and $|SIMD|$ as the size of a SIMD-register, called SIMD bandwidth: $k = \frac{|SIMD|}{m}$ defines the number of partitions for the k-ary search.

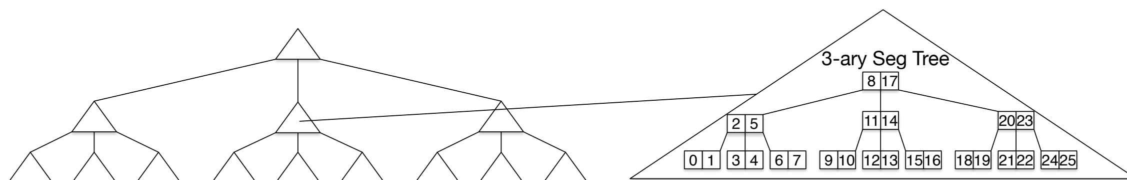

Figure 7.7: Inner node format of Seg-Tree

SIMD Adaptation

As mentioned before, each segment of the Seg-Tree is a k-ary search tree. To perform a k-ary search on a segment, Zeuch et al. liearize the elements of the segment. They show two algorithms for linearization using breadth-first search and depth-first search. Because of the condition $k = \frac{|SIMD|}{m}$, each partition of the k-ary search fits into a SIMD register and is compared to the search key. A perfect k-ary search tree contains $S_{max} = k^h - 1$ keys for an integer $h > 0$. The considered search algorithm only works for sequences with a multiple of $k - 1$ keys. In case of sequences with less than a multiple of $k - 1$ keys, they replenish the sequence with elements having the value $k_{max} + 1$ for the maximal key value k_{max} in the sequence. Consequently, the adapted search algorithm also works for sequences with less than a multiple of $k - 1$ keys.

Performance Improvement

The performance of Seg-Tree depends on k-ary search and horizontal vectorization. The smaller a key the more keys are compared parallel. Due to the relevance of 32 and 64-bit data types in modern systems, the k-ary search performance increases up to a factor of four for 32-bit types and two for 64-bit types.

Zeuch et al. also present the k-ary search on an adapted prefix trie *(trie for short)* called Seg-Trie. A trie is a search tree where nodes store parts of the key, called chunks. For example, a 32 bit key with a chunk size of 8 bit is stored in a trie with 4 levels. The Seg-Trie$_L$ is defined as a balanced trie where each node on each level contains one part of a key with L Bits. The tree has $r = \frac{m}{L}$ levels $(E_0, E_1, .., E_r)$, where m is the number of necessary bits to represent the data type. Similar to the Seg-Tree, each node is again designed as a k-ary search tree. Complete keys are stored in leaf nodes or are build by concatenating partial keys from the root node to a leaf node. This approach benefits from the separation of the keys in different levels of the tree. Consequently, they are smaller and more keys can be compared in parallel.

To perform a tree traversal on the Seg-Trie, the search key is split into r segments, each segment r_i being compared to level E_i. If a matching partial key is found in a node of E_i, the search continues at the referenced node for the partial key. If no match of the partial key is found, the Seg-Trie does not contain the search key and the search is finished. Consequently, the advantage of Seg-Trie against tree structures is the reduced comparison effort for non-existing key segments.

7.3.2 Fast Architecture Sensitive Tree

Kim et al. adapted a binary tree to optimize for architecture features like page size, cache-line size, and SIMD bandwidth called *Fast Architecture Sensitive Tree (FAST)* [KCS+10]. In contrast to Seg-Trees, FAST is also adapted to disk-based database systems. Kim et al. show the performance increase due to decreasing cache misses and better cache-line usage. In order to optimize for architectural features, tree nodes are rearranged by hierarchical blocking. In Figure 3, we show an index tree blocked in the three-level hierarchy introduced by Kim et al. They split the tree into a number of subtrees, each one fitting a page block. These sub-trees can be further

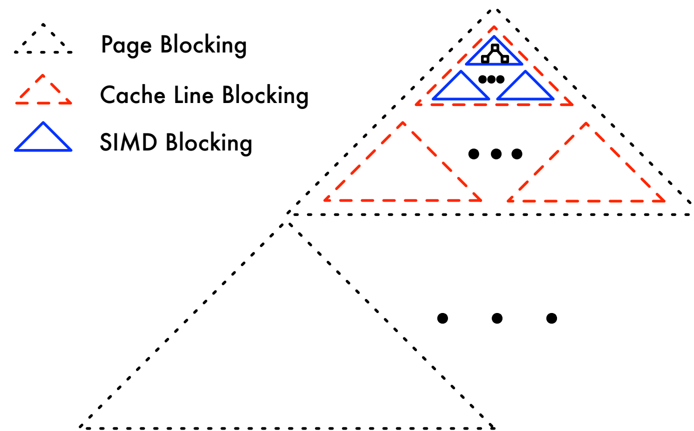

Figure 7.8: Index tree blocked in three-level hierarchy: First-level page blocking, second-level cache-line blocking, third-level SIMD blocking of FAST. Adapted from Kim et al. [KCS+10]

split into cache-line-sized subtrees building the second level in the hierarchy. Each of these subtrees that fit a cache line can be further split into a number of nodes that fit a SIMD register. These nodes are laid out in a breadth-first fashion (the first key as root, the next keys as children on depth 2) representing a SIMD block.

SIMD Adaptation

Kim et al. present implementations for building and traversing the tree adapted for CPU and also for GPU. Building up the tree, SIMD is used to computing the index for each set of keys within the SIMD-level block in parallel, achieving around 2X SIMD scaling as compared to the scalar code. Traversing the tree, they compare one search key to multiple keys of the index structure. To use the complete bandwidth of cache and main memory within the search, blocks are loaded completely into associated memory from large blocks to small blocks. For a page block, at first the page is loaded into main memory. Then cache-line blocks are loaded one after another in the cache and for each cache-line block, the included SIMD blocks are loaded into the SIMD register. All keys of this SIMD block are compared with one SIMD instruction. After examining the bit mask as result fo the comparison, the corresponding next SIMD block is loaded (including the load of the surrounding larger blocks) until the key is found or the last level of the index structure is reached.

Performance Improvement

Kim et al. consider the search performance as queries per second. The CPU search performance on the Core i7 with 64M 32-bit (key, rowID) pairs is 5X faster than the best reported number [SGL09], resulting in a throughput of 50M search queries per second. Considering larger index structures, the GPU performance increase exceeds the CPU performance increase, because TLB and cache misses grow up and the CPU search becomes bandwidth bound.

