Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Kompression von großen Invertierten Indizes für Datenseen

Maintainance of large Inverted Indices

Bachelorarbeit

im Studiengang Informatik

von

Nils Steffen Martel

Prüfer: Prof. Dr. Ziawasch Abedjan
Zweitprüfer: Prof. Dr. Sören Auer
Betreuer: Mahdi Esmailoghli

Hannover, 30.12.2022

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 30.12.2022

Nils Steffen Martel

Zusammenfassung

Im Rahmen der Arbeit wurden verschiedene Kompressionsstrategien und Kompressionsalgorithmen untersucht, kombiniert und ausgewertet, um Invertierte Indizes von Datenseen möglichst effizient zu komprimieren. Dabei liegt der Fokus darauf, diese Datenstrukturen im Arbeitsspeicher zu halten, und auf diesen effizient arbeiten zu können.

Abstract

Maintainance of large Inverted Indices

Various strategies and algorithms for compression where evaluated for fitness for the use case of compressing inverted indices. Special focus is on keeping these data structures in main memory, and respecting fast index speeds.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Größe von Datenseen, sowie die Wichtigkeit diese Daten zu verarbeiten, nimmt stetig zu. Zum Verarbeiten von Datenseen präsentieren Paper wie Josie [12] oder DataXFormer [1] Algorithmen, welche mittels eines Invertierten Index funktionieren. Um auch große Datenmengen in dieser Datenstruktur verarbeiten zu können, ist es wünschenswert, effiziente Kompression der Daten im Arbeitsspeicher vorzunehmen. Ziel ist es, weiterhin schnellen Zugriff auf die Daten zu gewähren und diese gleichzeitig signifikant zu verkleinert.

1.1 Problem

Wie effizient verschiedene Kompressionsalgorithmen auf bestimmte Daten wirken, ist sehr von der Domäne der Daten abhängig [3] und ist eine weitgehend empirische Wissenschaft. Wie bekannte Kompressionsstrategien auf Invertierte Indizes von Datenseen (IID (Singular) / IIDs (Plural) im Folgenden) wirken, ist unbekannt und muss durch Messungen ausgewertet werden. In anderen Domänen haben sich Kombinationen von Algorithmen hervorragend zur Kompression herausgestellt [3]. Welche Algorithmen jedoch für sinnvolle Kombinationen infrage kommen und welche Kompressionsraten mit diesen erreicht werden können, ist noch nicht abschließend ermittelt.

1.2 Lösungsansatz

Um eine effektive Kompressionsstrategie für den Einsatz an IIDs zu finden, wurden im Rahmen der vorliegenden Arbeit einige vielversprechende Ansätze aus relevanten Papern und Büchern ausgewählt. Diese beschäftigen sich unter anderem mit modernen Techniken zu schnellen Kompressionsalgorithmen [3] [2], mit Suchmaschinen [10] oder direkt mit Invertierten Indizes von Datenseen [12]. Es wurde pro Methode jeweils ein Invertierter Index implementiert, welcher diesen Ansatz nutzt. Zusätzlich wurden für

jeden sinnvollen Kombinationsansatz ebenfalls eigene Implementierungen von IIDs entwickelt. Am Beispiel von verschiedenen Datenseen mit Echtdaten wurde die Kompressionsrate, die Zugriffszeit und die Zeit zum Erstellen der Struktur gemessen. Ebenso wurden Kombinationen von Ansätzen auf Kompatibilität geprüft und als Konstellation ebenfalls ausgewertet.

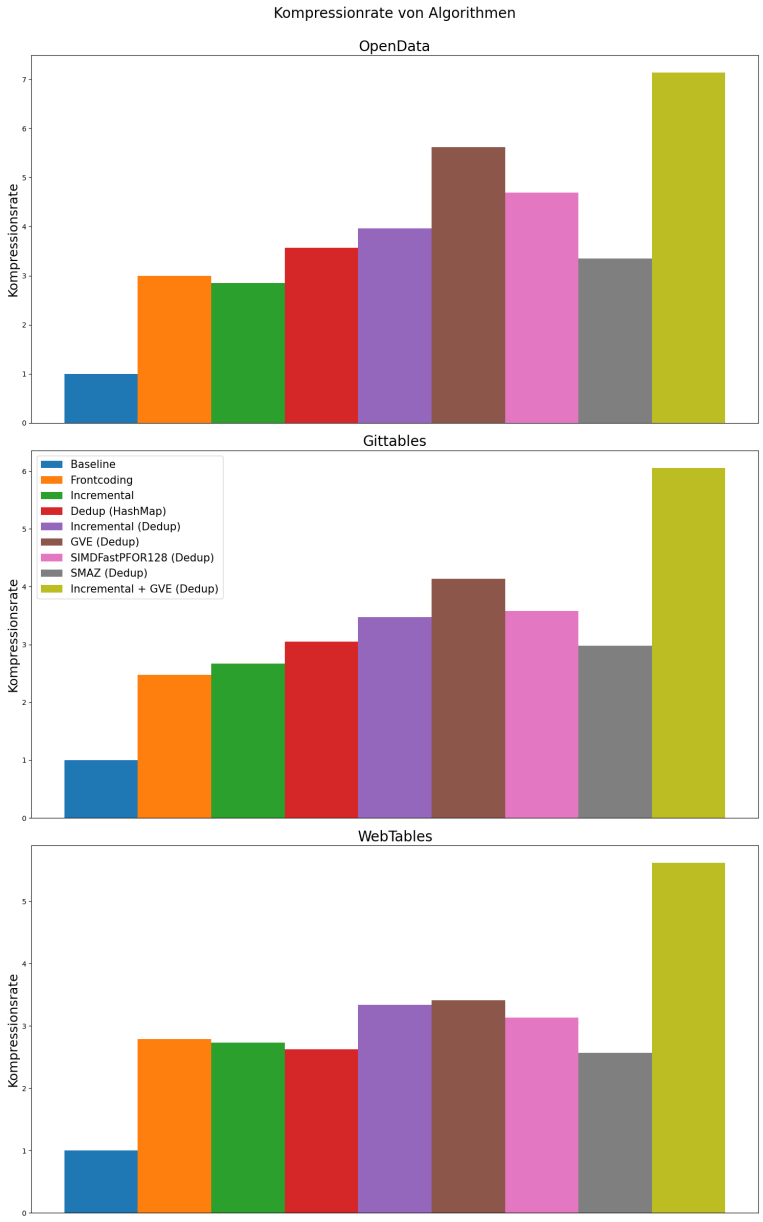


Abbildung 1.1: Die Kompressionsrate (y-Achse) von verschiedenen Algorithmen (x-Achse) gegenüber den unkomprimierten Daten (betitelt als 'baseline', linke Bar, blau)

1.3 Ergebnisse dieser Arbeit

Es wurde eine Kompressionsstrategie gefunden, welche die Größe der Daten im Arbeitsspeicher um einen Faktor von etwa 5 bis 7 verringert 1.1. Diese hohen Kompressionsraten sind in Abbildung 1.1 dargestellt und variieren je nach Eingabedatensee. Die größten Eingabedaten, welche unkomprimiert nicht vollständig in den 500 Gigabyte großen Arbeitsspeicher geladen werden konnten, ließen sich erst unter Anwendung von einer solchen Komprimierung verwenden. Dabei ist die Kompressionsgeschwindigkeit so hoch, dass selbst das Erstellen von enorm großen, komprimierten IIDs kaum länger als das eines unkomprimierten dauert. Auf kleineren Datenseen ist das Erstellen von komprimierten IIDs sogar deutlich schneller als das, von unkomprimierten. In Abbildung 1.3 sind die geringen Kosten der Komprimierung und der enorme Einfluss auf den Speicherbedarf gegenübergestellt.

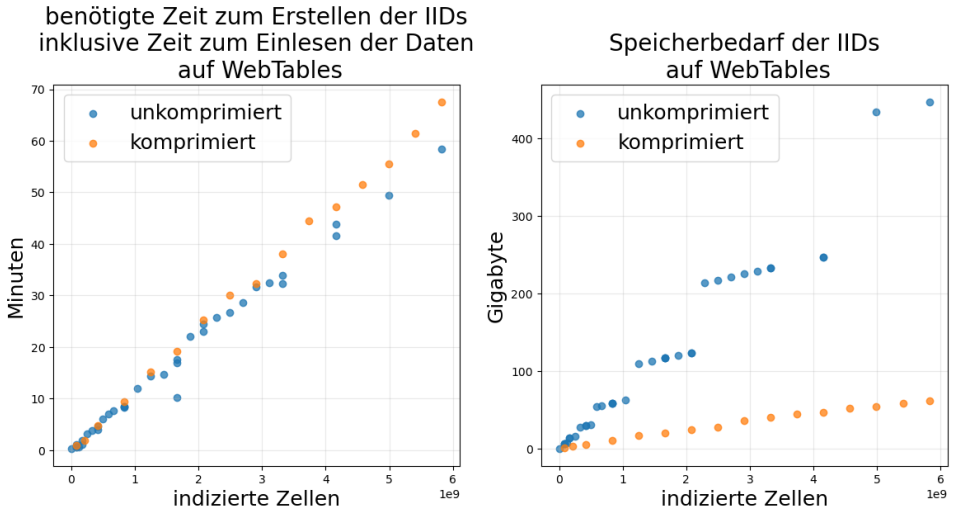


Abbildung 1.2: Graphen zeigen den Mehrwert von komprimierten IIDs

Im Rahmen der Arbeiten wurden Datenseen untersucht, aufgrund dessen finden sich Daten über Statistiken und Verteilungen von mehreren Seen als Ergebnis. Auf diesen Datenseen wurden viele verschiedene Kompressionsmethoden untersucht. Die Eignung von jedem einzelnem Ansatz an der gewählten Domäne kann in dieser Bachelorarbeit nachgelesen werden.

s

1.4 Herausforderungen

Während der Arbeit haben sich besonders beim Programmieren und Experimentieren verschiedene Herausforderungen gezeigt.

1.4.1 Messdaten verstehen

Eine immer wiederkehrende Herausforderung ist das Verstehen und Interpretieren von Messdaten. Dabei ist häufig ein Verständnis für tieferliegende Konzepte von Programmiersprachen und Computern erforderlich.

Ein paar konkrete Beispiele im Rahmen der These waren hierfür Folgende:

Fluktuierende Kompressionsrate der Baseline Messung

Die Baseline Messung zeigt einen treppenstufenförmig ansteigenden Speicherverbrauch. Diese zu interpretieren, erfordert Wissen über Implementierungen von Standarddatenstrukturen. Die Listen Implementierung von vielen Programmiersprachen alloziert doppelt so viel Kapazität wie nötig, nachdem diese aufgebraucht wird.

Beim Arbeiten besonders mit großen Datenmengen ist es immer wieder nötig, tieferliegende und konkrete Konzepte der unterliegenden Programmierung zu begreifen.

SMAZ verschlechtert Messergebnis nur nach Deduplizierung

Ein Beispiel für Messdaten, die widersprüchlich wirken. Diese kommen zustande, da vor der Deduplizierung sehr häufig vorkommende Strings automatisch stärker gewichtet wurden als danach. Und auf eben diese Strings ist SMAZ optimiert [11].

Integerkomprimierung verschlechtert Messergebnis

In den Experimenten war Integerkomprimierung hochgradig effektiv auf dedupli zierten Daten, während Experimente ohne diese Technik die Kompression verschlechtert haben. Nach der Deduplizierung werden Integer in größere Gruppen gruppiert, ohne diese Methode sind diese Gruppen stets nur drei Integer lang.

Diese drei Integer können und werden in der Baseline-Messung mit konstanter Größe gespeichert, ohne jegliche Pointerindirektion. Sobald man diese Werte komprimiert, ist jedoch eine variable Größe der Daten aufzutreffen, und es werden mindestens zwei Pointer benötigt. Sind die Gruppen von Integern lediglich drei Zahlen groß, ist der Speicherverbrauch von zwei Pointern bereits größer, als die Rohdaten der Zahlen überhaupt sein kann.

1.4.2 Entscheidungen

Es mussten viele subtile Entscheidungen getroffen werden, die Einfluss auf die Messergebnisse haben.

Wie soll die Kompressionsrate gemessen werden, wenn der Speicherverbrauch nicht linear ansteigt? Wie soll man damit umgehen, dass auf der Hardware nicht jedes Experiment ausgeführt werden kann? Schließlich haben manche Verfahren bei großen Datenseen einen zu großen Speicherbedarf.

Wie soll die Zeit zum Einlesen bewertet werden? Schließlich wurde eine Einlesemechanik entwickelt, welche speziell auf den Anwendungsfall optimiert ist.

Soll überschüssig allozierter Speicher mit eingerechnet werden? Es wurde sich dafür entschieden, diesen Überschüssigen Wert nicht herauszurechnen. Da der IID beim Erstellen sonst Spitzen an Speicherverbrauch hat, die die Messergebnisse nicht realistisch wiedergeben würden.Das würde wiederum dazu führen, dass zum Beispiel ein Verfahren aussieht, als würde es nur 20 GB Speicher benötigen, aber auf einem 24 GB System nicht angewendet werden kann, da es einen höheren Spitzenverbrauch hat. Zudem lässt sich dieser Speicher nicht für jedes Verfahren gleichermaßen sinnvoll herausrechnen.

1.4.3 Implementierung von Algorithmen

Für Forschungen ist ein wiederkehrendes Problem, dass Implementierungen von Algorithmen nicht in einheitlichen Programmiersprachen bereitstehen. Viele Algorithmen mussten für die Bachelorarbeit erst in Rust implementiert werden. Verfahren wie SIMDFastPFor [3] sind in ihrer ursprünglichen Sprache (C++) so stark optimiert, dass diese Leistung nicht ohne Weiteres garantiert werden konnte bei Übersetzungen. Hier mussten erst sichere FFI Schnittstellen programmiert werden, um die Vorteile der Implementierungen authentisch wiedergeben zu können.

Die Verfahren nicht auf Invertierte Indizes zugeschnitten oder auf sehr unterschiedliche Domänen optimiert. Mangels Ressourcen ist es so nicht trivial, das Verhalten von manchen Algorithmen auf den gegeben Eingabedaten zu interpretieren.

1.4.4 Samplen der Daten

Es ergibt Sinn, die Eingabedaten zu Samplen, um mehr Datenpunkte zu gewinnen. Eine Herausforderung ist, die Eigenschaften der Eingabedaten dabei so wenig zu verändern wie nur möglich.

1.4.5 Einlesen und sortieren der Messdaten

Das Sortieren und Einlesen der Messdaten aus einer Postgres Datenbank kann sehr lange dauern. Für WebTables waren dies mehrere Stunden. Es kam

regelmäßig zu Fehlern beim Einlesen, die sich nur subtil bemerkbar gemacht haben. Zu sehen waren diese unter anderem an einer unrealistisch niedrigen Entropie; es wurden nur die ersten n sortierten Zeilen der Datenbank gelesen und so kamen immer wieder dieselben Zelleninhalte als Eingabedaten.

Für die vorliegende Arbeit wurde ein neuartiges, komprimiertes Dateiformat entwickelt, um diese Daten effizient zu streamen. Das Format nutzt intern verschiedene Kompressionstechniken (Group-Varint-Encoding, Deduplizierung, SMAZ) und die Implementierung verwendet große Puffer, um die Leseperformance zu steigern.

Dieses Format wirkt sich nicht auf die Messergebnisse aus, lediglich auf eine sehr hohe Einlesegeschwindkeit der Daten. Deshalb wird diese Lesegeschwindigkeit bewusst herausgerechnet im Verlauf der Experimente.

1.4.6 Bugs in enorm großen Systemen

Beim Schreiben von Software treten immer Bugs auf.

Diese waren häufig nur sehr schwer nachvollziehbar, da mit so enorm vielen Daten gearbeitet wurde. Die Fehlerbedingung können oftmals nicht sinnvoll nachgestellt werden an minimalen Beispielen, während traditionelle Debuggingtechniken zwecklos bei den verarbeiteten Datenmassen sind.

1.4.7 Messungen

Mir war zu Beginn der Arbeit noch nicht klar, wie schwer es sein wird, den Arbeitsspeicher exakt zu messen. Das Betriebssystem hat keine Information darüber, welcher Speicher tatsächlich vom Invertierten Index selbst alloziert wurde und nicht nur beim Einlesen der Daten benötigt wird. Ich habe verschiedene Ansätze ausprobieren müssen, bis ich einen fand, der mir sinnvolle Daten liefert.

1.4.8 Finden von sinnvollen Messpunkten

In der Arbeit geht es vor allem und Kompression. Dazu gehört auch die Kompressions- und Dekompressionszeit. Diese isoliert zu messen, ergibt bei manchen Ansätzen keinen Sinn. Es wurde sich dazu entschieden als Proxy für diese Daten die Erstellungszeit und die Zugriffszeit in die Invertierten Indizes zu messen. Grund ist, dass beim Erstellen und Zugriff in der Praxisanwendung Probleme wie Cache-Misses auftreten. Diese sind häufig der größte Kostenpunkt in den Erstellungs- sowie Zugriffszeiten und so geben diese Werte sinnvollere Ergebnisse für realistische Praxisanwendungen wieder.

1.5 Struktur der Arbeit

In Kapitel 2 werden domänenspezifische Begriffe erläutert und die verschiedenen gewählten Algorithmen vorgestellt und erklärt. Zudem findet sich eine konkrete Erklärung der Problemstellung der Bachelorarbeit.

In Kapitel 3 werden Annahmen über die Eingabedaten, die verwendete Software, Hardware, und weitere Messbedingungen besprochen, sowie relevante Formeln für Messergebnisse erläutert. Es wird auf die Eingabedaten und die Methodik eingegangen.

