technische universität
dortmund

Bachelorarbeit

Schnittmengenberechnung auf Moderner Hardware (Implementierungstechniken)

Carsten Schulte
7. August 2014

Gutachter:
Prof. Dr. Jens Teubner
M. Sc. Sebastian Breß

Technische Universität Dortmund
Fakultät für Informatik
Datenbanken und Informationssysteme (LS-6)
http://dbis.cs.tu-dortmund.de/

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Hintergrund und Motivation 1

1.2 Ziele der Arbeit 2

1.3 Aufbau der Arbeit 2

2 Grundlagen **5**

2.1 Einleitung 5

2.2 Schnittmengenberechnung 6

2.3 Sort-Merge Intersection 7

2.3.1 Bitonic Sort 7

2.3.2 Berechnung der Überschneidungen 8

2.3.3 Hierarchische Überschneidungen 10

2.3.4 Datenlayout 11

2.3.5 Zusammenfassung 11

2.4 Hash-Intersection 12

2.4.1 Partitionierende Hash-Intersection 13

2.4.2 Parallele Hash-Intersection 14

2.4.3 Zusammenfassung 14

2.5 Radix-Cluster Algorithmus 15

2.5.1 Ablauf des Algorithmus 15

2.5.2 Paralleler Radix-Cluster Algorithmus 16

2.5.3 Zusammenfassung 16

2.6 Bit-Operationen 16

2.6.1 Umwandlung der relationaler Darstellung in einen Bitmap Index und zurück 18

2.6.2 WAH-Kompression 18

2.6.3 Zusammenfassung 19

2.7 Zusammenfassung 19

3 Implementierung **21**
3.1 Intel VTune Amplifier 21
3.2 Skalare Schnittmengenberechnung 23
3.3 Hashbasierte Schnittmengenberechnung 26
3.4 Bitvektorbasierte Schnittmengenberechnung 30
3.5 Zusammenfassung 33

4 Evaluation **35**
4.1 SIMD-basierte Schnittmengenberechnung 35
4.2 Radix-Cluster-Algorithmus zur Schnittmengenberechnung 38
4.3 Schnittmengenberechnung mit Hilfe von Bitvektoren 40
4.4 Vergleich der Algorithmen 46
4.5 Zusammenfassung 48

5 Zusammenfassung und Fazit **49**

A Weitere Informationen **51**

Abbildungsverzeichnis **55**

Algorithmen, Programmcode und SQL Anfragen **57**

Tabellenverzeichnis **59**

Literaturverzeichnis **64**

Erklärung **65**

Kapitel 1

Einleitung

In diesem Kapitel wird kurz das Thema der Arbeit beschrieben. Dazu wird der Hintergrund erläutert und die Arbeit motiviert. Darauf folgend werden die Ziele der Arbeit näher beschrieben. Den Abschluss bildet die Beschreibung des Aufbaus der Arbeit. Dazu wird auf den Inhalt der einzelnen Kapitel eingegangen.

1.1 Hintergrund und Motivation

Datenbanksysteme speichern und verwalten eine große Menge an Daten. Eine elementare Operation stellt dabei die Berechnung der Schnittmenge dar. Sie findet sich im Kontext verschiedener Nutzungen von Datenbanken wie das Data Mining, die Textanalyse, die Auswertung konjunktiver Anfragen und der Websuche wieder [DK11]. Die Schnittmengenberechnung verbindet zwei oder mehr Teilanfragen, welche zur Beantwortung der eigentlichen Anfrage notwendig sind.

Listing 1.1: Beispielhafte Anfrage

```
SELECT * FROM R WHERE R.a > 42 AND R.b > 17;
```

Listing 1.1 stellt eine konjunktive Anfrage dar. Zunächst werden die Anfragen der Teilbedingungen 'R.a > 42' und 'R.b > 17' ausgewertet. Dabei werden zwei Listen von Tupel-Identifikatoren gebildet. Anschließend wird eine Liste, die den Schnitt der Listen von record ids (RIDs) enthält, gebildet. Da die Schnittmenge auf meist sortierten Listen erfolgt, hat sich der Ansatz des sortierten Mischen (merging) im traditionellem Datenbankumfeld durchgesetzt. Online Analytical Processing (OLAP) nutzt eine andere Technik zur Berechnung von Schnittmengen. Dieses liegt in der Datenhaltung begründet. OLAP-Systeme speichern RID-Listen als Bitvektoren, bei denen RIDs einzelnen Bits zugeordnet sind. OLAP-Systeme beschäftigen sich mit der Analyse von Daten. Dazu werden komplexe Anfragen ausgewertet. Die Beantwortung solcher Anfragen, sei es im klassichen Datenbankumfeld oder in OLAP-Systemen, benötigt längere Zeit. Im Bereich des Information

Retrieval greift man auf posting lists zurück. Diese enthalten Einträge mit Referenzen zu Dokumenten. Die Berechnung der Schnittmenge wird von den Daten und der Hardware beeinflusst. Unterschiedliche Hardwarecharakteristika führen zu unterschiedlichen Algorithmen zur effizienten Bewältigung der Aufgaben. Unter den verschiedenen Charakteristika finden sich Multicore-Systeme wieder, bei denen ein Kern die Ausführung mehrere Threads ermöglicht. Dieses wird als Simultaneous Multithreading (SMT) [WOMF13] oder bei Intel als Hyper-Threading (HT) Technology [Inta] bezeichnet. Dazu kommen sich ständig erweiternde Hardwareergänzungen wie AVX [WOMF13] oder SIMD [ZR02]. Außerdem unterscheiden sich die genutzten Algorithmen in den jeweiligen Teilgebieten. Eine Untersuchung der genutzten Ansätze und die Übertragbarkeit dieser stellt eine interessante Frage dar.

1.2 Ziele der Arbeit

Eigenschaften moderner Hardware haben signifikante Auswirkungen auf die Laufzeit von Algorithmen. Daher stellt sich die Frage, wie bestehende Lösungsansätze effizient auf konkreter Hardware implementiert werden können. Auswirkungen der Hardware auf die Laufzeit der Algorithmen ergeben sich durch die Geschwindigkeit des Hauptspeichers, die Cacheeffizenz oder die Ausnutzung der parallelen Verarbeitung innerhalb eines Multithreading-Systems. Ziel der Arbeit ist es, bestehende Algorithmen möglichst hardware-effizient zu implementieren und diese anschließend zu evaluieren. Die Daten für eine Evaluation werden durch einen geeigneten Benchmark zur Verfügung gestellt. Dieser wird parallel zu dieser Arbeit entwickelt. Am Ende sollen Empfehlungen für Algorithmen ausgesprochen werden und Favoriten für Anwendungsfälle herausgearbeitet werden.

1.3 Aufbau der Arbeit

Nach einer kurzen Einführung in das Thema und der Erläuterung der Ziele dieser Bachelorarbeit wird im nächsten Kapitel auf die Schnittmengenberechnung genauer eingegangen. Diese wird anschaulich erklärt und in Bezug zu Datenbanken gesetzt. Anschließend erfolgt die Darstellung verschiedener Algorithmen zur Berechnung des Schnittes. Zuerst wird auf das klassische Verfahren, das sortierte Mischen, eingegangen und erläutert, inwiefern SIMD-Befehle bei der Berechnung helfen. Daraufhin werden hashbasierte Algorithmen, wie der Radix-Cluster Algorithmus, vorgestellt. Das Kapitel endet mit der Vorstellung von bitmapbasierten Verfahren. Darunter befindet sich ein Verfahren zur Komprimierung. Im darauf folgenden Kapitel wird auf die konkrete Implementierung der vorgestellten Algorithmen eingegangen. Dazu wird zunächst ein Programm zur Performancemessung vorgesetellt. Anschließend wird auf die Schnittmengenberechnung nach dem Vorbild des sortierten Mischen eingegangen. Darauf folgt der Radix-Cluster Algorithmus und seine

speziellen Eigenschaften bzgl. der Hardware. Den Abschluss bilden die Algorithmen auf Basis von Bitmaps.
Das Kapitel 4 beschäftigt sich mit der Evaluation und Bewertung der implementierten Algorithmen und zeigt Eigenheiten dieser auf. Dazu werden die Implementationen miteinander verglichen, die Auswirkungen von unterschiedlichen Eingabedaten beobachtet und untersucht.
Den Abschluss bildet eine Zusammenfassung. Diese lässt die vorherigen Kapitel Revue passieren und fasst die gewonnenen Erkenntnisse zusammen. Außerdem wird ein Ausblick auf mögliche neue Fragestellungen gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel wird zunächst die Wichtigkeit der Schnittmengenberechnung im Datenbankumfeld erläutert. Anschließend werden verschiedene Algorithmen zur Berechnung vorgestellt. Dabei wird auf die jeweilig verwendeten Techniken eingegangen.

2.1 Einleitung

Datenbanken helfen bei der Verwaltung von Daten, der Informationsgewinnung und der Darstellung dieser. Eine häufig anzutreffende Datenbankform ist die relationale Datenbank. Diese basiert auf Relationen, deren Semantik definiert wird durch die relationale Algebra von E. F. Codd [Cod70]. Relationen enthalten dabei gleichartige Daten die in Records abgespeichert werden. Abbildung 2.1 stellt dabei eine Ansammlung von Daten

Person	id	Name	Vorname	Alter	Geschlecht
	1	Meier	Anton	25	m
	2	Schmidt	Maria	36	w
	3	Schneider	Otto	47	m
	4	Fischer	Heinz	65	m
	5	Müller	Sebastian	29	m
	6	Meyer	Paula	18	w
	7	Wagner	Stefanie	16	w
	8	Engel	Anna	32	w
	9	Voigt	Dennis	23	m
	10	Pfeiffer	Tristan	35	m
	11	Schmitz	Nicole	35	w

Abbildung 2.1: Beispielhafte Darstellung einer Relation

von Personen dar. Zu jeder Personen sind neben Vor- und Nachname, das Alter sowie das Geschlecht vermerkt. Jeder Eintrag enthält eine eindeutige ID, die sogenannte Record-ID, kurz RID. Mit Hilfe der Datenbanksprache SQL (Structured Query Language) können

Daten innerhalb einer Datenbank definiert, manipuliert, abgefragt und geschützt werden [Ste14]. In dieser Arbeit sind Datenbankabfragen der folgenden Form im Fokus.

Listing 2.1: Anfrage an Relation Person

```
SELECT * FROM Person WHERE Alter < 40 AND Geschlecht = m;
```

Listing 2.1 stellt eine Anfrage dar, bei der alle männlichen Personen unter 40 Jahren zurückgeliefert werden. Bei der Beantwortung der Anfrage wird der Schnitt der Mengen der Personen unter 40 Jahren und der männlichen Personen berechnet. Zur Berechnung werden zuerst die RID Listen aufgestellt. Dazu wird über die Relation Person iteriert und zunächst alle RIDs heraus gefiltert, die die Bedingung Alter kleiner als 40 Jahre erfüllen. Der Vorgang wiederholt sich analog mit der zweiten Bedingung. Daraus ergeben sich die Mengen $Alter = \{1, 2, 5, 6, 7, 8, 9, 10, 11\}$ und $Geschlecht = \{1, 3, 4, 5, 9, 10\}$. Aus diesen beiden Mengen wird nun durch die Vereinigungsoperation die Schnittmenge berechnet.

$$Alter \cap Geschlecht = \{1, 5, 9, 10\}$$

Das Ergebnis der Operation wird durch Abbildung 2.2 visualisiert.

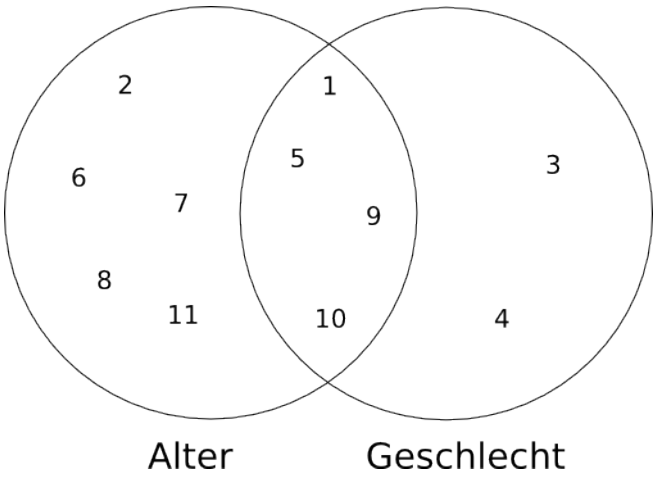

Abbildung 2.2: Darstellung der Schnittmenge durch die Anfrage 2.1

2.2 Schnittmengenberechnung

Zur Berechnung der Schnittmengen von RID Listen können Join-Algorithmen, die ähnliche Probleme lösen, betrachtet werden. Bei der Verbund-Operation handelt es sich in relationalen Datenbanken um eine viel genutzten Operation, sodass eine effiziente Implementierung dieser Algorithmen wichtig und erstrebenswert ist. Einige wichtige Algorithmen zur Berechnung sind der Nested-Loop-Join, Sort-Merge-Join oder der Hash-Join Algorithmus [Vos08]. Eine andere Form der Berechnung von Überschneidungen erfolgt mit

Hilfe von Bitmap Indizes, bei denen die jeweiligen Bits einer Record-ID zugeordnet sind. Die Schnittmenge kann so durch eine schnelle bit-weise AND-Operation zwischen zwei Vektoren berechnet werden. Außerdem ist durch die Nutzung von Bitmaps eine kompakte Repräsentation möglich, welche durch Kompression noch erhöht werden kann [Joh99]. Eine bekannte Form der Kompression stellt die Word-Aligned Hybrid (WAH)-Kompression dar. Dabei werden die Daten so gespeichert, dass wenig CPU Aufwand zum komprimieren und entpacken notwendig ist und die typische Wortlänge des Computersystems berücksichtigt wird [WSO04]. Die folgenden Kapitel stellen die verschiedenen Algorithmen im Einzelnen vor und erläutern die Besonderheiten.

2.3 Sort-Merge Intersection

Dieser Algorithmus lässt sich in zwei Phasen, das Sortieren und das "Mischen", aufteilen. In der ersten Phase werden die Werte der Verbund-Attribute, hier RIDs, sortiert. Die Sortierung kann ab- und aufsteigend erfolgen. Nach dem Sortieren werden beide Relationen sequentiell durchlaufen. Alle sich überschneidenden Werte werden in einer Ergebnisrelation zusammengefasst. Sollten die Verbund-Relationen bereits sortiert sein, so entfällt der Sortieraufwand und jede Relation wird nur einmal durchlaufen, woraus ein linearer Aufwand resultiert [Vos08]. Der hier verwendete und im folgenden dargestellte Algorithmus geht auf die Ergebnisse von [SWL11] zurück. Dieser versucht, die Ausführungszeit bei der Berechnung von Überschneidungen durch die Erhöhung der Parallelität auf Datenebene zu reduzieren. Hierfür werden spezielle SIMD Instruktionen verwendet, die einen kompletten Vergleich von acht 16-bit Werten oder sechzehn 8-bit Werten zulassen. Zur Ausführung werden sortierte Listen ohne Duplikate im Hauptspeicher des Computers vorausgesetzt. SIMD steht für Single Instruction Multiple Data und gewährleistet die Ausführung einer Rechenoperation zur gleichen Zeit auf mehreren Daten. Die Berechnung findet dabei auf einer CPU statt und das Ergebnis wird nicht durch die Ausnutzung mehrerer CPUs bestimmt. Intel stellt verschiedene SIMD Instruktionen zur Verfügung. Heute gibt es verschiedene Instruktionssätze für x86-Prozessoren wie MMX, SSE oder AVX [Intb].

2.3.1 Bitonic Sort

Zum Sortieren kann ein Sortierungsnetzwerk genutzt werden [Bat68]. Dabei erfolgen alle Vergleiche ohne unvorhersehbare Verzweigungen und mehrere Vergleiche innerhalb eines CPU-Zyklus. Dieses erlaubt nach Chhungani et al. [CNL+08] die Ausnutzung von SIMD Instruktionen. Bei der Berechnung bitonischer Teilfolgen bedient man sich dem Divide-and-Conquer-Prinzips. Es werden Vergleichsnetzwerke zum Sortieren und Mischen (Merge) konstruiert. Um eine Sequenz von je 4 sortierten Elementen zu mischen, wird die erste Eingabe in aufsteigender Reihenfolge erwartet und die zweite Eingabe in absteigender Reihenfolge. Eine solche Sequenz nennt sich *bitonic*. Abildung 2.3 veranschaulicht den

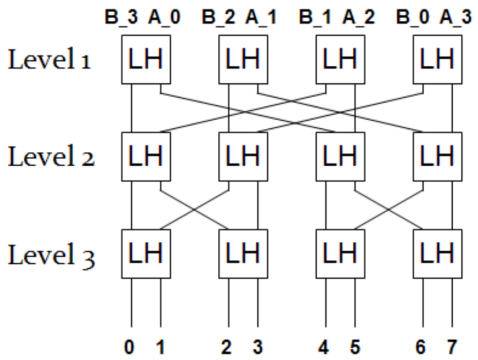

Abbildung 2.3: Bitonic Merge Netzwerk für eine Eingabe mit jeweils 4 Elementen [CNL+08].

Vorgang. Die Elemente werden mit Hilfe von Minimum- und Maximum-Operatoren parallel verglichen. Dabei landet das kleinere Element im L Register und das größere Element im H Register. Nach dem Vergleich werden die Daten entsprechend verschoben, damit im nächsten Level der nächste Vergleichszyklus starten kann. Dieses wird solange fortgesetzt, bis man die finale Sequenz als Ausgabe erhält. Um die Parallelität des Algorithmus zu verbessern, kann die Breite des Netzwerkes angepasst werden. So sind nicht nur 4x4 Netzwerke, wie in Abbildung 2.3 dargestellt, sondern auch 2x2, 8x8 oder größere Netzwerke möglich..

2.3.2 Berechnung der Überschneidungen

Als Eingabe erhält der Algorithmus aus Listing 2.2 neben den beiden sortierten Listen A und B, die dazugehörigen Längen und eine Ausgabeliste, die maximal so groß ist, wie die kleinste Liste von A und B, da es auch nur so viele mögliche Überschneidungen geben kann.

