Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Datenparallele Selektionen auf der multidimensionalen Indexstruktur Elf

Autor:

Kai Wolf

06.12.2019

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake

Dr.-Ing. David Broneske

Institut für Technische und Betriebliche Informationssysteme
Otto-von-Guericke-Universität Magdeburg

Zweitgutachter:

Dr.-Ing. Martin Schäler

Lehrstuhl für Systeme der Informationsverwaltung
Karlsruher Institut für Technologie

Wolf, Kai:
Datenparallele Selektionen auf der multidimensionalen Indexstruktur Elf
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, Dezember 2019.

Inhaltsangabe

Die Steigerung der Performanz eines Softwaresystems ist ein beständig aktuelles Bestreben – Datenbankmanagementsysteme bilden dabei keine Ausnahme. Auf abstrakter Ebene kann „Performanz“ im Kontext einer Datenbank als die Interaktionsgeschwindigkeit des Systems mit Anwendungen und Anwendern interpretiert werden. Zur Erhöhung dieser gibt es innerhalb eines Datenbankmanagementsystems verschiedene Stellschrauben. Jahrzehntelang war der Flaschenhals, der sich aus der sogenannten Zugriffslücke zwischen persistentem Platten- und flüchtigem Arbeitsspeicher ergibt, ein zentraler Untersuchungsgegenstand von Datenbankoptimierungen. Mit dem Aufkommen von Hauptspeicherdatenbanken wurde dieser gesamte Optimierungskomplex de facto irrelevant.

Im Gegenzug ergeben sich neue Herausforderungen wie der Zugriff auf die sich im Hauptspeicher befindlichen Daten durch die CPU und dem damit einhergehenden Thema des bewussten Umgangs mit den verschiedenen CPU-Caches. Letztere sind im Vergleich zum RAM von geringeren Zugriffszeiten geprägt. Die Kapazität von CPU-Caches befindet sich jedoch nicht im Giga- bis Terabytebereich wie es beim Hauptspeicher der Fall ist, sondern lediglich im oberen Kilo- bis unteren Megabytebereich. Unter anderem deswegen profitieren Datenbankoperatoren davon, Zwischenergebnisse bei der Bearbeitung von Anfragen möglichst klein zu halten. Gerade durch die im Zuge von Big-Data- und OLAP-Anwendungen immer größer werdenden Datenmengen kommt dem Selektionsoperator, der unter anderem eine derartige Verkleinerung zum Ziel hat, eine besondere Bedeutung zu.

In OLAP-Szenarien nimmt darüber hinaus die Multidimensionalität der Daten eine tragende Rolle ein. Zur effizienten Bearbeitung eben solcher Selektionen wurde von Broneske et al. die multidimensionale Indexstruktur Elf vorgestellt [BKSS17]. Zentraler Untersuchungsschwerpunkt und Beitrag dieser Arbeit soll eine hardwarebasierte Optimierung des Elf sein. Konkret werden mit SIMD (kurz für *__S__ingle __I__nstruction, __M__ultiple __D__ata*) die fortgeschrittenen Möglichkeiten moderner CPUs zur intraprozessualparallelen, vektorbasierten Durchführung der Anfragealgorithmen des Elf genutzt. Es werden im Verlaufe der Arbeit die dazu notwendigen datenstrukturellen und algorithmischen Anpassungen erläutert, bevor die Performanz der neuen Elf-Variante, genannt `Elf_SIMD`, gegenüber der Ursprungsimplementierung untersucht wird. Hierbei wird sich zeigen, dass ein Speedup von bis zu **1,65** mit `Elf_SIMD` bei der Bearbeitung von TPC-H-Benchmarkanfragen erreicht werden kann.

Danksagung

Ich bedanke mich bei allen Personen, die mich beim Schreiben dieser Arbeit unterstützt haben. Zunächst gilt mein Dank Prof. Dr. Gunter Saake, dessen Arbeitsgruppe für Datenbanken und Software Engineering mich das gesamte Studium begleitet und mir nun das Schreiben dieser Masterarbeit ermöglicht hat.
Ebenso möchte ich Dr. Martin Schäler an dieser Stelle meinen Dank aussprechen. Dafür, dass er sich bereiterklärt, diese Arbeit als Zweitkorrektor zu begutachten und mir bereits während des Schreibens konstruktive Verbesserungsvorschläge nahegelegt hat.
Zu guter Letzt möchte ich mich vielmals bei meinem Betreuer Dr. David Broneske bedanken. Er stand mir nicht nur ununterbrochen bis zum Schluss mit seinem Fachwissen und Rat zur Seite, sondern scheute sich vor allem nie vor zeitintensiven, stets konstruktiven Diskussionen. Diese trugen immens zur Verbesserung und zum Fortschritt der Arbeit bei.

Inhaltsverzeichnis

Abbildungsverzeichnis **xii**

Algorithmenverzeichnis **xiii**

Tabellenverzeichnis **xiii**

Codeverzeichnis **xv**

Abkürzungsverzeichnis **xvii**

1 Einleitung **1**
1.1 Beitrag dieser Arbeit 2
1.2 Gliederung der Arbeit 3

2 Grundlagen **5**
2.1 Indexstrukturen und deren Klassifikationsmöglichkeiten 5
2.1.1 Arten von Anfragen mit Selektionen 6
2.1.2 Zusammenhang zwischen Indexklassifikationen und Anfragearten 7
2.1.2.1 Ein- und Mehr-Attribut-Indexe 7
2.1.2.2 Dimensionalität von Indexen 8
2.2 Elf als multidimensionale Indexstruktur 10
2.2.1 Konzeptuelles Design des Elf 11
2.2.1.1 Dwarf als Basis des Elf 11
2.2.1.2 Aufbau des Elf im Vergleich zum Dwarf 13
2.2.2 Erstellen und Speicherung des Elf 15
2.2.2.1 Rekursion zur Erstellung des Elf 15
2.2.2.2 Linearisierung: Überführen des Baums in ein Array . 17
2.2.2.3 Abweichende Speicherung der ersten Dimension . . . 19
2.2.2.4 Eliminierung einelementiger Dimensionslisten 20
2.2.3 Anfragen mithilfe des Elf 21
2.2.3.1 Standardalgorithmus für partielle Bereichsanfragen . 21
2.2.3.2 `Cutoffs`: zusätzliche Datenstruktur zum frühen, direkten `TID`-Zugriff 27
2.2.4 Zusammmenfassung 34
2.3 SIMD: parallele Datenverarbeitung auf Instruktionsebene 35
2.3.1 Flynn'sche Taxonomie 35
2.3.2 Historische Entwicklung von SIMD 36

2.3.3 Durchführen von Operationen auf SIMD-Registern 38
2.4 Zusammenfassung 39

3 Konzeption und Implementierung 41
3.1 Programmatische Nutzung von SIMD 41
3.1.1 Herausforderungen 42
3.1.1.1 Kontrollflüsse in Schleifen 42
3.1.1.2 Limitierte Unterstützung von Datentypen und Operatoren 42
3.1.1.3 Speicheranordnung gebündelter Datensätze 42
3.1.2 Herangehensweisen zur Verwendung in Programmen 43
3.1.2.1 Automatische Vektorisierung 44
3.1.2.2 Nutzung von Intrinsiken 44
3.1.2.3 Vergleich der Methoden und Wahl für den Elf 44
3.2 SIMD zur Parallelisierung der Algorithmen im Elf 45
3.2.1 Untersuchung der Parallelisierungsmöglichkeiten 45
3.2.1.1 Scan der ersten Dimension 46
3.2.1.2 Scan der restlichen Dimensionen 46
3.2.1.3 Scan der `Monolisten` 46
3.2.2 Änderungen im Aufbau des Elf für SIMD 47
3.2.2.1 Elf als „Structure of Arrays“ 47
3.2.2.2 Explizite Speicherung der Länge von Dimensionslisten 54
3.2.3 Anpassungen der Scan-Algorithmen im Elf für SIMD 60
3.2.3.1 Abkehr von abgeschlossenen hin zu offenen Intervallen bei Bereichsanfragen 60
3.2.3.2 Datenparalleler `ScanMonolist`-Algorithmus 61
3.2.3.3 Datenparalleler `ScanDimlist`-Algorithmus 67
3.2.3.4 Datenparalleler `ScanDimlistCutoffs`-Algorithmus . 71
3.3 Zusammenfassung 75

4 Evaluierung 77
4.1 Aufbau der Experimente 77
4.1.1 Hardwareumgebung 77
4.1.2 Vorgehensweise bei der Evaluation 79
4.1.2.1 Evaluierte Anfragen 79
4.1.2.2 Evaluierte Elf-Varianten 81
4.1.2.3 Weitere Parameter 83
4.2 Experimente 84
4.2.1 Benötigter Speicherplatz 84
4.2.1.1 Ergebnisse 84
4.2.1.2 Diskussion 86
4.2.2 Erstellzeiten des Elf 87
4.2.2.1 Ergebnisse 87
4.2.2.2 Diskussion 88
4.2.3 Vergleich der Scan-Algorithmen 89
4.2.3.1 Ergebnisse 89
4.2.3.1.1 Ohne `Cutoffs` 90
4.2.3.1.2 Mit `Cutoffs` 90

4.2.3.2 Diskussion 92
4.2.3.2.1 `Elf_Separated` 92
4.2.3.2.2 `Elf_Separated_Length` 92
4.2.3.2.3 `Elf_SIMD` - Eindimensionale Anfragen . . . 93
4.2.3.2.4 `Elf_SIMD` - Mehrdimensionale Anfragen . . 95
4.2.3.2.5 Gesamtbetrachtung der neuen Elf-Varianten 96
4.2.3.2.6 `Q19`-Anfrage 97
4.2.4 Zusammenfassung 98

5 Verwandte Arbeiten **101**
5.1 Seg-Tree 101
5.2 FAST und VAST 102
5.3 ART 102
5.4 Fazit und Vergleich mit Elf 103

6 Zusammenfassung und Ausblick **105**
6.1 Zusammenfassung 105
6.2 Ausblick 106
6.2.1 Nutzung von AVX-512 106
6.2.2 Erweiterungen auf Softwareebene 107
6.2.3 Weiterführende Evaluationen 107

Literaturverzeichnis **109**

Abbildungsverzeichnis

2.1 Räumliche Auffassung von Anfragen auf einem zweidimensionalen Index 10
2.2 Beispiel-Dwarf 13
2.3 Elf für angepasste Dwarf-Beispieldaten 14
2.4 Erstellen des Elf anhand eines Beispiels 16
2.5 Linearisierter Elf (Daten) 18
2.6 Linearisierter Elf mit optimierter ersten Dimension (Daten) 19
2.7 Anteile einelementiger Dimensionslisten innerhalb des Elf für die `Lineitem`-Tabelle 20
2.8 Elf mit `Monolisten` 21
2.9 Linearisierter Elf mit optimierter ersten Dimension und `Monolisten` (Daten) 21
2.10 Suchpfade der Scan-Algorithmen bei einer Anfrage im Elf (ohne `Cutoffs`) 28
2.11 Linearisierter Elf mit `Cutoffs` (Daten) 33
2.12 Suchpfade der Scan-Algorithmen bei einer Anfrage im Elf (mit `Cutoffs`) 34
2.13 Rechnerarchitekturen der Flynn'schen Taxonomie 36
2.14 Vergleich der Ausführung einer Verlgeichsoperation auf Vektoren durch SISD und SIMD 38

3.1 Speicheranordnung gebündelter Datensätze: Zugriff auf „Array of Structures“ gegenüber „Structure of Arrays“ 43
3.2 Heterogenität der Dimensionslisteneinträge 47
3.3 Beispiel-SIMD-Operation auf heterogenen Dimensionslisten 48
3.4 Neugewonnene Homogenität der Dimensionslisteneinträge 48
3.5 Beispiel-SIMD-Operation auf homogenen Dimensionslisten 49
3.6 Linearisierter Elf als „Structure of Arrays“ (Daten) 50

3.7 Beispiel-Elf zur Datenkonstellation, die die maximale Länge von `Elf`, `Child_Pointers` und `Cutoff_Pointers` hervorruft 52
3.8 Prüfung einer Dimensionsliste gegen ein offenes Intervall mit SIMD bei Nutzung der impliziten Dimensionslistenlänge 56
3.9 Ermittlung des Dimensionslistenendes mit SIMD bei Nutzung der impliziten Dimensionslistenlänge 56
3.10 Elf mit gespeicherten Dimensionslistenlängen statt gesetzten MSBs (Daten) 57
3.11 Prüfung einer Dimensionsliste gegen ein offenes Intervall mit SIMD bei Nutzung der expliziten Dimensionslistenlänge 59
3.12 Beispielhafter Aufruf von `ScanMonolistSIMD` 66
3.13 Beispielhafter Aufruf von `ScanDimlistSIMD` 71
3.14 Beispielhafter Aufruf von `ScanDimlistCutoffsSIMD` 75

4.1 Benötigter Speicherplatz der Arrays der Elf-Implementierungsvarianten (ohne `Cutoffs`) 85
4.2 Benötigter Speicherplatz der Arrays der Elf-Implementierungsvarianten (mit `Cutoffs`) 85
4.3 Erstellzeiten der Elf-Implementierungen (ohne `Cutoffs`) 87
4.4 Erstellzeiten der Elf-Implementierungen (mit `Cutoffs`) 88
4.5 Anfragegeschwindigkeits-Speedups der neu implementierten Elf-Ansätze gegenüber `Elf64` (ohne `Cutoffs`) 91
4.6 Anfragegeschwindigkeits-Speedups der neu implementierten Elf-Ansätze gegenüber `Elf64` (mit `Cutoffs`) 91

Liste der Algorithmen

1 Verarbeitung von partiellen Bereichsanfragen im Elf: `ScanFirstDimlist` 23

2 Verarbeitung von partiellen Bereichsanfragen im Elf: `ScanDimlist` . . 25

3 Verarbeitung von partiellen Bereichsanfragen im Elf: `ScanMonolist` . . 26

4 Verarbeitung von partiellen Bereichsanfragen im Elf mit `Cutoffs`: `ScanFirstDimlistCutoffs` 30

5 Verarbeitung von partiellen Bereichsanfragen im Elf mit `Cutoffs`: `ScanDimlistCutoffs` 32

6 Verarbeitung von partiellen Bereichsanfragen im Elf unter Zuhilfenahme von SIMD: `ScanMonolistSIMD` 63

7 Verarbeitung von partiellen Bereichsanfragen im Elf unter Zuhilfenahme von SIMD: `ScanDimlistSIMD` 68

8 Verarbeitung von partiellen Bereichsanfragen im Elf unter Zuhilfenahme von SIMD mit `Cutoffs`: `ScanDimlistCutoffsSIMD` 73

Codeverzeichnis

2.1 Lineitem-Anfrage ohne Selektion 6

2.2 TPC-H Q10 6

2.3 TPC-H Q6 6

2.4 Exakte Mehr-Attribut-Lineitem-Anfrage (eindimensionaler Index) . . 8

2.5 Partielle Mehr-Attribut-Lineitem-Anfrage (eindimensionaler Index) . 8

4.1 TPC-H Q1 81

4.2 TPC-H Q14 81

4.3 TPC-H Q17 81

4.4 TPC-H Q19 auf Lineitem 81

4.5 TPC-H Q19 auf Part 81

Abkürzungsverzeichnis

AoS „Array of Structures“-Speicherlayout, bei dem zusammenhängende Datensätze (*Records*) innerhalb eines Arrays gespeichert werden (Details siehe Abschnitt 3.1.1.3).

AVX Advanced Vector Extensions

CPU Central Processing Unit

DBMS Datenbankmanagementsystem

DBS Datenbanksystem

FF Forschungsfrage

MIMD Multiple Instruction, Multiple Data

MISD Multiple Instruction, Single Data

MMX Multi Media Extension

MSB most significant bit

OLAP Online-Analytical-Processing

OLTP Online-Transaction-Processing

RAM Random-Access Memory

SIMD Single Instruction, Multiple Data

SISD Single Instruction, Single Data

SLS Die `SIMD_LINE_SIZE`: für die Implementierung relevante Anzahl der 32-Bit-Integer-Werte, die in ein einzelnes SIMD-Register geladen werden können.

SoA „Structure of Arrays“-Speicherlayout, bei dem zusammenhängende Datensätze (*Records*) nach ihren einzelnen Attributen getrennt in separaten Arrays gespeichert werden. Pendant zu AoS (Details siehe Abschnitt 3.1.1.3).

SQL	Structured Query Language
SSE	Streaming SIMD Extensions
SSSE	Supplemental Streaming SIMD Extensions
TMSI	Kurz für ***t****rue* ***m****ask* *SIMD of the current* ***i****teration*. Eine besondere Maske, die im Zuge der SIMD-basierten Scan-Algorithmen benötigt wird.
TPC	Transaction Processing Performance Council – eine Non-Profit-Organisiation, die standardisierte Benchmarks für DBMSs entwickelt hat, um eine vergleichbare Aussage über deren Leistungsfähigkeit hinsichtlich der Anfrage- und Transaktionsverarbeitung treffen zu können.
TPC-H	Ein spezieller Anfragebenchmark des TPC für Decision-Support, das vor allem durch die Verarbeitung von großen Datenmengen gekennzeichnet ist [Tra18].

1. Einleitung

Die Steigerung der Performanz eines Softwaresystems ist ein beständig aktuelles Bestreben – Datenbankmanagementsysteme (DBMS) bilden dabei keinesfalls eine Ausnahme. Auf abstrakter Ebene kann „Performanz" im Kontext eines DBMS als die Interaktionsgeschwindigkeit des Systems mit Anwendungen und Anwendern interpretiert werden. Für das Erfüllen dieser doch eher vagen Zielstellung gibt es eine Vielzahl von Mitteln, welche wiederum von einer Reihe von Faktoren abhängig sind, die sich zusammen mit den technischen Gegebenheiten immer im Wandel befinden.

Ein prominentes Beispiel hierfür ist der Flaschenhals, der sich aus der sogenannten Zugriffslücke zwischen persistentem Platten- und flüchtigem Arbeitsspeicher ergibt. Dieser bestand seit der Einführung von relationalen DBMS in den 1960er Jahren jahrzehntelang. Dementsprechend waren Datenbankoptimierungen häufig darauf ausgelegt, relevante Daten schnellstmöglich und klug in den Hauptspeicher zu laden und dort zu halten [HAMS08]. Mit dem Aufkommen von Hauptspeicherdatenbanken wie SAP HANA oder MonetDB wurde jener gesamte Optimierungskomplex de facto irrelevant [FCP+12, IGN+12].
Diese Art von Datenbanken ermöglicht ein permanentes Halten der gesamten Datenbank im RAM. Im Gegenzug ergeben sich natürlich neue Herausforderungen wie der Zugriff auf die sich im Hauptspeicher befindlichen Daten durch die CPU und dem damit einhergehenden Thema des bewussten Umgangs mit den verschiedenen CPU-Caches [MBK00]. Zwischen diesem und dem RAM besteht zwar eine bei weitem nicht so große Zugriffslücke wie zwischen Haupt- und Plattenspeicher (Faktor 10 im Vergleich zu Faktor 10^5, siehe Tabelle 1.1), nichtsdestotrotz gilt es, sie zu überwinden.
Außer der Unterschiede bezüglich der Zugriffszeiten wird in Tabelle 1.1 eine weitere Diskrepanz zwischen den verschiedenen Speicherebenen deutlich: ihre Kapazität. Unter Berücksichtigung dieser ist naheliegend, dass Datenbankoperatoren davon profitieren, wenn Zwischenergebnisse bei der Bearbeitung von Anfragen möglichst klein gehalten werden. Hierfür bietet sich insbesondere die bewährte Heuristik des frühen Ausführens von *Selektionen* an, weil diese erheblichen Einfluss auf die Größe der Daten besitzen, die von weiterführenden Operationen wie dem Verbund genutzt werden können. Gerade durch die im Zuge von Big-Data- und OLAP-Anwendungen

immer größer werdenden Datenmengen kommt dem Selektionsoperator eine besondere Bedeutung zu [KSS14].

Speicherart	typische Zugriffszeit	typische Kapazität
CPU-Cache	6ns	256 KB (L2) bis 32 MB (L3)
Hauptspeicher	60ns	1GB bis 1,5 TB
	——— Zugriffslücke (10^5) ———	
Magnetplattenspeicher	12ms	160GB bis 4 TB
Platten-Farm/-Array	12ms	im TB- bis PB-Bereich

Tabelle 1.1: Zugriffszeiten und Kapazitäten innerhalb der Speicherhierarchie (aus [SSH18])

In OLAP-Szenarien nimmt darüber hinaus die Multidimensionalität der Daten eine tragende Rolle ein. Das wird deutlich, wenn man sich die Anfragen eines gängigen Datenbankbenchmarks wie dem TPC-H-Benchmark ansieht: hier stellen Selektionen auf mehreren Spalten gleichzeitig die Regel dar.
Zur effizienten Bearbeitung genau solcher Selektionen wurde von Broneske et al. die multidimensionale Indexstruktur Elf vorgestellt [BKSS17]. Obgleich Elf bereits von Beginn an mit einigen Optimierungen präsentiert worden ist, waren diese Verbesserungen hauptsächlich datenstruktureller Art und somit auf der Softwareseite angesiedelt.
Zentraler Untersuchungsschwerpunkt und Beitrag dieser Arbeit soll eine hardwarebasierte Optimierung des Elf sein.

1.1 Beitrag dieser Arbeit

Mittelpunkt des hardwarebasierten Verbesserungsvorschlags sollen die fortgeschrittenen Möglichkeiten der datenparallelen, vektorbasierten Verarbeitung durch moderne CPUs darstellen. Konkret wird SIMD (kurz für ***S**ingle **I**nstruction, **M**ultiple **D**ata*) zur intraprozessualparallelen Durchführung der existierenden Anfragealgorithmen des Elf genutzt. Es sollen folgende Forschungsfragen (kurz FF) in dieser Arbeit beantwortet werden:

FF 1. Welche Änderungen...

(a) an der Struktur und

(b) an den Scan-Algorithmen

der Standardimplementierung des Elf von Broneske et al. [BKSS17] sind zur effizienten Nutzung von SIMD ratsam?

FF 2. Wie schneidet die aus **FF** 1. hervorgehende neue Variation des Elf, genannt `Elf_SIMD`, gegenüber der Ursprungsimplementierung ab in Bezug auf...

FF 1. den Speicherverbrauch der erstellten Indexstruktur,

FF 2. die benötigte Zeit zur Erstellung der Indexstruktur und

FF 3. die benötigte Zeit zur Bearbeitung von Anfragen des TPC-H-Benchmarks?

1.2 Gliederung der Arbeit

Zu Beginn der Arbeit sollen im Kapitel 2 die Grundlagen für die zwei zentralen Themen dieser Arbeit gelegt werden: der Elf als multidimensionale Indexstruktur und SIMD werden vorgestellt. Dabei werden zunächst Indexstrukturen allgemein erläutert, ehe Elf als spezieller Vertreter dieser präsentiert wird. Abschließend wird SIMD als Mittel zur datenparallelen Verarbeitung auf Instruktionsebene behandelt.

In Kapitel 3 werden die beiden Schwerpunkte zusammengeführt. Dazu werden im Voraus die allgemeine Herausforderungen bei der programmatischen Nutzung von SIMD beschrieben. Mit diesem Wissen sind die konkreten Anwendungsbereiche von SIMD für die Algorithmen des Elf und deren Voraussetzungen nachvollziehbar. Durch das Herausarbeiten der vorgenommenen Änderungen wird mit Abschluss von Kapitel 3 die erste wissenschaftliche Fragestellung dieser Arbeit beantwortet.
Das anknüpfende Kapitel 4 dient der Behandlung der zweiten Forschungsfrage. Es wird der Aufbau und Ablauf der dazu durchgeführten Experimente erklärt, bevor die Ergebnisse dieser analysiert und diskutiert werden.
Den Abschluss der Arbeit bilden die Kapitel 5 und 6. Ersteres schafft einen kurzen Überblick über relevante, verwandte Arbeiten. Kapitel 6 fasst die gewonnenen Erkenntnisse zusammen und gibt einen Ausblick bezüglich noch zu klärender Sachverhalte.

2. Grundlagen

Zur Beantwortung der Forschungsfragen und zum grundlegenden Verständnis dieser, sollen vorab mit diesem Kapitel Hintergrundwissen und Voraussetzungen geschaffen werden. Im weiteren Verlauf werden werden die zwei Kernthemen Elf und SIMD detailliert erklärt und motiviert.
Zum Verständnis der Konzepte und Anwendungsgebiete des Elf ist es sinnvoll, die Ausführungen eine Ebene höher zu beginnen: bei den Indexstrukturen.

2.1 Indexstrukturen und deren Klassifikationsmöglichkeiten

Es gibt im Wesentlichen zwei Mittel, um auf die Daten einer Datenbanktabelle zuzugreifen: per Relationenscan (engl. *full table scan*) oder unter Zuhilfenahme von Indexstrukturen (engl. *index scan*) [SSH18]. Bei einem Relationenscan wird eine Tabelle so durchlaufen wie sie im Speicher- (intern) und dem darauf verweisenden Zugriffssystem (logisch) des Datenbanksystems (DBS) abgespeichert ist. Dieses Vorgehen ist für Anfragen ohne Selektion – wie in Codeauszug 2.1 dargestellt – naheliegend. Betrachtet man jedoch beispielsweise die Anfragen aus dem TPC-H-Benchmark, die realistische, das heißt praxisrelevante, Systemlasten nachstellen sollen, wird schnell deutlich, dass SQL-Anfragen ohne Einschränkungen in der `WHERE`-Klausel die Ausnahme bilden. Tatsächlich gibt es keine Query in diesem Benchmark ohne Selektionsbedingung. Die Codeauszüge 2.2 und 2.3 zeigen die TPC-H-Anfragen `Q10` und `Q6`, die durch ein Selektionskriterium respektive mehrere per Konjunktion verknüpften Selektionskriterien gekennzeichnet sind. In der Regel lassen sich solche Anfragen mithilfe des zweiten Zugriffsinstruments, den Indexstrukturen, anstelle eines vollständigen Relationenscans beschleunigen [SSH18].

```
SELECT *
  FROM Lineitem
```

Codeauszug 2.1: `Lineitem`-Anfrage ohne Selektion

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

Codeauszug 2.2: Anfrage mit einem Selektionskriterium (`Q10`)

```
SELECT *
  FROM Lineitem
 WHERE l_shipdate ≥ [DATE] AND l_shipdate < [DATE] + '1Year'
   AND l_discount between [DISCOUNT] − 0.01 and [DISCOUNT] + 0.01
   AND l_quantity < [QUANTITY]
```

Codeauszug 2.3: Anfrage mit mehreren Selektionskriterien (`Q6`)

Indexe können als Zugriffsstrukturen definiert werden, die parallel zur eigentlichen Speicherstruktur einer Relation für ein oder mehrere Attribute gehalten werden und einen schnelleren Zugriff auf die Daten ermöglichen sollen. Ein Index besteht aus einem Suchschlüssel K, dem mindestens ein Datensatz oder Verweis auf diesen ($K \uparrow$) zugeordnet wird. Ist diese Abbildung bijektiv, zeigt K also auf genau ein $K \uparrow$, spricht man von einem Primärindex, wohingegen ein Verweis von K auf ein oder mehrere $K \uparrow$ als Sekundärindex bezeichnet wird [SSH18].
Über diese Differenzierung hinaus, lassen sich Indexe aufgrund weiterer Charakteristika klassifizieren. Um die sich unterscheidenden Anwendungsfälle der im Anschluss definierten Klassifikationen hervorzuheben, ist vorangehend eine Betrachtung möglicher Typen von Anfragen mit Selektionen sinnvoll.

2.1.1 Arten von Anfragen mit Selektionen

Durch ein oder mehrere Selektionen können in einer Anfrage Q für bis zu k Suchschlüssel Qualifikationsbedingungen spezifiziert werden. Erfüllen Schlüsselwerte diese Anforderungen, können sie in die Ergebnismenge von Q aufgenommen werden. Für derartige Anfragen lassen sich die folgenden Arten herausstellen, wobei zur Vereinfachung von numerischen Attributen ausgegangen wird [HR01]:

1. **Exakte Anfragen** (engl. *exact match query*): Jeder von k Suchschlüsseln (A) nimmt einen Wert a_i innerhalb der selektierenden Anfrage Q an.
   $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots \wedge (A_k = a_k)$

2. **Partielle Anfragen** (engl. *partial match query*): Lediglich s Suchschlüssel erhalten eine Wertzuweisung.
   $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots \wedge (A_s = a_s)$, wobei $s < k$

3. **Bereichsanfragen** (engl. *range query*): Für jeden Suchschlüssel wird ein Bereich r_i von Werten spezifiziert.
   $Q = (A_1 \in r_1) \wedge (A_2 \in r_2) \wedge \ldots \wedge (A_k \in r_k)$

4. **Partielle Bereichsanfragen** (engl. *partial range query*): Lediglich s Suchschlüssel erhalten eine Bereichseinschränkung.
   $Q = (A_1 \in r_1) \wedge (A_2 \in r_2) \wedge \ldots \wedge (A_s \in r_s)$, wobei $s < k$

Mithilfe zweier Anpassungen kann man mit partiellen Bereichsanfragen alle anderen Anfragen substituieren.
Zum einen lassen sich reine Bereichsanfragen abbilden, indem man die Vorgabe $s < k$ zu $s \leq k$ auflockert.
Zum anderen können die Prädikate $A_i = a_i$ von exakten und partiellen Anfragen auch als Bereiche repräsentiert werden. Dazu muss lediglich die untere und obere Grenze eines Bereichs r_i auf a_i gesetzt werden – also $(A_i = a_i) \equiv (A_i \in r_i) \Leftrightarrow r_i = [a_i, a_i]$. In Tabelle 2.1 sind alle denkbaren Prädikate für ganzzahlige Attribute und ihre Repräsentation als Bereiche beziehungsweise Intervalle wiederzufinden. *max* und *min* sind dabei die Maxima respektive Minima der Wertebereiche der Attribute. Diese Beobachtungen spielen für die Art der im Zuge dieser Arbeit implementierten Scan-Algorithmen eine wesentliche Rolle.

Prädikat	Intervallrepräsentation
$= a$	$[a, a]$
$< a$	$[min, a) \equiv [min, a-1]$
$\leq a$	$[min, a]$
$> a$	$(a, max] \equiv [a+1, max]$
$\geq a$	$[a, max]$
$\leq a \wedge \geq b, a \leq b$	$[a, b]$

Tabelle 2.1: Ganzzahlige Prädikate und ihre Intervallrepräsentationen (in Anlehnung an Broneske et al. [BKSS17], angepasst an die hier verwendete Notation)

2.1.2 Zusammenhang zwischen Indexklassifikationen und Anfragearten

Unter Berücksichtigung der soeben definierten Anfragearten lassen sich nun die weiteren Indexklassifikationen und deren Verwendungszwecke konkreter erläutern. Für diese Arbeit sind vor allem die Unterscheidungen nach Ein- und Mehr-Attribut-, sowie damit einhergehend nach ein- und multidimensionalen Indexstrukturen wesentlich.

2.1.2.1 Ein- und Mehr-Attribut-Indexe

Bei Ein-Attribut-Indexen wird die Indexstruktur für genau ein Attribut der Relation generiert und genutzt, während Mehr-Attribut-Indexe mindestens zwei Attribute gleichzeitig indexieren. Mehr-Attribut-Indexe sind besonders dann von Vorteil,

wenn Anfragen häufig alle im Index inkludierten Attribute gleichzeitig einschränken – im Optimalfall in Form einer exakten oder bereichseinschränkenden Anfrage für ebendiese Spalten. Existiert beispielsweise für `Q6` aus dem TPC-H-Benachmark (siehe Codeauszug 2.3) ein Mehr-Attribut-Index über `l_shipdate`, `l_discount` und `l_quantity`, muss auf den Index nur einmal zugegriffen werden. Gibt es hingegen lediglich drei Ein-Attribut-Indexe auf den genannten Spalten, so müsste auf drei Indexe zugegriffen und anschließend eine Konjunktion der entstandenen drei Ergebnismengen gebildet werden.

2.1.2.2 Dimensionalität von Indexen

Inwieweit ein Mehr-Attribut-Index für partielle Anfragen geeignet ist, hängt von dessen Dimensionalität ab. Diese beschreibt die Ordnung des Raumes, in welchem die Schlüssel eines Indexes gesucht werden. Ein-Attribut-Indexe sind immer eindimensional, wobei Mehr-Attribut-Indexe ein- und mehrdimensional[1] sein können [SSH18].
Sei für die `Lineitem`-Tabelle ein Mehr-Attribut-Index über den Spalten `l_returnflag` und `l_quantity` definiert.

Eindimensionale Indexe

Betrachten wir zunächst den eindimensionalen Fall. Hier werden die Kombinationen der Werte der verschiedenen Attribute, `l_returnflag` und `l_quantity`, konkateniert. In Tabelle 2.2 ist der Suchschlüssel K_{1d} für Beispieldaten mit einer Raute als Trennzeichen zwischen den Attributen dargestellt. `LID` ist als ein künstlicher Primärschlüssel gewählt, der den tatsächlichen der `Lineitem`-Tabelle, welcher sich zusammensetzt aus `l_orderkey`, `l_partkey` und `l_suppkey`, vereinfacht repräsentieren soll.

```
SELECT *
  FROM Lineitem
 WHERE l_returnflag = 1
   AND l_quantity = 4
```

Codeauszug 2.4: Exakte Mehr-Attribut-`Lineitem`-Anfrage

```
SELECT *
  FROM Lineitem
 WHERE l_quantity = 4
```

Codeauszug 2.5: Partielle Mehr-Attribut-`Lineitem`-Anfrage

Exakte Anfrage auf den beiden Spalten `l_returnflag` und `l_quantity` wie in Codeauszug 2.4 lassen sich effizient mit diesem eindimensionalen Index verwirklichen, indem die Suchattributwerte konkateniert und das Resultat dessen in der sortiert vorliegenden Indexstruktur gesucht wird. Genauso ist `Q10` (siehe Codeauszug 2.2) aufgrund ebendieser Sortiertheit des Indexes mithilfe einer Präfixsuche gut abbildbar. Im Gegensatz dazu ist eine partielle Anfrage, die nur `l_quantity`, also einen Suffix des Suchschlüssels K anspricht (siehe Codeauszug 2.5), nicht effizient mit einem eindimensionalen Index umsetzbar. Grund ist hierfür, dass die Ordnung des

[1] Die Begriffe „multidimensional“ und „mehrdimensional“ sind synonym zu betrachten.

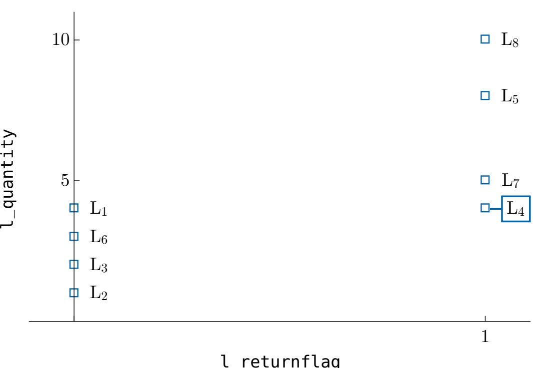
LID	...	l_quantity	...	l_returnflag	...	K_{1d}	K_{2d}
L_2	...	1	...	0	...	0#1	(0,1)
L_3	...	2	...	0	...	0#2	(0,2)
L_6	...	3	...	0	...	0#3	(0,3)
L_1	...	4	...	0	...	0#4	(0,4)
L_4	...	1	...	1	...	1#1	(1,1)
L_7	...	4	...	1	...	1#4	(1,4)
L_5	...	5	...	1	...	1#5	(1,5)
L_8	...	10	...	1	...	1#10	(1,10)

Tabelle 2.2: Beispiel eines eindimensionalen Mehr-Attribut-Indexes bei der `Lineitem`-Relation des TPC-H-Benchmarks

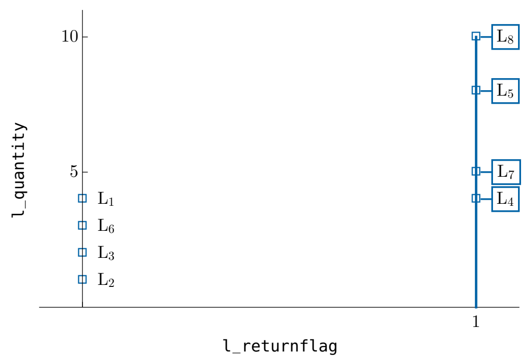
zweiten Attributs immer von der des ersten abhängig ist, wodurch ein „Pruning“ von Suchalgorithmen nicht genutzt werden kann.

Multidimensionale Indexe

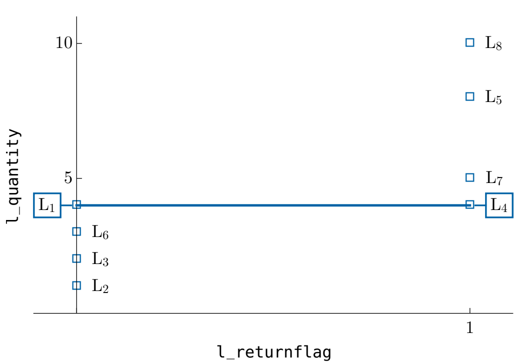
Den Nachteil der Relevanz der Reihenfolge der Indexattribute eines eindimensionalen Indexes sollen mehrdimensionale Indexstrukturen egalisieren. Ein multidimensionaler n-Attribut-Index spannt nämlich über die Ausprägungen seiner Attribute einen n-dimensionalen Raum auf.
Sei der Zwei-Attribut-Index über `l_returnflag` und `l_quantity` nun nicht mehr ein-, sondern zweidimensional. Damit wird durch ihn jetzt ein zweidimensionaler Raum erzeugt wird. Dieser ist in Abbildung 2.1 dargestellt. Auf der Abszisse ist `l_returnflag` mit den möglichen Werten 0 und 1 und auf der Ordinate `l_quantity` mit Werten in einem Bereich von 1 bis 10 wiederzufinden. Die Punkte im Raum stellen die Suchschlüssel K_{2d} auf Basis der Daten von Tabelle 2.2 dar. Gekennzeichnet werden die Punkte im Diagramm mithilfe des von ihnen referenzierten, künstlichen Primärschlüssels L_i der `Lineitem`-Relation. Beim Suchschlüssel $(0, 1)$, der im Beispiel auf L_2 aus der ersten Zeile von Tabelle 2.2 abbildet, entspricht – gemäß der zuvor verwendeten Notation bei der Indexdefinition – L_2 dem Datensatzverweis $K \uparrow$ des Suchschlüsselpaars $K_{2d} = (0, 1)$.
Bei einer exakten Anfrage wird somit genau ein Punkt aus diesem Datenraum angesprochen wird. In Abbildung 2.1 a ist dies für die Anfrage aus Codeauszug 2.4 abgebildet. Hervorgehoben ist hier L_4 als Ergebnismenge.
Im Gegensatz dazu erzeugen partielle Anfragen wie Codeauszüge 2.2 und 2.5 aufgrund ihrer Selektionskriterien Hyperebenen im Raum. Dementsprechend entstehen im zweidimensionalen Fall horizontale und vertikale Geraden, bei denen die sich auf diesen befindlichen Punkte die Ergebnismenge bilden. Im Beispiel entsteht im Datenraum bei der Einschränkung von `l_returnflag` wie in Codeauszug 2.3 eine Senkrechte bei `l_returnflag` $= 1$ (siehe Abbildung 2.1 b) und bei der Selektion von Codeauszug 2.5 auf `l_quantity` eine Waagerechte durch `l_quantity` $= 4$ (siehe Abbildung 2.1 c).



(a) Exakte Anfrage: Selektion auf beiden Indexattributen (siehe Codeauszug 2.4)



(b) Partielle Anfrage: Selektion auf dem erstem Indexattribut (siehe Codeauszug 2.2)



(c) Partielle Anfrage: Selektion auf dem zweitem Indexattribut (siehe Codeauszug 2.5)

Abbildung 2.1: Räumliche Auffassung von Anfragen auf einem zweidimensionalen Index

Das verdeutlicht, dass es – anders als im eindimensionalen Fall – eine untergeordnete Rolle spielt, auf welchen der indexierten Attribute die Selektion erfolgt. Ist es für die Performanz gänzlich irrelevant, in welcher Reihenfolge man die Attribute eines multidimensionalen Indexes eingrenzt, ist dieser darüber hinaus *symmetrisch* [KSS14]. Ein zweidimensionaler Index über `l_returnflag` und `l_quantity` kann also alle exemplarischen Anfragen mit diesen beiden Spalten (Codeauszüge 2.2, 2.4 und 2.5) effizient durchführen; ob die Selektionen mit nur einer der beiden Spalten `l_returnflag` beziehungsweise `l_quantity` jedoch gleich gut unterstützt werden, hängt von der Symmetrie der gewählten Indexstruktur ab.

2.2 Elf als multidimensionale Indexstruktur

Nachdem in dem vorangegangenen Abschnitt 2.1 Indexstrukturen allgemein und deren multidimensionale Vertreter im Speziellen mit ihren Vorteilen beschrieben worden sind, soll in diesem Abschnitt ein besonderer multidimensionale Mehr-Attribut-Index, der Elf, zunächst konzeptionell erläutert werden. Im weiteren Verlauf wird erklärt, wie der Elf erstellt, optimiert gespeichert und letztendlich für Anfragen genutzt werden kann.

2.2.1 Konzeptuelles Design des Elf

Nicht nur namentlich, sondern auch aufgrund der Herangehensweise bei der Speicherung von Daten schafft der Dwarf die Basis des Elf. Daher sollen die Idee und die daraus resultierenden Vorteile des Dwarf im folgenden Abschnitt näher erläutert werden und daraufhin ein einleitender Vergleich mit dem Elf gezogen werden.

2.2.1.1 Dwarf als Basis des Elf

Im Gegensatz zu einer Indexstruktur, die parallel zu den eigentlichen Daten einer Relation gehalten wird (siehe Abschnitt 2.1), beschreibt eine Speicherstruktur, wie diese Daten grundsätzlich intern organisiert sind. Es ist hierbei, im Gegenteil zu einigen Indexstrukturen, keine Transformation der in der Struktur gespeicherten Werte zur Rekonstruktion von Tupeln notwendig.
In ihrer Arbeit *Dwarf: Shrinking the PetaCube* stellen Sismanis et al. einen Ansatz vor, welcher auf die bis zu mehrere Petabytes große Data-Warehouse-Faktentabelle und dem darauf angewandten `Cube`-Operator abgestimmt ist [SDRK02]. Der Dwarf ist primär als Speicherstruktur angedacht, kann aber gleichermaßen als Index umgesetzt und genutzt werden kann. Das Hauptziel der Arbeit war es, sowohl die Daten als solche als auch deren Aggregationen durch den `Cube`-Operator komprimiert zu speichern, um zum einen den beanspruchten Speicherplatz zu reduzieren und zum anderen eine optimierte Ausführung des Operators zu gewährleisten.

Aufbau des Dwarf

In Tabelle 2.3 ist eine Relation dargestellt, bei der sowohl deren Tupel als auch die Ergebnisse des `Cube`-Operators in komprimierter Form in den Dwarf übertragen werden sollen. Im `Cube` befinden sich alle Kombinationen von Teil- und Gesamtaggregationen gruppiert nach den spezifizierten Attribute. Da alle Daten der Relation gespeichert werden sollen, muss der `Cube` auch dementsprechend über alle Attribute dieser gebildet werden. Das Ergebnis dessen ist in Tabelle 2.4 dargestellt; Aggregationen in Form von Summen werden hier über den Fakt Preis getätigt. `ALL`-Werte beschreiben die Super-Aggregate der jeweiligen Spalte [GCB+97].