7.3.3 Vector-Advanced and Compressed Structure Tree

Yamamuro et al. extended FAST by building an index structure called Vector-Advanced and Compressed Structure Tree (VAST) [YOHY12]. They adapt the blocking and aligning structure of FAST and add compression of nodes along with improved SIMD usage.

In order to decrease the size of the index structure to better fit into main memory, they compress inner nodes. Inner nodes above a given threshold are compressed to 16 bit keys using lossy compression, while nodes in deeper levels are compressed to 8 bit with lossy compression. In the leaf nodes, Yamamuro et al. decrease the node size with the lossless compression algorithm *P4Delta* [ZHNB06], which results in a good balance between compression ratio and decompression speed. To compensate errors that occur due to lossy compression, they present an algorithm for error correction. In a nutshell, they use prefix and suffix truncation to compress keys and calculate an offset Δw of the incorrect key to the correct key during tree traversal. If $\Delta w \neq 0$, VAST scans the leaf nodes sequentially until Δw becomes 0.

SIMD Adaptation

Along with the other considered index structures, Yamamuro et al. compare multiple keys to one search key with SIMD in the tree traversal. Due to the key compression of nodes, VAST compares more keys in parallel than FAST. Additionally, they reduce branch misses with an adapted SIMD usage. They use addition and multiplication operations on the results of a SIMD key comparison, instead of conditional branches (if-then paths), to find the next node in tree traversal.

Performance Improvement

Due to lossy and lossless compression of the majority of nodes, Yamamuro et al. reach 95% less space consumption of VAST compared to a binary tree or FAST, respectively. When considering an index with 2^{32} keys, they reach up to 6.0 and 1.24 times performance increase compared to a binary tree and FAST. Although errors occur due to lossy compression, the error correction does not have a major influence on the query execution speed.

7.3.4 Adaptive Radix Tree

Leis et al. adapted a radix tree for efficient indexing in main-memory database systems called *Adaptive Radix Tree (ART)* [LKN13]. Similar to the Seg-Trie, the height of a radix tree depends on the chunk size of the keys stored in each node. ART divides keys into 8-bit chunks. They differentiate between inner nodes and leaf nodes and adapt each of them in a different way.

Instead of using a constant node size for each inner node, they present four types of nodes with different numbers of keys and children. In Figure 5, we show these node types containing keys that are mapped to subtrees. The types of nodes, sorted ascending by their size, are *Node4*, *Node16*, *Node48*, and *Node256*.

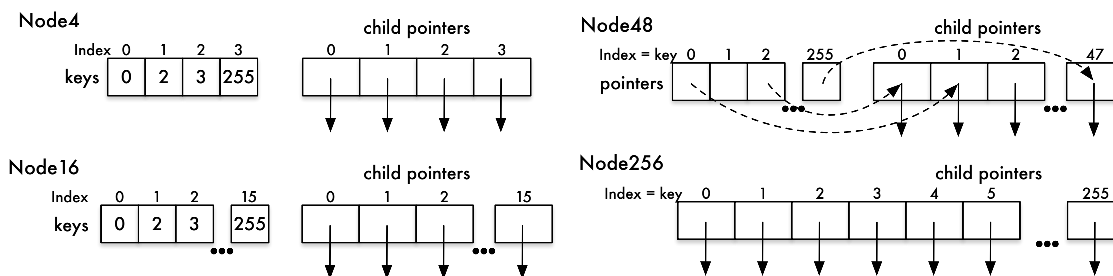

Figure 7.9: Inner nodes of ART. The partial keys 0, 2, 3, and 255 are mapped to pointers of the subtrees. Adapted from Leis et al. [LKN13]

Node4: The smallest node type consists of one array with up to four sorted keys and another array with up to four children. The keys and pointers are stored at corresponding positions.

Node16: This node type consists of arrays of 16 keys and 16 children, storing keys and children analogue to *Node4*.

Node48: To avoid searching keys in many elements, this node type does not store the keys explicitly. Instead, an array with 256 elements is used. This array can be indexed with key bytes directly. It stores indexes into a second array with the size of 48 elements containing the pointers to child nodes.

Node256: The largest node type is simply an array of 256 pointers. Consequently, the next node can be found efficiently using a single lookup of the key byte in that array.

When the capacity of a node is exhausted due to insertion, it is replaced by a larger node type. When a node becomes underfull (e.g., due to key removal), it is replaced by a smaller node type.

For leaf nodes, Leis et al. use a mix of pointer and value slots in an array. If the value fits within the slot, they store it directly in the slot. Otherwise, a pointer to the value is stored. They tag each element with an additional bit indicating if a pointer or a value is stored.

SIMD Adaptation

According to FAST and Seg-Tree, Leis et al. use SIMD within the tree traversal. They use horizontal vectorization, comparing the search key against multiple keys of a node. In contrast to Zeuch et al., using horizontal vectorization for each inner node, Leis et al. only compare the keys of nodes with type *Node16* in parallel. Therefor, they replicate the search key 16 times and compare these against all keys of nodes of type *Node16*.

In contrast to FAST and Seg-Tree, the goal of ART is also to reduce space consumption. Leis et al. use lazy expansion and path compression. The first technique, lazy expansion, is to create inner nodes only if they are required to distinguish at least two leaf nodes. The second technique, path compression, removes all inner nodes that have only one child.

Performance Improvement

Since SIMD is only used in the tree search, we do not consider the performance increases of ART in insert and update operations. Leis et al. show, that looking up random keys using ART is faster than Seg-Tree and FAST, because ART has less cache misses and less CPU cycles per comparison. They consider the performance increases for dense and sparse keys, while ART works better with dense keys. Also they show that a span of 8 results in better performance than a smaller span.

Criterion	Seg-Tree/ Trie	FAST	ART	VAST	Elf
Horizontal vectorization	x	x	x	x	-
Minimized key size	o	-	x	x	-
Specialized node sizes / types	-	-	x	-	-
Decreased branch misses	-	x	-	x	-
Exploit cache lines using blocking and alignment	-	x	-	x	x
Usage of Compression	o	-	x	x	x
Adapt search algorithm for linearized nodes	x	-	-	-	x

Legend: x = implements the issue; o = partially implements the issue;
- = does not implement the issue

Table 7.1: Comparison of the considered index structures based on extracted criteria

7.3.5 Comparison to Elf

In this section, we compare the optimizations made in Seg-Tree, FAST, ART, and VAST to increase performance and show differences to Elf. We summarize the optimizations in Table 7.1 and show which index structure implements which of the criteria. If the index structure does not implement a criterion, we consider if it is possible to implement it.

Horizontal Vectorization: All considered approaches except Elf use *horizontal* vectorization in search operation for a single (broadcasted) search key in the index structure showing significant performance improvements. Hence, an adaptation of Elf in this criteria is promising.