Listing 2.2: Paralleler Intersection-Algorithmus aus [SWL11]

```
1  int intersect(short *A, short *B, int l_a, int l_b, short* C) {
2    int count = 0;
3    short i_a = 0, i_b = 0;
4    while(i_a < l_a && i_b < l_b) {
5      // 1. Werte laden
6      __m128i v_a = _mm_load_si128((__m128i*)&A[i_a]);
7      __m128i v_b = _mm_load_si128((__m128i*)&B[i_b]);
8      // 2. Vergleich
9      __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8, _SIDD_UWORD_OPS|
           _SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
10     int r = _mm_extract_epi32(res_v, 0);
11     unsigned short a7 = _mm_extract_epi32(v_a, 7);
12     unsigned short b7 = _mm_extract_epi32(v_b, 7);
13     l_a += ( a7 <= b7 ) * 8;
```

```
14      l_b += ( a7 >= b7 ) * 8;
15      // Werte zurueckschreiben
16      __m128i p = _mm_shuffle_epi8(v_a, sh_mask[r]);
17      _mm_storeu_si128((__m128i*)&C[count], p);
18      count += _mm_popcnt_u32(r);
19    }
20    return count;
21  }
```

In Zeile 3 und 4 werden die Hilfsvariablen initialisiert. Danach startet eine While-Schleife, die erst beim Erreichen eines Vielfachen von Acht terminiert. Der Rest der Überschneidungen beider Listen wird durch ein sequentielles Ablaufen berechnet. Ein Vielfaches von Acht muss auf Grund der Länge eines Shorts (16 Bits) gewählt werden, da ein SIMD-Vektor eine Länge von 128 Bits verwendet. In Zeile 7 und 8 werden zwei Vektoren, v_a und v_b, durch eine load-Instruktion mit Werten befüllt. Anschließend erfolgt die Erstellung einer Maske. Innerhalb der Maske werden binär die Werte des Vektors v_a gespeichert, die einen entsprechenden Wert in Vektor v_b haben. Dieses wird durch Abbildung 2.4 visualisiert. Anschließend werden die größten Werte aus den beiden Vektoren entnom-

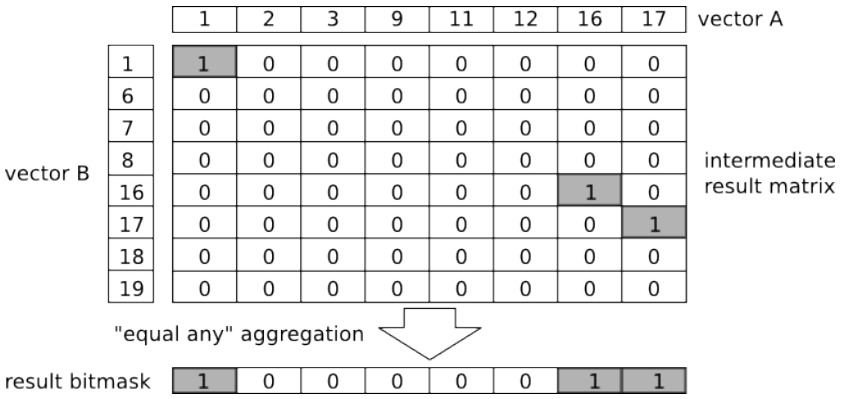

Abbildung 2.4: Voller Vergleich mit Hilfe der “equal any“ Methode. Der erste, siebte und achte Wert sind sowohl in Vektor A als auch in Vektor B zu finden und werden durch die resultierende Bitmaske indiziert [SWL11].

men und miteinander verglichen. Durch den Vergleich werden die Zeiger der Arrays in Zeile 13 und 14 angepasst. Wenn beide Werte der Vektoren gleich sind, so werden auch beide Zeiger um Acht erhöht. Sollten sie allerdings ungleich sein, so verschiebt sich nur der Zeiger des Arrays mit dem kleineren Wert. Dadurch wird sichergestellt, dass kein Wert ausgelassen werden kann. Am Ende werden die Ergebnisse aus dem Vergleich mit Hilfe des `_mm_shuffle_epi8`-Befehles und einem Vektor zur Sortierung in die richtige Ausgabereihenfolge gebracht und abgespeichert. Der Vektor zum Sortieren wird einer look-up-Tabelle entnommen und kann dadurch effizient realisiert werden. Daraufhin wird der Counter in Zeile 18 erhöht, indem die auf 1 gesetzten Bits der Maske gezählt und auf

den Wert des Counters addiert werden. Dieser bestimmt den Ort innerhalb des Arrays, in das die Ausgabe erfolgt.
Im Unterschied zur skalaren Berechnung der Überschneidungen benötigt der Algorithmus von [SWL11] bei der Ausführung mehr Vergleiche. Im besten Fall bei identischer Länge l und einer durch 8 teilbaren Anzahl an Elementen der Arrays werden $8 * l$ Vergleiche benötigt. Daraufhin schreitet der Algorithmus um 8 Werte der Arrays weiter. Schlegel et al. nennen für den schlechtesten Fall $16 * l - 64$ Vergleiche. Hierbei sind die letzten Werte immer unterschiedlich, sodass nur in einem Array weitergegangen werden kann. Eine 4- bis 8-fach schnellere Ausführung wäre unter der unrealistischen Annahme, dass die CPU die Maske in derselben Zeit wie einen einzigen Vergleich bei einem skalaren Überschneiden berechnet, möglich. Der Ablauf des Algorithmus wird durch die Abbildung A.1 visualisiert.

2.3.3 Hierarchische Überschneidungen

Da die o. g. Methode nur in der Lage ist, mit 8- oder 16-bit Integerwerten zu rechnen, stellt [SWL11] ein Verfahren vor, mit dessen Hilfe es möglich ist, Werte aus einem größerem Wertebereich (>16-bit) zu nutzen. Dazu werden die Eigenschaften der binären Darstellung genutzt und die Integerwerte in höher- und niederwertige Bits unterteilt. Dabei teilen sich die Werte mit denselben höherwertigen Bits eine Domäne. Zur Vorbereitung müssen alle gegebenen Werte in disjunkte Mengen unterteilt werden. Die Domänen enthalten die niederwertigen Bits. Bei der Ausführung wird die Schnittmenge aller Paare der Menge mit

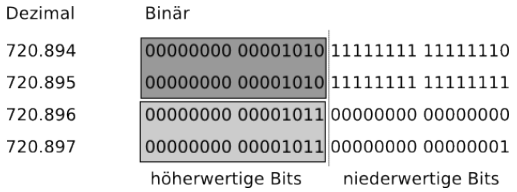

Abbildung 2.5: Abbildung visualisiert die binäre Darstellung von Zahlenwerten und deren Unterteilung in höher- und niederwertige Bits.

denselben höherwertigen Bits berechnet. Die so entstandenen Mengen werden nun, sollten in beiden Wertebereichen dieselbe Domäne vorliegen, wie in Abschnitt 2.3.2 beschrieben, berechnet. Die Berechnung der Überschneidungsmenge der höherwertigen Bits kann mit demselben Algorithmus erfolgen oder aber durch einen einfachen, skalaren Algorithmus realisiert werden. Schlegel et al. gibt eine geringe Auswirkung auf das Laufzeitverhalten durch die Mehrarbeit bei der Bildung der disjunkten Mengen an [SWL11]. Das Laufzeitverhalten hängt direkt von dem Grad an Überschneidungen innerhalb der niederwertigen Bitmenge ab. Anschließend werden die sechzehn höher- und niederwertigen Bits konkateniert. Die Beschreibung der zu leistenden Mehrarbeit vor und nach der Ausführung des Algorithmus sowie die Speicherung der Daten wird in Abschnitt 2.3.4 beschrieben.

2.3.4 Datenlayout

Eine generelle Unterteilung der Daten in zwei Level wird als Hauptidee zur Verringerung der Mehrarbeit in [SWL11] genannt. Dabei werden die 16-Bitwerte innerhalb eines durchgängigen Speicherbereiches vorgehalten. Die unterteilten Mengen $A_1, A_2, ..., A_n$ von A werden folgendermaßen abgespeichert. Zuerst wird der Wert für die höherwertigen Bits gespeichert und nachfolgend wird die Anzahl der Elemente innerhalb der Teilmenge gespeichert. Beide Informationen sind jeweils 16-Bit groß, da die Anzahl der Elemente innerhalb der Teilmenge maximal 65536 ($=2^{16}$) beträgt. In dem darauf folgenden Speicherbereich befinden sich die zugehörigen niederwertigen Bits. Abbildung 2.6 zeigt die Speicherung der Mengen in einem Array. Die Vorhaltung der Daten mit Hilfe des zwei geteilten Da-

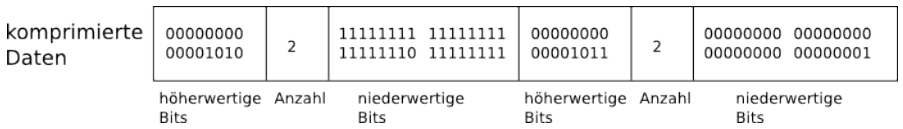

Abbildung 2.6: Abbildung visualisiert die Speicherung der Dezimalzahlen aus Abbildung 2.5 unterteilt in disjunkte Mengen

tenlayouts spart häufig Speicherplatz, da die Daten durch die einmalige Speicherung der höherwertigen Bits komprimiert werden. Im schlechtesten Fall würde jede Teilmenge nur ein Element enthalten, wodurch zusätzlich 16-Bit für die Speicherung der Länge anfallen. In realen Umgebungen ist dieses Szenario ungewöhnlich. Schon bei einer durchschnittlichen Anzahl von zwei Elementen innerhalb der Teilmengen gleicht sich der Speichermehraufwand aus. Nach der Berechnung der Schnittmenge werden allen Werten ihre zugehörigen höherwertigen Bits zugeordnet. Somit erfolgt die Ausgabe einer mit 32-bit Werten gefüllten RID Liste. Das hierarchische Speichermodell lässt sich auch auf größere bit-Werte anwenden. Für weitere 16-bit muss jeweils eine neue Ebene angelegt werden. In Zukunft kann mit einer noch größeren Datenparallelität gerechnet werden, da SIMD Instruktionssätze immer mehr Daten parallel verarbeiten können. Die passenden Instruktionen sind allerdings Voraussetzung zur Umsetzung dieses Algorithmus. Momentan ist man bei dem Vergleich, siehe Zeile 9 in Listing 2.2, auf 128-bit Vektoren beschränkt. Eine Instruktion die auf 256-bits aufbaut ist nicht verfügbar.

2.3.5 Zusammenfassung

In diesem Abschnitt wurde ein skalarer Algorithmus zur Berechnung von Schnittmengen von [SWL11] vorgestellt. Die Besonderheit liegt in der Art und Weise des Vergleichens. Im Gegensatz zum üblichen vorgehen, bei denen immer ein Paar verglichen wird, werden jeweils acht Werte miteinander verglichen. Bei der Verwendung von 32-bit Integerwerten ist es notwendige, diese in höher- und niederwertige Bits aufzuteilen.

2.4 Hash-Intersection

Bei der Berechnung der Schnittmenge zweier Relationen R und S wird auf die Reduzierung der Anzahl von Vergleichen geachtet. Der Hash-Join Algorithmus realisiert dieses durch die Bildung von Verbund-Partnern. Dieser Vorgang lässt sich auf die Berechnung von Schnittmengen adaptieren. Zur Bildung von Verbund-Partnern wird die erste Relation durchlaufen und die möglicherweise passenden Tupel heraus gefiltert. Anschließend wird die zweite Relation durchlaufen und die Ergebnisse mit der vorherigen, zugehörigen Menge verglichen [Vos08]. Der Ablauf des Algorithmus wird durch folgenden Pseudo-Code spezifiziert:

```
1 foreach tuple s ∈ S do
2 |   Füge s in Hash-Tabelle H ein;
3 end
4 foreach tuple r ∈ R do
5 |   untersuche H und füge passende Tupels zum Resultat hinzu;
6 end
```

Algorithmus 2.3 : Einfacher Hash-Join über die Relationen R und S

Um den Verbund R ⋈ S zu berechnen, findet zunächst die Bildungsphase (build) statt. Bei dieser wird eine Hash-Tabelle angelegt und die innere Join Relation S durchlaufen. Dabei werden alle Elemente aus S in die Hash-Tabelle eingeordnet. Diese besteht aus mehreren Teilbereichen, den sogenannten Buckets, welche alle Elemente mit demselben Hashwert beinhalten. Die Untersuchungsphase (probe) beginnt nach Abschluss der ersten Phase. Hierbei wird die gesamte Relation R durchlaufen. Für jedes Element wird in dem zugehörigen Teilbereich der Hash-Tabelle nach Äquivalenten gesucht. Wenn diese gefunden sind, so werden diese an das Ergebnis angefügt. Für die Schnittmengenberechnung bedeutet dieses, dass der entsprechende Teilbereich durchlaufen wird und bei einer Übereinstimmung das Ergebnis herausgeschrieben wird. Anschließend wird mit dem nächsten Element aus r fortgefahren, also nicht der gesamte Teilbereich durchlaufen. Die Abbildung 2.7 veranschaulicht den Ablauf. Der Algorithmus lässt sich parallelisieren, indem die Relationen R und S unterteilt werden. Jeder Teilbereich wird einzeln durchlaufen. Dabei wird zur Berechnung der Ablage der Elemente immer dieselbe Hash-Funktion genutzt. Durch die Parallelisierung ergeben sich Probleme beim Zugriff auf die gemeinsam genutzten Teilbereiche der Hash-Tabelle. Um den Zugriff zu gewähren, werden Sperren, sogenannte Locks, genutzt. Dieses Verfahren bietet durch die Sperren nur ein geringes Maß an Nebenläufigkeit. Der ständige, zufällige Zugriff auf die Daten innerhalb der gemeinsam genutzten Hash-Tabelle zieht jeweils einen Cache miss[1] mit sich. Damit ist die Laufzeit

[1] Tritt auf, wenn angefragtes Objekt nicht im Cache enthalten ist.

des Hash-Joins von den Latenzen des Speichers abhängig. In Kapitel 2.4.1 werden die Probleme bei der Nebenläufigkeit mit Hilfe von Partitionen überwunden.

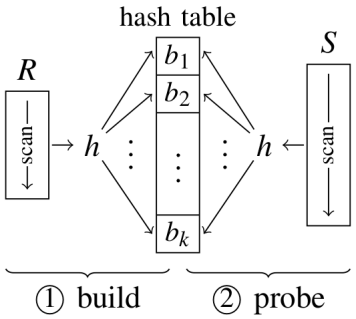

Abbildung 2.7: Hash-Join über die Relationen R und S aus [BTAO13]

2.4.1 Partitionierende Hash-Intersection

Um die Performance des Hash-Joins zu erhöhen, können die Relationen R und S mit Hilfe einer Hashfunktion geteilt werden. Die ergänzende Partitionierungsphase verkleinert beide Relationen möglichst so, dass diese in den CPU Cache passen. Dadurch werden die Cache misses während der Bildungs- und Untersuchungsphase reduziert. Der Mehraufwand für das Schreiben in den Speicher während der Partitionierungsphase ist nach [SKN94] geringer als die Kosten für das nicht Antreffen der Daten im Cache. Durch die Partitionierung entfällt die Notwendigkeit, Sperren während der parallelen Ausführung zu nutzen. Jede Partition wird einem einzelnen Thread zugeordnet und da jede Partition ihre eigene Hash-Tabelle nutzt, ist der Zugriff auf diese nicht mehr kritisch. Abbildung 2.8 veranschaulicht den Hash-Join mit vorhergehender Partitionierungsphase. Das Vermeiden von Cache mis-

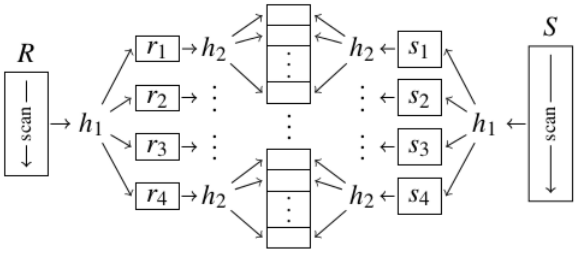

Abbildung 2.8: Partitionierender Hash-Join über die Relationen R und S aus [BTAO13]

ses während der Bildungs- und Untersuchungsphase mit Hilfe der Partitionierung zieht ein neues Problem bzgl. des Caches mit sich. Manegold et al. macht den zufälligen Zugriff auf die Daten verantwortlich [MBK02]. Dieser resultiert aus den vielen, verschiedenen Hash-Tabellen. Wenn die Anzahl der Hash-Tabellen zu groß wird und die Anzahl der TLB[2]

[2]Ein Translation Lookaside Buffer ist der Cache von Seitentabelleneinträgen

Einträge übersteigt, hat dieses für jede angefragte Speicherreferenz einen TLB miss zur Folge. Ein weiteres Problem ergibt sich aus der Überschreitung der verfügbaren Cache lines. Die Überlastung des Caches führt zu einer signifikanten Steigerung der Anzahl an Cache misses und wirkt sich damit negativ auf die Performance des Algorithmus aus.

2.4.2 Parallele Hash-Intersection

Die Ausführung des Hash-Intersection Algorithmus kann durch Ausnutzung mehrere CPU Kerne beschleunigt werden. Wie in Abschnitt 2.4.1 gesehen, entfällt nach der Partitionierung die Notwendigkeit Sperren zu nutzen. Kim et al. stellt ein Verfahren zur Erhöhung der Parallelität während der Partitionierungsphase vor [KKL+09]. Dabei werden zu Beginn die beiden Eingaberelationen in n Teilrelationen aufgeteilt. Die Anzahl der Teilrelation sollte die Anzahl der CPU Kerne nicht überschreiten. Anschließend wird für jede Teilrelation ein lokales Histogramm errechnet. Sind alle lokalen Histogramme bestimmt, kann anhand dieser lokalen Histogramme die endgültige Einfügeposition für die Tupel errechnet werden. Die Position an der Stelle j berechnet sich aus der Summe der vorherigen Positionen. Abbildung 2.9 veranschaulicht den Berechnungsvorgang. Durch die vorbestimmte Einfügeposition entfällt die Notwendigkeit, Sperren für den Zugriff auf die Datenstruktur zu verwenden. Nach der Berechnung der Histogramme können die Partitionen parallel gebildet werden und die Werte der Teilrelationen gemäß ihrem Hashwert an der richtigen Stelle gespeichert werden. Analog zum partitionierendem Hash-Join kann auch der parallele Hash-Join zur Schnittmengenberechnung genutzt werden.

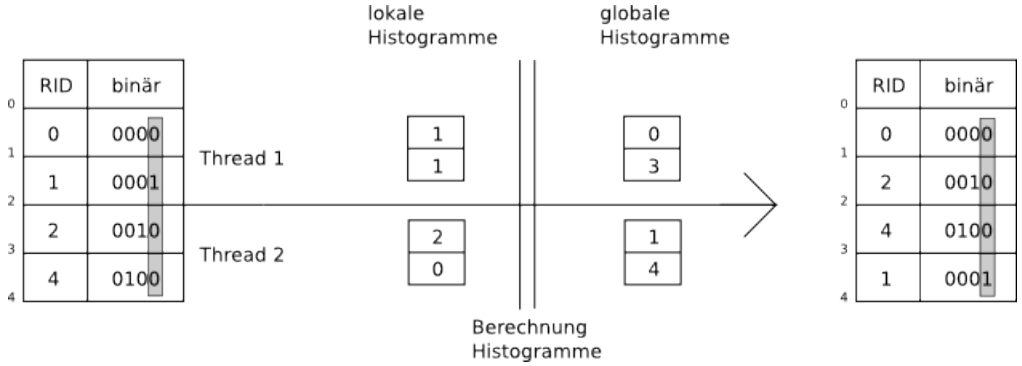

Abbildung 2.9: Parallele Partitionierung mit Berechnung der Histogramme

2.4.3 Zusammenfassung

In diesem Kapitel wurde der Hash-Intersection Algorithmus besprochen, um Schnittmengen zu berechnen. Durch die Ausnutzung von Parallelität lässt sich die Ausführungsgeschwindigkeit erhöhen. Dazu partitioniert man die Eingaberelationen zunächst mit einer Hashfunktion und berechnet anschließend mit einer zweiten Hashfunktion die einzelnen

Hashtabellen. Außerdem kann durch geschickte Implementierung der Partitionierungsphase die Zeit zur Partitionierung verringert werden.