Kapitel 4 bildet den Hauptteil der Arbeit. Hier werden die getätigten Experimente vorgestellt, Ergebnisse ausgewertet und auf diesen wird in Folgeexperimenten aufgebaut.

Kapitel 6.2 bildet den Schlussteil. Hier finden sich Abgrenzungen zu ähnlichen Arbeiten und Gedanken zu sinnvollen Forschungen, die auf den Ergebnissen dieser Arbeit aufbauen.

Kapitel 2

Grundlagen

Während Datenseen genereller sein können, wird der Begriff im Rahmen dieser Arbeit immer Synonym mit Datenseen von Tabellen verwendet.

Datenseen von Tabellen bezeichnen eine Menge an tabellarischen Daten in rechteckiger Form. Beispielsweise kann ein Ordner, gefüllt mit *.csv* Dateien als ein Datensee interpretiert werden. Datenseen können, anders als das Beispiel illustriert, sehr groß werden. Der größte in der Arbeit verwendete Datensatz umfasst etwa 125 Millionen verschiedene Tabellen [5].

Ein Invertierter Index ist eine Datenstruktur. Sie bildet einzelne Elemente auf die Menge an Mengen ab, in denen diese vorkommen. Von besonderer Bedeutung ist diese Struktur für Algorithmen wie in Josie [12] und DataXFormer [1] beschrieben, und ist essenziell für Suchmaschinen. Hier werden einzelne Wörter auf die Dokumente abgebildet, in denen diese vorkommen [10].

Im Rahmen der Arbeit liegen die Datenseen in invertierter Form vor, wie sie im Paper zu DataXFormer [1] beschrieben sind. Dabei ohne das 'term' Feld in zweifacher, sondern einfacher Ausführung. Zusätzlich wird angenommen, dass die Daten bereits alphabetisch vorsortiert nach Zelleninhalt sind. Diese Annahme wurde getroffen, da das Vorsortieren von Daten mittels Festplatte weitreichend verbreitet ist und somit für diese Bachelorarbeit mit Fokus auf Strukturen im Arbeitsspeicher nicht weiter interessant ist. Gleichzeitig ist dies essenziell für verschiedene Algorithmen, die in der Arbeit beleuchtet werden sollen.

Als Zellenposition wird das Tupel aus folgenden Werten bezeichnet:

1. Tabellen ID (fortführend TableID)
2. Spaltenindex (fortführend ColumnID)
3. Zeilenindex (fortführend RowID)

Als Zelleninhalt wird im Weiteren der Inhalt einer Tabellenzelle beschrieben. Für die Implementierung von IIDs wird dieser Tabelleninhalt grundsätzlich als Typ *String* interpretiert.

Als Zelle selbst wird im Verlauf der Arbeit der Zelleninhalt und die zugehörige Zellenposition bezeichnet.

Das heißt, die Daten sind als Liste von Zellen dieser Form vorliegend:

Feld	Typ
Zelleninhalt	String
TableID	32-Bit Integer
ColumnID	32-Bit Integer
RowID	32-Bit Integer

Im Rahmen der Arbeit werden verschiedene Implementierungen von IIDs betrachtet. Dabei wird von jeder Implementierung erwartet, dass diese eine Methode bereitstellt, um von einem *String* als Eingabe, auf eine Liste von *Zellenpostitionen* als Ausgabe abzubilden.

Kapitel 3

Auswertung

3.1 Problemstellung

Ziel der vorliegenden Arbeit ist es herauszufinden, welche Kompressionsmethoden sich sinnvoll kombinieren lassen und die Methode oder Kombination von Methoden zu finden, unter deren Verwendung der Speicherverbrauch von Invertierten Indizes möglichst gering ist.

Dabei wird von den Methoden erwartet, dass diese performant sind und das Arbeiten mit IIDs im Arbeitsspeicher ermöglichen.

Um sicherzustellen, dass die beste Methode als solche auch erkennbar ist, ist ein systematisches Vorgehen notwendig.

3.2 Eingabedaten

Die Eingabedaten wurden drei verschiedenen Datenseen von unterschiedlicher Größe entnommen.

Es wird nicht angenommen, dass im Vorhinein exakt bekannt ist, wie viele Daten in den Invertierten Index aufgenommen werden. Vor allem bedeutet dies, dass für die einzelnen IIDs nicht im Vorhinein bereits die genau benötigte Menge an Speicher angefragt wird.

3.2.1 Größe

Im folgenden Abschnitt wird die Größe der Rohdaten jeweils ohne jegliche Pointer angegeben, das heißt als Formel wird

$$Anzahl(Zellen) * (Durch.Zelllaenge + 12)$$

verwendet.

Diese gibt die Anzahl der Byte zurück, die für das Speichern aller Zellen und 12 Bytes für die Zellenposition gebraucht werden. Für die Verwendung der Daten im Arbeitsspeicher sind jedoch wesentlich mehr Bytes nötig. Unter

anderem werden in dieser Formel Pointer und Speicherausrichtung nicht berücksichtigt.

Ein realistischeres Ergebnis ist die tatsächliche Messung der Baseline in 4.1, welche jedoch mangels genügend Arbeitsspeicher den größten Datensee (WebTables) nicht darstellen kann.

1. Der German Open Data Corpus `https://www.govdata.de/` (im Folgenden OpenData bezeichnet). Größe etwa 15.99 Gigabyte

2. Die GitTables [6] mit einer Größe von etwa 38.8571 Gigabyte

3. der Dresden WebTables Corpus, welcher bei der Ausarbeitung des Papers [5] entstanden ist. Dieser hast eine Größe von 175.33 Gigabyte.

Um die Eigenschaften der Eingabedaten besser zu beleuchten, wird nicht nur die Größe, sondern auch die Verteilung der Eingabedaten (anhand ihrer Bitweite) betrachtet.

3.2.2 Verteilung der Eingabedaten

Unabhängig von der Größe der Datensätze, unterscheiden sich 'OpenData', 'GitTables' und 'WebTables' auch in weiteren Eigenschaften.

Abbildung 3.1 zeigt die Verteilung der Table-, Column- und RowID anhand der benötigten Bits, um diese zu kodieren (in anderen Worten, deren Bitweite). Verschiedene Farben wurden gewählt, um die unterschiedlichen Datenseen (OpenData, GitTables, WebTables) voneinander abzugrenzen.

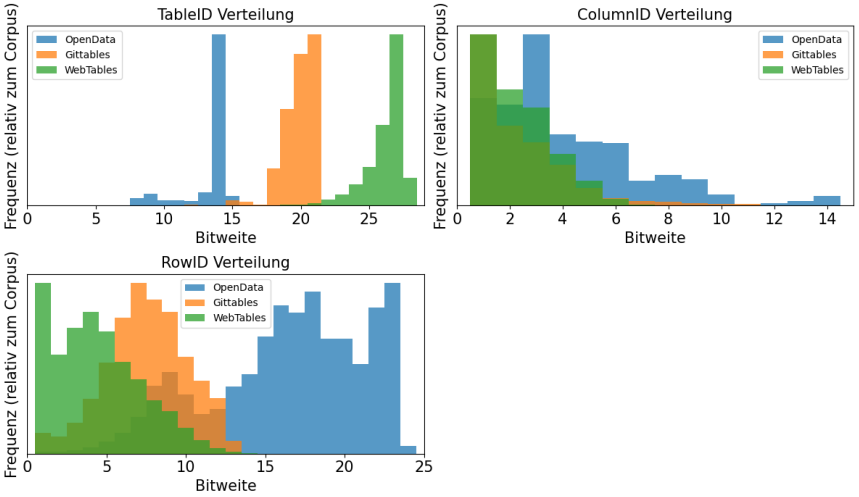


Abbildung 3.1: Der Graph zeigt, die Verteilung der TableID, ColumnID und RowID anhand ihrer Länge in Bit, bzw. Bitweite. Die Weite der Bits wird auf der x-Achse angegeben. Die Höhe der Balken zeigt an, wie häufig Zahlen mit der entsprechenden Bitweite im Datensatz zu finden sind. Dabei ist die Höhe immer relativ zum jeweiligen Datensatz normalisiert. So ist es leichter möglich, die verschiedenen Datensätze miteinander zu vergleichen, obwohl diese sehr unterschiedliche Größen aufweisen. Die unterschiedlichen Datenseen sind farblich kodiert.

Eine Aufteilung der Werte in ihre Bitweite wurde gewählt, da (die im Rahmen der Arbeit betrachteten) Intergerkompressionsalgorithmen (Group-Varint-Encoding, SIMDFastPfor) vor allem nach diesem Merkmal ihre Eingaben unterscheiden [3].

Die TableID ist in allen Datenseen zu erwarten gleichmäßig verteilt anzutreffen, da die TableID in keiner Korrelation zur Größe der ursprünglichen Tabelle steht. Eine TableID kommt genau einmal im Datensee je Zelle ihrer ursprünglichen Tabelle vor.

Man beachte, dass die Verteilung visuell exponentiell wirkt, da die IDs anhand ihrer Bitlänge in gemeinsame Eimer gelegt wurden, welche exponentiell mehr Zahlen abdecken (es gibt nur zwei Zahlen, die zwei-Bit weit sind, bereits 16 Zahlen sind vier-Bit weit u.s.w.).

Es fällt auch auf, dass in Corpus OpenData häufig sehr breite und auffällig lange Tabellen zu finden sind, wie sich an Column- und RowID ablesen lässt.

3.2.3 Zelleninhalte

Die Eingabedaten bestehen jedoch nicht nur aus Integern, sondern auch aus einem String-Wert, dem Zelleninhalt. In Tabelle 3.2.3 sind die durchschnittli-

che Länge dieser, sowie die durchschnittliche Frequenz, mit der individuellen Werte auftreten, auf zwei Nachkommastellen gerundet angegeben.

Corpus	Länge (Byte)	Frequenz
OpenData	7,99	40,45
GitTables	12,47	16,29
Main	9,09	12,89

Die Frequenz der Zelleninhalte ist besonders wichtig für das Thema dieser Bachelorarbeit, da dies jeweils redundante Informationen sind und starken Einfluss auf unterschiedliche Kompressionsalgorithmen hat.

Besonders im Corpus 'OpenData' treten individuelle Zellen redundant auf. Es ist zu erwarten, dass Deduplikation dadurch auf diesem Datensee besonders effiziente Kompression bietet. Damit zu Rechnen ist auch beim Arbeiten mit den 'GitTables', da hier neben einer mittelhohen Frequenz die redundanten Zellen deutlich mehr Speicher einnehmen, als bei den anderen Datenseen.

Alle Daten in diesem Abschnitt wurden mittels eines eigens dafür geschriebenen Programmes erhoben. Dabei wurde jeder Corpus vollständig durchlaufen.

3.3 Algorithmen

Es gibt verschiedene Kompressionsalgorithmen, die sich zur Evaluierung im Rahmen der Arbeit anbieten. Dabei werden viele verschiedene Techniken von diesen genutzt.

Während es Kompressionstechniken gibt, bei denen Information verloren geht ('Lossy' Kompression), werden im Rahmen dieser Arbeit lediglich Methoden betrachtet, bei denen sich die Eingabe vollständig wiederherstellen lässt. Es wird also ausschließlich 'Lossless' Kompression betrachtet.

In diesem Abschnitt werden diese Techniken vorgestellt.

3.3.1 Unkomprimierte IIDs

Der unkomprimierte IID ist modelliert, wie im Paper zu DataXFormer [1] beschrieben. Das heißt als Liste mit Zellen als Elemente, sortiert nach dem Zelleninhalt.

Das Finden von Elementen in dieser Struktur wurde durch zwei binäre Suchen implementiert; jeweils für das erste und das letzte Element einer zusammengehörigen Gruppe von Zellenpositionen.

Dieser Ansatz wird als Baseline für Messungen und Ergebnisse im Weiteren bezeichnet.

Die Baseline wird in Rust modelliert als

$$Vec < (String, TableLocation) >$$

wobei $TableLocation = (u32, u32, u32)$.

3.3.2 Deduplikationsstrategien

Bei allen Datenseen wird der Zelleninhalt im Schnitt mehrfach gespeichert. Innerhalb des Baseline-Verfahrens wird der Zelleninhalt je nach Datensee demnach größtenteils redundant gespeichert: Für WebTables etwa zwölf, für OpenData sind es sogar 39 redundante Kopien.

'Deduplizierung' beschreibt den Prozess, diese nur ein einziges Mal zu speichern und die restlichen Zellpositionen in einer Liste zusammen zu Gruppieren.

So wird der Zelleninhalt je nur ein mal gespeichert, und verbraucht weniger Speicherplatz.

Es gibt verschiedene Datenstrukturen, die sich eignen, um diese Abbildung vorzunehmen. Im Rahmen der Arbeit werden BTrees (beziehungsweise BTreeMaps) und HashMaps verwendet und untersucht.

HashMap

In Rust modellieren wir die IIDs mit HashMap als

$$HashMap < String, Vec < TableLocation >>$$

.

Der Hash eines Wertes ist eine Zahl mit konstanter Bitweite, welche sich deterministisch wieder errechnen lässt. Hashes sind in der Regel nicht auf ihren ursprünglichen Wert zurückzuführen.

HashMaps funktionieren grundsätzlich so, dass eine ungefüllte Liste alloziert wird, von Eingaben der Hashwert h genommen wird, und Elemente an Position $hmodlaenge(Liste)$ gespeichert werden.

So lässt sich auf die Werte innerhalb einer HashMap in $O(1)$ zugreifen.

Die benutze Implementierung der Rust-Standartbibliothek, ist ein Port von Googles 'SwissTable' Verfahren.

BTree

In Rust modellieren wir die IIDs mit BTree als

$$BTree < String, Vec < TableLocation >>$$

.

BTrees sind Bäume, die darauf optimiert sind, stets balanciert zu sein, während ihre Elemente in sortierter Form in der Struktur gespeichert sind. Elemente lassen sich über Binäre Suche so garantiert in $O(log(n))$ Finden.

Die genutzte Implementierung ist der Rust-Standartbibliothek entnommen und Quelloffen.

3.3.3 Kombinationsalgorithmen

Nachdem die Deduplikation angewendet ist, sind die Tabelleninhalte (Strings) und Tabellenpositionen (Integer) noch unkomprimiert.

Algorithmen, die auf diesen Daten arbeiten, lassen sich hervorragend mit Deduplizierung durch eine HashMap kombinieren.

Eine Ausnahme bilden die Präfixkodierungsverfahren, die später noch betrachtet werden. Diese komprimieren die Zelleninhalte dadurch, dass sie Information von mehreren verschiedenen Zellen in Relation zueinander betrachten. Diese Gruppierung lässt sich nicht trivial in dem HashMap-Verfahren abbilden, wo die Zelleninhalte stets isoliert betrachtet werden müssen, um als Schlüssel fungieren zu können.

3.3.4 Integer Kompression

Integerkompression ist sinnvoll, da ein großer Teil der Daten als Integer gespeichert werden. Im Folgenden wird ausgerechnet, wie groß dieser Anteil ist. Das erfolgt am Beispiel von deduplizierten IIDs, da bei diesen die Integer zu Listen gruppiert sind. Es ist zu erwarten, dass tatsächliche Messergebnisse höher sind, da die Unkosten verschiedener Datenstrukturen hier nicht einberechnet werden.

Die TableID, ColumnID und RowID sind allesamt als Integer mit je vier Byte kodiert. Der Datensee WebTables umfasst 645008253 individuelle Zellinhalte mit je etwa $9.1 Byte$. Für einen String in Rust werden jeweils 24 Byte (acht Byte für jeweils Start, Länge und Kapazität) zusätzlich benötigt.

also machen die Zellinhalte $645008253 * (9.1 + 24) Byte \approx 21.35 GB$ des Datensees nach Deduplikation aus.

Es gibt 8314523826 Tabellenpositionen mit je drei Zahlen à vier Byte. Auch diese haben 24 Byte Überschuss je individueller Zelle, da diese als Array gespeichert werden.

Mindestens etwa $115.25 GB$ werden für die Kodierung der Integer verwendet, was etwas über 84% des gesamten IIDs ausmacht.

Es liegt offensichtlich viel Wert darin, eine geeignete Kompression dieser Integer zu finden.

Die Kompression von Integern wird ausschließlich zusammen mit deduplizierenden Verfahren getestet, unter Verwendung einer HashMap. Grund ist, dass sonst pro Zelle je drei Integer komprimiert werden, was sehr wenig ist. Für jede einzelne Gruppe zahlt man Overhead an Speicher in Form von Pointern, da diese Daten von variabler Länge sind. Alleine die Information von Position und Länge würde 16 Byte kosten und damit bereits mehr, als die drei Integer direkt zu speichern.

Einige Algorithmen unterscheiden sich darin, dass diese Byte- oder Bit-orientiert sind. Byte-orientiert (englisch byte-aligned) bedeutet, dass ein Algorithmus so konzipiert ist, dass er nicht auf Bit-Ebene arbeitet, sondern auf Byte-Ebene. Dies wird getan, um Algorithmen möglichst performant zu machen. Bit-orientiert bedeutet, dass ein Algorithmus auf Granularität einzelner Bits arbeitet.

Letztere tauchen lediglich bei der Integer Kompression auf, zumindest im Rahmen der für diese Arbeit ausgewählten Algorithmen.

Eine Technik, die alle vorgestellten Verfahren nutzen, ist Null-Unterdrückung (englisch null-supression). Null-Unterdrückung ist eine Kompressionstechnik, bei der man Zahlen so kodiert, dass möglichst keine redundanten Nullen in den höherwertigen Stellen stehen. Beispielsweise kann man die Zahl 000101_b als 101_b darstellen und so mittels Null-Unterdrückung drei Bits einsparen, welche keine zusätzliche Information über den Wert der Zahl beinhalten.