Minimized Key Size: Minimizing the key size as much as possible speeds up the search performance, because more keys can be compared with a single SIMD instruction. Additionally, the smaller the keys are, the more keys fit into a cache line. Zeuch et al. minimize the key size for the Seg-Trie using a small chunk size (please mind the resulting bigger tree size), however, they do not use it in the Seg-Tree. Kim et al. do not minimize the key size for FAST, therefore the performance increase with SIMD also depends on the size of the used data type. Elf currently does not feature a special compression mechanism of node entries. However, since the value range is usually known, compression will definitely pay out in this scenario. Especially in combination with SIMD acceleration, a magnitude of performance improvement is reasonable to expect.

Specialized Node Sizes and Types: An important property of ART is that it offers different node types depending on the cardinality of stored data. However, the other index structures do not adapt to the stored data and, hence, leave this tuning opportunity open.

Decreased Branch Misses: Evaluating comparisons using conditional branches can lead to branch misses, if the CPU prefetches the wrong branch. Consequently, decreasing branch misses improves search performance because less CPU cycles are needed. VAST uses addition and multiplication operations to determine the next node in their evaluation of the comparison result, thus, avoiding branches. The other index structures use conditional branches in their search algorithms and careful considerations are necessary to transform conditional branches into conditional moves or data flow [SBS18].

Exploit Cache Lines Using Blocking and Alignment: Kim et al. segmented FAST into SIMD, cache, and page blocks. They highly optimize FAST for efficient cache-line usage by blocking the index structure into blocks with the size of a cache line, which shows considerable performance benefits. Although several structures allow for blocking due to their linearization (cf. Elf, Seg-Tree, ART), currently they do not block or align their nodes to cache lines or have a special blocking.

Usage of Compression: Key compression leads to better search performance with SIMD, because more keys are compared in parallel. While Seg-Tree and FAST do not use compression, VAST does. Yamamuro et al. use lossy block compression to decrease the key size of blocks. Consequently, they compare more keys with one SIMD instruction, whereas performance decrease of the occurring errors is smaller than the performance increase of the compression. Instead of compressing blocks, Seg-Trie and ART use path compression to decrease the tree height. This is also an idea that Elf exploits.

Adapt Search Algorithm for Linearized Nodes: Searching children in nodes with many keys can be speeded up with adapted search algorithms compared to linear search. Zeuch et al. introduce k-ary search for the Seg-Tree and Seg-Trie, which performs in $O(log_n)$ compared to $O(n)$ of linear search, where k is the SIMD bandwidth. Consequently, less keys are compared in one node, whereas a linearized storage of the keys in the node is required. For FAST and VAST, the k-ary search is not applicable because they are adapted from binary trees and have only one key in each node. Leis et al. use linear search in their adapted nodes of ART to find children. For the node type *Node256* k-ary search can speed up finding the correct child node. Similar to this, the sort order in nodes of Elf is not fixed. Hence, Elf provides the opportunity to use k-ary search.

Summary

Overall, this survey has shown that our index structure Elf already shares some characteristics with highly-tuned main-memory index structures. However, especially the most promising optimizations (horizontal vectorization, minimizing key sizes, and branch misses) are currently not used. Still, we have argued that all these optimizations are possible to be implemented in future work.

7.4 One-Dimensional Main-Memory Indexing

Due to the missing bottleneck of disk access in main-memory database systems, algorithms and data structures now have to be optimized for better cache performance [BMK99]. Hence, several common index structures – usually B-Tree derivatives – were proposed. In this section, we shortly describe the idea and optimizations of these structures.

Cache Sensitivity for B-Trees

Since B-Trees are the standard index structure for many database systems, tuning B-Trees for better cache performance in main-memory database systems is a reasonable step. At first, Rao and Ross adapt the B-Tree structure with a special linearization technique [RR99]. The resulting CSS-Tree has its intermediate nodes stored as a contiguous array in level-order. Since the CSS-Tree is static and needs to be rebuilt on insert, Rao and Ross propose the CSB^+-Tree [RR00]. The CSB^+-Tree stores only all child nodes of the current node in a contiguous array, which allows for cache sensitivity but also for incremental updates. Hence, these optimizations are an interesting future investigation for our Elf allowing for in-place updates.

Cracking

An important approach to create a sorted index structure adaptively during query processing is cracking. The idea is to use the query predicates to partition the data according to the query intervals. For example, a less-than predicate splits the table into two partitions of data – one that contains values that are bigger than or equal to the constant and one that contains values that are smaller than the constant. As a result, the table is step-wise split and is eventually sorted if all possible constants are queried. However, data reorganization creates a considerable overhead for query processing in the first steps and, hence, was avoided for a long time. Cracking was first introduced by Kersten and Manegold [KM05] and integrated into MonetDB by Idreos et al. adding specific operators [IKM07]. Special hardware-sensitive optimizations are done by Pirk et al. [PPI+14], who apply SIMD and predication to speed up the cracking process, and Petraki et al. [PIM15], who leverage the available parallelism of the machine for cracking.

Tree Structures for Non-Volatile Memory

Recent advancements in memory architectures brings the idea of persistency in RAM into practice. With *non-volatile memory (NVM)*, also called *storage class memory (SCM)* or *non-volatile RAM (NVRAM)*, implementations of byte-addressable main memory has been proposed that are based on different underlying techniques. Usually, the benefit of persistence comes with the drawback of asymmetric read-write latencies and memory cells that wear out when being used (similar to flash memory) [APW+08, OKW17, GvRL+18].

An important research topic for NVM is to adapt the B^+-Tree of database systems to be persisted in NVM. Chen et al. adapt a B^+-Tree, called wB^+-Tree, with specific instructions to manage NVM writes safely and also use an additional array that

buffers insertions and deletions to reduce real writes on B^+-Tree nodes in NVM [CJ15]. Yang et al. propose the NV-Tree which separates B^+-Tree nodes into critical nodes (i.e., leaf nodes) stored in NVM and reconstructable data (i.e., internal nodes) stored in DRAM [YWC+15]. Oukid et al. present the FPTree which combines the benefits of the wB^+-Tree and the NV-Tree by adding support for hardware transactional memory [OLN+16]. Arulraj et al. present the BzTree as a latch-free B^+-Tree using a multi-word compare-and-swap operation [ALML18]. All these adaptions show that a careful design for NVM is necessary. When using Elf on NVM, similar adaptions are necessary.