2.5 Radix-Cluster Algorithmus

Die o. g. Probleme führen zu dem Radix-Cluster Algorithmus von [MBK02]. Beim Entwurf des Algorithmus wurde sowohl auf die Verbesserung der Cacheeffizienz als auch auf die Optimierung des Codes geachtet. Dazu wird auf die Potentiale moderner CPUs bzgl. ihrer Parallelität eingegangen. Es werden Techniken wie *branch prediction*, die Voraussage, welche Instruktion nach der Fertigstellung der gerade ausgeführten benötigt wird, und die nicht blockierende Eigenschaft des Caches genannt. Unter der nicht blockierenden Eigenschaft des Cachespeichers versteht man, dass selbst ein Cache miss die Ausführungen auf der CPU nicht blockieren. Außerdem geht [MBK02] auf die verschiedenen Speicher eines Computersystems ein und benennt die einzelnen Kosten für einen Cache miss. Die als wichtig erachteten Parameter zur Steigerung der Performance bestehen wesentlich aus der Anzahl der Cluster, die Anzahl der Durchgänge, die zur Bildung der Cluster notwendig sind, und die während des Cluster-Prozesses genutzten Bits. Diese werden auch als Radix-Bits bezeichnet, dazu mehr in Kapitel 2.5.1.

2.5.1 Ablauf des Algorithmus

Der Radix-Cluster Algorithmus aus [MBK02] unterteilt eine Relation in mehreren Schritten zu einer Anzahl H von Clustern. Bis es zu einer Tabelle mit Werten kommt, bei denen der Hashwert nur noch von dem niederwertigsten Bit abhängt, ist es notwendig, die Teilungsphase mehrfach hintereinander zu durchlaufen. In jedem Durchlauf werden Tupel anhand der Werte der B_p bits geclustert. Das Verfahren startet mit dem hochwertigsten Bit. Die Anzahl der dabei gebildeten Cluster beschreibt die Formel $H = \prod_{1}^{p} H_p$ mit $H_p = 2^{B_p}$. H_p beschreibt dabei die Anzahl der neu entstehenden Cluster während der Teilungsphase. Sollte $P = 1$ gewählt werden, verhält sich der Algorithmus analog zu dem Algorithmus in 2.4. Wählt man B_p und P so, dass die Größe der Hash-Tabellen kleiner als die Anzahl der Cache lines und die Anzahl der TLB-Einträge ist, lässt sich die Überlastung dieser verhindern. Abbildung 2.10 zeigt den Ablauf des Algorithmus mit zwei Partitionierungsphasen. Bei der Speicherung der Cluster ist es nicht notwendig, die Grenzen zwischen diesen in einer ergänzenden Struktur zu speichern, da die Relation beim bilden der Cluster von den zugehörigen Radix-Bits abhängen, reicht es diese sequentiell zu durchlaufen und auf die zugehörigen Radix-Bits zu achten. Dadurch können die Grenzen zwischen den einzelnen Clustern ohne zusätzliche Struktur detektiert werden.

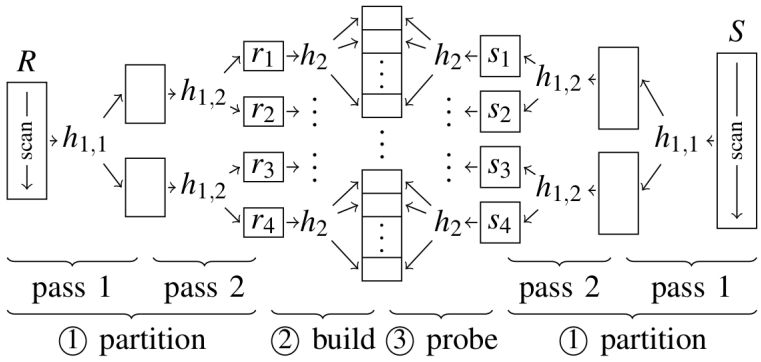

Abbildung 2.10: Radix-Cluster Hash-Join über die Relationen R und S aus [BTAO13]

2.5.2 Paralleler Radix-Cluster Algorithmus

In Abschnitt 2.4.2 wird die Ausnutzung mehrerer Kerne während der Bildung von Partitionen erläutert. Dieses lässt sich auch auf den Radix-Cluster Algorithmus übertragen. Hierbei ergibt sich ein Problem, da die Anzahl der Partitionen durch die CPU Kerne limitiert wird. Sollte nun eine potenziell große Menge n an Teilrelationen gebildet werden, so entsteht ein Wettbewerb um die CPU Kerne. Zur Verwaltung der n Threads wird typischerweise eine Task Queue implementiert, die die Arbeit aufteilt und den Wettbewerb zwischen den Threads verhindert [BTAO13]. Ständige Kontextwechsel auf der CPU wären im Hinblick auf die zusätzliche Arbeit zu teuer und würden eine effiziente Berechnung verhindern.

2.5.3 Zusammenfassung

In den vorherigen Abschnitten wurde auf die Abhängigkeiten der Laufzeit von den einzelnen Hardwarebausteinen wie Cache und TLB eingegangen. Auch wurde auf die parallele Berechnung während der Bildung von Partitionen eingegangen. Die Größe der Partitionen hängt dabei von den genutzten Radix-Bits ab. Die Wahl der richtigen Anzahl an Radix-Bits ist dabei maschinenabhängig. Die optimale Anzahl dieser kann durch Messungen herausgefunden werden. Die optimale Bestimmung hängt weiter von der Größe der Eingaberelationen ab.

2.6 Bit-Operationen

Eine besondere Form des Datenbankindex ist die Darstellung als Bitmap Index. Bei dieser besonderen Form der Datenspeicherung werden ein oder mehrere Attribute in Form eines Bitmusters gespeichert. Die Speicherung in dieser Form ist besonders bei einer geringen Kardinalität sinnvoll. Abbildung 2.11 enthält genutzte Verkehrsmittel. Die Spalten Auto bis Fußgänger enthalten Bitmuster. Jede 1 bedeutet, dass die Person mit der $ID = x$ das

Verkehrsmittel	id	Auto	Motorrad	Taxi	ÖPNV	Fußgänger
	1	1	1	0	0	1
	2	1	0	0	1	0
	3	0	0	0	1	0
	4	1	0	1	0	0
	5	1	1	0	0	0
	6	0	0	0	1	1
	7	0	1	1	0	0
	8	1	0	0	0	0
	9	1	0	1	0	0
	10	1	1	0	1	0

(a) Bitvektor

id	Verkehrsmittel
1	Auto
1	Motorrad
1	Fußgänger
2	Auto
2	ÖPNV
3	ÖPNV
4	Auto
4	Taxi
5	Auto
5	Motorrad
6	ÖPNV
6	Fußgänger
7	Motorrad
7	Taxi
8	Auto
9	Auto
9	Taxi
10	Auto
10	Motorrad
10	ÖPNV

(b) Darstellung ohne Bitmaps

Abbildung 2.11: Spalte "Verkehrsmittel" mit Nutzung von Bitmap Indizes und in der normalen Darstellung

besagte Verkehrsmittel nutzt. Der Nutzen dieser Darstellungsform liegt in der schnellen Berechnung von logischen Operationen durch Nutzung der binären Operationen *AND*, *OR* und *NOT* [Joh99]. Zur Berechnung von Schnittmengen ist die AND-Operation wichtig, da diese 1 ist, genau dann, wenn der Eintrag der einen Relation 1 und der Eintrag der anderen Relation, an derselben Stelle, 1 ist. Tabelle 2.1 zeigt alle möglichen Zustände der AND Operation auf. So können alle Personen die Auto und Motorrad fahren durch eine AND-

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 2.1: Logische AND-Verknüpfung

Verknüpfung der in der Spalte Auto und Motorrad vorliegenden Bitmuster herausgefunden werden. Eine entsprechende Anfrage ist in Listing 2.4 zu sehen. Diese liefert die IDs = {1,5,10} zurück.

Listing 2.4: Anfrage an Relation Verkehrsmittel

```
SELECT id FROM Verkehrsmittel WHERE Auto = 1 AND Motorrad = 1;
```

2.6.1 Umwandlung der relationaler Darstellung in einen Bitmap Index und zurück

In Abschnitt 2.1 wird eine Relation Person vorgestellt. Die Anfrage aus Listing 2.1 liefert alle männlichen Personen die nicht älter als 40 Jahre sind. Im Unterschied zur Schnittmengenberechnung mit vorheriger RID Selektion werden nun Bitvektoren gebildet, die zur Berechnung der Schnittmenge dienen. Der Bitvektor $v_{Alter} = 11001111111$ stellt dabei die RID Liste nach der Selektion des Alters dar. Jede Position im Bitvektor symbolisiert die zugehörige RID. So steht die Position 3 mit ihrer Null dafür, dass die RID = 3 die Selektionsbedingung Alter < 40 nicht erfüllt. Der Bitvektor zur Repräsentation des Geschlechtes sieht wie folgt aus: $v_{Geschlecht} = 10001000110$. Das Ergebnis der Anfrage 2.1 lautet $v_{Ergebnis} = 10001000110$. Durch die auf Eins gesetzten Bits ist das Extrahieren der RIDs möglich. Diese sind $\{1, 5, 9, 10\}$ und korrespondieren mit den Ergebnis aus Abschnitt 2.1. Die Berechnung kann mit Hilfe der Abbildung 2.12 verfolgt werden.

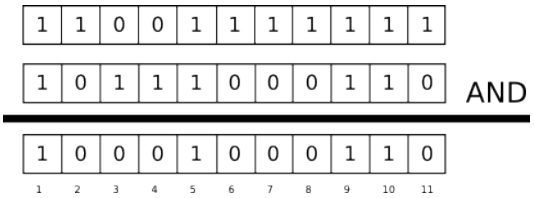

Abbildung 2.12: AND Operation zwischen zwei Bitvektoren

2.6.2 WAH-Kompression

Um Anfragen mit einer großen Kardinalität (> 5.000 Bitvektoren) beantworten zu können, wäre eine kompakte Darstellung genauso wie eine schnelle Latenz vom Vorteil. Häufig erweist sich die Performance bei der Ausführung logischer Operationen auf komprimierten Daten als schlechter, da die Daten erst dekomprimiert werden müssen. Als Folge dessen sind logische Operationen auf unkomprimierten Daten schneller [Joh99]. Der Nutzen von Kompression liegt vor allem in der Reduzierung der Anfragezeiten innerhalb eines Datenbanksystemes, indem die benötigten Zeiten der Nutzung von I/O und CPU reduziert werden. Für eine gute Kompression und die schnelle Ausführung von Bitoperation ist das Byte-aligned Bitmap Code (BBC) Schema bekannt. Wu et al. stellen mit dem Word-Aligned Hybrid (WAH) Schema eine Methode vor, die nur 50% mehr Speicherplatz bei 12-facher Geschwindigkeit bei der Berechnung von logischen Operationen benötigt [WOS01]. Die WAH-Kompression basiert auf der Idee des run-length encoding und verwendet keine Header words oder Bytes. Es wird nur zwischen den zwei Worttypen literal word und fill word unterschieden. Dabei gibt das am meisten signifikante Bit Aufschluss über die Typart. Eine 0 steht dabei für die literal words und eine 1 für die fill words. Das zweit signifikante Bit gibt dabei an, sollte es sich um ein fill word handeln, mit welchem

Wert die Bits aufzufüllen sind. Die Fülllänge ergibt sich dabei aus Anzahl der Bits innerhalb eines literal word. Bei einer 32-bit CPU beträgt die Länge $w = 32$, womit sich implizit die Länge eines literal word mit $w - 1$ ergibt. Der Autor [WOS06] nennt drei Vorteile der WAH-Kompression im Vergleich zum Byte-aligned Bitmap Code. Zuerst spricht er die einfache Komprimierung an. Außerdem unterliegen die komprimierten Bits keinen Abhängigkeiten. Als letztes Argument erwähnt er die bessere Kompressionsfähigkeit des BBC Schemas bei kleinen Fill words. Damit gehen höhere Kosten bzgl. logischen Operationen einher. Die WAH-Kompression nutzt in diesen Fällen literal words die ein besseres Laufzeitverhalten bieten. Zur Kompressionsgröße lässt sich keine genaue Angabe machen, da diese von den vorliegenden Bitmaps, den Attributwerten und den vorliegenden Daten abhängig ist.

2.6.3 Zusammenfassung

In diesem Abschnitt wurde auf den Bitmap Index als besondere Form der Darstellung von RID-Listen eingegangen und die Vorteile im Bezug auf die Berechnung mit Hilfe der AND-Operation erläutert. Außerdem wurde mit der WAH-Kompression eine Möglichkeit genannt, mit deren Hilfe man Bitvektoren komprimieren kann.

2.7 Zusammenfassung

In diesem Kapitel wurden verschiedene Algorithmen zur Schnittmengenberechnung vorgestellt. Sie bedienen sich unterschiedlichsten Konzepten, wie der skalaren Überschneidung aus Abschnitt 2.3.2, der Bildung von Hashtabellen aus Abschnitt 2.4 und 2.5.1 und der Nutzung von Bitvektoren aus Abschnitt 2.6. Der Algorithmus von Schlegel zur Schnittmengenberechnung setzt neben einer sortierten Eingabe die richtige Anzahl an Bits voraus. Bei der Nutzung der Hashverfahren ist es unerheblich, ob die Eingabe sortiert ist oder wie viele Bits genutzt werden. Bei der Nutzung von Bitvektoren ist eine Konvertierung zwingend erforderlich. Dabei haben die Algorithmen gewisse Vor- und Nachteile, die im Kapitel 4 näher erläutert werden.

Kapitel 3

Implementierung

In diesem Kapitel geht es um die Implementierungen der Algorithmen aus Kapitel 2. Es wird auf die Methoden zur Effizientssteigerung genauer eingegangen. Außerdem wird der Zusammenhang zwischen der Hardware und den Algorithmen näher erläutert. Dazu werden unter anderem SIMD und AVX Instrinsics erklärt und sich mit der Wirkung eines Threadpools bzgl. der Leistungssteigerung bei der Nutzung mehrerer Threads auseinandergesetzt. Diese und weitere Maßnahmen müssen auf Grund der Hardwareentwicklung berücksichtigt werden. Die Entwicklung der Hardware lässt die Performance der Prozessoren signifikant stärker steigen als die Geschwindigkeit des Arbeitsspeichers, sodass die Cacheeffizienz von großer Bedeutung ist. Auf Grund des langsamen Arbeitsspeichers warten viele CPUs ein Großteil ihrer Zeit auf den Hauptspeicher. Die Wartezeit kann durch eine bessere Nutzung der Caches reduziert werden, da der Speicher, je näher er an der CPU liegt, mit deutlich geringeren Latenzen aufwartet. So sind die Cachespeicher wesentlich schneller als der Hauptspeicher [HP06]. Zur Aufdeckung von Flaschenhälsen können Programme zur Performanceüberwachung genutzt werden. Sie messen neben der Laufzeit einzelner Programmabschnitt auch Cache-misses bzw. TLB-misses, die Speicherbandbreite und die Nebenläufigkeit.

3.1 Intel VTune Amplifier

Intel VTune Amplifier ist ein Programm zur Messung der Performance von Code. Es lassen sich einzelne Funktionen einer Applikation nach unterschiedlichen Kriterien messen und die gesammelten Daten können anschließen mit Hilfe einer grafischen GUI ausgewertet und analysiert werden. Folgende Punkte lassen sich nach [vtub] messen:

1. Funktionen, die ein Großteil der Zeit beanspruchen
2. Codeabschnitte, die vorhandene Zeit des Prozessors nicht optimal nutzen
3. Auffinden von Abschnitten zur sequentiellen oder parallelen Optimierung

4. Synchronisierende Objekt, die Auswirkung auf die Gesamtperformance nehmen
5. Gründe für Zeitaufwendungen von I/O-Operationen
6. Einfluss von verschiedenen Synchronisationsverfahren, Anzahl von genutzten Threads oder der Einfluss verschiedener Algorithmen
7. Aktivitäten der einzelnen Threads und die Übergänge
8. Hardware-basierte Flaschenhälse

Mit Hilfe verschiedener Messverfahren lassen sich verschiedene Daten sammeln. Grafik 3.1 zeigt die Übersicht der gesammelten Daten einer Advanced Hotspots Analyse. Hierbei werden Daten über den Call Stack, Context Switches und statistische Datenaufrufe gesammelt. Außerdem wird die Cycles per Instruction (CPI) bestimmt. Dieses stellt eine einfache Metrik dar und gibt Aufschluss über die durchschnittliche Zeit, die eine Instruktion pro Taktzyklus benötigt. Die Zusammenfassung einer Advanced Hotspots Analyse erfolgt in

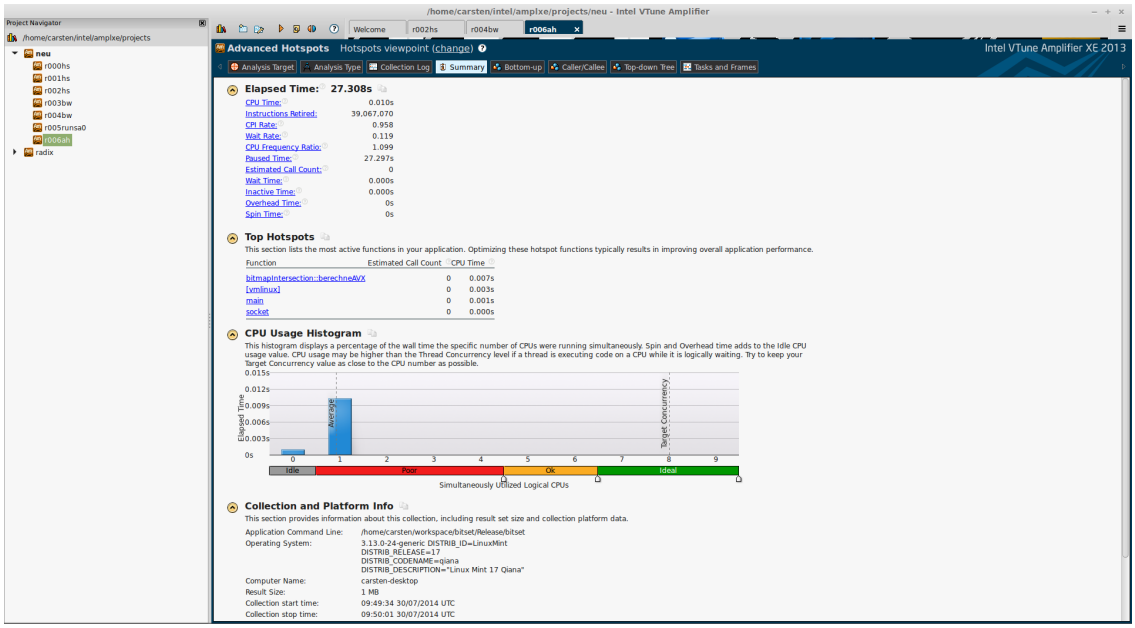

Abbildung 3.1: Screenshot der Zusammenfassung einer Advanced Hotspots Analyse im Intel VTune Amplifier Programm.

mehreren Kategorien. Diese beginnen mit einer groben Übersicht über die einzelnen Zeitwerte und die gemessenen Taktzyklen. Darunter befindet sich die vorher angesprochene CPI, die bei 0,958 liegt. Intel empfiehlt eine CPI Rate von unter 1 und benennt das theoretisches Optimum mit 0,25. Hohe Speicherlatenzen, branch mispredictions, SIMD-Instruktionen oder Floating-Point Operationen können negative Auswirkungen auf die CPI haben. Gefolgt werden die Zeitwerte von den “Top Hotspots“, den Funktionen, die am meisten Zeit auf der CPU benötigen. Anschließend stellt ein Diagramm die Nebenläufigkeit

des gemessenen Programm(-abschnitt) dar. Abschließend erfolgt eine Zusammenfassung des genutzten Betriebssystems und der verwendeten CPU. Die “Top Hotspots“ lassen sich durch anklicken genauer untersuchen. In der darauf folgenden Übersicht finden sich nähere Angaben zu den einzelnen Funktionen. Darunter sind die ausgeführten Instruktionen, die CPI Rate und die Ausnutzung der CPU Zeit. So ist die CPI Rate für die Funktion mit der meisten CPU Zeit in diesem Beispiel mit 0,768 geringer als der durchschnittliche Wert von 0,958.