Region	Kunde	Produkt	Preis
R1	K2	P2	70 €
R1	K3	P1	40 €
R2	K1	P1	90 €

Tabelle 2.3: Beispieldaten Dwarf (in Anlehnung an [SDRK02, KSS14])

Der Dwarf zu diesen Daten ist in Abbildung 2.2 dargestellt.
Beim Dwarf handelt es sich um einen gerichteten, azyklischen Graphen mit n Ebenen, wobei n der Dimension des `Cube`s entspricht (hier vier). Ein Blattknoten enthält immer einen Schlüssel der letzten Dimension sowie dessen Aggregationswert. In den

anderen Ebenen befinden sich in den Knoten die Schlüssel der Dimension sowie von diesen ausgehend Zeiger, die auf die darunterliegenden, korrespondierenden Kindknoten zeigen. Die Zeiger definieren *Pfade*, welche von je einem Nicht-Blattknoten beginnend einen Weg zu einem Blattknoten beschreiben. Darüber hinaus besitzt jeder der inneren Knoten die für den `Cube`-Operator benötigte `ALL`-Zelle [SDRK02]. In der letzten Spalte von Tabelle 2.4 wird die Verbindung zwischen den Ergebnissen innerhalb des `Cube`s und dem Graphen (Abbildung 2.2) hergestellt.

Region	Kunde	Produkt	Preis	Dwarf-Knoten
R1	K2	P2	70 €	3
R1	K3	P1	40 €	2
R2	K1	P1	90 €	4
R1	K2	ALL	70 €	3
R1	K3	ALL	40 €	2
R2	K1	ALL	90 €	4
R1	ALL	P2	70 €	1
R2	ALL	P1	90 €	4
ALL	K1	P1	90 €	4
ALL	K2	P2	70 €	3
ALL	K3	P1	40 €	2
R1	ALL	ALL	110 €	1
R2	ALL	ALL	90 €	4
ALL	K1	ALL	90 €	4
ALL	K2	ALL	70 €	3
ALL	K3	ALL	40 €	2
ALL	ALL	P1	130 €	5
ALL	ALL	P2	70 €	5
ALL	ALL	ALL	200 €	5

Tabelle 2.4: Cube-Operator im Dwarf (in Anlehnung an [SDRK02, KSS14])

Vorteile des Dwarf

Durch die Art der Speicherung im Dwarf ergeben sich vor allem vier Eigenschaften und damit einhergehend Vorteile, wovon – mit Ausnahme der Suffixredundanzeliminierung – alle wesentlich für die später beschriebene Adaptation durch den Elf sind [BKSS17]:

1. **Präfixredundanzeliminierung**: Jeder eindeutige Wert kommt nur einmal in einem Knoten einer Dimension vor. Das impliziert, dass jeder Präfix von Attributwerten entlang eines Pfades nur einmal gespeichert wird. Diese Eigenschaft ist vor allem für dicht besetzte Datenmengen (engl. *dense areas*) nützlich.

2. **Suffixredundanzeliminierung**: Teilen sich Knoten einer Dimension im Dwarf denselben Suffix, zeigen sie auch auf dieselben Kindknoten – der Suffix ist nur einmal im Dwarf zu finden. Das tritt in ausgeprägterer Weise für Dimensionen auf, bei denen die darauf folgende dünn besetzt (engl. *sparse areas*) ist.

3. **Sortierte Knotenelemente**: Die Werte innerhalb der Knoten liegen sortiert vor. Das erlaubt effiziente, die Ordnung ausnutzende Suchalgorithmen beim Auslesen der Struktur.

4. **Feststehende Suchtiefe**: Zur vollständigen Tupelrekonstruktion sind genau n Knoten respektive Dimensionen zu durchlaufen. Dadurch lässt sich eine obere Grenze bezüglich der Suchkosten festlegen, die lediglich von der Anzahl der Dimensionen und nicht von der der Tupel abhängt.

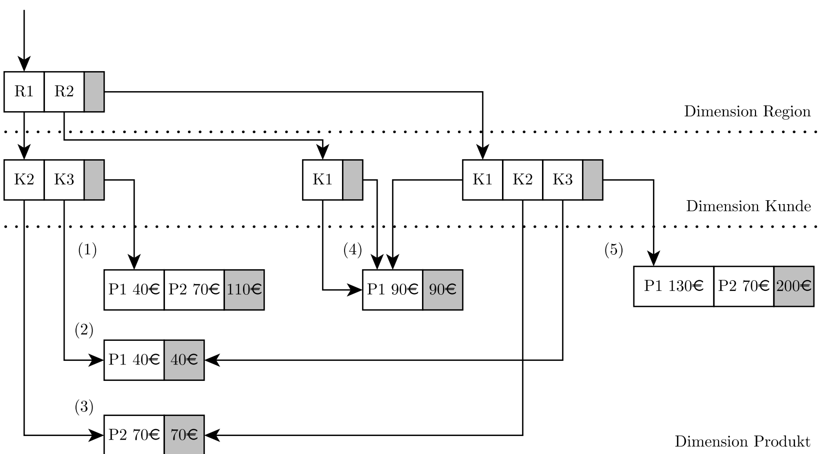


Abbildung 2.2: Beispiel-Dwarf (in Anlehnung an [SDRK02, KSS14])

2.2.1.2 Aufbau des Elf im Vergleich zum Dwarf

Der wesentliche Unterschied des Elf gegenüber des Dwarf ist das Wegfallen der `Cube`-spezifischen Optimierungen. Das hat zur Konsequenz, dass die `ALL`-Zellen (grau dargestellt in Abbildung 2.2) sowie deren Kinder (unter anderem Knoten (1) und (5) in Abbildung 2.2) entfallen. Dadurch ist der Elf wiederum in seiner „Urform“ konzeptionell betrachtet keine graph-, sondern eine baumbasierte Struktur. Denn im Gegensatz zum Knoten eines Graphens kann der eines Baums keine zwei eingehenden Kanten besitzen; dies ist im Dwarf eben wegen der `ALL`-Zellen möglich gewesen. Das erklärt auch den Wegfall der Suffixredundanzeliminierungseigenschaft: gleiche Suffixe treten im Elf mehrfach auf, da jeder Knoten immer nur eine eingehende Kante haben kann. Im Elf werden zudem vorrangig Verweise auf Tupel und

keine Fakten gespeichert – ein Indikator dafür, dass der Elf primär als Indexstruktur gedacht ist.
Die Ebenen des Elf sind die Dimensionen (Attribute) der zu indexierenden Tabelle. Die Knoten des Baums sollen fortan als `Dimensionslisten` bezeichnet werden . Diese beinhalten die Werte der jeweiligen Dimension sowie Zeiger zu möglichen Suffixen, die mit Werten der nächsten Dimension beginnen. Das Konzept eines Pfades und der dadurch implizierten Präfixe ist genauso im Elf wiederzufinden.
Unter der Annahme, dass statt des Preises als Fakt die einzelne Tupel eindeutig identifizierende `TID` (engl. *tuple identifier*) gespeichert wird, ergibt sich für die Datengrundlage des Dwarfs (Tabelle 2.3) ein Elf wie in Abbildung 2.3 dargestellt; die Daten für den Elf können Tabelle 2.5 entnommen werden.

Region	Kunde	Produkt	`TID`
R1	K2	P2	T1
R1	K3	P1	T2
R2	K1	P1	T3

Tabelle 2.5: Dwarf-Beispieldaten, angepasst für den Elfen.

Im Folgenden sind die Wirkungen der Eigenschaften des Elf anhand des Beispiels erläutert. Aufgrund der Einfachheit der Datengrundlage, fallen diese allerdings noch überschaubar aus:

1. **Präfixredundanzeliminierung**: Die Tupel mit der `TID` T1 und T2 teilen sich einen Präfix infolge ihrer gleichen Ausprägung (R1) für das Attribut Region. R1 wird nur einmal, nicht redundant im Elf gespeichert.

2. **Sortierte Knotenelemente**: Die Dimensionsliste des Kundenattributs mit dem Präfix Region = R1 liegt in sortierter Reihenfolge vor – K3 folgt auf K2.

3. **Feststehende Suchtiefe**: Die Suchtiefe ist bei allen Knoten von der Wurzel ausgehend drei.

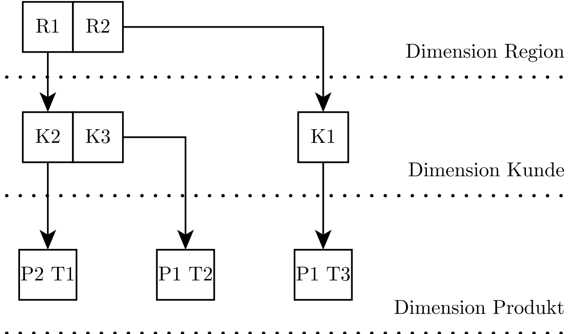


Abbildung 2.3: Elf für angepasste Dwarf-Beispieldaten (Tabelle 2.5)

2.2.2 Erstellen und Speicherung des Elf

Nachdem die Basis, der Dwarf, sowie das konzeptionelle Layout des Elf und dessen Vorteile im vorangegangenen Abschnitt 2.2.1 nahegebracht worden sind, soll nun die Umsetzung des Elf und damit einhergehend Optimierungen bei seiner programmatischen Speicherung erläutert werden. Daran anschließend wird vorgestellt, wie die Algorithmen aussehen, mit denen partielle Anfragen auf dem Elf durchgeführt werden können.
Für die Erläuterungen im folgenden Kapitel soll ein sogleich beschriebenes Beispiel als Grundlage dienen, welches im Vergleich zu dem vorangegangenen etwas komplexer ist.

2.2.2.1 Rekursion zur Erstellung des Elf

Der Elf setzt sich vereinfacht gesprochen aus Dimensionslisten zusammen, die über Zeiger miteinander verknüpft sind. Die Zeiger verweisen dabei immer von den Listenwerten einer Dimension auf den Beginn der dazugehörenden Dimensionsliste der nächsten Dimension.

Allgemeines Vorgehen

Bei der Erstellung des Elf wird mit der ersten zu indexierenden Spalte begonnen. Nachdem diese aufsteigend sortiert worden ist, werden ihre eindeutigen Werte in die erste Dimensionsliste aufgenommen, welche als Wurzel und somit bei späteren Suchen als Einstiegspunkt dienen wird. Für jeden der ermittelten Werte werden deren Suffixe, beginnend mit der nächsten indexierten Dimension, rekursiv betrachtet. Die sich daraus ergebenden Teilmengen werden wiederum nach der nächsten Dimension sortiert und deren eindeutige Werte ermittelt. Daraufhin erfolgt für diese erneut ein rekursiver Aufruf. Das Abbruchkriterium einer Rekursion ist das Erreichen der letzten Dimension.
Das Vorgehen beim Erstellen des Baums nimmt sich die Tiefensuche in Pre-Order-Form als Vorbild. Das heißt es wird immer zunächst die Wurzel eines Teilbaums und darauf folgend ihr linkes Kind vollständig und erst danach das rechte erzeugt.

Vorgehen anhand eines Beispiels

Zur beispielhaften Erläuterung ist in Tabelle 2.6 eine (unsortierte) Datengrundlage zu finden. Das Ergebnis der vollständigen Sortierung kann man Tabelle 2.7 entnehmen[1]. Für den Teilbaum, der sich für den ersten Wert (0) der ersten Dimensionsliste $[0, 1]$ ergibt, folgt nun eine Beschreibung der soeben erläuterten Vorgehensweise. Zur Veranschaulichung der Rekursionsschritte soll Abbildung 2.4 dienen.
Die Dimensionsliste der zweiten Dimension im Teilbaum `C1` $= 0$ ist in sortierter Form wie auch zuvor $[0, 1]$ (Schritt II). Der Zeiger von `C1` $= 0$ zeigt auf den Beginn dieser Liste, welche es nun zu durchlaufen gilt. Wie in Teilbild III erkennbar, ergibt sich für `C3` bei `C1` $= 0 \wedge$ `C2` $= 0$ die Liste [1,2]. Für den ersten Eintrag dieser Liste

[1] Algorithmisch nimmt man die Sortierung sukzessive jeweils beim Bearbeiten einer Dimensionsliste vor.

folgt für die letzte Dimension `C4` eine 1 und für den zweiten eine 2. Da es sich um die unterste Ebene handelt, werden diesen beiden die `TID`s T4 respektive T2 angehängt (Ausschnitt IV). Nun müssen die Rekursionen für den zweiten Wert der Dimensionsliste von `C2` im Teilbaum `C1` = 0 betrachtet werden. Für die dritte Dimension ergibt sich hier die Dimensionsliste [1], deren Zeiger auf den korrespondierenden Wert 1 der letzten Dimension und die damit einhergehende `TID` T6 verweist (Schritt V). Damit ist auch in diesem letzten rechten Teilbaum das Abbruchkriterium der Rekursion erreicht, sodass der Baum unter `C1` = 0 finalisiert ist.
Führt man das gleiche Vorgehen für den zweiten Wert (2) der ersten Dimension durch erhält man den letztendlichen Elf wie er im finalen Bild VI dargestellt ist.

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

Tabelle 2.6: Datengrundlage Elf (unsortiert)

→

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

Tabelle 2.7: Datengrundlage Elf (sortiert)

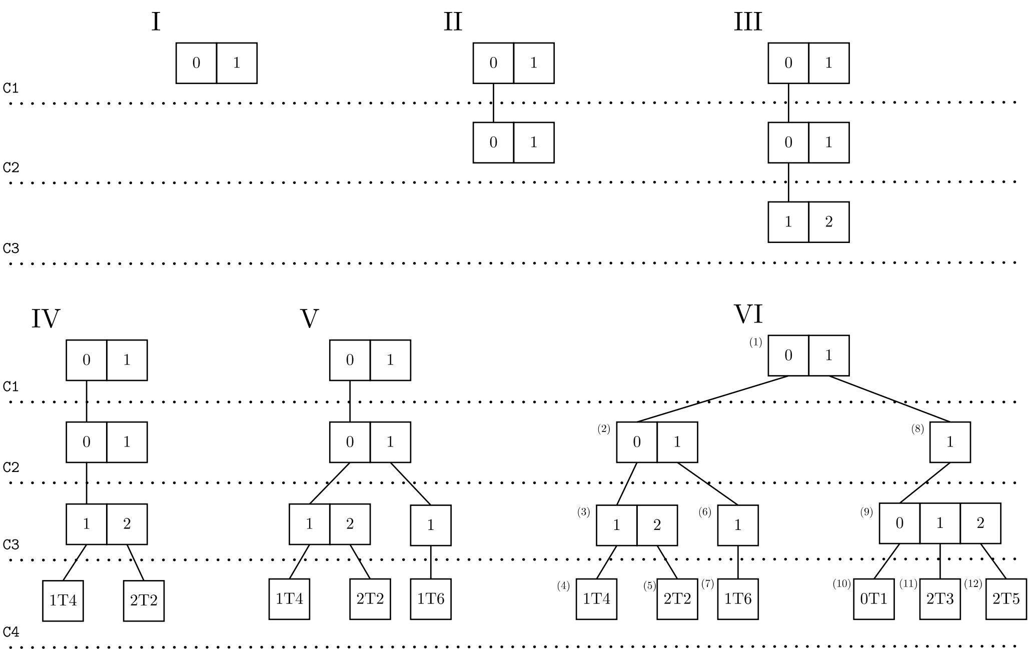


Abbildung 2.4: Erstellen des Elf anhand eines Beispiels (Tabelle 2.7)

Algorithmisches Vorgehen

Algorithmisch kann man das Erstellen des Elf über eine Hilfsstruktur, den sogenannten `InsertElf` umsetzen. Diese als Array umgesetzte Struktur ist darauf ausgelegt, dass das Einfügen, Aktualisieren und Löschen von Daten, also typische OLTP-Operationen, auf ihr selbst durchgeführt werden kann. Zwar besitzt der `InsertElf` ebenso die Eigenschaft der Präfixredundanzeliminierung, allerdings kann nicht garantiert werden, dass die Elemente der Dimensionslisten tatsächlich im Speicher nah beieinander liegen, wodurch eine Eignung für Anfragen aus OLAP-Szenarien eher bedingt gegeben ist [BKSS17]. Vom `InsertElf` ausgehend kann die Erstellung eines (sogleich näher thematisierten) *linearisierten* Elf erfolgen, welcher auf Anfragen spezialisiert ist. Dieses Vorgehen entspricht der ursprünglichen Umsetzung des Algorithmus zum Bauen des Elf. Der konkrete Algorithmus unter Verwendung dieser Methode ist in [BKSS17] zu finden. Indes wurde der `InsertElf` in einer späteren, überarbeiteten Umsetzung, die für diese Arbeit genutzt wird, durch einen rein sortierbasierten Ansatz ersetzt. Dieser kann [Bro19] entnommen werden.

2.2.2.2 Linearisierung: Überführen des Baums in ein Array

Zur Optimierung des Speicherlayouts des Elf wird eine Linearisierung vorgenommen [BKSS17]. Durch sie wird der Elf in einem eindimensionales Array gespeichert. Es wird davon ausgegangen, dass die zu speichernden Werte Zahlen sind. Das heißt, dass Daten oder Zeichenketten vorher beispielsweise über eine Wörterbuchkompression in eine numerische Repräsentation zur Speicherung überführt worden sind.
In der zugrundeliegenden Implementierung werden im Array drei Typen von Einträgen gespeichert: Dimensionswerte und `TID`s als 32-Bit-Integer und 64-Bit-Integer zur Umsetzung der Zeiger innerhalb des Elf. Der Einfachheit halber sollen jedoch für die nachfolgenden Erklärungen und Beispiele einheitlich 64-Bit-Werte Verwendung finden.
Im „Standardelf" folgt auf einen Dimensionswert immer sein dazugehöriger Zeiger zur nächsten Dimension. Die Ausnahme bildet hier lediglich die letzte Dimension, in der auf den Wert anstelle des Zeigers die Liste der zugeordneten `TID`s folgt.
Der zuvor generierte Elf aus Abbildung 2.4 VI ist in seiner finalen, linearisierten Form in Abbildung 2.5 zu finden. Eine Verbindung zwischen Abbildung und Tabelle soll durch die in runden Klammern stehenden Zahlen hergestellt werden.

Zeiger im Array

Die Zeiger werden in Abbildung 2.5 in eckigen Klammern dargestellt und sind als Arrayindexe zu verstehen. Der Wert [04] nach dem ersten Eintrag (0) der ersten Dimensionsliste bedeutet, dass, um zum Beginn des Kindes des Eintrages 0 zu gelangen, ein Sprung zur Position 4 des Arrays erforderlich ist.
Alternativ ließen sich die Zeiger auch als Offsets ausgehend von der aktuellen Position umsetzen. In diesem Fall würde beispielsweise der Zeiger [20] auf Position 3, als [17] gespeichert werden. Inhaltlich heißt das, dass man von der Arrayposition 3 zum 17. darauffolgenden Wert springen muss.
Die Nutzung von Offsets als implizite Form der Zeiger hat gegenüber der expliziten der Arrayindexe den Vorteil, dass in jedem Fall kleinere Werte gespeichert

	0	1	2	3	4	5	6	7	8	9
Elf[00]	(2)0	[04]	-1	[20]	(2)0	[08]	-1	[16]	(3)1	[12]
Elf[10]	-2	[14]	(4)1	-T4	(5)2	-T2	(6)1	[18]	(7)1	-T6
Elf[20]	(8)-1	[22]	(9)0	[28]	1	[30]	-2	[32]	(10)0	-T1
Elf[30]	(11)2	-T3	(12)2	-T5						

Abbildung 2.5: Linearisierung für finalen Elf aus Abbildung 2.4

werden. Das lässt sich wie folgt zeigen: da sich das $Offset$ mithilfe der aktuelle Position ($AktuellePosition$) und der $Zielposition$ als $Offset = Zielposition - AktuellePosition$ berechnen lässt, ist das $Offset$ mathematisch für alle $AktuellePosition$en echt kleiner als die $Zielposition$ – mit Ausnahme der Stelle 0. Nichtsdestotrotz wird bei der soeben gezeigten Implementierung der ersten Dimensionsliste an Position 0 nie ein Zeiger stehen, weshalb die Feststellung ohne Einschränkung für den gesamten Elf gilt.
Tatsächlich kann es zu größeren Differenzen zwischen der Höhe des $Offset$s und der einhergehenden $Zielposition$ kommen. Und zwar immer dann, wenn die Kinder der aktuellen Dimensionsliste relativ kleine Teilbäume sind. Für den Dimensionslistenwert an Position 20 ist zum Beispiel der direkte Index seines Kindes die Position 22. Als $Offset$ würde man hier lediglich [1] an Position 21 speichern. Diese Beobachtung kann relevant werden, wenn man ausschließlich 32-Bit-Integer im Elf nutzen möchte. Besitzt der Elf nämlich mehr als $2^{31} - 1$ Einträge, entstehen konsequenterweise direkte Zeiger, die die 32-Bit-Grenze überschreiten, wodurch man auf 64-Bit-Integer-Werte angewiesen ist. Nutzt man hingegen indirekte Zeiger (Offsets) und kein *Teilbaum* im Elf hat mehr als $2^{31} - 1$ Werte und Zeiger, ließe sich der Elf ausschließlich durch 32-Bit-Integer repräsentieren.

Speicherung der Länge von Dimensionslisten

Für die späteren Scan-Algorithmen ist es essentiell, das Ende eine Dimensionsliste und damit die Länge einer Dimensionsliste feststellen zu können. Daher ist eine Speicherung dieser Information bei der Erstellung des Elf unerlässlich. In der Ursprungsimplementierung wird die Dimensionslistenlänge implizit über das Setzen des *most significant bit*s (kurz MSB) der Zahl am Ende der Liste gespeichert. Visuell ist dies über ein „-“ in Abbildung 2.5 sichtbar gemacht worden. Jene Markierung wird von den angesprochenen Suchalgorithmen wie folgt genutzt: wird ein „negativer“ Wert erreicht, entspricht dieser dem letzten Eintrag und spätestens dann kann die Suche abgebrochen werden. Da `TID`-Zuordnungen prinzipiell nicht als bijektiv (vgl. Definition Primär- und Sekundärindex aus Abschnitt 2.1) angenommen werden, werden sie als Liste aufgefasst, um eine Speicherung von Tupeln mit unterschiedlichen `TID`s, aber gleichen Attributwerten zu ermöglichen. Daher ist auch für sie die MSB-Markierung am jeweils letzten Eintrag einer solchen `TID`-Liste notwendig.

2.2.2.3 Abweichende Speicherung der ersten Dimension

Die erste Dimension des Elf nimmt bei dessen Speicherung eine Sonderstellung ein. Aus der im Vorhergehenden definierten Struktur des linearisierten Elfen resultiert, dass in dieser Dimension, der Wurzel des Baums, alle eindeutigen Werte der ersten Spalte zu finden sind. Um später bei Intervallanfragen einen Bereich für das erste Attribut verarbeiten zu können, ist somit ein (eher langsamer) sequentieller Scan der gesamten ersten Dimensionsliste notwendig [BKSS17].
Bei einer Erweiterung des Elf, die diese Problematik aufgreift, werden statt Werten und Zeigern in der ersten Dimensionsliste ausschließlich Zeiger gespeichert. Dabei wird der ganzzahlige Wert selbst zum Index des Arrays und an die durch ihn verwiesene Position wird sein zugehöriger Zeiger zur zweiten Dimension geschrieben.
Formal ausgedrückt wird mit den Dimensionswerten V_i und ihren Zeigern P_i aus $DIMLIST_1 = [V_1, P_1, V_2, P_2, \ldots, V_n, P_n]$ mit $V_i \in DIM_1$ nun $DIMLIST_1' = [P_1, \ldots, P_2, \ldots, P_n]$ mit $Position(P_i) = V_i$.
Wendet man diese Anpassung auf das vorangegangene Beispiel aus Abbildung 2.5 an, muss dadurch nicht nur der Arraybereich, der die erste Dimensionsliste widerspiegelt, angepasst werden, sondern es verschieben sich auch alle anderen Zeiger im Array um zwei Positionen nach vorn. Grund hierfür ist, dass die zwei Dimensionswerte (0,1) des ersten Attributs nun eben nicht mehr im Array wiederzufinden sind. Das Endergebnis dieser Optimierung kann in Abbildung 2.6 betrachtet werden.

	0	1	2	3	4	5	6	7	8	9
Elf[00]	[02]	[18]	(2) 0	[06]	-1	[14]	(3) 1	-[10]	-2	-[12]
Elf[10]	(4) 1	T4	(5) 2	T2	(6) -1	[16]	(7) 1	T6	(8) -1	[20]
Elf[20]	(9) 0	[26]	1	[28]	-2	[30]	(10) 0	T1	(11) 2	T3
Elf[30]	(12) 2	T5								

Abbildung 2.6: Optimierung der Speicherung der ersten Dimension des linearisierten Elf aus Abbildung 2.5

Dünnbesetztheit der ersten Dimension

Sind zwischen den Werten der ersten Dimension anders als im Beispiel größere Abstände als 1 (formal $\exists i \mid V_{i+1} - V_i > 1$), so kommt mit der erläuterten Anpassung der Nachteil, dass der Arraybereich der ersten Dimension Lücken aufweist. Diese müssen durch spezielle (undefinierte) Zeiger repräsentiert werden.
Wären die eindeutigen Werte der ersten Spalte zum Beispiel $(0, 3, 4)$ müsste man einen Bereich von fünf Einträgen für die erste Dimension reservieren. Dieser sähe wie folgt aus: $DIMLIST_1 = [P_1, \perp, \perp, P_2, P_3]$. Diese Dünnbesetztheit ließe sich vermeiden, wenn man eine Transformation findet, die jede einzelne Ausprägung der ersten Dimension, auf $0, 1, ..., |DIM_1|$ abbildet. Die einfachste und naheliegendste Vorgehensweise wäre wohl das Nummerieren aller eindeutigen Werte der ersten Spalte und das Speichern dieser in einer zusätzlichen Lookup-Table.

2.2.2.4 Eliminierung einelementiger Dimensionslisten

Eine weitere Herausforderung des Elf – egal ob als Baum oder Array betrachtet – sind kürzer werdende Dimensionslisten je höher die Dimension ist. Im Extremfall sind die Listen sogar nur einelementig. Das bedeutet, dass es keine Präfixredundanzen mehr gibt, die man eliminieren konnte.
Indexiert man alle fünfzehn Spalten der `Lineitem`-Tabelle des TPC-H-Benchmarks beträgt der Anteil der einelementigen Listen ab der 11. Dimension mehr als 99%. Ab der 13. Spalte sind alle Dimensionslisten ausschließlich einelementig. In Abbildung 2.7 ist jener Anteil in Abhängikeit von den indexierten Spalten dargestellt.

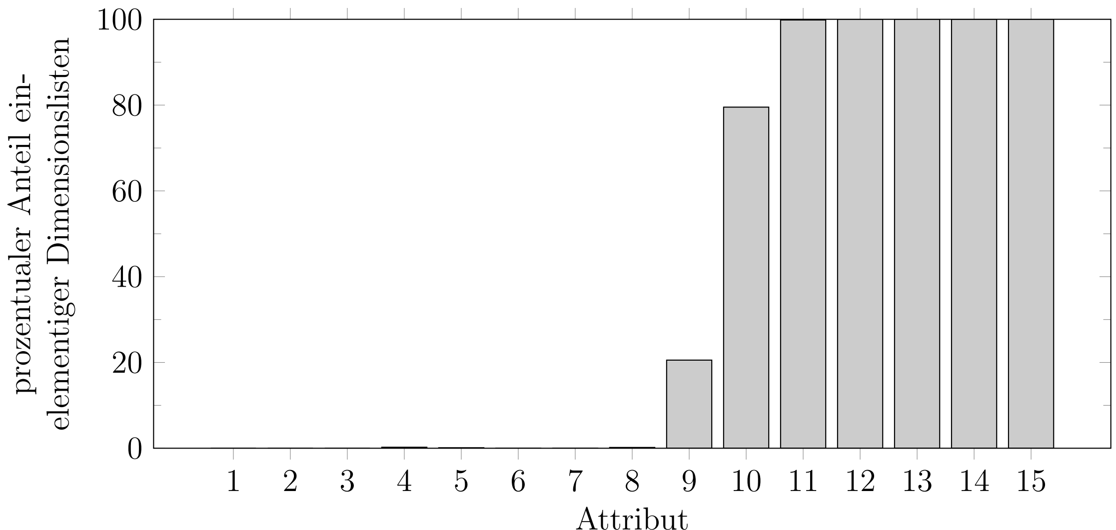


Abbildung 2.7: Prozentualer Anteil einelementiger Dimensionslisten je Attribut innerhalb des Elf für die `Lineitem`-Tabelle ([BKSS17]).

Die sogenannten `Monolisten` sollen diese Problematik effizienter als bislang vorgestellt lösen. Folgen auf einen Dimensionslisteneintrag d_i der Dimension i in den darunterliegenden Dimensionen $i+1, i+2, \ldots, n$ (n ist die Anzahl an Dimensionen) ausschließlich einelementige Dimensionslisten (mit Ausnahme der `TID`-Listen am Ende), werden diese beginnend mit dem Eintrag der Dimensionsliste der Spalte $i+1$ in einem Knoten, genannt `Monolist`, zusammengefasst. Dadurch erspart man sich $n-(i+1)$ Zeiger im Array. Im Beispiel betrifft das lediglich den Präfix „01“ – dieser spezifiziert bei den vorliegenden Daten das Tupel mit der `TID` T6 bereits eindeutig, sodass der anschließende Suffix „11“ der Dimensionen drei und vier in eine `Monolist` zusammengefasst werden kann. Die Überführung der Baumstruktur in das die `Monolist` implementierende Format ist Abbildung 2.8 zu entnehmen.

Für die linearisierte Speicherung ändert sich zum einen, dass sich bei solchen Konstellationen nicht mehr Wert und Zeiger innerhalb der `Monolist` abwechseln. Vielmehr reihen sich die Werte der nacheinanderfolgenden Dimensionen ohne Unterbrechung bis zur `TID`-Liste aneinander. Zum anderen werden Zeiger, die auf `Monolist` verweisen, wie auch Listenenden mithilfe des Markierens ihres MSBs entsprechend gekennzeichnet. Beides ist in Abbildung 2.9 wiederzufinden.

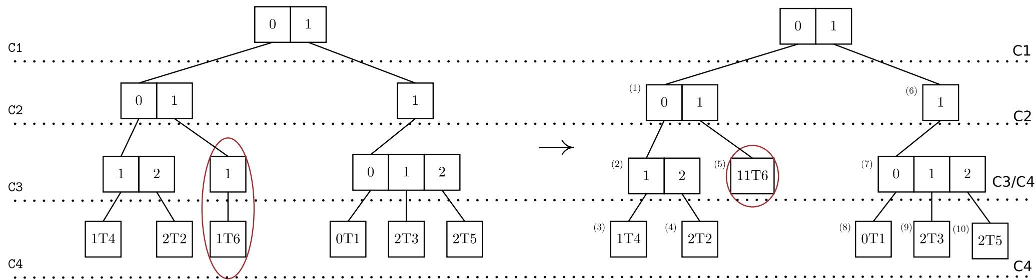


Abbildung 2.8: Finaler Elf aus Abbildung 2.4 (VI) mit `Monolist`.

	0	1	2	3	4	5	6	7	8	9
Elf[00]	[02]	-[17]	$^{(1)}$0	[06]	-1	-[14]	$^{(2)}$1	-[10]	-2	-[12]
Elf[10]	$^{(3)}$1	-T4	$^{(4)}$2	-T2	$^{(5)}$1	1	-T6	$^{(6)}$-1	[19]	$^{(7)}$0
Elf[20]	[25]	1	[27]	-2	[29]	$^{(8)}$0	-T1	$^{(9)}$2	-T3	$^{(10)}$2
Elf[30]	-T5									

Abbildung 2.9: Linearisierter Elf mit optimierter ersten Dimension und `Monolisten` aus Abbildung 2.8

2.2.3 Anfragen mithilfe des Elf

Im vorangegangenen Abschnitt wurde erläutert, wie der Elf prinzipiell programmatisch erstellt werden kann und welche Optimierungen bezüglich seiner Struktur ausgehend vom ursprünglichen, baumbasierten Elf vorgenommen werden können. Nun wird geschildert, wie man ihn als Indexstruktur tatsächlich nutzen kann.

2.2.3.1 Standardalgorithmus für partielle Bereichsanfragen

Wie bereits zu Beginn des Grundlagenkapitels erwähnt, ist das Ziel und damit der Nutzen eines Index das Beschleunigen von Anfragen mit Selektionsbedingungen. Es soll also jezuz darum gehen, wie Anfragen algorithmisch im Elf verarbeitet werden können.
Es werden dabei lediglich partielle Bereichsanfragen betrachtet, da diese, wie in Abschnitt 2.1.1 gezeigt, alle anderen abbilden können. Derartige Anfragen besitzen bekanntlich für $k \leq n$ Dimensionen (n ist die Anzahl an Dimensionen) Intervallbedingungen der Form $[lower, upper]$. Ergebnis der Durchführung einer Anfrage dieses Typs unter Berücksichtigung der vorgegebenen Einschränkungen soll eine Menge von `TIDs` sein. Das heißt, die Attribute der durch die `TIDs` verwiesenen Tupel liegen innerhalb der von der Bereichsanfrage definierten Grenzen. Ist für eine Spalte kein Bereich spezifiziert, sind die Ausprägung des Tupel für diese irrelvant. Eine solche Spalte ohne angegebene Intervalle sollen in den weiteren Ausführungen `Platzhalterspalte` genannt werden.

Konzeptionelles Vorgehen

Im Allgemeinen lassen sich für die Verarbeitung von Anfragen auf dem Elf zwei Hauptschritte herausarbeiten: das Evaluieren von Dimensionslisten und das von `Monolisten`.
Der Elf wird – entsprechend seines Aufbaus – zur Anfragebearbeitung in Form einer Pre-Order-Tiefensuche durchlaufen. Handelt es sich bei einer Dimension um eine `Platzhalterspalte`, wird jedem Zeiger der jeweiligen Dimensionsliste gefolgt. Liegt dahingegen ein Intervall für das Attribut vor, wird dieses ausgewertet. Das bedeutet, es wird ausschließlich den Zeigern der Werte, die in diesen Bereich fallen, in die nächste Dimension gefolgt. In diesem Fall kann die Sortiertheit der Dimensionslisten zum frühen (klugen) Pruning (engl. *early pruning*) ausgenutzt werden. Von einer aufsteigenden Sortierung ausgehend, kann der Scan einer Dimensionsliste nämlich immer dann beendet werden, wenn ein Wert gefunden wird, welcher größer als die definierte obere Grenze des entsprechenden Attributs ist.
Während die Dimensionslisten einen *intradimensionalen* Charakter besitzen, also immer nur Werte einer Spalte enthalten, sind `Monolisten` *interdimensional*. Aus diesem Grund lässt sich auf letztere nicht dasselbe *early pruning* anwenden. Bei `Monolisten` lässt sich zwar keine Sortierung ausnutzen, wohl aber die Tatsache, dass sich ein Tupel immer dann für die Ergebnismenge *dis*qualifiziert, wenn mindestens eines seiner Attribute außerhalb der für es spezifizierten Grenzen liegt. Wird dies für einen Wert der `Monolist` festgestellt, kann der sequentielle Scan auf ihr umgehend abgebrochen werden. In die Ergebnismenge werden folglich alle Tupel der `Monolisten` genommen, bei denen kein Abbruch erfolgte.

Programmatische Aufteilung in drei Algorithmen

Aufgrund der abweichenden Speicherung der ersten Dimension ergeben sich bei der Implementierung primär drei Funktionen zum Verarbeiten von partiellen Bereichsanfragen: `ScanFirstDimlist` (Algorithmus 1), `ScanDimlist` (Algorithmus 2) und `ScanMonolist` (Algorithmus 3).
Der `Elf` liegt in seiner Arrayform vollständig erstellt jedem dieser Algorithmen zum Zugriff auf Werte und Zeiger vor.
Alle Algorithmen haben die Parameter $lower[]$ und $upper[]$ gemein. Diese Arrays haben so viele Einträge wie es Dimensionen gibt und beinhalten an ihrer i-ten Position die unteren ($lower[]$) und oberen ($upper[]$) jeweils abgeschlossenen Intervallgrenzen des i-ten Attributs der Bereichsanfragen. Für den Bereich $lower[1] = 0,\ upper[1] = 4$ qualifizieren sich demnach alle Tupel, deren Werte d der zweiten Dimension die Bedingung $0 \leq d \leq 4$ erfüllen. Ob auf einem Attribut eine Selektionsbedingung vorliegt oder ob es sich um eine `Platzhalterspalte` handelt, wird über den Paramter $selDims[]$ als Bitvektor gespeichert. Ist Dimension i mit einer Intervalldefinition versehen, ist $selDims[i] = 1$, sonst wird an selber Stelle 0 gespeichert.

Eine weitere Vereinfachung betrifft ausschließlich die erste Dimension und damit Algorithmus 1. Wie in Abschnitt 2.2.2.3 bereits erwähnt, können bei der vorgeschlagenen Speicherung, die direkt über die Ausprägungen des ersten Attributs indexiert, Lücken entstehen. Das soll für die Beschreibung des Algorithmus außer Acht gelas-

sen werden. Das heißt, es gibt durchgängig ganzzahlige Werte von 0 bis $max\{D_0\}$, dem Maximum der ersten Dimension[1].

```
Input : Partielle Bereichsanfrage auf Tabelle mit n > 0 Dimensionen,
        Bereichsdefinitionen durch lower[n] und upper[n],
        selektierte Dimensionen als Bitvektor selDims[n]
Output : Ergebnismenge L mit TIDs

Function ScanFirstDimlist(lower[], upper[], selDims[]):
1     L ← ∅;
2     if selDims[0] then
          // Selektion auf 1. Dimension definiert
3         begin ← min{lower[0], 0}; end ← max{upper[0], max{D_0}};
4     else
          // Keine Selektion auf 1. Dimension, alle Werte betrachten
5         begin ← 0; end ← max{D_0};
6     end
7
8     for index ← begin to end do
9         pointer ← Elf[index];
10        if isMonolistPointer(pointer) then
11            ScanMonolist(lower, upper, selDims, unsetMSB(pointer), 1, L);
12        else
13            ScanDimlist(lower, upper, selDims, pointer, 1, L);
14        end
15    end
16    return L;
17 end
```

Algorithmus 1 : Verarbeitung der ersten Dimension des Elf als Einstiegspunkt für partielle Bereichsanfragen (adaptiert von Broneske [Bro19])

Scan der ersten Dimension. Betrachten wir nun konkret `ScanFirstDim` aus Algorithmus 1: Ergebnis, und damit Rückgabewert der Funktion, soll eine Menge L mit den qualifizierten `TID`s sein. L wird zunächst leer initialisiert (Zeile 1). Anschließend wird festgestellt, ob es sich beim ersten Attribut direkt um eine `Platzhalter-spalte` handelt (Zeile 2). Wenn dem so ist, lassen sich keine Beschränkungen der Werte vornehmen, sodass alle Werte von 0 (*begin*) bis bis $max\{D_0\}$ (*end*) betrachtet werden müssen (Zeile 5). Andernfalls können *begin* und *end* auf die jeweiligen Bereichsgrenzen gesetzt werden, wobei diese nicht größer als das Dimensionsminimum respektive -maximum sein dürfen (Zeile 3). Das abgesteckte Intervall von *begin* bis *end* gilt es nun in jedem Fall vollständig zu durchlaufen. Durch die spezielle Speicherung der ersten Dimension kann an die jeweiligen Zeiger zum Beginn der nächsten

[1] Die Minima und Maxima jeder Dimension werden beim Erstellen des Elf gesondert gespeichert beziehungsweise können sie im Allgemeinen dem Data Dictionary einer Datenbank entnommen werden.

Dimensionslisten über einen direkten Zugriff mithilfe des Attributwerts gelangt werden. Wären Lücken zwischen den Dimensionswerten, müsste man in jeder Iteration zunächst prüfen, ob es sich bei der aktuellen Position um eine zuvor festgelegte, sonst nicht vorkommende Ausprägung handelt. In diesem Fall ist die Position zu überspringen und mit dem nächsten Schleifendurchlauf fortzufahren. Ansonsten ist innerhalb der Schleife zu prüfen, ob der aktuelle Zeiger auf eine Monoliste weist (via `isMonolistPointer`), was gleichbedeutend mit dem Gesetztsein seines *most significant bit*s ist (Zeile `10`). Wenn dem so ist, muss das MSB über `unsetMSB` unmarkiert werden, bevor es an die entsprechende Funktion `ScanMonolist` weitergegeben werden kann (Zeile `11`). Das Testen, Setzen und Rückgängigmachen des MSBs wird in der verwendeten Implementierung über entsprechende Bitmasken realisiert. Für „normale“ Dimensionslisten muss dieses Vorgehen nicht angewandt werden; *pointer* kann direkt an `ScanDimlist` weitergereicht werden (Zeile `13`).

Scan regulärer Dimensionslisten. Für `ScanDimlist` (Algorithmus 2) dient der übergebene *pointer* als Startpunkt der jeweiligen Dimensionsliste im linearisierten Elf (Zeile `1`). Die Lösungsmenge L wird von diesem Algorithmus selbst nicht erweitert – sie wird lediglich zur Weitergabe an `ScanMonolist` benötigt. Der *dim*-Paramter beinhaltet die zu betrachtende Dimension und dient der Bestimmung, ob in dieser eine Selektion vorliegt, oder ob es sich um eine `Platzhalterspalte` handelt. Wenn eine Bereichsdefinition gegeben ist, muss die Dimensionsliste nach all den Werten gescannt werden, die innerhalb des Intervalls liegen (Zeile `3`). Binnen der dafür vorgesehenen Schleife (Zeilen `4` bis `19`) wird dies über `isIn` getestet. Diese Funktion prüft für ihre Paramter v, l und u, ob $l \leq v \leq u$ gilt und gibt einen entsprechenden `bool` zurück (Zeile `6`). Liegt der Wert im definierten Fenster, muss für diesen ein rekursiver Aufruf erfolgen. Das heißt, durch seinen Zeiger, der aufgrund der alternierenden Speicherung innerhalb der Dimensionsliste (Zeiger folgt stets auf Wert) genau ein Feld weiter im Elf liegt, wird der „Abstieg“ in die nächste Dimension initiiert. Dabei wird wieder zwischen einem Zeiger auf eine `Monolist` und auf eine normale Dimensionsliste differenziert und die jeweilige Funktion entsprechend gewählt (Zeilen `9` und `11`). Gibt `isIn` jedoch `false` zurück, kann die Schleife genau dann abgebrochen werden, wenn der Wert an der aktuellen Position bereits größer als die obere Grenze des momentan betrachteten Attributs ist (Zeile `14`). Kann diese Form von *Pruning* nicht angewandt werden, gilt es, die Schleife weiter zu durchlaufen. Dazu wird der Index um zwei inkrementiert, um zum nächsten Dimensionslistenwert zu gelangen (Zeile `18`). Dies wird sooft wiederholt bis der letzte Dimensionslisteneintrag erreicht worden ist. Die Feststellung dessen erfolgt erneut anhand des MSBs, welches für Werte mithilfe von `isEndOfList` überprüft wird (Zeile `5`). Handelt es sich beim aktuellen Aufruf von `ScanDimlist` bei der Dimension um eine `Platzhalterspalte`, kann keine `isIn`-Prüfung erfolgen, vielmehr muss allen Zeigern der Dimensionsliste gefolgt werden (Zeilen `20` bis `31`).