Group-Varint-Encoding

Group-Varint-Encoding [10] [3] ist ein Byte-aligned Kompressionsalgorithmus, um Null-Unterdrückung anzuwenden.

Das Verfahren kodiert vier 32-bit Zahlen auf einmal, und speichert jede in variabler Länge (e.g. 1,2,3 oder 4 Byte) nacheinander ab.

Der Algorithmus wurde beschrieben in 'Introduction to Information Retrieval' [10]. Für diese Arbeit wurde er ausgewählt, da dieser sich in Invertierten Indizes in Suchmaschinen bereits erfolgreich und praxiserprobt bewährt hat. Ebenso funktioniert er bereits bei einer sehr kleinen Menge an Zahlen optimal.

Die Effizienz vieler Kompressionsalgorithmen ist stark domänenabhängig [3] und die gegebenen Eingabedaten weisen sehr unterschiedliche Verteilungen auf, wie die Bitweitenverteilung 3.1 zeigt. Die Ergebnisse von 'Lightweight Data Compression Algorithms: An Experimental Survey' [3] legen nahe, dass dieser Ansatz vielversprechend ist, da dieser sehr resistent gegen ausreißende Werte ist.

Es ist zu erwarten, dass vor allem besonders ausreißerresistente Algorithmen wie Group-Varint-Encoding (fortführend auch als GVE bezeichnet) auf allen Datenseen gleichermaßen effektiv sind.

Da das Komprimieren, wie auch das Dekomprimieren in der verwendeten Implementierung immer auf vier 32-Bit Zahlen arbeitet, erfolgen beide Prozesse in $O(1)$.

Die verwendete Implementierung wurde für diese Arbeit geschrieben und quelloffen veröffentlicht unter `https://github.com/nilsmartel/group-varint-encoding`. Die verwendete Implementierung hängt '0' Werte an die Gruppen an, um diese auf Längen von vier aufzufüllen.

V-Byte

V-Byte [10] ist ebenfalls ein Byte-aligned Algorithmus, der Null-Unterdrückung betreibt.

Die Kodierung erfolgt, indem eine Zahl in 7-Bit Segmente aufgeteilt wird. Jedes Segment wird als Bytes hintereinander geschrieben und der achte Bit jedes Bytes enthält die Information, ob das Folgeelement zur Zahl gehört. Es existiert eine SIMD Variante des Verfahrens mit identischer Kompressionsrate [3], welches aber Blocks von mindestens 4 Integern erwartet und deshalb nicht verwendet wird in der Arbeit.

Der Algorithmus wurde beschrieben in 'Introduction to Information Retrieval' [10]. Wie auch der Group-Varint-Encoding ist dieser Algorithmus im Information Retrieval praxiserprobt und wurde deshalb für diese Arbeit als potenziell nützlich anerkannt. Zumal ist dieser Algorithmus nicht darauf angewiesen, Zahlen gruppiert zu komprimieren. Er kodiert nur eine Zahl auf einmal, was einen Vorteil darstellen kann.

VByte ist jedoch als für sich stehender Algorithmus im Allgemeinen weniger effizient als GVE, wie [3] nahelegt. Deshalb wird diese Methode alleinstehend nicht betrachtet.

Sie wird jedoch innerhalb von Präfixkodierungstechniken verwendet, und im Rahmen der Experimente zusammen mit anderen Techniken weiter ausgewertet.

Theoretisch lassen mit VByte Kodierung unbegrenzt große Zahlen darstellen. Für eine Zahl n liegt dabei der Speicherverbrauch, Dekodierungsaufwand und Kodierungsaufwand bei $O(log(n))$.

Die verwendete Implementierung wurde für diese Arbeit geschrieben und quelloffen veröffentlicht unter `https://github.com/nilsmartel/vbyte-compression`.

SIMD-BP128

In dem Paper 'Lightweight Data Compression Algorithms: An Experimental Survey' [3] sticht unter anderem dieser Algorithmus durch besonders schnelle und effiziente Kompression heraus. Das Paper zeigt jedoch, dass der Algorithmus nicht gut mit Ausreißern umgeht. Da unsere Zellenpositionen unsortiert sind und Indizes über den gesamten Datensee verteilt liegen, ist dies jedoch bei uns der Fall.

Mit dem Algorithmus wird im Rahmen dieser Bachelorarbeit nicht weiter gearbeitet.

SIMDFastPfor

Im Folgenden ist unter SIMDFastPFor immer die Implementierung SIMDFastPFor128 gemeint, welche auf Gruppen von 128 Integern optimiert ist.

Ein weiterer Algorithmus, der in dem Paper [3] hervorragende Eigenschaften hat, ist SIMDFastPFor. Besonders auf schnelle Dekompression und hohe Kompressionsraten [3] ist dieser optimiert.

SIMDFastPFor [3] nutzt Null-Unterdrückung in Form von Frame-of-Reference-Encoding und ist Bit-orientiert. Das bedeutet: für eine Gruppe von Zahlen errechnet der Algorithmus Maximum und Minimum. Dann wird die Bitweite, um den Abstand zwischen diesen zu kodieren, sowie das Minimum als Frame-of-Reference gespeichert. Die Gruppe an Zahlen wird folgend als Abstand zum Minimum gespeichert. Dabei speichert man alle Zahlen der Gruppe mit einer konstanten Menge an Bits, nämlich so vielen, wie nötig sind, um alle Zahlen zwischen Minimum und Maximum zu kodieren, also $\lceil \log_2(Maximum - Minimum) \rceil$ pro Zahl.

Um diese Eigenschaften beizubehalten, wurde die originale C++ Implementierung [8] verwendet, und ein Forgein Function Interface geschrieben, um diese Bibliothek in Rust nutzen zu können.

Der in der Arbeit verwendete SIMDFastPFor Algorithmus [8] verwendet also die Implementierung des ursprünglichen Autors D. Lemire. Für die Bachelorarbeit wurden lediglich Bindings in die Rust-Programmiersprache geschrieben.

Der verwendete Algorithmus ist optimiert, um SIMD Operationen zur Performancesteigerung zu verwenden, erhöht aber nicht die Kompressionsrate.

Der SIMDFastPFor Algorithmus wurde ausgewählt, weil er hervorragende Kompressions- und Dekompressionsgeschwindigkeiten aufweist, sowie eine hohe Kompressionsrate. Während dies auch auf BP-128 zutrifft, scheint SIMDFastPfor dabei besser mit Ausreißern arbeiten zu können[8].

Ein großer Unterschied zwischen dem Paper [3] und dem Anwendungsfall an Datenseen ist, dass in zuerst genanntem Fall 100 Millionen 32-Bit Integer komprimiert werden, während diese Größe bei den IDDs stark schwankt. Sie hängt davon ab, mit welcher Frequenz f individuelle Zelleninhalte im jeweiligen Datenseen auftauchen. Unter der Annahme, dass wir TableID, ColumnID und RowID zusammen gruppieren, sind bei den IIDs im Schnitt also $f * 3$ Integer zu komprimieren. Typischerweise sind unsere Gruppen also 39 (WebTables) bis 120 Integer (OpenData) groß.

3.3.5 Stringkompressionsmethoden

Die eigentlichen Zelleninhalte sind dargestellt als *String* in unserem Anwendungsfall. Für diese gibt es ebenfalls Kompressionsmethoden, die auf unterschiedliche Weisen wirken.

Integer Kompressions-Techniken

Der Zelleninhalt lässt sich als Array von Bytes, bzw. Zahlen zwischen 0 und 256 interpretieren; deshalb werden in diesem Abschnitt Überlegungen evaluiert, Integer Kompressionstechniken auf diesem anzuwenden.

1. V-Byte: Bei V-Byte Kodierung werden redundante Nullen unterdrückt, indem ein Bit dafür verwendet wird anzugeben, ob weitere Bytes zu dem aktuellen Element gehören. Diese Technik auf UTF-8 kodierte Strings anzuwenden, ist redundant, da UTF-8 bereits exakt diese Technik nutzt, um Zeichen von variabler Länge zu kodieren.

2. Group-Varint-Encoding: GVE ist byte-aligned und unterdrückt redundante Nullen. UTF-8 kodiert Strings bereits byte-aligned und vermeidet redundante Nullen, durch die in bereits erwähnte 1 Technik.

3. SIMDFastPfor: SIMDFastPfor ist Bit-orientiert und somit von den genannten Techniken die einzige, die sinnvoll UTF-8 Strings komprimieren könnte. Die Länge der zu betrachtenden Strings ist jedoch recht klein. Bei WebTables im Schnitt nur etwa 9.1 Byte. Das wirkt sich negativ auf die Kompressionsrate aus, da SIMDFastPfor hierauf nicht optimiert ist. 4.3.2.

SMAZ

SMAZ [11] [9] ist ein auf kleine Strings und Performance optimierter Kompressionsalgorithmus, der besonders gut für das Komprimieren für Strings in englischer Sprache anzuwenden ist.

Der SMAZ Algorithmus [11] ist für Stringencoding ausgelegt. Er verwendet einen Lookuptable, um häufige Teile von Wörtern mit weniger Bytes zu kodieren. Der Algorithmus ist darauf optimiert, Strings in englischer Sprache zu kodieren, kurze Strings erfolgreich zu komprimieren und möglichst schnell zu sein.

Die Vermutung ist, dass diese Beschreibung auf die gegebenen Eingabedaten zutrifft, weshalb dieser Algorithmus im Rahmen der Bachelorarbeit ausgetestet wird.

In der Implementierung wird im IID anstelle des Zelleninhaltes der Byte-Array gespeichert, welcher als Resultat von $smaz(Zelleninhalt)$ erhalten wird.

Die verwendete Implementierung 'fastsmaz' [9] ist Quelloffen und nutzt denselben Lookuptable, wie das Original. Sie wurde öffentlichen Quellen entnommen und nicht selbst geschrieben im Rahmen der Arbeit.

Frontcoding

Da die Daten so eingelesen werden, dass sie nach den Zelleninhalten geordnet sind, ist zu erwarten, dass nacheinander folgende Zelleninhalte häufig einen gemeinsamen Präfix haben.

Deshalb bieten sich Verfahren wie Frontcoding oder Incremental-Coding an, die diese redundanten Präfixe eliminieren.

Beim Frontcoding [10] werden alphabetisch sortierte Strings gruppiert, dann wird je Gruppe der gemeinsame Anfang aller Strings gelöscht, nur einmal gespeichert, und die restlichen Strings werden hintereinander geschrieben, zusammen mit Information darüber, wie lang diese restlichen Strings sind. In Abbildung 3.3.5 wird dieses Schema an Beispieldaten gezeigt. Durchgestrichene Buchstaben sind Informationen, die durch Präfixkodierung wegfallen.

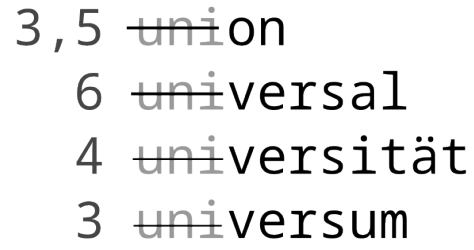


Abbildung 3.2: Eine Darstellung, wie Incremental-Coding das Speichern von redundanten Informationen verhindert

Die für diese Bachelorarbeit geschriebene Implementierung nutzt VByte Encoding, um Längeninformationen möglichst kompakt zu speichern.

Der Frontcoding Coding Algorithmus wurde beschrieben in 'Introduction to Information Retrieval' [10]. Der Algorithmus wurde im Rahmen der Arbeit eigenständig Implementiert.

Frontcoding wird zusammen mit Blocking (Blockgröße 8) verwendet, wie in [10] beschrieben.

Incremental-Coding

Das für die These implementierte Verfahren ist aus dem Paper 'Dictionary-based Order-preserving String Compression for Main Memory Column Stores' [2] abgewandelt.

Unterschiede zum Paper sind, dass kein Wert auf konstante Blocks gelegt wurde, um die Kompressionsrate zu erhöhen.

Incremental-Coding ist mit Frontcoding verwandt. Anstatt jedoch den gemeinsamen Präfix aller Strings in einer Gruppe zu eliminieren, wird der Präfix zum direkten Vorgänger String eliminiert. Der erste String jeder Gruppe ist unkomprimiert.

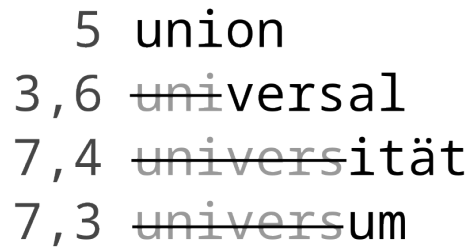


Abbildung 3.3: Eine Darstellung, wie Incremental-Coding das Speichern von redundanten Informationen verhindert

Gespeichert wird also für jeden String die Längeninformation. Hierbei ist es sinnvoll, nur die restliche Länge zu speichern, da diese Zahl kleiner gleich der Gesamtlänge ist. Außerdem wird die Länge des gemeinsamen Präfixes zum Vorgänger gespeichert, was beim ersten String stets null ist, und somit hier weggelassen wird. In 3.3.5 wird das Schema dargestellt. Die beiden Längeninformationen sind jeweils die erste und zweite Zahl auf der linken Seite. Die durchgestrichenen Buchstaben der vier Wörter stellen die Information dar, die durch Incremental-Coding reduziert wird.

Die gewählte Blockgröße von 16 wurde dem Paper [2] entnommen.

Die Vorteile des Verfahrens sind dieselben wie bei Frontcoding. Es ist jedoch möglich, dass sich die Kompressionsraten erhöhen, da lediglich der Präfix von zwei aufeinanderfolgenden Strings und nicht von acht auf einmal betrachtet wird. Es ist zu erwarten, dass dieser gemeinsame Präfix länger ist, als der einer ganzen Gruppe. Zusätzlich muss dafür jedoch für jeden Präfix individuell die Länge kodiert werden. Es ist experimentell herauszufinden, ob der zusätzliche Speicherbedarf der Länge oder das Ersparnis durch Speichern von weniger langen Präfixen überwiegt.

Diese Änderung ist subtil, und es ist zu erwarten, dass der Algorithmus sich sehr ähnlich wie Frontcoding verhält.

3.4 Methodologie

Verschiedene Algorithmen wurden im Rahmen der Arbeit ausgewertet. Für die Auswertung wird folgende Methodologie angewendet:

Zunächst wird beschrieben, aus welchen Gründen der spezifische Algorithmus geeignet für den Anwendungsfall erahnt wird.

Der Algorithmus wird dann in einem IDD implementiert und an Echtdaten getestet.

Es wird bei gegebenem Anlass auf Herausforderungen bei der Implementierung und Testung eingegangen.

Anschließend wird der Algorithmus anhand der erhaltenen Daten ausgewertet. Dabei wird besonders auf unerwartete Unterschiede eingegangen. Dies geschieht in der Regel anhand von einem Vergleich mit Daten verwandter Kompressionsansätze. Somit kann diese Auswertung nicht direkt für jeden Abschnitt erfolgen, sondern erst, nachdem eine Gruppe von Algorithmen getestet wurde.

Die Gruppierung und Gegenüberstellung von Algorithmen erfolgt nach dem Merkmal, ob diese inkompatibel zueinander zu verwenden sind. Damit ist gemeint, dass entweder der eine oder der andere verwendet werden kann, man diese aber nicht sinnvoll kombinieren kann, bzw. ein Kombinationsansatz nicht trivial ist.

Für das Auswerten der Kompression und des Speicherverbrauches wurde eine möglichst feingranulare Methode verwendet.

3.4.1 Messen der Kompressionsrate

Als Kompressionsrate eines Algorithmus wird der Faktor bezeichnet, um den die Ausgabe des Algorithmus kleiner ist, als die Eingabe. Ein Algorithmus mit Kompressionsrate 2 bildet also seine Eingabe der Größe b auf eine Ausgabe der Größe $\frac{b}{2}$ ab.

Sie ist immer in Bezug zur unkomprimierten Baseline gemessen.

Die Baseline wird im Rust-Code modelliert als

$$Vec < (String, u32, u32, u32) >$$

. Also einer Liste von Tupeln, welche jeweils die Zelle und die Zellenposition enthalten. Die Zellenposition wird modelliert mit drei positiven, 32-bit großen Integern.

Um die Kompressionsraten realistischer vergleichen zu können, wurden zwei Vorkehrungen getroffen.

Die Kompressionsrate lässt sich errechnen als

$$Kompressionsrate(a \in Algorithmen) = \frac{Speicher(Ausgabe_{baseline})}{Speicher(Ausgabe_a)}$$

.

Das setzt voraus, dass sich für jeden Algorithmus $Ausgabe_a$ errechnen lässt, was nicht für jeden Algorithmus der Fall ist.

Für zu große Eingabedaten lässt sich jedoch nicht jeder Algorithmus berechnen, da diese mangels limitiertem Arbeitsspeicher die Ausführung abbrechen.

Deshalb wurden die Eingabedaten gesampled und die Kompressionsrate gemessen als

$$Kompressionsrate(a \in Algorithmen) = \frac{BytesProZelle(Ausgabe_{baseline})}{BytesProZelle(Ausgabe_a)}$$

.

Wie 3.4 zeigt, fluktuiert die Anzahl an benötigten Bytes pro Zelle jedoch unterschiedlich stark bei verschiedenen Algorithmen. Besonders bei Verfahren ohne Deduplizierung fällt dies auf. Grund ist, dass hier Listen verwendet werden, und diese überschüssigen Speicher allozieren, wenn diese ihre Kapazität erreicht haben. Bei Listen in Rust wird stets die doppelte Menge der aktuellen Kapazität angefragt, weshalb die besonders nicht-kontinuierlichen Abschnitte des Graphen in immer wieder doppelt so großen Abständen liegen.