3.2 Skalare Schnittmengenberechnung

In Listing 3.1 sehen wir eine naive Implementierung zur Berechnung der Schnittmenge. Diese benötigt im Minimum genauso viele Vergleiche wie die minimale Anzahl an Elementen aus den Eingaben. Zur schnelleren Berechnung muss die Anzahl an Vergleichen reduziert werden, da viele Vergleiche zu verschiedenen Verschachtelungen führen und diese die Ausführungszeit negativ beeinflussen.

Listing 3.1: naive Implementierung des Intersectionalgorithmus

```
1  int intersection(unsigned int *r, unsigned int *s, unsigned int anzahlR,
       unsigned int anzahlS, unsigned int *ouput){
2    unsigned int counter = 0, zeigerR = 0, zeigerS = 0;
3    while(zeigerR < anzahlR && zeigerS < anzahlS){
4      if(r[zeigerR] < s[zeigerS]){
5        ++zeigerR;
6      }else if(r[zeigerR] > s[zeigerS]){
7        ++zeigerS;
8      }else {
9        output[counter] = r[zeigerR];
10       ++counter;
11       ++zeigerR;
12       ++zeigerS;
13     }
14   }
15   return counter;
16 }
```

Der Vorteil der Implementierung von Schlegel [SWL11] liegt in der Vermeidung von Vergleichen und der Erhöhung der Datenparallelität. Das von ihm vorgestellte Verfahren nutzt keine if-then-else Verzweigung, wie in Zeile 4 bis 14 des Listing 2.2 zu sehen ist. Er bedient sich des `_mm_cmpestrm`-Befehles. Dieser ist Bestandteil der Intel SSE 4.2 Instrinsics und vergleicht zwei Strings bei gegebener Länge. Die Art und Weise des Vergleiches wird mit Hilfe einer Kombination aus verschiedenen Befehlen gesteuert. Diese wären in unserem Fall:

1. _SIDD_UWORD_OPS - Bestimmt den Input als vorzeichenlose 16-bit characters
2. _SIDD_CMP_EQUAL_ANY - Vergleicht jeden einzelnen Inputwert auf Gleichheit
3. _SIDD_BIT_MASK - gibt eine Bitmaske als Ergebnis zurück

Intel gibt den Befehl mit einer Latenz von 11 und einem Durchsatz von 3 für die Sandy Bridge-, Ivy Bridge- und Haswell-Architektur an. Dieses bedeutet, dass es 11 Prozessorzyklen braucht, bis die berechneten Daten für eine andere Instruktion zur Verfügung stehen. Der Durchsatz von 3 bedeutet, dass genau diese Anzahl an Prozessorzyklen zur Berechnung benötigt werden. Prozessoren auf Basis der Nehalem- oder Westmere-Architektur kommen auf eine Latenz von 8 und einen Durchsatz von 2 [Intb]. Diese Performancezahlen sind im Vergleich zu denen einfacher logischer Operationen, wie die AND-Operation, schlecht. Dennoch steigert sich die Performance bei der Berechnung der Schnittmenge im Vergleich zur naiven Implementation. Wie in Zeile 9-12 des Listings 2.2 zu sehen ist, wird die Ergebnismaske und die beiden größten Zahlen, eine aufsteigende Reihenfolge bei der Sortierung vorausgesetzt, extrahiert. Darauf erfolgt die Erhöhung der einzelnen Zeiger ohne die Nutzung einer Verzweigung. Die if-Bedingung wird dabei direkt verwertet, ohne dass eine Fallunterscheidung gemacht werden muss. Dieses funktioniert, da eine Bedingung zu true (1) oder false (0) ausgewertet wird und es bei einer Multiplikation mit 8 unabhängig ist, wie die Bedingung ausgewertet wird. Die Addition des Wertes erfolgt anschließend und verschiebt den Zeiger entsprechend der Belegung des Vektors. In Zeile 18 wird das Ergebnis mit Hilfe einer Shuffle-Instruktion und der vorher generierten Bitmaske in die richtige Reihenfolge gebracht. Die Berechnung des Shuffle-Vektors entfällt, da die zu den Masken zugehörigen Vektoren bereits berechnet wurden und in einer look-up Tabelle gespeichert werden. Die Größe der look-up Tabelle lässt sich wie folgt berechnen:

$$\frac{256\,möglicheBitmasken * 128\,bit}{1024 * 8} = 4\,KB$$

Wenn man den L3-Cache einer modernen CPU[1] zu Grunde legt, so kann man sagen, dass 4 KB ohne weiteres in die 8 MB des L3-Caches passen. Ansonsten umfasst der L2-Cache für jeden Kern weitere 256 KB, also das 64-fache des benötigten Speichers. Außerdem sollten die 4 KB der look-up Tabelle sogar in den L1-Cache passen. Allerdings könnte es sein, dass Teile der Tabelle häufiger verdrängt werden, sollten zu viele Daten in den Cache geladen werden, da dieser doch wesentlich kleiner ist als der L2-Cache.
Der Algorithmus besteht neben der oben beschriebenen Schnittmengenberechnung aus zwei weiteren Teilen, der Konvertierung von 32-bit Integerwerten zu 16-bit Werten und zurück. Diese stellen einen nicht unerheblichen Zeitaufwand dar. Auf Grund dessen wird die Berechnung der komprimierten Werte, die Schnittmengenberechnung und die Dekomprimierung auf mehrere Threads aufgeteilt. Der sortierte Input besteht dabei aus zwei

[1] Hier wird von einem Intel Xeon E3-1230v3 oder vergleichbare Modelle der i7 Reihe ausgegangen.

Listen A und B. Die Anzahl der Elemente in Liste A wird durch die Anzahl N an Threads geteilt. Anschließend wird die Liste in $A_1, A_2, ..., A_n$ Teile zerlegt. Aus jeder Teilliste entnimmt man das Größte Element und sucht mit Hilfe der binären Suche nach der Position des Elements innerhalb der Liste B. Die binäre Suche zeichnet sich durch ihre Effizienz von $\mathcal{O}(log\, n)$ aus. Sollte kein gleiches Element gefunden werden, so nimmt man das nächst kleinere Element. Die Position dieses Elementes stellt nun die Position der Teilung der Liste B dar. Die Suche nach dem gleichen Element ist zwingend notwendig, da ansonsten mögliche Teile der Schnittmenge nicht gefunden werden können. Nach der Aufteilung der Listen A und B kann der Algorithmus wie gewohnt ausgeführt werden, nur das dieses mal die Ausführung parallel stattfindet.
Nun kommen wir zur Zerlegung der 32-bit Werte in disjunkte Mengen, die sich die 16 höherwertigen Bits teilen. Eine Beschreibung des Datenlayouts befindet sich in Kapitel 2 im Abschnitt 2.3.4. Die Implementierung befindet sich im Anhang als Listing A.1. Zeile 1 bis 7 enthalten zwei Hilfsfunktionen, um die höher- und niederwertigen Bits herauszufiltern. Beiden Funktionen steht das Schlüsselwort `inline` voraus. Dieses sagt dem Kompiler, dass er die gewünschte Funktion inline, d. h. an den Aufrufort, schreiben soll. Dieses bietet sich für kleine Funktionen an und eliminiert die bei einem Funktionsaufruf anfallende Arbeit. Es entfällt damit der Aufwand für das Speichern und Wiederherstellen der Register. Die in Zeile 20 beginnende for-Schleife iteriert über alle Listenelemente und entscheidet für jedes Element, ob es zu der vorherigen Menge gehört, sich also die oberen Bits gleichen. Wenn dieses der Fall ist, werden die 16 niederwertigen Bits gespeichert und der Zähler für die Anzahl an Elementen innerhalb der Menge erhöht. Wenn sich die oberen Bits unterscheiden, wird ein neuer Abschnitt initialisiert. Innerhalb der Initialisierung werden der vorherigen Menge die Anzahl der enthaltenden Elemente mitgeteilt und die höherwertigen Bits der nun kommenden Menge gespeichert. Nachdem die Eingabeliste komplett durchlaufen wurde, wird die Anzahl der Elemente innerhalb der Liste zurückgegeben. Die Anzahl setzt sich aus der Anzahl der Elemente des Inputs sowie der höherwertigen Bits und der Anzahl der zur Domäne gehörenden Bits zusammen. Durch die nachfolgende Formel kann die Anzahl an disjunkten Mengen berechnet werden.

$$\frac{Anzahl\ Elemente\ in\ komprimierter\ Liste - Anzahl\ Elemente\ Input}{2}$$

Die Anzahl an disjunkten Mengen sollte Einfluss auf das Laufzeitverhalten des Algorithmus haben. Je weniger disjunkte Mengen vorhanden sind und je mehr Werte innerhalb einer Menge zu finden sind, desto besser sollte der Vergleich der einzelnen Werte zur Schnittmengenberechnung von den SIMD-Befehlen profitieren. Ein anderer Fall liegt vor, wenn die höherwertigen Bits sich nicht überschneiden, denn dann entfällt der Aufwand für die Berechnung der niederwertigen Bits komplett. Dieses ist durch die Verzweigungen in Zeile 6 bis Zeile 24 aus Listing A.2 sichergestellt.
Interessante Messergebnisse förderte die Schnittmengenberechnung mit versetzten Listen

zu Tage. Damit sind Listen mit stetigen RIDs gemeint, bei denen sich ausschließlich der vordere Teil der ersten Liste von dem letzten Teil der zweiten Liste unterscheidet. Abbildung 3.2 zeigt zwei solche versetzten Listen. Durch das Anpassen der Liste, indem die

Abbildung 3.2: Darstellung der versetzten Listen

entsprechenden Enden abgeschnitten wurden, konnte die Berechnungszeit gesenkt werden. Die Berechnung der Punkte, an denen abgeschnitten wurde, hat keine messbaren negativen Auswirkungen auf andere Eingaben gezeigt. Um ein Gefühl von der gewonnenen Performance zu bekommen, sei zu nennen, dass bei einer 5%-igen Überschneidung die Berechnungszeit vor dem Anpassen bei $300\,ms$ lag und nach der Anpassung bei lediglich $15\,ms$. Die Begründung dieser immensen Einsparung liegt in dem deutlich geringeren Aufwand, der für das Zerlegen der Listen notwendig ist und der Reduzierung der Vergleiche auf der Ebenen der höherwertigen Bits.

3.3 Hashbasierte Schnittmengenberechnung

Wie in Abschnitt 2.4.2 und 2.5.2 in Kapitel 2 zu lesen ist, profitieren hashbasierte Algorithmen von der Nutzung mehrerer CPU-Kerne. Bei dem im Abschnitt 2.5.2 vorgestellten parallelem Radix-Cluster Algorithmus kann die Anzahl der einzelnen Cluster schnell die Anzahl der zur Verfügung stehenden Kerne übertreffen. Die Anzahl an Cluster ist abhängig von der Anzahl an genutzten Radix-Bits. Bei 11 Radix-Bits beträgt die Anzahl an Cluster $N = 2^{11} = 2048$. Sollten nun 2048 Threads erzeugt werden, so ist die CPU mit dieser hohen Anzahl an Threads überfordert. Ständige Wechsel zwischen den einzelnen Threads erhöht die Anzahl an Cache misses erheblich, da jedes mal der Ist-Zustand des verdrängten Threads gespeichert werden muss und der Zustand des auszuführenden Threads geladen werden muss. Der ständige Wechsel zwischen den einzelnen Threads hat daher negative Einflüsse auf die Performance des Algorithmus. Um die Einflüsse des Wechseln zu eliminieren, wird ein Threadpool zur Verwaltung der Threads implementiert.

Listing 3.2: Implementierung eines Threadpools nach [Byt]

```
1 class Threadpool {
2   std::deque<std::function<void()>> aufgaben;
3   std::list<std::shared_ptr<std::thread>> threads;
4   std::mutex mMutex;
5   std::condition_variable mCond;
```

```
6   bool fertig = false;
7
8   void arbeiter() {
9     bool running = true;
10    while (running) {
11      std::unique_lock < std::mutex > l(mMutex);
12      mCond.wait(l, [this]() {
13        return fertig || aufgaben.size() != 0;}); //Warte auf Signal zur
              Arbeit
14      if (!aufgaben.empty()) {
15        auto task = aufgaben.front();
16        aufgaben.pop_front();
17        l.unlock();
18        task();
19      } else {
20        running = !fertig;
21      }
22    }
23  }
24
25 public:
26  Threadpool(unsigned int anzahlThreads) {
27    for (unsigned int i = 0; i < anzahlThreads; ++i) {
28      threads.push_back(std::shared_ptr < std::thread > (new std::thread
            (&Threadpool::arbeiter, this)));
29    }
30  }
31
32  void addTask(std::function<void()>const &task) {
33    std::unique_lock < std::mutex > l(mMutex);
34    aufgaben.push_back(task);
35    mCond.notify_one(); //signalisiere einem wartenden Thread das Arbeit
          vorhanden ist
36  }
37
38  void joinAll() {
39    {
40      std::unique_lock < std::mutex > l(mMutex);
41      fertig = true;
42    }
43    mCond.notify_all(); //Alle Threads signalisieren, dass sie arbeiten
          sollen
44    for (auto t : threads) {
45      t->join();
46    }
47  }
48 };
```

Listing 3.2 zeigt den zugehörigen Quellcode der Implementation. Im Konstruktor in Zeile 26 ff. werden entsprechend der zur Verfügung stehenden Kerne Threads erzeugt. Jeder dieser Threads führt eine Funktion aus. Innerhalb dieser Funktion wird auf eine Datenstruktur zur Verwaltung der einzelnen Aufgaben zugegriffen. Um den exklusiven Zugriff auf die gemeinsam genutzte Ressource sicherzustellen wird ein Mutex verwendet. Dieser sorgt dafür, dass keine zwei Threads zur gleichen Zeit die kritischen Bereiche betreten können [Dij65]. Diese umfassen das Einfügen neuer Aufgaben (Zeile 32-36) und das Entnehmen der Aufgaben (Zeile 11-18). Die Threads führen nun parallel die einzelnen Aufgaben aus, ohne sich abwechseln müssen. Das Wechseln und somit die Speicherung des Ist-Zustandes zwischen den einzelnen Threads entfällt. Nachdem alle Aufgaben dem Threadpool übergeben worden sind, wird die Funktion `joinAll()` aufgerufen. Diese ruft auf jedem Thread `join()` auf. Bei dieser Operation wartet das Programm solange, bis der Thread mit seinen Aufgaben fertig ist. Dieses ist der Fall, sobald keine Aufgaben in der zugehörigen Datenstruktur enthalten sind. Die Berechnungen der einzelnen Threads können anschließend verwendet werden.
Die Reduzierung von Cache misses ist notwendig, da der Hauptspeicher eines Computers zu langsam ist, um eine CPU mit den benötigen Daten in Echtzeit zu versorgen. Abbildung 3.3 zeigt den hierarchischen Aufbau eines Speichersystems. In diesem Fall umfasst es

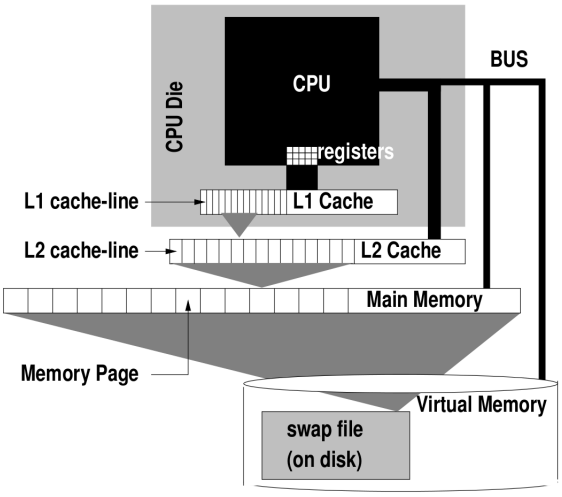

Abbildung 3.3: Hierarchischer Aufbau des Speichers innerhalb eines Computersystems aus [BMK99].

als unterste Ebene Massenspeicher wie Bandspeicher, Festplatten oder SSDs, gefolgt vom Arbeitsspeicher und den Level-2- und Level-1-Cache. Heutige Computersysteme nutzen in der Regel drei Cache-Level. Die Größe variiert wie die Geschwindigkeit mit der Positionierung des Caches zur CPU. So ist der L1-Cache der schnellste und kleinste Speicher und der L3-Cache der größte und langsamste. Zusätzlich zum Speicher gibt es noch TLB-Einheiten. Diese übersetzen virtuelle bzw. logische Adressen in physikalische Adressen und speichern

diese. Dadurch kann der Zugriff auf die am meisten genutzten Seiten beschleunigt werden. Sollte eine angeforderte Seite nicht im TLB zu finden sein, so kommt es zu einem TLB miss. Dieser kann dabei kostenintensiver sein als der Zugriff auf den Hauptspeicher [BMK99]. Je mehr unterschiedliche Daten von einem Algorithmus angefordert werden, desto mehr Seiten werden benötigt [MBK02]. Abbildung 3.4 zeigt detailiert die Cache Struktur eines

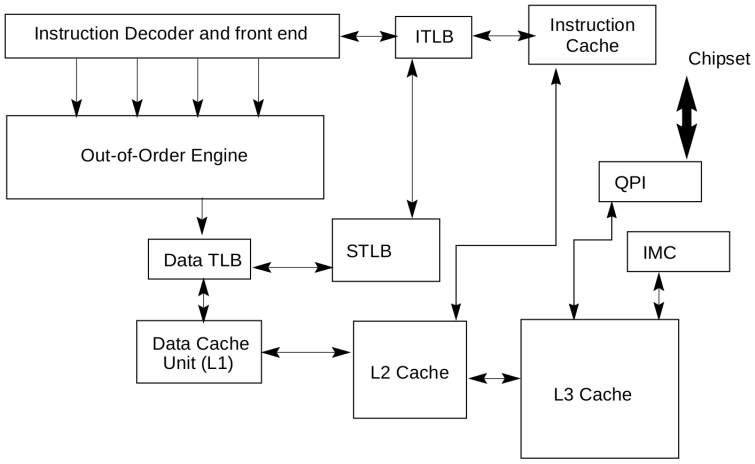

Abbildung 3.4: Cache Struktur eines Intel i7 Prozessors aus [sof14].

modernen Prozessors. Der Radix-Cluster Algorithmus reduziert die gleichzeitig benötigte Menge an Daten und somit die auftretenden Cache misses, indem dieser die Eingabedaten in viele Teile zerlegt. Durch die vielen verschiedenen Partitionen kommt es allerdings zur Überlastung des TLB und damit zu TLB misses. Um dieses Problem zu beseitigen werden Partitionen in mehreren Durchgängen gebildet, wie durch Abbildung 2.10 dargestellt. Blanas et al. [BLP11] hält sich an die strikte Ausführungsreihenfolge während der Ausführung des Radix-Cluster Algorithmus. Diese ist unterteilt in die Partitionierungsphase, das Bilden aller Hashtabellen und das Herausfiltern von Treffern. Dieses hat zur Folge, dass die bereits gebildeten Hashtabellen aus dem Cache verdrängt wurden. Balkesen et al. [BTAO13] haben durch eine veränderte Ausführungsreihenfolge gezeigt, dass die Anzahl an Cache misses reduziert werden kann, indem die Phasen für das Bilden der Hashtabelle und das Suchen der Treffer zusammen abläuft. Die Suche nach Treffern erfolgt mit Hilfe des Nested-Loop Algorithmus. Dieser vergleicht alle vorhandenen Elemente miteinander und berechnet die Schnittmenge. Die Anzahl an Elementen bestimmt dabei die Anzahl an notwendigen Vergleichen, so dass eine kleine Anzahl an Elementen zu bevorzugen ist.
Um die Performance weiter zu erhöhen, kann das Schlüsselwort *restrict* genutzt werden. Ein mit *restrict* versehener Zeiger ist ein Versprechen an den Kompiler, dass der Zugriff auf den durch ihn referenzierte Speicherbereich exklusiv erfolgt. Dieses Versprechen ermöglicht eine bessere Optimierung des Programmablaufes, da strukturelle Gegebenheiten berücksichtigt werden können. Bereits angeforderte Daten können wiederverwendet werden, sollte kein zwischenzeitlicher Zugriff erfolgt sein, da diese außerhalb des Programms

nicht verändert werden können. Der GNU G++ Kompiler versteht das Schlüsselwort an sich, kann es aber auf Grund des benötigten C99-Standards nicht übersetzen. Deshalb ist es notwendig, `__restrict__` oder `__restrict` zur Typqualifizierung zu nutzen [gnu]. In hashbasierten Algorithmen kommt der Hashfunktion eine zentrale Rolle zu. Sie sorgt für die Einteilung der Elemente, indem sie den Wert auf einen anderen abbildet. Um den abzubildenden Bereich einzugrenzen, bedient man sich der Modulo-Operation. Diese bestimmt den Restwert einer Division. Aus Performancegründen wäre eine schnellere Berechnung des Restwertes wünschenswert. Dieses ist mit Hilfe einer AND-Operation möglich, sollte der Divisor 2^x entsprechen, mit $x \in \mathbb{N}$. Listing 3.3 zeigt die Implementierung zweier Hashfunktionen mit dem Divisor $2^3 = 8$. Beide Funktionen erzeugen den gleichen Rückgabewert, bedienen sich jedoch unterschiedlicher Berechnungstechniken. Die Funktion `hash1` nutzt die MODULO-Operation, wohingegen die AND-Operation bei der Funktion `hash2` zum Einsatz kommt. Beim Radix-Cluster Algorithmus mit einer Eingabemenge $R = S = 128\,Millionen\,Elemente$ ist eine Zeitersparnis in Höhe von 25% messbar, durch die Nutzung der zweiten Variante.