Scan der `Monolisten`. In der dritten und damit letzten Basisfunktion zum Durchsuchen des Elf, `ScanMonolist` (Algorithmus 3), findet die Erstellung und Erweiterung des Ergebnisses der Anfrage, der Menge von `TIDs` (L), statt. Die `TIDs` stellen schließlich das Ende von `Monolisten` dar und der Zugriff auf sie ist, in der Standar-

Input : Bereichsdefinitionen durch $lower[n]$ und $upper[n]$,
selektierte Dimensionen als Bitvektor $selDims[n]$,
Beginn der zu durchsuchenden Dimensionsliste $beginPointer$,
aktuelle Dimension dim,
Ergebnismenge L mit `TIDs`

```
Procedure ScanDimlist(lower[], upper[], selDims[], beginPointer, dim, L):
1      index ← beginPointer ;
2      isLastEntry ← false;
3      if selDims[dim] then
           // Selektion auf aktueller Dimension definiert
4          do
5              isLastEntry ←isEndOfList(Elf[index]);
6              if isIn(Elf[index],lower[dim],upper[dim]) then
7                  pointer ← Elf[index + 1];
8                  if isMonolistPointer(pointer) then
9                      ScanMonolist(lower, upper, selDims,unsetMSB(pointer),
                                   dim + 1, L);
10                 else
11                     ScanDimlist(lower, upper, selDims, pointer, dim + 1, L);
12                 end
13             else
14                 if Elf[position] > upper[dim] then
15                     return;
16                 end
17             end
18             index ← index + 2;
19         while not isLastEntry;
20     else
           // keine Selektion auf akt. Dimension, alle Werte betrachten
21         do
22             pointer ← Elf[index + 1];
23             if isMonolistPointer(pointer) then
24                 ScanMonolist(lower, upper, selDims,unsetMSB(pointer), dim+1, L);
25             else
26                 ScanDimlist(lower, upper, selDims, pointer, dim + 1, L);
27             end
28             isLastEntry ←isEndOfList(Elf[index]);
29             index ← index + 2;
30         while not isLastEntry;
31     end
32 end
```

Algorithmus 2 : Rekursiver Scan von Dimensionslisten (adaptiert von Broneske [Bro19])

dimplementierung, erst an dieser Stelle möglich. Bevor allerdings das Endresultat erweitert werden kann, muss zunächst überprüft werden, ob sich auch alle Werte der `Monolist` qualifizieren. Dazu werden deren Werte sequentiell durchlaufen und gegen die (eventuell) definierten Intervalle von *lower*[] und *upper*[] getestet (Zeile 2). Sobald für einen der enthaltenen Werte `isIn` nicht `true` liefert, kann die Funktion abgebrochen werden (Zeilen 3 und 4). Kommt es zu diesem *Pruning*, erfolgt konsequenterweise keine Aufnahme der verbundenen `TID`s in die Ergebnismenge L. Sind dahingegen die Bedingungen für alle Dimensionen erfüllt, muss L um alle dem letzten Dimensionswert der `Monolist` folgenden `TID`s erweitert werden (Zeilen 8 bis 13).

Input : Bereichsdefinitionen durch $lower[n]$ und $upper[n]$,
selektierte Dimensionen als Bitvektor $selDims[n]$,
Beginn der zu durchsuchenden Dimensionsliste $beginPointer$,
aktuelle Dimension dim,
Ergebnismenge L mit `TID`s

```
Procedure ScanMonolist(lower[], upper[], selDims[], beginPointer, dim, L):
1    for dimIndex ← dim to |DIM| do
2        if selDims[dimIndex] then
             // Selektion auf Dimension dimIndex
3            if not isIn(Elf[beginPointer + dimIndex − dim],lower[dimIndex],
                         upper[dimIndex])
              then
                 // Abbruch, wenn Monolist-Wert außerhalb der Grenzen
4                return;
5            end
6        end
7    end
     // qualifizierte TIDs in Ergebnismenge aufnehmen
8    index ← |DIM| − dim;
9    while not isEndOfList(Elf[index]) do
10       L ← L ∪ Elf[index];
11       index ← index + 1;
12   end
13   L ← L ∪ unsetMSB(Elf[index]);
14 end
```

Algorithmus 3 : Scan von `Monolisten` als finaler Schritt zum Anfügen von Ergebnissen (adaptiert von Broneske [Bro19])

Beispielhafter Aufruf. Für eine beispielhafte partielle Bereichsanfrage mit den Intervallen $[[0,0],[0,0],[*],[*]]$ ($[*]$ sind `Platzhalterspalten`) des Elf aus Abbildung 2.9, können die Abfolgen und Prüfungen der drei vorgestellten Algorithmen Tabelle 2.8 entnommen werden. Aus Platzgründen wurden für die Funktionen Abkürzungen genutzt: `SFDL` entspricht `ScanFirstDimlist`, `SDL` `ScanDimlist` und `SML`

`ScanMonolist`. Es wurde sich auf die relevanten, sich ändernden Parameter der Funktionen beschränkt und deren Bezeichnungen, wenn erforderlich ebenso gekürzt ($bP \hat{=} beginPointer$, $d \hat{=} dim$). Die Funktion `isDimSelected` entspricht der Prüfung $selDim[dim] = 1$.
Dieser Ablauf inklusive seiner Prüfungen kann auch noch einmal grafisch in Abbildung 2.10 nachvollzogen werden – eine Verbindung zur Tabelle 2.8 kann hier über die in Kreise und Sterne gefassten Zahlen und Buchstaben hergestellt werden.
Anhand des Beispiels lässt sich Optimierungspotenzial bezüglich des `TID`-Zugriffs ableiten. Durch die partielle Bereichsanfrage sind lediglich die ersten beiden Spalten mit einem Intervall versehen worden. Die letzten beiden Dimensionen stellen `Platzhalterspalten` dar. Für die Scan-Algorithmen bedeutet das, dass die inter- und intradimensionalen Pruningmethoden von `ScanDimlist` und `ScanMonolist` für die Attribute drei und vier keine Verwendung finden können. In `ScanDimlist` werden schlichtweg alle möglichen rekursiven Aufrufe stumpf abgearbeitet.

Aufrufende Funktion (Caller)	Prüfung	aufgerufene Funktion (Callee)
`SFDL`	(a) `Elf`[0]	① `SDL`(bP: 2, d: 1, L: {})
① `SDL`(bP: 2, d: 1, L: {})	(b) `isIn`(0,0,0)	② `SDL`(bP: 6, d: 2, L: {})
② `SDL`(bP: 6, d: 2, L: {})	(c) `isDimSelected`	③ `SML`(bP: 10, d: 3, L: {})
③ `SML`(bP: 10, d: 3, L: {})	(d) `isDimSelected`	-
② `SDL`(bP: 6, d: 2, L: {T4})	-	④ `SML`(bP: 12, d: 3, L: {T4})
④ `SML`(bP: 12, d: 3, L: {T4})	(e) `isDimSelected`	-
① `SDL`(bP: 2, d: 1, L: {T4,T2})	(f) `isIn`(1,0,0)	-
`SFDL`	-	**return** L = {T4,T2}

Tabelle 2.8: Aufrufshierarchie und Prüfungen der Scan-Algorithmen des Elf aus Abbildung 2.9 bei der partiellen Bereichsanfrage mit den Intervallen $[[0, 0], [0, 0], [*], [*]]$

Erst und ausschließlich in `ScanMonolist` erfolgt der Zugriff auf die `TID`s. Eine Lösung für genau dieses Problem soll im folgenden Abschnitt vorgestellt werden: die sogenannten `Cutoffs`.

2.2.3.2 `Cutoffs`: zusätzliche Datenstruktur zum frühen, direkten `TID`-Zugriff

Wie die Übersetzung von `Cutoffs`, Abkürzungen, und der Kontext der `TID`-Zugriffe vermuten lässt, soll mit dieser Erweiterung des linearisierten Elf schneller an die `TID`s und damit zum gewünschten Anfrageergebnis gelangt werden.

Grundidee, Nutzen und Grenzen

Die Grundlage dafür wird beim Erstellen des Elf gelegt. Dort wird zusätzlich zu den Werten und Zeigern innerhalb von Dimensionslisten eine weitere Zeigart, genannt

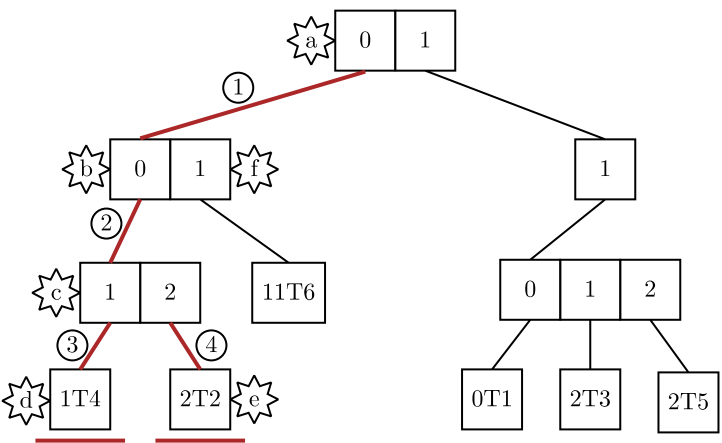


Abbildung 2.10: Visuelle Darstellung der Aufrufshierarchie und Prüfungen aus Tabelle 2.8

`Cutoff`-Zeiger, gespeichert. Anders als die bisherigen Zeiger verweisen diese jedoch nicht auf eine Position innerhalb des Elf-Arrays, sondern auf eine neue Datenstruktur namens `Elf_TIDs`. In diesem zweiten, parallel aufgebauten Array finden sich die `TIDs` in jener Reihenfolge, die durch die Sortierung der zu indexierenden Relation im Laufe der Erstellung des Elf entsteht. Diese entspricht in der Regel nicht der „natürlichen“ Ordnung innerhalb einer Relation wie T1, T2, ..., T6 aus Tabelle 2.6. Im gewählten Beispiel wäre `Elf_TIDs` ein Array mit den Einträgen [T4,T2,T6,T1,T3,T5]. Diese Abfolge kann aus der ersten Spalte der sortierten Tabelle 2.7 oder aus jedem der darauf basierenden Elf-Arrays (zum Beispiel Abbildung 2.6) durch Ausblenden aller Zeiger und Werte abgelesen werden.

Formale Definition von `Cutoffs`. Betrachten wir nun, wie auf das neue Array `Elf_TIDs` mithilfe der `Cutoff`-Zeiger verwiesen wird, indem wir diese formal beschreiben. Ein `Cutoff`-Zeiger c eines Dimensionswertes d, der den letzten Eintrag eines Präfix p bis zu einer Dimension dim darstellt, definiert den Beginn einer `TID`-Abfolge als Index von `Elf_TIDs`. Die Tupel, die zu den `TIDs` jener Abfolge gehören, teilen sich dabei alle denselben Präfix p bis zu dim. Der `Cutoff`-Zeiger c wird beim Bauen des Elf immer zwei Positionen nach d in dessen Dimensionsliste gespeichert. Das Ende der qualifizierten `TID`-Liste wird im Allgemeinen definiert durch den auf d innerhalb von dim folgenden Dimensionswert d' und dessen `Cutoff`-Zeiger c'. Als Anfrageergebnismenge zum Präfix p zählen dann alle `TIDs` aus dem `Elf_TIDs`-Array beginnend mit dem Index c bis ausschließlich c'. Handelt es sich bei d um den letzten Wert seiner Dimensionsliste, wird c' auf den `Cutoff`-Zeiger des Nachfolgereintrags des vorletzten Wertes vom Präfix p, also dem in Spalte $dim-1$ gesetzt. Ist der Wert von p aus $dim-1$ wieder der letzte, wird der aus $dim-2$ betrachtet. Dies kann sich rekursiv bis zur ersten Dimension fortsetzen. In diesem Fall entspricht c' der Anzahl der Zeilen der indexierten Tabelle. Betrachten wir beispielsweise den fiktiven Dimensionslistenauszug $[\ldots, 3, [06], [\mathit{01}], 5, [20], [\mathit{05}], \ldots]$. Hier wäre $d = 3$, $d' = 5$ und ihre `Cutoff`-Pointer $c = [\mathit{01}]$ respektive $c' = [\mathit{05}]$. Wäre die dazugehörige Dimension die

letzte Nicht-`Platzhalterspalte`, gälte es nun, alle Einträge aus `Elf_TIDs` mit den Indexen $1, 2, 3$ und 4 auszulesen und als Ergebnismenge zurückzuspielen.

Limitationen von `Cutoffs`. Die präfixbasierte Definition erklärt den Anwendungsfall und damit auch die Limitationen der `Cutoff`s. Man kann die Abkürzungen einer Dimension zu den `TID`s immer nur dann ausnutzen, wenn es sich bei dieser um die letztselektierte, also die letzte Nicht-`Platzhalterspalte`, handelt.
Anhand der obigen Definition lässt sich dies folgendermaßen erklären: ein `Cutoff`-Zeiger gehört immer zu einem bestimmten Präfix p. Die Ergebnismenge, die durch das Folgen der `Cutoff`-Zeiger entsteht, gilt für *alle* Kombinationen von Werten (Suffixe), die nach p folgen. Gibt es nach der aktuellen Dimension dim, der p durch seine Länge direkt zugeordnet ist, Attribute mit Selektionsbedingungen, entstehen dadurch verfeinerte, konkrete und nicht mehr beliebige Suffixe. Deshalb wird in der Regel die Ergebnismenge verkleinert, sodass die Lösungsmenge, die aus dem `Cutoff`-Zeiger von p entstünde, lediglich eine Obermenge der tatsächlichen darstellen würde. Die Mengen sind nur dann identisch, wenn die Anfrageintervalle der auf dim folgenden Attribute alle möglichen Attributsausprägungen des Teilbaums nach p umfassen.

Notwendige Erweiterungen beim Erstellen des Elf

Da der Fokus dieser Arbeit das Optimieren der Anfragebearbeitung innerhalb des Elf ist, sollen an dieser Stelle nur die Abänderungen der Scan- und nicht der Erstellalgorithmen für die Verwendung der `Cutoff`s detaillierter beschrieben werden.
Grob umrissen, wird beim Erzeugen des Elf ein `TID`-Zähler bei jedem Einfügen einer `TID` im Zuge der `Monolist`-Generierung an der aktuellen Position des Wertes um zwei Stellen versetzt hinterlegt und anschließend inkrementiert. Innerhalb einer `Monolist` werden weiterhin keine Zeiger gespeichert – auch keine zu den `Cutoff`s. Eine Ausnahme bildet erneut die erste Dimension, weil in dieser, wie in Abschnitt 2.2.2.3 vorgestellt, nur Zeiger gespeichert werden. Bei ihr werden die `Cutoff`-Zeiger direkt nach den gewöhnlichen Zeigern hinterlegt, wodurch dieser Arraybereich doppelt so lang wird. Das bedeutet, der `Cutoff`-Zeiger in der ersten Dimension für den Wert d ist an Position $d + 1$ zu finden.
In der Implementierung mit `Cutoff`s gibt es einen Parameter, bis zu welcher Dimension die entsprechenden Zeiger gespeichert werden. Hintergrund ist, dass diese Abkürzungen verständlicherweise vorrangig in den höheren Ebenen des Elf-Baums effektiv sind, weil dort die größten Teilbäume übersprungen werden können. Aufgrund dessen könnte man, um die Komplexität des Erstellens (etwas) zu reduzieren, aber hauptsächlich um den vom `Elf`-Array[1] benötigten Speicherplatz zu reduzieren, auf `Cutoff`s ab einer spezifizierten Dimension verzichten. Die einzige Dimension, an der `Cutoff`s nie einen Vorteil hervorrufen werden ist die letzte. Hier werden die `TID`s sowieso direkt nach den Werten gespeichert; ein Verweis auf die externe Datenstruktur `Elf_TIDs` wäre aufgrund eines abweichenden Speicherzugriffmusters sogar eher kontraproduktiv. Der vereinfachten Darstellung halber innerhalb der Algorithmen, wird von bestehenden `Cutoff`-Zeigern bis zur vorletzten Dimension ausgegangen.

[1] Das `Elf_TIDs`-Array wird in seiner Größe nicht durch den `Cutoff`-Höhenparameter beeinflusst. Sobald `Cutoff`s zum Einsatz kommen sollen, besitzt dieses so viele Einträge wie die Relation.

Nutzung durch Anpassungen in Scan-Algorithmen

Obgleich sich am Algorithmus für das Testen der `Monolists` wie eben beschrieben sich im Vergleich zum Elf ohne `Cutoffs` nichts ändert, müssen Anpassungen an den anderen beiden Scan-Algorithmen vorgenommen werden.

```
Input : Partielle Bereichsanfrage auf Tabelle mit n > 0 Dimensionen
        Bereichsdefinitionen durch lower[n] und upper[n],
        selektierte Dimensionen als Bitvektor selDims[n],
        letzte Dimension, auf der eine Selektion vorliegt (lastSelDim)

   Function ScanFirstDimlistCutoffs(lower[], upper[], selDims[], lastSelDim):
 1   if lastSelDim = 0 then
       // 1. Dimension ist letztselektierte → Cutoffs nutzen
 2     tidPointerBegin ← Elf[lower[0] + 1];
 3     if max{D_0} ∈ {lower[0], upper[0]} then
 4       tidPointerEnd = |R|;
         // |R| ≙ Gesamtanzahl Tupel in Relation
 5     else
 6       tidPointerEnd = Elf[upper[0] + 1];
 7     end
 8     return {Elf_TID[i] | tidPointerBegin ≤ i < tidPointerEnd};
 9   else
10     L ← ∅;
       // begin/end auf lower/upper setzen, wenn 1. Dim. selektiert
11     for index ← begin to end do
         // mit 2 multiplizieren wegen zusätzlicher Cutoff-Zeiger
12       pointer ← Elf[2 · index];
13       nextIndex ← 2 · index + 1;
14       if max{D_0} ∈ {lower[0], upper[0]} then
15         tidPointerEnd = |R|;
16       else
17         tidPointerEnd = Elf[2 · nextIndex + 1];
18       end
         // Aufruf von ScanMonolist/ScanDimlistCutoff
19     end
20     return L;
21   end
22 end
```

Algorithmus 4 : Verarbeitung der ersten Dimension des Elf als Einstiegspunkt für partielle Bereichsanfragen unter Ausnutzung von `Cutoffs`

Scan der ersten Dimension. Der Pseudocode zum Start der Anfrage in der ersten Dimension mit `Cutoffs`, welcher wieder über eine gesonderte Funktion realisiert wird, kann Algorithmus 4 entnommen werden. Gleich zu Beginn des Algorithmus

kann man die Stärke der `Cutoffs` für einen speziellen Fall erkennen. Liegt in der Anfrage lediglich auf der ersten Dimension eine Selektion vor (über den Parameter `lastSelDim` feststellbar (Zeile 1)), lassen sich sämtliche, in der Standardimplementierung eigentlich folgende Aufrufe der anderen Scan-Algorithmen, gänzlich überspringen. Mithilfe der `Cutoffs` kann die Menge der qualifizierten `TIDs` direkt als Ergebnis zurückgegeben werden. Dazu müssen lediglich die Start- ($tidPointerBegin$) und End-`Cutoff`-Zeiger ($tidPointerEnd$) ermittelt werden, was durch den direkten Zugriff über die Werte der Unter- und Obergrenzen schnell möglich ist (Zeile 2 bis 7). Nach Ermittlung dieser werden alle `TIDs`, für deren Index i in `Elf_TIDs` $tidPointerBegin \leq i < tidPointerEnd$ gilt, als Ergebnismenge zurückgegeben (Zeile 8).
Handelt es sich bei der ersten Spalte nicht um die letztselektierte, wird ähnlich zum Algorithmus 1 (ohne `Cutoffs`) verfahren (Zeilen 10 bis 20). Unterschied ist, dass die Dimensionsliste nun doppelt so lang ist, weshalb der Zugriff adäquat angepasst werden muss. Außerdem benötigt man für die Bearbeitung der jeweils letzten Werte der anschließenden Dimensionslisten für jeden Eintrag der ersten Dimension den `Cutoff`-Zeiger des Nachbareintrags (Zeilen 14 bis 18).

Scan der regulären Dimensionslisten. Auch beim Scan der restlichen Dimensionslisten (dargestellt in Algorithmus 5) ändert sich im Grunde genommen nur etwas für den Fall, dass man bei der Dimension angelangt ist, auf der der letzte Bereich der Anfrage definiert ist. Dann heißt es, den ersten (äußere Schleife, Zeilen 5 bis 24) und den letzten (innere Schleife, Zeilen 9 bis 21) Eintrag der Dimensionsliste zu finden, welche innerhalb des spezifizierten Intervalls liegen. Bleibt mindestens eines der beiden undefiniert, bedeutet das, dass kein Eintrag in der Dimensionsliste der Selektionsbedingung genügt (Zeile 25). Ansonsten werden der Ergebnismenge alle Einträge aus `Elf_TIDs` zugeordnet, deren Indexe zwischen dem ermittelten Beginn- (einschließlich) und Endzeiger (ausschließlich) liegen (Zeile 26). Sind `Cutoffs` hingegen noch nicht nutzbar, muss wie bei `ScanDimlist` (Algorithmus 2) vorgegangen werden mit dem Zusatz, dass hier wie bei der ersten Dimension immer der `Cutoff`-Zeiger des nächsten Wertes einer Schleifeniteration ermittelt werden muss.

Beispielhafter Aufruf. In Abbildung 2.11 ist der bekannte Beispiel-Elf mit `Cutoff`-Zeigern (wie normale Zeiger, nur kursiv hervorgehoben) und dem zusätzlichen Array `Elf_TIDs` zu finden. Die `Cutoffs` bestehen, wie zuvor festgelegt, bis zur vorletzten Dimension.
Darüber hinaus ist die zur Anfrage mit den Bereichsdefinitionen $[[0, 0], [0, 0], [*], [*]]$ gehörende Aufrufshierarchie in Tabelle 2.9 beziehungsweise visuell in Abbildung 2.12 zu finden. Die Funktions- und Parameternamen wurden erneut abgekürzt. Mit `SFDLC` ist `ScanFirstDimlistCutoffs`, mit `SDLC` `ScanDimlistCutoffs` und mit $pTPE$ der Parameter $parentTIDPointerEnd$ gemeint. Die Schleifen zum Finden der Beginn- und End-`Cutoff`-Zeiger wurden mit `findBegin` beziehungsweise `findEnd` substituiert.

```
Input : Bereichsdefinitionen durch lower[n] und upper[n]
        selektierte Dimensionen als Bitvektor selDims[n],
        Beginn der zu durchsuchenden Dimensionsliste beginPointer
        aktuelle Dimension dim,
        Ergebnismenge L mit TIDs
        letzte Dimension, auf der eine Selektion vorliegt (lastSelDim)
        End-Cutoff-Index des aufrufenden Knotens (parentTIDPointerEnd)

Procedure ScanDimlistCutoffs(lower[], upper[], selDims[], beginPointer, dim,
                             L, lastSelDim, parentTIDPointerEnd):
1     index ← beginPointer;
2     isLastEntry ← false;
3     if dim = lastSelDim then
          // akt. Dimension ist letztselektierte → Cutoffs nutzen
4         (tidPointerBegin, tidPointerEnd) ← N_DEF; index ← beginPointer;
5         do
              // ersten Wert innerhalb der Grenzen finden
6             isLastEntry ←isEndOfList(Elf[index]);
7             if Elf[index] ≥ lower[dim] then
8                 tidPointerBegin ←Elf[i + 2];
9                 while true do
                      // letzten Wert innerhalb der Grenzen finden
10                    if Elf[index] ≤ upper[dim] then
11                        if isEndOfList(Elf[index]) then
12                            tidPointerEnd ← parentTIDPointerEnd;
13                            break outer;
14                        else
15                            tidPointerEnd ←Elf[i + 5];
16                        end
17                    else
18                        break outer;
19                    end
20                    index ← index + 3;
21                end
22            end
23            index ← index + 3;
24        while not isLastEntry;
25        if N_DEF ∉ {tidPointerBegin, tidPointerEnd} then
26            L = L ∪ {Elf_TID[i] | tidPointerBegin ≤ i < tidPointerEnd};
27        end
28    else
          // Keine Cutoffs nutzbar → jeweils tidPointerEnd bestimmen und
             verfahren wie bei ScanDimlist
29    end
30 end
```

Algorithmus 5 : Rekursiver Scan von Dimensionslisten mithilfe von `Cutoffs`

Mit `insertCutoffTIDs` soll der Mengendefinition aus dem Pseudocode zu `ScanDimlistCutoff` (Algorithmus 5) zum Anhängen der `TID`s an die Ergebnismenge anhand der `Cutoff`-Zeiger nachempfunden werden.
Vergleicht man die Darstellungen mit ihren Pendants ohne `Cutoffs` (Tabelle 2.8 und Abbildung 2.10) stellt man einen klaren Vorteil der `Cutoff`-Variante im geringeren Funktionsaufrufs- und Prüfungsoverhead für ebendiese Anfrage fest. Der „Trade-off", den es dafür in Kauf zu nehmen gilt, ist der gesteigerte Aufwand beim Bauen des Elf sowie der erhöhte Speicherbedarf des Elf insgesamt.

`Elf`	0	1	2	3	4	5	6	7	8	9
`Elf`[00]	[04]	[*00*]	[23]	[*03*]	(1)0	[10]	[*00*]	-1	-[20]	[*02*]
`Elf`[10]	(2)1	-[16]	[*00*]	-2	-[18]	[*01*]	(3)1	-T4	(4)2	-T2
`Elf`[20]	(5)1	1	-T6	(6)-1	[26]	[*03*]	(7)0	-[35]	[*03*]	1
`Elf`[30]	-[37]	[*04*]	-2	-[39]	[*05*]	(8)0	-T1	(9)2	-T3	(10)2
`Elf`[30]	-T4									

`Elf_TIDs`	0	1	2	3	4	5	6	7	8	9
`E_T`[00]	T4	T2	T6	T1	T3	T5				

Abbildung 2.11: Elf aus Abbildung 2.6 mit `Cutoffs` und dem dazugehörigen `Elf_TIDs`-Array

Aufrufende Funktion (Caller)	Prüfung	aufgerufene Funktion (Callee)
`SFDLC`	(a) `Elf`[0]	(1) `SDLC`(bP: 4, d: 1, L: $\{\}$, $pTPE$: 3)
(1) `SDLC`(bP: 4, d: 1, L: $\{\}$, $pTPE$: 3)	(b) `findBegin`	-
(1) `SDLC`(bP: 4, d: 1, L: $\{\}$, $pTPE$: 3)	(c) `findEnd`	(2) $L =$ `insertCutoffTIDs`($b:0$, $e:2$)
(2) `insertCutoffTIDs`($b:0$, $e:2$)	(d) $0 \leq i < 2$	-
`SFDLC`	-	**return** $L = \{\text{T4,T2}\}$

Tabelle 2.9: Aufrufshierarchie und Prüfungen der Scan-Algorithmen des Elf aus Abbildung 2.9 bei der partiellen Bereichsanfrage $[[0,0],[0,0],[*],[*]]$ unter Zuhilfenahme von `Cutoffs`

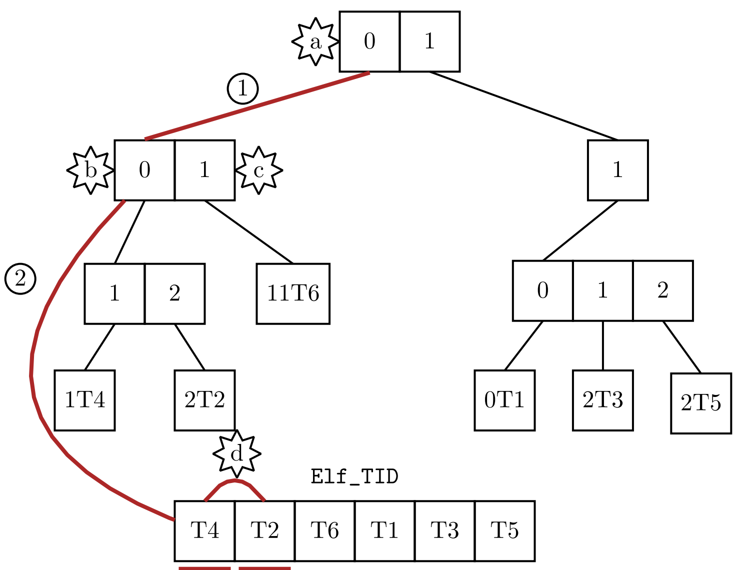


Abbildung 2.12: Visuelle Darstellung der Aufrufshierarchie und Prüfungen aus Tabelle 2.9

2.2.4 Zusammmenfassung

Im vorangegangen Abschnitt wurden die Konzepte rund um die Struktur und die Erstellung der multidimensionalen Indexstruktur Elf sowie die Anfragebearbeitung auf dieser ausführlich erläutert.
Dazu wurde zu Beginn der Dwarf von Sismanis et al. vorgestellt und motiviert. Daraus hervorgehend wurde beschrieben, wie der Elf aufgbaut ist und wie man ihn anhand einer Relation für diese als Index erzeugen kann. Bezüglich der Struktur wurden die Optimierungen der Linearisierung, besonderen Speicherung der ersten Dimension und der `Monolisten` zur Eliminierung einelementiger Dimensionslisten präsentiert.
Diese Anpassungen des ursprünglichen konzeptuellen Designs stellen die Basis für alle weiteren Betrachtungen dieser Arbeit bezüglich des Elf dar. Gleiches gilt für die drei vorgestellten Scan-Algorithmen zur effizienten Bearbeitung von Anfragen mit Selektionen. Für diese Algorithmen wurde zudem die sie beschleunigende Datenstruktur der `Cutoffs` erklärt und abschließend gezeigt, für welche Anfragearten sie besonders vorteilhaft sind.

2.3 SIMD: parallele Datenverarbeitung auf Instruktionsebene

Nachdem im vorangegangenen Kapitel sowohl der Aufbau als auch der Einsatz des Elf als multidimensionale Indexstruktur ausführlich beschrieben wurden, soll nun der zweite zentrale Aspekt dieser Arbeit, SIMD (kurz für ***S**ingle **I**nstruction, **M**ultiple **D**ata*), beleuchtet werden.
Während der Vorstellung des Elf wurden sukzessive strukturelle Optimierungen des Layouts und der darauf basierenden Algorithmen motiviert und angewandt. Diese Anpassungen zur Effizienzsteigerung fanden bislang ausnahmslos auf der *Softwareebene* statt. Mit SIMD soll nun auch die *Hardwarekomponente* zur Verbesserung der Berechnungen und programmatischen Abläufe zu Hilfe gezogen werden.
Zu Beginn soll zunächst der Ursprung und damit die Grundidee von SIMD beleuchtet werden.

2.3.1 Flynn'sche Taxonomie

1966 stellte Michael J. Flynn eine Klassifikation von Computerarchitekturen vor, welche unter der Bezeichnung *Flynn'sche Taxonomie* bekannt geworden ist.
In dieser differenziert Flynn grundlegend vier verschiedene Designs anhand der Anzahl der in ihnen gleichzeitig zur Verfügung stehenden und verarbeitbaren Befehls- und Datenströme [Fly72]:

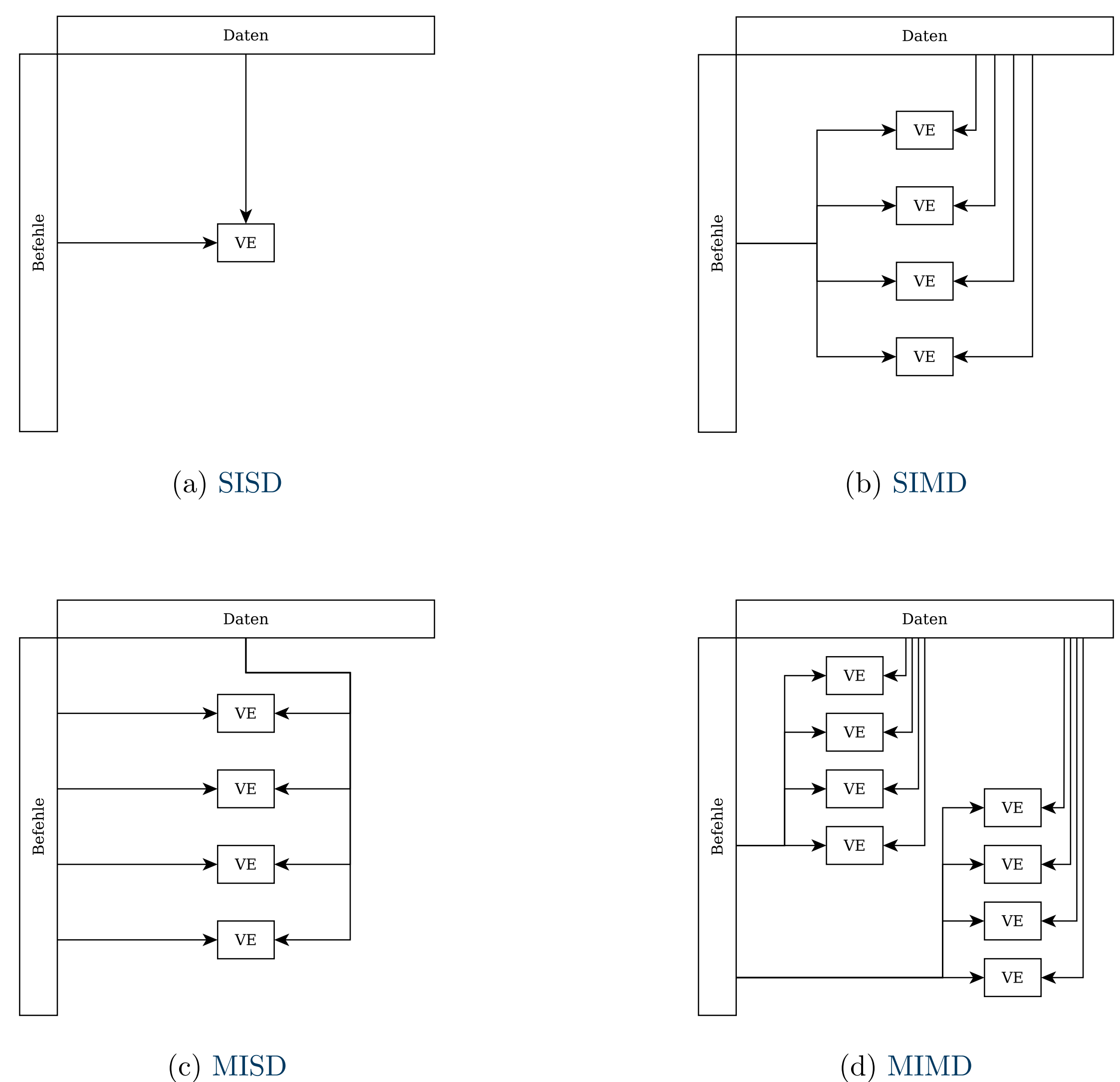
1. SISD (***S**ingle **I**nstruction, **S**ingle **D**ata*): Ein einzelner Prozessor ist für das Holen der Instruktionen und das Durchführen von Operationen verantwortlich. Aufgaben können daher nur sequentiell abgearbeitet werden (siehe Abbildung 2.13a).

2. SIMD (***S**ingle **I**nstruction, **M**ultiple **D**ata*): Einzelne Verarbeitungseinheiten (kurz VEs, engl. *processing units* (PUs)) sind in der Lage, eine Operation auf verschiedenen Datenströmen innerhalb eines Verarbeitungszyklus gleichzeitig durchzuführen (siehe Abbildung 2.13b).

3. MISD (***M**ultiple **I**nstruction, **S**ingle **D**ata*): Die Vearbeitungseinheiten von Maschinen solcher Klasse können verschiedene Operationen auf ein und demselben Datenstrom in einem Zyklus verarbeiten (siehe Abbildung 2.13c).

4. MIMD (***M**ultiple **I**nstruction, **M**ultiple **D**ata*): Verschiedene Instruktionen können durch diese Rechner auf unterschiedlichen Datenströmen simultan bearbeitet werden (siehe Abbildung 2.13d).

Von diesen vier Designs wurde SIMD als „revolutionär" eingestuft, da wissenschaftlichen Anwendungen oft Berechnungen zugrunde lagen, bei denen uniforme Anweisungen über einzelne Elemente von Arrays oder Matritzen benötigt wurden. Genau dafür eignet sich SIMD wie keine andere der Architekturen [CGS97].



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Abbildung 2.13: Flynn'sche Taxonomie: Klassifizierung von Rechnerarchitekturen nach der Möglichkeiten der gleichzeitigen Verarbeitbarkeit von Daten- und Befehlsströmen durch die Verarbeitungseinheit (VE) (angelehnt an [PS16])

2.3.2 Historische Entwicklung von SIMD

Hersteller moderner CPUs werden dem Prinzip von SIMD gerecht, eine Operation auf verschiedenen Daten gleichzeitig durchzuführen, indem sie dedizierte Register innerhalb der CPU mit einer festen Größe zur Speicherung reservieren. Darüber hinaus braucht es spezielle Operatoren, um zum einen Daten in diese Register zu laden und zum anderen die eigentlichen Befehle auf ihnen ausführen zu können.
Am Beispiel des Herstellers Intel lässt sich ein beständiger Entwicklungsprozess der Hardware für die Unterstützung von SIMD erkennen. Seit deren Erstimplementierung mit dem Namen *Multi Media Extension*, kurz MMX, aus dem Jahre 1997 hat Intel die unterstützten Operatoren und die Größe der Register sukzessive erweitert.

Eine Aufstellung der Historie ist in Tabelle 2.10 zu finden. Hier wurde sich auf diejenigen Befehle beschränkt, die für diese Arbeit wesentlich sind.

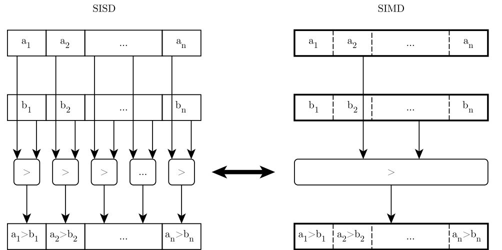
Name	Jahr	Wesentliche Erweiterung	RG	Auswahl hinzugekommener Befehle
MMX	1997	Einführung von 64-Bit Registern für SIMD-Operationen auf Integern	64	Vergleich auf Größersein Logisches UND Logisches XOR Logisches UND NICHT
SSE	1999	Verdopplung der Registerbreite Unterstützung von Gleitkommazahlen mit einfacher Genauigkeit (float)	128	Konvertieren des Vektors (vertikal) in eine Zahl (horizontal)
SSE2	2000	Unterstützung von Gleitkommazahlen mit einfacher Genauigkeit (double)	128	
SSE3	2004	Einführen komplexerer arithmetischer Operatoren, vor allem für grafische Anwendungen	128	
SSSE3	2006	Verbesserte Befehle für packed integers	128	
SSE4.1	2006	Neue Instruktionen für Bild-, Video- und 3D-Anwendungen	128	Prüfung auf vollständige Nichtgesetztheit eines Vektors
SSE4.2	2008	Ermöglichung erweiterter Operationen, die für Strings benötigt werden	128	
AVX	2008	Verdopplung der Registerbreite	256	
AVX2	2013	Adaptation der meisten SSE/AVX Integer-Instruktionen auf die neue Größe	256	
AVX-512	2013	Verdopplung der Registerbreite	512	Anzahl der gesetzten Werte im Vektor

Tabelle 2.10: SIMD-Erweiterungen und deren Registergrößen in Bit (kurz RG) sowie ausgewählte Befehle mit Fortschritt von CPUs [Int16]

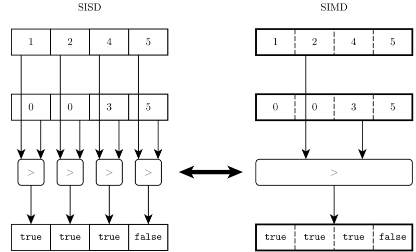
Mit Kenntnis über die möglichen Registergrößen und der in der Tabelle 2.10 erwähnten, für die spätere Optimierung des Elf relevanten Instruktionen, soll nun die Grundidee von SIMD anhand eines Beispiels nahegelegt werden.

2.3.3 Durchführen von Operationen auf SIMD-Registern

Zum Aufzeigen der Vorteile von SIMD soll der Vergleichsoperator „>" auf alle Werte zweier Vektoren angewandt und die Durchführungsweisen der jeweiligen Ansätze (SISD und SIMD) erörtert werden.
Innerhalb eines klassischen Einkernsystems, welches nach Flynn der SISD-Architektur entspricht, werden für diese Problemstellung alle Werte der zwei Vektoren jeweils geladen und die Vergleichsoperation anschließend auf ihnen durchgeführt. Wenn wir von einem Integer-Vektor der Länge vier ausgehen, heißt das, es müssten acht Werte geladen und mit ihnen vier >-Vergleiche durchgeführt werden.
Währenddessen sind mit SIMD unter der Voraussetzung, dass die Registergröße auf der CPU mindestens 128 Bit (= 16 Byte $\hat{=}$ 4 Integer) beträgt, bedeutend weniger Instruktionen notwendig – nämlich drei. Es müssen die beiden Vektoren in zwei Register geladen werden (zwei Ladeoperationen), worauf der Operand `cmpgt` (>, eine Vergleichsoperation) angewandt werden kann.
In Abbildung 2.14a werden die eben beschriebenen, notwendigen Schritte zur Durchführung des Befehls im Allgemeinen und in Abbildung 2.14b im Speziellen anhand eines Beispiels grafisch gegenübergestellt. An dieser Stelle sei angemerkt, dass die Ergebnisse innerhalb des SIMD-Ergebnisregisters nicht tatsächlich boolesche Werte annehmen, es aber aus Gründen der direkten Vergleichbarkeit dennoch so dargestellt ist.



(a) SIMD-registergrößenunabhängige Gegenüberstellung



(b) SIMD-Registergröße = 128 Bit, Integergröße = 4 Byte

Abbildung 2.14: Verlgeich der Ausführung einer Verlgeichsoperation auf Vektoren durch SISD und SIMD (angelehnt an [WEBS18])

Für das Auswerten von definierten Intervallen in (partiellen) Bereichsanfragen sind die Vergleichsoperatoren „<", „>" und „=" unabdingbar. Da SIMD diese Befehle, wie aus Tabelle 2.10 hervorgeht, von Haus aus unterstützt, ist die Verwendung von SIMD für die Ausnutzung der aus dem eben nahegebrachten Beispiel erkennbaren *Datenparallelität* ([PRR15]) im Elf vorstellbar. Die konkreten Anwendungen sollen im hierauf folgenden Kapitel der Konzeptionierung und Implementierung näher untersucht werden.

2.4 Zusammenfassung

In diesem Kapitel wurden die für diese Arbeit ausschlaggebenden Grundlagen vorgestellt. Dabei wurde der Fokus auf die zwei wesentlichen Aspekte der Untersuchungen gesetzt: den Elf als multidimensionale Indexstruktur und SIMD als möglicher Optimierungsansatz zur Ausnutzung von Datenparallelität im Elf.
Es wurden zunächst Indexstrukturen im Allgemeinen erläutert und ihre Klassifikationsmöglichkeiten hervorgehoben. Das Augenmerk wurde dabei insbesondere auf die Eigenschaft der Mutidimensionalität gelegt, die der Elf als spezielle Indexstruktur inne hat.
Im Anschluss wurde der Dwarf als Namens- und Ideengeber des Elf mit seinen Vorteilen vorgestellt. Mit dieser Grundlage konnte der konzeptionelle, baumbasierte Aufbau des Elf beschrieben werden. Es folgte die Herausarbeitung der Speicherung des Elf mit den Optimierungen der Linearisierung, differenzierten Handhabung der ersten Dimension und `Monolist`en. Als Abschluss des Abschnitts zum Elf wurde seine Nutzung zur Anfragedurchführung vermittelt. Dazu wurden die für Anfragen benötigten Scan-Algorithmen dargelegt. Zur Optimierung dessen sind die `Cutoff`s eingeführt und die ihretwegen adaptierten Standardalgorithmen präsentiert worden.

Das zweite Kernthema, SIMD, ist im Anschluss als ein Design der Flynn'schen Taxonomie zunächst allgemein und nachfolgend anhand eines Beispiels zur Gegenüberstellung mit SISD motiviert worden.