Damit Algorithmen wie Baseline keinen Nachteil davon erfahren, dass ein ungünstiges Sample genutzt wird, um ihre Kompressionsrate herzuleiten, wird stets die niedrigste und somit beste Messung an Bytes pro Zelle verwendet. Somit wird gewährleistet, dass die erreichten Kompressionsraten dieser Bachelorarbeit unter realen Bedingungen mindestens erreicht werden können. Die Messwerte würden andernfalls besser ausfallen, da besonders die Baseline als Referenzwert stark von Fluktuationen getroffen wird.

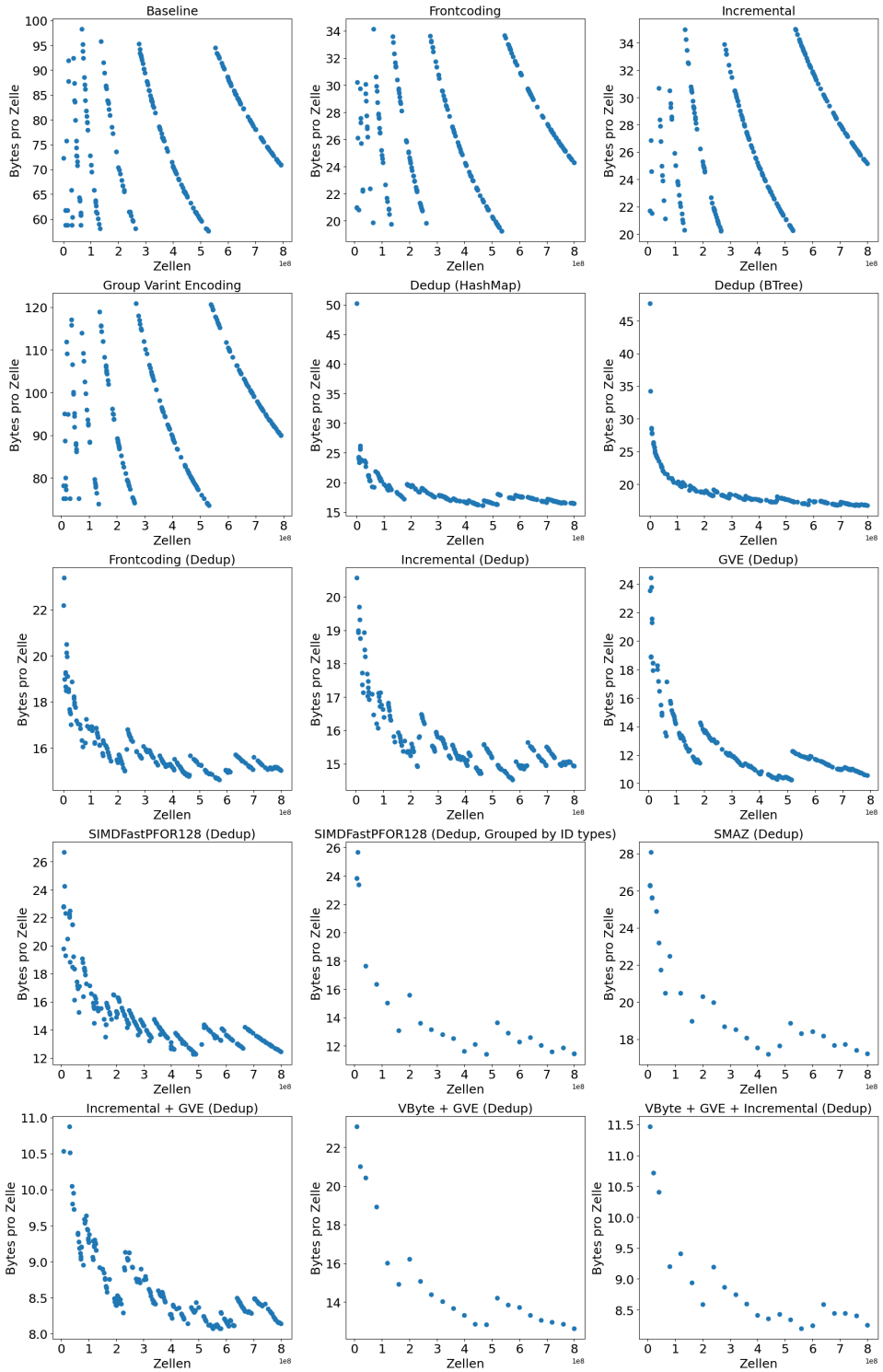


Abbildung 3.4: Die Anzahl an Bytes, die im Schnitt für eine Zelle gebraucht wird (y-Achse), hängt von der Anzahl an indizierten Zellen (x-Achse) ab. Niedrige Werte sind besser. Die Sample wurden anhand des Datensees 'OpenData' gemessen. Die Graphen geben nicht nur wieder, wie viele Bytes pro indizierter Zelle im Schnitt nötig sind. Es zeigt sich auch, dass dieser Wert unterschiedlich stark fluktuiert, je nach Algorithmus.

3.5 Versuchsaufbau

Der Versuchsaufbau berücksichtigt zahlreiche Faktoren. Um reproduzierbare Ergebnisse zu liefern, werden im Folgenden die Randbedingungen für die Experimente festgehalten und definiert.

3.5.1 Hardware und Betriebssystem

Alle Experimente wurden auf einem Linux Server durchgeführt. Der Server hat folgende Eigenschaften, welche für die Arbeit relevant sind:

1. 512 GiB Arbeitsspeicher (8 mal 64 GiB DDR4 RAM 3200MHz)
2. AMD EPYC 7702P Prozessor mit 64 Kernen und 128 Threads
3. x86_64 Architektur
4. Debian 5.10.149-2 Kernel

Nicht immer geschah dies im isolierten Betrieb, was zu leichtem Rauschen in den Messdaten zur Erstellungs- und Zugriffszeit beitrug. Um dieses Rauschen zu minimieren, wurden Experimente mehrfach durchgeführt und Ausreißer in den Daten manuell gefiltert. Schlussendlich sind die Zugriffszeiten reproduzierbar. Der Mehrbenutzerbetrieb hat keine nennenswerte Auswirkung auf die Aussagekraft der erhaltenen Daten. Auf die erreichten Kompressionsarten hatte dies keinen Einfluss.

Pointer

Pointer sind Zeiger auf Adressen im Arbeitsspeicher.Eigenschaften Sie sind zwingend erforderlich, um Daten von variabler Länge zu strukturieren. Im Rahmen der Arbeit wird immer davon ausgegangen, dass Zeiger 8 Byte groß sind. Auf der verwendeten Hardware ist dies der Fall. Es existieren Plattformen, für die diese Annahme nicht zutrifft. Da diese in der Regel jedoch maximal $2^{32}Byte = 2GB$ Arbeitsspeicher ansprechen können, sind sie für das Thema dieser Arbeit irrelevant.

Wie viel Kontrolle man über die Verwendung von Pointern hat, hängt von der verwendeten Programmiersprache ab.

3.5.2 Verwendete Programmiersprache

Die Implementierung der IIDs erfolgte in der Sprache Rust (Version 1.67.0-nightly), kompiliert unter dem 'release' Profil. Die Sprache wurde gewählt, da

1. Bei Bedarf Strukturen ohne Pointer Overhead (wie in Java und Python der Fall) generiert werden können.
2. Kein Garbage Collector genutzt wird, der zusätzlichen Speicher benötigt.
3. Der Compiler garantieren kann, dass kein Speicher geleakt oder nach dem deallozieren weiter genutzt wird.

3.5.3 Allokatoren

Ein Allokator ist in der Programmierung dafür verantwortlich, dass dynamisch unterschiedliche Mengen an Speicher angefordert werden können. Dabei steht es dem Allokator frei, mehr Speicher als nötig zurückzugeben, was als Optimierung gängige Praxis ist. Da das Verhalten von unterschiedlichen Allokatoren Auswirkungen auf den Speicherbedarf der IDD-Implementierungen hat, wird im Rahmen dieser Arbeit der Standardallokator des Betriebssystems (welcher von OS zu OS unterschiedlich ist) vermieden, und der gängige Allokator jemalloc [4] in Version 5.x verwendet.

Zudem bedient sich die Implementierung der IIDs zu großen Teilen der Standardbibliothek von Rust, welche quelloffen ist. Besonders die BTree, HashMap und Listen (e.g. Vec) Implementierungen sind entscheidend.

3.5.4 Optimierungen von Algorithmen

Eine Auffälligkeit ist, dass sich Algorithmen in der Regel deutlich optimieren lassen [3] für bestimmte Anwendungsfälle, oder in Spezialfällen bestimmte Datenstrukturen besser performen.

Ein in der Arbeit wiederkehrendes Beispiel hierfür sind die Standard-String und -Listen (‘Vec<T>‘) Typen in der Standardbibliothek von Rust. Diese allozieren bei wiederholten ‘push‘ Operationen immer etwas mehr Speicher, als exakt nötig. So muss nicht bei jedem einzelnen Hinzufügen eines Elementes die gesamte Datenstruktur neu kopiert werden, man kann einfach in den bereits befreiten Bereich schreiben. Darüber hinaus belegen beide genannten Strukturen mindestens 24 Byte, davon 8 für Byte, um die gesamte Kapazität zu speichern. Häufig sind diese redundant für untereren Anwendungsfall.

SIMD-Operationen und Arenaallozierung sind weitere Beispiele für gängige und relevante Optimierungen.

SIMD bezeichnet bestimmte Rechenoperationen, welche massiv die Geschwindigkeit eines Programmes erhöhen können. Es erfordert eine bestimmte Strukturierung der Algorithmen und nicht jedes Problem eignet sich, diese Optimierung ausnutzen. SIMD Operationen lassen sich immer durch semantisch äquivalente Standardoperationen ersetzen.

Im Rahmen dieser Arbeit werden grundsätzlich erst einmal die Standard-Datenstrukturen der Rust Bibliothek und keine expliziten Optimierungen (besonders Arenaallozierungen) verwendet, außer diese sind bereits der direkteste Weg, um ein Problem zu lösen. Dies ist beim Nutzen von C/C++ Funktionen häufig der Fall.

Optimiert wird, wenn eine Methode besonders vielversprechend ist, oder eine besondere Fragestellung dadurch besser erkundet werden kann.

Anzumerken ist, dass die Rust Standardbibliothek bereits sehr optimiert ist.

3.5.5 Probenahme

Um ein deutlicheres Bild darüber zu bekommen, wie sich die Kompressionsstrategien auf Datenseen von unterschiedlicher Größe verhalten, wurden zu jedem Datenseen Sample verschiedener Größe genommen und als Eingaben ausgewertet.

Es wurde darauf geachtet, dass die Sample wahrhaft zufällig ausgewählt wurden.

Bei dem Entnehmen von Samplen der Eingabedaten, wurde jede einzelne Zelle mit einer Wahrscheinlichkeit p ausgewählt, sodass die indizierten Zellen etwa gleich der gesamten Zellen mal p sind.

Ein alternativer und performanterer Ansatz ist, nur die ersten n Zellen in den IDD aufzunehmen. Dabei würden sich aber die Eigenschaften der Eingabedaten stark verändern. Zum Beispiel hätte man sehr wahrscheinlich viele mit 'A' beginnende Zellen im Corpus und weniger wahrscheinlich solche, die mit 'Z' anfangen. Dies würde zu stark die Entropie der Eingabedaten senken, um für alle Algorithmen realistische Ergebnisse zu zeigen.

Die Sample wurden bei jeder Messung neu gewählt. Daraus resultiert, dass zwei Messungen, mit sehr ähnlicher Anzahl von indizierten Zellen, leicht unterschiedliche Ergebnisse liefern können.

3.5.6 Messen des Speicherbedarfes

Um den Speicherbedarf einzelner Strukturen in einem Programm zu messen, muss innerhalb des Applikation selbst Code dafür geschrieben werden. Das Betriebssystem und externe Werkzeuge haben keinen Zugriff auf diese Informationen, die über die Granularität von den insgesamt angeforderten Memory-Pages des Programmes hinaus gehen.

Um den Speicherbedarf eines Invertierten Index zu messen, wird sichergestellt, dass das Programm auf nur einem Thread läuft. Zunächst wird der gesamte Speicherbedarf a gemessen, der IID wird vollständig dealloziert, dann wird Speicherbedarf b gemessen. Hierfür lässt sich die API von jemalloc nutzen. Der gesamte Speicherbedarf des IID ist die Differenz a - b. Da wir keine parallelen oder konkurrenten Threads haben, kann in der Zwischenzeit kein weiterer Zugriff auf den Allokator stattfinden, der die Ergebnisse verfälscht. Der Rust Compiler garantiert, dass das Deallozieren vollständig und ohne Fehler geschieht. Das Resultat ist auf den Byte genau und beinhaltet auch den Speicherverbrauch durch Pointer, überschüssig allozierten Speicher von Strukturen, und vom Allokator aufgerundete Mengen an Speicher.

Dieses Verfahren wurde gewählt, weil es möglichst realitätsnah den tatsächlichen verbrauchten Speicher wiedergibt, und weitreichend Rechenfehler ausschließt (zum Beispiel, dass Pointer oder überschüssig allozierter Speicher vergessen werden einzurechnen)

Aus dem Speicherverbrauch des Invertierten Index ergibt sich die Kompressionsrate.

Die Technik zum Messen des Speicherbedarfs einzelner Strukturen ist entnommen aus [7].

3.5.7 Messen der Erstellungszeit

Je nach verwendetem Algorithmus dauert es unterschiedlich lange, den Invertierten Index zu Erstellen. Dabei ist der größte Zeitfaktor beim Erstellen das Lesen der Eingabedaten von der Festplatte, da diese wesentlich langsamer ist, als der Arbeitsspeicher. Dazu sei gesagt, dass es noch länger dauert, die Daten vorzusortieren, diese Zeit wird jedoch im Weiteren nicht berücksichtigt.

Für alle Messungen von Algorithmen stellt das Einlesen der Eingabedaten einen theoretisch konstanten Zeitaufwand dar. In der Praxis ist dieser von Mal zu Mal etwas unterschiedlich, was von verschiedenen Faktoren abhängt. Durch häufigeres Samplen wird dieser Effekt ausreichend minimiert.

Um etwas Kontext zu geben, wie viel Zeit der gesamte Erstellungsprozess in Anspruch nimmt, ist in Abbildung 3.5 ein Überblick dieser Zeiten zu sehen. Dieser wurde aufgeteilt nach den genutzten Eingabedaten, welche zusätzlich mehrfach gesampled wurden. Anzumerken ist, dass eine besonders inperformante Implementierung von 'Group-Varint-Encoding' herausgerechnet wurde, um die Lesbarkeit zu verbessern.

Es zeigt sich, dass für den Corpus

1. 'OpenData' zwischen 6 und < 12 Minuten gebraucht wird,

2. 'GitTables' in der Regel nach 16 ± 1 Minuten indiziert sind,

3. und der 'WebTables' Corpus in der Regel über 80 bis 90 Minuten zum Indizieren benötigt.

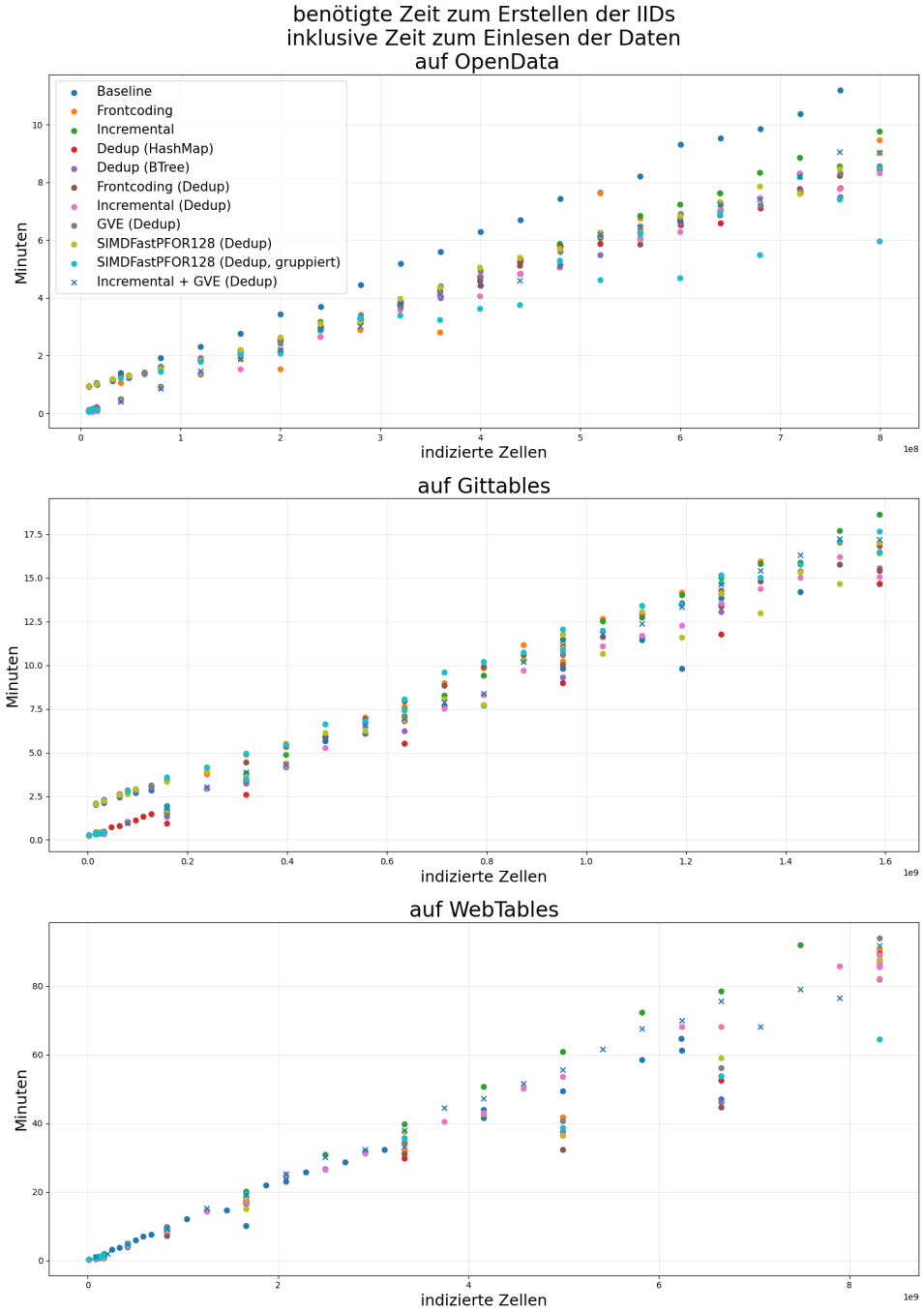


Abbildung 3.5: Die x-Achse zeigt die Anzahl an indizierten Zellen. Anhand der y-Achse lässt sich erkennen, wie lange der Erstellungsprozess (in Minuten) gedauert hat.