Listing 3.3: Hashfunktionen

```
1 inline uint hash1(uint key) {
2   return (key % 8);
3 }
4
5 inline uint hash2(uint key) {
6   return (key & 7);
7 }
```

3.4 Bitvektorbasierte Schnittmengenberechnung

Bevor die Schnittmenge mit Hilfe einer schnellen bitweisen AND-Operation berechnet werden kann, muss die Eingabe, die als 32-bit Integerwerte vorliegt, in die binäre Darstellung umgeformt werden. Dazu wird das größte Elemente aus der Eingabe entnommen und eine korrespondierende Bitmap erzeugt. Diese enthält genauso viele Bits wie zur Darstellung aller Zahlen zwischen 0 und dem Maximum notwendig sind. Da in der Programmiersprache C/C++ keine native Darstellung von Bits möglich ist, sondern die kleinste mögliche Form 1 Byte (8 Bits) darstellt, benutzt man genug Bytes um die gewünschte Anzahl an Bits darstellen zu können. Die Anzahl an Bytes ergibt sich durch $\lceil(Anzahl\,benötigter\,Bits)/8\rceil$. Da bei der Berechnung in C++ abegrundet wird, empfiehlt sich die Anwendung eines Tricks. Bei diesem wird zu der *Anzahl benötigter Bits* Sieben hinzu addiert und anschließend durch Acht geteilt. Die Berechnung hat somit folgende Form: $(Anzahl\,benötigter\,Bits + 7)/8$. Mit Hilfe der `calloc()`-Funktion wird eine mit ausschließlich mit Nullen gefüllte Bitmap erzeugt. Anschließend wird über die Eingabe

iteriert und für jedes Element das korrespondierende Bit gesetzt. Dieses wird durch eine OR-Operation zwischen dem Byte, in dem sich das Bit befindet, und einer erzeugten Bitmaske gesetzt. Die Bitmaske errechnet sich aus dem Wert der Eingabe und einer Modulo 8 Operation. Diese kann durch eine schnelle AND-Operation mit dem Wert 7 ersetzt werden. Um bei dem Setzen von Bits durch Parallelität zu profitieren, wird die Eingabe entsprechend der zur Verfügung stehenden Kerne aufgeteilt. Anschließend wird für jede Teileingabe eine eigene Bitmap erzeugt. Nach der Erstellung können die einzelnen Bitmaps schrittweise mit Hilfe der OR-Operation verodert werden. Abbildung 3.5 veranschaulicht den Vorgang. Die Anzahl der genutzten Kerne wird dabei schrittweise zurückgefahren.

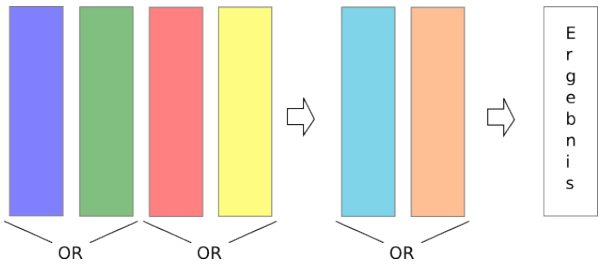

Abbildung 3.5: Anwendung des OR-Operators auf die gesamten Bitmap

Des Weiteren kann die Parallelität über den gesamten Zeitraum der OR-Operation genutzt werden, indem die einzelnen Bitmaps in Teile zerlegt werden und jeder Kern die zusammengehörigen Teile verodert, wie durch Abbildung 3.6 dargestellt wird. Nachdem

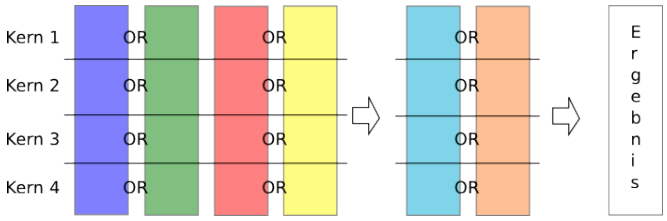

Abbildung 3.6: Anwendung des OR-Operators auf eine Teilmengen der Bitmap

nun für jede Eingabemenge eine Bitmap vorliegt, kann es an die Berechnung der Schnittmenge gehen. Dazu greift man auf die logische AND-Operation zurück, wie in Abschnitt 2.6 bereits beschrieben wurde. Um die volle Bandbreite zwischen Hauptspeicher und der CPU zu nutzen, wird erneut das Konzept der Datenparallelität angewendet. Der Intel Advanced Vector Extensions (AVX)-Intrinsic Befehl `_m256_and_si256` kann 256 Bits parallel verunden. Das ist die 32-fache Menge der normalen AND-Operation zwischen 8 Bits. Im Vergleich zu dem Befehl `_mm_cmpestrm` aus dem Abschnitt 3.2 ist die Latenz mit 1 deutlich niedriger und leitet eine schnellere Bearbeitung durch die CPU ab. Listing 3.4 gibt den Bereich der AND-Operation wieder. Bevor es zu dieser kommt, werden zwei Vektoren mit Werten befüllt. Um den `_m256_load_si256`-Befehl nutzen zu können, muss der Speicher auf 32-byte aligned werden.

Listing 3.4: AND-Operation zwischen zwei Inputvektoren

```
1 vektor1 = _mm256_load_si256((__m256i*) &data1[index]);
2 vektor2 = _mm256_load_si256((__m256i*) &data2[index]);
3 vektor1 = _mm256_and_si256(vektor1, vektor2);
4 _mm256_store_si256((__m256i *) &resultData[index], vektor1);
```

Dieses ist notwendig, damit die geladenen Ressourcen in die vorhandenen Cachelines passen und die Daten nicht aus mehreren Cachelines erstellt werden müssen. AVX bietet auch Befehle an, um unangeglichenen Speicher zu laden. Drepper gibt hierfür jedoch ein schlechteres Laufzeitverhalten an [Dre07]. Um den Speicher anzugleichen, wird der `_mm_malloc`-Befehl verwendet. Dieser erwartet, wie der bekannte `malloc`-Befehl, die angeforderte Größe in Bytes. Als zusätzlicher Parameter wird die alignment Spezifikation übergeben. Um den mit `_mm_malloc` angeforderten Speicher frei zugeben, gibt es analog zum `free`-Befehl den Befehl `_mm_free`. Nachdem nun $2 * 256$ Bits geladen wurden, kommt es zur eigentlichen AND-Operation. Das Resultat der Operation wird anschließend abgespeichert. Der `_mm256_store_si256`-Befehl setzt genauso wie der `load`-Befehl angeglichenen Speicher voraus. Nachdem die Schnittmenge über die gesamte Eingabe mit Hilfe der AND-Operation erfolgt ist, kann mit der Rekonstruktion der RID Liste begonnen werden.
Um Speicherbandbreite zu sparen, kann das WAH-Kompressionsverfahren von [WOS06] genutzt werden. Bei diesem unterscheidet sich der Prozess zur Erstellung der Bitmaps genauso wie die Berechnung der Schnittmenge und die Rekonstruktion von RID-Listen. Bei der Erstellung wird eine sortierte Liste von RIDs gelesen. Sollte der Wert der Eingabe nicht innerhalb des durch die Kompression vorgegebenen Bereiches liegen, so kann dieser, da sich nur Null-Bits darin befinden, komprimiert werden. Dieses erfolgt, indem das erste Bit auf Eins gesetzt wird. Dieses zeigt an, dass das nachfolgende Bit die Art der Komprimierung darstellt. In diesem Fall werden Null-Bits komprimiert, daher ist das zweite Bit auf Null gesetzt. Sollte die Eingabe innerhalb des Bereiches liegen, so ist keine Komprimierung möglich, wodurch das erste Bit auf Null gesetzt wird. Es folgen $N - 1$ Bits bei einer gewählten Komprimierung von N. Solange die Werte der Eingabe innerhalb des Bereiches liegen, werden die entsprechenden Bits mit Hilfe von Bitoperationen gesetzt. Um eine Kompression mit Einsen zu erkennen, muss die Eingabe genau mit dem Wert des ersten Bits korrespondieren. Anschließend werden die nächsten $N - 2$ Eingabewerte durchlaufen und überprüft, ob diese in dem definierten Bereich liegen. Sollte dieses der Fall sein, werden zwei auf Eins gesetzte Bits geschrieben. Diese symbolisieren komprimierte Einsen. Ansonsten wird der Bereich nicht komprimiert und die entsprechenden Werte, durch auf Eins gesetzte Bits, dargestellt. Nach der Erstellung der komprimierten Bitmaps geht es zur Berechnung der Schnittmenge. Dabei werden zwei Bitmaps durchlaufen und anhand ihrer Werte entschieden, wie das weitere Vorgehen ist. So ist das Ergebnis

E der AND-Operation unabhängig von der Eingabe der Bitmaps, nennen wir sie R und S, sollte eine dieser Eingaben komprimierte Nullen enthalten. In diesem Fall enthält das Ergebnis ausschließlich Nullen und kann komprimiert abgespeichert werden. Die Tabelle 3.1 zeigt die möglichen Kombinationen und das entsprechende Ergebnis E. Im Fall, dass beide Eingaben nicht komprimiert sind und es zu einer AND-Operation kommt, ist es wichtig, die Anzahl der sich überschneidenden Einsen zu zählen. Sollte die Anzahl gleich Null entsprechen, so kann das Ergebnis komprimiert werden, da es nur Nullen enthält. Die Rekonstruktion von RID-Listen erfolgt, indem die Bits schrittweise gelesen werden,

Vergleich	R = 11	R = 10	R = 0...
S = 11	E = 11	E = 10	E = R
S = 10	E = 10	E = 10	E = 10
S = 0...	E = S	E = 10	E = R & S

Tabelle 3.1: Vergleich zwischen zwei Bitmaps mit entsprechender Ausgabe

um auf die Eigenheiten der Komprimierung reagieren zu können. Dieses ist auf Grund der fill words und literal words notwendig. Bei gesetzten Bits, die nicht die Komprimierung anzeigen, werden die RIDs entsprechend ihres Wertes generiert.

3.5 Zusammenfassung

In diesem Kapitel wurde stellvertretend das Intel VTune Amplifer Programm zur Messung der Performance vorgestellt. Es wurde gezeigt, wie innerhalb von Programmen von Datenparallelität profitiert werden kann und diese durch SIMD bzw. AVX-Instruktionen realisiert werden. Außerdem wurde die Nutzung eines Threadpools beschrieben und die Hardwarecharakteristika näher erläutert. Außerdem wurde auf die Eigenheiten bei der Berechnung der Schnittmenge auf komprimierten Vektoren eingegangen.

Kapitel 4

Evaluation

Innerhalb dieses Kapitels wird sich mit dem Einfluss externer Parameter auf das Laufzeitverhalten der in Kapitel 2 vorgestellten Algorithmen beschäftigt. Dazu werden die Grenzen der Algorithmen ausgelotet und auslösenden Probleme benannt. Außerdem findet eine kritische Auseinandersetzung mit den einzelnen Algorithmen statt. Abschließend findet ein Vergleich zwischen den gesamten Algorithmen statt, bei denen die Stärken und Schwächen zusammengefasst werden. In Tabelle 4.1 werden die technischen Daten der zugrunde liegenden Hardware detailliert spezifiziert. Der Quellcode wurde mit Hilfe des `gcc`-Kompiler in Version 4.8.2 unter Nutzung des Flag `-O3` übersetzt.

	Intel Haswell
CPU	Intel Xeon E3-1230v3
Cores/Threads	4/8
Cache Größe (L1/L2/L3)	4x32 KB/4x256 KB/8 MB
Speicher	8 GB DDR3 1600 MHz

Tabelle 4.1: Spezifikationen Testsystem

4.1 SIMD-basierte Schnittmengenberechnung

In diesem Abschnitt geht es zunächst um den Vergleich des skalaren Algorithmus von Schlegel et al. [SWL11] mit der parallelen Standardimplementierung `set_intersection` der Standard C++ -Library. Dazu wird auf das Messverfahren aus dem Paper [SWL11] eingegangen. Anschließend erfolgt eine kritische Auseinandersetzung und eine Empfehlung bezüglich der skalaren Implementierungen. Schlegel et al. nutzen gleichverteilte Daten und variieren die Selektivität, also den prozentualen Anteil an gemeinsamen Daten. Bei der Messung wird ausschließlich auf die Zeit, die die Berechnung der Überschneidung benötigt, eingegangen. Die Laufzeit für die Zerlegung von 32-bit Integerwerte und das Zusammenfügen werden nicht betrachtet. Da in realen Systemen 16-bit Integerwerte mit

einem maximalen Wertebereich von $2^{16} = 65536$ selten anzutreffen sind, sollte ein Vergleich die Kosten für Vor- und Nachbearbeitung enthalten. Messungen, mit 100 Millionen RIDs für jede zu schneidende Menge zeigen einen signifikanten Anteil dieser Kosten an der Gesamtlaufzeit. Die Abbildung 4.1 visualisiert die einzelnen Zeitabschnitte und stellt

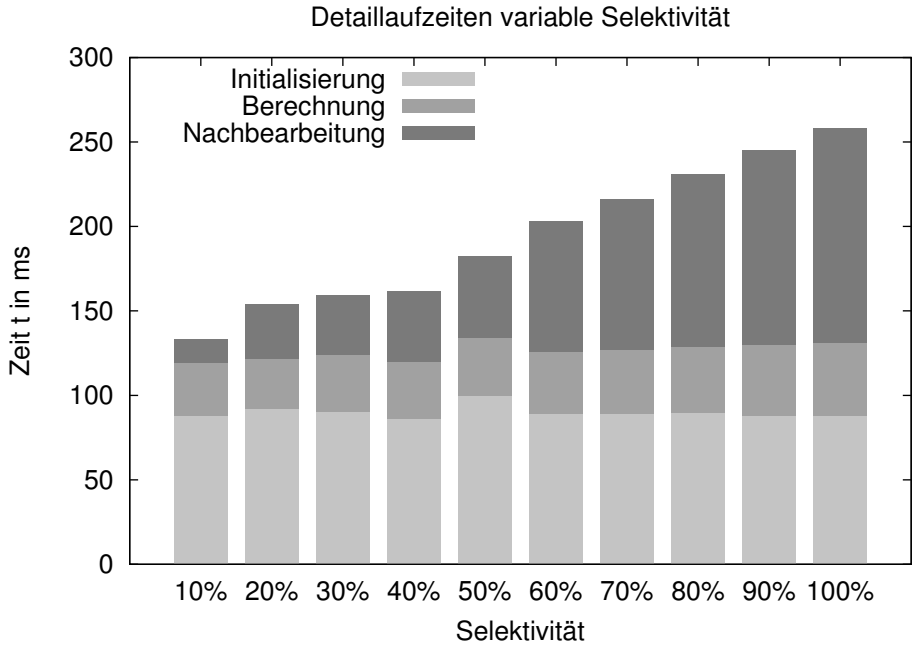

Abbildung 4.1: Das Laufzeitverhalten des Algorithmus in Abhängigkeit von der Ergebnisselektivität

diese in Relation zu der Gesamtlaufzeit dar. Insgesamt lassen sich während der Initialisierungsphase nur geringe Schwankungen feststellen. Die Berechnungsphase an sich zeigt ein sehr konstantes Laufzeitverhalten. Signifikante Änderungen zeigen sich in der Zeit, die für die Nachbearbeitung notwendig ist. Dieses erscheint logisch, da die Anzahl an gefundenen Überschneidungen den Aufwand für das Zusammensetzen der unteren und oberen 16-Bits direkt bestimmt. Innerhalb der Standardimplementierung entfällt dieser Aufwand, da 32-Bit Integerwerte direkt miteinander verglichen werden. Jedoch ist die eigentliche Berechnungszeit der Schnittmenge wesentlich länger. Auf Grund der signifikant schlechteren Laufzeit durch die anfallende Mehrarbeit während der Nachbearbeitungsphase ist eine Gegenüberstellung beider Berechnungen empfehlenswert. Hierbei gilt es auszuloten, ab wann welcher Algorithmus die bessere Performance aufweist und ob es überhaupt zu einem solchen Punkt kommt. Abbildung 4.2 trägt die Laufzeiten der beiden Algorithmen gegeneinander auf. Der Punkt, an dem der Mehraufwand nicht mehr durch die schnellere Berechnung ausgeglichen werden kann, liegt bei einer Selektivität von ca. 75%.
Bei einem Ergebnis mit sehr niedriger Selektivität von ca. 1,3% ist der SIMD-Algorithmus mehr als doppelt so schnell. Gemessen wurde dieses mit Hilfe von zufällig erstellten Zah-