3. Konzeption und Implementierung

Nachdem im vorangegangenen Kapitel der Elf als multidimensionale Indexstruktur sowie die Grundidee und damit einhergehend die Vorteile von SIMD vorgestellt worden sind, sollen diese Themengebiete nun zusammengeführt werden. Konkret wird im Folgenden erläutert, wie SIMD für die Scan-Algorithmen im Elf zur Ausnutzung von Datenparallelität und damit gegebenenfalls zur Beschleunigung ihrer Durchführung genutzt werden kann.
Dazu wird zunächst beschrieben, welche Probleme prinzipiell beim Überführen von skalaren, nicht auf SIMD ausgelegten hin zu adäquat vektorisierten Code auftreten können. Im Anschluss wird beschrieben, welche konkreten Methoden es zur Nutzung von SIMD im Programmcode gibt.
Erst danach folgen die Erläuterungen zu den spezifischen datenstrukturellen und algorithmischen Anpassungen innerhalb des Elf zur Nutzung von SIMD.

3.1 Programmatische Nutzung von SIMD

Um SIMD und seine Vorteile innerhalb eines Programms nutzen zu können, gilt es zwei Fragen zu beantworten:

- Wie bereitet man den bestehenden Programmcode auf, sodass eine vektorbasierte Verarbeitung überhaupt möglich ist?
- Wie lässt man den angepassten Code kompilieren und anschließend ausführen?

Zur Beantwortung der ersten Frage sind verschiedene Hürden zu überwinden, die nun thematisiert werden sollen.

3.1.1 Herausforderungen

Bei den im Folgenden beschriebenen Problemen handelt es sich lediglich um eine Auswahl von jenen, die allgemein verbreitet und bekannt sind. Vorrangig werden die Herausforderungen berücksichtigt, die bei der SIMD-basierten Umsetzung der Elf-Algorithmen aufgetreten sind.
So wird beispielsweise die Eliminierung von Datenabhängigkeiten innerhalb von Schleifen aufgrund ihrer Irrelevanz in den Scan-Algorithmen des Elf bewusst nicht thematisiert, obgleich sie eine der am häufigsten auftretenden Herausforderungen bei der vektorbasierten Parallelisierung ist [HP11, NRZ06].

3.1.1.1 Kontrollflüsse in Schleifen

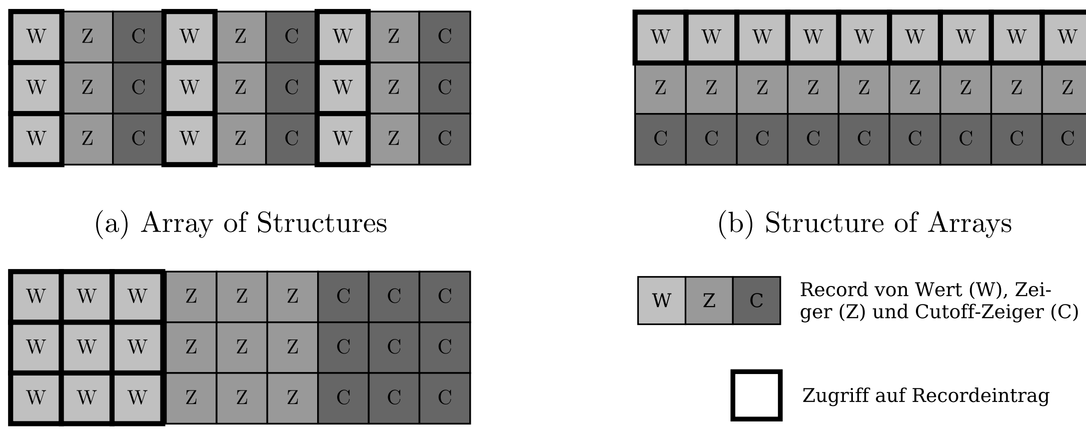
Treten innerhalb von Schleifen Bedingungen auf, die einzelne, skalare Elemente eines Vektors betreffen und weiter auszuführenden Code mit sich ziehen, kann dies für eine angestrebte Vektorisierung problematisch werden [EFGL14, LHW12]. SIMD ist nämlich prinzipiell darauf ausgelegt, auf Vektoren in ihrer Gänze zu operieren, nicht zur Auswertung der Ausprägung ausgewählter Einträge. Als Hilfsmittel, um solche Konstellationen dennoch abbilden zu können, dienen Masken. Diese können zur Quasiauswertung einzelner Elemente eines Vektors genutzt werden [Kar15].
Im Elf sich lassen solche Kontrollflussproblematiken gleich zweimal finden.
Zum einen wird beim Scan der regulären Dimensionslisten ein rekursiver Aufruf von `ScanDimlist` für alle jene Einträge ausgelöst, die innerhalb des definierten Intervalls liegen. Es wird also auf Grundlage des Ergebnisses einer Prüfung zu einem Element des Vektors gegebenenfalls weiterer Code, hier in Form einer Funktion, aufgerufen.
Zum anderen entscheidet ein einzelnes Element (jenes, bei dem das MSB gesetzt ist) darüber, ob die Schleife zum Scan der Dimensionsliste abgebrochen wird.

3.1.1.2 Limitierte Unterstützung von Datentypen und Operatoren

Nicht alle verbreiteten Arithmetik- oder Vergleichsoperatoren werden von SIMD unterstützt. Gleiches gilt für die Datentypen, die in ein SIMD-Register geladen werden können; auch bei ihnen gibt es Einschränkungen [NRZ06].
Es gibt auf der einen Seite Operatoren und Datentypen, die eine entsprechend fortgeschrittene Hardware benötigen. So werden beispielsweise die Vergleichsoperatoren $\leq$ und $\geq$ für die 256-Bit-weiten Register, die mit AVX eingeführt worden sind, erst mit dem fünf Jahre später erschienenen AVX-512 nativ unterstützt [Int19]. Auf der anderen Seite gibt es auch Operatoren beziehungsweise Datentypen, die selbst nicht durch die letzte Iteration von SIMD unterstützt werden – zum Beispiel die logische Negation. Treten solche Operationen auf, gilt es, sie adäquat zu substituieren (etwa mithilfe logischer Umformungen).
Im Elf tritt dieses Problem unter anderem bei den Intervalldefinitionen von Selektionsbedingungen auf. Diese Fenster sind in der Standardimplementierung mit einschließenden Grenzen versehen und müssen nun auf ausschließende umgemünzt werden, da die angestrebte Umsetzung in einer AVX2-Umgebung kein „$\geq$" implementiert hat.

3.1.1.3 Speicheranordnung gebündelter Datensätze

Das Holen von Daten ist um ein Vielfaches schneller, wenn die Daten aneinandergereiht im Speicher vorliegen [LHW12]. Daher ist ein sogenanntes „Array of



(a) Array of Structures

(b) Structure of Arrays

(c) Hybrid Structure of Arrays

Abbildung 3.1: Speicheranordnung gebündelter Datensätze: Zugriff auf „Array of Structures“ gegenüber „Structure of Arrays“

Structures“-Layout (kurz „AoS“) von Daten nicht geeignet, per SIMD parallel geladen zu werden. Bei diesem werden eine semantische Einheit (genannt *Record*) definierende Datensätze zusammengefasst und immer im Verbund abgespeichert. Es wird beim Holen derselben Elemente mehrerer Records ein Zugriff auf nichtbenachbarte Daten (sogenanntes Prinzip von *scatter and gather*) notwendig, welcher die Performanz erheblich negativ beeinflusst [LHW12]. Dieses Zugriffsproblem ist in Abbildung 3.1a am Beispiel des Elf erkennbar. Dargestellt sind die drei verschiedenen Arten von Einträgen von Dimensionslisten des Elf: Dimensionslistenwerte (W) und die dazugehörenden Zeiger (Z) zur nächsten Dimension sowie gegebenenfalls die korrespondierenden `Cutoff`-Zeiger (C). In der Abbildung 3.1a ist der suboptimale (weil verteilte), nicht horizontal zusammenhängende Zugriff auf ein Element (Dimensionslistenwert) mehrerer Records, deutlich sichtbar.
In einem sogenannten „Structure of Arrays“ (kurz „SoA“, siehe Abbildung 3.1b), dem Pendant zum „Array of Structures“, wird genau dieses Problem umgangen. Alle Dimensionslistenwerte verschiedener Records sind gebündelt im Speicher. Daher eignet es sich eben aufgrund der direkten Nachbarschaft der benötigten Daten wesentlich besser für SIMD [LHW12, MAPK19].
Die von Intel selbst als „best practice“ eingestufte dritte Art der Speicheranordnung [Int16], ein Hybrid aus den beiden vorher genannten Layouts (siehe Abbildung 3.1c), ist für den SIMD-Einsatz für die Scan-Algorithmen des Elf hingegen nicht gegenüber der „Structure of Arrays“ zu präferieren. Grund ist hierfür, dass die angestrebten SIMD-Operationen zur Dimensionslistenauswertung einzig und allein auf den Werten, nicht auf den Zeigern, operieren werden. Der Hybrid hingegen ist darauf ausgelegt, dass auf allen Attributen des Records immer zusammen und im gleichen Umfang gearbeitet wird. Dazu werden die Attribute in Blöcke gegliedert, die so groß sind wie die SIMD-Registerbreite [Int16].

3.1.2 Herangehensweisen zur Verwendung in Programmen

Während soeben die Herausforderungen umrissen worden sind, die es bei der Vektorisierung von skalar operierenden Programmen zu lösen gilt, wird nun betrachtet,

welche Werkzeuge es für die tatsächliche Umsetzung von SIMD gibt. Im Wesentlichen lassen sich zwei Methoden zur Nutzung von SIMD im Programmcode herausstellen: die automatische Vektorisierung durch den Compiler oder die explizite Nutzung von Intrinsiken. Im Folgenden sollen diese beiden Wege diskutiert und die Wahl für einen dieser für die Umsetzung im Elf begründet werden.

3.1.2.1 Automatische Vektorisierung

Wie die Bezeichnung vermuten lässt, wird bei der automatischen Vektorisierung, ein potenziell von SIMD profitierender Programmcode vom Compiler bei der Übersetzung erkannt und in das Programm ohne weiteres Zutun des Entwicklers integriert. Die Vorteile für dieses Vorgehen sind sehr naheliegend: es entsteht bei der Entwicklung kein Mehraufwand und es sind keine Instruktionen notwendig, die eventuell nicht von jeder Architektur unterstützt werden. Diese relative Hardwareunabhängigkeit ist daran erkennbar, dass viele Compiler diese automatische Erkennung integriert haben [BGGT02, CBB+05, NH06].
Auch für die zuvor beschriebenen, konkreten Schwierigkeiten, wie den Kontrollflüssen [Shi07] oder sogar der Überführung von „Arrays of Structures“ in „Structure of Arrays“ [MBZ+14, MMBS16], gibt es (teil-)automatisierte Lösungsansätze.
Jedoch sind solche automatischen Überführungen von skaleren Code durch den Compiler lediglich in einem begrenzten Umfang möglich. Je komplexer die Datenabhängigkeiten und Kontrollflüsse werden, desto unwahrscheinlicher bis zu unmöglich werden die Vektorisierungsmustererkennungen [EFGL14]. Tatsächlich werden für die Nutzung von SIMD nicht „nur“ Anpassungen explizit für die geänderte Hardwarebasis vorgenommen, sondern es muss die Implementierung der Algorithmen selbst angepasst werden [LHW12].

3.1.2.2 Nutzung von Intrinsiken

Das exakte Gegenteil der automatischen Vektorisierung ist der explizite Einsatz von Datentypen und Operationen, die direkt für das Operieren auf und mit SIMD-Registern vorgesehen sind: die sogenannten Intrinsiken. Die Nachteile der automatisierten Methode sind deshalb auch die Vorteile der Intrinsiken und umgekehrt. Die Entwicklung mit Intrinsiken geschieht in jedem Fall hardwarespezifisch, was die Portierungsfähigkeit des Programmcodes stark einschränkt – nicht nur zwischen CPU-Herstellern, sondern eben auch zwischen den einzelnen CPU-Generationen desselben Herstellers. Außerdem ist dieser Code exklusiv per vektorbasierter Ausführung nutzbar; eine „normale“, skalare Verwendung ist ausgeschlossen [LHW12, EFGL14].
Im Gegenzug bekommt der Programmierer allerdings Zugriff auf sämtliche von der Hardware unterstützten Datentypen und Operatoren und kann diese für seinen (komplexen) Anwendungsfall gezielt einsetzen. Dadurch werden mit den Intrinsiken weiterhin die besseren Ergebnisse erzielt – trotz der Fortschritte der automatischen Vektorisierung [MGG+11, RGB+15, HP11, WPP04].

3.1.2.3 Vergleich der Methoden und Wahl für den Elf

Eine direkte Gegenüberstellung der compilergesteuerten, automatischen Vektorisierung und der Verwendung von Intrinsiken wurde von Maleki et al. für verschiedene Applikationen im Multimediabereich vollzogen. Sie stellen fest, dass verschiedene

Anwendung	XLC		ICC		GCC	
	autom.	manuell	autom.	manuell	autom.	manuell
DNS	1.02	6.98	1.21	1.26	1.25	1.96
MILC	1.46	1.46	-	1.13	-	1.14
JPEG Encoder	-	1.39	1.33	2.13	1.15	1.79
MPEG2 Decoder	-	1.37	-	1.45	1.13	1.63

Tabelle 3.1: Vergleich des Speed-Ups durch automatische Vektorisierung und die Nutzung von Intrinsiken (manuell) gegenüber einer sequentiellen Umsetzung für Teilfunktionen von Multimediaanwendungen (aus [MGG+11])

Muster von den untersuchten Compilern (ICC, XLC, GCC) gar nicht erst erkannt werden. Bei denen, die erkannt wurden, war die manuelle Optimierung mit Intrinsiken in den meisten Fällen weiterhin überlegen. Die Speed-Ups der beiden Methoden gegenüber der sequentiellen Umsetzung können Tabelle 3.1 entnommen werden. Ein „-“ bedeutet hier, dass keine automatische Vektorisierung durch den Compiler stattgefunden hat. Man erkennt an diesen Ergebnissen, dass die Nutzung von Intrinsiken mit der automatischen Vektorisierung in jedem Fall mindestens gleichauf und in den meisten Fällen bezüglich ihres Speed-Ups überlegen ist.

Aufgrund der gezeigten Vorteile bei der Performanz und auch der besonderen Struktur des Elf (wie in Abschnitt 3.1.1.3 erwähnt handelt es sich bei den Dimensionslisten aufgrund der Verwendungsmuster um ein besonderes „Array of Structures“) ist die Umsetzung von SIMD für die Scan-Algorithmen des Elf über die Intel-Intrinsiken erfolgt.
Die konkreten Anpassungen, die bei den Herausforderungen schon angedeutet worden sind, sollen nun ausführlich beschrieben werden.

3.2 SIMD zur Parallelisierung der Algorithmen im Elf

Nach der Beschreibung der grundlegenden Funktionsweise von SIMD in Abschnitt 2.3, der Implementierungsherausforderungen und -methoden in Abschnitt 3.1, soll es nun darum gehen, SIMD und den Elf zusammenzuführen.
Bevor die konkreten benötigten Anpassungen im Elf erläutert werden, soll zunächst einmal analysiert werden, an welchen Stellen SIMD zur Parallelisierung von Programmcode infrage kommt.

3.2.1 Untersuchung der Parallelisierungsmöglichkeiten

Im Grundlagenkapitel wurden der Aufbau des Elf sowie die drei Algorithmen, die für die Durchführung von Anfragen notwendig sind, umfangreich geschildert.
In dieser Arbeit soll es vorrangig um die Nutzung des Elf für Anfragen gehen; die

Erstellung des Elf soll daher lediglich als Mittel zum Zweck betrachtet werden. Deshalb wird die Möglichkeit der Verwendung von SIMD zur Generierung des Elf etwa zur Sortierung der Attribute nicht näher betrachtet.
Da sich der Scanvorgang im Elf auf drei wesentliche Algorithmen aufteilen lässt, ist es sinnig, genau diese getrennt bezüglich ihrer Eignung für eine Optimierung durch SIMD zu betrachten. Den Anfang soll dabei die erste Dimension und ihr entsprechender Algorithmus, `ScanFirstDimlist` (siehe Algorithmus 1 beziehungsweise Algorithmus 4 mit `Cutoffs`), machen.

3.2.1.1 Scan der ersten Dimension

Die erste Dimension(sliste) im linearisierten Elf hat die Besonderheit, dass auf die in ihr gespeicherten Zeiger direkt über die Ausprägungen des ersten Attributs der indexierten Relation zugegriffen werden kann (siehe Abschnitt 2.2.2.3). Es ist folglich keinerlei Vergleich der Werte gegen untere oder obere Grenzen erforderlich. Es finden darüber hinaus keine Berechnungen auf den Einträgen der ersten Dimension statt. Es sind lediglich punktuelle Speicherzugriffe auf einzelne Positionen innerhalb der Liste notwendig; mit den daraus ausgelesenen Zeigern – egal ob Dimensionslisten- oder `Cutoff`-Zeiger – werden weitere Speicherzugriffe initiiert.
Für diesen Algorithmus findet sich *kein* Verwendungszweck der vektorbasierten Verarbeitung via SIMD.

3.2.1.2 Scan der restlichen Dimensionen

Die Dimensionen nach der ersten, die keinen `Monolist`-Charakter besitzen, werden über den Algorithmus `ScanDimlist`(`Cutoffs`) (siehe Algorithmus 2 und Algorithmus 5) durchsucht. In diesen findet, sofern eine Selektion auf der betrachteten Dimension vorliegt, eine Evaluation eines definierten Integer-Intervalls für ebensolche Werte statt.
Dafür wird eine Schleife durchlaufen, die dann terminiert, wenn ein Wert erreicht wird, der entweder größer als die vorgegebene obere Grenze ist oder das Listenende darstellt (über sein MSB erkennbar). Innerhalb der Schleife wird mit allen Werten, die in das Fenster fallen, ein rekursiver Aufruf initiiert. Das heißt mit jeder Schleifeniteration, die ein Element i betrachtet, ist die Prüfung $lower[dim] \leq i \leq upper[dim]$ notwendig.
Eine solche Schleife ist für eine datenparallele Verarbeitung mithilfe von SIMD prinzipiell geeignet, da keinerlei Datenabhängigkeit vorliegt und pro Iteration lediglich ein logischer Vergleich von Integern getätigt wird.
Es gibt allerdings zwei größere, wenngleich lösbare Herausforderungen, die es hier zur Nutzung von SIMD zu lösen gilt. Zum einen sind das die zwei Kontrollflüsse innerhalb der Schleife, die den rekursiven Aufruf oder Schleifenabbruch hervorrufen können. Zum anderen müssen die Heterogenitäten der Daten- (32- und 64-Bit-Integer) und Eintragstypen (Werte, Zeiger) innerhalb der Dimensionslisten berücksichtigt werden, da sie das Laden in die SIMD-Register und den anschließenden Vergleich erschweren (Stichwort Speicherlayout, siehe Abschnitt 3.1.1.3).

3.2.1.3 Scan der `Monolisten`

Das Durchsuchen der `Monolisten` durch `ScanMonolist` (siehe Algorithmus 3) ist ohne Einschränkungen mit einer Parallelisierung durch SIMD vereinbar.

Im Vergleich zu den regulären Dimensionslisten besitzen `Monolisten` in der Standardimplementierung von Haus aus zwei Vorteile, die den Einsatz von SIMD erleichtern. Erstens kommen in den `Monolisten` lediglich die Werte, die es vektoriell zu vergleichen gilt, vor – es herrscht in diesem Fall eine Homogenität der Eintragsarten und damit auch der Datentypen. Zweitens wird das Ende der zu durchsuchenden Schleife nicht durch einen markierten Eintrag gekennzeichnet, sondern mit dem Erreichen einer festen Anzahl an Iterationen. Diese lässt sich nämlich aus dem an den Algorithmus `ScanMonolist` übergebenen Dimensionsparameter *dim* und der Gesamtanzahl der Dimensionen leicht berechnen. Wie sich später zeigen wird, lässt sich die Länge gut über Masken in SIMD berücksichtigen und stellt einen einfacher abzubildenden Kontrollfluss als das Prüfen des MSBs dar. Außerdem gibt keinerlei Datenabhängigkeiten, die einer Verwendung von SIMD entgegensprechen oder diese verkomplizieren könnte.

3.2.2 Änderungen im Aufbau des Elf für SIMD

Zur Lösung der soeben angesprochenen Herausforderungen, die vorrangig beim Scannen der regulären Dimensionslisten auftreten, sind zunächst Anpassungen am Aufbau des Elf, und damit der Basis für die darauf angewandten Algorithmen, notwendig. Konkret geht es um die SIMD-optimierte Speicherung der Dimensionslisten zur Verbesserung des Ladens in die Register und zur vereinfachten Handhabung der Kontrollflüsse in `ScanDimlist`.

3.2.2.1 Elf als „Structure of Arrays“

Es wird sich zeigen, dass das Prinzip der „Structure of Arrays“ dem des „Array of Structures“ auch im Elf überlegen ist. Bevor die Umsetzung dessen im Elf detailliert wird, soll zunächst veranschaulicht werden, warum dieser Schluss naheliegend ist.

Vergleich „Array of Structures“ und „Structure of Arrays“ im Elf

Als Veranschaulichung, warum die aktuelle Speicherung des Elf als „Array of Structures“ für den Einsatz von SIMD ungeeignet ist, soll der Ausschnitt aus einem Elf in Abbildung 3.2 dienen. Die Notation und Farbgebung folgt hier den vorangegangen Beispielen: „einfache“ Zahlen sind die Dimensionswerte, gefolgt von ihrem Zeiger zur nächsten Dimension in eckigen Klammern und dem `Cutoff`-Zeiger darauf in selbigen Klammern jedoch zusätzlich in Kursivschrift. Es wird davon ausgegangen, dass in ein SIMD-Register acht 32-Bit-Integer geladen werden können und die Zeiger der Einfachheit halber ebenso von diesem Datentypen sind.

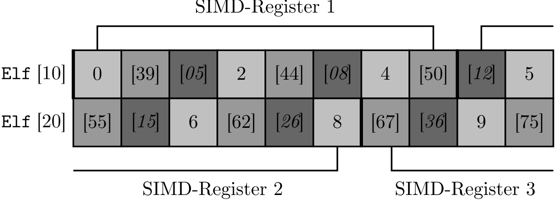


Abbildung 3.2: Heterogenität der Dimensionslisteneinträge: der Elf als „Array of Structures“

Unter dieser Annahme lassen sich in ein Register – wie in der Abbildung erkennbar ist – im Bestfall drei Dimensionswerte laden. Letztendlich kommt es für den Scan-Algorithmus ausschließlich auf die Dimensionswerte an, weil (eigentlich) nur diese über SIMD-Operationen mit der unteren und oberen Grenze der aktuellen Dimension verglichen werden sollen. Die Zeiger sind für diese Prüfung unwesentlich, ja sogar kontraproduktiv. Durch sie kann einerseits nicht einmal die Hälfte des SIMD-Registers mit den tatsächlich relevanten Daten gefüllt werden, andererseits werden zum Ausblenden der irrelevanten Zeiger auch noch zusätzliche Masken benötigt. Letzteres soll im Beispiel von Abbildung 3.3 deutlich werden.
Vergleicht man hier die untere Grenze ($lower[dim]$) mit der heterogenen Dimensionsliste (`Dimlist`) mit dem „>“-Operator, erhält man für sämtliche Zeiger ebenfalls den Wert `true`. Was aus der Perspektive der bloßen mathematischen Operation korrekt, semantisch aber wertlos ist. Die Vergleiche mit den Zeigern sind nichtssagend und müssen daher über eine – nicht trivial zu bestimmende – Maske und durch einen oder mehrere verknüpfte Operatoren herausgefiltert werden, ehe das gewünschte Endergebnis erreicht wird. Dieser Prozess soll im rechten Teilbild nachempfunden sein.

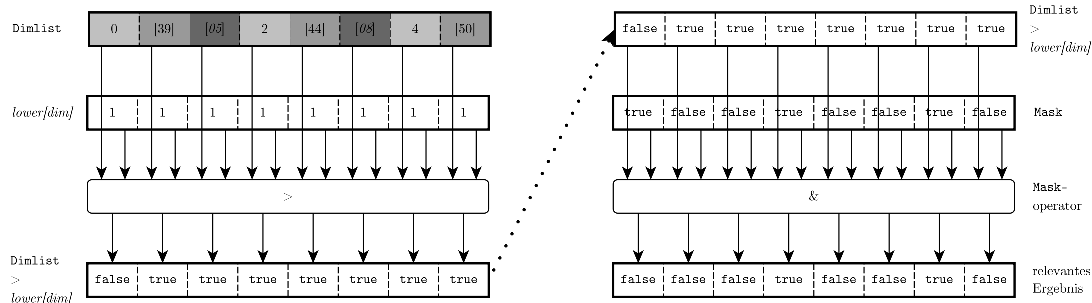


Abbildung 3.3: Beispiel-SIMD-Operation auf heterogenen Dimensionslisten

Zieht man den direkten Vergleich zwischen Abbildung 3.2 und Abbildung 3.4, in der eine homogene Speicherung innerhalb der Dimensionsliste umgesetzt wurde, wird das verlorene Potenzial aus Abbildung 3.2 deutlich. In Abbildung 3.4 können alle acht Registerplätze von den ausschlaggebenden Dimensionswerten besetzt werden.

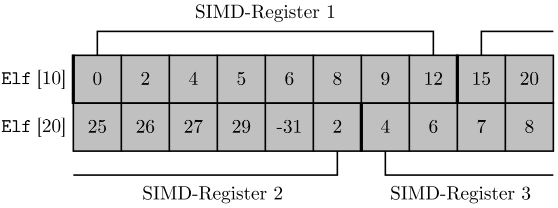


Abbildung 3.4: Homogenität der Dimensionslisteneinträge: der Elf „Structure of Arrays“

Für das erste SIMD-Register aus Abbildung 3.4 lässt sich damit ein vollständiger Vergleich mit den Grenzen für diese Dimension durchführen – ohne dass Masken

nötig wären[1]. Das wird in Abbildung 3.5 sichtbar. Das relevante Endergebnis, der Vergleich der Dimensionswerte mit der unteren Grenze, wird bereits nach einem SIMD-Lade und -Vergleichszyklus erzielt.

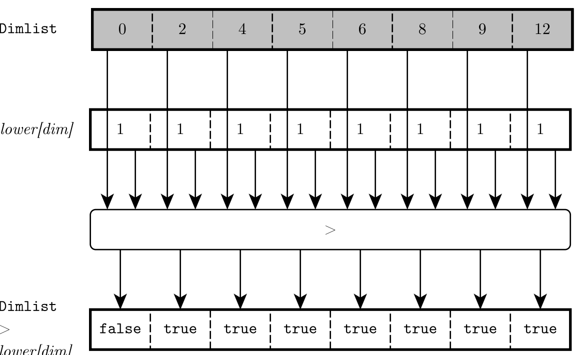


Abbildung 3.5: Beispiel-SIMD-Operation auf homogenen Dimensionslisten

In der ursprünglichen Implementierung gab es darüber hinaus noch ein weiteres Problem: die Dimensionslistenzeiger waren als 64-Bit-Integer umgesetzt, wohingegen die Dimensionslistenwerte und `Cutoff`-Zeiger durch 32-Bit-Integer repräsentiert worden sind. Es war also sowohl neben der bisher thematisierten Heterogenität der Eintragstypen auch eine der Datentypen gegeben. Dadurch würden pro Dimensionslistenzeiger zwei 32-Bit-Registerplätze vereinnahmt werden – das würde eine weitere Verringerung der tatsächlich für den Scan relevanten Werte im Register bedeuten.

Umsetzung

Zur Realisierung des „Structure of Arrays“-Prinzips wurden alle im Elf vorkommenden Eintragstypen voneinander getrennt und in eigens für sie vorgesehene, separate Arrays überführt. Dadurch entstehen beim Erstellen des Elf insgesamt maximal fünf Arrays – drei mehr als bei der Standardimplementierung:

- `Elf` (32-Bit-Integer): beinhaltet ausschließlich Dimensionswerte, die nicht in `Monolisten` vorkommen.
- `Elf_TIDs` (32-Bit-Integer): unveränderte Speicherung aller `TIDs` in der Sortierreihenfolge der Relation, die als Nebenprodukt bei der Elf-Erstellung entsteht. Wird nur gefüllt, wenn `Cutoff`-Zeiger verwendet werden.
- `Monolists` (32-Bit-Integer): enthält die Werte und `TIDs` sämtlicher `Monolisten` des Elf.
- `Child_Pointers` (64-Bit-Integer): speichert an Position i den Zeiger auf den Beginn der zu `Elf`$[j]$ gehörenden Kind-Dimensionsliste (mit $j = i - |DIM_1|$)[2]. Folgt auf `Elf`$[j]$ eine `Monolist` repräsentiert der Zeiger `Child_Pointers`$[i]$

[1] Im zweiten SIMD-Register würden auch hier Masken benötigt werden, da das letzte Element (2) nicht mehr zur aktuellen Dimensionsliste gehört und demzufolge ausgeblendet werden muss.

[2] Dies ergibt sich aufgrund der besonderen Beschaffenheit und Speicherung der ersten Dimension und wird im weiteren Verlauf näher erklärt.

einen Index aus `Monolists`, in den restlichen Fällen einen aus `Elf`. Wird auf `Monolists` verwiesen, wird der Zeiger zur Unterscheidung über sein MSB markiert.

- `Cutoff_Pointers` (32-Bit-Integer): speichert an Position i den `Cutoff`-Zeiger für `Elf`$[j]$ (mit $j = i - |DIM_1|$). Dieser Zeiger entspricht einem Index aus `Elf_TIDs` (Näheres dazu in Abschnitt 2.2.3.2).

Die Sinn- und Vorteilhaftigkeit der Auslagerung der beiden Zeigertypen in separate Arrays wurde zuvor ausführlich erörtert. Das Übertragen der Einträge von `Monolisten` in eine gesonderte Struktur lässt sich unter anderem mit der Konsequenz der Einhaltung des „Structure of Array“-Prinzips begründen. Beim `ScanDimlist`-Algorithmus stünden die `Monolist`-Werte einer SIMD-Operation generell nicht im Wege, da sie sich nicht *innerhalb* einer Dimensionsliste befinden, sondern höchstens an eine anschließen. Das ist kein Hindernis, da die Ergebnisse eines SIMD-Vergleichs, die nach dem Ende einer Dimensionsliste liegen, ohnehin per Maske ausgeblendet werden müssen. Ob dies Werte einer `Monolist` oder einer folgenden Dimensionsliste sind, ist unerheblich. Darüber hinaus befänden sich in `Monolisten` und Dimensionslisten die gleichen Datentypen. Allerdings, und das spricht für das gesonderte `Monolists`-Array, ergäben sich innerhalb des Zeiger-Arrays `Child_Pointers` Lücken; nämlich in den Bereichen, an denen die `Monolisten` in `Elf` stehen würden, weil diese Art der Dimensionslisten keine Zeiger beinhaltet.

`Elf`	0	1	2	3	4	5	6	7	8	9
`Elf`[00]	(1)0	-1	(2)1	-2	(6)-1	(7)0	1	-2		

`Child_Pointers`	0	1	2	3	4	5	6	7	8	9
`Ch_P`[00]	[00]	[04]	[02]	-[04]	-[00]	-[02]	[05]	-[07]	-[09]	-[11]

`Monolists`	0	1	2	3	4	5	6	7	8	9
`ML`[00]	(3)1	-T4	(4)2	-T2	(5)1	1	-T6	(8)0	-T1	(9)2
`ML`[10]	-T3	(10)2	-T5							

`Cutoff_Pointers`	0	1	2	3	4	5	6	7	8	9
`CO_P`[00]	[00]	[03]	[00]	[02]	[00]	[01]	[03]	[03]	[04]	[05]

`Elf_TIDs`	0	1	2	3	4	5	6	7	8	9
`E_T`[00]	T4	T2	T6	T1	T3	T5				

Abbildung 3.6: Linearisierter Elf aus Abbildung 2.8 als „Structure of Arrays“

Für den Elf aus dem Grundlagenkapitel, beispielsweise in Abbildung 2.8 dargestellt, können die Arrays Abbildung 3.6 entnommen werden. Hier lassen sich bereits einige Eigenschaften oder gar Besonderheiten der einzelnen Strukturen erkennen, die im Folgenden aufgegriffen werden sollen.

Besonderheiten der Elf-Arrays. Da im Elf nur noch die Dimensionswerte zu finden sind und in der ersten Dimension lediglich Zeiger gespeichert werden, ist diese Dimension nicht im `Elf`-Array zu finden. Der erste Eintrag in `Elf` entspricht daher im Regelfall dem Beginn der Dimensionsliste der zweiten Dimension, die den ersten Wert des ersten Attributs als Präfix besitzt. Das ist nur dann nicht der Fall, wenn dem ersten Wert der ersten Spalte eine `Monolist` folgt.
`Child_Pointers` besitzt wegen jener Sonderstellung der ersten Dimension, die eben nicht in `Elf` zu finden ist, immer $|DIM_1|$ (Anzahl eindeutiger Werte der ersten Dimension) mehr Einträge als `Elf`. Für die Abfrage des Zeigers, der zum j-ten Wert aus dem Elf gehört, muss daher an die Stelle $j + |DIM_1|$ in `Child_Pointers` geschaut werden. Dieses Array besitzt als einziges 64-Bit-Integer als Einträge und ist garantiert vollständig besetzt, weist also keine Lücken auf.
Anders verhält sich Letzteres bei `Cutoff_Pointers`. Nur wenn, wie im Beispiel aus Abbildung 3.6, die `Cutoff`-Zeiger bis zur vorletzten Dimension gebildet werden, ist `Cutoff_Pointers` vollständig besetzt. In diesem Fall besitzt es auch genauso viele Einträge wie `Child_Pointers`, weil für jeden Eintrag in `Elf` ein Eintrag in `Cutoff_Pointers` existieren muss. Andernfalls kann es in `Cutoff_Pointers` Bereiche im Array geben, die nicht belegt sind. Grund ist hierfür, dass der Elf per Pre-Order-Suche aufgebaut wird. Diese Suchform besitzt einen „vertikalen“ Charakter, weil sie den Baum nicht Ebene für Ebene durchgeht, sondern zunächst einzelnen Pfaden bis zu den Blättern folgt. Die Festlegung einer Ebene, bis zu der die `Cutoff`-Zeiger gebildet werden, definiert jedoch eine „horizontale“ Beschränkung, wodurch Lücken entstehen (können).

Das Erstellen der Arrays erfordert verständlicherweise eine Anpassung des Algorithmus zum Bauen des Elf, die allerdings nicht Gegenstand dieser Arbeit sein soll. Ein Aspekt, der dabei jedoch eine nicht unwesentliche Rolle einnimmt und für spätere Interpretationen von Ergebnissen interessant sein könnte, ist die Ermittlung der maximalen Einträge, die die Arrays besitzen können.

Maximale Größen der Arrays bei besonderen Datenkonstellationen. Wie bereits soeben erwähnt, stehen die Größen von `Elf`, `Child_Pointers` und `Cutoff_Pointers` in direktem Zusammenhang miteinander.
Die maximale Größe dieser Arrays soll im Folgenden hergeleitet werden, indem die maximale Größe von `Elf` ermittelt wird. Im Wesentlichen lässt sich dieses Optimierungsproblem auf zwei Teilprobleme aufteilen:

1. Minimierung der Anzahl der Dimensionswerte in den `Monolisten` des Elf
2. Maximierung der Breite des Elf-Baums durch Minimierung der Präfixredundanzen

Im Elf-Array sind alle Dimensionslistenwerte enthalten, bis auf jene, die in Monolisten vorkommen. Daher ist Punkt 1 logisch: minimiert man die Anzahl der Dimensionswerte in den Monolisten, maximiert man die Anzahl der Werte in Elf. Monolisten treten immer dann auf, wenn der Pfad zur im Blatt befindlichen TID[1] eindeutig ist, sodass, mit anderen Worten, keine Redundanzeliminierung mehr möglich ist. Der Inhalt von Monolisten lässt sich mit einer speziellen Datenkonstellation ausschließlich auf TIDs reduzieren, wodurch ausnahmslos alle Dimensionswerte in Elf zu finden sind. Dazu müssen die Daten folgende Besonderheit aufweisen: der Pfad zu einer TID muss einen redundanten Präfix bis einschließlich zur vorletzten Dimension besitzen. In der letzten Dimension folgt dann ein praktisch die TID identifizierender, eindeutiger und somit nicht redundanter Wert. Ein solcher Fall ist in Tabelle 3.2 zu finden. Fasst man Pfade im Elf als mathematische Folgen auf, lässt sich diese Datenzusammenstellung mit den Pfaden p, q des Elf wie folgt formulieren: $\forall p \exists q : p_i = q_i \wedge p_{|DIM|} \neq q_{|DIM|}$ für $0 \leq i < |DIM|$

TID	C1	C2	C3	C4
T5	1	1	1	1
T2	1	1	1	2
T1	2	1	1	1
T4	2	1	1	2
T6	3	1	1	1
T3	3	1	1	2

Tabelle 3.2: Beispieldaten, die die maximale Länge von Elf, Child_Pointers und Cutoff_Pointers hervorrufen

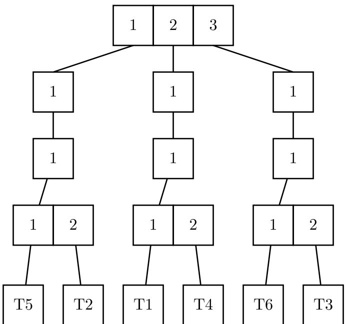


Abbildung 3.7: Beispiel-Elf zur Datenkonstellation aus Tabelle 3.2, die die maximale Länge von Elf, Child_Pointers und Cutoff_Pointers hervorruft

Mit der Minimierung der Größe der Monolisten, maximiert man, visuell vorgetellt, die Höhe des Baums[2] zum Elf. Nun gilt es, Punkt 2, die Maximierung der Breite des

[1] Es kann im Blatt auch eine TID-Liste vorkommen; zur vereinfachten Formulierung wird aber im Folgenden nur von einer TID ausgegangen.

[2] Zwar wurde zuvor von einer festen Suchtiefe als Vorteil des Elf gesprochen, jedoch schloss das Monolisten mit ein – diese werden an dieser Stelle außen vor gelassen.

Baums zu erreichen.
Will man die Breite des Baums maximieren, muss man dazu in der ersten Dimension beginnen, weil dort der „Branch-out“ am höchsten ist. Um an späterer Stelle (d^*) längere Dimensionslisten zu erhalten, bräuchte es bis zu d^* mehr Präfixredundanzen – doch diesen Vorteil des Elf gilt es nun zu vermeiden. Die maximale Breite, das heißt Anzahl der Elemente der Dimensionsliste, würde in Dimension 1 erreicht werden, wenn alle Ausprägungen des ersten Attributs nur einmal vorkämen. Dann wäre die Breite mit der Tupelanzahl der Relation gleichzusetzen. Dies hätte aber zur Folge, dass sich auf all diese Werte jeweils `Monolisten` anschließen würden. Letztere gilt es jedoch, wie in Punkt 1 gezeigt, bis zur letzten Dimension zu vermeiden. Daher darf *kein* Wert der ersten Dimension eindeutig sein. Man erhält die größte Anzahl nicht eindeutiger Werte genau dann, wenn jeder Wert exakt einen „Partner“ hat. Das heißt, jeder Wert muss genau zweimal vorkommen, damit keine `Monolisten` entstehen, aber gleichzeitig die Länge der ersten Dimensionsliste maximiert wird. Innerhalb der Dimensionslisten der zweiten Dimension, die an eine derartige erste anschließen, können jeweils nur zwei Werte auftreten – wären diese *nicht* gleich, entstünden wieder `Monolisten`. Nur im Falle einer Gleichheit und damit Redundanz, ließen sich `Monolisten` vermeiden. Dies setzt sich bis zur letzten Dimension so fort. Hier muss es dann je Präfix bis dorthin genau zwei unterschiedliche Werte geben.

Die Kombination der Punkte 1 und 2 beschreibt die Worst-Case-Datenkonstellation, die die maximale Länge der Arrays `Elf`, `Child_Pointers` und `Cutoff_Pointers` hervorruft. Dazu ist ein Beispiel-Elf mit der bereits erwähnten Datengrundlage aus Tabelle 3.2 in Abbildung 3.7 dargestellt.

Führt man die Erkenntnisse zusammen, ergeben sich Formeln für die Arraygrößen, die abhängig von der Tupel- und Spaltenanzahl der Relation sind. Diese sollen im Folgenden erklärt werden.
Sei $|R|$ die Anzahl der Tupel der Relation und $|DIM|$ die Anzahl der Dimensionen. Die maximalen Längen der Arrays lassen sich bei einer derartigen Datenkonstellation als Summen zweier Größen berechnen. Der erste Summand ist $|R|$, da es auf der letzten Ebene vor den `TIDs` im Elf so viele Einträge gibt wie Tupel existieren. Der zweite Summand ergibt sich aus der Größe der restlichen Dimensionen. Diese beinhalten jeweils halb so viele Einträge, also $|R|/2$. Die Anzahl der restlichen Dimensionen, das heißt derer ohne die bereits berücksichtigte letzte Dimension, ist für `Child_Pointers` und `Cutoff_Pointer` $|DIM| - 1$. In `Elf` entfällt zusätzlich die erste Dimension, weil dort nur Zeiger gespeichert werden, also: $|DIM| - 2$. Zusammenfassend ergeben sich folgende Formeln:

- $max\{|\texttt{Elf}|\} = |R| + \frac{|R|}{2} \cdot (|DIM| - 2) = \frac{|R|}{2} \cdot |DIM|$
- $max\{|\texttt{Child_Pointers}|\} = |R| + \frac{|R|}{2} \cdot (|DIM| - 1) = \frac{|R|}{2} \cdot (|DIM| + 1)$
- $max\{|\texttt{Cutoff_Pointer}|\} = max\{|\texttt{Child_Pointers}|\}$

Für die anderen beiden Arrays ist ihre maximale Länge leichter herleitbar.
Wie auch schon in Abschnitt 2.2.3.2 erwähnt, ist die Größe von `Elf_TIDs` in jedem Fall mit der Anzahl der Tupel in der indexierten Relation gleichzusetzen, sobald `Cutoff`-Zeiger beim Erstellen berücksichtigt werden – egal bis zu welcher Dimension:

- $max\{|\texttt{Elf_TIDs}|\} = |R|$

Der Worst Case für `Monolists` ist ein anderer als für `Elf` – praktisch die entgegengesetzte Konstellation. Er wurde bei der Herleitung der maximalen Einträge in `Elf` bereits erwähnt. Genau dann, wenn in der ersten Dimension nur eindeutige Werte vorherrschen, sind alle folgenden Suffixe `Monolisten`, die konsequenterweise in `Monolists` gespeichert werden müssen. Es kämen in diesem Fall $|R|$ `TIDs` und $R \cdot (|DIM| - 1)$ Attributsausprägungen in `Monolists` vor:

- $max\{|\texttt{Monolists}|\} = |R| \cdot (DIM - 1) + |R| = |R| \cdot |DIM|$

Indirektion des Zeigerzugriffs. Der Vorteil für SIMD, der durch die Auffassung des Elf als „Structure of Arrays“ beim Scan der Dimensionslisten entsteht, wurde ausführlich hervorgehoben. Jedoch gibt es bezüglich der Traversierung im Elf durch das vorgeschlagene Speicherlayout einen nicht unerheblichen Nachteil. Um von einer Dimensionsliste zur darunterliegenden zu springen, ist es nun nicht mehr ausreichend, den benachbarten Zeiger innerhalb desselben Arrays auszulesen und dann an die von ihm verwiesene Position zu springen. Stattdessen ist ein Navigieren in ein anderes Array, dem `Child_Pointers`-Array, erforderlich, nur um dann wieder in das „Ursprungsarray“ `Elf` zurückzukehren. Besonders bei zunehmender Größe der involvierten Arrays könnte ein solcher zusätzlicher Sprung im Speicher ressourcenaufwändig werden.