Anhand der Färbung der Punkte im Graphen lassen sich unterschiedliche

Algorithmen differenzieren. Es zeigt sich, dass bei keinem Algorithmus signifikant mehr Zeit in Anspruch genommen wird, als bei anderen.

Der größte erreichte Unterschied findet sich am Beispiel des Datensees 'OpenData', wo 'Group-Varint-Encoding' mit Deduplikation zweimal langsamer ist, als der bestmögliche Algorithmus.

Die Daten werden dabei nicht aus einer Datenbank direkt gelesen, sondern liegen vorsortiert und in komprimierter Form in einem speziell für die Arbeit angefertigtem Dateiformat vor, damit diese möglichst effizient gestreamt werden können. Beim Arbeiten mit Datenbanken ist ein langsameres Einlesen zu erwarten, was den Unterschied zwischen den Algorithmen weiter schrumpfen lässt.

In Folge der geringen Unterschiede bei der gesamten Erstellungszeit wird diese als weniger ausschlaggebend bei der Bewertung der Algorithmen betrachtet. Im Rahmen der Arbeit wird diese nur noch in Sonderfällen betrachtet, zum Beispiel bei besonders effizienten Lösungen und um zwei Ansätze zu vergleichen.

Im Verlauf der restlichen Arbeit wird die Erstellungszeit ohne die konstante Einlesezeit betrachtet. Um diese Zeit herauszurechnen, wurde beim Erstellen der IIDs nur dann die Zeit gemessen, wenn ein Element hinzugefügt wird, das bereits im Arbeitsspeicher geladen ist. Danach wird die Zeitmessung pausiert, bis das nächste Element bereit ist.s

Besondere Maßnahmen im Hinblick auf Caches

Die Caches von CPUs sind Speicher, die bedeutend schneller als der Arbeitsspeicher sind, dafür in der Regel auch wesentlich kleiner. Greift man auf Speicher zu, der nicht in diesen Caches geladen ist, spricht man von einem Cache-Miss. Cache-Misses können die Effizienz eines Programmes massiv verschlechtern.

Bei Messungen muss daher darauf geachtet werden, dass keine Cache-Misses durch die Messung auftreten oder vermieden werden, die ohne Messung nicht entstanden wären.

Beim Stream der Eingabedaten wurde besonders darauf geachtet, nicht die CPU-Cache zu überschreiben, mit der der IDD geladen ist. Hierzu wurden die Daten in einem eigenen Betriebssystem-Thread eingelesen und nur in kleinen Einheiten an den Thread übergeben, der den IDD erstellt.

Das eigentliche Erstellen des IIDs erfolgt stets auf nur einem Thread.

3.5.8 Messen der Abrufzeit

Für das Messen der Abrufzeit wurden je gesampleten IID etwa zehntausend Schlüssel (also Zellen) zufällig dem Invertieren Index entnommen. Es wurde für jeden dieser Schlüssel sequentiell die entsprechenden Zellenpositionen abgerufen, und die durchschnittliche Zeit gemessen.

Ausreißer in den Daten entstehen vor allem dann, wenn einer der zufällig gewählten Zellen besonders häufig in Tabellen vorkommen. Folglich ist die Menge der resultierenden Zellenpositionen entsprechend groß. Es dauert wesentlich länger, diese zu dekomprimieren oder zu kopieren. Ein solches Beispiel hierfür ist der Zellenwert ‘0‘. Im WebTables Corpus kommt dieser Wert fast 366 Mio. Mal vor. Die durchschnittliche Frequenz von individuellen Zellen liegt dabei gerade mal bei etwa 12. Damit ist dieser Wert über 30 Millionen mal häufiger vertreten, als üblich.

Um die Werte nicht durch solche Ausreißer zu verfälschen, muss man bei den nicht-deduplizierenden Algorithmen beim Wählen der Schlüssel acht geben. Entnimmt man zufällige Elemente aus der Menge der Zellen, werden diese Ausreißer bedeutend wahrscheinlicher gewählt, da sie mit viel höherer Frequenz repräsentiert sind im Datensatz. Dadurch wird der Einfluss von diesen auf die Messzeit besonders stark gewichtet.

Für die Arbeit wurden deshalb erst die Schlüssel in eine ungeordnete, deduplizierte Menge umgewandelt, und anschließend wurde von dieser Menge zufällig ausgewählt.

Es musste hierbei darauf geachtet, dass der Speicherbedarf dieser Menge nicht in den gemessenen Speicherbedarf der IIDs einfließt.

Kapitel 4

Experimente

4.1 Baseline

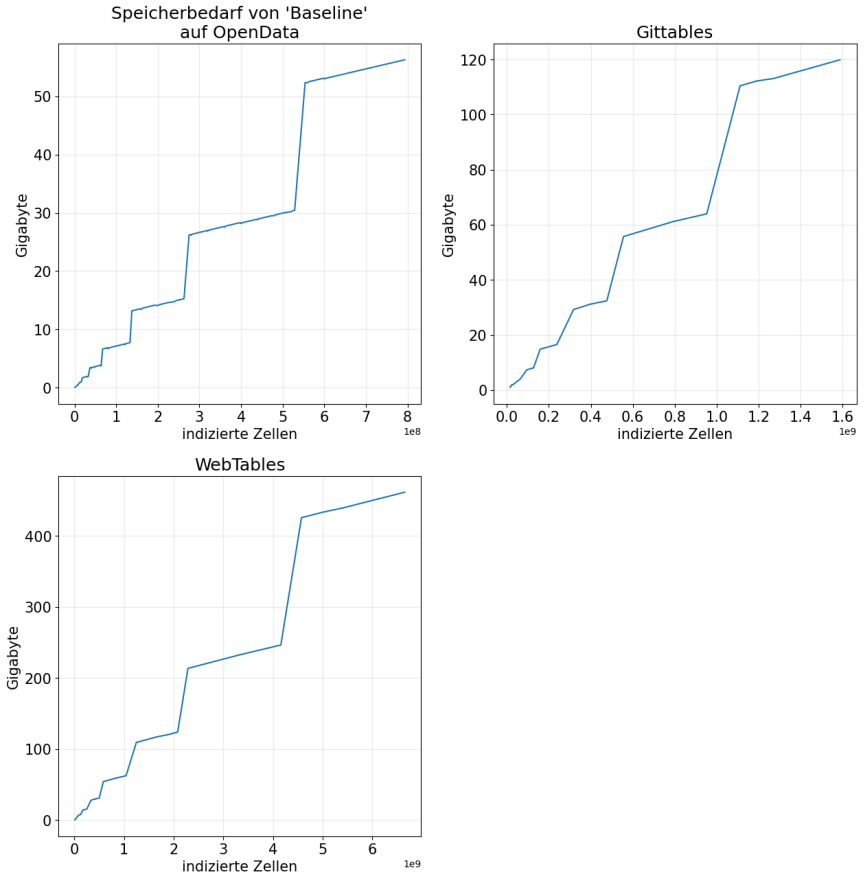


Abbildung 4.1: Der Graph stellt den tatsächlich benötigten Arbeitsspeicherbedarf für die Baseline anhand der drei Datenseen dar. Die x-Achse zeigt die Anzahl der indizierten Zellen, die y-Achse die Menge an angefordertem Speicher in Gigabyte. Der Corpus WebTables konnte nicht vollständig geladen werden, nur gesampelte Teilmengen von diesem.

In Abbildung 4.1 zeigt sich ein treppenstufenförmig ansteigender Speicherbedarf des Baseline Algorithmus. Das liegt daran, dass wiederholt Elemente einer Liste hinzugefügt werden, welche in bestimmten Abständen zusätzlichen Speicherplatz reserviert. Die Listen-Implementierung von Rust fordert beim Hinzufügen von Elementen doppelt so viel zusätzlichen Speicher an wie aktuell belegt, wenn dieser keine Kapazität für neue Elemente hat. Dies ist eine Standardtechnik, welche nicht Rust-spezifisch ist.

Vor allem ist zu beachten, dass der Corpus Main nicht vollständig in den Arbeitsspeicher geladen werden kann. Der gesamte Corpus umfasst mehr als $8 * 10^9$ Zellen.

4.2 Deduplikation

Es gibt zwei Datenstrukturen, deren Effizienz für Deduplizierung ausgewertet wird.

4.2.1 BTree und HashMap

Abbildung 4.2 zeigt, dass die Deduplikation sehr effektive Kompression bietet. Gegenübergestellt sieht man an der blauen Linie, dass die unkomprimierte Baseline-Messung deutlich mehr Speicher belegt. Es zeigt sich also, dass die Deduplikationsverfahren sehr effektive Kompression bieten. Der Unterschied zwischen BTree- und HashMapkompression ist hingegen sehr gering. In Bezug auf Speichereffizienz scheinen diese Verfahren austauschbar.

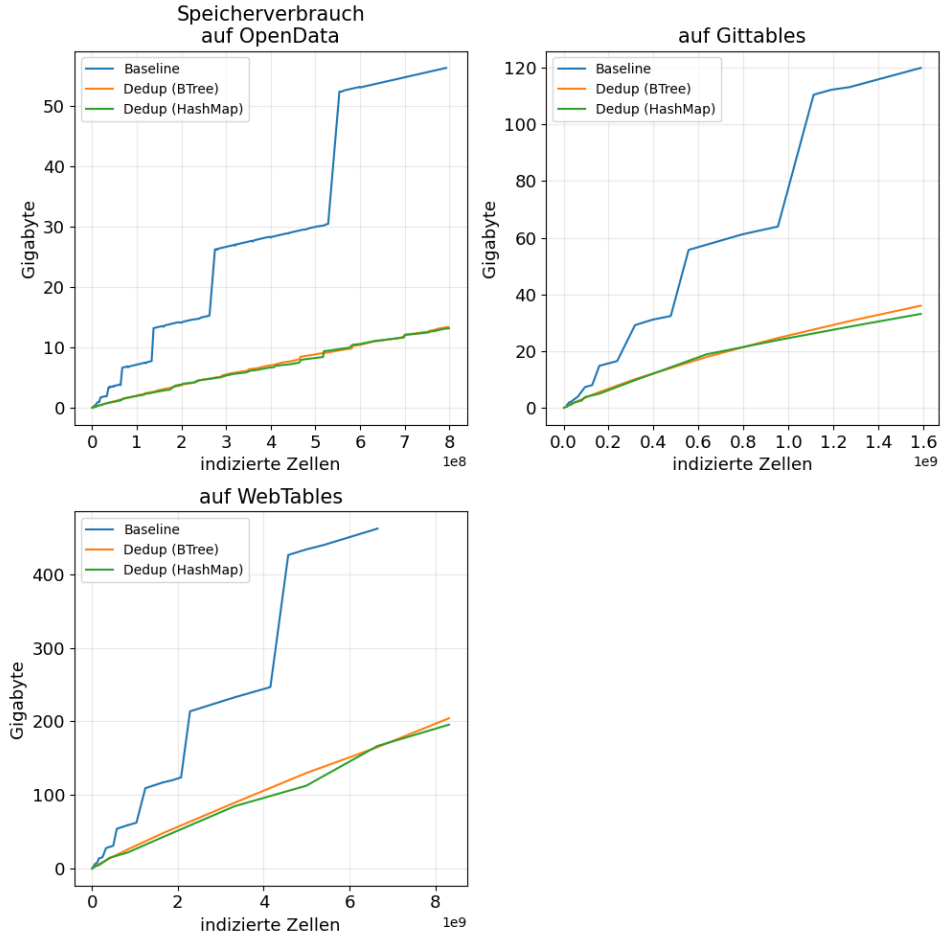


Abbildung 4.2: Der Graph zeigt, wie sich der Speicherverbrauch in Gigabyte (y-Achse) in Bezug zu der Menge der indizierten Zellen (x-Achse) verändert.

Es gibt verschiedene Ebenen, auf denen die beiden Verfahren verglichen wurden, unter anderem die Erstellungszeit der IIDs. Abbildung 4.3 zeigt diese Erstellungszeit. Es ist anzumerken, dass aus dieser Erstellungszeit die Zeit zum Laden der Eingabedaten herausgerechnet worden sind. Wie zu erwarten, ist die Baseline deutlich schneller zu erstellen, als die beiden anderen Verfahren. Allgemein lässt sich auch sagen, dass die HashMap mehr Zeit in Anspruch nimmt, um erstellt zu werden, als der BTree. Diese zeitlichen Kosten fallen jedoch wesentlich geringer aus, als vorher angenommen. Zumal bei der Erstellungszeit noch Unkosten durch das Laden der Eingabedaten hinzukommen; weshalb dieser zeitliche Unterschied in der Praxis stark an Wichtigkeit verliert.

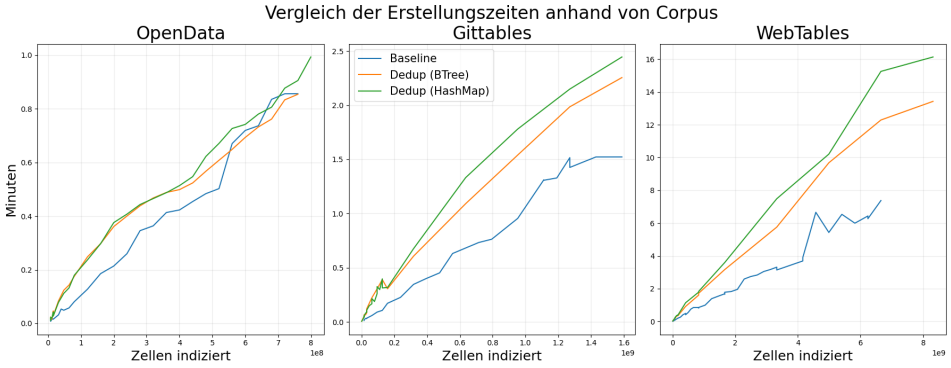


Abbildung 4.3: Der Graph zeigt, wie viel Zeit benötigt wird, um die IIDs zu bauen (y-Achse, Einlesezeiten herausgerechnet) in Bezug zu der Menge der indizierten Zellen (X-Achse). Es zeigt sich, dass mit zunehmenden Zellen der Unterschied zur Baseline wächst und sich die Erstellungszeit verschlechtert.

Was in der Praxis jedoch sehr wichtig ist, sind die Zugriffszeiten. Messungen zu dieser sind in Abbildung 4.4 festgehalten.

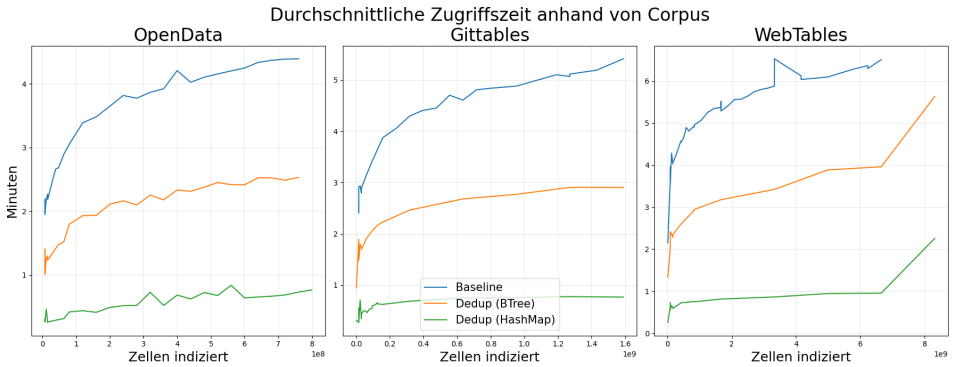


Abbildung 4.4: Der Graph zeigt, wie schnell diese Zugriffszeit (y-Achse) in Bezug zu der Menge der indizierten Zellen (x-Achse) verändert. Es zeigt sich, dass auf die HashMap bedeutend schneller zugegriffen werden kann, als auf BTree oder Baseline Strukturen.

Während der BTree etwas schneller zu erstellen ist, ist dieser bedeutend langsamer im Zugriff, als die HashMap. 0.28 Als Grund dafür kommen vor allem Cache-Misses bei der Binären Suche an: Bei der Binären Suche werden unterschiedliche Teile des Baumen durchforstet. Unsere Daten sind mehrere Gigabyte groß und daher ist anzunehmen, dass diese unterschiedlichen Teile jeweils nicht in der Cache vor dem Zugriff liegen. Abbildung 4.4 zeigt zudem, dass die Baseline etwa doppelt so viel Zeit beim Zugriff benötigt, wie der BTree. Bei der Baseline werden zwei Binäre Suchen getätigt, was diese Annahme weiter unterstützt. Zudem erklären Cache-Misses, warum die HashMap so unwahrscheinlich effizient ist; Hier wird anhand des Hashes der Daten direkt ihr Index herausgefunden, und so kommt es nur zu einem Speicherzugriff, bei dem ein Cache-Miss geschehen kann.

Die Erstellungszeit ist ein weniger wichtiges Merkmal von Algorithmen, als die Zugriffszeit. Die Erstellungszeit wird in realen Anwendungsfällen unabhängig durch einen konstanten Faktor ergänzt, nämlich das Einlesen der Rohdaten. Dadurch sinken die Unterschiede der Algorithmen relativ zueinander deutlich in der Praxis.

Die schnellere Zugriffszeit der HashMap ist ausschlaggebend dafür, dass im Rahmen der Bachelorarbeit weiter mit dieser gearbeitet wird. Der BTree wird nicht weiter betrachtet.

4.2.2 Effektivität der Deduplikation

Es ist anzunehmen, dass die Kompressionsrate von Deduplikationsverfahren abhängig davon ist, wie viele doppelte Zelleninhalte in den Datenseen vorliegen.

In Abbildung 4.5 wird dies anhand der Kompressionsrate der HashMap verglichen. Die Kompressionsrate ist dabei in Bezug zur Baseline gemessen.

Es zeigt sich, dass diese Werte wie erwartet etwa korrelieren.

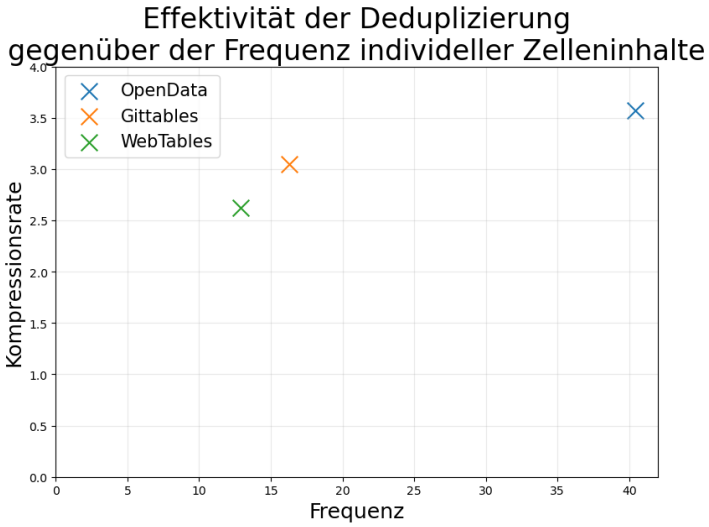


Abbildung 4.5: Der Graph stellt das Verhältnis der Frequenz individueller Zelleninhalte zur Kompressionsrate dar.

4.3 Integer Kompression

4.3.1 Group-Varint-Encoding

Die Ergebnisse der Messungen werden in Abbildung 4.3.1 dargestellt. Es zeigt sich, dass die Kompression gegenüber den unkomprimierten Integern konsistent erfolgreich ist, und den Speicherbedarf der IIDs senkt.

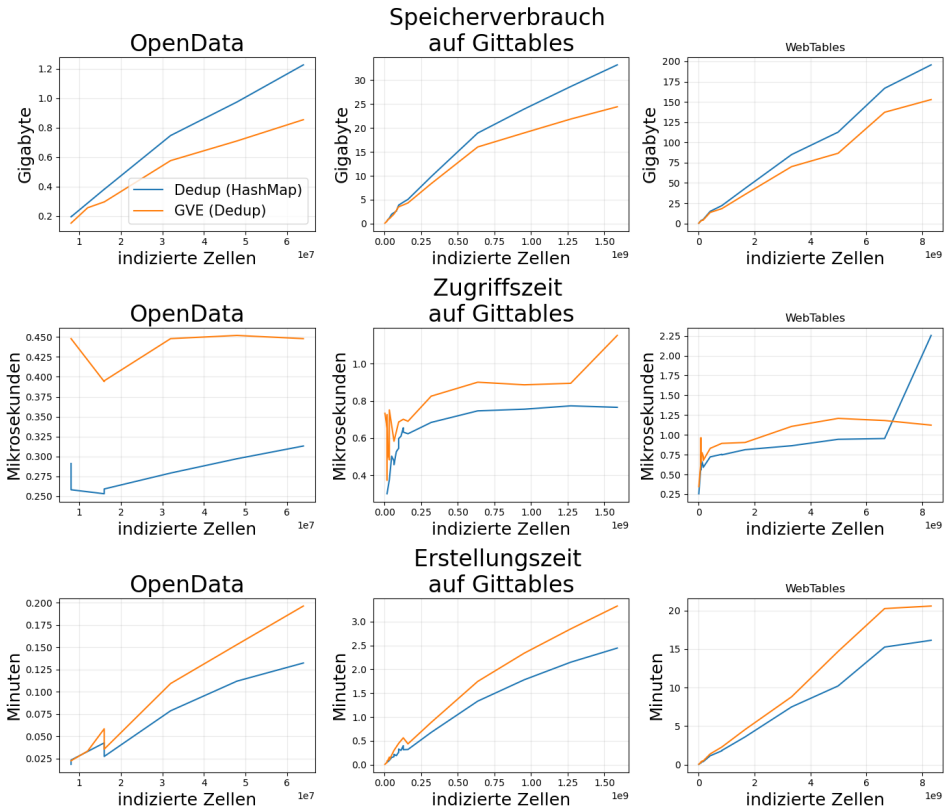


Abbildung 4.6: Graphen, die verschiedene Messwerte Experimenten mit Group-Varint-Encoding gegenüberstellen.

Die Kompressionsrate gegenüber der alleinstehenden Deduplizierung ist etwa

1. $\frac{1{,}22GB}{0{,}85GB} = 1.435...$ auf OpenData
2. $\frac{33GB}{25GB} = 1.32$ auf GitTables
3. $\frac{200GB}{150GB} = 1.333...$ auf WebTables

Sowohl beim Erstellen, als auch beim Zugriff ist der Overhead des Komprimieren und Dekomprimieren der Tabellenpositionen negativ bemerkbar

4.3.1. Es ist damit zu rechnen, dass das Nutzen des GVE Ansatzes das Arbeiten mit den IIDs verlangsamt, jedoch lediglich um einen konstanten Faktor ≤ 1.5. Dieser Wert wurde anhand der Erstellungs- und Zugriffszeit der Messungen auf OpenData gewählt, wo der Einfluss besonders ausgeprägt ist. Die Erstellungszeit ist im Echtenanwendungsfall jedoch zu vernachlässigen, da hier noch der konstante Zeitfaktor für das Einlesen der Daten hinzukommt.

Das Arbeiten mit den IIDs ist also weiterhin möglich, wenn auch etwas langsamer. Da der Fokus der Arbeit auf Kompression liegt, wird das GVE Verfahren als erfolgreich betrachtet, und im Verlauf der Arbeit weiter genutzt.

4.3.2 SIMDFastPFor

Obwohl SIMDFastPFor resistenter ist gegen Ausreißer, als Verfahren wie BP-128 [3], wird das Verhalten dennoch von ihnen beeinflusst (anders als es bei Group-Varint-Encoding der Fall ist).

Deshalb wurden für das Verfahren zwei Implementierungen geprüft, beide zusammen mit Deduplikation mittels HashMap. Bei einer Implementierung wurden TableID, ColumnID und RowID separat komprimiert (in Abbildung 4.3.2 als 'SFP (IDs separiert)' bezeichnet). Bei der anderen wurden diese Integerfelder zusammen gruppiert. In der Abbildung 4.3.2 als 'SFP' zu sehen. Dadurch werden zwar Ausreißer provoziert, da die Integer aus verschiedenen Domänen mit unterschiedlichen Eigenschaften gemischt werden, jedoch werden die zu komprimierenden Gruppen größer und weniger.

Der bisher beste Kompressionsansatz ist Group-Varint-Encoding (in der Abbildung 4.3.2 als 'GVE' abgekürzt). Kombinationen von SIMDFastPFor und GVE sind wenig vielversprechend. Obwohl SIMDFastPFor auf Bitlevel arbeitet und GVE lediglich auf Bytelevel, vollziehen beide Algorithmen Null-Unterdrückung. Das heißt, zuerst GVE und dann SFP zu kombinieren würde GVE redundant machen. Erst SFP und dann GVE zu nutzen ist ebenfalls nicht sinnvoll, da SFP bereits deutlich feiner arbeitet, als GVE.

Die Algorithmen stehen also in direkter Konkurrenz zueinander, und so werden sie in Abbildung 4.3.2 zusammen betrachtet. Man beachte, dass hier zum Teil verrauschte Daten abgebildet sind. Besonders auf OpenData sind die Daten stark verrauscht, da hier Versuche häufig parallel durchgeführt wurden, um effizienter Datenpunkte zu erhalten. Diese wurden ungefiltert dargestellt, da sich auch mit den Ausreißern ein sehr deutliches Bild zeigt.

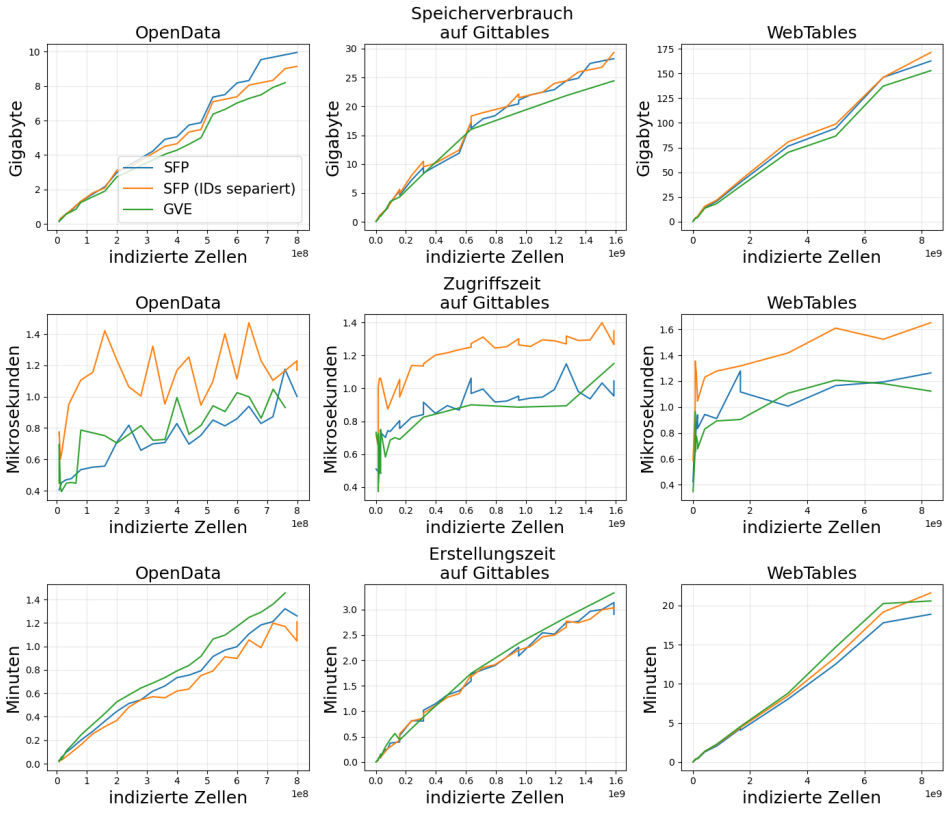


Abbildung 4.7: Graphen, die verschiedene Messwerte aus Experimenten mit SIMDFastPFor und Group-Varint-Encoding gegenüberstellen.

In Abbildung 4.3.2 zeigt sich, dass das Separieren der IDs auf den GitTables und den WebTables die Kompressionsrate verschlechtert, während diese auf OpenData etwas besser wird. Grund ist, dass auf OpenData meist deutlich größere Gruppen von Integern auf einmal komprimiert werden, durch die höhere Frequenz individueller Zelleninhalte.

Im Allgemeinen ist der SIMDFastPFor Algorithmus in der Domäne der IIDs jedoch weniger effektiv als Group-Varint-Encoding. Grund ist, dass die Gruppen von Integern hier nicht groß genug sind, damit SIMDFastPFor effektiv ist. Zudem ist Group-Varint-Encoding völlig unbeeinflusst von Ausreißern. Die Messungen des Speicherverbrauches auf OpenData legen nahe, dass SIMDFastPFor dadurch verschlechtert wird, sonst würde das Separieren der IDs keinen Vorteil bringen.

Das Separieren der IDs verschlechtert auch die Zugriffszeit. Auf die Erstellungszeit hat es allerdings keinen Einfluss. Die Dekompressionsgeschwindigkeit von SIMDFastPFor und GVE sind sehr ähnlich.

Die Erstellungszeit (und somit Kompressionsgeschwindigkeit) ist eben-

falls vergleichbar.

Im Verlauf der Arbeit wird mit SIMDFastPFor nicht weiter gearbeitet, da es sich grundsätzlich nicht sinnvoll mit GVE kombinieren lässt und der GVE Ansatz erfolgreicher komprimiert.

Neben den Tabellenpositionen sind die Tabelleninhalte selbst ein unabhängiger Bereich, der Potenzial für Kompression bieten.

4.4 String Kompression

Dieser Teil der Arbeit bezieht sich auf das Komprimieren des Zelleninhaltes. Der Zelleninhalt ist in den Eingabedaten in Form eines UTF-8 kodierten String präsent.

4.4.1 SMAZ

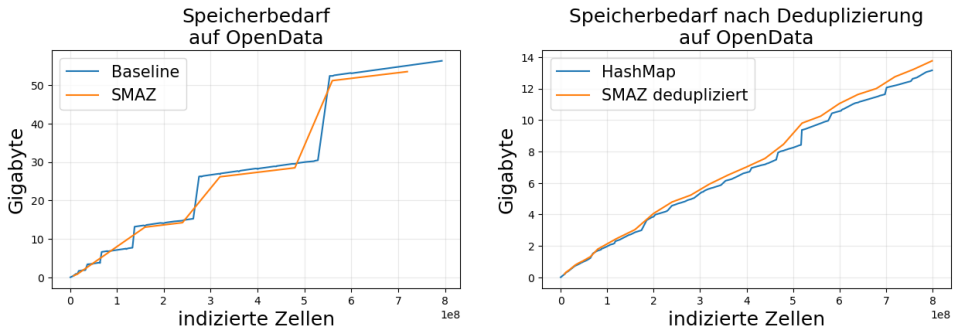


Abbildung 4.8: Messergebnisse von Experimenten mit SMAZ Kompression.

In Abbildung 4.8 zeigen sich zwei Graphen, die Ergebnisse aus verschiedenen Experimenten repräsentieren. Links sehen wir Experimente ohne den Einsatz von Deduplikationstechniken, der rechte Graph zeigt Ergebnisse aus deduplizierten Experimenten. Die Duplikatentfernung wurde für letztere mittels HashMap durchgeführt, auch für die Kurve 'SMAZ dedupliziert' ist diese zum Einsatz gekommen. In den Experimenten ohne Deduplikation senkt die Anwendung von SMAZ den Speicherverbrauch leicht, und erhöht so die Kompressionsrate.

Da die Deduplikation gegenüber der Baseline so effektiv ist, wird auch dieses Kombinationsverfahren überprüft an dieser Stelle. Auffällig ist, dass unter Anwendung von Deduplikation der Speicherverbrauch sogar erhöht wird, verglichen mit selben IDD ohne SMAZ Anwendung. Dies lässt sich damit erklären, dass SMAZ darauf optimiert ist, häufig vorkommende Wörter und Wortfragmente zu komprimieren ('the' wird zum Beispiel als ein Byte gespeichert [11]).

Gleichzeitig ist zu erwarten, dass eben diese häufig vorkommenden Strings auch mit hoher Redundanz in den Datenseen zu finden sind. Daher werden sie im Fall ohne Deduplizierung automatisch stärker gewichtet. Kommt ein String hundertmal vor, nimmt er hundertmal mehr Speicher ein und jeder eingesparte Byte an einer Instanz wird hundertfach eingespart.

Im Fall mit Deduplizierung werden selten vorkommende Zelleninhalte genauso stark gewichtet, wie diese besonders häufig vorkommenden. Auf diese ist jedoch SMAZ nicht optimiert und so verschlechtert sich die

Kompressionsrate.

Die Varietät der gegebenen Eingabedaten ist zu hoch, um von SMAZ sinnvoll behandelt zu werden. Im Zuge der Bachelorarbeit wird die Weiterarbeit mit diesem Kompressionsverfahren nicht fortgeführt.

4.4.2 Frontcoding

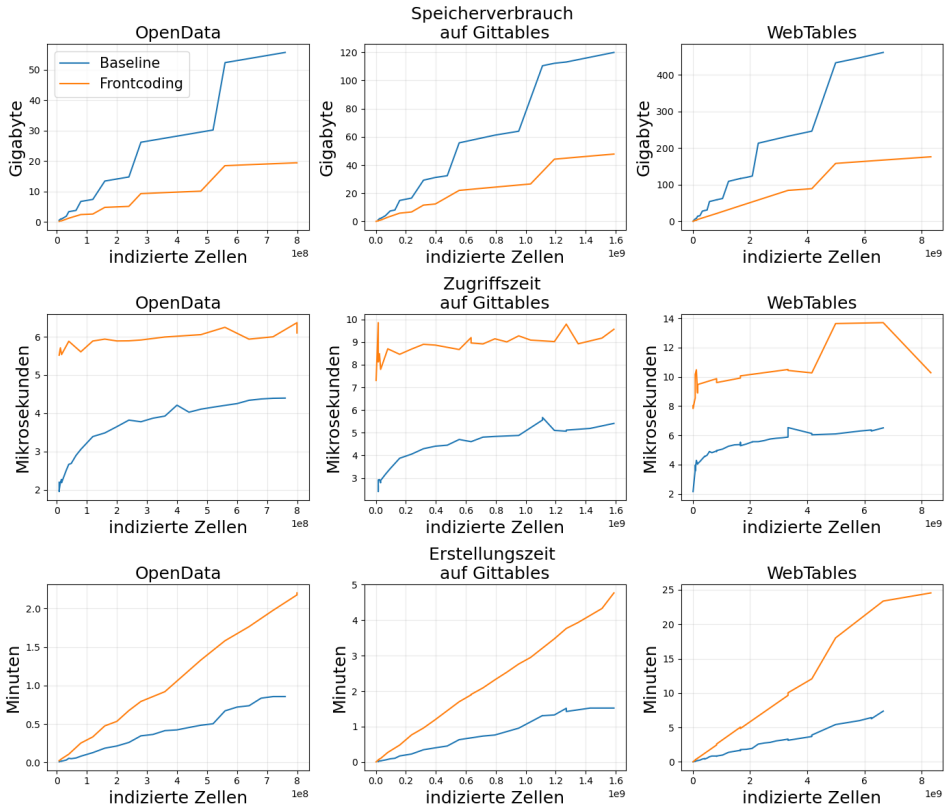


Abbildung 4.9: Messergebnisse unter Anwendung der Frontcoding-Technik, gegenübergestellt zur Baseline

Wie in Abbildung 4.4.2 zu sehen ist, ist die Kompressionsrate durch Anwendung von Frontcoding konsistent fast dreimal so groß, wie die der Baseline.

Die Kompressionszeit und Dekompressionszeit ist deutlich höher, als die der Baseline.

Diese Implementierung ist nicht auf Performance optimiert und es ist zu erwarten, dass hier potenziell noch schnellerer Zugriff möglich ist. Zum Beispiel eignet sich SIMD, um den gemeinsamen Präfix von vielen Strings zu vergleichen, was jedoch aktuell nicht implementiert ist.

Da der Algorithmus vielversprechend ist, wird er weiter untersucht.

4.4.3 Incremental-Coding

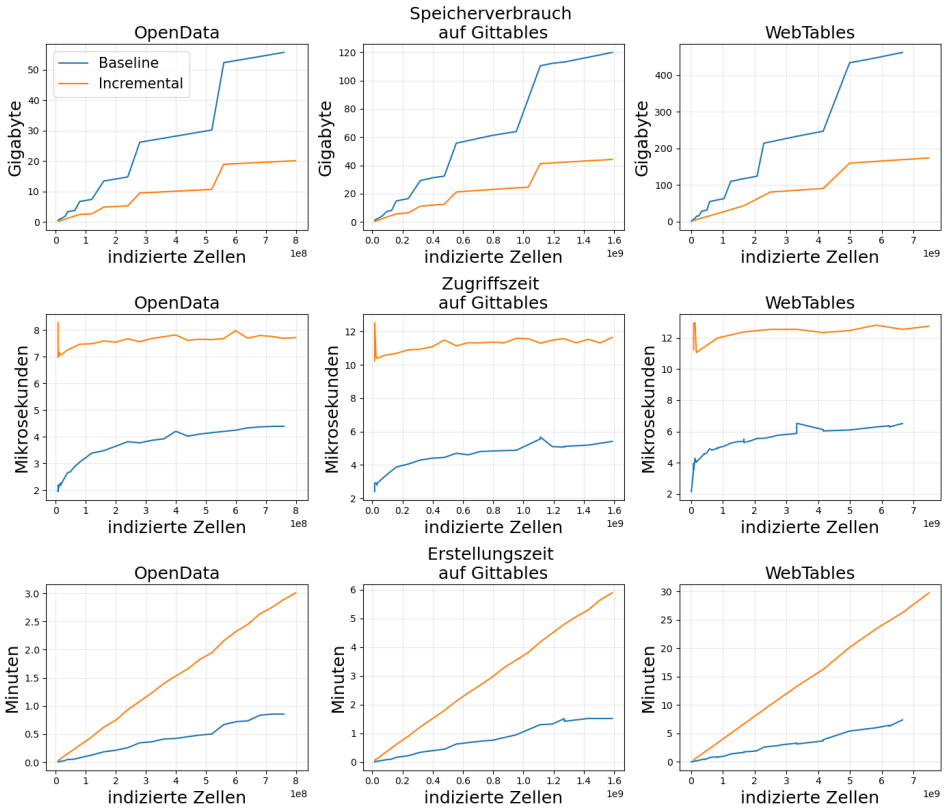


Abbildung 4.10: Messergebnisse unter Anwendung von Incremental-Coding, gegenübergestellt zur Baseline.

Wie Abbildung 4.4.3 zeigt, weist Incremental-Coding dieselben Vor- und Nachteile wie Frontcoding 4.4.2 auf.

Die Techniken sind eng miteinander verwandt und nutzen zum Komprimieren beide dieselben Redundanzen in den Eingabedaten aus, welche in der Ausgabe deutlich verringert sind. Somit ist eine Kombination der beiden Algorithmen nicht sinnvoll. Sie schließen sich gegenseitig für kombinierte Kompressionstechnik aus.

Die Algorithmen werden deshalb in direkten Vergleich gestellt. Beide Algorithmen zeichnen sich durch eine hohe Kompressionsrate und lange Kompressions- und Dekompressionszeiten (bzw. Erstellungs- und Zugriffszeit) aus. Da beide Algorithmen verhältnismäßig schlechte (De-) Kompressionszeiten nicht zuletzt aufgrund ihrer Implementierung aufweisen, wird diese nicht weiter berücksichtigt. Sollte sich herausstellen, dass ein Ansatz besonders vielversprechend ist, wird dieser weiter optimiert.

Ebenfalls wurde der deduplizierte Fall der Algorithmen betrachtet: Es ist zu erwarten, dass die Algorithmen sich leicht anders verhalten, da Frontcoding eine Sequenz identischer Strings besser komprimieren kann, als Incremental-Coding. Grund ist, dass bei Incremental-Coding für jedes Element die Länge des Präfixes zusätzlich speichert. Im Falle der deduplizierten Daten ist jedoch keine Sequenz von identischen Zelleninhalten möglich. Da bei beiden Verfahren die Zelleninhalte nicht isoliert voneinander komprimiert werden, ist eine Implementierung der Deduplizierung mittels HashMap nicht sinnvoll. Es wurde für beide Algorithmen ein eigener Ansatz implementiert, der ausnutzt, dass die Zelleninhalte sortiert eingelesen werden. So muss für die Deduplizierung beim Einlesen nur das nächste Element mit dem aktuellen verglichen werden, um die Information zu erhalten, wann eine neue Gruppe von Zellen beginnt.

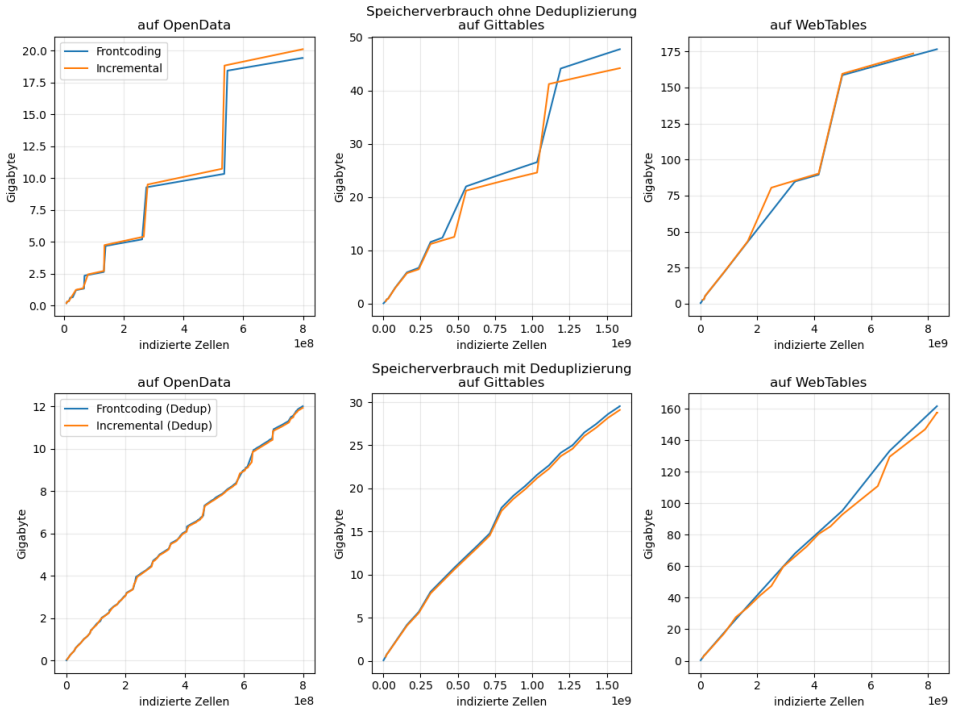


Abbildung 4.11: Vergleich zwischen Incremental-Coding und Frontcoding.

Wie Abbildung 4.4.3 zeigt, verhalten sich die Algorithmen sehr ähnlich. Auf OpenData und ohne Deduplizierung ist Frontcoding wie zu erwarten etwas besser im Komprimieren der Daten, da im Schnitt > 40 identische Wiederholungen von jedem Zelleninhalt präsent sind. Dabei muss Incremental-Coding jedes Mal die Länge des Präfixes speichern, obwohl dieser sehr häufig konstant innerhalb einer Gruppe ist, nämlich das gesamte Wort. Im

Allgemeinen ist jedoch das Incremental-Coding Verfahren leicht effizienter, besonders bei dem deduplizierten Verfahren.

Die HashMap ist nicht trivial mit Incremental-Coding kombinierbar. Deshalb werden auch diese Algorithmen in direkten Vergleich gestellt. Dabei wird die deduplizierte Variante des Incremental-Coding betrachtet.

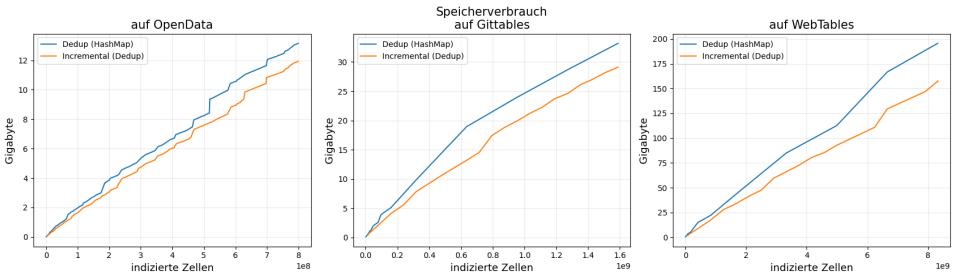


Abbildung 4.12: Vergleich zwischen Incremental-Coding und HashMap zur Deduplizierung.

Wie Abbildung 4.4.3 zeigt, ist der Vorteil durch Incremental-Coding deutlich sichtbar. Besonders auf den WebTables wird deutlich, bei welchem der Speicherbedarf der Incremental-Coding Lösung lediglich 80% von dem der HashMap Implementierung entspricht.

Da die Incremental-Coding Technik so erfolgreich ist, wird diese weiter untersucht. Es ist anzumerken, dass in der getesteten Implementierung ineffiziente Routinen genutzt werden. Diese haben auf die Kompressionsrate keine Auswirkung, bieten jedoch sehr viel Potenzial für Optimierungen.

Weiteres Vorgehen

Insbesondere wird getestet, ob sich eine effizientere Implementierung findet und ob sich durch dedizierte Datenstrukturen noch Pointer einsparen lassen, um den Speicherverbrauch weiter zu senken. In der hier gezeigten Implementierung wurden die individuellen Blöcke von Zelleninhalte in Listen gespeichert, die einen für unseren Anwendungsfall redundanten Pointer zum Speichern der Kapazität besitzen.

Zudem ist die Zugriffszeit potenziell deutlich zu reduzieren. Da beim Incremental-Coding das erste Element immer unkomprimiert vorliegt, ist es beim binären Suchen erst im letzten Schritt nötig, innerhalb des Blocks Strings zu vergleichen. Davor ist es grundsätzlich möglich, Vergleiche nur an dem ersten, unkomprimierten Stringwert durchzuführen. Aktuell wird bei jedem Suchschritt der gesamte Block dekomprimiert. Der richtige Block ist genau dann gefunden, wenn das erste Element vom aktuell betrachteten Block kleiner gleich des Suchelementes ist und das des Folgeblockes größer ist.

4.5 Kombinationsansätze

4.5.1 Incremental-Coding, Deduplizierung und Group-Varint-Encoding

Verschiedene Kompressionstechniken lassen sich ausgezeichnet kombinieren. Besonders dann, wenn diese unterschiedliche Redundanzen in den Eingabedaten ausnutzen.

Die Zelleninhalte lassen sich hervorragend durch Präfixkompression mit Incremental-Coding verkleinern.

Deduplizierung ist sehr allgemein anwendbar, wobei sich hier die HashMap und das Verfahren für Incremental-Coding besonders hervorgetan haben. Es bleibt zu überprüfen, ob das Incremental-Coding Verfahren performant genug sein kann, um mit der HashMap zu konkurrieren.

In Folgeexperimenten wird dafür eine optimierte Implementierung verwendet, welche dasselbe Kompressionsschema verwendet.

Die Zellenpositionen zu komprimieren, hat sich besonders mit Group-Varint-Encoding als erfolgreich herausgestellt.

Daher wird im Folgenden betrachtet, wie effizient diese drei Techniken (Incremental-Coding, Deduplizieren und Group-Varint-Encoding) zusammen wirken. In Abbildung 4.5.1 als 'Incremental + GVE (Dedup)' bezeichnet. Die Implementierung von Incremental-Coding ist bereits optimiert, wie in 4.4.3 beschrieben.

Zusätzlich wird Deduplizierung mittels HashMap ausgewertet, welche in kombination mit Group-Varint-Encoding implementiert ist.

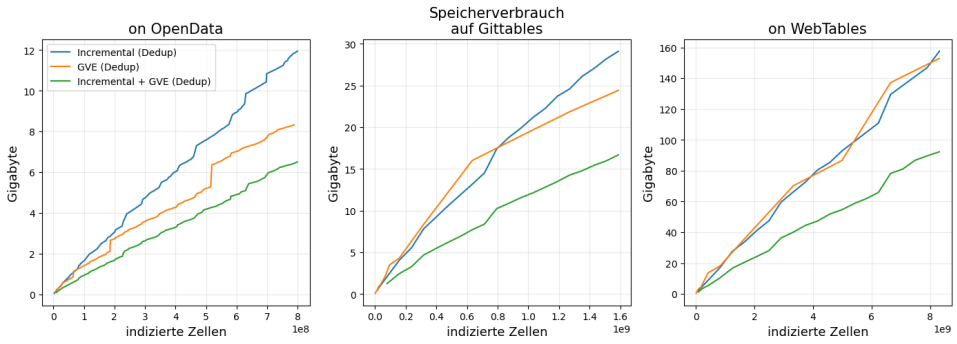


Abbildung 4.13: Vergleich von deduplizierten Verfahren, anhand ihres Speicherverbrauches.

Wie in Abbildung 4.5.1 zu sehen ist, ist das kombinierte Verfahren 'Incremental + GVE (Dedup)' deutlich und konsistent effizienter im Komprimieren der IIDs, als String- oder Integerkompressionsverfahren alleine.

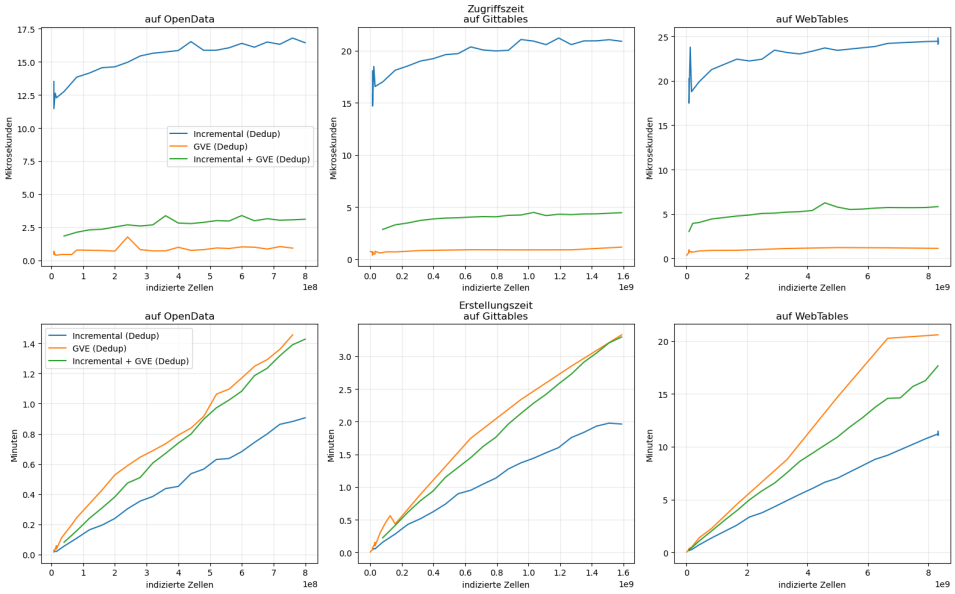


Abbildung 4.14: Vergleich diverser Techniken, anhand von Zugriffs- und Erstellungszeit.

Es stellt sich die Frage, wie effizient das Kombinationsverfahren ist, nachdem eine unoptimierte Version von Incremental-Coding besonders hohe Zugriffszeiten verzeichnet hat.

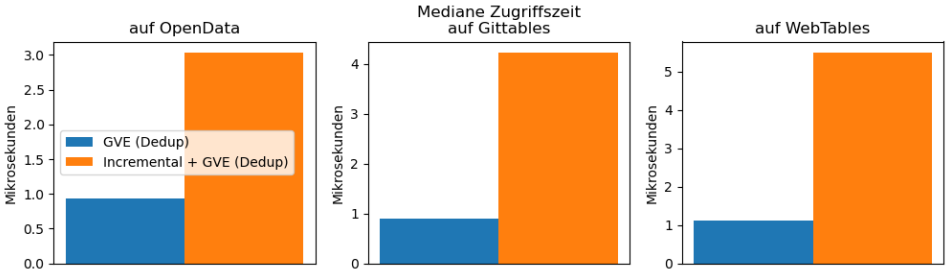
In Abbildung 4.5.1 ist die Incremental-Coding, mit Deduplizierung und GVE kodierten Zellenpositionen zwei Vergleichsgrößen gegenübergestellt: Einmal dem unoptimierten Incremental-Coding Verfahren wie in der Sektion zu String Kompression beschrieben 4.4.3, ohne GVE. Darüber hinaus wurde das alternative Duplikationsverfahren mittels HashMap und GVE betrachtet.

Auffällig ist, dass das Incremental-Coding Verfahren deutlich schneller zu erstellen ist, als das HashMap Verfahren. Das liegt vor allem daran, dass in der Implementierung der HashMap die Information nicht genutzt wird, dass die Eingabedaten vorsortiert sind.

Es ist auch sehr deutlich, dass die Optimierungen am Incremental-Coding die Zugriffszeit massiv gesenkt haben. Aus der Abbildung 4.5.1 lässt sich dadurch die Zugriffszeit zwischen dem HashMap Verfahren 'GVE (Dedup)) und dem Incremental-Coding Verfahren schlecht vergleichen.

Der direkte Vergleich der Zugriffszeit ist in Abbildung 4.5.1 anhand der medianen Zugriffszeit zu sehen.

Es fällt auf, dass der Abstand zwischen den beiden Vergleichsgrößen mit zunehmender Größe der Eingabedaten wächst. Der Grund ist derselbe, aus dem die HashMap so viel schneller ist, als der BTree. Bei der binären Suche



sind Cache-Misses bei großen Datenmengen unvermeidlich und nehmen mit der Menge der Daten logarithmisch zu, da die Anzahl der Vergleiche für die Suche logarithmisch zunimmt.

Bei der HashMap ist die Zugriffszeit eine konstante Größe.

Die dieser Unterschied sich logarithmisch wächst, und der Fokus dieser These auf der Kompressionsrate liegt, wird diese weiter betrachtet.

4.5.2 Sonderfälle für VByte Komprimierung

VByte Kodierung ist in sehr bestimmten Sonderfällen besser als Group-Varint-Encoding geeignet; wenn die zu komprimierende Zahl eine Bitweite von sieben, vierzehn, usw. hat [3].

Eingehende Betrachtung der Eingabedaten zeigt Potenzial dafür, dass VByte manche Felder besser komprimiert, als GVE.

Da die TableID sehr gleichmäßig verteilt ist, wird angenommen, dass diese durch Group-Varint-Encoding besser kodiert wird, wie [3] nahelegt.

Untersucht man die Verteilung der ColumnIDs und RowIDs anhand ihrer Bitweite, findet man Grund zur Annahme, dass VByte Kodierung für diese sehr effizient sein könnte. Abbildung 4.5.2 zeigt diese Verteilung. Zusätzlich wurden die Balken, die Integermengen repräsentieren, farblich kodiert. Die farbliche Kodierung gibt an, wie viele Byte der VByte Algorithmus zum Komprimieren der repräsentierten Integer benötigt.

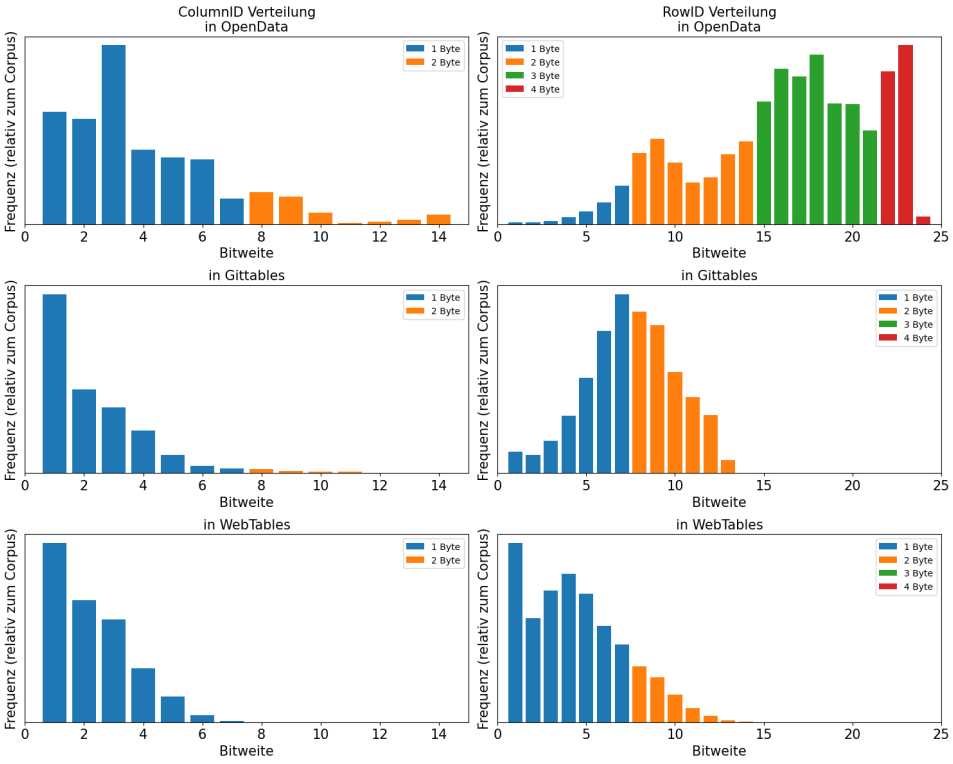


Abbildung 4.15: Der Graph zeigt die Verteilung der Eingabedaten anhand der Bitweite, farblich kodiert danach, wie viele Byte der VByte Algorithmus zum Komprimieren dieser benötigt.

Es zeigt sich bei der ColumnID, dass unabhängig von der Tabelle

angenommen werden kann, dass die große Mehrheit der Zahlen mit nur einem Byte kodiert werden kann. Für ein byte-aligned Verfahren wie VByte oder Group-Varint-Encoding ist das eine optimale Kodierung.

Besonders auf den Corpora GitTables und WebTables scheint VByte annähernd immer ideal.

Ebenso ist die RowID häufig sehr gut geeignet, um von dem VByte Encoding zu profitieren. Auf GitTables und WebTables werden annähernd immer nur ein oder zwei Byte benötigt. Dabei wird ausgerechnet eine Bitweite von vierzehn nicht übertroffen, aber immer noch regelmäßig erreicht. Das ist signifikant, da ab einer Bitweite von fünfzehn bereits ein zusätzlicher Byte zum kodieren benötigt wird, in dem bis zu sechs Bits redundant sind.

Im Corpus OpenData wurden wesentlich längere Tabellen indiziert. Die Codes für die RowID haben bis zu 24 Bit. In Rot eingefärbt sehen wir die größte Menge an Integern repräsentiert, welche drei Byte zur Kodierung benötigt unter Nutzung von VByte Komprimierung. Diese Daten sind so gleichmäßig verteilt, dass die Annahme gerechtfertigt ist, es sei unwahrscheinlich, dass diese mittels der V-Byte Technik besser komprimiert werden, als durch Group-Varint-Encoding.

Implementierung

In den folgenden Experimenten wurde der 'Incremental + GVE (Dedup)' Ansatz modifiziert und mit seiner Ursprungsform verglichen. Die Modifikation ist, dass ColumnID und RowID durch VByte Kodierung komprimiert wurden, während die TableID weiterhin durch GVE kodiert wird. Die Hoffnung ist, dass die gewählte Domäne für VByte Kodierung so gut geeignet ist, dass das Verfahren hier GVE übertrifft und so die Kompressionsrate erhöht wird. Bei der Implementierung wurde besonders auf Struct-alignment, Pointer-overhead und Memory-alignment geachtet, um einen minimalen Speicherverbrauch zu gewährleisten und die Stärken der VByte Technik nicht zu überschatten.

Auswertung

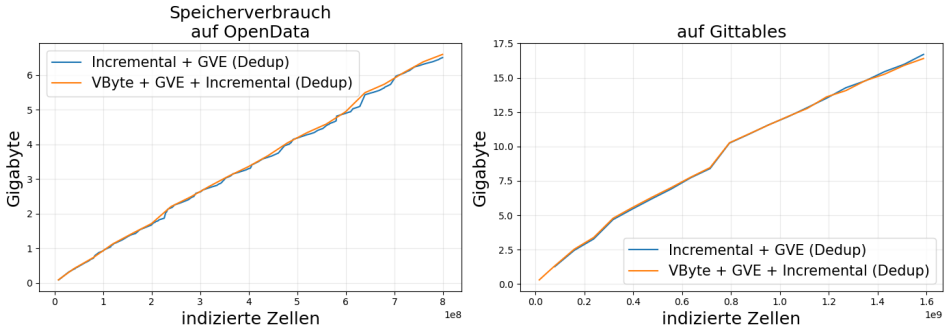


Abbildung 4.16: Der Graph vergleicht die Incremental + GVE Technik, wobei eine Testreihe zusätzlich durch VByte optimiert wird.

Abbildung 4.5.2 zeigt anhand von IIDs auf OpenData und GitTables, dass die zusätzliche Technik nicht zuverlässig die Kompressionsrate erhöht. Tatsächlich verschlechtert sich diese sogar anhand von dem Corpus OpenData leicht. Generell ist der Unterschied minimal. Die Technik wird nicht weiter verfolgt.

Kapitel 5

Verwandte Arbeiten

Verschiedene Arbeiten im Bereich der Invertierten Indizes wurden bereits veröffentlicht.

Sinnvolle Kompressionstechniken sind bereits im Buch 'Introduction to Information Retrieval' [10] beschrieben. Im Rahmen der vorliegenden Arbeit werden jedoch Datenseen von Tabellen als Eingabedaten betrachtet, nicht Dokumente. Wichtiger noch wird nicht nur die Effizienz eines Verfahrens betrachtet, sondern verschiedene Verfahren evaluiert und gegenübergestellt im Verlauf dieser Bachelorarbeit.

Im Paper 'Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).' [3] wurden verschiedene Kompressionsalgorithmen anhand von Experimenten ausgewertet. Eben dies ist auch in dieser Arbeit vollzogen worden. In dem Paper wurde jedoch keinen Wert auf variable Länge der Eingabedaten gelegt, was für die Domäne der Datenseen bedeutend ist. Es wurden stets ausreichend und konstant große Datenmengen gewählt, um die Algorithmen auszutesten. Auch sind hier keine Stringkompressionstechniken betrachtet worden und es wurden keine Zugriffsmethoden berücksichtigt.

Derselbe Unterschied lässt sich zu dem Paper 'Decoding billions of integers per second through vectorization' erkennen. Hier ist die Domäne der Daten so entscheidend, dass in unserem Anwendungsfall andere Algorithmen bessere Ergebnisse geliefert haben.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen der Arbeit wurden einige nützliche Schlüsse über das Komprimieren von Invertierten Indizes von Datenseen gezogen.

6.1 Zusammenfassung

Es hat sich gezeigt, dass Methoden aus dem Information Retrieval Bereich sehr erfolgreich in Datenseen angewendet werden können [10].

Es lassen sich bei Invertierten Indizes viele verschiedene Faktoren durch Kompression angreifen, was Kombinationstechniken besonders effektiv macht.

Integerkompression, die bereits mit kleinen Gruppen von Daten arbeiten kann, ist besonders effektiv. HashMaps haben extrem gute Zugriffszeiten, besonders bei hohen Datenmengen. Für die Kompression von Strings sind Präfixkodierungen sehr effizient.

Die Bachelorarbeit hat das Problem behandelt, mit großen Invertierten Indizes sinnvoll im Arbeitsspeicher arbeiten zu können.

Durch eine Verfahren mit fünf- bis achtfache Kompressionsrate, konnte selbst mit dem größten Datensee auf der gegebenen Hardware noch im Arbeitsspeicher gearbeitet werden. Dabei waren die verwendeten Algorithmen leichtgewichtig genug, um schnelle Zugriffs- und Erstellungszeiten zu gewährleisten.

Als abschließendes Fazit der Arbeit ist auf den Nutzen hinzuweisen, die Kompression auch im Arbeitsspeicher bringt. Selbst einfach zu implementierende Strategien können ausgesprochen effektiv sein, und wieder verwendet werden in unterschiedlichen Domänen.

Gleichzeitig wird bestätigt, dass Kompressionsraten stark von der Domäne der Eingabedaten abhängig sind. Es zeigt sich viel Wert darin, seine Daten gut zu erforschen und methodisch auszutesten.

6.2 Ausblick

Es gibt verschiedene Techniken, die Potenzial für weitere, sinnvolle Forschungen bereithalten.

SIMD Optimierungen haben in verschiedenen Anwendungsfällen zu deutliche Performanceboosts geführt [3] [8]. Techniken wie Group-Varint-Encoding könnten womöglich dadurch profitieren, da die Längenbestimmung der Eingabedaten grundsätzlich ohne Branching vektorisiert werden kann.

In der Arbeit wurden keine parallelen Ansätze untersucht. Moderne Computersysteme besitzen in der Regel mehrere Kerne, weshalb durch Multithreadingtechniken hohe Performancesteigerung zu erwarten ist.

Die HashMap hat sich als außerordentlich effizient beim Indizieren der Daten herausgestellt. Dieser Vorteil konnte beim Nutzen von Präfixkodierungstechniken nicht weiter genutzt werden. Es wäre lohnend, einen Ansatz zu finden, der das Arbeiten mit dieser Technik weiter ermöglicht.

Die Arbeit legt als Grundannahme fest, dass die Eingabedaten sortiert vorliegen. Das hat viele Nachteile. Es wäre sehr nützlich, wenn man die erfolgreichsten Kompressionstechniken auf unkomprimierten Daten anwenden kann. Potenzial hierfür sehe ich darin, dass man zum Beispiel die unsortierten Eingabedaten in eine HashMap einließt, die als Schlüssel die ersten beiden Bytes der Eingabedaten hat. So haben wir 256^2 kleinere Daten als Werte der HashMap, die sich wieder leichter und unabhängig voneinander im Speicher sortieren lassen. Womöglich schon iterativ beim Einlesen oder spekulativ, indem man darauf 'wettet', dass in eine bestimmte Gruppe keine weiteren Eintrage kommen. Gleichzeitig haben wir durch die HashMap potenziell weniger Cache-Misses und durch die isolierten Gruppen von Daten können wir massiv parallele Operationen innerhalb dieser ausführen.

In den Eingabedaten zeigt sich, dass in den meisten Fällen weniger als vier Byte zum Speichern mancher Felder nötig war. Im Zusammenhang mit VByte Encoding wurde bereits versucht, das auszunutzen. Hier bietet sich ein sinnvoller Einsatz für spezielle Techniken aus dem Data-Oriented-Design. Kommen Werte mit einer Bitweite von über 16 nur selten vor, lassen sich die Daten mit zwei Byte kodieren, und die größeren Zahlen werden in einer separaten Map gespeichert. Hier ist besonders auf Speicherausrichtung zu achten. Diese sollte auf ein Byte festgelegt werden, was ausschließlich in einigen hardwarenahen Programmiersprachen möglich ist.

Im Rahmen der Arbeit wurden ausschließlich Lossless-Compression Techniken ausgewertet. Es ist durchaus möglich, dass Lossy-Compression in vielen Anwendungsfällen genauso nützlich ist, jedoch signifikant höhere Kompressionsraten bietet. Es wäre lohnenswert, dieses Potenzial zu erforschen

Literaturverzeichnis

[1] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1134–1145, Los Alamitos, CA, USA, may 2016. IEEE Computer Society.

[2] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 283–296, New York, NY, USA, 2009. Association for Computing Machinery.

[3] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K.-U. Sattler, and S. Breß, editors, *EDBT*, pages 72–83. OpenProceedings.org, 2017.

[4] A. David et al. jemalloc Allocator. `https://github.com/jemalloc/jemalloc`. Version 5.x.

[5] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Top-k entity augmentation using consistent set covering. SSDBM '15, 2015.

[6] M. Hulsebos, Ç. Demiralp, and P. Groth. Gittables: A large-scale corpus of relational tables. *arXiv preprint arXiv:2106.07258*, 2021.

[7] A. Kladov. Measuring Memory Usage in Rust. `https://rust-analyzer.github.io/blog/2020/12/04/measuring-memory-usage-in-rust.html`, Dezember 2020.

[8] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137, 2012.

[9] J. Lopes. Fast smaz. `https://gitlab.com/Kores/fast-smaz`, 2022.

[10] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, England, July 2008.

[11] S. Sanfilippo. smaz. `https://github.com/antirez/smaz`, 2012.

[12] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 847–864, New York, NY, USA, 2019. Association for Computing Machinery.