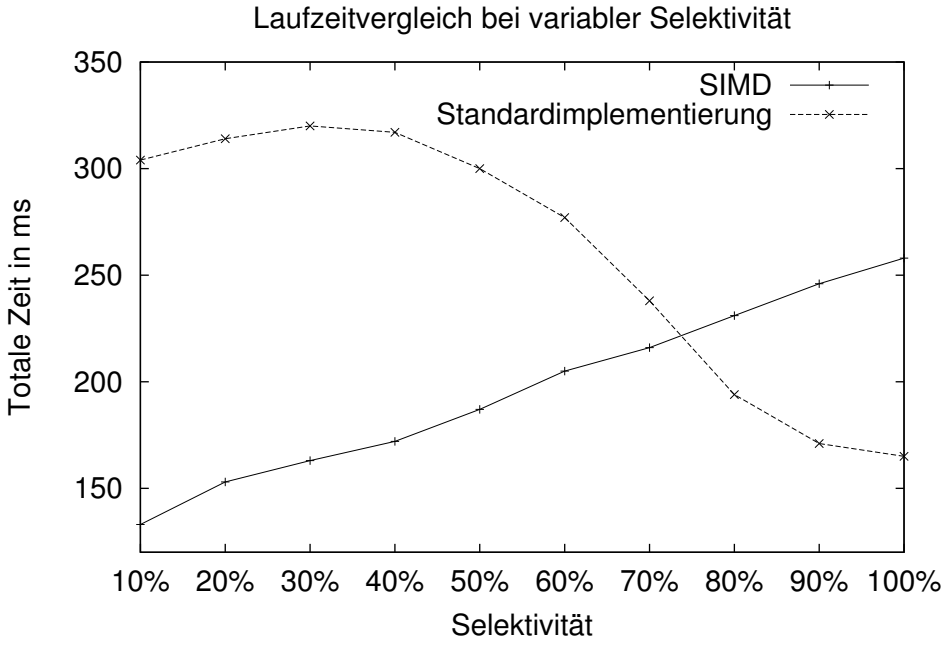

Abbildung 4.2: Laufzeitvergleich zwischen der SIMD- und der Standardimplementation bei variabler Ergebnisselektivität

len aus einem Wertebereich von 0 bis 2,1 Milliarden. Die RID-Listen enthielten bei dieser Messung jeweils ca. 30 Millionen RIDs. Im Durchschnitt braucht der SIMD-Algorithmus $42,4\,ms$, wohingegen die Standardimplementierung $89\,ms$ benötigt. Die vielen notwendigen Vergleiche und die damit einhergehende schlechte Vorhersagbarkeit (branch predicition) helfen dem SIMD-Algorithmus genauso, wie die wenigen Elemente, die bei der Wiederherstellung zur Ursprungsform von 32-bit Integerwerten notwendig sind.
Schlegel et al. [SWL11] geben Geschwindigkeitsvorteile von bis zu 5,3x an, die ihre Implementierung gegenüber den anderen skaleren Vertretern zur Berechnung der Schnittmenge erreichen kann. Die oben vorgestellten Messungen geben diese Performancesteigerung nicht her, da die gesamte Laufzeit betrachtet wurde. Unter der Voraussetzung einer niedrigen Selektivität sind Performancesteigerungen von 2x machbar.
Vor- und Nachbearbeitung haben einen großen Einfluss auf die Gesamtlaufzeit, da ihr Anteil an der Laufzeit im Verhältnis zur eigentlichen Berechnung bei dem 3- bis 5-fachem liegt. Dieser lässt sich auch durch die versetzten Listen aus dem Abschnitt 3.2 unterstreichen, bei denen das Abschneiden der Überhänge signifikante Leistungssteigerungen gebracht hat. Eine effizientere Implementierung wäre daher erstrebenswert. Sollten Leistungssteigerungen in diesem Bereich machbar sein, so könnte der Algorithmus von Schlegel durchaus die bekannten, skalaren Verfahren zur Schnittmengenberechnung dauerhaft verdrängen.

4.2 Radix-Cluster-Algorithmus zur Schnittmengenberechnung

Der Radix-Cluster Algorithmus zeichnet sich durch die Eigenschaft aus, dass er keine sortierte Eingabe erwartet, sondern auf unsortierten RID-Listen arbeitet. Aus diesem Grund soll die Laufzeit des Algorithmus mit der SIMD-basierten Schnittmengenberechnung verglichen werden. Außerdem wird der Einfluss der Hardware in Korrelation mit der Eingabemenge auf die Laufzeit dargestellt, indem die Anzahl an Radix-Bits variiert wird und die Auswirkung auf den Cache bzw. TLB gezeigt werden.
Abbildung 4.3 trägt die Laufzeiten des Radix-Cluster und der SIMD-Implementierung ab. Untersucht wurden jeweils 32 Millionen Tupel bei einer Ergebnisselektivität von 25%. Dabei wurde die SIMD-Implementierung ohne und mit Sortieraufwand betrachtet. Bei den Sortierverfahren wurde neben dem parallelen Standardsortieralgorithmus auch die Laufzeit des Bitonic-Sortierverfahrens geschätzt und im Diagramm vermerkt. Außerdem findet

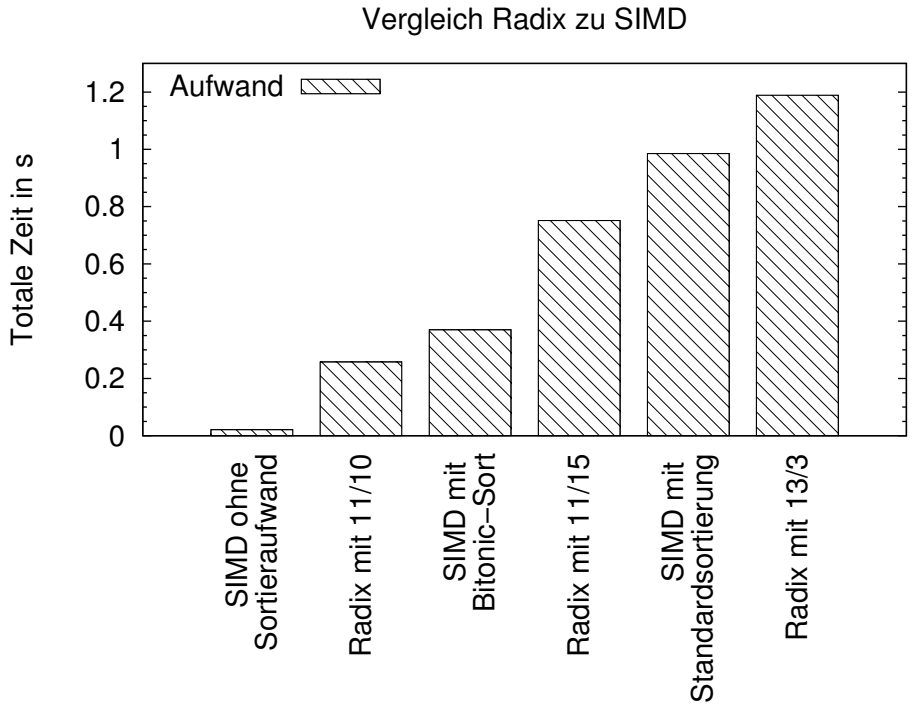

Abbildung 4.3: Laufzeitvergleich zwischen dem Radix-Cluster Algorithmus und der SIMD-Implementierung mit und ohne Sortieraufwand

sich der Radix-Cluster Algorithmus mit drei unterschiedlichen Konfigurationen wieder. Zweimaliges Clustern beinhalten dabei alle drei Variationen. Das beste Laufzeitverhalten weist in diesem Fall die Wahl von 10 Radix-Bits während der ersten Runde und 11 Radix-Bits während der zweiten Runde auf. Außerdem wurden einmal 15 gefolgt von 11 Radix-Bits und 3 gefolgt von 13 Radix-Bits gewählt.

Die Messerergebnisse legen dar, dass der Radix-Cluster Algorithmus bei der Wahl der richtigen Konfiguration gegenüber der SIMD-Implementierung bei einer unsortierten Eingabe im Vorteil ist. Der Hauptaufwand der SIMD-Implementierung liegt hierbei in der Sortierung der Eingabemengen. Eine effiziente Realisierung ist in diesem Fall erstrebenswert. Bei schlechter Wahl der Radix-Bits ist der SIMD-Algorithmus im Vorteil und bietet zudem eine gute Prognose bezüglich des Laufzeitverhaltens, da dieses direkt von den Faktoren der Selektivität und der Anzahl der Elemente der zu schneidenden Relationen abhängt. Die Anzahl der Elemente spielt auch eine Rolle für den Sortieraufwand. Dieser steigt mit wachsender Anzahl an Elementen.
Beim Radix-Cluster Algorithmus hängt die Laufzeit, neben der Größe der Eingabe, von der Wahl der Radix-Bits und der Wiederholungsrate des Cluster-Vorganges ab. Diese Wahl kann signifikante Auswirkungen auf die Laufzeit haben. So ist in dem genannten Beispiel eine vierfach bessere Laufzeit durch Wahl der Konfiguration möglich.
Misst man verschiedene Hardwarecounter[1], so lässt sich ein Anstieg bei dem Warten auf den Daten-TLB feststellen. Die in Abbildung 4.4 festgehaltenen Zahlen beziehen sich

Hardware Event Type	Hardware Event Count
CPU_CLK_UNHALTED.REF_TSC	764,361,146,540 - 152,012,228,018 = 612,348,918,522
DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	151,804,554 - 17,700,531 = 134,104,023
DTLB_LOAD_MISSES.PDE_CACHE_MISS	8,100,243 - 17,100,513 = -9,000,269
DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Not changed, 0
DTLB_LOAD_MISSES.WALK_COMPLETED_4K	126,000,189 - 0 = 126,000,189
DTLB_LOAD_MISSES.WALK_DURATION	7,992,011,988 - 678,001,017 = 7,314,010,971

Abbildung 4.4: Vergleich zwischen zwei Messungen der Hardwarecounter

auf verschiedene Counter und vergleichen zwei unterschiedlich konfigurierte Durchläufe. `DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK` bezieht sich zum Beispiel auf alle vorhandenen DTLB Level und zählt dabei die Anzahl der TLB misses, die zu einem *page walk* führten [vtua]. Unter *page walk* versteht man die Anzahl der Aufrufe, deren angefragte Adresse sich nicht im TLB befinden und durch den TLB neu berechnet werden müssen. Dieses dauert im Vergleich zu der Prozessorgeschwindigkeit lange und beeinträchtigt das gesamte Laufzeitverhalten negativ. `DTLB_LOAD_MISSES.PDE_CACHE_MISS` stellt dabei die Anzahl der TLB misses dar, bei denen die Übersetzung der physikalischen Adresse nur geringen Einfluss auf die Laufzeit hat, da nur ein geringer Teil der Adresse fehlt und übersetzt werden muss. So ist auch der deutlich geringere Wert bei dem Counter `DTLB_LOAD_MISSES.WALK_DURATION` nicht weiter verwunderlich, da dieser die Zeit misst, in der der TLB mit der Übersetzung von logischen in physikalische Adressen beschäftigt ist. `CPU_CLK_UNHALTED.REF_TSC` steht für die CPU-Zyklen, in denen die CPU sich nicht im Ruhezustand befindet oder anders gesagt aktiv ist. Ein hoher Wert spiegelt

[1] Hardwarecounter ermöglichen das Auslesen spezieller Events, die durch konfigurierbare Counter abgefragt werden.

bei diesem Event die höhere Gesamtlaufzeit wieder.
Die Gesamtanzahl an Radix-Bits hängt von der Anzahl an Elementen der Eingaberelationen ab. Je mehr Elemente diese enthalten, desto mehr Bits müssen gewählt werden. Bei einer Messung mit zwei Eingabe Relationen $R = S = |128M.|$ hat die Reduzierung der Radix-Bits um 10 zu einem erheblichen Leistungseinbruch geführt. Erklärbar ist dieses durch die längere Aufenthaltszeit im Nested-Loop, bei denen die radix-sortierten Elemente auf Gleichheit geprüft werden. Durch die geringere Anzahl an Radix-Bits steigt die Anzahl der zu überprüfenden Elemente erheblich. Dabei ist es wenig hilfreich, dass der Teil des Algorithmus ein Laufzeitverhalten von $\mathcal{O}(n^2)$ aufweist, wodurch sich die Auswirkungen verstärken. Messungen für diese Funktionen geben einen Zeitunterschied von ca. 50 Sekunden an. Unter Betrachtung der Gesamtlaufzeit der richtigen Konfiguration mit ca. 1,2 Sekunden lassen sich die Konsequenzen leicht verdeutlichen.
Es lässt sich festhalten, dass die richtige Wahl der Anzahl an Cluster-Vorgängen und die Anzahl an Radix-Bits über die Performance entscheiden. Die richtige Wahl an Radix-Bits ist dabei abhängig von der Eingabe und der zugrunde liegenden Hardware. Richtig konfiguriert ergeben sich Laufzeiten auf unsortierten Eingaben, die den skalaren Implementationen, bei denen der Sortieraufwand hinzukommt, voraus sind. Falsche Konfigurationen können die Performance erheblich verschlechtern. Kim et al. [KKL+09] stellen eine Verschiebung der Performance in Richtung von Sort-Merge-Join Algorithmen in Aussicht, da diese Verfahren durch SIMD-Instruktionen, mehrere Kernen und einer geringeren Speicherbandbreite pro Kern ein höheres Potential besitzen. Balkesen et al. [BATO13] sehen hash-basierte Join Algorithmen knapp im Vorteil, auch wenn die anderen Algorithmen hardwareseitig, durch breitere SIMD-Instruktionen, unterstützt werden. Sie stellen außerdem fest, dass die Gleichwertigkeit hinsichtlich der Performance bei sehr großen Eingabemengen zwischen beiden Arten der Berechnung gegeben ist. Diese Beobachtungen könnten auf die Berechnung von Schnittmengen übertragbar sein.

4.3 Schnittmengenberechnung mit Hilfe von Bitvektoren

In diesem Abschnitt beschäftigen wir uns mit den Bitmap-basierten Verfahren zur Schnittmengenberechnung, wie sie häufig in OLAP-Systemen zum Einsatz kommen. Es sollen die Grenzen und Probleme der einzelnen Implementierungen aufgezeigt werden, indem verschiedene Eingabefälle betrachtet werden. Untersucht werden außerdem die Auswirkungen der Wahl des Kompressionsfaktors bei der WAH-Kompression.
Zunächst wird die unkomprimierte Form der Bitvektoren betrachtet. Diese zeichnen sich durch eine sehr schnelle Berechnung der Schnittmenge durch die Nutzung logischer Operationen aus. Damit diese allerdings möglich ist, müssen die RID-Listen in Bitvektoren umgeformt werden. Nach der Berechnung erfolgt die Umformung in die andere Richtung. Abbildung 4.5 vergleicht die Gesamtlaufzeit der Bitvektor-Implementierung mit der des

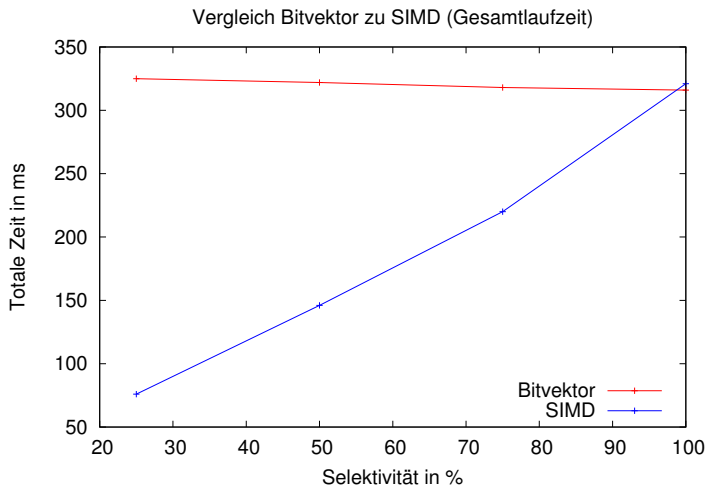

Abbildung 4.5: Vergleich der Gesamtlaufzeit zwischen Bitvektor und SIMD-Implementierung mit unterschiedlicher Selektivität des Ergebnisses

SIMD-Algorithmus nach [SWL11] bei sortierter Eingabe. Die Kosten für die Umformung von RID-Listen in Bitmaps und zurück ist mit eingeschlossen. Die Eingaben enthalten jeweils 128 Millionen Elemente mit einer Selektivität von 50%. Bei der Ausgabe variiert die Selektivität des Ergebnisses. Wie in Abschnitt 4.1 begründet wird die Laufzeit der SIMD-Implementierung mit einer zunehmenden Anzahl an gefundenen Überschneidungen schlechter. Die Berechnung basierend auf Bitvektoren zeigt fast konstantes Laufzeitverhalten und lässt sich nicht von der Selektivität des Ergebnisses beeinflussen. Abbildung

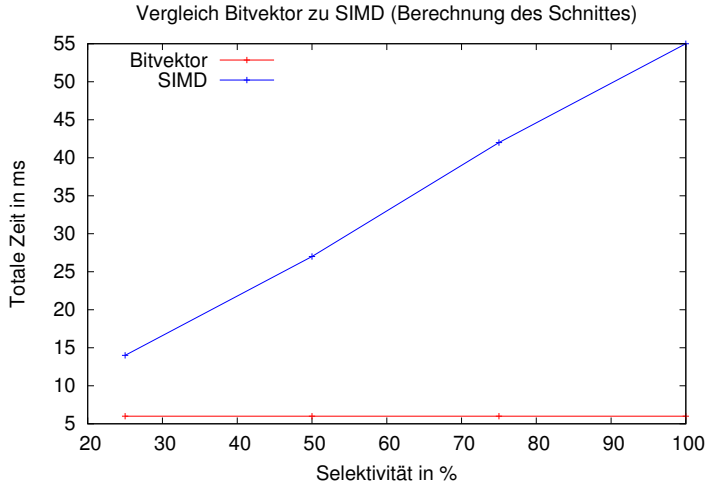

Abbildung 4.6: Vergleich der Laufzeit der eigentlichen Berechnung der Schnittmenge zwischen Bitvektor und SIMD-Implementierung mit unterschiedlicher Selektivität des Ergebnisses

4.6 trägt die Zeit, die zur Berechnung des Schnittes notwendig ist, auf. Auch hier zeigt die Implementierung auf Basis von Bitvektoren ein konstantes Laufzeitverhalten, welches dem

der SIMD-Implementierung zu jeder Zeit überlegen ist. Die Berechnungzeit der Schnittmenge hängt bei Bitvektoren von ihrer Größe ab. So kann es bei zwei Elementen im besten Fall ausreichend sein, wenn zwei Bits gesetzt sind. Dieses wäre bei einer Eingabe von Null und Eins der Fall. Damit würde dem Computer 1 Byte zur Speicherung ausreichen. Im schlechtesten Fall würde die Eingabe aus einer Null und 4.294.967.295 bestehen. Dann wären 512 MB = 536.870.912 Byte zur Speicherung notwendig. Da der Schnitt einer Menge über alle Bits des Vektors berechnet wird, ist es notwendig, alle vorhandenen Bits per AND-Operation zu verunden. Die Berechnung ist beendet, sobald das Supremum der beiden Eingabemengen erreicht ist. Sollten beide Relationen das größte, darstellbare Element enthalten, würde dieses im schlechtesten Falle bedeuten, dass für die Berechnung insgesamt 1536 MB an Daten bewegt werden müssen. Dabei werden $2x512$ MB zur CPU hin und 512 MB in den Arbeitsspeicher befördert. Bei jeweils zwei 32-bit Integerwerten pro Liste fallen bei einer konventionellen Berechnung per skalarem Algorithmus maximal 24 Byte an übertragenen Daten an. Außerdem erhöht sich der Aufwand für die Umwandlung der Bitvektoren in Integerwerte, da über die gesamten Bits des Ergebnisses iteriert werden muss. Bei über 4,2 Milliarden Bits bedeutet dieses einen erheblichen Zeitaufwand. Ein Vergleich zwischen der skalaren Berechnung mit Hilfe des SIMD-Algorithmus und der Variante mit Bitvektoren auf Eingabedaten mit unterschiedlicher Selektivität und gleichbleibender Ergebnisselektivität von 100%, dieses stellt den schlechtesten, annehmbaren Fall für den SIMD-Algorithmus auf Grund der hohen Kosten für das Zusammensetzen der 16-Bit Integerwerte dar, zeigt, dass die SIMD Variante beim variieren der Eingabeselektivität ein konstantes Laufzeitverhalten hat. Der Algorithmus auf Bitvektoren zeigt dieses Verhalten erwartungsgemäß nicht. Bei einer Selektivität der Eingaberelationen von 100%, also eine Eingabe mit stetig und aufeinander folgenden Integerwerten, braucht der SIMD-basierte Ansatz $313\,ms$ für die komplette Berechnung mit Umformen der Eingabe. Die Eingabe enthält dabei jeweils $|128M.|$ Elemente. Bei den Bitvektoren beläuft sich die Gesamtlaufzeit auf $252\,ms$. Wenn die Eingabe jeden hundertsten Wert, also eine Selektivität von 1%, aufweist, verändert sich die Laufzeit bei dem skalarem Verfahren nicht. Das andere Verfahren braucht für die gleiche Berechnung $3777\,ms$ und damit ca. 15 mal länger als bei einer Eingabe mit 100% Selektivität. Abbildung 4.7 zeigt das Verhalten des Algorithmus bei unterschiedlicher Selektivität der Eingabe. Die Selektivität des Ergebnisses bleibt dabei gleich, sodass dieses die Messungen nicht beeinflusst. Bei einer Eingabe von jeweils $|64M.|$ sortierten Integerwerten hat der Teil, der die Schnittmenge berechnet, kaum Einfluss an der Gesamtlaufzeit. Die Gesamtlaufzeit wird durch die Überführung der RID-Listen in Integerwerten und durch die Kosten für die Wiederherstellung bestimmt.
Einen negativen Einfluss auf die Laufzeit hat, neben einem sehr großen Eingabeintervall, eine unsortierte Eingabe. Bei dieser dauert die Generierung von Bitvektoren erheblich länger, da es beim Setzen einzelner Bits zu Cache misses kommt, da wahllos einzelne Bereiche des Speichers angefordert werden.