3.2.2.2 Explizite Speicherung der Länge von Dimensionslisten

Mit der Separierung der Typen von Einträgen in den Dimensionslisten durch die eben erläuterte Aufsplittung des linearisierten Elf, wurde ein wesentlicher Schritt für die Nutzung von SIMD in `ScanDimlist(Cutoffs)` gemacht. Es bleibt jedoch noch die Herausforderung der SIMD-effizienten Prüfung, wann eine Dimensionsliste aufhört und somit die Schleife zum Intervallabgleich ihre letzte Iteration erfährt.

Wie in Abschnitt 2.2.2.2 beschrieben, wird die Länge einer Dimensionsliste in der Standardimplementierung implizit über das Markieren des MSBs des letzten Eintrages gespeichert. Zum Verständnis, warum diese Vorgehensweise für eine vektorielle Bearbeitung ungünstig ist, soll zunächst erläutert werden, wie das Markieren und anschließende Prüfen des `MSBs` in der sequentielle Implementierung umgesetzt ist.

Die Dimensionslistenwerte im Elf werden als vorzeichenlose (engl. *unsigned*) 32-Bit-Integer gespeichert. Zum Markieren des letzten Eintrages werden die letzten Werte beim Erstellen des Elf über ein bitweises Oder mit 2^{31} verknüpft. Dadurch wird der beim vorzeichenlosen Integer mögliche Bereich von 2^{31} bis $2^{32} - 1$ aufgegeben.

Bei den Scan-Algorithmen wird zum Erkennen dieses Kriteriums geprüft, ob ein Element größer oder gleich $2^{31} - 1$ ist. Das heißt eine Scan-Schleife wird so lange ausgeführt wie das aktuelle Element kleiner als eben jene Zahl ist. Zusätzlich ist es erforderlich, das Markieren wieder rückgängig zu machen, damit auch das Listenende gegen den eventuell definierten Bereich geprüft werden kann. Dazu wird der Endwert mit per bitweisen Unds mit $2^{31} - 1$ verknüpft. Tatsächlich kann aber, und so

ist es auch umgesetzt, jeder Listenwert direkt mit $2^{31}-1$ ohne Semantikverlust „verundet“ werden bevor er mit den Grenzen verglichen wird. Mit dieser Vorgehensweise benötigt es keine extra Handhabung des letzten Wertes für die Vergleichsoperation.

SIMD bei Umsetzung über MSB. Zur Hervorhebung der Schwierigkeiten, die bei der Nutzung dieser impliziten Speicherung der Länge mit SIMD auftreten, soll ein beispielhafter Scan über den Elf aus Abbildung 3.6 dienen.
Gehen wir für die Übersichtlichkeit der Darstellung von SIMD-Registern aus, die, entgegen der bisher angenommenen acht Dimensionswerte, nunmehr nur noch vier fassen können. Zum Scannen der Dimensionsliste $[0,-1]$ für das offene Intervall $(0,3)$ wird folglich der vierelementige Vektor $[0,-1,1,-2]$ in ein Register geladen. Zum Abgleich mit den definierten Grenzen und Herausfinden der Elemente, für die ein rekursiver Funktionsaufruf durchgeführt werden muss sind folgende Schritte notwendig (wenn SIMD zum Einsatz kommen kann, wird das in Klammern deutlich):

1. Vergleich der Dimensionsliste mit dem Intervall
   (a) Eliminierung der Markierungen der MSBs in allen Elementen des Registers mit den Dimensionslisten. (SIMD)
   (b) Vergleich der modifizierten Dimensionsliste aus 1a mit der unteren Grenze. (SIMD)
   (c) Vergleich der modifizierten Dimensionsliste aus 1a mit der oberen Grenze. (SIMD)
   (d) Bildung des Schnitts der beiden Vergleichsergebnisse aus 1b und 1c. (SIMD)
   (e) „Horizontalisieren“ des Ergebnisses aus 1d.
2. Bestimmen der Endmarkierung
   (a) Vergleich der unmodifizierten Dimensionsliste mit 2^{31}. (SIMD)
   (b) „Horizontalisieren“ des Ergebnisses aus 2a.
   (c) Bestimmen des ersten gesetzten Bits aus 2b.
3. Durchlaufen des Ergebnisses aus 1e bis Bit aus 2c erreicht ist.

Punkt 1 ist in Abbildung 3.8 und Punkt 2 in Abbildung 3.9 visualisiert – jeweils ohne die Horizontalisierung.
Mit „Horizontalisieren“ ist das Zurückführen des Vektors in eine skalare Form gemeint. Dazu wird das SIMD-Register in ein Integer umgewandelt. Aus dem Ergebnisvektor [`false`,`true`,`true`,`true`] aus Abbildung 3.8 wird so $(1110)_2 = 14$. Es liegt nah, dass eine solche Operation möglichst selten und wenn notwendig zum spätmöglichsten Zeitpunkt in einer vektorbasierten Umsetzung vorkommen soll, weil sie dem Sinn von SIMD widerspricht, auf Vektoren in ihrer Gänze zu arbeiten. Durch das gesetzte MSB beziehungsweise die Prüfung dessen entstehen die Punkte 1a und 2 als vermeidbarer Mehraufwand. Vor allem Punkt 2 sticht hier durch das früh notwendige Horizontalisieren hervor.

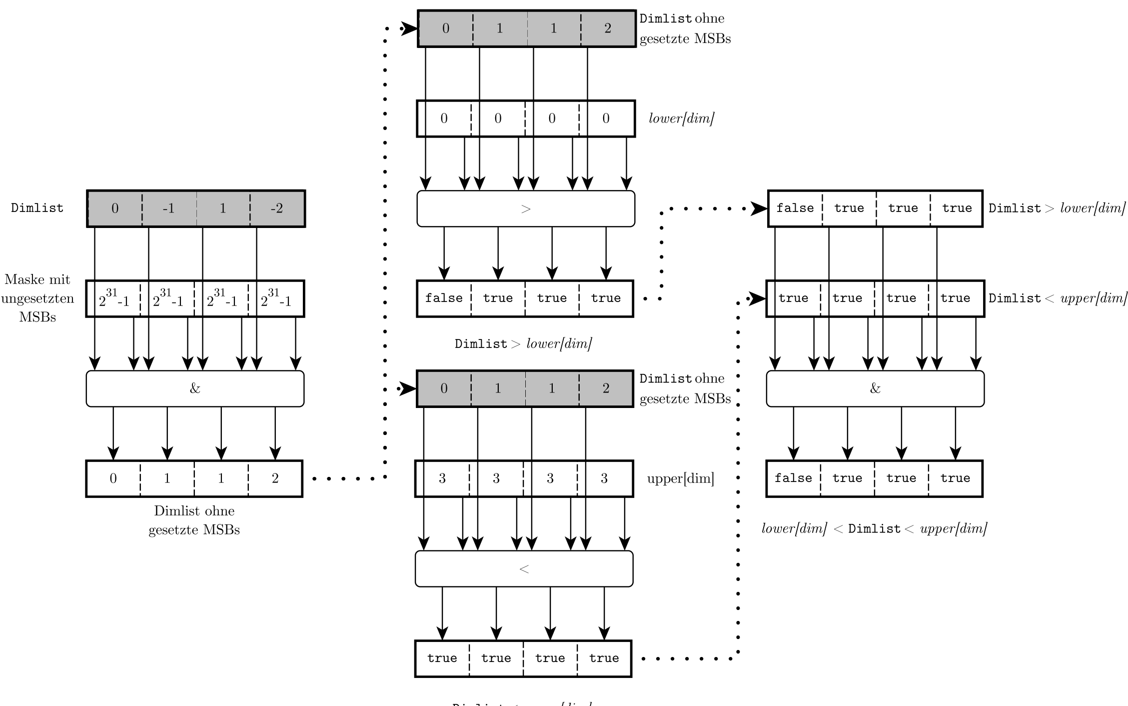


Abbildung 3.8: Prüfung einer Dimensionsliste gegen ein offenes Intervall mit SIMD bei Nutzung der impliziten Dimensionslistenlänge über das MSB

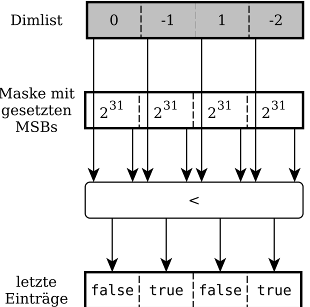


Abbildung 3.9: Ermittlung des Dimensionslistenendes mit SIMD bei Nutzung der impliziten Dimensionslistenlänge über das MSB

Das Endergebnis, der „Treffer“ 1, wird erst nach insgesamt 9 Operationen erreicht, wovon nur 5 von SIMD-Registern profitieren und davon 2 nur wegen der Besonderheit des MSBs überhaupt durchgeführt werden müssen.

Wie SIMD von der expliziten Speicherung der Länge profitieren kann, soll im Folgenden erläutert werden. Dazu wird zunächst beschrieben, an welchen Stellen die Längen im Elf zu finden sind.

Lokation der Länge im Elf. Um die Längen der Dimensionslisten aus dem Elf bei den Scan-Algorithmen auslesen zu können, ist verständlicherweise erneut eine

Adaptation des Aufbaualgorithmus und der Struktur des Elf notwendig.
Die Länge der regulären Dimensionslisten, das heißt mit Ausnahme derer der ersten Dimension und der `Monolisten`, wird beim Erstellen des Elf im Array `Elf` zu Beginn einer jeden solchen Liste gestellt. Das hat zur Folge, dass die Zeiger aus `Child_Pointers` nun nicht mehr auf den ersten Wert der nächsten Dimensionsliste referenzieren, sondern auf deren Länge. Außerdem entstehen so in `Child_Pointers` und `Cutoff_Pointers` Lücken an den Stellen, an denen in `Elf` die Länge gespeichert ist, weil sich gegen eine Auslagerung der Längen in ein weiteres Array entschieden worden ist. Grund ist hierfür, dass kein erkennbarer Mehrwert bei einer separaten Speicherung gegeben ist – man substituiert so lediglich die Leerstellen in `Child_Pointers` mit Zeigern (Integern) und schafft einen weiteren Sprung im Speicher.
Der bekannte Beispiel-Elf aus diesem Kapitel ist mit gespeicherten Dimensionslistenlängen (unterstrichen hervorgehoben) in Abbildung 3.10 zu finden.

`Elf`	0	1	2	3	4	5	6	7	8	9
`Elf`[00]	(1) 2	0	1	(2) 2	1	2	(6) 1	1	(7) 3	0
`Elf`[10]	1	2								

`Child_Pointers`	0	1	2	3	4	5	6	7	8	9
`Ch_P`[00]	[00]	[06]		[03]	-[04]		-[00]	-[02]		[08]
`Ch_P`[10]		-[07]	-[09]	-[11]						

`Monolists`	0	1	2	3	4	5	6	7	8	9
`ML`[00]	(3) 1	-T4	(4) 2	-T2	(5) 1	1	-T6	(8) 0	-T1	(9) 2
`ML`[10]	-T3	(10) 2	-T5							

`Cutoff_Pointers`	0	1	2	3	4	5	6	7	8	9
`CO_P`[00]	[00]	[03]		[00]	[02]		[00]	[01]		[03]
`CO_P`[10]		[03]	[04]	[05]						

`Elf_TIDs`	0	1	2	3	4	5	6	7	8	9
`E_T`[00]	T4	T2	T6	T1	T3	T5				

Abbildung 3.10: Elf aus Abbildung 3.6 mit gespeicherten Dimensionslistenlängen statt gesetzten MSBs

Durch das Hinzukommen der expliziten Längen nimmt der vereinnahmte Speicherplatz des Elf bedingt durch die Vergrößerung der betroffenen Arrays, in jedem Fall zu – das ist als Nachteil gegenüber der MSB-Implementierung zu sehen.
Dadurch müssen auch die Formeln für das Worst-Case-Szenario der Einträge in den Arrays `Elf`, `Child_Pointers` und `Cutoff_Pointers` angepasst werden. Für die letzten beiden kommen keine tatsächlich neuen Einträge hinzu, jedoch müssen die Leerbereiche bei einer Allokation ebenso mit berücksichtigt werden.
In der aufgezeigten Sonderkonstellation der Daten kommen pro Dimension mit Ausnahme der ersten $|R|/2$-Einträge für die Speicherung der Längen hinzu. Die Längen werden von Dimension 2 bis einschließlich zur vorletzten, $|DIM| - 1$, immer den Wert 1 und in der letzten Dimension den Wert 2 annehmen.

- $max\{|\texttt{Elf}|\} = |R| + \frac{|R|}{2} \cdot (|DIM| - 2) + \frac{|R|}{2} \cdot (|DIM| - 1) = |R| \cdot (|DIM| - \frac{1}{2})$
- $max\{|\texttt{Child_Pointers}|\} = |R| + \frac{|R|}{2} \cdot (|DIM| - 1) + \frac{|R|}{2} \cdot (|DIM| - 1) = |R| \cdot |DIM|$
- $max\{|\texttt{Cutoff_Pointer}|\} = max\{|\texttt{Child_Pointers}|\}$

An der Formel für `Monolists` ändert sich nichts, da sich an deren Worst-Case-Szenario nichts ändert und die Längen in `Monolists` nicht explizit gespeichert werden. Die letzte `TID` wird weiterhin über das MSB markiert, weil die `TID`s der `Monolisten` von SIMD ohnehin nicht berücksichtigt werden.
Für `Elf_TIDs` bleibt ebenso die bisherige Formel bestehen, da sich mit Hinzunahme der Längen für dieses Array nichts ändert.

Anpassung innerhalb der Scan-Algorithmen. Grundlegend müssen die Dimensionslisten-Scan-Algorithmen bezüglich ihrer verwendeten Schleife und deren Abbruchkriteriums angepasst werden. Ein Dimensionslisten-Scan wird spätestens abgebrochen, wenn der letzte Eintrag erreicht worden ist. Mit der MSB-Prüfung kann dies wie in `ScanDimlist` (Algorithmus 2) ersichtlich wird, in Form einer `while`-Schleife erfolgen, die solange ausgeführt wird bis das MSB des betrachteten Eintrags gesetzt ist. Durch die explizite Längenspeicherung steht nun die Anzahl der Iterationen und damit das Abbruchkriterium im Voraus fest, sodass eine klassische `for`-Schleife Anwendung finden kann. Das konkrete Vorgehen wird im weiteren Verlauf beim Vorstellen der SIMD-Scan-Algorithmen erkennbar.
Mit dem Wissen, wie die Dimensionslistenlänge im Elf gespeichert und wie diese innerhalb des Dimensionslisten-Scans prinzipiell verwendet wird, sollen jetzt das zuvor benannte Anfragebeispiel und die Schritte, die zur Bearbeitung dieser mithilfe der impliziten Länge über das MSB notwendig waren (siehe Abbildung 3.8), durch Ausnutzung der expliziten Längen neu betrachtet werden.

SIMD bei Umsetzung über explizite Länge. Im Vergleich zur MSB-Variante ergeben sich folgende grundlegende Schritte beim Abgleich einer Dimensionsliste gegen ein offenes Intervall:

1. Vergleich der Dimensionsliste mit der unteren Grenze. (SIMD)
2. Vergleich der Dimensionsliste mit der oberen Grenze. (SIMD)
3. Bildung des Schnitts der beiden Vergleichsergebnisse aus 1 und 2. (SIMD)
4. Bildung des Schnitts aus 3 mit Maske, die mit n `true`-Einträgen ab Vektorstelle 1 versehen ist, wobei n der Länge der Dimensionsliste (2) entspricht. (SIMD)
5. „Horizontalisieren“ des Ergebnisses aus 4.
6. Durchlaufen aller gesetzten Bits aus 5.

Der Unterschied zur MSB-Variante wird zum einen in der geringen Gesamtanzahl der Schritte als auch im höheren Anteil der vektoriellen SIMD-Operationen deutlich. Zudem kommt der entscheidende Vorteil hinzu, dass lediglich einmal ein Register nach Abschluss der SIMD-Operationen horizontalisiert werden muss. Grafisch ist diese Schrittfolge in Abbildung 3.11 dargestellt. Hier wird statt der Dimensionsliste $[0, -1, 1, -2]$ wie in Abbildung 3.8 jetzt $[2, 0, 1, 2]$ in die SIMD-Register geladen. Die Werte mit der Ausprägung „2“ sind nun jeweils Längen von Dimensionslisten.

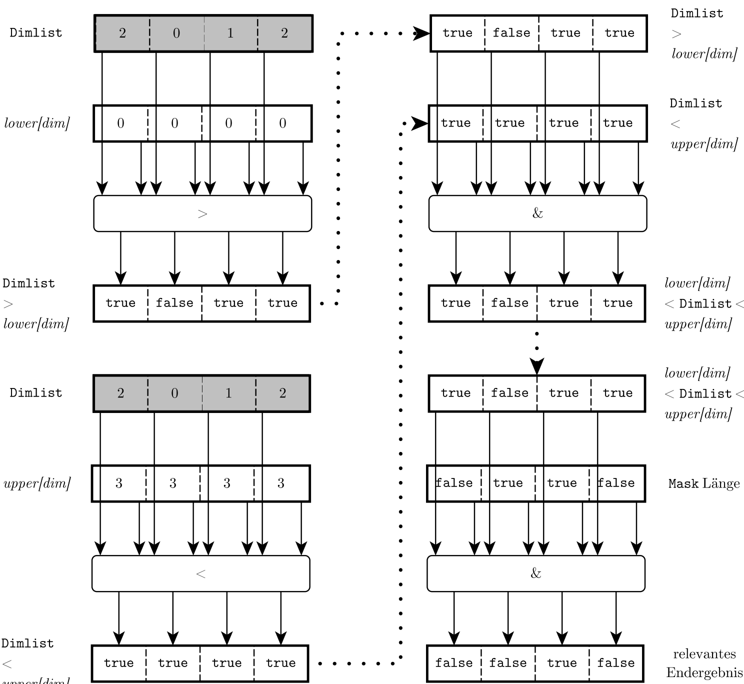


Abbildung 3.11: Prüfung einer Dimensionsliste gegen ein offenes Intervall mit SIMD bei Nutzung der expliziten Dimensionslistenlänge

3.2.3 Anpassungen der Scan-Algorithmen im Elf für SIMD

Durch den Umbau des Elf als eine „Structure of Arrays“ und der expliziten Speicherung der Dimensionslistenlängen wurde die Basis für eine effiziente Verwendung von SIMD geschaffen. Nun sollen die Scan-Algorithmen auf eine vektorbasierte Vorgehensweise angepasst werden. Dabei wurde sich auf die Algorithmen zur Bearbeitung partieller (Bereichs-)Anfragen konzentriert, weil mit dieser Anfrageart alle restlichen abbildbar sind (siehe Abschnitt 2.1.1).

3.2.3.1 Abkehr von abgeschlossenen hin zu offenen Intervallen bei Bereichsanfragen

In den sequentiellen Umsetzungen (Algorithmen 1 bis 5) wurden die Intervalle einer partiellen Bereichsanfrage als abgeschlossen betrachtet. Zur Prüfung, ob Werte innerhalb eines solchen Fensters liegen, wurde die Funktion `isIn` genutzt, welche für ihre drei Parameter v, l und u untersucht, ob $l \leq v \leq u$ gilt. Da die Operationen „$\leq$“ und „$\geq$“ keine SIMD-Äquivalente für die in dieser Arbeit verwendete Registergröße von 256-Bit besitzen, müssen sie adäquat substituiert werden. Dazu können sie auf die für `SIMD`-Register vorhandenen „$<$“- respektive „$>$“-Operationen abgebildet werden, indem die Beschaffenheit ganzer Zahlen ausgenutzt wird: $l \leq v \leq u \Leftrightarrow (l-1) < v < (u+1)$, für $l, u, v \in \mathbb{Z}$. Das bedeutet, dass in der Implementierung für SIMD die unteren Grenzen um 1 verringert und die oberen Grenzen um 1 erhöht werden mussten, um die abgeschlossenen Intervalle durch offene abzulösen. In Tabelle 3.3 sind die Transformationen möglicher Anfrageprädikate zu offenen Intervallen zu finden (vgl. Tabelle 2.1).

Prädikat	Intervallrepräsentation
$= a$	$(a-1, a+1)$
$< a$	$(min-1, a)$
$\leq a$	$(min-1, a+1)$
$> a$	$(a, max+1)$
$\geq a$	$(a-1, max+1)$
$\geq a \land \leq b, a \leq b$	$(a-1, b+1)$

Tabelle 3.3: Ganzzahlige Prädikate und Repräsentation als offene Intervalle (vgl. Tabelle 2.1)

In der Implementierung zieht diese Anpassung auch eine notwendige Änderung der verwendeten Datentypen mit sich. Bislang konnten und wurden die unteren Bereichsgrenzen als nichtnegative ganze Zahlen ($l \in \mathbb{N}_0$), also vorzeichenlose (`unsigned`) Integer aufgefasst. Hat die untere Grenze zuvor 0 angenommen, bedeutet das für die soeben erläuterte, neue, SIMD-angepasste Implementierung, nun den Wert -1, welcher keinen zulässigen `unsigned` Integer darstellt. Daher mussten die unteren Grenzen auf vorzeichenbehaftete (`signed`) Integer abgeändert werden.

Der Überlauffall auf der „gegenüberliegenden“ Seite, wenn die obere Grenze zuvor im abgeschlossenen Fall $2^{32}-1$ (Maximalwert von `unsigned` Integer) betragen hätte und darauf nun 1 addiert werden müsste, wurde nicht betrachtet. Grund ist hierfür, dass dieser Umstand für den verwendeten `TPC-H`-Benchmark keine Relevanz besitzt. Darüber hinaus waren alle Werte von 2^{31} bis $2^{32}-1$ aufgrund der impliziten Speicherung der Längen über das MSB in der vorherigen Elf-Implementierung ohnehin nicht abbildbar.

Mit der Verwendung von offenen anstelle von geschlossenen Bereichsdefinitionen wurde die letzte grundlegende Vorkehrung zur Nutzung von SIMD für die Scan-Algorithmen erklärt. Im Folgenden sollen die konkreten Modifikationen der einzelnen Algorithmen im Detail betrachtet werden. Den Anfang soll dabei `ScanMonolistSIMD` als datenparallele Entsprechung von `ScanMonolist` (Algorithmus 3) machen.

3.2.3.2 Datenparalleler `ScanMonolist`-Algorithmus

Das prinzipielle Verfahren, um mit einem vektorbasierten Ansatz festzustellen, ob sich die am Ende der `Monolist` befindlichen `TID`s für die Ergebnismenge qualifizieren, lässt sich in vier Schritte einteilen:

1. Laden der `Monolist` sowie der dazugehörigen unteren und oberen Grenzen als Vektoren in die SIMD-Register.
2. Vergleich mit den Grenzen.
3. Herausfiltern von Ergebnissen im Vektor, ...
   (a) ... zu denen keine Selektionsbedingung existiert.
   (b) ... die nach dem Ende der `Monolist` liegen.
4. Abbrechen, sobald eine Selektionsbedingung verletzt worden ist; ansonsten Wiederholen der Punkte 1 bis 3 bis alle Werte der `Monolist` verarbeitet worden sind.

Wie dieser Ablauf tatsächlich implementiert worden ist, soll nun Schritt für Schritt anhand des Algorithmus 6 erläutert werden.

Initialisierung

In den Zeilen `1` bis `3` werden die Vorbereitungen zum Vergleich der `Monolist`en-Werte mit den Grenzen innerhalb der „Hauptschleife“ (Zeilen `4` bis `21`) geschaffen. Die Größe $monolistSize$ der zu betrachtenden `Monolist` lässt sich anhand der Gesamtanzahl der Dimensionen, $|DIM|$, und des Dimensionsparameters dim bestimmen – befindet man sich beispielsweise in der zweiten Dimension von insgesamt sechs, besitzt die `Monolist` fünf Einträge.
Während der Index der `Monolist`, $monolistIndex$, bei 0 beginnt, beginnt der der Dimensionen, $dimIndex$, bei der aktuellen Dimension dim. $dimIndex$ wird als Positionsbestimmung für den Beginn des Zugriffs auf die Arrays der unteren ($lower[]$)

und oberen (*upper*[]) Grenzen sowie der selektierten Spalten (*selDims*[]) benötigt.

Da in ein SIMD-Register gegebenenfalls mehr Werte geladen werden, als dass sie in der `Monolist` vorhanden sind, diese aber das Ergebnis des Vergleichs mit den Grenzen verfälschen könnten, müssen die Werte nach dem `Monolist`-Ende ausgeblendet werden. So könnte beispielsweise eine Selektionsbedingung von einem Wert einer anschließenden `Monolist` verletzt werden und die aktuelle `Monolist` wäre ohne Maskierung, das heißt mit Berücksichtigung dieses Negativergebnisses, fälschlicherweise disqualifiziert. Zum Filtern dessen wird daher in Zeile `3` die Anzahl der relevanten Einträge im Register ab Position 0 bestimmt: *maskSize*. Diese ist abhängig von zwei Größen: der (verbleibenden) Länge der `Monolist` und der global geltenden, hardwarearchitekturabhängigen `SIMD_LINE_SIZE` (kurz SLS). Letztere gibt an, wie viele 32-Bit-Integer-Werte in ein SIMD-Register geladen werden können. Es ist naheliegend, dass *maskSize* mindestens so groß wie *monolistSize*, maximal jedoch so groß wie SLS werden kann.

Laden in die SIMD-Register

Nach der geleisteten Vorarbeit mit der initialen Setzung der genannten Variablen, gilt es nun, die relevanten Vektoren in SLS-großen Sprüngen iterativ zu laden und zu verarbeiten.

Das Laden der Vektoren in die SIMD-Register findet in den Zeilen `7` bis `10` statt. Dazu wird die Funktion `loadVecIntoSIMD` genutzt. Diese lädt, beginnend mit der übergebenen Adresse zu einer Position im jeweiligen Array, SLS viele Werte in ein SIMD-Register. Ist die Startposition beispielsweise 0 und ein SIMD-Register kann bis zu acht Werte fassen, wird das Register mit den Werten aus dem Array von Position 0 bis 7 befüllt. Für die `Monolist` und die Grenzvektoren ist dieser Ladevorgang in den Zeilen `8` bis `10` ersichtlich. Hier wird die Startposition zum Laden aus `Monolists` und *lower*/*upper* über die Schleifenvariablen *monolistIndex* respektive *dimIndex* gesteuert.

Der Ladevorgang für den Bitvektor der selektierten Spalten ist dahingegen ein anderer. Grund ist hierfür die Tatsache, dass sowohl die Grenzen als auch die `Monolist`-Einträge beliebig sein können, wohingegen die selektierten Spalten durch ihren binären Charakter (Spalten können „aus- und eingeschalten" werden) abzählbar viele Kombinationsmöglichkeiten besitzen – nämlich $2^{|DIM|}$. Diese Anzahl lässt sich auf 2^{SLS} weiter reduzieren, weil für einen Schleifendurchlauf die Selektion von maximal SLS vielen Spalten wesentlich ist. Deshalb wurde sich bei der Implementierung dafür entschieden, alle möglichen Kombinationen von selektierten Spalten als ein vordefiniertes Array von SIMD-Registern zu speichern (`SEL_DIMS_SIMD`). An der Stelle 0 besitzt dieses Array das SIMD-Register, welches die Konstellation repräsentiert, dass auf *keiner* Spalte eine Selektion existiert, während an der Stelle $2^{\text{SLS}}-1$ Selbiges für Selektionskriterien auf *jeder* Spalte lokalisiert ist. Für SLS $= 8$ ist ein solches Maskenarray in Tabelle 3.4 zu finden.

Zur Bestimmung, welches dieser Register für die aktuelle Schleifeniteration das richtige ist, wird der Index des Registerarrays als *curSelDims* bestimmt. Dazu wird in Zeile `5` die Zahlenrepräsentation des Bitvektors (Funktion `getLong`) bitweise nach rechts um *dimIndex* Stellen verschoben wird und per bitweisen Unds mit $2^{\text{SLS}}-1$ zum Beschränken der Länge verknüpft wird.

```
Procedure ScanMonolistSIMD(lower[], upper[], beginPointer, dim,
                            selDims[], L):
 1    monolistIndex ← 0; monolistSize ← |DIM| − dim;
 2    dimIndex ← dim;
 3    maskSize ← min{monolistSize, SIMD_LINE_SIZE};
 4    while monolistIndex < monolistSize do
          // in Iteration relevanten sel. Spalten ermitteln
 5        curSelDims ← getLong(selDims) ≫ dimIndex & 2^SIMD_LINE_SIZE;
          // Vergleich nur ausführen, wenn überhaupt Spalten selektiert
 6        if curSelDims > 0 then
              // relevante sel. Spalten als SIMD-Vektor laden
 7            selDimsSIMD ←
               loadVecIntoSIMD(SEL_DIMS_SIMD[curSelDims − 1]);
              // Werte der Monolist als SIMD-Vektor
 8            monolistSIMD ← loadVecIntoSIMD(Monolists[beginPointer+
                                                  monolistIndex]);
              // SIMD-Vektoren mit unteren und oberen Grenzen vergleichen
 9            lowerBoundsSIMD ← loadVecIntoSIMD(lower[dimIndex]);
10            upperBoundsSIMD ← loadVecIntoSIMD(upper[dimIndex]);
              // Vergleich der Intervalle mit Monolist
11            compSIMD ←
               andSIMD(cmpgtSIMD(monolistSIMD, lowerBoundsSIMD),
                       cmpgtSIMD(upperBoundsSIMD, monolistSIMD));
              // Herausfiltern relevanter Ergebnisse anhand sel. Spalten
12            compSIMD ←
               xorSIMD(andNotSIMD(compSIMD, selDimsSIMD),
                       TRUE_MASKS_SIMD[maskSize − 1]);
              // Ausblenden der Werte, die nach dem Monolist-Ende liegen
13            compSIMD ←
               xorSIMD(andSIMD(compSIMD,TRUE_MASKS_SIMD[maskSize − 1]),
                       TRUE_MASKS_SIMD[maskSize − 1]);
              // alle Werte im Register müssen nun wahr sein, sonst
                 Monolist disqualifiziert
14            if not testZSIMD(compSIMD, compSIMD) then
15                return;
16            end
17        end
          // Inkrementation der Schleifenvariablen
18        monolistIndex ← monolistIndex + SIMD_LINE_SIZE;
19        dimIndex ← dimIndex + SIMD_LINE_SIZE;
20        maskSize ← min{monolistSize, SIMD_LINE_SIZE};
21    end
      // qualifizierte TIDs in L aufnehmen - analog seq. Algorithmus
22 end
```

Algorithmus 6 : Scan von Monolisten unter Zuhilfenahme von SIMD

Dieses Vorgehen soll an dieser Stelle anhand eines Beispiels unter folgenden Bedingungen erklärt werden:

- es gibt zehn Dimensionen in der Anfrage, $|DIM| = 10$
- der Selektionsvektor hat die Ausprägung $selDims = [1, 0, 0, 1, 1, 1, 0, 0, 0, 1]$
- die maximale SIMD-Registergröße fässt acht 32-Bit-Integer, SLS $= 8$
- man befindet sich in Dimension zwei, $dimIndex = 1$

Unter diesen Annahmen müsste *selDims* ab Position 1 bis Position 8 im weiteren Verlauf in ein SIMD-Register geladen werden – also $[0, 0, 1, 1, 1, 0, 0, 0]$. Dieser Vektor befindet sich im beschriebenen Registerarray an Position 28. Auf 28 kommt man, indem man das Bitarray $[0, 0, 1, 1, 1, 0, 0, 0]$ als die Folge von Bits einer Binärzahl auffässt und diese in eine Dezimalzahl überführt. Im Algorithmus wird diese logische Herleitung eben über das bitweise Verschieben und „Verunden“ gelöst: $\texttt{getLong}(selDims) = 1000111001_2 = 569 \Rightarrow 569 \gg 1 = 284 = 100011100_2 \Rightarrow 284 \,\&\, 255 = 28 = 00011100_2$.
In Zeile `6` wird überprüft, ob überhaupt auf irgendeiner der betrachteten Dimensionen eine Selektion vorliegt. Nur wenn das der Fall ist, müssen Selektionsbedingungen überprüft werden. Sonst kann direkt mit der Inkrementation der Schleifenvariablen für den nächsten Durchlauf fortgefahren werden (Zeilen `18` bis `20`).

Index	Einträge im SIMD-Register							
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
128	0	1	1	1	1	1	1	1
129	1	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
254	1	1	1	1	1	1	1	0
255	1	1	1	1	1	1	1	1

Tabelle 3.4: SIMD-Registerarray `SEL_DIMS_SIMD` zur Speicherung von Kombinationsmöglichkeiten von Spaltenselektionen (SLS$= 8$)

Vergleich der `Monolist` mit den Grenzen

Im Anschluss an das Laden der relevanten Vektoren, gilt es zunächst, den Vergleich der `Monolist`-Werte mit allen unteren und oberen Grenzwerten durchzuführen. Dazu wird in Zeile `11` mit der SIMD-Operation `cmpgtSIMD` der Größer-als-Vergleich

jeweils mit den unteren und oberen Grenzen durchgeführt und die Ergebnisse daraus anschließend per SIMD-Konjunktion (`andSIMD`) zusammengefügt.
Hierbei können zwei Arten von Werten mit in den Vergleich einbezogen werden, die nicht dazu beisteuern dürfen, dass sich das Tupel der `Monolist` disqualifiziert. Das sind zum einen Werte zu Dimensionen, auf denen keine Selektion vorliegt und zum anderen Werte, die nach dem Ende der `Monolist` liegen.

Ausblenden irrelevanter Ergebnisse

Der Ausschluss von Vergleichsergebnissen zu nicht selektierten Spalten kann über das Abbilden einer Implikation erfolgen: wenn ein Attribut selektiert ist, es sich also um keine `Platzhalterspalte` handelt, muss die Selektionsbedingung für dieses erfüllt sein – der Wert der `Monolist` muss im vorgegebenen Intervall liegen. Mathematisch kann man es als $selDims[i] \rightarrow comp[i]$ definieren, wobei i eine Dimension ist und $comp[i]$ dem Vergleichsergebnis des `Monolisten`-Wertes dieser Dimension mit den dazugehörigen Grenzen entspricht. Für die SIMD-Umsetzung besteht an dieser Stelle erneut die Herausforderungen, eine passende Umformung für eine Operation, hier die Implikation, zu finden, die als solche nicht nativ unterstützt wird.
Die Implikation lässt sich alleinig mit den vorhanden SIMD-Operationen `andNotSIMD`[1] und `xorSIMD` umsetzen (siehe Zeile `12`), weil Folgendes gilt: $selDims[i] \rightarrow comp[i] \Leftrightarrow \neg selDims[i] \vee comp[i] \Leftrightarrow \neg(selDims[i] \wedge \neg comp[i]) \Leftrightarrow (selDims[i] \wedge \neg comp[i]) \oplus$ `true`. Im Kontext von SIMD wird `true` als ein Vektor aufgefasst werden, der an jeder Stelle den Wert `true` annimmt. Innerhalb eines Schleifendurchlaufs reicht es allerdings aus, wenn der `true`-Vektor von den Stellen 0 bis $maskSize - 1$ den Wert `true` besitzt, da nur die Werte bis $maskSize - 1$ relevant sind. Dieser spezielle Vektor wir in allen SIMD-Algorithmen benötigt und soll im Folgenden TMSI (kurz für ***t**rue **m**ask **S**IMD of the current **i**teration*) genannt werden.
Wie zuvor bei der Initialisierung von $maskSize$ beschrieben, kann $maskSize$ maximal so groß wie SLS sein. Daher wird zur Abbildung aller denkbaren TMSIs ähnlich wie bei den Möglichkeiten der selektierten Spalten vorgegangen: es werden insgesamt SLS viele SIMD-Register-Masken vorab in einem Array, genannt `TRUE_MASKS_SIMD`, gespeichert. Für SLS $= 8$ befindet sich an Stelle 0 das SIMD-Register zum Vektor $[0, 0, 0, 0, 0, 0, 0, 1]$ und an Stelle 7 der Vektor $[1, 1, 1, 1, 1, 1, 1, 1]$. Fasst man diese Vektoren wieder als Bits von Binärzahlen auf, dann steht an Stelle i des Arrays $(1 \ll (i + 1)) - 1$ für $0 \leq i <$ SLS.
Im Anschluss an das gerade beschriebene Herausfiltern von `Platzhalterspalten` müssen die Ergebnisse zu Werten nach dem letzten Eintrag der `Monolist` von der Betrachtung ausgeschlossen werden. In Zeile `13` soll genau dies gemacht werden: es wird unter anderem zwischen dem bisherigen Vergleichsergebnis $compSIMD$ und TMSI eine Konjunktion via `andSIMD` gebildet. Zusätzlich wird in dieser Zeile erneut `xorSIMD` angewandt. Die Bedeutung dessen soll nun erläutert werden.

(Nicht-)Aufnahme von `TIDs` in die Ergebnismenge

Nach den Maskierungen sollten alle Elemente im SIMD-Register die Ausprägung `true` besitzen, sofern sich das Tupel der `Monolist` für die Ergebnismenge noch qua-

[1] Das Äquivalent aus der Intel-Intrinsik `_mm256_andnot_si256`(V_1, V_2) mit den Parametern berechnet etwas kontraintuitiv $\neg V_1 \wedge V_2$; um der Implementierung zu folgen, soll `andNot` in den Algorithmen dieses Kapitels auch so aufgefasst werden

lifiziert. Es gibt jedoch keine `SIMD`-Operation, um zu festzustellen, ob *ausnahmslos jedes* Element des Vektors `true` ist. Währenddessen existiert mit `testZSIMD` eine Funktion, um zu prüfen, ob jedes Element *nicht* auf `true` gesetzt ist. Es ist offensichtlich, dass, wenn jeder Eintrag des Vektors `true` ist, sein Komplement in jedem Eintrag den Wert `false` annimmt. Folglich ist die Idee, zu prüfen, ob der Komplementvektor, welcher über das angesprochene `xorSIMD` mit TMSI in Zeile `13` bestimmt werden kann, ausschließlich den Wert `false` besitzt. Ist dem so, gibt `testZSIMD` wahr zurück und es können in den Zeilen `18` bis `20` die Werte für die nächste Iteration gesetzt werden. Ansonsten kann die Schleife abgebrochen werden und die `TID`s am Ende der `Monolist` werden nicht in die Ergebnismenge aufgenommen.
Das Hinzufügen der `TID`s zu der Ergebnismenge verläuft identisch zum sequentiellen Algorithmus `ScanMonolist` (Algorithmus 3).

Beispielhafter Aufruf

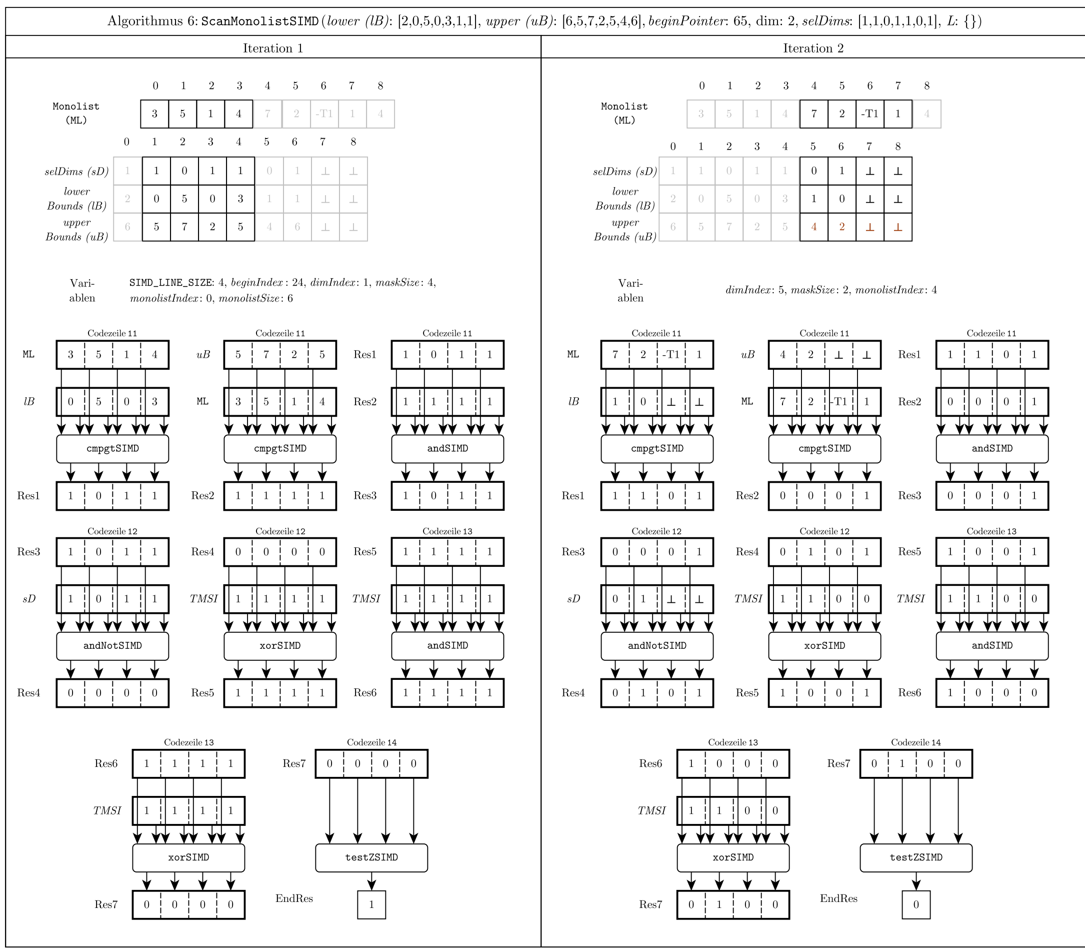


Abbildung 3.12: Beispielhafter Aufruf von `ScanMonolistSIMD` (Algorithmus 6)

Ein konkretes Beispiel für den Ablauf eines Aufrufs von `ScanMonolistSIMD` ist in Abbildung 3.13 zu finden. Es wird für die Übersichtlichkeit der Darstellung von einer maximalen SIMD-Registergröße von vier 32-Bit-Integerwerten ausgegangen und `true` und `false` innerhalb der Register werden durch „1“ respektive „0“ ersetzt. Die

Codepassagen der einzelnen Operationen können über die über den Operationen stehenden Codezeilen, die sich auf Algorithmus 6 beziehen, identifiziert werden. Das Beispiel ist so gewählt, dass die `Monolist` ihrer Länge wegen auf zwei Iterationen aufgeteilt werden muss, wobei in der ersten Iteration alle Bedingungen erfüllt sind, in der zweiten jedoch ein Ausschluss der `TID`s erfolgt. Grund für das Exkludieren ist, dass Element 5 der `Monolist` nicht im offenen Intervall $(0, 2)$ liegt.
Das Falsum ($\perp$) soll einen undefinierten Wert widerspiegeln. Dieser kommt zustande, weil lediglich sieben Dimensionen vorliegen, in die SIMD-Register aber unabhängig davon immer vier Werte geladen, wodurch in der zweiten Iteration bei *selDims*, *lowerBounds* und *upperBounds* zwei undefinierte Werte mitgeladen werden. Zur Demonstration der Sinnhaftigkeit aller Schritte, insbesondere derer zum Ausblenden von irrelevanten Einträgen, sind die Ergebnisse von Operationen mit $\perp$ willkürlich gewählt.