7.5 Multi-Dimensional Main-Memory Indexing

There is a plethora of multi-dimensional index structures already introduced for spatial and multi-media database systems. For a comprehensive overview, we refer to the surveys of Gaede and Günther [GG98] and Böhm et al. [BBK01]. However, most of these multi-dimensional index structures are tuned for window-based range queries or similarity queries. Hence, there is only little work investigating their benefit for multi-column selection predicates.

Partitioning Multi-Dimensional Data With Cracking

Apart from the BB-Tree [SSL18a, SSL19], the field of new recent multi-dimensional main-memory indexing is only sparsely populated. An interesting field is to apply a multi-dimensional cracking to built a multi-dimensional index on the table. Pavlovic et al. use cracking for structuring raw data by using spatial queries [PSHA18] and Holanda et al. propose to build a KD-Tree from multi-column selection predicates queries on the underlying data [HNdAM18]. Both approaches create a partial ordering but do not reach a full partitioning of the space like Elf. Only when enough diverse queries are issued, cracking pays off in the long run.

Adapting State-of-the-Art Multi-Dimensional Index Structures for Modern Hardware

A recent benchmark, called *Genomic Multidimensional Range Query Benchmark (GMRQB)*, that is specifically designed for multi-dimensional range queries in multi-threaded main-memory systems was proposed by Sprenger et al. [SSL18b]. In their evaluation, they test parallelized versions of well-known multi-dimensional structures. These structures are kd-Tree [Ben75], VA-File [WSB98], R^*-Tree [BKSS90] and a full-table scan on a row-store or column-store data layout. In order to reach multi-threaded tree structures, Sprenger et al. horizontally partition the data and create one tree per partition in order to search these trees independently in parallel [SSL18b]. Furthermore, nodes are kept fully in main-memory and nodes sizes were adapted to expand beyond page limits of disks.

A kd-Tree is a universal index structure that is used in many image-processing workloads. Hence, kd-Trees have been ported to the architecture of GPUs, especially focussing on parallel insertions [ZHWG08, SSK07]. Garcia et al. optimize similarity queries on kd-Trees [GDB08]. However, Sprenger et al. are the first to investigate parallel multi-dimensional partial-match queries on CPUs [SSL18b].

Since R-Trees are based on B-Trees, Kim et al. apply optimizations of CSB-Trees to R-Trees [KCK01]. Their CR-Tree encodes the coordinates of a minimum bounding rectangle (MBR) in relation to its parent. Thus, coordinates usually contain many trailing 0s, which can be compressed. Hence, they apply compression which also optimizes cache utilization.

8. Conclusion

In this chapter, we conclude our thesis. To this end, we summarize the contributions and results of each chapter in detail by also referring to the goals presented in the introduction. Since the goal of Level 2 has a wide range and several important aspects, we split the contributions to this goal into four sub-levels Level 2.1-2.4.

Level 1: Hardware-Sensitive Scans

The contribution of the first level is found in our optimizations for hardware sensitive scans. It spreads across three main contributions, which are the definition of code optimizations, an assessment of their impact for mono and multi-column full-table scans, and the design of a framework for variant tuning using code optimizations. These contributions and results are explained in the following in detail.

Code Optimizations: As first contribution, we introduce the concept of code optimizations as a powerful abstraction level to reach hardware-sensitivity for arbitrary operators. We also review for each code optimization its application field in other operators. As a result, we give a comprehensive overview for the state of the art in code optimizations.

Code Optimized Full-Table Scans: A second contribution is to test the applicability of code optimizations for mono and multi-column full-table scans. Our results show that there is not one best optimized scan, but depending on the workload, some scan variants perform better than others. Consequently, a DBMS has to be equipped with a set of code optimized scans to reach peak performance.

Adaptive Reprogramming: Due to the fluctuation of use cases for code optimizations depending on the workload, there is a need to automate the generation of hardware-sensitive code for database operators. As a consequence, we introduced our concept of the adaptive reprogramming framework that is able

to tune database operators during runtime. Furthermore, we present its application and similarity to already existing execution engines of main-memory database systems.

As a result, we achieve the first goal of our thesis – an important extension of hardware sensitive scans to reach peak performance. These scans serve as a powerful baseline to evaluate subsequent contributions. In summary, our contributions answer the first two research questions RQ 1 and RQ 2 concerning the question how to reach hardware-sensitivity for full-table scans and how to automate their tuning and selection.

Level 2.1: Accelerating Multi-Column Selection Predicates

Despite the comprehensive optimization of full-table scans in the first level for a variety of use cases, which leads to a powerful baseline, there are still open challenging use cases. Especially, the pain point of full-table scans is still that the number of memory accesses increases linearly by the number of tuples and also increasing the number of evaluated columns adds a considerable amount of overhead. Hence, a multi-dimensional index structure is a reasonable addition to hardware-sensitive scans to investigate.

Our investigation shows that exploiting prefix-redundancies in a sort-based multi-dimensional tree structure, called Elf, is a well performing approach to exploit the relation between multiple columns of a selection predicate. In order to reach peak performance, we optimize this structure for different data distributions with the introduction of `MonoLists`, a hash map for the first index level and a clever storage design for cache-consciousness. Our evaluation shows that Elf can outperform the full-table scans of adaptive reprogramming and of MonetDB as well as other multi-dimensional index structures by several magnitudes for multi-column selection predicates of the TPC-H benchmark with considerably high selectivities.

The result of this level shows that index structures still have important use cases in main-memory database systems – of course in combination with optimized full-table scans. This answers RQ 3 concerning the design of an alternative structure for accelerating multi-column selection predicates.

Level 2.2: Accelerating Complex Selection Predicates

Due to the flexibility of full-table scans for arbitrary selection predicates, our introduced multi-dimensional index structure lacks efficient support for complex selection predicates (i.e., IN-predicates and column-column comparisons). Hence, we investigate for Level 2.2 how to efficiently support complex predicates in Elf. This results in the introduction of two algorithms – one for evaluating IN-predicates and one for evaluating column-column predicates – which are adapted to all levels of the Elf, i.e., its hash map, its `DimensionLists`, and its `MonoLists`.

Our subsequent evaluation shows the usability for Elf for different complex predicate queries, which shows a better scaling behavior than a full-table scan when increasing the number of IN-values and number of column-column comparisons. Hence, our Elf is capable to support arbitrary selections that can be expressed with SQL, which also answers research question RQ 4.

Level 2.3: Query Acceleration in Main-Memory Database Systems with a Multi-Dimensional Sort-Based Index Structure

With the introduction of Elf, we reached a milestone in this thesis for accelerating complex multi-column selection predicates due to its enormous performance benefits. However, still the question arises whether it is possible to equip a query engine of a main-memory database system with Elf and afterwards reach the same performance speedups (cf. RQ 5). To this end, we integrated Elf into MonetDB and test its query performance against MonetDB's full-table scans for a variety of TPC-H queries.

The results of the integration tests show a positive outcome. Overall, Elf shows remarkable benefits for queries with reasonably high selectivity and its integration shows only minimal overhead. However, for some TPC-H queries, the interoperability of Elf's selection result and the access pattern of subsequent operators is still a major bottleneck. Due to the unsorted output of TIDs of Elf, subsequent joins and projections add a considerable performance overhead due to random memory accesses. Hence, it is open for future work to solve or minimize this overhead. Still, we deem the integration as successful (and thus the answer to RQ 5) since the query times of the integrated Elf are often better than those with full-table scans and for some queries, Elf delivers a performance in at least the same range as queries backed by full-table scans. Hence, we argue that Elf is a reasonable structure to be used in arbitrary main-memory database systems in order to reach peak performance.

Level 2.4: Maintenance of a Multi-Dimensional Sort-Based Index Structure

The last contribution to Level 2 is an efficient maintenance of our proposed Elf. Since maintenance jobs (i.e., building, inserting values) is an important and easy operation for column stores (and by concept also for full-table scans), it is also important to support maintenance jobs within Elf. To this end, we present an efficient build algorithm as well as a design for efficient inserts, updates, and deletes. The insertion algorithm for data-warehouse-like scenarios allows to frequently append data efficiently in Elf with only minimal overhead. The idea is based on delta stores allowing to stage data and after a specific threshold of insertions to merge both structures.

Our results for building and maintaining Elf shows that it cannot always outperform simpler structures, but its performance is in the same order of magnitude. Due to the proven performance benefit of Elf by several magnitudes, we argue that such maintenance performance is acceptable, which concludes the outcome of research question RQ 6.

Thesis Conclusion

As an overall conclusion, the contribution of all chapters leverages the performance of selections to another level. By automatically generating hardware-sensitive database operators using code optimizations, the whole system is capable to optimize for arbitrary underlying (future) hardware. Furthermore, by exploiting the underlying data distributions with code optimizations as well as with our multi-dimensional sort-based index structure, we are able to push database performance beyond tuning it only for the hardware, but also for a arbitrary use cases. Hence, the contribution of the thesis is a Swiss-army knife for a query engine to accelerate arbitrary selection predicates.

9. Future Work

In this chapter, we give an overview of open or newly arisen challenges and goals that can be inferred from the results of the thesis. Of course, the future work for this thesis spans across all levels that we worked on in the thesis. Hence, we first review future work for hardware-sensitive full-table scans and afterwards new ideas on extending our Elf approach. Although the thesis may give the impression that there are only small optimizations left for improving the applicability of Elf, however, the clever design became a door-opener for another level that goes beyond the optimization of selection predicates.

Extending Level 1: Hardware-Sensitive Database Operators

In this thesis, we characterized code optimizations that are well suited for selection predicates and proposed a methodology to exploit code optimizations for arbitrary operators. Due to the focus of this thesis, a comprehensive extension of both contributions is needed to reach hardware-sensitivity without borders. This goal calls for an extended set of code optimizations and a more powerful automation of variant generation.

Extending Code Optimizations

The result of this thesis is a set of code optimizations that are applicable for full-table scans. We also included further optimizations that were proposed in related work. However, to optimize database operations for different use cases, it is necessary to create a comprehensive set of possible code optimizations. This includes to review and also extend code optimizations for different database operators, workloads, as well as for different devices, while being coupled with performance evaluations to assess their usefulness. Especially code optimizations for highly parallel architectures, such as GPUs or Xeon Phis, is an important task. Furthermore, advances in data-parallel processing forces the support for the ever-growing set of SIMD instructions. As a result, we are able to exploit the given hardware and workload characteristics at its maximum reaching *bare-metal speed*.

Automating Code Optimization and Generation

Since several frameworks already implement our idea of adaptive reprogramming, there are only small changes to be done to fully exploit the underlying hardware and workload for arbitrary database workloads. However, the future work in the direction of code optimization rather lies in the direction of software engineering. So far, there is no method that enables an easy and reliable way to apply arbitrary code optimizations on our database operators. Probably, newest advances in lightweight-modular staging (i.e., abstraction without regret [SKK18]) or software product lines [BDKM14, MTS+17, KPK+18] may come to the rescue in this regard. However, a comprehensive test for their applicability is still open.

Extending Level 2: Multi-Column Selection Predicates in Elf

Our multi-dimensional sort-based index structure Elf is already a powerful approach to accelerate queries with low-selectivity on multiple columns. However, there are still challenges that limit the applicability of Elf, which we intend to solve in the future. The following optimizations range from conceptual optimizations on the Elf design to use-case-specific optimizations.

Extending Elf's Design

The conceptual design of Elf dictates that we have to travers down to the leaf levels of the Elf to reach TIDs. However, an identified problem from our benchmarks in Section 4.5.2 is that especially mono-column selection predicates suffer high performance penalties. To optimize even these workloads, we can use the property of Elf that it creates a total ordering of the data space and, hence, also on the TIDs. Thus, a reasonable optimization would be to include additional pointers that point from an upper level in the Elf to the matching partition of the TIDs. These pointers (i.e., *Shortcuts*) can then be used in case the multi-column selection predicate ends at a specific level in order to shortcut to the matching TIDs. Nevertheless, this optimization poses further challenges, because not all levels of the Elf need to be equipped with additional pointers (by incurring additional storage space consumption) in order to minimize the overhead for other queries as well as the storage consumption of the query. In addition to that, different node designs (e.g., storing nodes as a k-ary search tree [SGL09]) and linearization strategies are a reasonable extension, as we already identified in Section 7.3.5. Especially for domains with huge data cardinalities (e.g., protein databases [HSZ+17, ZSJ+18]), such optimizations are essential.

Widetable Approach for Elf

In our evaluation of Elf's impact on the whole query runtime (cf. Section 5.3.2), we have identified that the unordered TID list diminishes performance benefits of the selections due to an overhead in subsequent joins and projections. A reasonable approach to avoid join processing is to use a denormalized schema, a so-called *wide table* [BYT+17, LP14]. In a wide table, all joins are already executed leading to a table with all columns and a lot of data redundancy. Since Elf is able to compress the redundancy of column values, it is a valuable goal to use Elf to accelerate a wide table approach. Furthermore, by including shortcuts, the additional columns will also only minimally impact the performance behavior of Elf. However, how well Elf can accelerate a wide table approach is open for an extensive evaluation.

Solving the Column Ordering Problem

Currently, we use a simple heuristic to order the columns for our executed queries, which is based on the frequency of usage of the columns and cardinality of the values in the column. So far, this heuristic has proven well and has been approved by the cost model of Jonas Schneider [Sch15]. However, there are still several extensions necessary to guarantee a comprehensive modeling for Elf's performance. At first, the cost model needs to be extended for complex predicates in order to get a comprehensive cost model for arbitrary selection predicates. Furthermore, the possibility of using Elf as an index structure leaves the opportunity open to generate several query-sensitive Elfs. With this in mind, a cost model could also generate several partially-overlapping Elfs that balance storage consumption and query performance for a set of queries.

Introducing Level 3: Elf as a Full-Fledged Accelerator for Hybrid Relational and Similarity Queries

The last important part of future work covers the capabilities of Elf beyond accelerating only selection predicates. For instance, there is already a plethora of related work (cf. Chapter 7) of multi-dimensional index structures to accelerate similarity queries (e.g., k-nearest-neighbor (kNN) queries). By combining selections and for instance similarity queries, we are able to accelerate whole queries without result materialization leading to an enormous benefit compared to usual query engines suffering from intermediate result materialization. Hence, we present our ideas for Elf's applicability on further relational and similarity query operators in the following.

Extensions for Relational Operators

Interestingly, the main properties of Elf (i.e., sorted node entries and values of each level correspond to a single column) allow for an efficient support for sort, group, and join operators. While sorting and grouping drills down to a merge of pre-sorted data (i.e., `DimensionLists`), the join operator needs more effort. For instance, joins of two Elf's on a join column could be done with a grouping phase on the first (bigger) Elf and one selection per group on the second Elf. Another possible optimization is to use co-clustering similar to BDCC [BBS16]. In summary, the goal of this extension is to architect clever approaches to efficiently combine all relational operators into one traversal of the Elf.

Extensions for Similarity Operators

A last extension of the Elf is to support similarity operators like similarity-based selections or queries. This extension in combination with other operators allows to do similarity-based analyses on a specific subset of the data (i.e., a selected subpartition of the data). A valuable property of Elf is that it puts a total ordering of the space, which means that all points with the same prefix have the same distance to the query point. As a result, Elf's partitioning can reduce distance computations of a set of points to a single distance computation to their common prefix. This extension, however, does not only need an efficient operator implementation, but also an adaption of the cost model.

Bibliography

[ABH+13] Daniel J. Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. Foundations and Trends in Databases, 5(3): 197-280, 2013.

[ABP+17] Iya Arefyeva, David Broneske, Marcus Pinnecke, Mudit Bhatnagar, and Gunter Saake. Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems. In *Proceedings of the GI-Workshop Grundlagen von Datenbanken (GvDB)*, CEUR Workshop Proceedings, pages 24–29. CEUR-WS, 2017.

[ACP+18] Iya Arefyeva, Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. Low-Latency Transaction Execution on Graphics Processors: Dream or Reality? In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 16–21, 2018.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 266–277, 1999.

[AIA14] Ioannis Alagiannis, Stratos Idreos, and Anastassia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1103–1114. ACM, 2014.

[AKM+16] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastassia Ailamaki, and Mark Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[AKPA17] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastassia Ailamaki. The Case For Heterogeneous HTAP. In *Proceedings*

of the International Conference on Innovative Data Systems Research (CIDR), 2017.

[ALML18] Joy Arulraj, Justin Levandoski, Umar F. Minhas, and Per-Ake Larson. BzTree: A High-performance Latch-free Range Index for Non-volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 466–475. IEEE, 2007.

[Ant97] Gennady Antoshenkov. Dictionary-Based Order-Preserving String Compression. *The VLDB Journal*, 6(1):26–39, 1997.

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 583–598. ACM, 2016.

[APW+08] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 57–70, 2008.

[BATÖ13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward Hardware-Sensitive Database Operations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 229–234, 2014.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[BBS14] David Broneske, Sebastian Breß, and Gunter Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, Lecture Notes in Computer Science (LNCS), pages 97–111. Springer, 2014.

[BBS16] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. Bitwise Dimensional Co-Clustering for Analytical Workloads. *The VLDB Journal*, 25(3):291–316, 2016.

[BDKM14] David Broneske, Sebastian Dorok, Veit Köppen, and Andreas Meister. Software Design Approaches for Mastering Variability in Database Systems. In *Proceedings of the GI-Workshop Grundlagen von Datenbanken (GvDB)*, volume 1313 of *CEUR Workshop Proceedings*, pages 47–52. CEUR-WS, 2014.

[Ben75] Jon Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Bet18] Florian Bethe. Elf Meets MonetDB: Integrating a Multi-Column Structure Into a Column Store. Master's thesis, University of Magdeburg, 2018.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[BKF+18] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal*, 27(6):797—822, 2018.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 28–39, 1996.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 322–331. ACM, 1990.

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017.

[BKSS18] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *Transactions on Knowledge and Data Engineering (TKDE)*, 2018. Accepted in April 2018.

[BLC+16] Debabrota Basu, Qian Lin, Weidong Chen, Hoang T. Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. Regularized Cost-Model Oblivious Database Tuning With Reinforcement Learning. *Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)*, 28:96–132, 2016.

[Blo59] Erich Bloch. The Engineering Design of the Stretch Computer. In *Proceedings of the International Workshop on Managing Requirements Knowledge (MARK)*, pages 48–59, 1959.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding With Allowable Errors. *Communications of the ACM*, 13:422–426, 1970.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, 9(3):231–246, 1999.

[BMS17] David Broneske, Andreas Meister, and Gunter Saake. Hardware-Sensitive Scan Operator Variants for Compiled Selection Pipelines. In *Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 403–412, 2017.

[Bre13] Sebastian Breß. Why it is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMSs. *The VLDB PhD Workshop*, 6(12):1398–1403, 2013.

[Bre14] Sebastian Breß. The Design and Implementation of CoGaDB: A Column-Oriented GPU-Accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[Bro15] David Broneske. Adaptive Reprogramming for Databases on Heterogeneous Processors. In *SIGMOD/PODS Ph.D. Symposium*, pages 51–55. ACM, 2015.

[BS17a] David Broneske and Gunter Saake. Exploiting Capabilities of Modern Processors in Data Intensive Applications. *it - Information Technology*, 59(3):133–140, 2017.

[BS17b] David Broneske and Martin Schäler. Single Instruction Multiple Data – Not Everything is a Nail for this Hammer. In *Proceedings of the International Workshop on Failed Aspirations in Database Systems (FADS)*, 2017.

[BTAÖ13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[BYT+17] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. Wide Table Layout Optimization Based on Column Ordering and Duplication. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 299–314. ACM, 2017.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the International*

Conference on Innovative Data Systems Research (CIDR), pages 225–237, 2005.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CPP+18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya Sekeran, Fabian Rodriguez, and Laxmi Balami. GridFormation: Towards Self-Driven Online Data Partitioning using Reinforcement Learning. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, volume 1, pages 1–7, 2018.

[DBS08] Jens Dittrich, Lukas Blunschi, and Marcos Salles. Dwarfs in the Rearview Mirror: How Big Are They Really? *Proceedings of the VLDB Endowment*, 1(2):1586–1597, 2008.

[DIR07] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases (FTDB)*, 1(1):1–140, 2007.

[DKF+18] Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, and Hasso Plattner. Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans. In *Proceedings of the International Workshop on Big Data Management on Emerging Hardware (HardBD)*, pages 102–109, 2018.

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the International Conference on Supercomputing (SC)*, pages 1–12, 2008.

[DYZ+15] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. 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.

[EN15] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 7 edition, 2015.

[FBBO99] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GBD+18] Bala Gurumurthy, David Broneske, Tobias Drewes, Thilo Pionteck, and Gunter Saake. Cooking DBMS Operations Using Granular Primitives - An Overview on a Primitive-Based RDBMS Query Evaluation. *Datenbank-Spektrum*, 18(3):183–193, 2018.

[GBP+18] Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand, and Gunter Saake. SIMD Vectorized Hashing for Grouped Aggregation. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*, Lecture Notes in Computer Science (LNCS), pages 113–126. Springer, 2018.

[GDB08] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k Nearest Neighbor Search Using GPU. In *Proceedings of the International Workshop on Computer Vision and Pattern Recognition (EMMCVPR)*, pages 1–6. IEEE, 2008.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GvRL+18] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. Data Management on Non-Volatile Memory: A Perspective. *Datenbank-Spektrum*, 18(3):171–182, 2018.

[HKHL15] Carl-Philip Hänsch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Plan Operator Specialization using Reflective Compiler Techniques. In *Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 363–382, 2015.

[HLH13] Jiong He, Mian Lu, and Bingsheng He. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, 2013.

[HLY+09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Co-Processing on Graphics Processors. In *ACM Transactions on Database Systems (TODS)*, volume 34, pages 1–39. ACM, 2009.

[HNdAM18] Pedro Holanda, Matheus Nerone, Eduardo Cunha de Almeida, and Stefan Manegold. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper). In *Proceedings of the International Conference on Data Science, Technology and Applications (DATA)*, pages 393–399, 2018.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4. edition, 2007.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5. edition, 2011.

[HSP+13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[HSZ+17] Robert Heyer, Kay Schallert, Roman Zoun, Beatrice Becher, Gunter Saake, and Dirk Benndorf. Challenges and Perspectives of Metaproteomic Data Analysis. *Journal of Biotechnology*, (261):24–36, 2017.

[HYF+08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Joins on Graphics Processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 511–524. ACM, 2008.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007.

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 297–308. ACM, 2009.

[Inm05] William H. Inmon. *Building the Data Warehouse*. Wiley, 4. edition, 2005.

[Int16] Intel 64 and IA-32 Architectures Optimization Reference Manual, 2016.

[ISO99] *ANSI/ISO/IEC International Standard (S) Database Language SQL – Part 2: Foundation (SQL/Foundation), ISO/IEC 9075-2:1999 (E)*, 1999.

[JLC+15] Saurabh Jha, Mian Lu, Xuntao Cheng, Bingsheng He, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015.

[JLR+94] Hosagrahar Visvesvaraya Jagadish, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudershan. Dali: A High Performance Main Memory Storage Manager. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 48–59, 1994.

[KBSS15] Veit Köppen, David Broneske, Gunter Saake, and Martin Schäler. Elf: A Main-Memory Structure for Efficient Multi-Dimensional Range and Partial Match Queries. Technical Report 002-2015, Otto-von-Guericke-University Magdeburg, 2015.

[KCK01] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 139–150. ACM, 2001.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 339–350. ACM, 2010.

[KKRC14] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building Efficient Query Engines in a High-level Language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.

[KLMV12] Tim Kaldewey, Guy M. Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 55–62. ACM, 2012.

[KM05] Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 213–224, 2005.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 195–206. IEEE, 2011.

[KPK+18] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience*, 48(3):402–427, 2018.

[KSB19] Veit Köppen, Martin Schäler, and David Broneske. *Emerging Perspectives in Big Data Warehousing*, chapter Index Structures for Data Warehousing & Big Data Analytics. IGI Global, 2019.

[KSC+09] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[KSS14] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien*. mitp, 2. edition, 2014.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism With Multimedia Instruction Sets. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 35(5):145–156, 2000.

[Lam75] Leslie Lamport. Multiple Byte Processing With Full-Word Instructions. *Communications of the ACM*, 18(8):471–475, 1975.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[LP13] Yinan Li and Jignesh M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 289–300. ACM, 2013.

[LP14] Yinan Li and Jignesh M. Patel. WideTable: An Accelerator for Analytical Data Processing. *Proceedings of the VLDB Endowment*, 7(10):907–918, 2014.

[Lüb17] Andreas Lübcke. *Automated Query Interface for Hybrid Relational Architectures*. PhD thesis, University of Magdeburg, 2017.

[LY04] Xiang Longgang and Feng Yucai. Fast Computation of Iceberg Dwarf. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 203–212. IEEE, 2004.

[MBNK04] Stefan Manegold, Peter A. Boncz, Niels Nes, and Martin L. Kersten. Cache-Conscious Radix-Decluster Projections. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 684–695, 2004.

[MOL09] Ilias Michalarias, Arkadiy Omelchenko, and Hans-Joachim Lenz. FCLOS: A Client-Server Architecture for Mobile OLAP. *Data & Knowledge Engineering*, 68(2):192–220, 2009.

[MSL+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1123–1136. ACM, 2015.

[MTA11] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting Networks on FPGAs. *The VLDB Journal*, 21(1):1–23, 2011.

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability With FeatureIDE*. Springer, 1. edition, 2017.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[OKW17] Ismail Oukid, Robert Kettler, and Thomas Willhalm. Storage Class Memory and Databases: Opportunities and Challenges. *it - Information Technology*, 59(3):109, 2017.

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 371–386. ACM, 2016.

[OOC09] Pat O'Neil, Betty O'Neil, and Xuedong Chen. Star Schema Benchmark - Revision 3, 2009.

[PBDS17] Marcus Pinnecke, David Broneske, Gabriel Campero Durand, and Gunter Saake. Are Databases Fit for Hybrid Workloads on GPUs?

A Storage Engine's Perspective. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1599–1606, 2017.

[PFRE14] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. In *Gartner*, 2014.

[PIM15] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic Indexing in Main-Memory Column-Stores. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1153–1166. ACM, 2015.

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–2. ACM, 2009.

[PMZM16] Holger Pirk, Oscar Moll, Matei Zaharia, and Samuel Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *Proceedings of the VLDB Endowment*, 9(14):1707–1718, 2016.

[PPI+14] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Database Cracking: Fancy Scan, Not Poor Man's Sort! In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:8. ACM, 2014.

[PR13] Orestis Polychroniou and Kenneth A. Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 6:1–6:6. ACM, 2013.

[PR14] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom Filters for Advanced SIMD Processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 6:1–6:6. ACM, 2014.

[PR15] Orestis Polychroniou and Kenneth A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 9:1–9:6. ACM, 2015.

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1493–1508. ACM, 2015.

[PSHA18] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastassia Ailamaki. QUASII: Query-Aware Spatial Incremental Index. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 325–336, 2018.

[RAD15] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proceedings of the VLDB Endowment*, 9(3):96–107, 2015.

[RBB+18] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. OLTPshare: The Case for Sharing in OLTP Workloads. *Proceedings of the VLDB Endowment*, 11(12):1769–1780, 2018.

[RBZ13] Bogdan Răducanu, Peter A. Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1231–1242. ACM, 2013.

[RHVM15] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The Operator Variant Selection Problem on Heterogeneous Hardware. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2015.

[Rom12] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL Lausanne, 2012.

[Ros02] Kenneth A. Ross. Conjunctive Selection Conditions in Main Memory. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 109–120. ACM, 2002.

[Ros04] Kenneth A. Ross. Selection Conditions in Main-Memory. *ACM Transactions on Database Systems (TODS)*, 29:132–161, 2004.

[RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 78–89, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD Record*, 29(2):475–486, 2000.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-Oriented DBMS. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[SBS18] Lars-Christian Schulz, David Broneske, and Gunter Saake. An Eight-Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors. *Proceedings of the VLDB Endowment*, 11(11):1550–1562, 2018.

[Sch15] Jonas Schneider. Analytic Performance Model of a Main-Memory Index Structure. Bachelor thesis, Karlsruhe Institute of Technology, 2015.

[SDKR03] Yannis Sismanis, Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Hierarchical Dwarfs for the Rollup Cube. In *Proceedings of the International Workshop on Data Warehousing and OLAP (DOLAP)*, pages 17–24. ACM, 2003.

[SDRK02] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the PetaCube. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 464–475. ACM, 2002.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary Search on Modern Processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 52–60. ACM, 2009.

[SGS+13] Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. *Proceedings of the VLDB Endowment*, 6(14):1654–1665, 2013.

[SK13] Lefteris Sidirourgos and Martin L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 893–904. ACM, 2013.

[SKK18] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. Building Efficient Query Engines in a High-Level Language. *Transactions on Database Systems (TODS)*, 43(1):4:1–4:45, 2018.

[SKP+16] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1907–1922. ACM, 2016.

[SM17] Lefteris Sidirourgos and Hannes Mühleisen. Scaling Column Imprints Using Advanced Vectorization. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:8. ACM, 2017.

[SR13] Evangelia Sitaridi and Kenneth A. Ross. Optimizing Select Conditions on GPUs. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:8. ACM, 2013.

[SSH11] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken – Implementierungstechniken.* mitp, 3 edition, 2011.

[SSH18] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken. Konzepte und Sprachen.* mitp, 6 edition, 2018.

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly Parallel Fast KD-Tree Construction for Interactive Ray Tracing of Dynamic Scenes. In *Computer Graphics Forum*, volume 26, pages 395–404. Wiley Online Library, 2007.

[SSL18a] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. BB-Tree: A Practical and Efficient Main-Memory Index Structure for Multidimensional Workloads. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2018.

[SSL18b] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. Multidimensional Range Queries on Modern Hardware. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 4:1–4:12. ACM, 2018.

[SSL19] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. BB-Tree: A Practical and Efficient Main-Memory Index Structure for Multidimensional Workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019.

[Tra14] Transaction Processing Performance Council. TPC BENCHMARK H (Decision Support). Technical Report 2.17.1, 2014.

[Tra15] Transaction Processing Performance Council. TPC BENCHMARK DS. Technical Report 2.1.0, 2015.

[WBP+09] Thomas Willhalm, Yazan Boshmaf, Hasso Plattner, Nicolae Popovici, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[WEBS18] Marten Wallewein-Eising, David Broneske, and Gunter Saake. SIMD Acceleration for Main-Memory Index Structures – A Survey. In *Proceedings of the International Conference Beyond Databases, Architectures and Structures (BDAS)*, pages 105–119. Springer, 2018.

[WOMF13] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing Database Column Scans With Complex Predicates. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2013.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *Proceedings of the International Conferenece on Very Large Data Bases (VLDB)*, 98:194–205, 1998.

[YOHY12] Takeshi Yamamuro, Makoto Onizuka, Toshio Hitaka, and Masashi Yamamuro. VAST-Tree: A Vector-Advanced and Compressed Structure for Massive Data Tree Traversal. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 396–407. ACM, 2012.

[YWC+15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the USENIX*

Conference on File and Storage Technologies (FAST), pages 167–181. USENIX Association, 2015.

[ZAP+16] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases With Hybrid Indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1567–1581. ACM, 2016.

[ZF15] Steffen Zeuch and Johann-Christoph Freytag. Selection on Modern CPUs. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, Lecture Notes in Computer Science (LNCS), pages 5:1–5:8. Springer, 2015.

[ZHB06] Marcin Zukowski, Sándor Héman, and Peter A. Boncz. Architecture-Conscious Hashing. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 6:1–6:6. ACM, 2006.

[ZHF14] Steffen Zeuch, Frank Huber, and Johann-christoph Freytag. Adapting Tree Structures for Processing With SIMD Instructions. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 97–108. OpenProceedings.org, 2014.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 59. IEEE, 2006.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-Time KD-Tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126:1–126:11, 2008.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 145–156. ACM, 2002.

[ZSJ+18] Roman Zoun, Kay Schallert, Atin Janki, Rohith Ravindran, Gabriel Campero Durand, Wolfram Fenske, David Broneske, Robert Heyer, Dirk Benndorf, and Gunter Saake. Streaming FDR Calculation for Protein Identication. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*, Lecture Notes in Computer Science (LNCS), pages 80–87. Springer, 2018.

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 14. Juni 2019

David Broneske