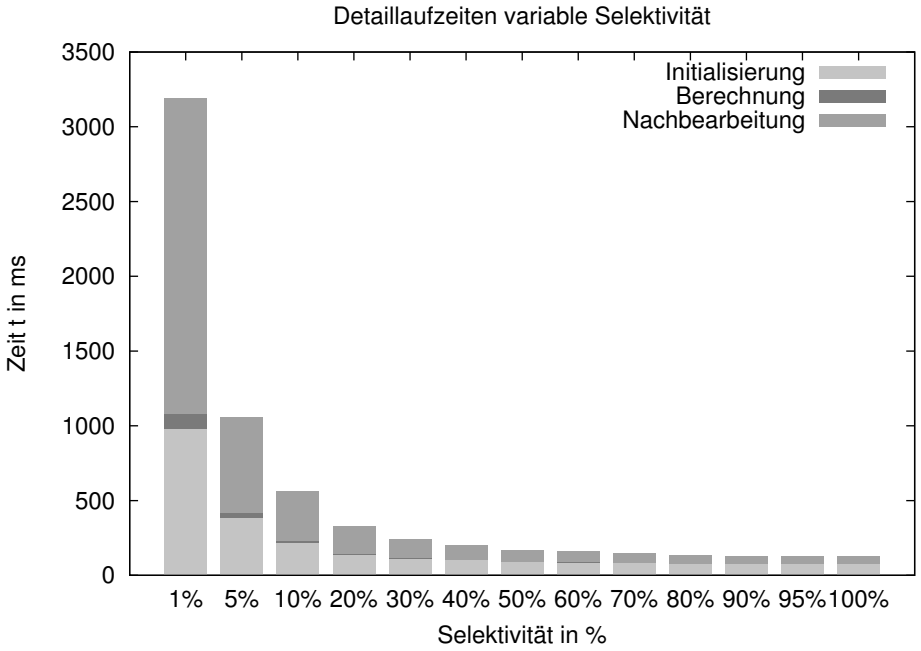

Abbildung 4.7: Vergleich der Laufzeit von Bitvektoren bei variabler Selektivität der Eingabe und gleichbleibender Selektivität des Ergebnisses

Um die Berechnungszeit bei der Nutzung von Bitvektoren zur verringern, bietet sich die parallele Berechnung einzelner Bereiche an, wie die Abbildung 3.6 aus Abschnitt 3.4 zeigt. Dabei kommt es sehr schnell zu Limitierungen durch die Hardware. Die Geschwindigkeit des Arbeitsspeichers zur Versorgung der CPU mit Daten reicht nicht mehr aus. Die Berechnung des Schnittes auf einem Kern sorgte schon für eine Speicherbandbreitenauslastung von ca. 12 GB/s. Systeme mit einer höheren Speicherbandbreite und mehreren RAM-Modulen können damit von einer parallelen Berechnung mit einhergehender Verringerung der Berechnungszeit profitieren. Dabei muss allerdings auf die Verteilung der Daten innerhalb des Arbeitsspeichers geachtet werden, da einzelne RAM-Module nur eine begrenzte Datentransferrate aufweisen.
Da die eigentliche Berechnungszeit von Bitvektoren sehr gering ist, bietet sich die Speicherung von RID-Listen in dieser Form an, um den Aufwand für die Konvertierung zu sparen. Diese Art der Speicherung findet häufig im Umfeld von OLAP-Systemen Verwendung. Die Speicherung der einzelnen Bitvektoren geht dabei zulasten des Hauptspeichers, der schnell an seine Grenzen stoßen kann. 1 Million Einträge innerhalb einer Tabelle sind keine Seltenheit. Sollte der Attributwert, über denen die Bitvektoren aufgebaut werden, eine Kardinalität von 1000 besitzen, so ergeben sich $1000000 * 1000 = 1000000000$ Bits zur Speicherung. Dieses entspricht einer Datenmenge von insgesamt 0,93 GB. Aus diesem Grund kann eine Komprimierung der Daten sinnvoll sein, wie sie durch die WAH-Kompression möglich ist. Dadurch lässt sich die Menge an Daten, die im Hauptspeicher gehalten werden, erhöhen.

Das Auslagern von Daten auf langsamen Festplattenspeicher kann damit möglicherweise umgangen werden.
Der Kompressionsfaktor der WAH-Kompression kann verändert werden. Die Auswirkungen verschiedener Stufen soll in Abhängigkeit von Eingabewerten mit variierender Selektivität untersucht werden. Im schlechtesten Fall benötigt die WAH-Kompression mehr Speicher als die ursprüngliche Speicherung von Daten als Bitvektor. Dieses liegt darin begründet, dass bei der Komprimierung von 32 Bits (WAH-32) das erste Bit angibt, ob das nachfolgende Bit als fill word verstanden werden soll oder ein literal word mit 31 Bit vorliegt. Auf Grund dieser Gegebenheit würde sich die Anzahl an notwendigen Bits erhöhen, sollte nicht komprimiert werden können.

$$Bits + \left\lceil \frac{Bits}{Kompression - 1} \right\rceil$$

Die Formel stellt dabei die obere Schranke der benötigten Bits dar. Je kleiner die Komprimierungsrate gewählt wird, desto mehr Bits werden im schlechtesten Fall benötigt. Die Anzahl an komprimierbaren Bits hängt dabei stark von der zu komprimierenden Verteilung ab. Je mehr Nullen oder Einsen aufeinander folgen, desto höher ist die Anzahl an fill words. Abbildung 4.8 zeigt die das prozentuale Einsparpotential bei RID-Listen mit

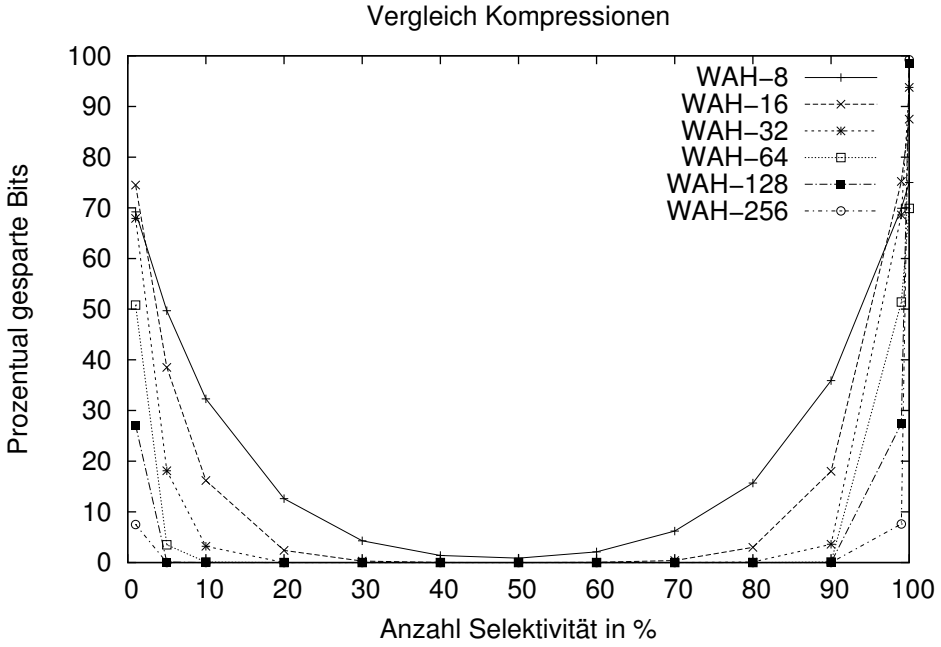

Abbildung 4.8: Vergleich verschiedener Kompressionsstärken bei verschiedenen Eingaben

zufällig gesetzten RIDs. Je häufiger es zu Wechsel zwischen Nullen und Einsen in den korrespondierenden Bitvektoren kommt, so geringer ist die Anzahl an Bits, die komprimiert werden können. Die Grafik veranschaulicht außerdem, dass große Kompressionsfaktoren viel schneller an den Punkt ankommen, an dem sie mehr Bits benötigen, als ihre äqui-

valente, unkomprimierte Form. In den Randbereichen zeigen große Kompressionsfaktoren ihre Vorteile, da fast alle Bits entweder gesetzt oder nicht gesetzt sind. Dadurch lässt sich eine ganze Reihe an Bits und damit Speicherplatz einsparen. Überträgt man dieses Wissen auf reale Systeme, so lässt sich festhalten, dass Komprimierungstechniken besonders bei zusammenhängenden Daten effektiv arbeiten. Dabei ist es unerheblich, ob die Bitvektoren fast ausschließlich aus Nullen oder Einsen bestehen oder diese im Wechsel vorkommen, wobei zwischen den Wechseln eine Reihe an gleichbleibenden Bits vorkommen muss. Die Größe der Reihe sollte dabei mit dem Kompressionsfaktor korrelieren, um eine optimale Kompression zu gewährleisten. Ansonsten führen ständige Wechsel zu dem aus Abbildung 4.8 bekannten Verhalten.
Beim beobachten des Laufzeitverhaltens lässt sich feststellen, dass die Implementierung mit steigender Selektivität bei den Eingabewerten ein besseres Laufzeitverhalten bei der Berechnung der Schnittmenge aufweist. Abbildung 4.9 zeigt die Laufzeit im Vergleich zur

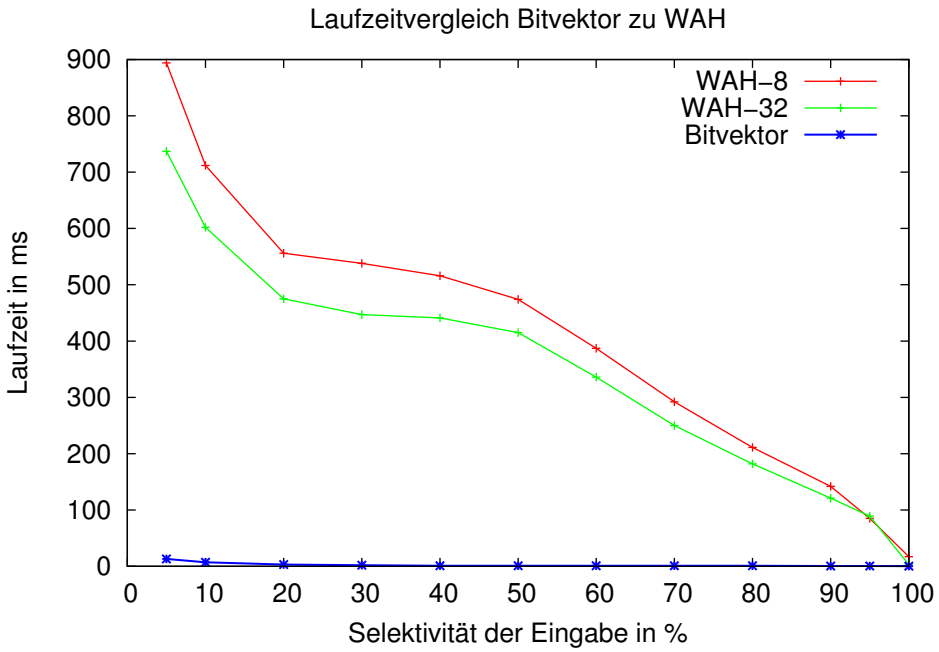

Abbildung 4.9: Laufzeitverhalten bei unterschiedlicher Selektivität der beiden Eingaben

unkomprimierten Berechnung. Die Berechnung der unkomprimierten Bitvektoren erweist sich als schneller. Die Laufzeit bei der Berechnung WAH-komprimierter Bitvektoren nähert sich mit steigender Selektivität an. Dieses kann an den Verzweigungen während der Berechnung liegen. Tabelle 3.1 aus Abschnitt 3.4 gibt die auftretenden Möglichkeiten an. Sollte es zu häufigen Sprüngen im Programmcode kommen, da die komplexen if-then-else-Konstrukte großflächig durchlaufen werden, kann dieses die schlechte Laufzeit, auf Grund der schlechten branch prediciton bei Eingabewerten mit einer geringen Selektivität, erklären.

Die komprimierten Bitvektoren belasten die Speicherbandbreite zwischen Arbeitsspeicher und der CPU, im Vergleich zur Schnittmengenberechnung mit unkomprimierten Bitvektoren, deutlich weniger. Dadurch ergeben sich Vorteile bei der Nutzung mehrerer Kerne, die verschiedene Anfragen parallel bearbeiten können. Die Berechnung der Schnittmenge zwischen zwei komprimierten Bitvektoren lässt sich nicht einfach auf mehrere Kerne verteilen, da die Bits nicht an ihren ursprünglichen Wert gebunden sind. Durch die Komprimierung verschieben sich die Bits. Wenn man trotzdem von einer parallelen Berechnung profitieren möchte, ist die Speicherung von Punkten, bei denen die Bitwerte übereinstimmen, notwendig. Diese Punkte müssen dabei die Komprimierung berücksichtigen und sich am Anfang eines jeden fill oder literal word befinden.
Bei der Erstellung von komprimierten Bitvektoren nach dem WAH-Verfahren sind aufsteigend sortierte RID-Listen notwendig. Es kommen unter Umständen die Sortierungskosten bei der Berechnung der Schnittmenge hinzu. Dieses macht die Anwendung des Verfahrens auf unsortierten Daten uninteressant, da der Nutzen nicht gegeben ist.
Auf Grund der Eigenschaften biete sich ein komprimierte Bitvektor bei einer Eingabemenge mit sehr geringer oder sehr hoher Selektivität, bei Daten, bei denen viele gleichartige Bits aufeinander folgen, oder bei nicht ausreichend dimensioniertem Hauptspeicher an. Ohne Kenntnisse über die Daten ist es schwierig abzuschätzen, ob sich die Verwendung des WAH-Verfahrens lohnt. Außerdem ist das gesamte Laufzeitverhalten schwer zu kalkulieren. Die Wahl des Komprimierungsfaktors ist im Zusammenhang mit konkreten Daten vorzunehmen.

4.4 Vergleich der Algorithmen

In diesem Abschnitt soll es um die Vor- und Nachteile der o. g. Verfahren zur Schnittmengenberechnung gehen. Die Stärken der skalaren Implementierung von Schlegel [SWL11] sind festzustellen, wenn die berechnete Schnittmenge eine geringe Selektivität aufweist. Dieses ist der Fall, sobald die RID-Listen sehr unterschiedlich sind und nur eine geringe Übereinstimmung besitzen. Die Zeit zur Berechnung des eigentlichen Schnittes ist dabei abhängig von der Anzahl der Eingaben. Die benötigte Zeit zur Berechnung des Schnittes bei Bitvektoren ist im Gegensatz zur SIMD-Implementierung unabhängig, da sowieso alle Bits miteinander verundet werden. Bei Bitvektoren ist der Wertebereich der Eingabe entscheidend. Dieser entscheidet über die notwendige Größe des Bitvektors und damit über die Zeit, die zur Berechnung nötig ist. Im Gegensatz zum SIMD-Algorithmus ist die Erstellung von Bitvektoren auch mit Hilfe von unsortierten RID-Listen möglich. Die hohen Konstruktionskosten eliminieren die schnelle Berechnungszeit bei Bitvektoren, so dass ein Vorhalten dieser empfehlenswert ist. Bei nicht ausreichendem Hauptspeicher kann es empfehlenswert sein, die Bitvektoren in komprimierter Form zu hinterlegen. Auf unsortierten RID-Listen kann der Radix-Cluster Algorithmus durch seine gute Performance glänzen, allerdings er-

SIMD	RADIX	BITVEKTOR	WAH
Stärken bei nicht korrelierten Eingabedaten	schnell bei unsortierten RID-Listen	sehr schnelle Berechung der Schnittmenge	Einsparung von Speicher abhängig von Eingabe
Laufzeit abhängig von Anzahl an Elementen der Eingabe	Wahl der Radix-Bits nicht immer klar	hohe Kosten für die Überführung von RID-Liste zu korresponierendem Bitvektor	Berechnungszeit abhängig von Selektivität der Bitvektoren
Hohe Kosten für Umrechnung von 32- zu 16- und 16- zu 32-Bit	Laufzeit abhängig von Anzahl an Elementen der Eingabe	Größe und Laufzeit abhängig vom Wert des größten Elementes	Voraussage der Laufzeit schwierig
Sortierte Eingabe wird vorausgesetzt		Berechnung der Schnittmenge unabhängig von der Selektivität der Eingabe	Sortierte Eingabe wird vorausgesetzt

Tabelle 4.2: Eigenschaften der verschiedenen Implementierungen

weist sich die Wahl der richtigen Radix-Bits nicht immer als einfach. Die Tabelle 4.2 fasst die Eigenschaften der verschiedenen Verfahren zusammen. Abhängig vom Einsatzzweck und der vorhandenen Hardware kann die Wahl des Verfahrens zur Schnittmengenberechnung eine Menge Zeit einsparen. Einen klaren Favoriten gibt es nicht. Wenn genügend Arbeitsspeicher zur Verfügung steht, empfiehlt sich die Speicherung und Nutzung von unkomprimierten Bitvektoren, da diese eine sehr schnelle Berechnung der Schnittmenge ermöglichen. Die Zeit zur Berechnung kann durch die Nutzung von AVX-512 Instruktionen in absehbarer Zeit weiter reduziert werden. Dadurch ist die AND-Operation auf Eingabedaten mit 512 Bits anstatt 256 Bits möglich. Sollte es nicht möglich sein, die RID-Listen als Bitvektoren zu speichern, dann sollte auf unsortierten Mengen der Radix-Cluster Algorithmus gewählt werden. Bei sortierten RID-Listen ist die skalare Variante durch die SIMD-Implementierung im Vorteil. Durch weitere Verbesserungen bei der Berechnung der disjunkten 16-Bit Mengen und der anschließenden Wiederherstellung der 32-Bit Werten kann die gesamte Ausführungszeit des Algorithmus weiter reduziert werden, so dass dieser in Verbindung mit einem effizienten Sortierverfahren im Vorteil sein kann. Auch bei diesem Algorithmus könnten AVX-Instruktionen, die passende Instruktion vorausgesetzt, die Berechnungszeit der Schnittmenge in Zukunft reduzieren. Bis auf das WAH-Verfahren profitieren die vorgestellten Algorithmen bei der Berechnung der Schnittmenge von mehreren CPU-Kernen. Im Fall der AND-Operation auf Bitvektoren muss die Nutzung von

mehreren Kernen kein Vorteil darstellen, da die Operation hohe Anforderungen an die Speicherbandbreite stellt.