3.2.3.3 Datenparalleler `ScanDimlist`-Algorithmus

Aufgrund der Intradimensionalität der Einträge von regulären Dimensionslisten unterscheidet sich der datenparallele Scan-Algorithmus zum Verarbeiten von Dimensionslisten, genannt `ScanDimlistSIMD`, in einigen Punkten vom just beschriebenen `ScanMonolistSIMD`-Algorithmus. `ScanDimlistSIMD` lässt sich grob in sieben Schritte gliedern:

1. Einmaliges Laden der unteren und oberen Dimensionsgrenze in alle Einträge von je einem SIMD-Register.
2. Laden des relevanten Dimensionslistenabschnitts als Vektor in ein SIMD-Register.
3. Vergleich des Dimensionslistenabschnitts mit den Grenzen.
4. Herausfiltern von Ergebnissen im Vektor aus Punkt 3, die nach dem Ende der Dimensionsliste liegen.
5. „Horizontalisieren" des Ergebnisses aus Punkt 4.
6. Durchlaufen aller gesetzten Bits aus Punkt 5 zum rekursiven Aufruf von `ScanDimlistSIMD`.
7. Abbrechen, sobald ein Wert außerhalb des Fensters ist; ansonsten Wiederholen der Punkte 2 bis 6 bis alle Werte der Dimensionsliste verarbeitet worden sind.

Die konkrete Implementierung von `ScanDimlistSIMD` ist in Algorithmus 7 erkennbar. Eine nähere Beschreibung dessen soll nun erfolgen.
Die angesprochene zugrundeliegende Intradimensionalität hat im Wesentlichen zwei Vereinfachungen zur Folge. Auf der einen Seite gibt es nur jeweils eine zu berücksichtigende untere und obere Grenze. Bei diesen genügt es, sie einmal zu Beginn des Algorithmus in die SIMD-Register zu laden. Auf der anderen Seite muss der *selDims*-Vektor nie in ein SIMD-Register geladen werden und folglich müssen keine Operationen mit ihm erfolgen. Mit *selDims* muss lediglich geprüft werden, ob man die Intervallprüfung durchführen und damit SIMD ausnutzen kann (Zeile `1`). Liegt eine `Platzhalterspalte` vor, folgt wie im sequentiellen Fall ein rekursiver Aufruf von `ScanDimlistSIMD` für alle Einträge der aktuellen Dimensionsliste.

```
Procedure ScanDimlistSIMD(lower[], upper[], beginPointer, dim, selDims[], L):
1    if selDims[dim] then
         // Dimensionslistenlänge an Stelle bP, erster Wert bei bP + 1
2        beginIndex ← beginPointer + 1 ;
3        dimlistIndex ← 0; dimlistSize ← Elf[beginPointer];
4        maskSize ← min{dimlistSize, SIMD_LINE_SIZE};
         // untere/obere Grenze der Dim. in Register laden
5        lowerBoundSIMD ← loadIntIntoSIMD(lower[dimIndex]);
6        upperBoundSIMD ← loadIntIntoSIMD(upper[dimIndex]);
7        matched ← false;
8        while dimlistIndex < dimlistSize do
             // Laden der Dimensionsliste
9            dimlistSIMD ← loadVecIntoSIMD(Elf[beginIndex + dimlistIndex]);
             // Vergleich mit oberer Grenze ausreichend, falls Treffer zuvor
10           if matched then
11               compSIMD ←cmpgtSIMD(upperBoundSIMD, dimlistSIMD);
12           else
                 // sonst Vergleich mit unterer und oberer Grenze
13               compSIMD ←
                   andSIMD(cmpgtSIMD(dimlistSIMD, lowerBoundSIMD),
                           cmpgtSIMD(upperBoundSIMD, dimlistSIMD));
14           end
             // Ausblenden von Werten nach dem Dimensionslistenende
15           compSIMD ←andSIMD(compSIMD, TRUE_MASKS_SIMD[maskSize − 1]);
             // prüfen, ob mind. ein Wert im Intervall liegt
16           if testZSIMD(compSIMD, compSIMD) then
                 // kein Treffer akt. Iteration, aber zuvor → abbrechen
17               if matched then
18                   return;
19               else
20                   continue;
21               end
22           end
23           matched ← true;
24           comp ← moveMaskSIMD(compSIMD); // Horizontalisieren
             // Anz. Treffer (gesetzte Bits) und 1. Treffer (1. gesetztes Bit) erm.
25           setBits ← popCount(comp); firstSetBit ← countTrailingZeros(comp);
26           for i ← 0 to setBits − 1 do
27               pointer ← Child_Pointers[beginIndex + dimlistIndex+
                                          firstSetBit + i];
                 // rekursiver Aufruf von ScanMonolistSIMD/ScanDimlistSIMD
28           end
29           if i + firstSetBit ≠ SIMD_LINE_SIZE then
30               break;
31           end
32           dimlistIndex ← dimlistIndex + SIMD_LINE_SIZE;
33           maskSize ← min{dimlistSize − dimlistIndex, SIMD_LINE_SIZE};
34       end
35   else
         // keine Selektion auf akt. Dimension, alle Werte Betrachten
36   end
37 end
```

Algorithmus 7 : Scan von Dimensionslisten unter Zuhilfenahme von SIMD

Initialisierung

Wie auch zu Beginn von `ScanMonolistSIMD` (Algorithmus 6) gilt es zunächst, in den Zeilen `2` bis `4` von Algorithmus 7 Variablen für den Schleifendurchlauf zu initialisieren. Die Länge der aktuellen Dimensionsliste, $dimlistSize$, lässt sich aufgrund der vorliegenden Struktur des Elf an der Startposition ablesen, die über den Parameter $beginPointer$ mitgeteilt wird. Sie wird zum Schleifenabbruch und für die Bestimmung der bereits bekannten `true`-Masken, den TMSIs, benötigt. Die Speicherung der Länge hat zur Folge, dass die Position des ersten Dimensionslisteneintrags aus $beginIndex = beginPointer + 1$ berechnet werden muss.
Ebenfalls neu ist das Laden der Grenzwerte, welches nun über `loadIntIntoSIMD` vollzogen wird. Diese Funktion schreibt den übergebenen Integer, hier die Ausprägungen von *lower* und *upper* an der Position der aktuellen Dimension *dim*, in alle Einträge eines SIMD-Registers (Zeilen `5` und `6`).
Den Abschluss der Variableninitialisierung bildet Zeile `7`. Hier wird *matched* auf `false` gesetzt. *matched* soll speichern, ob in einer Iteration mindestens ein Dimensionslistenwert innerhalb der Grenzen gefunden worden ist.

Pruning beim Vergleich der Dimensionsliste mit den Grenzen

Nach der Initialisierung der notwendigen Variablen folgt eine Schleife (Zeile `8` bis `34`), die die Dimensionsliste SIMD-„getreu“ in SLS-großen Abschnitten durchläuft und die für die weiteren Aufrufe entscheidenden Vergleiche durchführt. Dazu wird zu Beginn der Vektor mit der vorweg eingeführten Funktion `loadVecIntoSIMD` entsprechend der aktuellen Iteration, von der Stelle $beginIndex + dimlistIndex$ anfangend, geladen und anschließend mit dem Dimensionsintervall verglichen. Kam es in einer vorherigen Iteration noch zu keinem Treffer, ist man also noch auf der Suche nach dem ersten Wert, der sich im Fenster befindet, muss das Dimensionslistenregister sowohl mit der oberen als auch mit der unteren Grenze über `cmpgtSIMD` verglichen werden. Per `andSIMD` wird danach der Schnitt der beiden Vergleichsregister gebildet. Gab es indes zuvor bereits einen Treffer, lässt sich die aufsteigende Sortierung der Liste zum *Pruning* ausnutzen; der erste Wert innerhalb des Fensters wurde nämlich bereits in einem vorherigen Schleifendurchlauf entdeckt, sodass alle anschließenden Einträge auf jeden Fall größer als die untere Grenze sind und die Schleife nun nur noch bis zum letzten Wert fortgeführt werden muss, der kleiner als die obere Grenze ist (Zeilen `10` bis `14`).
Nach dem schon bekannten Ausblenden von Werten nach einem Listenende (Zeile `15`) folgt, konträr zu `ScanMonolistSIMD`, die Prüfung, ob mindestens ein Treffer innerhalb des Vergleichsregisters $compSIMD$ vorliegt. Eine Negativprüfung dessen wird mit `testZSIMD` in Zeile `16` bewerkstelligt. Auch an dieser Stelle ist entscheidend, ob *matched* in einem vorangegangenen Durchlauf auf `true` gesetzt worden ist. Wenn es bereits einen Treffer gab, aber im aktuellen Durchlauf keinen, kann die Funktion abgebrochen werden – es wird definitiv kein Wert folgen, der noch innerhalb des Intervalls liegt (Zeile `18`). Ist *matched* dahingegen noch `false`, muss die Suche im nächsten Fenster fortgesetzt werden (Zeile `20`).

Horizontalisierung zum Ermitteln rekursiver Aufrufe

Wenn es in der aktuellen Iteration mindestens einen Treffer gibt, `testZSIMD` aus Zeile `16` also fehlschlägt, muss für alle Treffer ein rekursiver Aufruf von `ScanDimlistSIMD` erfolgen.
Zum Durchführen jener Rekursionen lässt sich ein „Horizontalisieren“ an dieser Stelle schwerlich vermeiden, weil nun einzelne Elemente betrachtet werden müssen. Der Befehl dazu ist `moveMaskSIMD` (Zeile `24`). Hieraus resultiert eine Zahl, bei der die Bits an den Stellen auf 1 gesetzt sind, an denen die Vergleiche positiv verliefen.
Wegen der aufsteigenden Sortierung und des Ausblendens der Werte nach dem Dimensionslistenende, sind, sofern es mehr als ein gesetztes Bit gibt, die „1“-en stets aufeinander folgend. Begründen lässt sich dies wie folgt: alle Werte, die die Bedingung erfüllen ($\in (lower[dim], upper[dim])$) folgen aufeinander und nach einem `false`-Eintrag kann wegen der Maskierung nie ein `true`-Eintrag im selben Register folgen. Der Vektor $[1, 1, 1, 0, 0, 1, 1, 0]$ ist beispielsweise keine mögliche Ausprägung von $compSIMD$, da die zwei `true`-Werte an den Positionen 6 und 7 nur zustande kommen können, wenn in das SIMD-Register Einträge aus einer anderen Dimensionsliste geladen worden sind. Diese sind aber eben durch TMSI ausgeblendet worden.
Es reicht demzufolge zur Bestimmung der anstehenden Rekursionen aus, zu ermitteln, an welcher Stelle das erste gesetzte Bit der transformierten Zahl *comp* zu finden ist und wie viele gesetzte Bits es insgesamt gibt. Das erste gesetzte Bit wird ermittelt, indem über die Funktion `countTrailingZeros` die Anzahl angehängter Nullen gezählt wird. Die Anzahl insgesamt gesetzter Bits wird mithilfe von `popCount` bestimmt (beides in Zeile `25`). Ist beispielsweise der SIMD-Vektor $[0, 0, 0, 1, 1, 1, 1, 0]$ für $SLS = 8$, wird von `moveMaskSIMD` die Zahl $120 = 1111000_2$ zurückgeliefert. `countTrailingZeros` liefert hierbei 3 und die Anzahl der gesetzten Bits über `popCount` ist 4. So ergeben sich im Zusammenspiel mit der Startposition $beginIndex$ und dem von der Iteration abhängigen Index der Dimensionsliste, $dimlistIndex$, die rekursiven Funktionsaufrufe (Zeile `27`).

Bevor die Schleifenvariablen für den nächsten Durchlauf in den Zeilen `32` und `33` angepasst werden, kann ein weiteres *Pruning* erfolgen. Entspricht, wie im eben angeführten Beispiel, das *letzte* gesetzte Bit nicht dem (relevanten) *Ende* des Ergebnisvektors, kann an dieser Stelle ein Abbruch erfolgen, weil kein Wert mehr folgen kann, der innerhalb des Selektionsfensters liegt (Zeilen `29` und `30`).

Beispielhafter Aufruf

In Abbildung 3.13 ist ein Beispiel für den Ablauf des Algorithmus 7 dargestellt. Man erkennt hier den Effekt des Prunings, welches basierend auf den Ergebnissen aus der ersten Iteration in der zweiten genutzt werden kann – statt der ursprünglichen fünf SIMD-Operationen im ersten Lauf sind danach nur noch drei notwendig. Darüber hinaus muss keine Iteration 3 gestartet werden, da der letzte Treffer, der Eintrag mit dem Wert 12, nicht das Ende der Dimensionsliste darstellt. Die Rekursionen müssen für die Werte 6, 10 und 12 erfolgen.

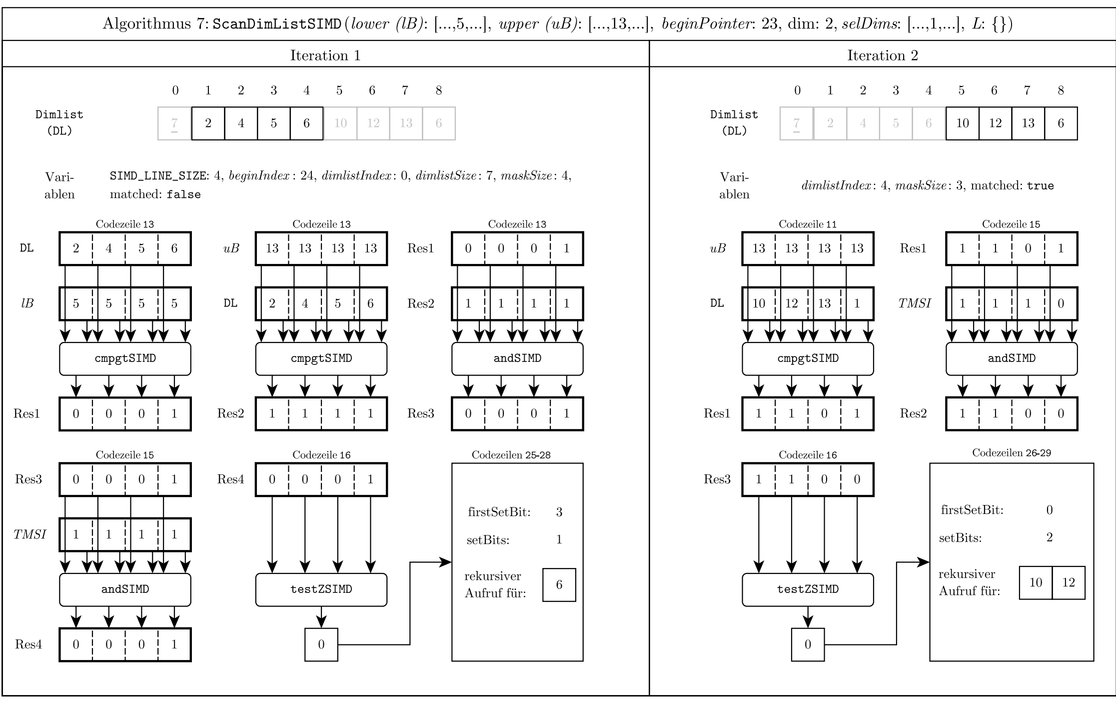


Abbildung 3.13: Beispielhafter Aufruf von `ScanDimlistSIMD`

3.2.3.4 Datenparalleler `ScanDimlistCutoffs`-Algorithmus

Der letzte Algorithmus, bei dem in dieser Arbeit eine SIMD-basierte Vorgehensweise implementiert worden ist, ist der Scan von Dimensionslisten unter Zuhilfenahme von `Cutoffs` – `ScanDimlistCutoffsSIMD`. Da bei der Nutzung von `Cutoffs` ebenso ein Abgleich von intradimensionalen Dimensionslistenwerten mit je einer unteren und oberen Grenze erforderlich ist, wäre es möglich, dieselbe Schrittfolge zu nutzen wie sie für `ScanDimlistSIMD` vorgestellt worden ist. Es gibt jedoch einen entscheidenden Unterschied: während man sich bei `ScanDimlistSIMD` für alle Werte interessiert, die in das Selektionsfenster fallen, weil für diese eine Rekursion veranlasst werden muss, werden bei der Verwendung von `Cutoffs` ausschließlich der erste und letzte Dimensionslistenwert innerhalb des Intervalls benötigt. Es ergibt sich folgende Vorgehensweise:

1. Einmaliges Laden der unteren Grenze in ein SIMD-Register.
2. Laden des relevanten Dimensionslistenabschnitts als Vektor in ein SIMD-Register.
3. Vergleich der Dimensionslistenwerte mit der unteren Grenze.
4. Abbrechen, sobald ein Wert gefunden ist, der größer als die untere Grenze ist und dessen Position speichern; ansonsten Wiederholen der Punkte 2 und 3 bis alle Werte der Dimensionsliste durchlaufen worden sind.
5. Liefert Punkt 4 ein Ergebnis, dann...
   (a) Einmaliges Laden der oberen Grenze in ein SIMD-Register.

(b) Laden des relevanten Dimensionslistenabschnitts als Vektor in ein SIMD-Register beginnend ab der Position aus Punkt 4.

(c) Vergleich der Dimensionslistenwerte mit der oberen Grenze.

(d) Bestimmen der Endposition als das letzte gesetzte Bit aus dem Ergebnisregister aus Punkt 5c.

(e) Wiederholen der Punkte 5b bis 5d solange wie in Punkt 5c mindestens ein Treffer vorliegt.

6. `TID`-Positionen aus `Cutoff_Pointers` zwischen den Ergebnissen aus den Punkten 4 und 5 auslesen.

Die Details der Implementierung sind in Algorithmus 8 auffindbar und sollen jetzt näher beschrieben werden.

Initialisierung

Analog zum sequentiellen Algorithmus (Algorithmus 5) ist die Grundvoraussetzung, dass die `Cutoffs` genutzt werden können, dass die aktuelle Dimension *dim* die letztselektierte (Parameter *lastSelDim*) ist (Zeile `1`). Die anschließende Initialisierung der notwendigen Variablen zum Durchlaufen der Dimensionslisten in Vektormanier ist in den Zeilen `2` bis `4` identisch zu `ScanDimlistSIMD`. Ein Unterschied zum letztgenannten Algorithmus ist, dass zu Beginn lediglich die untere Grenze über `loadIntIntoSIMD` in ein SIMD-Register geladen wird (Zeile `5`). Des Weiteren werden die relevanten Endergebnisse, die in *indexLower* und *indexUpper* gespeichert werden sollen, zunächst mit einem undefinierten Wert beschrieben. Bleibt mindestens eine der beiden Variablen im weiteren Verlauf bis zum Abschluss der Schleifen undefiniert, bedeutet das, dass kein Dimensionslistenwert zwischen der unteren und oberen Grenze liegt.

Finden des ersten Wertes innerhalb der Grenzen

In der `while`-Schleife von Zeile `7` bis `17` wird nach dem ersten Wert gesucht, der innerhalb des Intervalls liegt. Dazu wird der relevante Dimensionslistenabschnitt als Vektor via `loadVecIntoSIMD` in ein SIMD-Register kopiert (Zeile `8`) und anschließend mit der unteren Grenze verglichen (Zeile `9`). In derselben Zeile werden zudem wie auch in den zuvor vorgestellten Algorithmen 6 und 7 die unwesentlichen, das Ergebnis eventuell verfälschenden Einträge über die bekannte Maske TMSI per Schnittbildung herausgefiltert. Sofern es mindestens einen Wert gibt, der im Register *compSIMD* auf `true` steht, kann die Suche eingestellt werden. Die Prüfung dessen erfolgt wie üblich über `testZSIMD` (Zeile `10`). Falls dieser Test negativ ausfällt, muss *compSIMD* per `moveMaskSIMD` horizontalisiert werden, damit über `countTrailingZeros` die Position des ersten Treffers (*indexLower*) bestimmt werden kann (Zeile `11` und `12`).

```
Procedure ScanDimlistCutoffsSIMD(lower[], upper[], beginPointer, dim, selDims[], L, lastSelDim,
                                 parentTIDPointerEnd):
1   if dim = lastSelDim then
        // akt. Dimension ist letztselektierte → Cutoffs nutzen
        // Dimensionslistenlänge an Stelle beginPointer, erster Wert an beginPointer + 1
2       beginIndex ← beginPointer + 1 ;
3       dimlistIndex ← 0; dimlistSize ← Elf[beginPointer];
4       maskSize ← min{dimlistSize, SIMD_LINE_SIZE};
        // untere Grenze der Dimension in Register laden
5       lowerBoundSIMD ← loadIntIntoSIMD(lower[dimIndex]);
6       (indexLower, indexUpper, startOffsetUpper) ← N_DEF;
        // ersten Wert innerhalb der Grenzen finden
7       while dimlistIndex < dimlistSize do
            // Laden Dimensionsliste
8           dimlistSIMD ← loadVecIntoSIMD(Elf[beginIndex + dimlistIndex]);
            // Vergleich mit unterer Grenze; Ausschließen Werte nach Dimensionslistenende
9           compSIMD ←
              andSIMD(cmpgtSIMD(dimlistSIMD, lowerBoundSIMD), TRUE_MASKS_SIMD[maskSize − 1]));
            // bei Treffer ist erster Wert innerhalb Grenze gefunden → Schleife abbrechen
10          if not testZSIMD(compSIMD, compSIMD) then
                // Start der Suche nach letztem Wert innerhalb der Grenzen um Position des Treffers
                   versetzen
11              startOffsetUpper ← dimlistIndex + countTrailingZeros(moveMaskSIMD(compSIMD));
12              indexLower ← beginIndex + startOffsetUpper;
13              break;
14          end
15          dimlistIndex ← dimlistIndex + SIMD_LINE_SIZE;
16          maskSize ← min{dimlistSize − dimlistIndex, SIMD_LINE_SIZE};
17      end
        // Suche nach letztem Wert innerhalb Grenzen; nur, wenn erster Wert gefunden
18      if startOffsetUpper ≠ N_DEF then
19          dimlistIndex ← startOffsetUpper;
            // oberer Grenze der Dimension in Register laden
20          upperBoundSIMD ← loadIntIntoSIMD(upper[dimIndex]);
21          maskSize ← min{dimlistSize − dimlistIndex, SIMD_LINE_SIZE};
22          while dimlistIndex < dimlistSize do
                // Laden Dimensionsliste
23              dimlistSIMD ← loadVecIntoSIMD(Elf[beginIndex + dimlistIndex]);
                // Vergleich mit oberer Grenze; Ausschließen Werte nach Dimensionslistenende
24              compSIMD ←
                  andSIMD(cmpgtSIMD(upperBoundSIMD, dimlistSIMD),
                          TRUE_MASKS_SIMD[maskSize − 1]));
                // kein einziger Treffer → vorher gesetzter letzter Wert innerhalb Grenzen bleibt
                   bestehen
25              if testZSIMD(compSIMD, compSIMD) then
26                  break;
27              else
                    // letzter Wert innerhalb Grenzen = letzter Treffer im Register
28                  setBits ← popCount(moveMaskSIMD(compSIMD));
29                  indexUpper ← beginIndex + dimlistIndex + setBits;
30                  if setBits ≠ maskSize then
31                      break;
32                  end
33              end
34              dimlistIndex ← dimlistIndex + SIMD_LINE_SIZE;
35              maskSize ← min{dimlistSize − dimlistIndex, SIMD_LINE_SIZE};
36          end
37      end
        // wenn unterer und oberer Wert innerhalb der Grenze gefunden, Cutoffs nutzen
38      if N_DEF ∉ {indexLower, indexUpper} then
            // wenn oberer Wert das Dimensionslistenende ist, gilt Endzeiger des Elter
39          if indexUpper = (beginIndex + dimlistSize) then
40              L = L ∪ {Elf_TID[i] | C_O[indexLower] ≤ i < parentTIDPointerEnd};
41          else
42              L = L ∪ {Elf_TID[i] | C_O[indexLower] ≤ i < C_O[indexUpper]};
43          end
44      end
45  else
        // keine Cutoffs nutzbar → jeweils tidPointerEnd bestimmen und verfahren wie bei
           ScanDimlistSIMD
46  end
47 end
```

Algorithmus 8 : Scan von Dimensionslisten unter Zuhilfenahme von SIMD mit `Cutoffs`

Die Variable $startOffsetUpper$ soll als Form des *Prunings* dazu dienen, dass die anschließende Schleife zur Ermittlung des letzten Wertes im Selektionsfenster nicht erneut am Beginn der Dimensionsliste mit der Suche einsetzt, sondern erst ab einschließlich der Position, an der der erste Wert im Intervall gefunden worden ist. Ist kein Eintrag im Vergleichsregister auf `true` gesetzt, wird die aktuelle Schleife fortgesetzt – dazu werden in den Zeilen `15` und `16` die entsprechenden Variablen für die nächste Iteration angepasst.

Finden des letzten Wertes innerhalb der Grenzen

Die zweite `while`-Schleife (Zeilen `22 - 36`) zur Bestimmung des letzten Wertes innerhalb der Grenzen, muss nur dann ausgeführt werden, wenn in der ersten Schleife ein Treffer vorlag. Festgestellt wird das in Zeile `18`, indem $startOffsetUpper$ auf Definiertheit geprüft wird. Ist diese Variable gesetzt, kann die obere Grenze der Dimension in ein SIMD-Register geladen (Zeile `20`) und die Suche nach dem letzten Wert im Intervall begonnen werden. Dafür wird der Vergleich dieses Mal mit der oberen Grenze durchgeführt, wobei das Maskieren nicht benötigter Elemente bestehen bleibt. Anders als bei der ersten Schleife führt ein vollständig auf `false` stehendes SIMD-Register $compSIMD$ direkt zum Abbruch der Schleife (Zeile `25`). Inhaltlich bedeutet das nämlich, dass kein einziger Dimensionswert im aktuell betrachteten Vektor kleiner als die obere Grenze ist – und wegen der aufsteigenden Reihenfolge der Werte wäre dies auch so für folgende Schleifendurchläufe. Schlägt `testZSIMD` allerdings fehl, muss das relevante Ergebnis $indexUpper$ auf den Eintrag im Register gesetzt werden, der als letztes auf `true` steht (Zeilen `28` und `29`). Wie in den Erläuterungen zu `ScanDimlistSIMD` (Algorithmus 7) bereits beschrieben, muss der letzte `true`-Eintrag dabei – wegen der Sortiertheit und Filterung des Ergebnisses – der Anzahl aller `true`-Einträge verringert um 1 entsprechen[1]. Das *Pruning* in Zeile `30` folgt dem gleichen Muster wie in `ScanDimlistSIMD`: entspricht $indexUpper$ nicht der Position des letzten relevanten, also eingeblendeten, Wertes (+1), kann kein Eintrag, der auf $indexUpper$ folgt noch innerhalb des Intervalls liegen.
Deckungsgleich zum sequentiellen Algorithmus lässt sich bei gesetztem $indexLower$ und $indexUpper$ die Ergebnismenge um die Elemente aus `Elf_TID` erweitern, deren Indexe zwischen `Cutoff_Pointers`[$indexLower$] und `Cutoff_Pointers`[$indexUpper$] liegen. Auch hier gibt es wieder ein gesondertes Vorgehen, wenn $indexUpper$ den letzten Eintrag der Dimensionsliste darstellt. Dann bildet der Parameter $parentTID$-$PointerEnd$ den Endindex aus `Elf_TID` (Zeilen `39` bis `43`).

Beispielhafter Aufruf

Ähnlich zu den anderen beiden SIMD-Algorithmen soll die Abbildung Abbildung 3.14 den Ablauf eines vorstellbaren Aufrufs von `ScanDimlistCutoffsSIMD` widerspiegeln. In Iteration 1 wird hier der erste Wert gefunden, der innerhalb des offenen Intervalls $(4, 14)$ liegt. Mit dieser Position (4) kann in Iteration 2 sogleich mit dem Vergleich mit der oberen Grenze begonnen werden. Da alle betrachteten Werte echt kleiner als die obere Grenze sind, benötigt es eine dritte Iteration, in welcher der finale Wert innerhalb des Fensters gefunden wird.

[1] Der Definition von `Cutoffs` folgend, benötigt man den Nachfolger des letzten Wertes im Intervall, weshalb im Algorithmus eine Verringerung um 1 nicht zu finden ist.

Anhand der Ergebnisse aus Iteration 1 und 3 (*indexLower* und *indexUpper*) lassen sich dann die `TIDs` für die Lösungsmenge L auslesen.

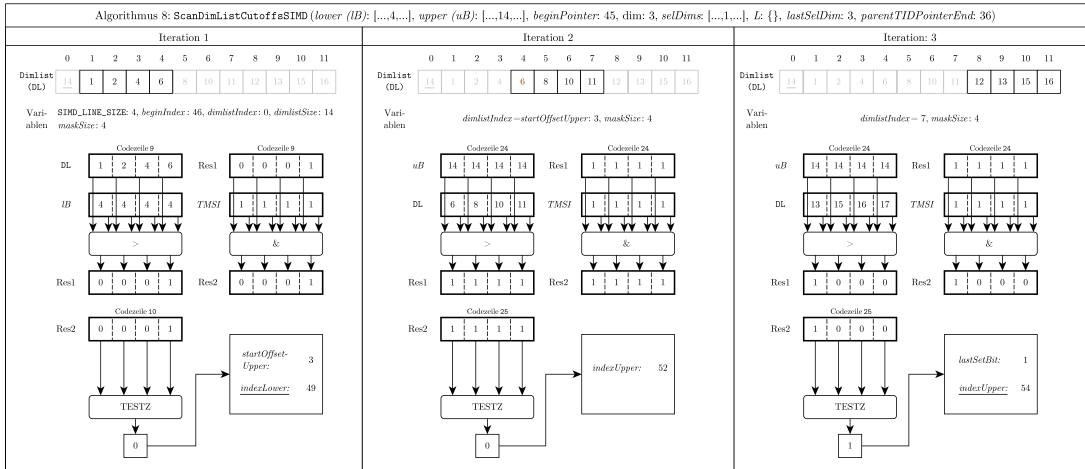


Abbildung 3.14: Beispielhafter Aufruf von `ScanDimlistCutoffsSIMD`

3.3 Zusammenfassung

Während des Konzeptionierungs- und Implementierungskapitels wurden die zwei Hauptaspekte dieser Arbeit zusammengebracht: die Datenparallelität durch SIMD und die Scan-Algorithmen der multidimensionalen Indexstruktur Elf.
Zu Beginn wurde zu diesem Zweck die programmatische Nutzung von SIMD beleuchtet. Dabei wurden allgemein auftretende Herausforderungen erläutert und deren Relevanz für die Besonderheiten im Elf aufgezeigt. Daran anknüpfend wurden die existierenden Techniken zum Einsatz von SIMD in (bestehendem) Programmcode für die vektorbasierte, parallele Verarbeitung von Daten vorgestellt und erörtert. Als Folge der Auseinandersetzung mit zwei Implementierungsmöglichkeiten wurde die Wahl zur Verwendung der Intrinsiken für diese Arbeit begründet.
Im Anschluss wurden die Einsatzgebiete von SIMD konkret für die Scan-Algorithmen innerhalb des Elf betrachtet. Dazu wurden sich anbietende Stellen der Parallelisierung durch SIMD herausgearbeitet.
Bevor die algorithmischen Anpassungen durchgeführt werden konnten, galt es, Änderungen im Aufbau des Elf vorzunehmen. Der Elf als „Structure of Arrays“ wurde ebenso wie die explizite Speicherung der Dimensionslistenlängen motiviert. Diese zwei Anpassungen sind als Antworten auf Punkt (a) der ersten wissenschaftliche Fragestellung **FF** 1. zu betrachten, da der Elf in seiner Ursprungsfassung ohne diese strukturellen Modifikationen nicht für die Verwendung SIMD ausgelegt war.
Den Abschluss bildete dann die detaillierte Beschreibung sämtlicher Modifikationen am Programmcode der Scan-Algorithmen. Diese wurden sowohl im Pseudocode als auch anhand von Schaubildern im Ablauf vorgestellt und stellen die Beantwortung von Punkt (b) der ersten Forschungsfrage **FF** 1. dar.

4. Evaluierung

In Kapitel 3 wurde vorgestellt, wie sich zwischen den beiden Schwerpunkten dieser Arbeit, der multidimensionalen Indexstruktur Elf und SIMD eine Schnittmenge bilden lässt. Nun soll es darum gehen, zu untersuchen, ob sich die Überlegungen bezüglich der Voraussetzungen für SIMD und der letztendlichen Umsetzung als sinnvoll beziehungsweise vorteilhaft hinsichtlich der Performanz bei den Scan-Algorithmen bewahrheiten. Während der Hauptaugenmerk dabei bei den Laufzeiten der Anfragebearbeitungen liegen soll, werden außerdem mögliche Trade-offs in Bezug auf die Erstellzeiten und den vereinnahmten Speicherplatz bei den neuen Implementierungen analysiert.
Bevor die konkreten Experimente beschrieben, analysiert und interpretiert werden, soll vorangestellt die Testumgebung, der Versuchsaufbau und die allgemeine Vorgehensweise erläutert werden.

4.1 Aufbau der Experimente

Zur Schaffung einer einheitlichen Vergleichsbasis der zu evaluierenden Implementierungen bedarf es der Verwendung einer konsistent arbeitenden Testumgebung. Damit ist sowohl der softwareseitige Rahmen gemeint, in dem die verschiedenen Ansätze selbst sowie eine objektive Möglichkeit der Gegenüberstellung dieser umgesetzt sind, als auch die zugrundeliegende Hardwarearchitektur. Gerade Letztere ist, wie im vorangegangenen Kapitel deutlich geworden ist, von essentieller Bedeutung für die Verwendbarkeit von SIMD.

4.1.1 Hardwareumgebung

Die Ausführung der Experimente erfolgte auf einer Maschine mit einer AVX2-unterstützenden Intel-CPU (Xeon E5-2630 v3), die auf 2,4 GHz getaktet ist und auf 8 Kerne mit 16 Threads zurückgreifen kann.
Mit AVX2 stehen SIMD-Register mit einer Breite von insgesamt 256 Bit zur Verfügung. So ist es möglich, acht 32-Bit-große Integer auf einmal vektorbasiert zu bearbeiten.

Bezeichnung in Kapitel 3	Implementierung	Beschreibung
`loadVecIntoSIMD`	`_mm256_loadu_si256`	Lädt 256-Bits von Integern aus dem Speicher in ein SIMD-Register
`loadIntIntoSIMD`	`_mm256_set1_epi32`	Schreibt einen 32-Bit-Integer in alle Elemente eines SIMD-Registers
`cmpgtSIMD`	`_mm256_cmpgt_epi32`	Berechnet den Größer-als-Vergleich der 32-Bit-Integer Werte zweier SIMD-Vektoren und speichert das Ergebnis in ein neues SIMD-Register
`andSIMD`	`_mm256_and_si256`	Berechnet das bitweise UND der 256-Integer-Bits von zwei SIMD-Registern und speichert das Ergebnis in ein neues SIMD-Register
`andNotSIMD`	`_mm256_andnot_si256`	Berechnet die Negation des ersten SIMD-Registers, konjugiert dies mit dem zweiten und speichert das Ergebnis in ein neues SIMD-Register
`xorSIMD`	`_mm256_xor_si256`	Berechnet das bitweise XOR der 256-Integer-Bits von zwei SIMD-Registern und speichert das Ergebnis in ein neues SIMD-Register
`testZSIMD`	`_mm256_testz_si256`	Berechnet das bitweise UND der 256-Integer-Bits von zwei SIMD-Registern, gibt 1 zurück, wenn das Ergebnis in allen Fällen 0 ist
`movemaskSIMD`	`_mm256_movemask_ps`	Gibt einen 32-Bit-Integer zurück, dessen Bit an der Stelle i dem MSB des i-ten Elements eines SIMD-Registers entspricht
`popCount`	`_mm_popcnt_u32`	Zählt die Anzahl auf 1 gesetzter Bits in einem vorzeichenlosen Integer
`countTrailingZeros`	`__builtin_ctz`	Zählt die Anzahl angehängter 0-Bits, beginnend mit dem *least significant bit*

Tabelle 4.1: Tatsächlich zur Implementierung zur Verfügung stehende, hardwareabhängige SIMD-Funktionen im Vergleich zur Bezeichnung im Pseudcode der Algorithmen aus Abschnitt 3.2.3

In Abschnitt 3.2.3 wurden in den Algorithmen allgemeine Bezeichnungen für die letztendlich verwendeten, eng von der Harware abhängigen SIMD-Intrinsikfunktionen genutzt. In Tabelle 4.1 sind diese Funktionen konkret benannt und ihrer bisherigen Bezeichnung im Konzeptionierungskapitel gegenübergestellt.
Die Entwicklung fand auf einem linuxbasierten System statt, wobei der GNU-Compiler in der Version 4.8.5 genutzt worden ist. Dadurch stand die compilerspezifische Methode `__builtin_ctz` zur Verfügung.
Während `_mm_popcnt_u32` bereits mit der SIMD-Iteration SSE 4.2 vorhanden war, bedingen alle restlichen Funktionen AVX2.

Sowohl die zu indexierenden Tabellen selbst als auch die erstellten Elfs werden für die Experimente im RAM gehalten. Die verwendete Maschine besitzt Arbeitsspeicher mit einer maximalen Kapazität von einem Terabyte.

4.1.2 Vorgehensweise bei der Evaluation

Die Umsetzung des SIMD-Scans für den Elf, im weiteren Verlauf `Elf_SIMD` genannt, erfolgte analog zur bereits bestehenden Implementierung in C++ im Kontext der soeben beschriebenen Hardware.

Vor dem Vergleich der Ansätze hinsichtlich ihrer Ausführungszeiten von Anfragen musste zunächst die Korrektheit der Arbeitsweisen der Algorithmen und damit ihrer produzierten Ergebnisse sichergestellt werden.
Dazu wurden Anfragen auf Basis der TPC-`Lineitem`- und -`Part`-Tabellen generiert, die bezüglich ihrer selektierten Spalten (*selDims*) und der Intervalle *lower* und *upper* zufällig gewählt worden sind. Die Ergebnismengen der verschiedenen Elf-Ansätze, die durch die Ausführung der erzeugten Anfragen entstanden, wurden mit jener, die Resultat eines sequentiellen Relationenscans (*full table scan*) war, auf Identität abgeglichen.
Um eine fehlerhafte Generierung von randomisierten Anfragen auszuschließen, wurde außerdem dasselbe Vorgehen (der Ergebnisvergleich mit einem *full table scan*) für die im Folgenden beschriebenen TPC-H-Anfragen angewandt.

4.1.2.1 Evaluierte Anfragen

Wie auch in den bisherigen Untersuchungen zum Elf (vgl. [BKSS17]) wird für die hiesige Evaluation auf ausgewählte, leicht modifizierte Anfragen des TPC-H-Benchmarks zurückgegriffen. Dieser ist durch große Datenmengen aus dem Anwendungsgebiet der Entscheidungsunterstützungssysteme (engl. *decision support systems*) gekennzeichnet [Tra18].
Dabei werden grundlegend zwei Anfragearten unterschieden:

1. Anfragen mit einem Selektionsprädikat (eindimensionale Anfragen)

2. Anfragen mit mehreren Selektionsprädikaten (mehrdimensionale Anfragen)

Zwar liegt die Stärke des Elf aufgrund seiner Multidimensionalität in der Bearbeitung mehrdimensionaler Anfragen (Punkt 2), allerdings ist eine Untersuchung von

eindimensionalen Anfragen (Punkt 1) als vermeintliche Schwäche gerade deswegen nicht weniger interessant.
Konkret wurden die Anfragen `Q1`, `Q10` und `Q14` als eindimensionale und `Q6`, `Q17P` und `Q19`(`P`) als mehrdimensionale Vertreter ausgewählt. Dabei handelt es sich sowohl um reine partielle Anfragen (`Q17P`) als auch partielle Bereichsanfragen (zum Beispiel `Q14`). Ihre Selektivitäten σ, das heißt das Verhältnis der Datensätze, die die Selektionsprädikate erfüllen zu der Gesamtanzahl der Datensätze innerhalb der Relation, können der Tabelle 4.2 entnommen werden. Darüber hinaus wird hier ersichtlich, welche Spalten konkret aus den beiden TPC-H-Tabellen `Lineitem` (Präfix `L` in der Spaltenbezeichnung) und `Part` (Präfix `P` in der Spalten-, Suffix `P` bei der Anfragebezeichnung) selektiert werden und welche Position diese in der Elf-Indexstruktur einnehmen[1]. Letztere ist, wie in den bisherigen Analysen zum Elf gezeigt wurde, entscheidend für die Performanz bei der Anfragebearbeitung – insbesondere für eindimensionale Anfragen [BKSS17].

Anfrage	Selektivität σ	Prädikatspalten	Position der Spalte im Elf
`Q1`	98,0	`l_shipdate`	0
`Q10`	1,72	`l_returnflag`	4
`Q14`	1,3	`l_shipdate`	0
`Q6`	1,72	`l_shipdate`, `l_discount`, `l_quantity`	{0,1,2}
`Q17P`	0,099	`p_brand`, `p_container`	{1,2}
`Q19`	1,4	`l_quantity`, `l_shipinstr`, `l_shipmode`	{2,5,6}
`Q19P`	0,083	`p_brand`, `p_container`, `p_quantity`	{2,5,6}

Tabelle 4.2: Details zu den evaluierten TPC-H-Anfragen (aus [BKSS17])

Die konkreten SQL-Anweisungen zu den Anfragen `Q1`, `Q14`, `Q17P`, `Q19` und `Q19P` sind in den Codeauszüge 4.1 bis 4.5 wiederzufinden. `Q6` und `Q10` wurden bereits als einleitende Beispiele in Kapitel 2 aufgegriffen (Codeauszüge 2.2 und 2.3). Hierbei ist anzumerken, dass die Gruppierungen sowie die dazugehörigen Aggregationsfunktionen und Projektionen aus dem TPC-H-Benchmark bewusst weggelassen worden sind, da der Fokus von Indexstrukturen und damit auch des Elf in der Optimierung von Selektionen liegt. Des Weiteren wurden ausschließlich die Tabellen `Lineitem` und `Part` berücksichtigt und keine Verbunde zwischen diesen oder mit anderen Relationen betrachtet.

[1] Es wurden immer alle Spalten von `Lineitem` beziehungsweise `Part` durch den Elf indexiert.

```
SELECT *
  FROM Lineitem
 WHERE l_shipdate ≤ ‘1998-12-01’ − [DELTA] ‘Days’
```

Codeauszug 4.1: `Q1`

```
SELECT *
  FROM Lineitem
 WHERE l_shipdate ≥ [DATE]
   AND l_shipdate <
       [DATE] +‘1month’
```

Codeauszug 4.2: `Q14`

```
SELECT *
  FROM Part
 WHERE p_brand = [BRAND]
   AND p_container = [CONTAINER]
```

Codeauszug 4.3: `Q17P`

```
SELECT *
  FROM Lineitem
 WHERE (l_quantity ≥ [QUANT1] AND l_quantity ≤ [QUANT1] + 10
    OR  l_quantity ≥ [QUANT2] AND l_quantity ≤ [QUANT2] + 10
    OR  l_quantity ≥ [QUANT3] AND l_quantity ≤ [QUANT3] + 10)
   AND  l_shipmode IN (‘AIR’,‘AIR REG’)
   AND  l_shipinstr = ‘DELIVER IN PERSON’
```

Codeauszug 4.4: `Q19`

```
SELECT *
  FROM Part
 WHERE (p_brand = [BRAND1] AND Psize ≥ 1 AND Psize ≤ 5
   AND  p_container IN (‘SM CASE’,‘SM BOX’,‘SM PACK’,‘SM PKG’))
    OR (p_brand = [BRAND2] AND Psize ≥ 1 AND Psize ≤ 10
   AND  p_container IN (‘MED BAG’,‘MED BOX’,‘MED PACK’,‘MED PKG’))
    OR (p_brand = [BRAND3] AND Psize ≥ 1 AND Psize ≤ 15
   AND  p_container IN (‘LG CASE’,‘LG BOX’,‘LG PACK’,‘LG PKG’))
```

Codeauszug 4.5: `Q19P`

4.1.2.2 Evaluierte Elf-Varianten

Die vorgestellten TPC-H-Anfragen wurden in den Experimenten mit folgenden vier Implementierungsvarianten des Elf bearbeitet:

1. `Elf64`: `Elf64` stellt die Standardimplementierung des Elf inklusive der ursprünglichen Scan-Algorithmen `ScanFirstDimlist`(`Cutoffs`),

`ScanDimlist`(`Cutoffs`) und `ScanMonolist` dar. Für die regulären Dimensionslistenzeiger werden 64-Bit-Integer verwendetet, während alle anderen Eintragstypen im linearisierten Elf 32-Bit-Integer darstellen. Die Dimensionslistenzeiger sind nicht als direkte Zeiger sondern als Offsets umgesetzt. Die Dimensionslistenlänge wird implizit über das MSB gespeichert.

2. `Elf_Separated`: Die Erweiterung der Standardimplementierung des Elf um die Betrachtung des Elf als eine „Structure of Arrays“ wird `Elf_Separated` genannt (vgl. Abschnitt 3.2.2.1). Hier erfolgt eine Separierung der Eintragstypen in eigens für sie vorgesehene Arrays (`Elf`, `Child_Pointers`, `Monolists` (, `Elf_TIDs`,`Cutoff_Pointers`)).
3. `Elf_Separated_Length`: Basierend auf `Elf_Separated` speichert `Elf_Separated_Length` die expliziten Längen der Dimensionslisten im `Elf`-Array vor dem Beginn einer jeden Dimensionsliste (vgl. Abschnitt 3.2.2.2).
4. `Elf_SIMD`: Die SIMD-optimierte, das heißt datenparallele, vektorbasierte Implementierung der Scan-Algorithmen, die `Elf_Separted_Length` als „Unterbau“ nutzt, heißt `Elf_SIMD`. Die hier zur Anwendung kommenden Algorithmen `ScanDimlist`(`Cutoffs`)`SIMD` und `ScanMonolistSIMD` wurden ausführlich in Abschnitt 3.2.3 vorgestellt.

Anhand der Beschreibungen der Elf-Varianten lassen sich die Beweggründe für die Auswahl genau dieser zur Evaluation gut erkennen.
Während `Elf64` als die sequentielle Standardimplementierung des Elf den Ausgangspunkt darstellt, sind `Elf_Separated` und `Elf_Separated_Length` Zwischenschritte von `Elf64` hin zum Hauptuntersuchungsgegenstand dieser Arbeit, `Elf_SIMD`. Sie setzen nämlich genau die Punkte um, die in Abschnitt 3.2.2 als notwendige strukturelle Änderungen zur Ermöglichung einer effizienten, vektorbasierten Arbeitsweise mit SIMD genannt werden. So ist `Elf_Separated` die „Structure of Arrays“-Entsprechnung von `Elf64` und in `Elf_Separated_Length` werden auf Basis dieses Speicherlayouts zusätzlich im `Elf`-Array die Dimensionslistenlängen explizit gespeichert. `Elf_SIMD` ist mit `Elf_Separated_Length` hinsichtlich des zugrundeliegenden Aufbaus deckungsgleich und unterscheidet sich nur hinsichtlich der datenparallel umgesetzten Scan-Algorithmen.

Implementierungsvariante	Speicherlayout		Speicherung der Länge		Datenparallelität
	„AoS“	„SoA“	indirekt	direkt	
Elf64	✓	×	✓	×	×
Elf_Separated	×	✓	✓	×	×
Elf_Separated_Length	×	✓	×	✓	×
Elf_SIMD	×	✓	×	✓	✓

Tabelle 4.3: Gegenüberstellung der Unterschiede in den Implementierungen der evaluierten Elf-Varianten

Durch dieses schrittweise Vorgehen lassen sich die Wirkungen der einzelnen Maßnahmen auf die Performanz bei der Anfragebearbeitung feingranularer untersuchen als wenn lediglich `Elf_SIMD` mit `Elf64` verglichen werden würde. Die „reine“ Auswirkung von SIMD für die Scan-Algorithmen lässt sich demzufolge auch eher beim Vergleich von `Elf_Separated_Length` mit `Elf_SIMD` feststellen.
Eine auf die wesentlichen Unterscheidungsmerkmale heruntergebrochene Gegenüberstellung der Implementierungsvarianten ist in Tabelle 4.3 dargestellt.

4.1.2.3 Weitere Parameter

Die zugrundeliegenden TPC-H-Anfragen und ausgewählten Varianten der Implementierung des Elf für deren Bearbeitung sind die Hauptparameter der Evaluationen. Darüber hinaus gibt es jedoch noch weitere Parameter, die Einfluss auf die Experimente als solche sowie deren Aussagekraft genommen haben:

- Verwendung von `Cutoffs`
- Größe der TPC-H-Tabellen
- Anzahl der Wiederholungen und Testfälle der einzelnen Experimente

Verwendung von `Cutoffs`

Die in Abschnitt 2.2.3.2 vorgestellten `Cutoffs` sollen die Zeit verkürzen, die benötigt wird, um effektiv auf die für das Anfrageergebnis relevanten `TIDs` zugreifen zu können. Sie wirken sich also auf die Anfragezeit aus, weshalb die Untersuchungen sowohl den Fall der Ausnutzung von `Cutoffs` als auch den ohne beleuchten sollen.

Größe der TPC-H-Tabellen

Bei der Generierung des TPC-H-Benchmarks lässt sich spezifizieren, wie groß die gesamte Datenbank sein soll, die die betrachteten Tabellen `Lineitem` und `Part` inkludiert. Für diese Arbeit wurden die Größen 1GB, 50GB und 100GB verwendet.
Es wurde zudem nicht direkt auf den eigentlichen Daten `Lineitem`- und `Part`-Tabellen des Benchmarks gearbeitet, sondern diese über eine Wörterbuchkompression in ein Äquivalent mit ausschließlich Integerwerten überführt.

Anzahl der Wiederholungen und Testfälle der einzelnen Experimente

Zur Erhöhung der Aussagekraft und zur Mittlung der Ergebnisse der Experimente wurden diese je Testfall zehnmal wiederholt. Ein Testfall ist dabei eine Anfrage, bei denen die Parameter dieser (in eckigen Klammern zum Beispiel in Codeauszüge 4.1 bis 4.5 dargestellt) zufällig generiert worden sind. Es wurden je TPC-H-Anfrage 100 Testfälle generiert. Alle Implementierungsvarianten wurden denselben Testfällen unterzogen.

4.2 Experimente

Mit Kenntnis der Parameter lassen sich nun die verschiedenen Experimente zur Evaluation der beschriebenen neuen Elf-Implementierungsvarianten betrachten und deren Ergebnisse diskutieren.
Zunächst werden die „sekundären“ Untersuchungsgegenstände in Form der benötigten Erstellzeiten für die Indexstrukturen und des benötigten Speicherplatzes dieser im Vordergrund stehen, bevor die Auswirkung einer datenparallelen Verarbeitung mit SIMD auf die Scan-Algorithmen betrachtet wird.

4.2.1 Benötigter Speicherplatz

Der Standardelf `Elf64` wurde im Zuge der Implementierung für die Nutzung von SIMD bei den Scan-Algorithmen in zwei Punkten strukturell angepasst. Zum einen wurden die verschiedenen Eintragstypen des Elf in eigens für sie vorgesehene Arrays ausgelagert („Structure of Arrays“). Zum anderen wurden zusätzlich die Dimensionslistenlängen explizit gespeichert (vgl. Abschnitt 3.2.2). Daher ist es naheliegend, die konkreten Auswirkungen der Implementierungen auf den Speicherplatzbedarf zu analysieren. Gerade weil ein Vorteil des Elf die komprimierte Speicherung der Daten durch seine Präfixredundanzeliminierung ist (welche aber durch keine Variante verloren geht).

4.2.1.1 Ergebnisse

Der konkret benötigte Speicherplatz in Bytes kann den Abbildungen 4.1 und 4.2 entnommen werden. Hier wurden die Elfs von `Elf64`, `Elf_Separated` und `Elf_Separated_Length` für die `Lineitem`-Tabelle[1] mit der mittleren der drei TPC-H-Größen (50GB) jeweils einmal mit `Cutoffs` (Abbildung 4.1) und einmal ohne (Abbildung 4.2) erstellt und ihr vereinnahmter Speicherplatz gemessen. Dazu wurde jeweils die Anzahl der Einträge der Arrays mit der Größe ihrer einzelnen Einträge (32- beziehungsweise 64-Bit-Integer) multipliziert. Aufgrund der Art der Generierung der verschiedenen Größen des Benchmarks ist es ausreichend, sich auf die Betrachtung einer Größe zu beschränken, um Tendenzen zu abzulesen. Die Zahlen innerhalb der Balken sind die relativen Anteile der einzelnen Arrays in Bezug auf die Gesamtgröße. Da `Elf_SIMD` in seinem Aufbau vollständig auf `Elf_Separated_Length` beruht und so keine Unterschiede zwischen diesen bezüglich ihres Speicherplatzes existieren, wurde auf eine Darstellung von `Elf_SIMD` in den Abbildungen verzichtet.
Während ohne `Cutoffs` `Elf64` aus nur einem Array besteht (`Elf`), kommt mit den `Cutoffs` `Elf_TIDs` als Array der `TIDs` hinzu. `Elf_Separated` besitzt ohne `Cutoffs` die drei Arrays `Elf`, `Child_Pointers` und `Monolists`. Unter Berücksichtigung von `Cutoffs` werden diese Arrays durch `Cutoff_Pointers` und wiederum `Elf_TIDs` komplettiert. Da `Elf_Separated_Length` die Variante `Elf_Separated` als Basis besitzt und die Längen nicht extern in einem weiteren Array gehalten werden, stimmen diese beiden Ansätze in Art und Anzahl ihrer Arrays überein.

[1] Da das Hauptaugemerk auf der späteren Analyse der Scan-Algorithmen liegen soll, wurde die `Part`-Tabelle nicht für die Untersuchungen des Speicherplatzes und der Erstellzeit herangezogen.

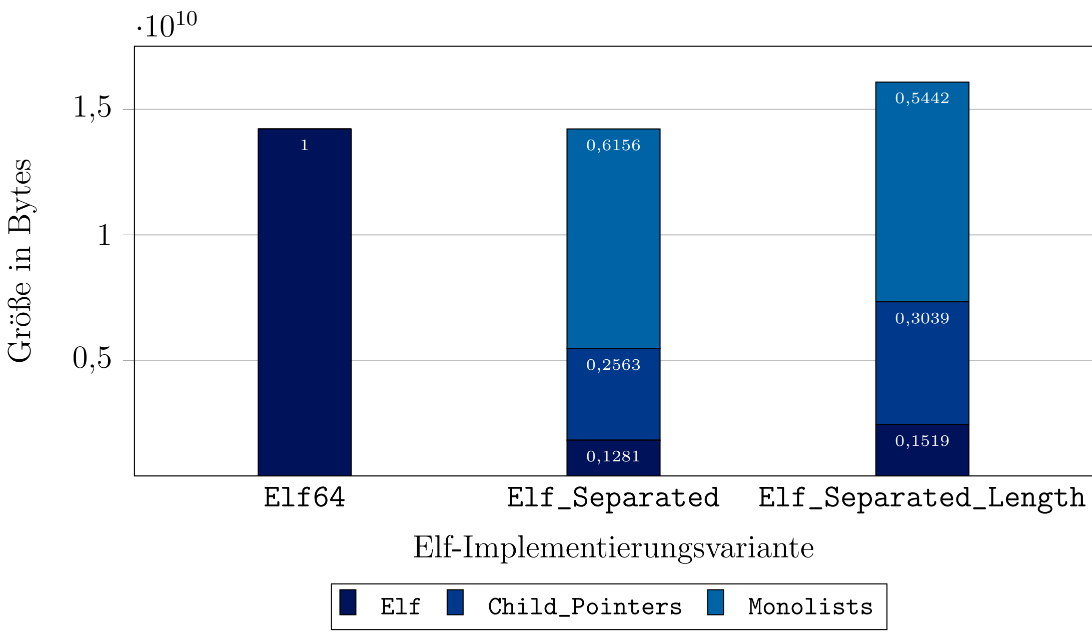


Abbildung 4.1: Benötigter Speicherplatz der Arrays der Elf-Implementierungsvarianten bei der Indexierung der `Lineitem`-Tabelle mit einer TPC-H-Größe von 50GB (ohne `Cutoffs`)

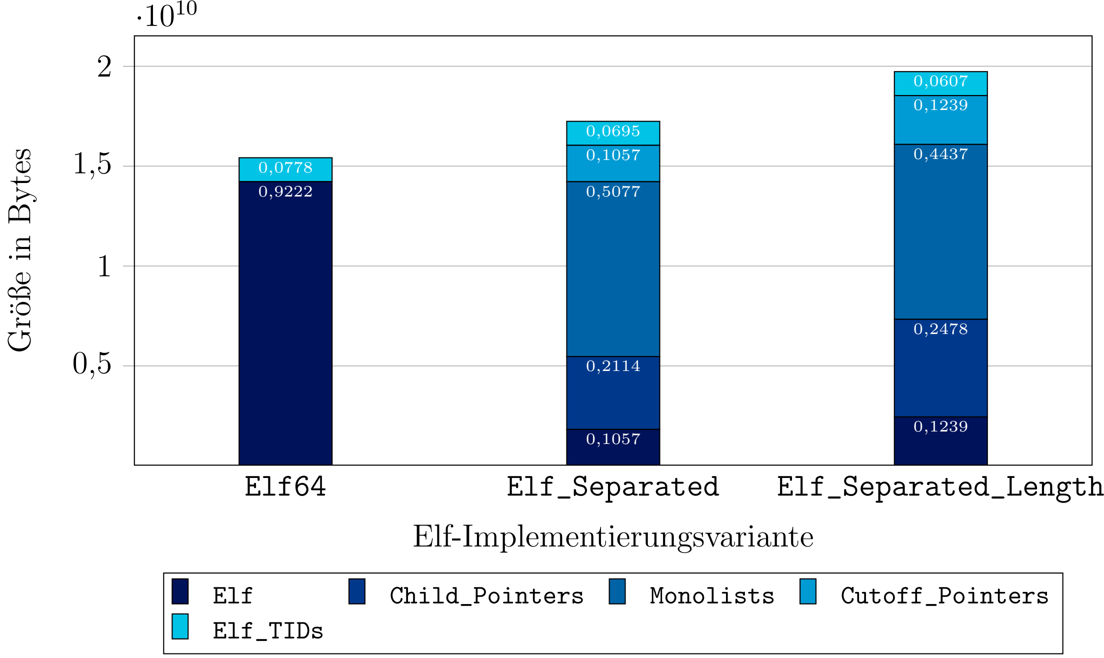


Abbildung 4.2: Benötigter Speicherplatz der Arrays der Elf-Implementierungsvarianten bei der Indexierung der `Lineitem`-Tabelle mit einer TPC-H-Größe von 50GB (mit `Cutoffs`)

4.2.1.2 Diskussion

Beobachtungen ohne Cutoffs

Ohne Cutoffs besitzen Elf64 und Elf_Separated wie erwartet praktisch dieselbe Größe. Für Elf_Separated kommt lediglich ein irrelevant kleiner Speicheroverhead hinzu, der durch die verwendeten C++-Vektoren selbst entsteht. Die Anzahl der Einträge in den Arrays Elf, Child_Pointers und Monolists entspricht der des Elf-Arrays in Elf64.
Es wird deutlich, dass aufgrund der fehlenden Komprimierung respektive Präfixredundanzeliminierung die Monolisten den mit Abstand größten Anteil des Speicherbedarfs ausmachen.
Vergleicht man Elf_Separated mit der in dieser Arbeit ebenfalls neu eingeführten Implementierungsvariante Elf_Separated_Length, die zusätzlich die Dimensionslistenlängen speichert, fällt als erstes auf, dass durch die Extraeinträge innerhalb der Arrays Elf (Längen) und Child_Pointers (leere Einträge) konsequenterweise die Gesamtgröße ansteigt. Die Längen lassen den Elf insgesamt um 13% wachsen; das ist auf einen lokalen Anstieg von je 34% in Elf sowie Child_Pointers zurückzuführen.
Bei den beiden neuen SoA-Ansätzen ist das Child_Pointers-Array exakt doppelt so groß wie das Elf-Array, da die Anzahl der Einträge in beiden übereinstimmt, jedoch anstelle der 32-Bit-Werten in Elf 64-Bit-Einträge in Child_Pointers gespeichert werden. Die Absolutgröße von Monolists ist in diesen Elf-Ansätzen deckungsgleich – bloß der Anteil an der Gesamtgröße fällt in Elf_Separated_Length kleiner aus, weil die Speichernutzung insgesamt höher ist.
Als Nebenprodukt des Speicherverbrauchvergleichs von Elf_Separated und Elf_Separated_Length lassen sich die Anzahl der Dimensionslistenlängeneinträge und damit der regulären Dimensionslisten ($|DL|$) bestimmen als
$|DL| = |\texttt{Elf}_{\texttt{Elf_Separated_Length}}| - |\texttt{Elf}_{\texttt{Elf_Separated}}|$.

Beobachtungen mit Cutoffs

Bei der Verwendung von Cutoffs besitzt das allen drei Ansätzen gemeine Elf_TIDs-Array immer dieselbe Größe – lediglich die prozentualen Anteile weichen wegen der unterschiedlichen Gesamtgrößen voneinander ab.
Aufgrund des Aufbaus des Elf in Pre-Order-Manier entstehen, wie in Abschnitt 3.2.2.1 beschrieben, Lücken im Cutoff_Pointers-Array, die den nun abweichenden Speicherplatzverbrauch von Elf64 und Elf_Separated erklären.
Die Speicherung von Dimensionslistenlängen bei Elf_Separated_Length wirkt sich genauso auf Cutoff_Pointers aus wie auf Elf und Child_Pointers – es entstehen weitere Einträge in Form von Leerstellen. Die prozentuale Steigerung durch die Längen beträgt dabei gleichermaßen wie im Fall ohne Cutoffs 34% für diese drei Arrays. Die Auswirkung der expliziten Speicherung von Dimensionslistenlängen auf den insgesamt vom Elf vereinnahmten Speicherplatz beträgt nunmehr 14% und damit insignifikant mehr als im Fall ohne Cutoffs.

4.2.2 Erstellzeiten des Elf

Der zweite Aspekt, der neben des benötigten Speicherplatz direkt mit der Erstellung des Elf einhergeht, ist die dafür benötigte Zeit.

4.2.2.1 Ergebnisse

In den Abbildungen 4.3 und 4.4 sind die Erstellzeiten in Sekunden für die drei betrachteten TPC-H-Benchmarkgrößen und drei Ansätze `Elf`, `Elf_Separated` und `Elf_Separated_Length` der Darstellung halber in logarithmischer Skalierung abgebildet. Die nicht normalisierten Erstellzeiten können Tabelle 4.4 entnommen werden. Auf `Elf_SIMD` wurde in beiden Darstellungen aus dem gleichen Grund wie bei den Speicherverbräuchen verzichtet – `Elf_SIMD` wird durch `Elf_Separated_Length` repräsentiert.

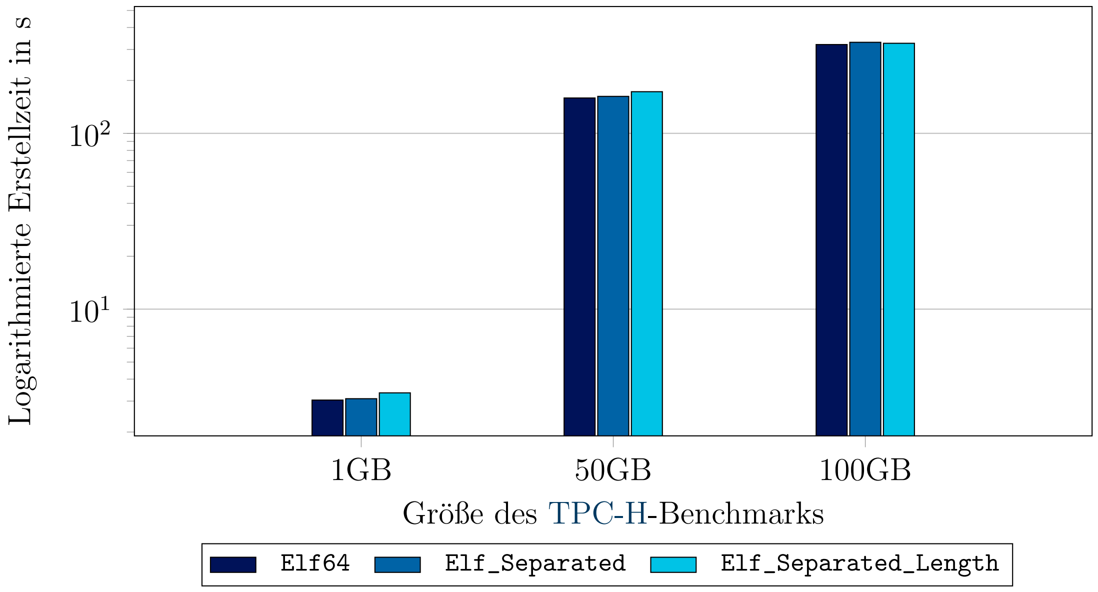


Abbildung 4.3: Erstellzeiten der Elf-Implementierungen für die TPC-H-`Lineitem`-Tabelle für unterschiedliche Größen (ohne `Cutoffs`)

Betrachtet man die Abbildungen 4.3 und 4.4, lässt sich erkennen, dass die Standardimplementierung `Elf64` die Indexgenerierung in allen Fällen am schnellsten abschließt. Abgesehen von dem Szenario 100GB mit `Cutoffs` lässt sich folgende Reihenfolge festmachen: `Elf64` > `Elf_Separated` > `Elf_Separated_Length`.
Bei einer TPC-H-Benchmarkgröße von 1GB liegen die Ansätze jedoch praktisch gleichauf, wenn man sich die realen Zeiten in Tabelle 4.4 anschaut. Die Differenzen liegen hier im kaum feststellbaren Bereich von weniger als einer halben Sekunde. Bei 50GB bleibt der Unterschied zwischen `Elf64` und `Elf_Separated` immer noch im einstelligen Sekundenbereich. Indes liegt die Indexerstellung durch `Elf_Separated_Length` bereits mehr als zehn Sekunden gegenüber `Elf_Separated` im Verzug – sowohl mit als auch ohne `Cutoffs`. Während 100GB ohne `Cutoffs` ob der Beobachtung, dass die Varianten hier wieder sehr dicht beieinander sind, praktisch eine Ausnahme in Anbetracht des zuvor abgezeichneten Trends bilden, sind

mit Cutoffs die größten Diskrepanzen der Testreihe festzustellen. Elf64 liegt hier gut 36 Sekunden vor Elf_Separated_Length.

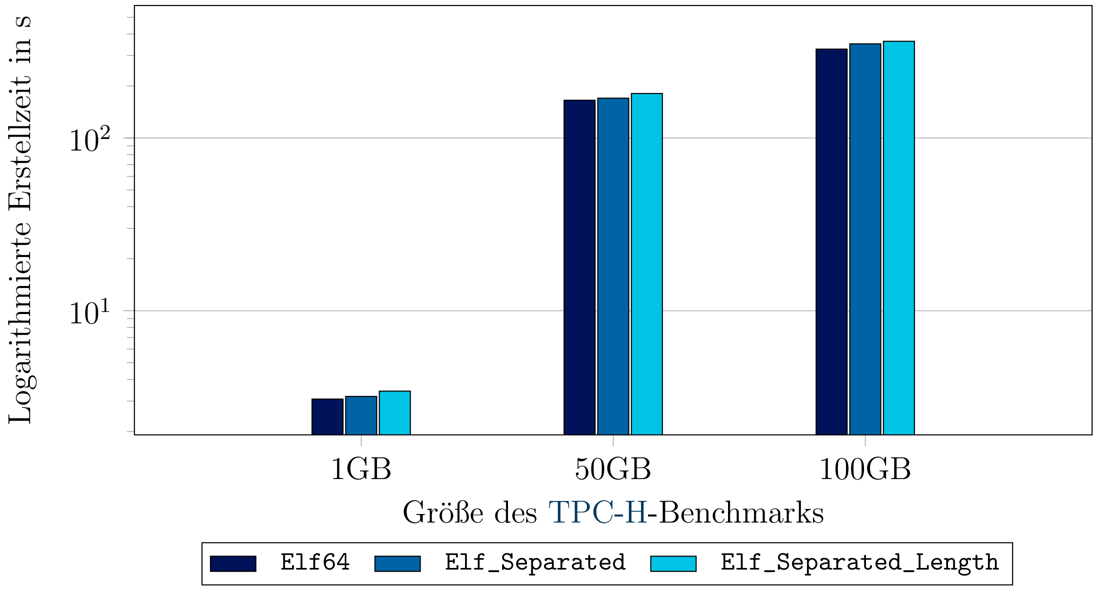


Abbildung 4.4: Erstellzeiten der Elf-Implementierungen für die TPC-H-Lineitem-Tabelle für unterschiedliche Größen (mit Cutoffs)

Größe	Cutoffs	Erstellzeit in s		
		Elf64	Elf_Separated	Elf_Separated_Length
1GB	ohne	3,04	3,09	3,34
	mit	3,08	3,19	3,43
50GB	ohne	158,94	162,26	172,53
	mit	165,35	170	180,92
100GB	ohne	319,95	329,5	325,29
	mit	326,91	350,7	362,84

Tabelle 4.4: Erstellzeiten der Elf-Implementierungen für die TPC-H-Lineitem-Tabelle für unterschiedliche Größen in Sekunden

4.2.2.2 Diskussion

Dass die Ursprungsimplementierung besser als ihre SoA-Erweiterungen abschneidet, kann wohl vor allem durch ihren konsekutiven, stets nur „anhängenden" Schreibzugriff auf *ein* zentrales Array begründet werden. Alle Eintragsarten werden hier zentral lückenlos aneinandergereiht, ohne dass Sprünge im Speicher beim Schreiben bewerkstelligt werden müssen. Selbst ohne Cutoffs sind es bei Elf_Separated und Elf_Separated_Length bereits drei Arrays, die während des Bauens praktisch parallel gefüllt werden müssen. Mit Cutoffs kommt für Elf64 auch nur ein Array hinzu,

nämlich das der TIDs, wohingegen für die anderen Implementierungen darüber hinaus noch das Cutoff_Pointers-Array Berücksichtigung finden muss. So entstehen für Elf64 im Maximum zwei und für die SoA-Implementierungen insgesamt fünf Arrays.
Der überwiegend sichtbare Vorsprung von Elf_Separated gegenüber Elf_Separated_Length könnte hauptsächlich darauf zurückzuführen sein, dass das Markieren des MSB über eine simple Bitoperation (Konjunktion) ein sehr günstiges Vorgehen ist. An dessen Stelle tritt bei der expliziten Dimensionslistenlängenspeicherung eine zusätzliche in den Speicher schreibende Operation. Letztere vergrößert außerdem, wie in der vorherigen Untersuchung gezeigt, zwei der drei Arrays ohne beziehungsweise drei von fünf Arrays mit Cutoffs, was den Schreibzugriff aufwändiger machen kann.

4.2.3 Vergleich der Scan-Algorithmen

Mit der angestiegenen Gesamtgröße der Indexstruktur und bedingt langsameren Erstellzeit wurden die Nachteile aufgezeigt, die mit den strukturellen Anpassungen für die effiziente Nutzung von SIMD einhergehen. Im folgenden Abschnitt soll analysiert werden, ob sich diese Nachteile als zumutbare Trade-offs im Gegenzug einer schnelleren Anfragebearbeitung relativieren lassen.

4.2.3.1 Ergebnisse

In den Abbildungen 4.5 und 4.6 sind die Speedups der drei neu eingeführten Implementierungsvarianten Elf_Separated, Elf_Separated_Length und Elf_SIMD gegenüber der Standardimplementierung Elf64 hinsichtlich der Anfragebearbeitungsgeschwindigkeit mit und ohne Cutoffs dargestellt. Je höher der Speedup ist, desto besser ist eine Variante. Ein Speedup von 2 bedeutet, dass ein Ansatz die jeweilige TPC-H-Anfrage doppelt so schnell wie Elf64 bearbeitet hat. Ein Speedup von 0,5 hingegen besagt das genaue Gegenteil: der Ansatz braucht doppelt so lang wie Elf64. Der Speedup 1 beschreibt, dass eine Umsetzung genauso abschneidet wie Elf64. Aus diesem Grund ist Elf64 in den Graphen nicht dargestellt – der Balken hiervon würde stets bis 1 reichen. Stattdessen ist eine horizontale Gerade bei einem Speedup von 1 zur Orientierung eingezeichnet.
Die vertikale Trennlinie soll die TPC-H-Anfragen nach ihrer Dimensionalität gruppieren. Auf der linken Seite befinden sich die eindimensionalen Anfragen und rechts die mehrdimensionalen.
Es wurde sich auf eine TPC-H-Benchmarkgröße von 100GB in den Abbildungen 4.5 und 4.6 beschränkt, weil sich herausgestellt hat, dass die Beobachtungen bis auf leichte Schwankungen nicht von der Größe des TPC-H-Benchmarks abhängen. Deutlich wird das an den in Tabelle 4.5 dargestellten Standardabweichungen der mittleren Speedups über alle Anfragen. Ohne Cutoffs ist die Standardabweichung hier für alle Ansätze sehr gering. Mit Cutoffs fällt die Standardabweichung durch den „1GB-Ausreißer“ höher aus, weil die Standardimplementierung in diesem Fall erkennbar schlechter abschneidet. Hierfür könnte die abweichende Größenverteilung des Elf bei 1GB verantwortlich sein. Im Gegensatz zu den anderen Testgrößen, nimmt die Größe des Elf durch Hinzuziehen von Cutoffs ohne Elf_TIDs um 17% statt um 0,7% (50GB) beziehungsweise 0,3% (100GB) zu. Dadurch müssen bei 1GB von Elf64 relativ gesehen mehr Cutoff-Zeiger bei den Dimensionslisten mit geladen werden als

bei den anderen Größen, weshalb die Dimensionslisten-Scans verlangsamt werden könnten. Die Auswirkung ist bei den „SoA"-basierten Ansätzen nicht gegeben, weil die Cutoffs hier extern gespeichert werden und beim Abgleich durch ScanDimlist-Cutoffs nicht mit geladen müssen.

Testreihe		Standardabweichung mittlerer Speedups		
		Elf_Separated	Elf_Separated_Length	Elf_SIMD
ohne Cutoffs		0,0077	0,0101	0,0524
mit Cutoffs	mit 1GB	0,0535	0,0618	0,291
	ohne 1GB	0,0225	0,0021	0,0257

Tabelle 4.5: Standardabweichungen der mittleren Speedups der Elf-Varianten über alle Größen mit und ohne Cutoffs

4.2.3.1.1 Ohne Cutoffs

Ohne Cutoffs wird in Abbildung 4.5 eine klare Reihenfolge der Elf-Varianten erkennbar: Elf_SIMD ist schneller als seine strukturelle Basis Elf_Separated_Length, welche wiederum die Oberhand gegenüber Elf_Separated behält. Elf_Separated ist als SoA-Entsprechung von Elf64 mit dieser bis auf Q19 praktisch gleichauf – ersichtlich wird dies an einem Speedup, der sich um 1 bewegt. Elf_Separated_Length hingegen ist mit einem durchschnittlichen Speedup von 1,2 erkennbar schneller als die Standardimplementierung. Die datenparallele Verarbeitung von Elf_SIMD ist die mit Abstand schnellste Umsetzungsalternative. Im Durchschnitt ist Elf_SIMD mehr als 50% schneller als Elf64.
Es fällt auf, dass alle Ansätze bei Q19 die schlechtesten Ergebnisse erzielen.

4.2.3.1.2 Mit Cutoffs

Während ohne Cutoffs die Dimensionalität der Anfrage praktisch keine Rolle bezüglich der Höhe der Speedups gespielt hat, wird in Abbildung 4.6 das genaue Gegenteil bei der Zuhilfenahme von Cutoffs ersichtlich. Für eindimensionale Anfragen ist keine der vorgestellten Elf-Varianten schneller als der Ausgangs-Elf Elf64. Der Speedup bewegt sich hierbei im Schnitt um 0,94. Für die betrachteten mehrdimensionalen Anfragen lassen sich Verbesserungen in der Reaktionszeit bei Anfragen vorrangig durch Elf_SIMD feststellen. Letzteres ist für alle multidimensionalen Testfälle schneller als Elf64. Insbesondere sticht hier Q19P auf der Part-Tabelle mit einem Speedup von etwas mehr als 1,4 hervor. Für diese Anfrage erreicht auch Elf_Separated_Length einen für die Cutoff-Testreihe überdurchschnittlichen Speedup von 1,22.
Auch mit Cutoffs fällt Q19 als Negativausreißer im Vergleich zu den anderen mehrdimensionalen Anfragen auf – vor allem in Hinblick auf die beiden sequentiellen „SoA"-Ansätze. Aufgrund dessen soll in der anschließenden Diskussion Q19 separat betrachtet werden.

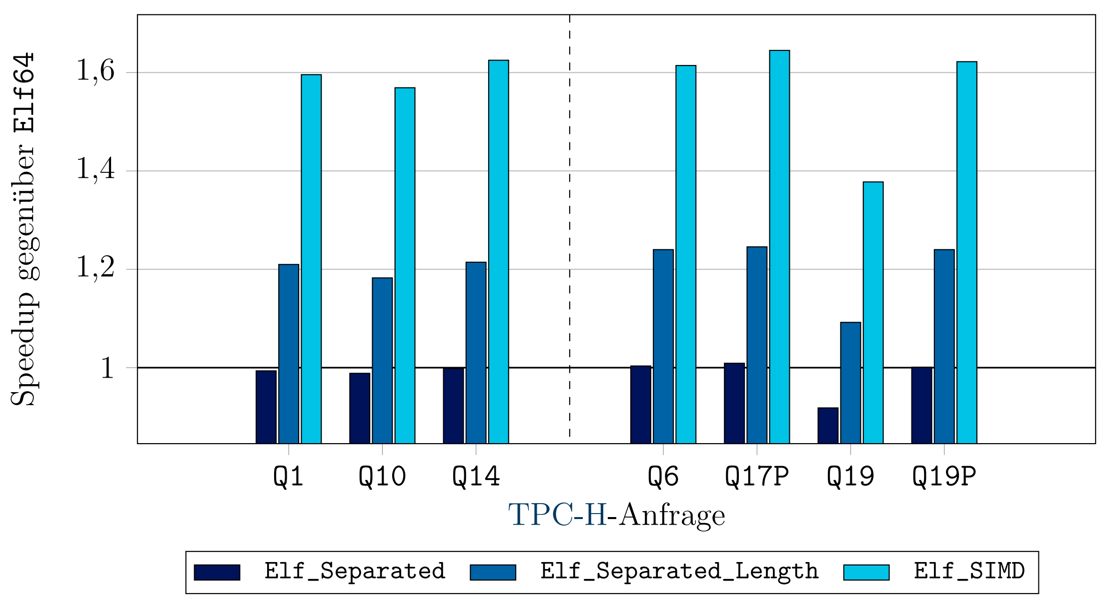


Abbildung 4.5: Speedups der neu implementierten Elf-Ansätze gegenüber `Elf64` für TPC-H-Anfragen mit einer TPC-H-Größe von 100GB (ohne `Cutoffs`)

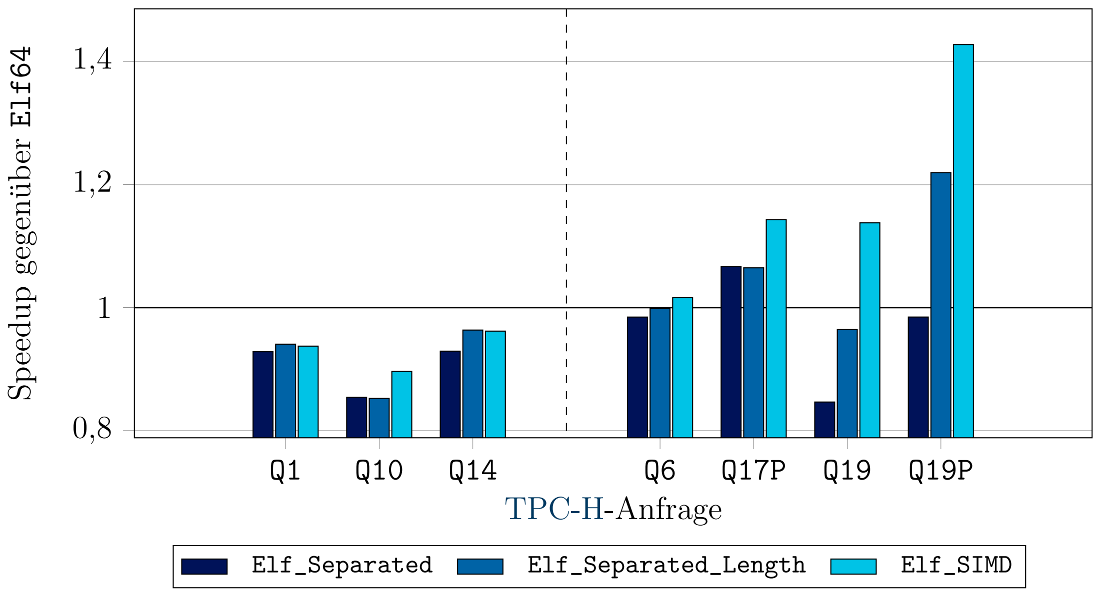


Abbildung 4.6: Speedups der neu implementierten Elf-Ansätze gegenüber `Elf64` für TPC-H-Anfragen mit einer TPC-H-Größe von 100GB (mit `Cutoffs`)

4.2.3.2 Diskussion

Die nun anschließende Diskussion der Ergebnisse bei der Anfragebearbeitung soll sich an der folgenden Struktur orientieren. Zunächst wird das Abschneiden der zwei „Vorstufen“ von `Elf_SIMD`, die Umsetzungen `Elf_Separated` und `Elf_Separated_Length`, näher ausgewertet und begründet. Da es sich bei `Elf_SIMD` um den Kernuntersuchungspunkt dieser Arbeit handelt, wird diese Variante etwas näher als die beiden anderen Ansätze betrachtet. Hier werden ein- und mehrdimensionale Anfragen getrennt jeweils mit und ohne `Cutoffs` analysiert. Daraufhin werden alle drei Ansätze noch einmal zusammen analysiert, bevor die `Q19`-Betrachtung den Abschluss bildet.

4.2.3.2.1 `Elf_Separated`

`Elf_Separated` unterscheidet sich für alle getesteten TPC-H-Größen praktisch nicht von der Standardimplementierung `Elf64`. Der Speedup bewegt sich immer – unabhängig ob mit oder ohne `Cutoffs` – um 1 und beträgt im Durchschnitt über alle Größen und Anfragen 0,96. Es lässt sich also sagen, dass `Elf_Separated` nur unwesentlich schlechter als `Elf64` für die getesteten Anfragen ist. Gleichermaßen lässt sich folgern, dass die Auffassung des Elf als „Structure of Arrays“ als eine der beiden für die SIMD-Nutzung vorgenommenen strukturellen Änderungen für Anfragen außer `Q19` weder einen negativen noch positiven Einfluss auf die Anfragebearbeitungsgeschwindigkeit des Elf besitzt. Erklären lässt sich der tendenziell eher unter 1 bewegende Speedup durch die hinzugekommene Indirektion des Zeigerzugriffs während des Scans. Während bei `Elf64` der relevante Zeiger zur nächsten Dimensionsliste sogleich vom benachbarten Arrayeintrag ausgelesen werden kann, muss bei der „SoA“-Entsprechung zunächst in ein anderes (Zeiger-)Array geschaut werden, ehe der eigentlich angestrebte Sprung innerhalb des `Elf`-Arrays durchgeführt werden kann.

4.2.3.2.2 `Elf_Separated_Length`

Im Gegensatz zu seiner Basis `Elf_Separated` mit einer impliziten Längenspeicherung über das MSB, berechnet `Elf_Separated_Length` für einen Großteil der Anfragen die Ergebnismenge schneller als die Ursprungsimplementierung `Elf64`. Da der einzige Unterschied zwischen den beiden „SoA“-Ansätzen die Speicherung der Dimensionslistenlänge ist, muss der Grund für das deutlich bessere Abschneiden von `Elf_Separated_Length` genau darin gesucht werden. Der Unterschied lässt sich für partielle Bereichsanfragen weiter konkretisieren: während bei der expliziten Längenspeicherung zum Scannen der Dimnensionslisten eher das Prinzip einer klassischen `for`-Schleife genutzt wird[1], findet bei `Elf_Separated` wie bei der Ursprungsimplementierung eine typische `while`-Schleife Anwendung. Gemeint ist hiermit, dass bei der `for`-Variante die maximale Anzahl an Iterationen, die Dimensionslistenlänge, von vornherein feststeht. Bei `Elf_Separated` hingegen ist das nicht der Fall. Hier lässt sich nicht bestimmen, wann eine Dimensionsliste ihr Ende findet, ohne jedes

[1] Im Pseudocode aus Abschnitt 3.2.3 wurde lediglich der Darstellung halber weiterhin eine `while`-Schleife verwendet.

Element auf die Gesetztheit des MSBs geprüft zu haben. Die Ergebnisse zeigen, dass das `for`-Schleifen-Prinzip vom verwendeten GNU-Compiler deutlich besser optimiert werden kann als die `while`-Schleifenvariante.

4.2.3.2.3 `Elf_SIMD` - Eindimensionale Anfragen

Ohne `Cutoffs`

Die eindimensionalen Vertreter der Testreihe sind `Q1`, `Q10` und `Q14`. Man könnte annehmen, dass `Elf_SIMD` für diese Anfragen keinen großen Speedup gegenüber `Elf_Separated_Length`[1] erreichen kann. Schließlich ist es bei solch einer Anfrage so, dass vor und nach der selektierten Dimension ein in allen Ansätzen identisch umgesetzter, bedingungsloser rekursiver Aufruf für alle Dimensionslisteneinträge erfolgen muss, da auf den anderen Spalten eben keine Einschränkungen vorliegen. Ein Vorteil von SIMD, nämlich der datenparallele Vergleich von Dimensionslistenwerten gegen ein Intervall, ist somit auf lediglich eine Dimension beschränkt. Schlimmer noch: wenn die Selektion nur auf der ersten Dimension liegt, wie es bei `Q1` und `Q14` der Fall ist, kann in einer Dimensionsliste nie SIMD zum Einsatz kommen. Grund ist hierfür die besondere Speicherung der ersten Dimension, bei der Vergleiche von Werten durch direkte Zugriffe substituiert werden, wodurch keine vektorbasierte Verarbeitung Anwendung finden kann (vgl. Abschnitt 3.2.1.1).
Dennoch ist der Unterschied zwischen `Elf_Separated_Length` und `Elf_SIMD` beachtlich. Wenn die Ursache hierfür nicht in der Verarbeitung der regulären Dimensionslisten liegen kann, muss sie in der Verarbeitung von `Monolisten` gesucht werden.

Zur Identifizierung des konkreten Vorteils von `Elf_SIMD` gegenüber den anderen Implementierungsvarianten bei der Handhabung von `Monolisten`, gilt es, die Funktionsweise des Scans dieser zu rekapitulieren. Der Scan von `Monolisten` bricht erst dann ab, wenn auf einer *selektierten* Dimension eine Intervallsvorgabe verletzt ist; ansonsten qualifiziert sich die zur `Monolist` gehörende `TID`-Liste. Nun gibt es aber bei eindimensionalen Anfragen nur *eine* selektierte Dimension. Bei den analyisierten Anfragen nimmt diese Dimension erschwerenderweise auch noch die erste (`Q1` und `Q14`) respektive fünfte (`Q10`) Position innerhalb der Indexstrutkur ein. Das sind Ebenen im Indexbaum, bei denen quasi keine `Monolisten` vorkommen (siehe Abbildung 2.7 aus dem Grundlagenkapitel). Kurzum: es werden innerhalb des `Monolisten`-Scans nie Intervallsprüfungen durchgeführt, sondern jeder Wert wird „durchgewunken“, weil auf keiner der enthaltenen Dimensionen eine Selektion vorliegt. Der entscheidende Vorteil von SIMD ist hierbei, dass – im Fall von AVX2 – acht Dimensionen gleichzeitig betrachtet werden können und somit viel schneller als bei den restlichen Implementierungsvarianten erkannt wird, dass eben für gar kein `Monolisten`-Element ein zu prüfendes Intervall vorliegt.
Am deutlichsten wird das beschriebene Verhalten, wenn man `Q1` oder `Q14` als Beispiel nimmt. Hier wird durch die abweichende Speicherung der ersten Dimension direkt auf die untere und obere Grenze von `l_shipdate` zugegriffen; anschließend werden alle folgenden Dimensionen ohne weitere Einschränkungen durchlaufen, um an die `TIDs` zu gelangen. Dennoch wird in `ScanMonolist` (siehe Algorithmus 3), welches

[1]Es ist explizit nicht gegenüber `Elf64` gemeint. Da `Elf_SIMD` auf `Elf_Separated_Length` basiert, ist der direkte Vorteil, den man durch SIMD erhält, wie zuvor erwähnt am ehesten durch einen Vergleich von `Elf_SIMD` mit `Elf_Separated_Length` feststellbar.

von allen Ansätzen außer Elf_SIMD genutzt wird, bei jedem Aufruf (unnötigerweise) für jedes Element der Monolist *einzeln* geprüft, ob auf der korrespondierenden Dimension eine Selektion vorliegt. Bei Elf_SIMD hingegen wird die Anzahl dieser Prüfungen im besten Fall durch acht geteilt.
Diese Überlegenheit wird potenziert durch die aus der Größenanalyse der einzelnen Elf-Arrays hervorgehenden Tatsache, dass innerhalb des Arrays Monolists mehr als die Hälfte der Einträge des Gesamt-Elfs lokalisiert sind.
Man könnte argumentieren, dass man eine Art *loop unrolling* zur Optimierung von ScanMonolist für die Ansätze Elf64, Elf_Separated und Elf_Separated_Length nutzen könnte. Die vorgestellten Ausgangslagen könnten somit als „voreingenommen" gegenüber der SIMD-Variante interpretiert werden. Gegen diese Argumentation sprechen jedoch zwei Feststellungen. Zum einen entspricht ein solches Vorgehen nicht der Standardimplementierung des Elf, mit der der Vergleich vordergründig stattfinden soll. Zum anderen wäre das ein Optimieren auf einen speziellen Anfragetypus hin und kein allgemein optimales Vorgehen. Währenddessen liegt ein „Vorausschauen" wie es hier der Fall ist in der Natur der Sache bei der Anwendung von SIMD.

Ungeachtet der erläuterten Stärken von Elf_SIMD bei der Monolisten-Verarbeitung, läge die Vermutung nahe, dass Q10 noch mehr von der datenparallelen Vorgehensweise profitieren müsste als die anderen beiden eindimensionalen Anfragen. Hier findet die Selektion schließlich in der fünften Dimension statt, wodurch in dieser Dimension auch SIMD innerhalb von ScanDimListSIMD (siehe Algorithmus 7) genutzt werden kann und nicht ausschließlich bei den Monolisten wie es bei Q1 und Q14 der Fall ist. Allerdings ist der Wertebereich des beschränkten Attributs l_returnflag mit maximal drei möglichen Werten („R", „A" oder „N") zu klein, als dass das volle Potenzial einer vektorbasierten Bearbeitung, die acht Werte gleichzeitig betrachten könnte, ausgenutzt wird.

Der letzte Analyseaspekt für eindimensionale Anfragen soll die Selektivität der Anfrage sein. Zwar ist die Anzahl der zu betrachtenden Monolisten abhängig von dieser, sie beeinflusst jedoch alle Ansätze gleichermaßen. Deshalb ist für eindimensionale Anfragen der „Vorsprung" von Elf_SIMD gegenüber den anderen Varianten nicht bedingt durch die Selektivität der Anfrage. Sichtbar wird das dadurch, dass sich die Reihenfolge der Speedups von Elf_SIMD zwischen Q1, Q10 und Q14 bei den unterschiedlichen Größen abwechselt und keine Tendenz zu erkennen ist.

Mit Cutoffs

Dass im eindimensionalen Fall auch SIMD keine Beschleunigung der Anfragebearbeitung mit Cutoffs erreichen kann, lässt sich erneut aufgrund der Handhabung der Monolisten erklären. Ohne Cutoffs war der Hauptvorteil von SIMD, dass die Prüfung, ob eine Selektion auf einer Spalte einer Monolist vorliegt, durch die vektorbasierte Bearbeitung effizienter umgesetzt werden konnte. Durch Cutoffs entfällt aber dieser Vorteil *vollständig*, da nach der zuletzt selektierten Dimension direkt zu den TIDs gesprungen werden kann – es muss also im Anschluss an diese Dimension nie wieder geprüft werden, ob eine Selektion auf einem Attribut vorliegt oder nicht. Das wird wegen der innerhalb des Elf weit vorn platzierten Positionen der von Q1,

Q10 und Q14 eingeschränkten Attribute (Stellen 1 und 5) und der Tatsache, dass Monolisten hauptsächlich in den „hinteren“ Dimensionen auftreten, verstärkt (vgl. wieder Abbildung 2.7). Das bedeutet konkret, dass für die vorliegenden eindimensionalen Anfragen ohne Cutoffs bei *allen* im Elf vorkommenden Monolisten die durch SIMD optimierte Selektionsprüfung durchgeführt wurde und mit Cutoffs *kein einziges Mal*.
Darüber hinaus wurde für Q1 und Q14 für alle Implementierungsvarianten derselbe Algorithmus verwendet: ScanFirstDimlistCutoffs (Algorithmus 4), sodass für diese zwei Anfragen gar kein Vor- oder Nachteil entstehen kann.

4.2.3.2.4 Elf_SIMD - Mehrdimensionale Anfragen

Ohne Cutoffs

Im Gegensatz zu eindimensionalen Anfragen kann bei Q6, Q17P und Q19P zusätzlich SIMD beim Scan der regulären Dimensionslisten mit ScanDimListSIMD genutzt werden – nicht nur bei den Monolisten. Mit p_brand, p_container, l_discount und l_quantity werden zudem Spalten selektiert, die eine Wertebreichsspanne von mindestens 25 Werten aufweisen. Entsprechend können längere Dimensionslisten auftreten, von denen Elf_SIMD gegenüber der anderen Ansätzen profitieren könnte. Es kann hier also die Erwartungshaltung entstehen, dass die Speedups noch einmal höher sind als im eindimensionalen Fall. Das lässt sich jedoch anhand der Experimente nicht bestätigen. Vielmehr liegt ein ähnliches Niveau wie bei Q1, Q10 und Q14 vor. Begründet werden kann das damit, dass der Speedup bekanntlich die Verbesserung der Anfragebearbeitungszeit bezüglich Elf64 beschreibt. Wie es von Broneske et al. gezeigt wurde, handelt es sich bei mehrdimensionale Anfragen um das „Steckenpferd“ des (Ursprungs-)Elf [BKSS17]. Da Elf64 für diesen Anfragetypus sehr gut abschneidet, fällt das Optimierungspotenzial der neuen Ansätze niedriger aus; das spiegelt sich in den nicht außerordentlich großen Speedups wider.

Mit Cutoffs

Mit Cutoffs wird bei multidimensionalen Anfragen vor allem durch Elf_SIMD ein Speedup gegenüber Elf64 bei allen vorliegenden Anfragen außer Q6 sichtbar. Wie bei der Vorstellung von Cutoffs erläutert, profitiert die Anfragebearbeitung am meisten von den Cutoffs, wenn die letztselektierte Dimension eine Position möglichst weit vorn einnimmt. Das erklärt, warum die größten Speedups von Elf_SIMD bei Q19 und Q19P zu finden sind. Bei diesen muss bis zur siebenten Dimension der „normale“ Scan-Ablauf verfolgt werden, ehe Cutoffs genutzt werden können. Je mehr dieser SIMD-optimierten regulären Dimensionslisten-Scans ausgeführt werden, desto größer wird der Vorteil von Elf_SIMD. Da aber die Monolisten praktisch wieder fast vollständig übersprungen werden, entfällt ein beschriebener Mehrwert, der durch die SIMD-Nutzung entsteht, sodass mit Cutoffs auch im mehrdimensionalen Fall nicht die Speedups erreicht werden können wie ohne sie.

4.2.3.2.5 Gesamtbetrachtung der neuen Elf-Varianten

Betrachtet man alle drei neu vorgestellten Implementierungsvarianten zusammen, fällt auf, dass sie für gleiche Anfragen sehr ähnlich abschneiden, wenn man diese Anfrage im Vergleich zu den restlichen betrachtet. Mit anderen Worten: arbeitet ein Ansatz für eine Anfrage besonders gut, so ist Selbiges für einen anderen Ansatz festzustellen.
Am deutlichsten wird das für 100GB sichtbar. Hierfür sind die „Rangfolgen" der Speedups zu einer Anfrage im Vergleich zu den restlichen für alle drei Ansätze in Tabelle 4.6 dargestellt. Rang 1 von `Q17P` bei `Elf_SIMD` ohne `Cutoffs` sagt aus, dass bei keiner anderen Anfrage durch `Elf_SIMD` ein höherer Speedup gegenüber `Elf64` ohne `Cutoffs` entstanden ist. Wie man sieht, verhalten sich `Elf_Separated` und `Elf_Separated_Length` ohne `Cutoffs` identisch; mit `Cutoffs` gibt es leichte Abweichungen. Bei `Elf_SIMD` gibt es, unabhängig von der Zuhilfenahme von `Cutoffs`, geringe Verschiebungen bezüglich der Rangfolgen der anderen beiden Ansätze.

`Cutoffs`	TPC-H-Anfrage	Speedup-Rangfolge		
		`Elf_Separated`	`Elf_Separated_Length`	`Elf_SIMD`
ohne	`Q1`	5	5	5
	`Q10`	6	6	6
	`Q14`	4	4	2
	`Q6`	2	2	4
	`Q17P`	1	1	1
	`Q19`	7	7	7
	`Q19P`	3	3	3
mit	`Q1`	5	5	6
	`Q10`	6	6	7
	`Q14`	4	4	5
	`Q6`	3	3	4
	`Q17P`	1	2	3
	`Q19`	7	7	2
	`Q19P`	2	1	1

Tabelle 4.6: Rankings der Speedups von Anfragen im Vergleich zu denen der restlichen Anfragen ausgeführt durch die Elf-Varianten bei einer TPC-H-Größe von 100GB

Zu erklären sind die Beobachtungen dadurch, dass es sich bei den drei Ansätzen jeweils um schrittweise Optimierungen handelt. `Elf_Separated` wird durch die expliziten Längen mit `Elf_Separated_Length` direkt verbessert und dieser Ansatz

wird wiederum durch seine SIMD-Entsprechung `Elf_SIMD` optimiert. Aufgrund dessen spielt die Beschaffenheit der Anfrage (ein- oder multidimensional, wenig oder hoch selektiv, Position der Spalte im Elf) für das Abschneiden der Varianten im direkten Vergleich eine untergeordnete Rolle.

4.2.3.2.6 `Q19`-Anfrage

`Q19` ist für alle Größen und Elf-Varianten die mit Abstand am schlechtesten abschneidende untersuchte TPC-H-Anfrage – mit und ohne `Cutoffs`.
Der Grund kann in der Art der partiellen Anfrage, die bei `Q19` vorliegt, gefunden werden – es handelt sich nämlich um eine partielle *Bereichs*anfrage. Konkret ist die Bereichsdefinition von `l_quantity` Ursache für das schlechte Abschneiden der neu vorgestellten „SoA"-basierten Ansätze.

Vergleich mit den anderen Anfragen

Das Intervall von maximal 10 Werten (siehe Codeauszug 4.4) für `l_quantity` ist – im Vergleich zum Rest der Testreihe – relativ groß. Anders als bei `Q1`, `Q6` oder `Q14` (siehe Codeauszüge 2.3, 4.1 und 4.2), die ebenfalls breitere Wertebereiche einschränken, liegt die relevante Selektion nicht auf der ersten Dimension vor. Für `l_shipdate` von `Q1`, `Q6` und `Q14` werden zwar auch größere Bereiche angegeben, allerdings kann hier in der ersten Dimension aufgrund deren abweichender Speicherung viel schneller an die Zeiger zu den darunterliegenden Dimensionen durch direkten Zugriff gelangt werden. Auch die anderen Intervalldefinitionen von `Q6` unterscheiden sich von der von `Q19`: sie geschehen zu einem späteren Zeitpunkt, zu dem bereits eine Vorselektion stattgefunden hat, wodurch der Kandidatenkreis schon kleiner geworden ist.
Die bezüglich des Namens verwandte `Q19P`-Anfrage ist in drei wesentlichen Punkten von `Q19` verschieden. Zum einen ist das Intervall von `p_brand` halb so groß wie das für `l_quantity` in `Q19`. Zum anderen wird hier die gegenüber der `Lineitem`-Relation viel kleinere `Part`-Tabelle angefragt. Außerdem ist die Selektivität von `Q19P` viel geringer.

Ursache für das schlechtere Abschneiden der neuen Ansätze

Durch das breite Wertintervall bei `Q19` als erstes auszuwertendes Selektionskriterium können sich potenziell mehr Werte der untersuchten Dimensionslisten der dritten Spalte qualifizieren. Dementsprechend müssen relativ betrachtet mehr Zeiger ausgelesen und verfolgt werden. Hier liegt wahrscheinlich der Knackpunkt: die Indirektion von Zeiger- und anschließendem Dimensionslistenwertzugriff. Bei den „SoA"-Ansätzen wird durch das Auslagern der Zeiger in `Child_Pointers` wie bereits zuvor erwähnt ein weiterer Speichersprung notwendig, der bei `Elf64` nicht vorhanden war. Bei `Elf_Separated_Length` und `Elf_SIMD` wird dieser Nachteil durch deren zuvor bei den restlichen Anfragen beschriebenen Vorteile etwas aufgefangen – bei `Elf_Separated` hingegen nicht.
Diese Beobachtung ist unabhängig davon, ob `Cutoffs` genutzt werden oder nicht, da `Cutoffs` keinen Einfluss auf die Bearbeitung der dritten Dimension nehmen.

4.2.4 Zusammenfassung

Dieses Kapitel sollte dazu dienen, die Forschungsfrage **FF** 2. zu beantworten. Bevor die Ergebnisse vorgestellt und interpretiert wurden, wurde der Experimentaufbau dafür beschrieben. Dazu wurde die zugrundeliegende Hardwareumgebung, in Form einer AVX2-unterstützenden Maschine sowie die untersuchten TPC-H-Anfragen auf den Tabellen `Lineitem` und `Part` vorgestellt. Gleichzeitig wurden die Parameter der Experimente erläutert.
Die Evaluierungen richteten sich nach den drei Gliederungspunkten der zweiten wissenschaftlichen Fragestellung. Es wurde die Größe der erstellten Elfs, die benötigten Zeiten zur Generierung der Strukturen und die Bearbeitungsgeschwindigkeit von Anfragen im Vergleich zur Standardimplementierung `Elf64` von Broneske et al. [BKSS17] analysiert. Folgende Antworten ließen sich bezüglich der durch diese Arbeit hervorgebrachten Elf-Varianten `Elf_Separated`, `Elf_Separated_Length` und `Elf_SIMD` finden:

1. Die explizite Speicherung der Längen bei `Elf_Separated_Length` (und damit auch `Elf_SIMD`) benötigt wie erwartet mehr Speicherplatz als `Elf64` und `Elf_Separated`. Letztere sind dahingegen in Bezug auf ihre Größe praktisch gleichauf ohne `Cutoffs`. Das Hinzuziehen von `Cutoffs` sorgt dafür, dass auch `Elf_Separated` wegen seiner algorithmischen Erzeugung in Pre-Order-Form mehr Speicher einnimmt als `Elf64`.

2. Die Varianten `Elf_Separated` und `Elf_Separated_Length` (und damit auch `Elf_SIMD`) benötigen länger, um ihren Elf zu generieren als `Elf64`. Erklärt werden kann dies durch den Overhead des gleichzeitien Schreibens von verschiedenen Arrays, die mit `Cutoffs` und expliziten Längen darüber hinaus insgesamt größer und somit „schreibaufwändiger“ sind als bei `Elf64`.

3. Die Erkenntnisse bezüglich der Anfragegeschwindigkeiten lassen sich wie folgt zusammenfassen:

   (a) Die Variante `Elf_Separated` ist lediglich etwas langsamer als `Elf64`. Erklärt werden kann das durch die hinzugekommene Indirektion von Zeigern.

   (b) Die Variante `Elf_Separated_Length` ist für einen Großteil der getesteten TPC-H-Anfragen schneller als die Ursprungsimplementierung `Elf64`. Zurückzuführen ist der ohne `Cutoffs` höhere Speedup von bis zu knapp 1,25 durch die besser „abschätzbare“ Scan-Abbruchbedingung von `Elf_Separated_Length`.

   (c) Die datenparallele Variante `Elf_SIMD` ist für nahezu alle Anfragen (eindimensionale Anfragen mit `Cutoffs` bilden die Ausnahme) die mit Abstand schnellste Umsetzungsalternative. Es werden durch die vektorbasierte, „vorausschauend“ agierende Verarbeitung der `Monolisten` und Dimensionslisten Speedups von bis zu 1,65 gegenüber `Elf64` erreicht – sowohl bei ein- als auch bei mehrdimensionalen Anfragen. Der durchschnittliche Vorsprung ist ohne `Cutoffs` höher als mit.

(d) Die neuen Elf-Varianten verhalten sich insgesamt betrachtet wegen des aufeinander aufbauenden Charakters (`Elf_SIMD` basiert auf `Elf_Separated_Length`, welches auf `Elf_Separated` beruht) bei gleichen Anfragen ähnlich. Ohne `Cutoffs` ist diese Beobachtung deutlicher.

5. Verwandte Arbeiten

In diesem Kapitel soll ein grober Überblick über verwandte Arbeiten vorgestellt werden. Da `Elf_SIMD` den Kernbeitrag dieser Arbeit darstellt, wird sich hierbei auf die Nutzung von SIMD innerhalb anderer, baumbasierter Indexstrukturen konzentriert. Konkret sollen Seg-Tree, FAST, VAST und ART betrachtet werden. Im Gegensatz zu
`Elf_SIMD` sind diese Ansätze auf den eindimensionalen Einsatz ausgelegt.
Grundlage des Kapitels ist das Selbige der Dissertation von David Broneske, auf die an dieser Stelle verwiesen sei [Bro19]. Diese bietet darüber hinaus eine tiefergehende Analyse der verwandten Arbeiten – auch hinsichtlich anderer multidimensionaler, wenngleich nicht SIMD-optimierter Indexstrukturen.

5.1 Seg-Tree

In *Adapting Tree Structures for Processing with SIMD Instructions* stellen Zeuch et al. den sogenannten Seg-Tree vor [ZFH14]. Dieser basiert auf einem B^+-Baum, wobei die inneren Knoten, genannt Segmente, einen k-ären Suchbaum repräsentieren. Der Parameter k wird in Abhängigkeit von der maximalen Größe der SIMD-Register und der maximalen Anzahl an Bits zur Repräsentation des gespeicherten Datentyps errechnet.
Zur Suche innerhalb des Baums, wird wie auch beim Elf eine Linearisierung von Zeuch et al. vorgenommen. Hierfür werden sowohl Pre- als auch Level-Order-Varianten vorgestellt. Jede Partition, die während des k-ären Suchalgorithmus betrachtet wird, passt in ein SIMD-Register.
Die Performanz dieses Ansatzes ist abhängig von den zu speichernden Datentypen – je kleiner diese sind, desto mehr Daten können datenparallel verarbeitet werden. Im 32-Bit-Integer-Fall wird ein Verbesserung von Faktor 4, für 64-Bit-Integer noch Faktor 2 gegenüber eines regulären k-ären Suchbaums erreicht.

5.2 FAST und VAST

5.2.1 FAST

Fast Architecture Sensitive Tree Search, kurz *FAST*, ist ein SIMD-optimierter Binärbaumansatz von Kim et al. [KCS+10]. Die Performanzsteigerung soll bei diesem Index durch einen bewussten Umgang mit den Caches erreicht werden: Cache-Lines sollen optimal ausgenutzt und Cache Misses damit möglichst gering gehalten werden.
Zentrale Idee ist dafür das Einteilen des Suchbaums in drei verschiedene Granularitäten. Auf oberster Ebene befinden sich Teilbäume, die auf eine Seite geschrieben werden können. Darunter befinden sich weitere, kleinere Teilbäume, die wiederum jeweils in eine Cache-Line passen. Innerhalb der dritten und letzten Ebene sind Knoten der Teilbäume der zweiten Ebene lokalisiert, die in ein SIMD-Register passen.
Zum Vergleich mit anderen Ansätzen wurde die Anzahl an bearbeitbaren Suchanfragen je Sekunde gemessen – diese sei fünfmal besser als die zuvor von Schlegel et al. angegebene Anzahl [SGL09].

5.2.2 VAST

Yamamuro et al. stellen mit *VAST* eine Erweiterung von *FAST* vor. Dazu nutzen sie die vorgeschlagene Hierarchie von Einteilungen des Suchbaums und erweitern diese um die Kompression von Knoten und einer verbesserten Ausnutzung der SIMD-Potenziale [YOHY12]. Konkret werden die inneren Knoten auf 16-Bit-Schlüssel mit einer verlustbehafteten Kompressionsmethode (engl. *lossy compression*) verkleinert, wenn sie innerhalb des Baumns über einem parametrisierten Schwellwert liegen. Wenn sie darunter liegen, erfolgt die Kompression auf 8-Bit-Werte. Auf Blattebene wird die verlustfreie Kompression (engl. *lossless compression*) „*P4Delta*“ angewandt. Zum Ausgleich der Kompressionsfehler auf den höheren Ebenen wird ein entsprechender Algorithmus vorgestellt.

5.3 ART

Der *Adaptive Radix Tree*, kurz *ART*, von Leis et al. beruht auf der Idee, anstelle konstanter Knotengrößen variable im Indexbaum zu nutzen. Dazu unterteilen sie die Knoten in vier Arten, die jeweils 8-Bit-Einträge besitzen:

- **Node4**: besteht aus zwei Arrays mit je vier Einträgen. Ein Array speichert die Suchschlüssel, das andere die korrespondierenden Zeiger auf die Kindknoten.
- **Node16**: ist analog *Node4* aufbaut, nur dass sechzehn statt der vier 8-Bit-Werte gespeichert werden können.
- **Node48**: speichert keine Suchschlüssel direkt, sondern nutzt ein 256-Einträge fassendes Array, auf welches direkt über die Ausprägung der Schlüssel zugegriffen werden kann. Innerhalb dieses Arrays sind Indexe des zweiten, 48-Einträge großen Zeiger-Arrays enthalten.
- **Node256**: beinhaltet bis zu 256 Zeiger. Der Zugriff erfolgt direkt über die Ausprägung der Schlüssel.

Sollte ein Knoten eines Typs dessen Kapazität überschreiten, wird er in den nächstgrößeren umgewandelt. SIMD wird von Leis et al. bei der Indexbaumtraversierung genutzt.
Im direkten Vergleich der Autoren wird gezeigt, dass *ART* wegen der geringeren Cache-Misses und CPU-Zyklen bei einer zufälligen Schlüsselsuche besser als *Seg-Tree* oder *FAST* agiert.

5.4 Fazit und Vergleich mit Elf

Die vorgestellten Indexstrukturen nutzen verschiedene Ansätze, wie die Komprimierungstechniken, horizontale Vektorisierung oder spezialisierte Eintragstypen, um SIMD (besser) einsetzen zu können.
In *VAST* kommt beispielsweise eine Komprimierung zum Einsatz, die es ermöglichen soll, mehr Werte auf einmal mit SIMD vergleichen zu können. Die Performanzsteigerung dadurch überwiegt den Zeitverlust, der durch das Korrigieren möglicher Fehler der verlustbehafteten Komprimierung entsteht. Für lange, die SIMD-Registergröße übersteigende Dimensionslisten wäre dieses Vorgehen auch für den Elf vorstellbar. Bei `Monolisten` hingegen ist eine Komprimierung schwieriger, da man hier nicht innerhalb eines Attributs, sondern über mehrere Dimensionen agiert.
Alle Ansätze implementieren die Idee der horizontalen Vektorisierung: ein konstanter Suchwert wird gegen einen Vektor von Werten verglichen. In `Elf_SIMD` findet sowohl horizontale als auch vertikale Vektorisierung statt. In `ScanDimlistSIMD` wird jeweils die untere und obere Intervallgrenze der selektierten Spalte gegen die Dimensionsliste (horizontal) verglichen. Dahingegen wird in `ScanMonolistSIMD` ein Abgleich verschiedener unterer und oberer Grenzen gegen Werte unterschiedlicher Dimensionen (vertikal) vollzogen.
ART sieht spezielle Eintragstypen vor, abhängig von der Größe der zu speichernden Daten. Dieses Konzept wird nicht direkt in `Elf_SIMD` genutzt – allerdings schon im übertragenen Sinne. Schließlich stellte das Überführen des Elf in eine „Structure of Arrays“ eine Grundvoraussetzung für die Nutzung von SIMD dar. Insgesamt kann zwischen fünf Eintragstypen im Elf unterschieden werden: Dimensionslisten- und `Monolisten`werte, Dimensionslisten- und `Cutoff`-Zeiger sowie `TID`s. Damit wurde die Basis für etwaige weiterführende Optimierungen basierend auf dem Typen eines Eintrags geschaffen.

6. Zusammenfassung und Ausblick

In diesem Kapitel soll es darum gehen, die Erkenntnisse dieser Arbeit in Kürze zusammenzufassen und einen Ausblick darauf zu geben, welchen Fragen und Problemstellungen sich in zukünftigen Auseinandersetzungen mit dem Thema gewidmet werden könnte.

6.1 Zusammenfassung

Indexstrukturen dienen dazu, den Zugriff auf die Daten einer Datenbanktabelle bei Anfragen mit Selektionen zu beschleunigen. Für die Bearbeitung von Selektionen auf mehreren Attributen sind multidimensionale Indexstrukturen besonders geeignet. Der Elf von Broneske et al. ist ein Vertreter dieser Klasse von Indexstrukturen [BKSS17]. Innerhalb dieser Arbeit wurde mit `Elf_SIMD` eine datenparallele, vektorbasierte Implementierung des Indexes von Broneske et al. eingeführt und analysiert. Dazu wurden im Verlauf dieser Arbeit folgende zwei, an dieser Stelle verkürzt formulierte, Forschungsfragen beantwortet:

FF 1. Welche Änderungen an der Struktur und den Scan-Algorithmen der Implementierung des Elf von Broneske et al. sind zur effizienten Nutzung von SIMD ratsam?

FF 2. Wie schneidet `Elf_SIMD` gegenüber der Ursprungsimplementierung bei der Indexerstellung und der Anfragebearbeitung ab?

Um SIMD im Elf nutzen zu können, bedurfte es unter anderem strutkureller Änderungen, welche in Abschnitt 3.2.2 erläutert wurden. Diese Anpassungen brachten als Nebenprodukte die zwei Elf-Varianten `Elf_Separated` und `Elf_Separated_Length` hervor. In `Elf_Separated` wurde der Elf von einer „Array of Structures“ in eine „Structure of Arrays“ überführt. Jenes Konstrukt diente wiederum als Basis für `Elf_Separated_Length`, mit welchem eine Abkehr von der impliziten Längenspeicherung über das Setzen des MSBs stattfand. Die Dimensionslistenlängen werden für SIMD explizit in der Indexstruktur speichert. Über die datenstrukturellen Anpassungen hinaus, mussten die drei zentralen Scan-Algorithmen des

Elf bezüglich einer vektorbasierten Arbeitsweise adaptiert werden. In Abschnitt 3.2.3 wurden erforderliche Maßnahmen dazu erläutert. Diese Änderungen sind zusammen mit den strukturellen Adaptationen als erster wesentlicher Beitrag dieser Arbeit zu sehen – sie beantworten die wissenschaftliche Fragestellung **FF** 1..
Mit dem Evaluationskapitel, Kapitel 4, wurde die zweite Forschungsfrage **FF** 2. beantwortet. Es wurde gezeigt, dass sich für die TPC-H-Anfragen, die in vorherigen, den Elf thematisierenden Arbeiten analysiert worden sind, Speedups von bis zu **1,65** bei der Anfragebearbeitung durch Elf_SIMD unter Verwendung von AVX2 erreichen lassen. Im Gegenzug ist zum einen der Trade-off eines im Schnitt 19% höheren Speicherverbrauchs zu verzeichnen. Zum anderen benötigt Elf_SIMD durchschnittlich 9% länger zur vollständigen Indexgenerierung für die Lineitem-Tabelle.
Wie in der Tabelle 6.1 ersichtlich wird, ist das Abschneiden der neuen Varianten bei den untersuchten TPC-H-Anfragen abhängig davon, ob die Hilfsstrutkur der Cutoffs genutzt wird oder nicht. Mit Cutoffs wird ein Absprung im Baum von höheren Ebenen direkt zu den ergebnisrelevanten TIDs geschaffen.

Hilfsstruktur	Durchschnittliche Speedups gegenüber Elf64		
Cutoffs	Elf_Separated	Elf_Separated_Length	Elf_SIMD
ohne	0,9874	1,2035	1,5784
mit	0,9420	1,0005	1,0743

Tabelle 6.1: Mittlere Speedups der Elf-Varianten gegenüber der Standardimplementierung Elf64 über alle getesteten TPC-H-Anfragen bei einer TPC-H-Größe von 100GB

6.2 Ausblick

Anhand unterschiedlicher Abschnitte dieser Arbeit lassen sich mögliche Themen für zukünftige Arbeiten ableiten. Diese sollen an dieser Stelle exemplarisch umrissen werden werden.

6.2.1 Nutzung von AVX-512

Es wäre interessant, zu untersuchen, wie sich der hier vorgestellte Ansatz Elf_SIMD unter Zuhilfenahme der neusten SIMD-Iteration, AVX-512, verhält. Mit dieser ist es möglich, die *doppelte* Anzahl an 32-Bit-Integerwerten im Vergleich zur hier gezeigten Umsetzung mit AVX2 in ein einzelnes SIMD-Register zu laden. Es könnten also 16 statt der aktuell maximal 8 Dimensions- respektive Monolistenwerte in ein Register geladen werden. Es ist vorstellbar, das davon insbesondere die Bearbeitung von Bereichsanfragen mit größeren Intervallen profitieren könnte. Darüber hinaus stehen mit AVX-512 weitere Funktionen zur Verfügung, die beispielsweise das zu vermeidende „Horizontalisieren“ weiter hinauszögern könnten. Es ist beispielsweise möglich, die Anzahl der gesetzten Bits über popcnt direkt im Register zu bestimmen.

6.2.2 Erweiterungen auf Softwareebene

Breitensuche-basierter Aufbau

Aktuell wird der Elf in Pre-Order-Manier aufgebaut. Eine weitere Form der Baumtraversierung ist die Breitensuche (engl. *breadth-first* oder *level-order*). Wie in Abschnitt 3.2.2.1 beschrieben, hätte ein Vorgehen in Form einer Breitensuche beim Aufbau des Elfs den Vorteil, dass bei der Verwendung von `Cutoffs` im `Cutoff_Pointers`-Array keine Lücken entstünden. Damit würde der Elf im `Cutoff`-Fall abhängig von der Höhe der `Cutoff`-Dimension weniger Speicherplatz als in der aktuellen Umsetzung benötigen. Des Weiteren wäre es eventuell möglich, SIMD auch bei potenziell kürzeren Dimensionslisten besser nutzen zu können, da alle Dimensionslisten einer Dimension „nebeneinander“ im `Elf`-Array lägen. So wäre es möglich, eigentlich erst später betrachtete Dimensionslisten bereits mit zu laden, gegen die Selektionsbedingungen zu prüfen und dadurch insgesamt SIMD-Register effizienter auszunutzen. Eventuell gilt es allerdings hierbei auch solche Vergleiche wieder zu verwerfen, wenn eine im Voraus abgeglichene Dimensionsliste durch ihre Elterliste disqualifiziert worden ist.

`Elf_SIMD` für Spaltenvergleiche

Eine für eine in Normalform vorliegende Datenbank unerlässliche Funktion für Anfragen ist der Verbund. Auch hierfür ist eine SIMD-basierte Optimierung vorstellbar. Den ersten Schritt dazu könnte eine Umsetzung eines vektorbasierten Scan-Algorithmus zum Vergleich zweier Spalten im Elf machen.

6.2.3 Weiterführende Evaluationen

In den Evaluationen dieser Arbeit wurde sich wegen der Vergleichbarkeit mit früheren Arbeiten auf die dort verwendeten TPC-H-Anfragen beschränkt. Wie die Sonderstellung von `Q19` gezeigt hat, kann die hinzugekommene Indirektion der Zeiger durch das Auffassen des Elf als „Structure of Arrays“ starke Negativauswirkungen auf die Performanz besitzen. Es wäre generell interessant, weitere Anfragen mit Selektionen auf im Elf eher „tiefer“ positionierten Dimensionen auszuwerten. Es wurde diesbezüglich während der Ergebnisinterpretation gemutmaßt, dass Anfragen mit sehr langen inneren Dimensionslisten und entsprechend breit gewählten Intervallbedingungen für `Elf_SIMD` vorteilhaft sein könnten. Eine Untersuchung dessen stünde noch aus.

Literaturverzeichnis

[BGGT02] Bik, Aart J. C. ; Girkar, Milind ; Grey, Paul M. ; Tian, Xinmin: Automatic Intra-Register Vectorization for the Intel® Architecture. In: *International Journal of Parallel Programming* 30 (2002), Nr. 2, S. 65–98. http://dx.doi.org/10.1023/A:1014230429447. – DOI 10.1023/A:1014230429447 (zitiert auf Seite 44)

[BKSS17] Broneske, David ; Köppen, Veit ; Saake, Gunter ; Schäler, Martin: Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2017, S. 647 – 658 (zitiert auf Seite iii, 2, 7, 12, 17, 19, 20, 79, 80, 95, 98 und 105)

[Bro19] Broneske, David: *Accelerating Mono and Multi-Column Selection Predicates in Modern Main-Memory Database Systems*, Otto-von-Guericke-Universität Magdeburg, Dissertation, 2019 (zitiert auf Seite 17, 23, 25, 26 und 101)

[CBB+05] Chatterjee, S. ; Bachega, L. R. ; Bergner, P. ; Dockser, K. A. ; Gunnels, J. A. ; Gupta, M. ; Gustavson, F. G. ; Lapkowski, C. A. ; Liu, G. K. ; Mendell, M. ; Nair, R. ; Wait, C. D. ; Ward, T. J. C. ; Wu, P.: Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. In: *IBM Journal of Research and Development* 49 (2005), Nr. 2, S. 377–391. http://dx.doi.org/10.1147/rd.492.0377. – DOI 10.1147/rd.492.0377 (zitiert auf Seite 44)

[CGS97] Culler, David E. ; Gupta, Anoop ; Singh, Jaswinder P.: *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997 (zitiert auf Seite 35)

[EFGL14] Estérie, Pierre ; Falcou, Joel ; Gaunard, Mathias ; Lapresté, Jean-Thierry: Boost.SIMD: Generic Programming for Portable SIMDization. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2014, S. 1–8 (zitiert auf Seite 42 und 44)

[FCP+12] Färber, Franz ; Cha, Sang K. ; Primsch, Jürgen ; Bornhövd, Christof ; Sigg, Stefan ; Lehner, Wolfgang: SAP HANA Database: Data Management for Modern Business Applications. In: *Proceedings of the International Conference on Management of Data (SIGMOD)* 40

(2012), Nr. 4, S. 45–51. http://dx.doi.org/10.1145/2094114.2094126. – DOI 10.1145/2094114.2094126 (zitiert auf Seite 1)

[Fly72] FLYNN, Michael J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* 21 (1972), Nr. 9, 948–960. http://dx.doi.org/10.1109/TC.1972.5009071 (zitiert auf Seite 35)

[GCB+97] GRAY, Jim ; CHAUDHURI, Surajit ; BOSWORTH, Adam ; LAYMAN, Andrew ; REICHART, Don ; VENKATRAO, Murali ; PELLOW, Frank ; PIRAHESH, Hamid: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In: *Data Mining and Knowledge Discovery* 1 (1997), Nr. 1, S. 29–53. http://dx.doi.org/10.1023/A:1009726021843. – DOI 10.1023/A:1009726021843 (zitiert auf Seite 11)

[HAMS08] HARIZOPOULOS, Stavros ; ABADI, Daniel J. ; MADDEN, Samuel ; STONEBRAKER, Michael: OLTP Through the Looking Glass, and What We Found There. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2008, S. 981–992 (zitiert auf Seite 1)

[HP11] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011 (zitiert auf Seite 42 und 44)

[HR01] HÄRDER, Theo ; RAHM, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer-Verlag, 2001 (zitiert auf Seite 6)

[IGN+12] IDREOS, Stratos ; GROFFEN, F. ; NES, Niels ; MANEGOLD, Stefan ; MULLENDER, Sjoerd ; KERSTEN, Martin: MonetDB: Two Decades of Research in Column-oriented Database Architectures. In: *IEEE Data Engineering Bulletin* 35 (2012), Nr. 01, S. 40–45 (zitiert auf Seite 1)

[Int16] INTEL CORPORATION (Hrsg.): *Intel 64 and IA-32 Architectures Optimization Reference Manual*. : Intel Corporation, 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (zitiert auf Seite 37 und 43)

[Int19] INTEL CORPORATION (Hrsg.): *Intel Intrinsics Guide*. : Intel Corporation, 2019. https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (zitiert auf Seite 42)

[Kar15] KARRENBERG, Ralf: *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer Vieweg, 2015 (zitiert auf Seite 42)

[KCS+10] KIM, Changkyu ; CHHUGANI, Jatin ; SATISH, Nadathur ; SEDLAR, Eric ; NGUYEN, Anthony D. ; KALDEWEY, Tim ; LEE, Victor W. ; BRANDT, Scott A. ; DUBEY, Pradeep: FAST: Fuast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2010, S. 339–350 (zitiert auf Seite 102)

[KSS14] Köppen, Veit ; Saake, Gunter ; Sattler, Kai-Uwe: *Data Warehouse Technologien, 2. Auflage.* mitp, 2014 (zitiert auf Seite 2, 10, 11, 12 und 13)

[LHW12] Leissa, Roland ; Hack, Sebastian ; Wald, Ingo: Extending a C-like Language for Portable SIMD Programming. In: *ACM SIGPLAN Notices* 47 (2012), Nr. 8, S. 65–74. http://dx.doi.org/10.1145/2370036.2145825. – DOI 10.1145/2370036.2145825 (zitiert auf Seite 42, 43 und 44)

[MAPK19] Mittal, Sparsh ; Anand, Osho ; P Kumarr, Visnu: A Survey on Evaluating and Optimizing Performance of Intel Xeon Phi. (2019). https://www.researchgate.net/publication/333384596 (zitiert auf Seite 43)

[MBK00] Manegold, Stefan ; Boncz, Peter A. ; Kersten, Martin L.: Optimizing Database Architecture for the New Bottleneck: Memory Access. In: *The VLDB Journal* 9 (2000), Nr. 3, S. 231–246. http://dx.doi.org/10.1007/s007780000031. – DOI 10.1007/s007780000031 (zitiert auf Seite 1)

[MBZ+14] Majeti, Deepak ; Barik, Rajkishore ; Zhao, Jisheng ; Grossman, Max ; Sarkar, Vivek: Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In: *Euro-Par 2013: Parallel Processing Workshops*, 2014, S. 188–197 (zitiert auf Seite 44)

[MGG+11] Maleki, Saeed ; Gao, Yaoqing ; Garzarán, Maria J. ; Wong, Tommy ; Padua, David A.: An Evaluation of Vectorizing Compilers. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, S. 372–382 (zitiert auf Seite 44 und 45)

[MMBS16] Majeti, Deepak ; Meel, Kuldeep S. ; Barik, Rajkishore ; Sarkar, Vivek: Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In: *Proceedings of the International Conference on Compiler Construction (CC)*, 2016, S. 240–250 (zitiert auf Seite 44)

[NH06] Nuzman, Dorit ; Henderson, Richard: Multi-platform Auto-vectorization. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006, S. 281–294 (zitiert auf Seite 44)

[NRZ06] Nuzman, Dorit ; Rosen, Ira ; Zaks, Ayal: Auto-vectorization of Interleaved Data for SIMD. In: *ACM SIGPLAN Notices* 41 (2006), Nr. 6, S. 132–143. http://dx.doi.org/10.1145/1133255.1133997. – DOI 10.1145/1133255.1133997 (zitiert auf Seite 42)

[PRR15] Polychroniou, Orestis ; Raghavan, Arun ; Ross, Kenneth A.: Rethinking SIMD Vectorization for In-Memory Databases. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015, S. 1493–1508 (zitiert auf Seite 38)

[PS16] PLOSKAS, Nikolaos ; SAMARAS, Nikolaos: *GPU Programming in MATLAB*. Morgan Kaufmann Publishers Inc., 2016 (zitiert auf Seite 36)

[RGB+15] RUCCI, Enzo ; GARCÍA, Carlos ; BOTELLA, Guillermo ; DE GIUSTI, Armando ; NAIOUF, Marcelo ; PRIETO-MATÍAS, Manuel: An Energy-aware Performance Analysis of SWIMM: Smith-Waterman Implementation on Intel's Multicore and Manycore Architectures. In: *Concurrency and Computation: Practice and Experience (CCPE)* 27 (2015), Nr. 18, S. 5517–5537. http://dx.doi.org/10.1002/cpe.3598. – DOI 10.1002/cpe.3598 (zitiert auf Seite 44)

[SDRK02] SISMANIS, Yannis ; DELIGIANNAKIS, Antonios ; ROUSSOPOULOS, Nick ; KOTIDIS, Yannis: Dwarf: Shrinking the PetaCube. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2002, S. 464–475 (zitiert auf Seite 11, 12 und 13)

[SGL09] SCHLEGEL, Benjamin ; GEMULLA, Rainer ; LEHNER, Wolfgang: K-ary Search on Modern Processors. In: *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2009, S. 52–60 (zitiert auf Seite 102)

[Shi07] SHIN, Jaewook: Introducing Control Flow into Vectorized Code. In: *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, S. 280–291 (zitiert auf Seite 44)

[SSH18] SAAKE, G. ; SATTLER, K.U. ; HEUER, A.: *Datenbanken - Konzepte und Sprachen*. mitp, 2018 (zitiert auf Seite 2, 5, 6 und 8)

[Tra18] TRANSACTION PROCESSING PERFORMANCE COUNCIL (Hrsg.): *TPC benchmark H (decision support)*. : Transaction Processing Performance Council, 2018. (2.18.0) (zitiert auf Seite xviii und 79)

[WEBS18] WALLEWEIN-EISING, Marten ; BRONESKE, David ; SAAKE, Gunter: SIMD Acceleration for Main-Memory Index Structures – A Survey. In: *Proceedings of the International Conference Beyond Databases, Architectures and Structures (BDAS)*, 2018, S. 105 – 119 (zitiert auf Seite 38)

[WPP04] W. P. PETERSEN, P. A.: *Introduction to parallel computing. A practical guide with examples in C*. Oxford University Press, 2004 (zitiert auf Seite 44)

[YOHY12] YAMAMURO, Takeshi ; ONIZUKA, Makoto ; HITAKA, Toshio ; YAMAMURO, Masashi: VAST-Tree: A Vector-advanced and Compressed Structure for Massive Data Tree Traversal. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2012, S. 396–407 (zitiert auf Seite 102)

[ZFH14] Zeuch, Steffen ; Freytag, Johann-Christoph ; Huber, Frank: Adapting Tree Structures for Processing with SIMD Instructions. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2014, S. 97–108 (zitiert auf Seite 101)

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

Magdeburg, den 06.12.2019

Kai Wolf