4.5 Zusammenfassung

In diesem Kapitel wurde sich mit dem Laufzeitverhalten der Algorithmen beschäftigt. Dazu wurden die Grenzen der einzelnen Implementierungen aufgezeigt und die zu Grunde liegenden Probleme erläutert. Am Ende wurden die Algorithmen miteinander verglichen und Empfehlungen für gewisse Anwendungsfälle gegeben. Außerdem wurde ein kurzer Ausblick auf zukünftige Verbesserungen bei der Laufzeit durch Nutzung neuer Instruktionen gewährt.

Kapitel 5

Zusammenfassung und Fazit

Zu Beginn dieser Arbeit wurde auf das Problem der Schnittmengenberechnung eingegangen und in seinem Kontext erläutert. Außerdem wurden neben den RID-Listen, wie sie in klassischen Datenbanksystemen zu finden sind, Bitvektoren vorgestellt. Diese finden sich häufig in OLAP-Systemen. Im Anschluss wurden verschiedene Verfahren zur Berechnung der Schnittmenge vorgestellt. Darunter sind neben dem Ansatz des sortierten Mischens in Form einer effizienten Implementierung durch Ausnutzen von Datenparallelität auch hashbasierte Verfahren. Bitvektoren nutzen die AND-Operation zur Berechnung der Schnittmenge.
Kapitel 3 beschäftigt sich mit der effizienten Implementierung der Algorithmen. Dabei wurde auf die Eigenschaften von moderner Hardware eingegangen. Beim sortierten Mischen und beim Schneiden von unkomprimierten Bitvektoren kann von der Datenparallelität profitiert werden. Beim Radix-Cluster Algorithmus ist es notwendig, die Eigenschaften des TLB zu verstehen, damit eine Überlastung vermieden wird. Die effektive Nutzung des Caches spielt ebenso eine Rolle wie die Verwendung eines Threadpools. Dieser sorgt für eine gute Auslastung aller Prozessorkerne und reduziert die Wechsel zwischen einzelnen Threads, sollte die Anzahl der Threads die Anzahl der zur Verfügung stehenden Kerne übertreffen.
Im anschließendem Kapitel wurden die Algorithmen miteinander Verglichen und Eigenheiten benannt. Diese finden sich kompakt zusammengefasst in Tabelle 4.2 wieder. Ein Vergleich zwischen Algorithmen, die RID-Listen als Eingabe haben, und den Algorithmen auf Basis von Bitvektoren ist auf Grund der Kosten für die Konvertierung schwierig. Mit Hilfe von unkomprimierten Bitvektoren ist die Berechnung des Schnittes häufig sehr schnell. Die notwendige Zeit und der Speicherplatz sind dabei abhängig von dem Wert des größten Elementes der Eingabe. Außerdem stößt die Berechnung im Bezug auf die Speicherbandbreite schnell an die Grenzen der Hardware. Die Zeit für die Erstellung von Bitvektoren variiert zwischen sortierten und unsortierten Daten. Eine unsortierte Eingabe belasten den Cache durch wahllose Zugriffe auf Daten mehr. Bei unsortierten RID-Listen

empfiehlt sich der Radix-Cluster Algorithmus, der auf sortierten RID-Listen im Vergleich zur skalaren Implementierung das Nachsehen hat.
Innerhalb der Arbeit wurde auf einige Aspekte zur effizienten Implementierung von Algorithmen zur Schnittmengenberechnung eingegangen. Diese wurden während der Implementation angewendet. Die Wahl der Algorithmen ist dabei nicht nur von der zu Grunde liegenden Hardware abhängig, sondern auch von den Daten. Die Verteilung und die Art der Repräsentation von RIDs haben Einfluss auf das Laufzeitverhalten. Da RID-Listen häufig in sortierter Form vorliegen, stellt der traditionelle Ansatz des sortierten Mischen eine gute Wahl dar. Zukünftige Entwicklungen der Hardware können die Wahl der Implementierung verändern. So wäre es interessant, die Performance beim sortierten Mischen, sollte die Notwendigkeit der Zerlegung von 32-bit Integerwerten in 16-bit Werte wegfallen, zu untersuchen. Voraussetzung wäre ein äquivalenter `_mm_cmpestrm`-Befehl, der auf 32-bit Werten arbeitet. Ein weiterer Punkt wäre die Umsetzung der Algorithmen im Zusammenhang mit dem Programmierparadigma des future-oriented software development (FOSD). Dieses beschäftigt sich mit der Generierung von Algorithmen unter Berücksichtigung der spezifischen Hardware aus einer gemeinsamen Codebasis [BBHS14, BBS14].

Anhang A

Weitere Informationen

Inputvektor 1	0	1	5	6	8	9	11	15
Inputvektor 2	1	4	5	7	9	11	14	21
Ergebnismaske	0	1	1	0	0	1	1	0
Ergebnisvektor	1	5	9	11				

Nach dem ersten Durchlauf wurden 4 Elemente gefunden, im Ergebnisvektor gespeichert und der erste Inputvektor auf die nächsten 8 Elemente verschoben

Inputvektor 1	16	20	21	22	23	24	27	30
Inputvektor 2	1	4	5	7	9	11	14	21
Ergebnismaske	0	0	1	0	0	0	0	0
Ergebnisvektor	1	5	9	11	21			

Nach dem zweiten Durchlauf wurde 1 Element gefunden, im Ergebnisvektor gespeichert und der zweite Inputvektor auf die nächsten 8 Elemente verschoben

Inputvektor 1	16	20	21	22	23	24	27	30
Inputvektor 2	22	24	25	26	27	28	29	30
Ergebnismaske	0	0	0	1	0	1	1	1
Ergebnisvektor	...	11	21	22	24	27	30	

Nach dem dritten Durchlauf wurden 4 Elemente gefunden, im Ergebnisvektor gespeichert und beide Inputvektoren auf die nächsten 8 Elemente verschoben

Inputvektor 1	31	34	35	36	37	39	41	42
Inputvektor 2	32	33	38	40	43	44	45	46
Ergebnismaske	0	0	0	0	0	0	0	0
Ergebnisvektor	...	11	21	22	24	27	30	

Nach dem vierten und letzten Durchlauf wurden keine Elemente gefunden.

Abbildung A.1: Beispielhafter Verlauf des von Schlegel vorgestellten Algorithmus zur skaleren Schnittmengenberechnung

Listing A.1: Implementierung der Aufteilung des Inputs in disjunkte Mengen

```
1  inline unsigned short upperBits(unsigned int ArrayWert) {
2    return ((unsigned short) (ArrayWert >> 16));
3  }
4
5  inline unsigned short lowerBits(unsigned int ArrayWert) {
6    return ((unsigned short) ArrayWert);
7  }
8
9  unsigned int IntIntersection::formatierung(unsigned int * __restrict__
       ArrayA, unsigned int l_a, unsigned short * __restrict__ Array16) {
10   unsigned short hoch = 0;
11   unsigned int anzahl_niederwertig = 1, position_anzahl_niederwertig = 1,
         counter = 3;
12
13   //initialisierung der Liste - Listenanfang
14   unsigned short upperlevel = upperBits(ArrayA[0]);
15   unsigned short lowerlevel = lowerBits((ArrayA[0]));
16   hoch = upperlevel;
17   Array16[0] = upperlevel;
18   Array16[2] = lowerlevel;
19
20   for (unsigned int i = 1; i < l_a; i++) {
21     upperlevel = upperBits(ArrayA[i]);
22     lowerlevel = lowerBits((ArrayA[i]));
23     if (hoch == upperlevel) {
24       Array16[counter++] = lowerlevel; //hinzufuegen der niederwertigen
             Bits
25       ++anzahl_niederwertig;
26     } else { //neue Initialiserung
27       hoch = upperlevel;
28       Array16[counter++] = upperlevel;
29       ++counter; //ueberspringen des Platzes, an dem die Anzahl der
             Nachfolgenden Elemente gespeichert werden
30       Array16[counter++] = lowerlevel;
31       Array16[position_anzahl_niederwertig] = anzahl_niederwertig;
32       anzahl_niederwertig = 1;
33       position_anzahl_niederwertig = (counter - 2);
34     }
35   }
36   Array16[position_anzahl_niederwertig] = anzahl_niederwertig;
37   return counter;
38 }
```

Listing A.2: Implementierung der Berechnung der komprimierten Darstellung des Inputs

```
1 unsigned int IntIntersection::berechneFormatierteSchnittmenge(unsigned
      short *ArrayA, unsigned short *ArrayB, unsigned int l_a, unsigned int
      l_b, unsigned short *Ergebnis){
2   unsigned int ZeigerA = 0, ZeigerB = 0;
3   size_t counter = 0, tempA = 0, tempB = 0, anzahl_niederwertig = 0;
4
5   while (ZeigerA < l_a && ZeigerB < l_b) {
6     if (ArrayA[ZeigerA] < ArrayB[ZeigerB]) {
7       ZeigerA += (ArrayA[ZeigerA + 1] + 2); //springt auf die naechsten
            hoeherwertigen Bits
8     } else if (ArrayB[ZeigerB] < ArrayA[ZeigerA]) {
9       ZeigerB += (ArrayB[ZeigerB + 1] + 2); //springt auf die naechsten
            hoeherwertigen Bits
10    } else {
11      Ergebnis[counter++] = ArrayA[ZeigerA];
12      tempA = (ArrayA[ZeigerA + 1] == 0) ? 65536 : ArrayA[ZeigerA + 1];
13      tempB = (ArrayB[ZeigerB + 1] == 0) ? 65536 : ArrayB[ZeigerB + 1];
14      anzahl_niederwertig = intersection(&ArrayA[ZeigerA + 2],
15           &ArrayB[ZeigerB + 2], tempA, tempB, &Ergebnis[counter + 1]);
16      if (anzahl_niederwertig == 0) {
17        --counter;
18      } else {
19        Ergebnis[counter++] = anzahl_niederwertig;
20        counter += anzahl_niederwertig;
21      }
22        ZeigerA += (tempA + 2);
23        ZeigerB += (tempB + 2);
24    }
25  }
26  return counter;
27 }
```

Abbildungsverzeichnis

2.1 Relation 5
2.2 Schnittmenge 6
2.3 Bitonic Merge Netzwerk 8
2.4 Erstellung Bitmaske 9
2.5 Unterteilung von Integerwerten 10
2.6 Speicherung von komprimierten Integerwerten 11
2.7 Einfacher Hash-Join 13
2.8 Partitionierender Hash-Join 13
2.9 Berechnung Histogramme 14
2.10 Radix-Cluster Hash-Join 16
2.11 Bitmap Relation 17
2.12 AND Operation zweier Vektoren 18

3.1 VTune Screenshot Zusammenfassung 22
3.2 Versetzte Liste 26
3.3 Hierarchischer Speicheraufbau 28
3.4 Cache Struktur 29
3.5 Berechnung gesamte Bitmap 31
3.6 Berechnung Teilmengen 31

4.1 Laufzeit SIMD-Algorithmus 36
4.2 Laufzeitvergleich SIMD/STD 37
4.3 Laufzeitvergleich Radix/SIMD 38
4.4 Hardwarecounter 39
4.5 Vergleich Bitvektor vs. SIMD 41
4.6 Vergleich Bitvektor vs. SIMD 41
4.7 Vergleich Bitvektor 43
4.8 Kompression WAH 44
4.9 Geschwindigkeit WAH 45

A.1 Ablauf Beispiel 52

Algorithmen, Programmcode und SQL Anfragen

1.1 Beispielhafte Anfrage 1
2.1 Anfrage an Relation Person 6
2.2 Paralleler Intersection-Algorithmus aus [SWL11] 8
2.3 Einfacher Hash-Join 12
2.4 Anfrage an Relation Verkehrsmittel 17
3.1 naive Implementierung des Intersectionalgorithmus 23
3.2 Implementierung eines Threadpools nach [Byt] 26
3.3 Hashfunktionen 30
3.4 AND-Operation zwischen zwei Inputvektoren 32
A.1 Implementierung der Aufteilung des Inputs in disjunkte Mengen 53
A.2 Implementierung der Berechnung der komprimierten Darstellung des Inputs 54

Tabellenverzeichnis

2.1 AND Operation 17

3.1 Vergleich zwischen zwei Bitmaps mit entsprechender Ausgabe 33

4.1 Spezifikationen Testsystem 35
4.2 Eigenschaften der verschiedenen Implementierungen 47

Literaturverzeichnis

[Bat68] Batcher, Kenneth E.: Sorting networks and their applications. In: *AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer conference.* New York, NY, USA : ACM, 1968, S. 307–314

[BATO13] Balkesen, Cagri ; Alonso, Gustavo ; Teubner, Jens ; Özsu, M T.: Multi-core, main-memory joins: Sort vs. hash revisited. In: *Proceedings of the VLDB Endowment* 7 (2013), Nr. 1

[BBHS14] Broneske, David ; Bress, Sebastian ; Heimel, Max ; Saake, Gunter: Toward Hardware-Sensitive Database Operations. In: *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)*, OpenProceedings.org, 2014, 229–234

[BBS14] Broneske, David ; Bress, Sebastian ; Saake, Gunter: Database Scan Variants on Modern CPUs: A Performance Study. In: *Proceedings of the 2nd International Workshop on In-Memory Data Management and Analytics (IMDM)*, 2014

[BLP11] Blanas, Spyros ; Li, Yinan ; Patel, Jignesh M.: Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs, ACM New York, NY, USA, 2011. – ISBN 978–1–4503–0661–4, S. 37–48

[BMK99] Boncz, Peter A. ; Manegold, Stefan ; Kersten, Martin L.: Database architecture optimized for the new bottleneck: Memory access. In: *VLDB* Bd. 99, 1999, S. 54–65

[BTAO13] Balkesen, Cagri ; Teubner, Jens ; Alonso, Gustavo ; Özsu, M T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: *29th International Conference on Data Engineering (ICDE)* IEEE, 2013, S. 362–373

[Byt] Bytes'n'Objects: *Implementation eines Threadpools.* Website, . – `http://bytesnobjects.dev.geekbetrieb.de/cpp/signalisierung`, besucht am 5.8.2014

[CNL+08] Chhugani, Jatin ; Nguyen, Anthony D. ; Lee, Victor W. ; Macy, William ; Hagog, Mostafa ; Chen, Yen-Kuang ; Baransi, Akram ; Sanjeev, Kumar ; Dubey, Pradeep: Efficent Implementation of Sorting on Multi-Core SIMD CPU Architecture, VLDB Endowment, 2008, S. 1313–1324

[Cod70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), Juni, Nr. 6, S. 377–387

[Dij65] Dijkstra, E. W.: Solution of a problem in concurrent programming control. In: *Communications of the ACM* 8 (1965), Nr. 9, S. 569

[DK11] Ding, Bolin ; König, Arnd C.: Fast set intersection in memory. In: *Proceedings of the VLDB Endowment* 4 (2011), Nr. 4, S. 255–266

[Dre07] Drepper, Ulrich: *What Every Programmer Should Know About Memory.* 2007

[gnu] *GNU Compilers Document.* – `https://gcc.gnu.org/onlinedocs/gcc-4.8.3/gcc/index.html`, besucht am 25. Juli 2014

[HP06] Hennessy, John L. ; Patterson, David A.: *Computer Architecture - A Quantitative Approach.* 4. Aufl. San Francisco : Morgan Kaufmann, 2006. – ISBN 978–0–080–47502–8

[Inta] Intel: *Intel Hyper-Threading Technology - How operating systems can do more and perform better.* Website, . – `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`, besucht am 6.8.2014

[Intb] Intel: *Intel Intrinsics Guide.* Website, . – `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, Version 3.2.1, besucht am 6.8.2014

[Joh99] Johnson, Theodore: Performance measurements of compressed bitmap indices. In: *Proceedings of the 25th International Conference on Very Large Data Bases* Morgan Kaufmann Publishers Inc., 1999, S. 278–289

[KKL+09] Kim, Changkyu ; Kaldewey, Tim ; Lee, Victor W. ; Sedlar, Eric ; Nguyen, Anthony D. ; Satish, Nadathur ; Chhugani, Jatin ; Di Blas, Andrea ; Dubey, Pradeep: Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. In: *Proceedings of the VLDB Endowment* 2 (2009), Nr. 2, S. 1378–1389

[MBK02] Manegold, Stefan ; Boncz, Peter ; Kersten, Martin: Optimizing main-memory join on modern hardware. In: *Transactions on Knowledge and Data Engineering* 14 (2002), Nr. 4, S. 709–730

[SKN94] Shatdal, Ambuj ; Kant, Chander ; Naughton, Jeffrey F.: *Cache conscious algorithms for relational query processing.* University of Wisconsin-Madison, Computer Sciences Department, 1994

[sof14] *Intel 64 and IA-32 Architectures Software Developer's Manual.* Bd. 3. Intel, Juni 2014. – `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`

[Ste14] Steiner, René: *Grundkurs Relationale Datenbanken.* 8. überarb. u. erw. Aufl. Wiesbaden : Springer Vieweg, 2014. – ISBN 978–3–658–04287–5

[SWL11] Schlegel, Benjamin ; Willhalm, Thomas ; Lehner, Wolfgang: Fast Sorted-Set Intersection using SIMD Instructions. In: *The Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS'11)*, 2011, S. 1–8

[Vos08] Vossen, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme.* 5. überarb. u. erw. Aufl. München : Oldenbourg Verlag, 2008. – ISBN 978–3–486–27574–2

[vtua] *Intel® VTune™ Amplifier XE Documentation.* Dokumentation, . – Enthalten im Softwareprogramm Intel VTune Amplifier XE 2013

[vtub] *Introducing the Intel VTune Amplifier.* – `https://software.intel.com/de-de/node/496918`, besucht am 30. Juli 2014

[WOMF13] Willhalm, Thomas ; Oukid, Ismail ; Müller, Ingo ; Faerber, Franz: Vectorizing Database Column Scans with Complex Predicates. In: *ADMS@ VLDB*, 2013, S. 1–12

[WOS01] Wu, Kesheng ; Otoo, Ekow J. ; Shoshani, Arie: A performance comparison of bitmap indexes. In: *Proceedings of the tenth international conference on Information and knowledge management* ACM, 2001, S. 559–561

[WOS06] Wu, Kesheng ; Otoo, Ekow J. ; Shoshani, Arie: Optimizing bitmap indices with efficient compression. In: *ACM Transactions on Database Systems (TODS)* 31 (2006), Nr. 1, S. 1–38

[WSO04] Wu, K. ; Shoshani, A. ; Otoo, E.: *Word aligned bitmap compression method, data structure, and apparatus*. http://www.google.com/patents/US6831575. Version: 14 Dezember 2004. – US Patent 6,831,575

[ZR02] Zhou, Jingren ; Ross, Kenneth A.: Implementing database operations using SIMD instructions. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* ACM, 2002, S. 145–156

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift