TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Datenbanken

Masterarbeit

LEICHTGEWICHTIGE KOMPRESSIONSALGORITHMEN UND ANFRAGESCHNITTSTELLEN FÜR FLEXIBLE STORAGE-ARCHITEKTUREN

Paul Peschel
Matr.-Nr: 3661530

Betreut durch:
Prof. Dr.-Ing. Wolfgang Lehner

und:
Dipl.-Ing. Thomas Kissinger
M.Sc. Patrick Damme

Eingereicht am 9. Dezember 2015

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 9. Dezember 2015

ABSTRACT

Mit wachsenden Datenmengen wird die Datenkomprimierung für eine geschwindigkeitsoptimierte Anfrageverarbeitung in Datenbanken notwendig. Somit können mehr Werte in den schnellen CPU-Cache verschoben und analytische Anfragen auf den Daten beschleunigt werden. Um den Einfluss der Komprimieralgorithmen auf die Architektur des Datenbanksystems gering zu halten, werden allgemeine Anfrageschnittstellen für die komprimierten Daten benötigt. Das erlaubt die direkte Anwendung von Anfrageoperatoren auf den komprimierten Daten durch eine allgemeine Schnittstelle, wodurch sich eine schnellere Datenverarbeitung erreichen lässt. In dieser Arbeit werden allgemeine Anfrageschnittstellen auf komprimierten Daten und die Umsetzung der leichtgewichtigen Kompressionsalgorithmen Run-Length-Encoding, Dictionary-Encoding und dem Null-Suppression Verfahren SIMD-FastPFOR in dem Datenbanksystem ERIS für flexible Storage-Architekturen gezeigt.

INHALTSVERZEICHNIS

1 Einleitung **9**

2 Verwandte Arbeiten **11**

2.1 ERIS 11

2.2 Leichtgewichtige Techniken für Kompression und Transformation 12

2.3 Dekodierung von milliarden Zahlenwerten pro Sekunde durch Vektorisierung . . 13

2.4 Integration von Kompression und entsprechender Anfrageausführung in spaltenorientierten Datenbanksystemen 14

2.5 Einfache Lösungen für die Anfrageausführung auf komprimierten Daten in vektorisierten Datenbanksystemen 15

2.6 Kompression und Anfrageausführung in spaltenorientierten Datenbanken 17

3 Grundlagen **19**

3.1 ERIS 19

3.2 Column-Store 20

3.3 Leichtgewichtige Kompressionsalgorithmen 22

4 Column-Store Kompression **25**

4.1 Kompressionsinterface 25

4.2 Page-Packing 27

4.3 Kompressionsverkettung 30
4.4 Auto-Kompression 31
4.4.1 FastPFOR Heuristik 31
4.4.2 Run-Length-Encoding Heuristik 32
4.4.3 Dictionary-Encoding Heuristik 32

5 Kompressionsoperatoren **33**
5.1 Interface 33
5.1.1 CompressionOperators-Interface 34
5.1.2 CompressedOperation-Interface 35
5.2 Null-Suppression Operatoren 37
5.3 Run-Length-Encoding Operatoren 38
5.4 Dictionary-Encoding Operatoren 39
5.5 Operatortypen 40
5.5.1 Aggregationsoperatoren 40
5.5.2 Junktoren 41

6 Evaluation **43**
6.1 Kompressionsraten 44
6.1.1 Kompressionskette 44
6.1.2 Auto-Kompression 50
6.2 Kompressionsperformance 51
6.3 Page-Packing Performance 53
6.4 Kompressionsoperatorperformance 55
6.4.1 Ausführungszeiten Kompressionsoperatoren 55
6.4.2 Ausführungszeiten Aggregationsoperatoren 57

7 Fazit **59**

8 Zukünftige Arbeiten **61**

1 EINLEITUNG

Mit den wachsenden Datenmengen, die in Datenbanken verarbeitet werden müssen, steigen zugleich auch die Leistungsansprüche an heutige Datenbanken. Um weiterhin eine schnelle Verarbeitung der Daten zu gewährleisten, werden in Computersystemen die Anzahl an CPU-Kernen sowie die Größe des Arbeitsspeichers stetig erhöht. Zum Beispiel hat der aktuell beste Computer der Welt[1] mit dem Namen *Tianhe-2*[2] über 3 Millionen CPU-Kerne und über 1 Million GB Arbeitsspeicher.

Eine Hauptaufgabe von modernen Datenbanksystemen besteht darin, die Daten an die CPU-Kerne für eine optimale Verarbeitung zu verteilen. Die Speicheranbindung der CPUs bildet dabei einen Flaschenhals, der die Datenverarbeitung verzögert. Daher haben CPU-Kerne mit dem *Cache* einen eigenen Speicher. Dieser ist zwar wesentlich schneller als der Arbeitsspeicher, hat jedoch eine deutlich geringere Speicherkapazität.

Ein Anwendungsgebiet moderner Datenbanksysteme ist die Analyse der enthaltenen Daten. Da die bisher übliche zeilenorientierte Speicherarchitektur alle Attributwerte eines Eintrags in einem Tupel abspeichert, erfordert das Auslesen der Werte nur eines Attributes das Laden kompletter Tupel. Daher etablierten sich spaltenorientierte und hybride Speichermodelle. Diese erlauben das schnelle Auslesen aller Werte eines Attributs, mit denen Analysen effektiv ausgeführt werden können. Hybride Speichermodelle kombinieren die spalten- und zeilenorientierte Datenspeicherung sowie deren Vorteile.

Ein weiterer Vorteil spaltenorientierter Speicherarchitekturen ist die für gewöhnlich geringe Varianz und ein hoher redundanter Informationsgehalt der Werte eines Attributes. Diese Eigenschaft ermöglicht die Kompression der Daten mit leichtgewichtigen Kompressionsalgorithmen. Die dadurch erreichte Reduzierung der Speichergröße erlaubt es, eine größere Anzahl an Werten über den Speicher-Bus in die Caches der CPU-Kerne zu laden. Dort können die Daten mit Hilfe von Kompressionsoperatoren direkt effizient verarbeitet werden. Diese vermeiden eine aufwendige Dekompression der Daten zur Verarbeitung mit Standard-Operatoren. Weiterhin ermöglicht die Kompression, größere Datenmengen im System zu speichern.

[1] http://www.top500.org/
[2] http://www.top500.org/system/177999

In dieser Arbeit wurden leichtgewichtige Kompressionsalgorithmen und Kompressionsoperatoren in das Datenbanksystem ERIS integriert. Zusätzlich wurden Möglichkeiten zur Verbesserung der Kompression und zur adaptiven Auswahl von Kompressionsalgorithmen entwickelt.

In Kapitel 2 werden verwandte Arbeiten zur Thematik der leichtgewichtigen Kompression und der Anwendung von Operatoren auf komprimierten Daten vorgestellt. Kapitel 3 beschreibt die Grundlagen zur spaltenorientierten Speicherung von Daten (*Column-Store*) sowie die in dieser Arbeit verwendeten Kompressionsalgorithmen FastPFOR, Run-Length-Encoding und Dictionary-Encoding. Kapitel 4 beschreibt die Umsetzung der Kompressionsalgorithmen in ERIS. Der vorgestellte Page-Packing-Algorithmus in Abschnitt 4.2 verbessert durch Verschiebung komprimierter Daten die Belegung der zur Verfügung gestellten Speicherbereiche. Außerdem wurde die Möglichkeit der Verkettung (siehe Abschnitt 4.3) und adaptiven Auswahl (siehe Abschnitt 4.4) von Kompressionsalgorithmen beschrieben. Die Umsetzung der Kompressionsoperatoren wird in Abschnitt 5 gezeigt. Die Evaluation der Kompressionsalgorithmen und der Kompressionsoperatoren wird in Kapitel 6 vorgestellt. Das Fazit und Empfehlungen für aufbauende Arbeiten befinden sich in Kapitel 7 und 8.

2 VERWANDTE ARBEITEN

Die folgenden verwandten Arbeiten beschreiben das zugrunde liegende Datenbanksystem ERIS in Abschnitt 2.1 sowie die implementierten Kompressionsalgorithmen in Abschnitt 2.2 und 2.3. Die Sektionen 2.4, 2.5 und 2.6 beschreiben die Ausführung von Datenbankoperatoren auf komprimierten Daten.

2.1 ERIS

ERIS [8] ist eine all-in-memory Datenbank. Das Datenbanksystem wurde für *non-uniform memory access* (NUMA) Speicherarchitekturen entwickelt. In NUMA-Systemen hat jeder Prozessor seinen eigenen Arbeitsspeicher. Über ein Kommunikationsnetzwerk können Prozessoren auf den lokalen Arbeitsspeicher anderer Prozessoren zugreifen. Da Zugriffe auf entfernten Speicher durch hohe Latenzzeiten und geringere Bandbreiten sehr kostenintensiv sind, müssen Datenbanksysteme die Arbeitslast über die Prozessoren verteilen. Somit sollen die Kosten für die Datenverarbeitung so gering wie möglich gehalten werden.

ERIS führt auf jedem Prozessorkern einen Worker-Thread aus, der eine Autonomous Execution Unit (AEU) bildet. Um jede AEU effizient auszunutzen, verwendet ERIS eine Datenpartitionierung. Die Datenpartitionen werden gleichmäßig den AEUs zur Verarbeitung zugeordnet. Die Partitionierung der Datenobjekte kann über ihre physische Größe als auch über die durchschnittliche Ausführungszeit und Anzahl der Datenzugriffe der Anfragen erfolgen. Die Datenobjekte und Anfragen werden von ERIS überwacht. Die daraus erstellten Metriken werden einem konfigurierbaren Load-Balancing-Algorithmus übergeben. Erkennt der Load-Balancing-Algorithmus eine ungleiche Auslastung der AEUs, werden die Datenobjekte neu partitioniert und an die AEUs verteilt.

In dieser Arbeit wird ERIS als Basissystem verwendet. Da die Datenobjekte sehr groß werden können, sollen leichtgewichtige Komprimierungsalgorithmen verwendet werden, um die Transferzeit der Datenobjekte über den Speicher-Bus zu verkürzen und mehr Daten in den schnellen CPU-Cache zu laden. Eine kürzere Ausführungszeit lässt sich durch direktes Anwenden von Anfrageoperatoren auf den komprimierten Daten erreichen.

2.2 LEICHTGEWICHTIGE TECHNIKEN FÜR KOMPRESSION UND TRANSFORMATION

Die Masterarbeit von Damme [5] beschreibt die effiziente Implementierung leichtgewichtiger Kompressions- und Transformationsalgorithmen. Da Komprimierungsalgorithmen zusätzlichen Rechenaufwand benötigen, müssen diese möglichst effizient implementiert werden. Dies zeigt Damme mit Hilfe von Single Instruction Multiple Data (SIMD) Operationen für die Kompressionsalgorithmen Run-Length-Encoding, Dictionary-Encoding und Null-Suppression. SIMD Operationen verarbeiten Daten parallel mit nur einer Instruktion.

Das von Damme verwendete Run-Length-Encoding-Schema (siehe Abschnitt 3.3) speichert, im Gegensatz zu dem RLE-Schema von Abadi (siehe Abschnitt 2.4), nicht die Startposition des Runs. Demnach ist die Startposition aus der Position des kodierten Paares oder auch aus den akkumulierten Längen der vorherigen Paare zu ermitteln. Das verwendete Dictionary-Encoding basiert auf einem *Order-Preserving-Dictionary* (siehe Abschnitt 3.3). Hierbei ist das Dictionary den Werten nach sortiert und enthält diese im unkomprimierten Zustand. In Abadis Arbeit muss der Index eines Dictionary-Wertes erst durch Shifting und einer AND-Verknüpfung ermittelt werden. Da das Order-Preserving-Dictionary als Eintrag einen Dictionary-Index speichert, kann auf den jeweiligen Dictionary-Wert direkt zugegriffen werden. Dies vereinfacht die Verarbeitung mit Kompressionsoperatoren. Jedoch wird eine geringere Kompressionsrate erreicht.

Je nach Anforderung an Kompressions-/Dekompressionszeiten sowie Anwendung von Datenbankoperatoren auf komprimierten Daten müssen andere Kompressionsformate verwendet werden. Da komprimierte Daten nur zu genau einem Kompressionsformat zugeordnet werden können, bedarf es der Möglichkeit der Transformation der Daten in ein anderes Kompressionsformat. Eine triviale Variante ist die Dekompression und Rekompression in das entsprechende Format. Eine wesentlich effizientere Variante zeigt Damme in seiner Masterarbeit mit Transformationsalgorithmen, die komprimierte Daten direkt in ein anderes Komprimierungsschema überführen können.

Die Evaluation wurde mit synthetischen Daten, für die Dateneigenschaften wie Run-Length, Anzahl unterschiedlicher Werte und Anzahl effektiver Bits festgelegt werden können, durchgeführt. Damme zeigt, dass Run-Length-Encoding mit steigender Run-Length eine bessere Kompressionsrate aufweist. Jedoch nähert sich die Kompressionsgeschwindigkeit einer oberen Grenze an, je länger die Run-Lengths werden. Ursache hierfür ist die begrenzte Bandbreite des Speicher-Busses. Während bei steigender Run-Length die zu schreibenden Daten reduziert werden, verringert das Auslesen der Daten aus dem Speicher die Geschwindigkeit. Analog steigt auch die Dekompressionsgeschwindigkeit mit zunehmender Run-Length an und nähert sich einer oberen Grenze. In diesem Fall belasten mit wachsender Run-Length die zu schreibenden Daten den Speicher-Bus.

Das Schema Order-Preserving-Dictionary zeigte bei steigender Anzahl an unterschiedlichen Werten eine Verringerung der Kompressions- und Dekompressionsgeschwindigkeit an. Eine ansteigende Anzahl verschiedener Werte führte bei der Kompression zu einer Vergrößerung der Buckets der Hash-Map und damit zu einer Reduzierung der Zugriffszeit solange kein *rehash* ausgeführt wurde. Die Geschwindigkeit der Dekompression nimmt ab, sobald das Dictionary die Größe des CPU-Caches überschreitet und somit aus einem langsameren Speicher geladen wer-

den muss.

2.3 DEKODIERUNG VON MILLIARDEN ZAHLENWERTEN PRO SEKUNDE DURCH VEKTORISIERUNG

Da der Bus zwischen Arbeitsspeicher und CPU einen Bottleneck darstellt, müssen Daten in komprimierter Form in den wesentlich schnelleren CPU-Cache geladen werden. In der CPU lassen sich SIMD-Instruktionen verwenden, um Daten parallel mit einer Instruktion zu verarbeiten. Lemire beschreibt in seiner Arbeit [10] innovative Kompressionsschemas wie SIMD-BP128, die auf dieser Basis nahezu doppelt so schnell sind wie die Schemas varint-G8IU und PFOR. Das Kompressionsschema SIMD-FastPFOR dekomprimiert Daten bis zu 30% schneller als PFOR und erreicht dabei gleichzeitig eine bis zu 10% höhere Kompressionsrate.

PFOR [18] basiert dabei auf dem Kompressionschema *Binary-Packing* [4], welches der *Frame-Of-Reference (FOR)* [6] Kompression ähnlich ist. In PFOR werden die zu komprimierenden Daten in Blöcken mit einer festen Anzahl an Werten aufgeteilt. Danach wird der minimale und maximale Zahlenwert der in dem Block vorkommt kodiert. Die Differenz eines Wertes zum Minimalwert wird daraufhin mit einer festen Bitlänge und dem Minimalwert als Referenz kodiert. Lemire beschreibt in seiner Arbeit ein Beispiel, in dem ein Block mit dem Wertebereich [1000;1127] gegeben sei. Dann lässt sich jede Zahl mit der Bitlänge 7 als Abstand vom Minimalwert 1000 kodieren ($\lceil \log_2(1127 + 1 - 1000) \rceil = 7$). Eine weitere Form des Binary-Packing erlaubt auch variable Blocklängen. Hierfür muss die Blocklänge B pro Block zusätzlich zur Bitlänge b gespeichert werden. Die Speicherkosten berechnen sich danach mit der Formel $bB + k$, wobei k ein fester Wert ist, der einen overhead pro Block darstellt. Mit einer Minimierung der Kosten über die Variation der Blocklänge werden bessere Kompressionsraten erreicht.

Effiziente Kompressionsraten lassen sich mit Binary-Packing nur erreichen, wenn die zu kodierenden Zahlen einen möglichst minimalen Abstand zum Referenzwert aufweisen. Sind zum Beispiel folgende Zahlen gegeben: 1, 4, 12, 4294967295 werden alle Zahlen in diesem Block mit einer Bitlänge von 32 kodiert, da $4294967295_{10} = FFFFFFFF_{16}$ bereits 32 Bit zur Darstellung benötigt. Zukowski et al. [18] beseitigte diesen Nachteil mit *patching*. Hierbei wird aus Datensamples durch Minimierung einer Kostenfunktion eine Bitlänge b ermittelt, die eine optimale Kompressionsrate ergibt. Alle Werte mit einer Bitlänge größer gleich b werden auf b Bits gekürzt. Die Werte werden zuvor als *Exceptions* in eine Exception-Tabelle geschrieben, die sich an einer separaten Position im Block befindet. Jeder Block erhält am Blockanfang einen Marker mit der Position der ersten Exception im Block und der Position der dazugehörigen Exception in der Exception-Tabelle. Die Werte im Block werden mit Binary-Packing kodiert. Der offset berechnet sich dabei aus der Differenz des Index des aktuellen Exception-Wertes und dem des nächsten Exception-Wertes in der Exception-Tabelle minus eins.

Im Kompressionsschema Fast-PFOR wird b pro Block ermittelt und in einer Page zusammen mit der Exception-Tabelle, der maximalen Bitlänge und bei Bedarf einem Exception-Zähler c sowie den Positionen der Exceptions innerhalb des Blocks als offsets gespeichert. Eine Page kann aus mehreren Blöcken bestehen. Die Kostenfunktion, über die b ermittelt wird, betrachtet gleichzeitig auch die Größe der Exception-Tabelle. In den Blöcken werden nur die b kleinsten Bits je Zahl gespeichert. Ist die maximale Bitlänge größer als b wird zusätzlich eine Variable c gespeichert, die

die Anzahl der Exceptions angibt. Alle Blöcke werden nacheinander kodiert. Zum Schluss werden die gesammelten Metadaten und deren Größe nach den kodierten Blöcken gespeichert. Für einen schnellen Zugriff auf die Metadaten wird deren Position am Anfang der Page gespeichert.

Da die Anzahl der Werte, die in einem Block gespeichert werden, stets durch 32 teilbar ist, kann jeder Block an einer 32-Bit ausgerichteten Adresse gespeichert werden. Dies erlaubt die schnelle Kodierung/Dekodierung der Blöcke mit SIMD-Operationen.

2.4 INTEGRATION VON KOMPRESSION UND ENTSPRECHENDER ANFRAGEAUSFÜHRUNG IN SPALTENORIENTIERTEN DATENBANKSYSTEMEN

C-Store [2, 15] ist ein spaltenorientiertes Datenbanksystem. Abadi et al. haben in ihrer Arbeit [3] Kompressionsalgorithmen als Sub-System in C-Store eingeführt und die direkte Verwendung von Datenbankoperatoren auf komprimierten Daten untersucht. Die Ergebnisse aus der Evaluation der Arbeit flossen in einen Entscheidungsgraphen ein, der Datenbankentwicklern zur Findung einer geeigneten Kompressionsmethode helfen soll (siehe Abb. 1 im Anhang). Umgesetzt wurden die Kompressionschemas Null-Suppression, Dictionary-Encoding, Run-Length-Encoding und Bit-Vector-Encoding sowie das schwergewichtige Kompressionsverfahren Lempel-Ziv-Encoding (*LZ-Encoding*).

Da eine performante Anfrageverarbeitung im Vordergrund der Arbeit stand, wurde auf ein Order-Preserving-Dictionary-Encoding Schema verzichtet, da es variable Längen der Dictionary Einträge zulässt. Ein Schema mit festgelegten Eintragslängen erlaubt die Dekodierung des Index eines Eintrags mit nur einer AND-Operation und einer Shift-Operation. Zudem kann die Größe des Dictionarys im Vorfeld berechnet werden und dieses für einen schnellen Zugriff in den CPU-Cache verschoben werden. Feste Längen ermöglichen außerdem die Dekodierung mehrerer Werte gleichzeitig. Abadi zeigte das in folgendem Beispiel: Zunächst wird die Anzahl verschiedener Werte ermittelt und daraus die Bitlänge berechnet. Danach wird aus diesem Wert ein Index mit 1, 2, 3 oder 4 Bytes erstellt. Hat eine Column zum Beispiel 32 unterschiedliche Werte, werden 5 Bit zur Darstellung benötigt. Das heißt, ein Wert lässt sich mit einem Byte darstellen, drei Werte mit zwei Bytes, vier Werte mit drei Bytes und sechs Werte mit vier Bytes. Jetzt lässt sich ein Mapping aus den möglichen 5 Bit Werten und den originalen Werten erstellen. Wurde zum Beispiel die 3-Werte/2-Byte Variante verwendet, können die 5 Bits 00000_2 auf 1 mappen, 00001_2 auf 25 und 00010_2 auf 31. Dann dekodiert der Bitstring $X000000000100010_2$ (von rechts nach links gelesen) zu 31 25 1. Somit lassen sich auch direkt über Bitmasken und shift-Operationen Indizes auslesen, mit denen auf einzelne Werte aus dem Dictionary zugegriffen werden kann.

Das verwendete Run-Length-Encoding Schema ersetzt jeden Run durch ein Triple der Form *(value, start position, run length)*. Jedes Element dieses Triples hat eine feste Bitlänge. Bit-Vector Encoding eignet sich für Attribute mit wenigen möglichen Werten. Für jeden möglichen Wert wird ein Bitstring angelegt, der eine 1_2 an der Stelle enthält, an der der jeweilige Wert in der Spalte vorkommt, ansonsten eine 0_2.

Abadi et al. haben in ihrer Arbeit gezeigt, wie man auf komprimierten Daten arbeiten kann.

Dafür hat er zu jedem Kompressionsschema zwei Klassen hinzugefügt: Die erste Klasse repräsentiert die komprimierten Daten und wird *compression block* genannt. Diese Klasse enthält einen Puffer für komprimierte Daten und bietet verschiedene Funktionen, um Zugriff auf den Puffer zu erlangen (siehe Tabelle 2.1).

Eigenschaften	**Iteratoren**	**Blockinformationen**
isOneValue()	getNext()	getSize()
isValueSorted()	asArray()	getStartValue()
isPosContig()		getEndPosition()

Tabelle 2.1: API der *compression block*-Klasse

Die Funktion $isOneValue()$ gibt an, ob die Daten im Puffer nur aus einem Wert bestehen. Während $isValueSorted()$ angibt, ob die Daten im Puffer sortiert sind, zeigt $isPosContig()$ an, ob die Daten ihren Positionen folgend nacheinander angereiht sind. Bei Bit-Vector-Encoding ist dies zum Beispiel nicht der Fall, da ein Block eine Untermenge eines Bitvektors eines Wertes enthält. Die Iteratorfunktion $getNext()$ liefert den nächsten Wert des Puffers in dekomprimierter Form und seine Position zurück. Die Funktion $asArray()$ dekomprimiert den kompletten Puffer und liefert diesen als Array zurück. Die Position des letzten Wertes in dem Block wird durch $getEndPosition()$ zurückgegeben. Die Anzahl an Elementen gibt die Funktion $getSize()$ zurück. Den ersten dekomprimierten Wert liefert die Funktion $getStartValue()$. Die Blockinformationen lassen sich ohne Dekomprimierung ermitteln. Das Anwenden von Datenbankoperatoren kann über die Iteratoren direkt auf den komprimierten Daten geschehen. Über die zusätzlichen Funktionen lassen sich je nach verwendetem Kompressionsschema Datenbankoperatoren effizienter anwenden. Zum Beispiel liefert bei verwendeter RLE-Kompression die Funktion $getStartValue()$ den Wert und $getSize()$ die Runlength des Wertes. Ein SUM-Aggregationsoperator kann damit durch einfache Multiplikation beider Werte das Ergebnis liefern. Somit muss jedoch jeder Operator für jedes Kompressionsschema separat implementiert werden, was zu einer höheren Code-Komplexität führt.

Aus den Experimenten leiteten Abadi et al. einen Entscheidungsbaum ab (siehe Abb. 1 im Anhang), der es Datenbankentwicklern ermöglichen soll, ein geeignetes Kompressionsschema zu wählen. Grundlage bilden Metriken über die zu speichernden Daten. Da die Entscheidungen durch den Vergleich mit statischen Werten geschieht, ist der Entscheidungsbaum nur bedingt geeignet.

2.5 EINFACHE LÖSUNGEN FÜR DIE ANFRAGEAUSFÜHRUNG AUF KOMPRIMIERTEN DATEN IN VEKTORISIERTEN DATENBANKSYSTEMEN

Bislang mussten für die Implementierung von Kompressionsoperatoren intrusive Eingriffe in das Datenbanksystem vorgenommen werden. Insbesondere waren Änderungen im QueryExecutor notwendig. Łuszczak hat in ihrer Arbeit [11] Methoden vorgestellt, die die Umsetzung der Kompressionsoperatoren in dem Datenbanksystem VectorWise [1] vereinfachen und vorhandene Datenbankkomponenten so gering wie möglich beeinflussen sollen. Unter Berücksichtigung zusätzlicher Komplexität wurde auf Funktionen ohne signifikante Verbesserung der Performance

verzichtet. Beschrieben wurden eine Vereinfachung von Kompressionsoperatoren basierend auf Run-Length-Encoding, sowie die Vorteile von Dictionary-Encoding mit globalen Wörterbüchern und *on-the-fly*-Wörterbüchern.

In VectorWise werden die Daten nicht tupelweise verarbeitet, sondern in vertikale Vektoren unterteilt. Datenbankoperatoren werden auf alle Elemente eines Vektors angewendet. Operatoren, die direkt auf den Daten in den Vektoren arbeiten können, werden als *Primitives* bezeichnet. Primitives stellen einfache Operatoren wie zum Beispiel arithmetische Operatoren dar, die der Compiler für super-scalar CPU Funktionen wie SIMD-Operationen leicht optimieren kann. Die Funktionssignatur eines Primitives enthält neben einer eindeutigen Bezeichnung des Primitives auch alle für die Operation notwendigen Parameter. Aus der Sicht heutiger objekt-orientierter Programmiersprachen mit der Möglichkeit der Funktionsüberladung widersprechen solche Signaturen der Empfehlung, die Anzahl von Funktionsparametern möglichst gering zu halten, um der Komplexität vorzubeugen. Einen Eindruck dafür soll die Funktionssignatur in Listing 2.1 geben.

```
1 int map_add_sint_col_sint_val(int n, sint* result, sint* param1, sint* param2)
```

Listing 2.1: Funktionssignatur Primitive VectorWise

Primitives werden in *Expressions* für komplexere Gleichungen verwendet. Expressions werden intern in einer Baumstruktur als Ausführungsgraph dargestellt. Das Ergebnis einer Expression wird durch rekursives Ausführen der Kindexpressions ermittelt und in einem Ergebnisvektor gespeichert.

Kompressionschemas werden in VectorWise automatisch ausgewählt. Es besteht aber auch die Möglichkeit, ein Kompressionsschema vorzugeben. In dieser Arbeit wurden diese Möglichkeiten ebenfalls implementiert (siehe Abschnitt 4.4).

Łuszczak zeigt weiterhin den nötigen Implementierungsaufwand pro Primitive, um auf RLE-komprimierten Daten arbeiten zu können. Für jeden Operator werden vier Funktionsversionen benötigt: eine Funktion für zwei unkomprimierte Vektoren, eine Funktion für zwei RLE-komprimierte Vektoren und zwei Funktionen für einen unkomprimierten und einen RLE-komprimierten Vektor. Soll die Möglichkeit bestehen unkomprimierte oder RLE-komprimierte Ergebnisvektoren zurück zu liefern, verdoppelt sich die Anzahl der Funktionen. Operatoren, die stets ein valides Ergebnis liefern, wie zum Beispiel Vergleichsoperatoren, müssen auf jedem Run ausgeführt werden. Łuszczak nennt dies *full computation*.

Die Ausführung von Expressions untergliedert sich in zwei Phasen: Die *Build*-Phase und die *Execution*-Phase. Die Build-Phase erstellt den Operatorenbaum, indem die jeweiligen Primitive-Funktionen festgelegt werden. Die Execution-Phase wertet die Argumentvektoren aus und entscheidet, welche Primitive-Funktionsvariante verwendet werden soll. Dies geschieht durch einfache String-Substitution in der Funktionssignatur. Da Primitive-Funktionen Vektoren oder Literale als Argument entgegennehmen, ersetzt die Executor-Phase in der Funktionssignatur *col* durch *val*, wenn ein Literal gegeben ist. Vektoren die nur aus einem Wert bestehen, werden als *constant vector* bezeichnet und stellen ein Literal dar. Der Austausch der Primitive-Funktionen wird als *primitive swapping* bezeichnet.

Aggregationsoperatoren bestehen aus einer *Build*-Phase und einer *Produce*-Phase. In der ersten

Phase werden die Aggregationsoperatoren auf die Daten angewendet. Die zweite Phase liefert die Ergebnisse der ersten Phase zurück.

2.6 KOMPRESSION UND ANFRAGEAUSFÜHRUNG IN SPALTENORIENTIERTEN DATENBANKEN

Basierend auf C-Store hat Ferreira in seiner Arbeit [7] einen Queryexecutor vorgestellt, der auf komprimierten Daten arbeiten kann. Da für jedes weitere Kompressionschema die Operatoren angepasst werden müssen, wird die Komplexität der Operatoren erhöht. Durch Abstrahierung der Kompressionschemas und Einführung eines allgemeinen Interfaces über das der Queryexecutor Zugriff auf die Kompressionsalgorithmen erlangt, vereinfacht sich die Implementierung der Operatoren. Dabei werden die Kompressionsschemas nach ihren Eigenschaften klassifiziert und diese Informationen den Operatoren für die Optimierung der Verarbeitung übergeben.

Ähnlich wie bei Abadi et al. [3], wird auf die in Blöcken komprimierten Daten über einen Iterator zugegriffen. Die zusätzlichen Informationen zu den Eigenschaften der Daten, zum Beispiel ob ein Block nur einen Wert beinhaltet, kann der Operator für die Verarbeitung nutzen.

In C-Store wurden drei Arten von Kompressionsschemas implementiert. *Speicheroptimierungen* beziehen sich auf die optimale Speicherung eines Wertes. Ein Kompressionschema dieser Art ist Null- Suppression. Es kodiert Werte entsprechend der für sie maximal notwendigen Bytes zur Darstellung. Wird zum Beispiel der Wert 123 mit vier Bytes gespeichert, ersetzt Null-Suppression diese durch ein Byte. *Leichtgewichtige Kompressionsschemas* beziehen sich auf eine Sequenz von Werten. Arten dieses Schemas sind Run-Length-Encoding oder Delta-Encoding. *Schwergewichtige Kompressionsschemas* beziehen sich auf ein Array von Bytes. Dabei werden gefundene Bytemuster in ein Dictionary geschrieben und jedes weitere Auftreten dieses Musters durch den Index im Dictionary ersetzt. Ein schwergewichtiges Kompressionsschema ist zum Beispiel Lempel-Ziv-Encoding [17] (kurz: LZ) bzw. Lempel-Ziv-Oberhumer-Encoding [13] (kurz: LZO, erreicht bei der Dekodierung eine höhere Geschwindigkeit als LZ).

Basierend auf den verwendeten Kompressionsschemas, hat Ferreira Kriterien entwickelt (siehe Tabelle 2.2), die die Auswahl eines Schemas entscheiden sollen.

Dateneigenschaften	Wenige Werte	Viele Werte
Folgen wiederholender Werte	RLE	RLE (Folgen gleicher Werte müssen existieren)
Sortiert	RLE (Folgen gleicher Werte müssen existieren)	DeltaOnValue, LZO
Unsortiert	Dictionary, DeltaOnPosition	LZO
Große Werte	Dictionary	LZO
Kleine Werte	Dictionary, Null-Suppression	Null-Suppression

Tabelle 2.2: Auswahlkriterien für Kompressionsalgorithmen nach Ferreira basierend auf der zu kodierenden Sequenz von Werten. DeltaOnPosition kodiert Werte auf Basis ihrer Position (Wert & Startposition, sowie Deltawerte zu weiteren Positionen dieses Wertes werden gespeichert). DeltaOnValue kodiert Werte auf Basis ihrer Differenz zueinander (Startwert und Differenzen zu folgenden Werten werden gespeichert)

3 GRUNDLAGEN

In diesem Kapitel werden die für diese Arbeit notwendigen Grundlagen beschrieben. Da die spaltenorientierte Speicherarchitektur prädestiniert für leichtgewichtige Kompressionsalgorithmen ist, wurde ein Column-Store für ERIS entwickelt. Der Aufbau der komprimierten Pages sowie des Column-Stores dienen als Grundlage für das Verständnis der Funktionsweise der Kompressionsoperatoren.

Das verwendete Datenbanksystem ERIS wird in Abschnitt 3.1 erklärt. Der Aufbau des für ERIS entwickelten Column-Stores wird in Abschnitt 3.2 dargestellt. Die Funktionsweise der Kompressionsalgorithmen und die Architektur der resultierenden Daten beschreibt der Abschnitt 3.3. Es wurden die leichtgewichtigen Kompressionsalgorithmen SIMD-FastPFOR, Run-Length-Encoding und Dictionary-Encoding in ERIS verwendet.

3.1 ERIS

Das Datenbanksystem ERIS hält eingefügte Daten komplett im Arbeitsspeicher. ERIS erstellt für jeden verfügbaren CPU-Kern eine *Autonomous Execution Unit* (AEU). Werden Daten in ERIS eingefügt, werden diese über einen konfigurierbaren *Load-Balancer* verteilt. In der Abbildung 3.1 ist dies für vier AEUs schematisch dargestellt. Der Load-Balancer verteilt die Daten standardmäßig im Round-Robin-Verfahren an die AEUs. Dadurch ergibt sich eine Datenpartitionierung. Die Datenpartitionen werden parallel auf jeder AEU verarbeitet.

Daten werden in ERIS in *Storage-Containern* gespeichert. Die Container sind über *Hints* konfigurierbar. Standardmäßig werden Werte in den Containern zeilenorientiert gespeichert. Für diese Arbeit wurde zusätzlich eine spaltenorientierte Speicherung ermöglicht (siehe Abschnitt 3.2). Diese erlaubt eine effektive Kompression der Daten. Innerhalb der spaltenorientierten Speicherarchitektur wurde auch eine hybride Variante implementiert. Diese ermöglicht es, Werte verschiedener Attribute in einer Spalte abzulegen. Die Daten in den Containern werden in Pages abgespeichert.

Der Kompressionsalgorithmus kann in ERIS manuell ausgewählt werden. Es wurde zusätzlich

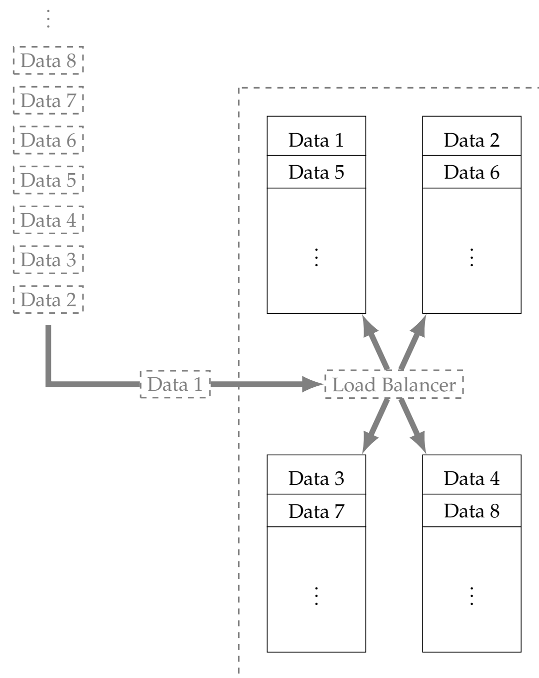


Abbildung 3.1: Einfügen von Daten in ERIS mit 4 AEUs über einen Round-Robin Load-Balancer

eine adaptive Kompressionsauswahl implementiert (siehe Abschnitt 4.4). Diese wählt anhand der eingefügten Daten einen geeigneten Kompressionsalgorithmus aus. Die gewählten Kompressionsalgorithmen können für jede Page variieren.

3.2 COLUMN-STORE

Der Column-Store wird mit dem *Columnar-Hint* aktiviert. Alle eingefügten Daten, werden spalteorientiert in *Pages* gespeichert. Eine Page des Column-Stores besteht aus einem *ColumnStoreHeader* und einem *ColumnStoreBody* (siehe Abb. 3.2). Die Pages in einem Container sind über den *next*-Pointer einfach verlinkt. Das Attribut *bmpSize* gibt die Größe einer Bitmap in Bytes an. Die Größe einer Bitmap wird anhand der maximalen Gesamtanzahl der Elemente einer Page berechnet:

$$bmpSize = \left\lceil \frac{elements_{max}}{8} \right\rceil$$

Das Attribut *elements* gibt die Anzahl der aktuell in der Page gespeicherten Werte an. Die Größe der Page wird durch *pageSize* in Bytes angegeben. Die Größe der initialen Page ist konfigurierbar. Ist die maximale Größe einer Page erreicht, wird für das nächste Element eine weitere Page der zweifachen Größe der aktuellen Page erstellt. Der next-Pointer der aktuellen Page wird auf die Startadresse der neuen Page gesetzt. Das Attribute *offset* gibt die Position innerhalb des Bodys der Page zurück, an dem neue Elemente eingefügt werden. Den Beginn des Bodys bilden die Bitmaps *defined* und *null*. Die defined-Bitmap gibt an, ob ein Wert gesetzt wurde. Zum Beispiel

wird eine nachträglich hinzugefügte Column mit *undefined*-Werten gefüllt. Die null-bitmap gibt an, ob ein Wert NULL ist. Elemente, die null oder undefined gesetzt wurden, werden im Body nicht gespeichert.

Mit Einführung der Kompression wurde die Page in einen unkomprimierten und komprimierten Bereich unterteilt. Da die Größe einer Page nach der Kompression nicht mehr ihrer PageSize entspricht, wird der Pointer *compressEnd* genutzt, um das neue Ende der Page zu kennzeichnen. Um einen einfachen Zugriff auf den compressEnd-Pointer zu gewährleisten, befindet sich dieser als erstes Attribut in einem unkomprimierten Teil der Page. Als Voraussetzung für die Nutzung von SIMD-Operationen, wie sie zum Beispiel in SIMD-FastPFOR (siehe Abschnitt 3.3) genutzt werden, muss die Page nach 16-Byte ausgerichtet sein. Daher hat der unkomprimierte Teil eine Mindestgröße von 16 Byte und bietet auf einem 64-Bit System Platz für zwei Pointer. Um ein schnelles Page-Packing (siehe Abschnitt 4.2) zu ermöglichen, befindet sich auch der next-Pointer im unkomprimierten Teil der Page.

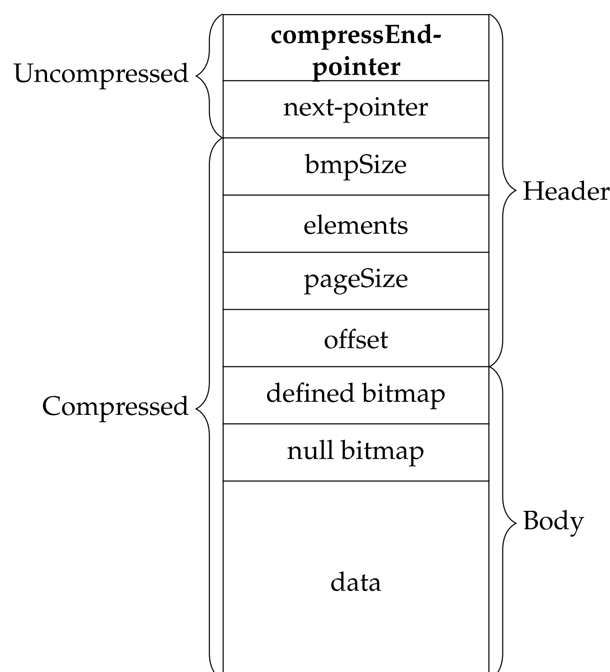


Abbildung 3.2: Aufbau einer Page des Column-Stores in ERIS

Um hybride Storage-Architekturen zu ermöglichen, speichern Column-Groups mehrere Attribute in einer Column. Demnach enthält ein Eintrag in einer Column ein Tupel von Werten. Jedes Tupelelement entspricht einem Attributwert. Da nur eine Column für den Zugriff auf mehrere Attribute verwendet werden muss, führt dies zu einer Optimierung von Operationen auf den Werten eines Tupels. Für eine optimierte Nutzung der Kompressionsoperatoren (siehe Kapitel 5) werden die Attribute nach ihrer Datengröße gruppiert. Somit befinden sich in jedem Tupel Werte der gleichen Größe.

3.3 LEICHTGEWICHTIGE KOMPRESSIONSALGORITHMEN

Null-Suppression

Das Null-Suppression Kompressionsschema SIMD-FastPFOR [9] (siehe Abschnitt 2.3) wird in dieser Arbeit neben der Datenkompression auch für die Kompressionsoperatoren verwendet. Der Aufbau des Schemas ist in Abbildung 3.3 gezeigt.

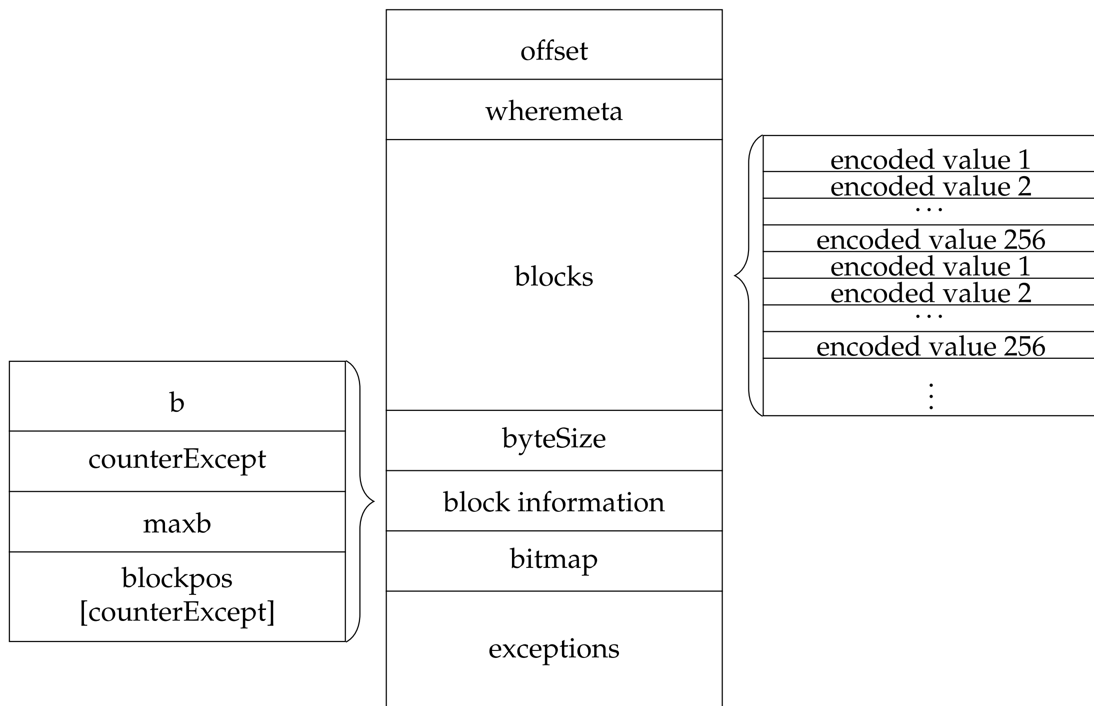
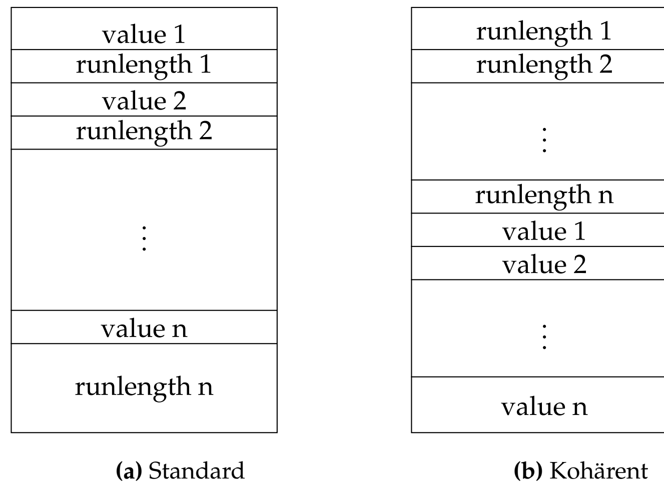


Abbildung 3.3: Aufbau einer komprimierten Page im FastPFOR Format

Der *offset* gibt die Gesamtgröße der Page an. Das Attribut *wheremeta* enthält die Größe der Datenblöcke, damit von diesem Punkt aus direkt zu den Metadaten gesprungen werden kann. Die Datenblöcke enthalten je Block 256 kodierte Werte, die nacheinander gespeichert werden. Da aufgrund der Blockgröße ein 16-Byte Speicheralignment ermöglicht wird, lassen sich die Datenblöcke mit SIMD-Operationen verarbeiten. Nach den Datenblöcken werden die Kompressionsmetadaten bestehend aus Blockinformationsdaten und den exceptions gespeichert. Das Attribut *byteSize* gibt die Größe der Blockinformationen an. In den Blockinformationen werden in der Reihenfolge der Blöcke für jeden Block Metadaten gespeichert. Diese bestehen aus der zur Kodierung verwendeten Bitgröße *b*, der Anzahl der exceptions *counterExcept*, der maximalen Bitgröße *maxb* und den Positionen der exceptions innerhalb des Blocks *blockpos*. Für jede Bitgröße *b* wird ein Array erstellt, in dem die exceptions gespeichert werden. Um zu ermitteln, welches der 32 Arrays leer ist, wird eine *bitmap* vor den exceptions gespeichert. Jedes gesetzte Bit signalisiert ein gefülltes exception-Array an dieser Stelle. Werte, die keinen weiteren Block füllen konnten, werden mit dem *Variable Byte* Algorithmus nach Zhang et al. [16] kodiert/dekodiert.

Run-Length-Encoding

Der Run-Length-Encoding Algorithmus kodiert gleiche Werte in einer Sequenz in ein Paar *(Wert; Runlength)* und legt diese Paare nacheinander in die Column ab. Das Attribut *runlength* gibt hierbei die Anzahl der Werte *value* einer Sequenz an (siehe Abbildung 3.4a).



(a) Standard **(b)** Kohärent

Abbildung 3.4: Aufbau einer komprimierten Page im Standard- und kohärenten Run-Length-Encoding Format

Aufgrund der Kodierung in Wert-Runlength-Paaren entsteht eine Symmetrieeigenschaft, die eine Änderung des Formats mit Hilfe von swap-Funktionen vereinfacht. In dieser Arbeit wurde eine Konvertierungsfunktion bestehend aus einer swap-Funktion entwickelt (siehe Algorithmus 1), die eine RLE-kodierte Page in eine Value-Sektion und Runlength-Sektion unterteilt (siehe Abbildung 3.4b). Die dadurch entstehende Kohärenz optimiert eine anschließende weitere Komprimierung. Eine Anwendung ist in Kapitel 4.3 mit dem SIMD-FastPFOR Kompressionsschema dargestellt.

Data: Eine RLE-kodierte Page mit den Paaren $Wert_1 Runlength_2 \dots Wert_{n-1} Runlength_n$
Result: Eine RLE-kodierte Page mit kohärenten Bereichen der Form $Runlength_n \dots Runlength_2 Wert_{n-1} \dots Wert_1$

```
start := 1;
end := n;
while start < end do
    temp := Wert_start;
    Wert_start := Runlength_end;
    Runlength_end := temp;
    start := start + 2;
    end := end - 2;
end
```

Algorithm 1: Kovertierungsfunktion für eine kohärente Partitionierung einer RLE-kodierten Page

Dictionary-Encoding

In dieser Arbeit wurde das *Order-Preserving-Dictionary-Encoding*-Format verwendet (siehe Abbildung 3.5). Die Erstellung des Wörterbuchs erfolgt durch das Speichern der zu kodierenden Werte in einem Array. Jeder Eintrag des Arrays hat eine Größe von vier Byte. Jeder Wert in der Column wird durch den Wörterbuch-Index des Wertes ersetzt. Die Größe des verwendeten Datentyps eines Index-Werts hängt von der Anzahl an Einträgen im Dictionary ab. Es wird der kleinstmögliche Datentyp verwendet, mit dem alle Indizes des Dictionarys abgebildet werden können. Sollte der Wert noch nicht im Wörterbuch vorhanden sein, wird der Wert dem Wörterbuch hinzugefügt. Das Array wird als Wörterbuch am Ende der Page gespeichert.

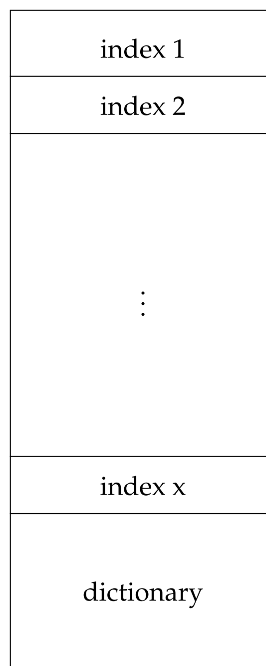


Abbildung 3.5: Aufbau einer komprimierten Page im Dictionary-Encoding Format

4 COLUMN-STORE KOMPRESSION

Die spaltenorientierte Speicherung der Daten im Column-Store ermöglicht die Anwendung leichtgewichtiger Kompressionsalgorithmen auf ihnen. Diese reduzieren den Speicherverbrauch der Pages einer Column. In diesem Kapitel wird die Anwendung der verwendeten Kompressionsalgorithmen (siehe Abschnitt 3.3) beschrieben.

In Abschnitt 4.1 wird das allgemeine Kompressionsinterface für die konkreten Kompressionsalgorithmen beschrieben. Neben der manuellen Auswahl von konkreten Kompressionsalgorithmen wurde auch die adaptive Auswahl basierend auf Heuristiken umgesetzt (siehe Abschnitt 4.4).

Um den für die Pages allokierten Speicher möglichst optimal auszunutzen, wurde ein Page-Packing-Algorithmus implementiert (siehe Abschnitt 4.2). Dieser wird nach der Kompression einer Page ausgeführt und verschiebt die komprimierten Daten in den freien Speicherbereich einer anderen komprimierten Page. Der Speicherplatz der ursprünglichen Page kann daraufhin deallokiert werden.

Um höhere Kompressionsraten zu erreichen, wurde die Verkettung von Kompressionsalgorithmen ermöglicht (siehe Abschnitt 4.3). Die Kompressionsalgorithmen können beliebig verkettet werden. Dabei bildet die Ausgabe einer Kompression die Eingabe für die nächste Kompression. Folgende Kompressionsalgorithmen können die entstandenen Dateneigenschaften der vorherigen Kompression für eine weitere Verringerung der Speichergröße ausnutzen.

4.1 KOMPRESSIONSINTERFACE

Das Kompressionsinterface ist in Abbildung 4.1 mit den abstrakten Klassen *ColumnStoreCompression* und *CompressionOperators* gezeigt. Das vollständige UML-Diagramm ist in Abbildung 2 im Anhang dargestellt.

Um eine einfache Implementierung der konkreten Kompressionsklassen zu gewährleisten, wurde das *Template-Method-Pattern* verwendet. Die Funktionalität der Kompressionsverkettung (siehe

Abschnitt 4.3) wird in die Funktionen des Public-Scopes (gekennzeichnet mit dem Zeichen "+" vor der Funktion) der Klasse `ColumnStoreCompression` ausgelagert. Die konkreten Kompressionsklassen fügen ihre Funktionalität durch Implementierung der virtuellen Funktionen des Protected-Scopes (gekennzeichnet mit dem Zeichen "#" vor der Funktion) ein.

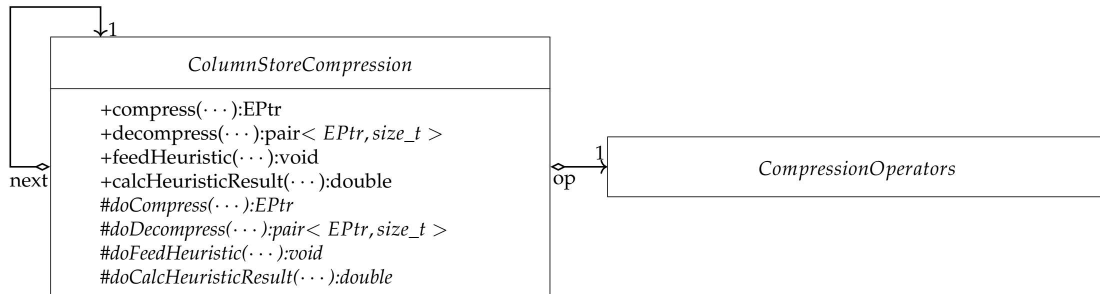


Abbildung 4.1: UML-Klassendiagramm des Kompressionsinterfaces (die Funktionsparameter und die *CompressionOperators*-Funktionen sind zur Übersichtlichkeit nicht dargestellt. Das vollständige Diagramm ist im Anhang abgebildet. EPtr ist ein Pointer-Datentyp)

Jedes `ColumnStoreCompression`-Objekt besitzt ein `CompressionOperators`-Objekt. Konkrete Objekte von Kompressionsklassen erben von der abstrakten Klasse `ColumnStoreCompression` und erstellen eine Instanz des jeweiligen `CompressionOperators`. Eine konkrete `CompressionOperators`-Klasse erbt von der abstrakten Klasse `CompressionOperators`.

Die Kompression wird mit Hilfe des *CompressionHint* aktiviert. Dafür muss dem CompressionHint eine konkrete Instanz des gewünschten Kompressionsschemas übergeben werden. Der Column-Store komprimiert eine Page, sobald diese voll ist, mit der `compress`-Funktion der gesetzten `ColumnStoreCompression`-Instanz. Die `compress`-Funktion führt zunächst die Kompressionskette (siehe Abschnitt 4.3) aus, sofern der `next`-Pointer gesetzt ist. Die konkrete Kompression wird durch den Aufruf der virtuellen `doCompress`-Funktion ausgeführt. Diese führt die Kompression in einem temporären Speicher aus. Erbrachte die Kompression eine Verbesserung der Speichergröße, wird die Page durch den temporären Speicher ersetzt. Zurückgegeben wird ein Pointer (`EPtr`), der auf das neue Ende der komprimierten Page zeigt. Wenn ein nullpointer zurückgegeben wird, konnte keine Verbesserung im Speicherverbrauch erzielt werden. Die Kompression wird daraufhin verworfen und die originale Page bleibt erhalten. Der Rückgabe-Pointer wird im Attribut *compressEnd* im Page-Header (siehe Abschnitt 3.2) gespeichert.

Die Dekompression erfolgt ohne Nutzung von Kompressionsoperatoren direkt bei Anfrage der Daten. Sofern das *compressEnd*-Attribut der Page gesetzt ist, wird die Startadresse und die *compressEnd*-Adresse der Page an die `decompress`-Funktion der `ColumnStoreCompression`-Instanz übergeben. Ist der `next`-Pointer gesetzt, führt die `decompress`-Funktion die Dekompressionskette aus. Die konkrete Dekompression erfolgt in der virtuellen `doDecompress`-Funktion. Das Ergebnis der Dekompression wird in einen temporären Speicher geschrieben, dessen Startadresse und Größe in Form eines Paares zurückgegeben wird.

Neben den konkreten Kompressionsklassen besteht auch die Möglichkeit einer automatischen Auswahl eines geeigneten Kompressionsschemas (siehe Abschnitt 4.4). Dafür bietet jede Kompressionsklasse Heuristiken an, die über die Funktion *doFeedHeuristic* die eingefügten Werte übergeben bekommt. Anhand dieser Werte kann die Kompressionsrate abgeschätzt werden. Über die Funktion `calcHeuristicResult` berechnet jedes Kompressionsschema die jeweils mögliche

Kompressionsrate. Verwendet wird das Kompressionsschema mit der besten Kompressionsrate.

4.2 PAGE-PACKING

Um den durch die Kompression frei gewordenen Speicher möglichst optimal zu nutzen, werden komprimierte Pages in den freien Speicher anderer Pages verschoben. Der Vorgang ist schematisch in Abbildung 4.2 mit den drei Pages A, B, C dargestellt. Die Bereiche A_c, B_c, C_c stellen eine komprimierte Page dar. Der unkomprimierte Teil des Headers (siehe Abschnitt 3.2) ist zur einfacheren Darstellung Teil dieses Blocks. Die jeweils darunter befindlichen Bereiche A_f, B_f, C_f kennzeichnen den nach der Kompression frei gewordenen Speicher der jeweiligen Page. Die grauen Pfeile zeigen ein Verschieben komprimierter Pages an. Die grau hinterlegten Bereiche A_c', B_c' zeigen den neuen Speicherort einer verschobenen komprimierten Page an. Die Funktion $del()$ wird nach einer Pageverschiebung ausgeführt und deallokiert den Bereich, der als Argument übergeben wird. Die Ausführung $del(B_c + B_f)$ löscht zum Beispiel die Bereiche B_c und B_f und damit die komplette Page B.

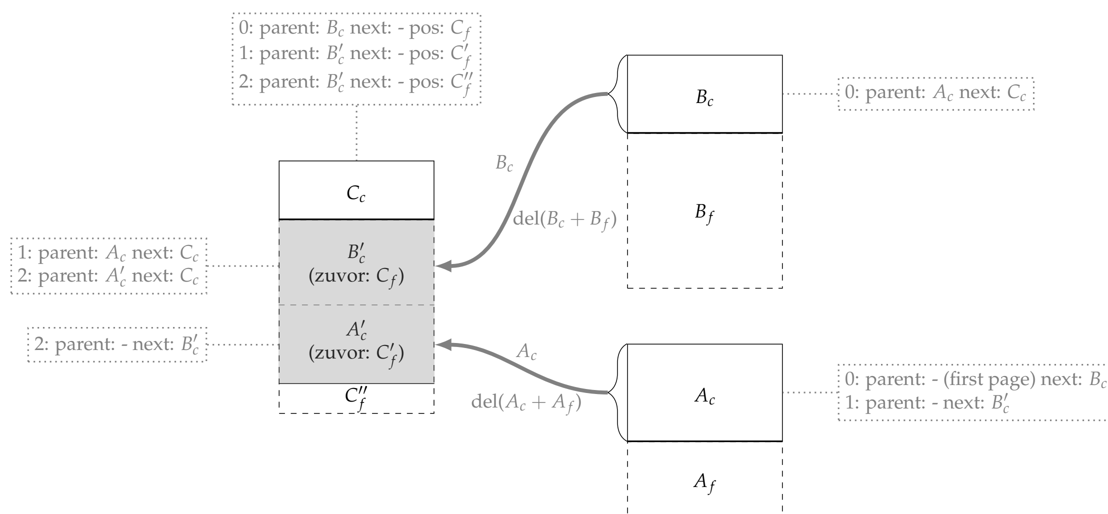


Abbildung 4.2: Page-Packing-Algorithmus. Freier Speicher ist gestrichelt umrandet. Komprimierte Pages werden in den frei gewordenen Speicher anderer Pages verschoben (grau hinterlegt)

Wenn Page-Packing über den CompressionHint aktiviert wurde, werden Speicherinformationen jeder komprimierten Page in zwei Tabellen gespeichert. Die Tabellen nehmen nur eine begrenzte Anzahl an Einträgen auf. An einem Beispiel soll der Algorithmus verdeutlicht werden. In der Tabelle 4.1 befinden sich die Informationen drei komprimierter Pages aufsteigend sortiert nach ihrem freien Speicherplatz. In der Spalte *free* ist der freie Speicherplatz in Bytes angegeben. Die Spalte *pos* gibt die aktuelle Position in der Page an der weitere Pages hinzugefügt werden können. Die Spalte *originalPage* gibt die Speicheradresse der originalen Page an.

Die Tabelle 4.2 ist absteigend nach ihrem belegten Speicher sortiert, wobei alle Einträge an das Ende der Tabelle verschoben werden. Einträge mit Werten kleiner vier sind von der Sortierung ausgeschlossen, da diese Werte als Flag dienen. Zum Beispiel zeigen diese an, ob eine Page verschoben wurde oder Abhängigkeiten zu anderen Pages aufweist und damit aus den Tabellen

	free	pos	originalPage
A	670	A_f	A
B	1626	B_f	B
C	11574	C_f	C

Tabelle 4.1: Tabelle sortiert nach freiem Speicherplatz

nicht entfernt werden darf. Die Spalte *pos* enthält die aktuelle Position, zu der andere Pages verschoben werden können. Die Größe des allokierten Speichers der originalen Page liefert *pageSize*. Die Speicheradresse der vorherigen Page enthält *parentPage*. Der next-Pointer der *parentPage* zeigt auf die aktuelle Page. Der Wert *pagesIn* gibt an, wie viele Pages sich im Speicherbereich der Page dieses Eintrags befinden. Die Anzahl der Pages, die in diese Page verschoben worden sind, gibt demnach $pagesIn - 1$ an. Der Wert *offset* liefert den Speicheroffset in Bytes, den diese Page zu ihrer überliegenden Page hat. Ein offset ist für die Einhaltung des Speicheralignments notwendig.

	used	pos	pageSize	parentPage	pagesIn	offset
B	6566	B_f	8192	A_c	1	0
C	4810	C_f	16384	B_c	1	0
A	3426	A_f	4096	0	1	0

Tabelle 4.2: Tabelle sortiert nach belegtem Speicherplatz

Mit jeder neuen komprimierten Page wird diese in die Tabellen an erster Stelle eingefügt. Sollte sich bereits ein Eintrag an erster Stelle befinden, wird dieser invalidiert. Dafür wird der Eintrag, der die zu invalidierende Page als parent-Page hat, mit einem Flag versehen. Dieser verhindert die weitere Verschiebung der Page, da nach einer Entfernung der zu invalidierenden Page nicht mehr der next-Pointer dieser Page gesetzt werden kann. Sollte diese Page bereits verschoben worden sein, werden die Pages, die sich im gleichen allokierten Speicherbereich befinden, ebenfalls mit einem Flag zur Verhinderung einer weiteren Verschiebung gekennzeichnet. Ursache ist die mögliche Verschiebung dieser Page, wenn der komplette Speicherbereich verschoben werden sollte. Nachdem die neue Page in die Tabellen eingefügt wurde, versucht der Algorithmus Einträge zu finden, die die Bedingung $used \leq free$ erfüllen. Dafür wird durch die Tabellen 4.1 und 4.2 iteriert. Aufgrund der Sortierung wird in den kleinstmöglichen freien Speicher einer Page aus Tabelle 4.1 eine Page aus Tabelle 4.2 mit dem größten belegten Speicher verschoben.

Schritt 0: Für das Beispiel wurden in die Tabellen bereits drei Einträge hinzugefügt. In diesem initialen Zustand ist der Eintrag A die root-Page, da sie keine parent-Page besitzt. Ihr folgt der Eintrag B mit A_c als parent-Page. Eintrag C ist die zuletzt eingefügte Page für dieses Beispiel mit B_c als parent-Page. Die Spalte *pos* aller Pages zeigt auf ihren jeweiligen freien Speicherbereich.

Schritt 1: In dem gegebenen Beispiel wird aus Tabelle 4.1 der Eintrag C gewählt. Aus der Tabelle 4.2 wird der Eintrag B gewählt. Die komprimierte Page B_c des gefundenen Eintrags B aus Tabelle 4.2 wird in den freien Speicherbereich C_f des gefundenen Eintrags C aus Tabelle 4.1 verschoben. Nach der Verschiebung wird der *compressEnd*-Pointer von B'_c an die neue Position angepasst. Der *next*-Pointer von A wird auf die neue Speicheradresse von B'_c gesetzt. Innerhalb der Tabelle 4.1 wird die Spalte *free* um die Größe von B_c für den Eintrag C verringert. Die Spalte *used* in Tabelle 4.2 wird für den Eintrag C um die Größe von B_c erhöht. Ebenso wird die Speicheradresse *pos* des Eintrags C auf den neuen freien Bereich C'_f gesetzt. Der Eintrag B aus Tabelle 4.2 erhält in seiner *used*-Spalte den Wert 3 zur Kennzeichnung einer stattgefundenen Verschiebung. Die Page

C, die B als parent-Page hatte, erhält die neue Adresse B_c' als parent-Page. Wurde beim Verschieben der Page ein offset benötigt, um das Speicheralignment einzuhalten, wird dies in der Spalte *offset* gespeichert. Für B_c' wurde ein offset von 6 Byte benötigt, um ein 16-Byte Speicheralignment einzuhalten. Nach der Verschiebung ist die ursprüngliche Page B (bestehend aus B_c und B_f) leer und wird über die Funktion $del(B_c + B_f)$ deallokiert. Der Wert der Spalte $free$ aus Tabelle 4.1 wird für den Eintrag B auf 0 gesetzt.

	free	**pos**	**originalPage**
B	0	B_f	B
A	670	A_f	A
C	5007	C_f'	C

Tabelle 4.3: Tabelle sortiert nach freiem Speicherplatz nach der Verschiebung

	used	**pos**	**pageSize**	**parentPage**	**pagesIn**	**offset**
B	3	B_f	8192	A_c	1	6
C	11377	C_f'	16384	B_c'	2	0
A	3426	A_f	4096	0	1	0

Tabelle 4.4: Tabelle sortiert nach belegtem Speicherplatz nach der Verschiebung

Die Abbildung 4.2 zeigt zusätzlich den nächsten Schritt des Algorithmus, der in den gezeigten Tabellen nicht dargestellt wurde. Nachdem die komprimierte Page B_c verschoben wurde, wird der Bereich A_c nach C_f' verschoben, da die Bedingung $used_{A_c} \leq free_{C_f'}$ erfüllt wird. Die beteiligten Größen und Speicheradressen werden nach der Verschiebung angepasst und die Page A mit der Funktion $del(A_c + A_f)$ deallokiert.

Schritt 2: In dem gegebenen Beispiel wird aus Tabelle 4.3 der Eintrag C gewählt. Aus der Tabelle 4.4 wird der Eintrag A gewählt. Die komprimierte Page A_c des gefundenen Eintrags A wird in den freien Speicherbereich C_f' des gefundenen Eintrags C verschoben. Nach der Verschiebung wird der *compressEnd*-Pointer von A_c' an die neue Position angepasst. Innerhalb der Tabelle 4.3 wird die Spalte *free* des Eintrags C um die Größe von A_c verringert. Die Spalte *used* in Tabelle 4.4 wird für den Eintrag C um die Größe von A_c erhöht. Ebenso wird die Speicheradresse *pos* auf den neuen freien Bereich C_f'' gesetzt. Die Page B_c', die A_c als parent-Page hatte, erhält die neue Adresse A_c' als parent-Page. Nach der Verschiebung ist die ursprüngliche Page A (bestehend aus A_c und A_f) leer und wird über die Funktion $del(A_c + A_f)$ deallokiert. Der Wert der Spalte $free$ aus Tabelle 4.3 wird für den Eintrag A auf 0 gesetzt.

Pages, die verschobene Pages enthalten, können in weitere Pages verschoben werden, sofern die Speicherbedingung erfüllt wird. Dafür müssen die beteiligten Speicheradressen aller enthaltenen Pages neu gesetzt werden. Der Algorithmus iteriert durch alle enthaltenen Pages und passt die Speicheradressen in deren Header an. In obigem Beispiel enthält die Page C die Pages B als B_c' und A als A_c'. Gibt es im weiteren Verlauf eine Page D für die gilt $used_{C_c''} \leq free_{D_f}$ ($used_{C_c''} = used_{C_c} + used_{B_c'} + used_{A_c'}$), so werden die Bereiche C_c, B_c' und A_c' nach D_f verschoben. Die beteiligten Speicheradressen der drei verschobenen Pages werden entsprechend angepasst. Die Abbildung 3 im Anhang verdeutlicht das Szenario der Verschiebung mehrerer Pages.

Die Tabellen wurden für dieses Beispiel gekürzt und vereinfacht. Die originalen Tabellen mit Speicheradressen befinden sich im Anhang (siehe Tabellen 1, 2, 3, 4 im Anhang). Um direkt auf

den Header zugreifen zu können, enthalten die Tabellen zusätzlich die Spalte *currentPage*, welche die Startadresse der aktuellen Page enthält.

4.3 KOMPRESSIONSVERKETTUNG

Über den `next`-Pointer in der Klasse `ColumnStoreCompression` (siehe Abbildung 4.1) lassen sich Kompressionsalgorithmen verketten. Dabei ist die Ausgabe einer Kompression die Eingabe der nächsten Kompression. In einer optimalen Kompressionskette nutzen nachfolgende Kompressionsalgorithmen Eigenschaften der vorherigen Kompression aus, um eine Verbesserung der Kompressionsrate zu erreichen. Zum Beispiel kann eine kohärente Run-Length kodierte Page (siehe Abschnitt 3.3) die Kompressionsrate von SIMD-FastPFOR verbessern, da insbesondere der Run-Length Bereich häufig kleine Werte enthält.

Da für eine einfache Implementierung der konkreten Kompressionsklassen die Logik der Kompressionskette in die `ColumnStoreCompression`-Klasse ausgelagert wurde, haben die Kompressionsklassen keinen Einfluss auf die Kompressionskette. Somit können auch keine spezifischen Teile einer komprimierten Page (wie zum Beispiel einzelne Blöcke bei FastPFOR oder Index-Einträge/Dictionary bei Dictionary-Encoding) an den nächsten Kompressionsalgorithmus in der Kette übergeben werden. Eine effektivere Kompression kann erreicht werden, wenn in der Kette folgende Kompressionsalgorithmen Daten bekommen, die für sie optimale Eigenschaften aufweisen. Da für diese Variante die `ColumnStoreCompression`-Klasse Informationen über die Eigenschaften und Optimierungsmöglichkeiten der Kompressionsalgorithmen der Kette haben muss, erhöht dies die Komplexität erheblich und wurde in dieser Arbeit nicht weiter betrachtet. Eine weitere Möglichkeit der Verbesserung der Kompressionsrate besteht darin, als nachfolgenden Kompressionsalgorithmus eine Implementierungsvariante zu wählen, die speziell auf die Ausgabe des vorherigen Kompressionsalgorithmus abgestimmt ist.

Die Kompressionsverkettung ist schematisch in Abbildung 4.3 dargestellt. Eine unkomprimierte Page wird mit dem Kompressionsalgorithmus A komprimiert. Die erstellte komprimierte Page wird anschließend als Eingabe für den Kompressionsalgorithmus B verwendet, dessen Ergebnis ist die Eingabe für Algorithmus C. Die Dekompression erfolgt analog in umgekehrter Reihenfolge. Wenn eine Kompression erfolgreich war, wird am Ende der komprimierten Page ein Bit entsprechend der Position in der Kompressionskette gesetzt. Brachte eine Kompression keine Vorteile, wird die Page unverändert direkt dem nächsten Kompressionsalgorithmus in der Kette übergeben.

Eine Dekompression wird erst dann ausgeführt, wenn die ausgelesene Bitmap die aktuelle Position in der Kompressionsliste beinhaltet.

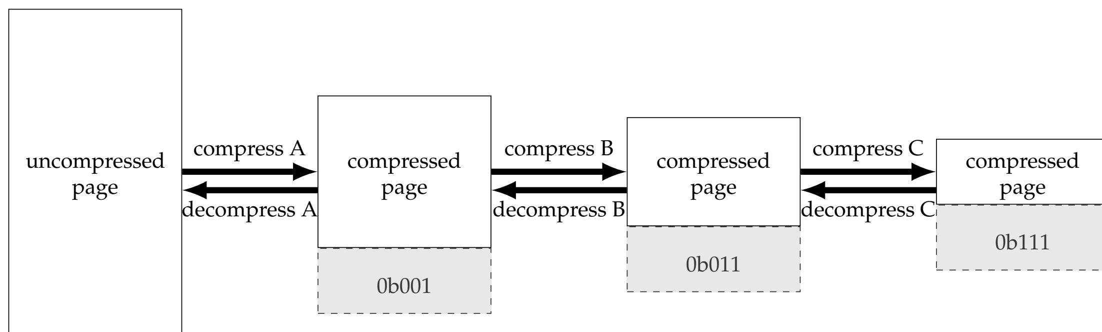


Abbildung 4.3: Kompressionsverkettung mit den drei Kompressionsalgorithmen A, B und C

4.4 AUTO-KOMPRESSION

Zusätzlich zu den drei Kompressionsklassen Run-Length-Encoding, Null-Suppression und Dictionary-Encoding wurde auch eine Auto-Kompressionsklasse entwickelt, die auf Basis der in den Column-Store eingefügten Daten ein geeignetes Kompressionsschema selbstständig auswählen kann. Voraussetzung ist die Implementierung einer Heuristik je konkretem Kompressionsschema. Einer Auto-Kompressionsinstanz wird eine Menge von Kompressionsalgorithmen als Auswahlkandidaten übergeben (Attribut *candidates* in Abb. 2 im Anhang). Jeder eingefügte Wert wird an die Heuristiken der gegebenen Kompressionalgorithmen übergeben. Es werden 32Bit Werte für die Heuristiken betrachtet. Sobald die Kompression einer Page stattfinden soll, fragt die Auto-Kompression alle gegebenen Kompressionen nach dem Ergebnis ihrer jeweiligen Heuristik ab. Das Ergebnis einer Heuristik soll die wahrscheinlich erreichbare Kompressionsrate ($\frac{compressedSize}{uncompressedSize}$) angeben. Die Auto-Kompression berücksichtigt dabei nur Werte im Intervall $[0, 1)$. Seien die Ergebnisse der Kompressionsheuristiken $h_1 \dots h_n$ gegeben und jedes Ergebnis liegt im Intervall $[0, 1)$. Die Entscheidung für ein Kompressionsschema k erfolgt durch Wahl eines Heuristik-Ergebnisses h_k nach Gleichung 4.1.

$$h_k = min(h_1, h_2, \dots, h_n) \tag{4.1}$$

Die Heuristiken der Kompressionsalgorithmen versuchen aus den erhaltenen Daten, auf Grundlage ihres Kompressionsschemas, die Kompressionsrate abzuschätzen. Nach jeder Kompression wird der Index des aus der Kandidatenliste gewählten Kompressionsalgorithmus in ein Byte am Ende der Page gespeichert. Somit ist sichergestellt, dass die Dekompression mit dem korrekten Algorithmus erfolgt.

4.4.1 FastPFOR Heuristik

Die Heuristik für das Kompressionsschema SIMD-FastPFOR ermittelt aus dem Durchschnittswert der gegebenen Werte ($avgValue$) die mittlere Bitlänge. SIMD-FastPFOR erreicht insbesondere mit kleinen Werten bessere Kompressionsraten. Die Heuristik bildet das Verhältnis der mittleren Bitlänge zur Bitlänge 32 ab, da SIMD-FastPFOR zur Kodierung einen 4Byte großen Daten-

typ wählt. Je kleiner das Verhältnis desto besser wird die Kompressionsrate von SIMD-FastPFOR eingeschätzt (siehe Gleichung 4.2).

$$h_{NullSuppression} = \frac{\log_2(avgValue) + 1}{32} \tag{4.2}$$

Der ermittelte Durchschnittswert aus den eingefügten Daten, die für die Heuristik zur Verfügung stehen, entspricht dem Wert $avgValue$ in der Gleichung. Die Werte für die Heuristik werden vom Column-Store mit Hilfe der `feedHeuristic`-Funktion übergeben.

4.4.2 Run-Length-Encoding Heuristik

Die Heuristik des Kompressionsschemas Run-Length-Encoding ermittelt zunächst die Anzahl unterschiedlicher Runs ($valueChanges$). Die Variable n entspricht der Anzahl aller Werte.

$$h_{RunLength} = \frac{2 * (valueChanges + 1)}{n} \tag{4.3}$$

Da Run-Length-Encoding für jeden Run ein Wertepaar erstellt, lässt sich die Kompressionsrate über die Anzahl wechselnder Werte im Verhältnis zur Gesamtanzahl der Werte ermitteln.

4.4.3 Dictionary-Encoding Heuristik

Für die Heuristik des Kompressionsschemas Dictionary-Encoding ist die Anzahl unterschiedlicher Werte notwendig. Diese wird in der Variable n_{dist} gespeichert und über eine boolsche Hash-Map ermittelt. Wurde ein eingefügter Wert in der Hash-Map nicht gefunden, wird der boolsche Wert an dieser Stelle auf `true` gesetzt und n_{dist} inkrementiert. Mit einer boolschen Hash-Map wird nur ein Zugriff auf die Map je Wert benötigt. Die Anzahl aller Werte wird in n gespeichert.

Die Gleichung 4.4 bestimmt das Verhältnis der Anzahl unterschiedlicher Werte zur Anzahl aller Werte unter Berücksichtigung der Einträge im Dictionary. Je weniger unterschiedliche Werte in einer Page existieren, desto weniger Einträge werden im Dictionary benötigt. Das Verhältnis $\frac{n_{dist}}{n}$ dient als Korrekturwert, um den Einfluss der Größe des Dictionarys auf die Kompressionsrate zu kompensieren. Die Variable *effectiveBytes* enthält die verwendete Größe eines Index-Eintrags in Bytes. Dementsprechend gibt das Verhältnis $\frac{n * effectiveBytes}{4 * n}$ den Einfluss der Index-Einträge auf die Kompressionsrate an.

$$h_{Dictionary} = \frac{n * effectiveBytes + 4 * n_{dist}}{4 * n} \tag{4.4}$$

5 KOMPRESSIONSOPERATOREN

Soll eine Scan-Operation auf komprimierten Daten ausgeführt werden, ist im trivialen Fall eine Dekompression der Daten zuvor notwendig. Die anschließende Scan-Operation lädt die Werte wieder über den Speicher-Bus in die CPU. Da der Speicher-Bus einen Bottleneck darstellt, hat die Dekompression den zeitaufwendigsten Anteil an der kompletten Scan-Operation. Zudem verursacht die Dekompression eine hohe Auslastung der CPU. Mit Hilfe der Verwendung von Kompressionsoperatoren, die direkt auf komprimierten Daten Operationen ausführen können, ist die Dekompression der Daten nicht mehr notwendig.

Für die Verwendung von Kompressionsoperatoren müssen aus jedem Kompressionsschema Informationen zu den Originaldaten ausgelesen werden. Ein schnelles direktes Verarbeiten komprimierter Daten kann durch das Überspringen von Datenbereichen auf Grundlage von Kompressionsmetadaten und der Anwendung der Operationen auf den komprimierten Daten erfolgen.

In diesem Kapitel wird das Interface für die Kompressionsoperatoren (siehe Abschnitt 5.1.1) und deren Funktionsweise beschrieben (siehe Abschnitt 5.1.2). Spezielle Eigenschaften und Funktionsweisen der Operatoren werden für die einzelnen Kompressionsalgorithmen in den Abschnitten 5.2, 5.3 und 5.4 beschrieben. Die Umsetzung von Aggregationsoperatoren und Verknüpfungsoperatoren werden in den Abschnitten 5.5.1 und 5.5.2 dargestellt.

5.1 INTERFACE

Über das Interface der Kompressionsoperatoren können Ausdrücke übergeben werden, die auf den komprimierten Daten ausgeführt werden sollen. Die folgenden beiden Abschnitte beschreiben den Aufbau des Interfaces und die Funktionsweise der Kompressionsoperatoren.

5.1.1 CompressionOperators-Interface

Das Interface zu den Kompressionsoperatoren ist in Abbildung 5.1 dargestellt. Im Gegensatz zu den bisherigen Möglichkeiten, auf komprimierten Daten zu arbeiten (siehe Abschnitt 2.4), erlauben die Kompressionsoperatoren die direkte Anwendung von Operationsausdrücken auf den komprimierten Daten. Damit sind Zwischenergebnisse für das Verarbeiten mehrteiliger Operationen nicht mehr notwendig. Stattdessen lassen sich mehrteilige Datenoperationen wie zum Beispiel $(x < 100) + 42$ definieren. Der Ausdruck wird direkt auf den komprimierten Daten ausgeführt und besteht aus einer $lookup$-Operation, die alle Werte kleiner 100 auswählt und dann mit diesen eine Addition mit der Zahl 42 ausführt.

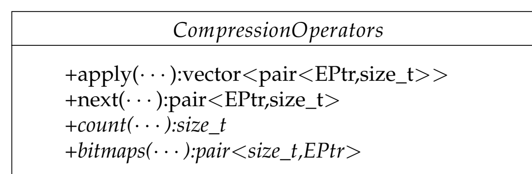


Abbildung 5.1: Interface-Klasse der Kompressionsoperatoren (Das vollständige Diagramm ist im Anhang abgebildet. EPtr ist ein Pointer-Datentyp)

Die Ausdrücke werden mit der $apply$-Funktion angewandt. Die $apply$-Funktion nimmt einen Ausdruck entgegen und gibt einen Ergebnisvektor mit Wert-Positions-Paaren zurück. Die *CompressedValue*-Variablen (siehe Abschnitt 5.1.2) müssen die komprimierten Pages übergeben bekommen.

Die Funktion *count* erhält eine komprimierte Page und einen optionalen Datenoperationsausdruck. Ist kein Ausdruck gegeben, liefert *count* die Anzahl aller Werte zurück. Die Anzahl aller Werte ist im Header-Attribut *elements* (siehe Abschnitt 3.2) gespeichert. Da ein Kompressionsoperator diesen Wert durch eine Dekomprimierung des zweiten Header-Attributs erhält, wird diese Funktion von den Kompressionsoperatorklassen der Kompressionsschemas direkt implementiert. Ebenso wird die Funktion *bitmaps* direkt implementiert, die die Größe einer Bitmap in Bytes und einen Pointer auf den Beginn beider Bitmaps zurückliefert.

Die $next$-Funktion ermöglicht die Iteration durch die komprimierten Daten wie in der Arbeit von Abadi et al. [3] (siehe Abschnitt 2.4) vorgestellt. Der Funktion wird eine komprimierte Page und ein Relationsausdruck übergeben. Der Relationsausdruck wird auf den zurückzugebenden Wert ausgeführt. Liefert der Ausdruck einen positiven Wert zurück, wird der aktuelle Wert dekomprimiert zusammen mit seiner Position zurückgegeben. Wurde der Ausdruck nicht erfüllt, wird mit dem nächsten Wert fortgefahren. Die weitere Verarbeitung der Werte erfolgt über den zurückgegebenen Ergebnisvektor.

Kompressionsoperatoren können auch auf Column-Groups ausgeführt werden. Dafür muss dem Kompressionsoperator die Anzahl an Columns und der Index der Column innerhalb der Group übergeben werden. Der Operator wird auf der gewählten Column der Group ausgeführt.

5.1.2 CompressedOperation-Interface

In dem Operationsausdruck (*expression*) $(x < 100) + 42$ stellt die Variable x einen komprimierten Wert dar, der eine Instanz der Klasse `CompressedValue` ist. Ein `CompressedValue`-Objekt wird in einem `CompressedOperation`-Objekt den Funktionen der `CompressionOperators` übergeben. Ein gekürztes UML-Diagramm des Kompressionsoperator-Interface ist in Abbildung 5.2 dargestellt. In der Abbildung 5.4 ist eine schematische Darstellung zur Funktionsweise der Kompressionsoperatoren dargestellt.

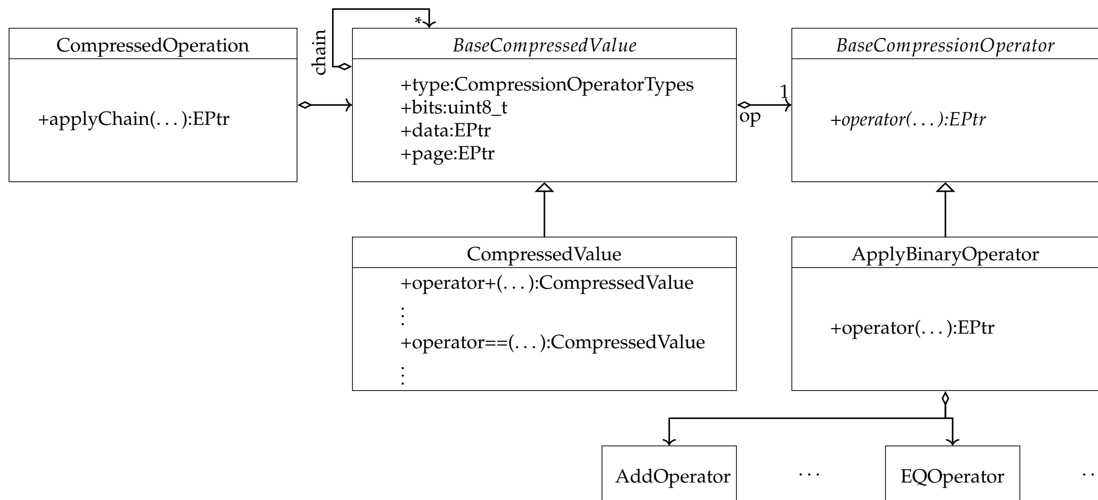


Abbildung 5.2: Interface der Kompressionsoperatoren

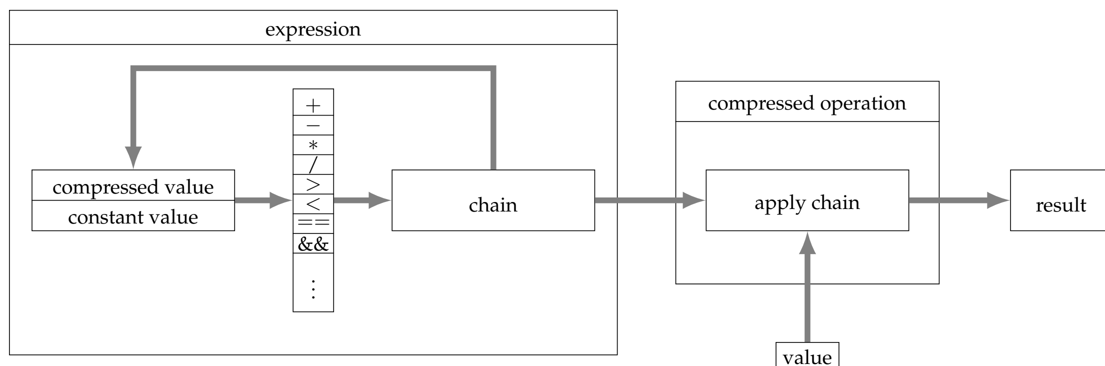


Abbildung 5.3: Schematische Darstellung der Funktionsweise der Kompressionsoperatoren

Die Klasse `CompressedOperation` nimmt eine `BaseCompressedValue`-Instanz auf. Eine `CompressedOperation` führt eine *expression* auf einen übergebenen Wert aus. Die konkrete `CompressedValue`-Klasse erbt von `BaseCompressedValue` und repräsentiert einen komprimierten Wert, der in den *expressions* verwendet werden kann. Dafür überlädt die `CompressedValue`-Klasse Operatorfunktionen, um Informationen über die Operanden in der jeweiligen `CompressedValue`-Instanz zu speichern.

Ein Operand kann eine andere `CompressedValue`-Instanz sein oder ein konstanter Wert. Der Operationsausdruck $(x < 100) + 42$ enthält die Operanden 42, x und 100. x stellt einen kompri-

CompressionOperatorTypes	**Bedeutung**
ARITHMETIC	Arithmetische Operation, standard
LT	<
GT	>
EQ	=
COPY	Kopiert Werte (entspricht einer Dekompression)
JUNCTION	Junktion (zum Beispiel &)

Tabelle 5.1: Typen von Kompressionsoperatoren

mierten Wert dar, 42 und 100 jeweils einen konstanten Wert. Der Operand wird in Form eines Pointers im Attribut `data` gespeichert. Ist der Operand ein konstanter Wert, wird seine Bitlänge im Attribut `bits` gespeichert. Das Attribut `type` enthält den Operatortyp (siehe Tabelle 5.1). Sollen mehrere komprimierte Pages mit einem Operationsausdruck verarbeitet werden, muss dem `CompressedValue`-Konstruktor ein Pointer zur Page mitgegeben werden. Dieser Pointer wird im Attribut `page` gespeichert. Für die Verarbeitung der Operanden müssen einer `CompressedValue`-Instanz bei ihrer Erstellung Datentypinformationen mitgegeben werden.

Die Verwendung der Kompressionsoperatoren beginnt mit der Erstellung einer *expression*. Dafür wird ein Operationsausdruck definiert, der auf den komprimierten Werten angewendet werden soll. In einer *expression* muss mindestens ein `CompressedValue`-Objekt verwendet werden, um dem Compiler mitzuteilen, die Operatoren für komprimierte Werte zu verwenden. Der Operationsausdruck $(x < 100) + 42$ wird zum Beispiel wie folgt gebaut: Zu erst wird der Teilausdruck $x < 100$ evaluiert. Anhand Abbildung 5.3 wird ein *compressedvalue* (x) und ein *constantvalue* (100) mit dem binären Relationsoperator $<$ verwendet und der *chain* hinzugefügt. Der erstellte Teilausdruck wird zusammen mit dem *constantvalue* 42 dem binären Operator $+$ übergeben und der *chain* hinzugefügt. Die *chain* ist ähnlich zu den *evaluator chains*, die in dem Papier *DB2 with BLU acceleration* [14] beschrieben werden. Evaluator chains bestehen aus DB2 BLU Operatoren und führen jeweils eine Operation aus. Sie werden zu einem Ausführungsplan zusammengesetzt und erhöhen dadurch die Ausführungsgeschwindigkeit.

Die *chain* aus der Expression wird einem *compressedoperation*-Objekt übergeben, das als Eingabe für den Kompressionsalgorithmus dient. Die Anwendung der Expression auf den komprimierten Werten wird in Abbildung 5.4 dargestellt.

Zu Beginn jeder Verarbeitung müssen der Page-Header und die Bitmaps übersprungen werden, um direkt die Datensektion des Bodys zu erreichen. Sofern ein erfolgreiches Ergebnis erwartet werden kann, wird auf jeden dekomprimierten Wert die *compressedoperation* ausgeführt und ein valides Ergebnis in einem Ergebnisvektor gespeichert. Die *compressedoperation* wendet auf den übergebenen Wert die $Expression - chain$ an und bricht ab, sobald ein Teilausdruck in der chain ein invalides Ergebnis liefert. Wurde die chain erfolgreich angewendet, enthält der Rückgabewert das Ergebnis der Expression für den übergebenen Wert. Die Anwendung der *compressedoperations* wird für jedes Kompressionsschema in den folgenden Abschnitten erklärt.

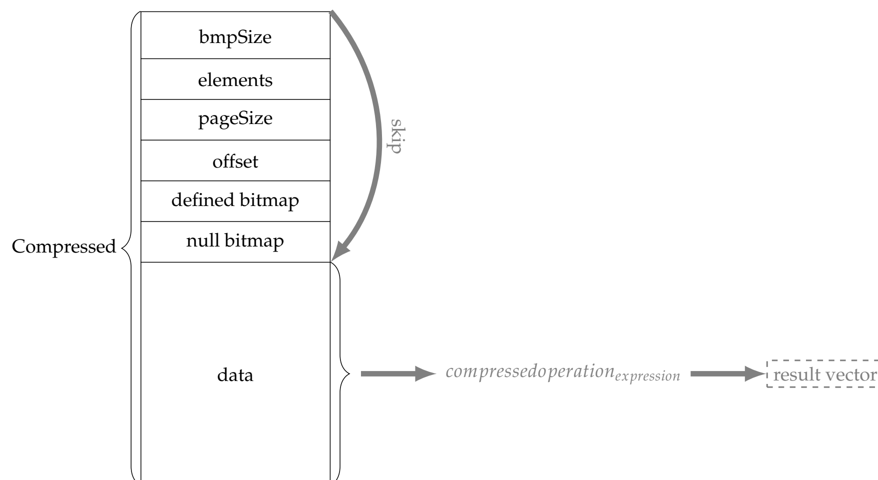


Abbildung 5.4: Schematische Darstellung der Anwendung von Kompressionsoperatoren auf einer komprimierten Page

5.2 NULL-SUPPRESSION OPERATOREN

Bevor die eigentliche Verarbeitung einer komprimierten Page beginnen kann, muss der Header und die Bitmaps einer Page übersprungen werden (siehe Abschnitt 3.2). Daher werden die ersten Blöcke dekomprimiert, bis die Größe der dekomprimierten Bytes der Größe des Headers entspricht. Der erste Wert des dekomprimierten Headers enthält die Größe der Bitmaps und kann direkt ausgelesen werden. Die Größe des Headers, addiert mit den Größen der *defined-bitmap* und *null-bitmap*, wird als offset für den aktuellen Block verwendet. Erst ab diesem offset beginnt der Datenbereich des Bodys. Jeder weitere Block enthält einen Teil des Datenbereichs, sodass der offset-Wert für diese Blöcke auf null gesetzt wird. Eine Dekompression des Headers erfolgt somit einmalig pro komprimierter Page.

Das SIMD-FastPFOR Kompressionsschema speichert für jeden Block die durchschnittliche und die maximale Bitlänge der Werte ab (siehe Abschnitt 3.3). Lediglich die maximale Bitlänge gibt Aufschluss über die enthaltenen Werte und kann somit für die Entscheidung, einen Datenblock zu überspringen, verwendet werden. Indem die Bitlängen aus dem Attribut *bits* aus der Klasse *BaseCompressedValue* mit der maximalen Bitlänge verglichen werden, kann eine Entscheidungsfindung nur über GT-oder EQ-Operatoren geschehen (siehe Abbildung 5.5). Ist die maximale Bitlänge kleiner als die Bitlänge des Operanden, kann es in diesem Block keinen Wert geben, der größer-oder gleich dem Operandenwert ist. Demnach muss der Block nicht weiter verarbeitet werden und wird somit übersprungen.

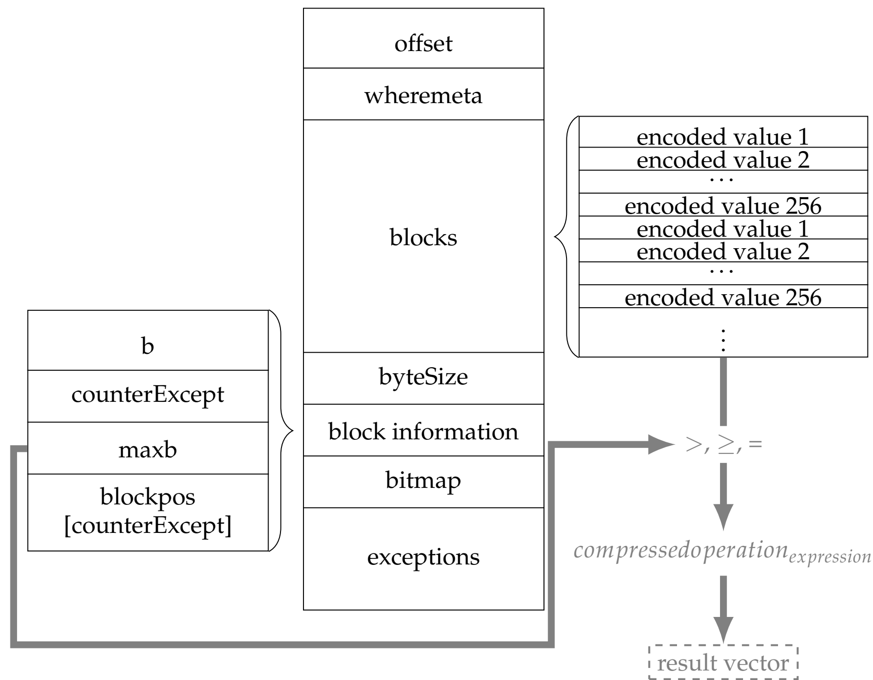


Abbildung 5.5: Anwendung der Kompressionsoperatoren in SIMD-FastPFOR

5.3 RUN-LENGTH-ENCODING OPERATOREN

Das Kompressionsschemas Run-Length-Encoding kodiert Werte als Paar der Form *(Wert,runlength)* (siehe Abschnitt 3.3). Um den Header und die Bitmaps der komprimierten Pages zu überspringen, wird das erste Header-Attribut *bmpSize* dekomprimiert und die Größe des Headers sowie die zweifache Größe der Bitmapgröße übersprungen. Dadurch wird direkt der Datenbereich erreicht.

Die Verarbeitung des komprimierten Datenbereiches erfolgt durch Anwendung des Datenoperationsausdrucks auf den *Wert*-Teil des jeweiligen Kodierungspaares. Wurde der Ausdruck erfolgreich angewendet, wird das Ergebnis gemäß der Run-Length in den Ergebnisvektor geschrieben (siehe Abbildung 5.6). Nach Verarbeitung aller Kodierungspaare wird der Ergebnisvektor $r_1, \ldots, r_{rln}$ zurückgegeben.

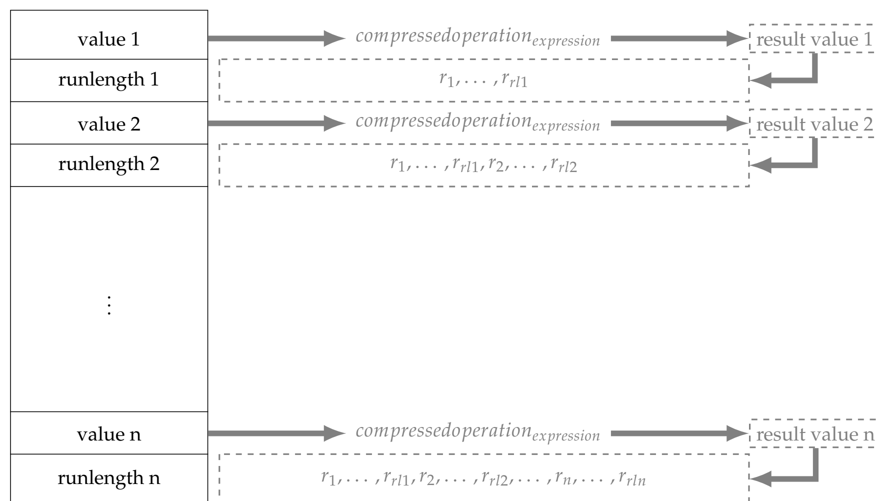


Abbildung 5.6: Anwendung der Kompressionsoperatoren in Run-Length-Encoding

5.4 DICTIONARY-ENCODING OPERATOREN

Da sich die Werte einer Dictionary kodierten Page (siehe Abschnitt 3.3) unkomprimiert im Dictionary befinden, wird der Datenoperationsausdruck direkt auf den Dictionary-Werten angewendet. Die Ergebnisse erfolgreicher arithmetischer Operationen werden temporär in einem Ergebniswörterbuch gespeichert (siehe Abbildung 5.7). Die Verarbeitung der Werte und Speicherung der Ergebnisse im Ergebniswörterbuch erfolgt mit der Iteration durch die Indizes der komprimierten Page. Für jeden verarbeiteten Indexwert wird ein Bit in einer Bitmap gesetzt. Das Ergebnis von Ausdrücken, die nicht angewendet werden konnten, gleicht einem leeren Eintrag des Ergebniswörterbuches. Daher wird ein erneutes Ausführen durch Überprüfung der Bitmap verhindert. Tritt der Indexwert erneut auf und ist das entsprechende Bit sowie der Wert aus dem Ergebniswörterbuch gesetzt, wird dieser in den Ergebnisvektor kopiert. Wenn alle Indizes verarbeitet wurden, wird der Ergebnisvektor zurückgegeben und das temporäre Ergebniswörterbuch deallokiert.

Vor der Erstellung des Ergebniswörterbuchs muss der Header und die Bitmaps der Page übersprungen werden. Dafür wird das *bmpSize*-Attribut dekomprimiert, um die Größe der Bitmaps zu ermitteln. Durch die Indizes wird solange iteriert, bis die Größe des Headers und der Bitmaps erreicht wurde. Mit der nächsten Iteration beginnt der Datenbereich der komprimierten Page.

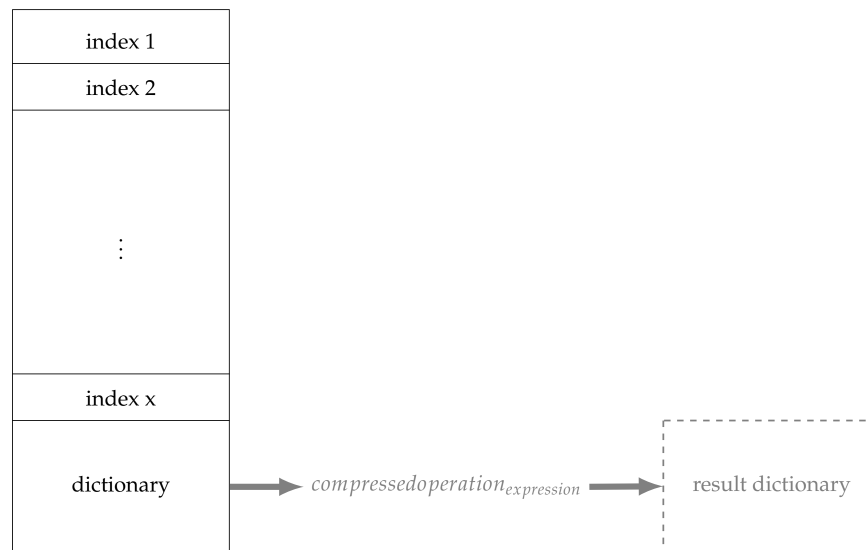


Abbildung 5.7: Anwendung der Kompressionsoperatoren in Dictionary-Encoding

5.5 OPERATORTYPEN

Die folgenden Abschnitte beschreiben die Anwendung von Aggregationsoperatoren und Verknüpfungsoperatoren.

5.5.1 Aggregationsoperatoren

Teilweise können Werte für Aggregationsoperatoren direkt aus den Page-Headern verwendet werden. Zum Beispiel ist für den Operator *Count* die Akkumulation des Werts *elements* aus dem Page-Header jeder Page (siehe Abschnitt 3.2) ausreichend. Für einen effizienten Zugriff auf das Header-Attribut *elements* wurde der Aggregationsoperator *Count* in den Kompressionsalgorithmen umgesetzt. Da aufgrund des Speicher-Alignments einer Page von 16Byte (siehe Abschnitt 3.2) für ein weiteres Attribut im unkomprimierten Bereich 16Byte bereitgestellt werden müssten, wurde in dieser Arbeit die Möglichkeit des Auslesens des *element*-Attributs aus dem komprimierten Header betrachtet.

Die Algorithmen überspringen das erste Attribut *bmpSize* und liefern das zweite Attribut *elements* dekomprimiert zurück (siehe Abbildung 5.8). Das direkte Auslesen des Attributs für den *Count*-Operator kann nur dann angewendet werden, wenn die Anzahl aller enthaltenen Werte zurückgegeben werden soll. Die erreichten Zeitersparnisse sind in Abschnitt 6.4.2 dargestellt.

Um die Auswahl der Werte für den *Count*-Operator einzuschränken, ist die Ausführung einer *expression* auf den Daten zuvor möglich. Dafür wird dem *Count*-Operator neben der komprimierten Page auch ein `CompressedOperation`-Objekt mit der *expression* übergeben. Der *Count*-Operator entspricht dann einem Teilausdruck, der am Ende der *chain* hinzugefügt wird. Für die Realisierung des Operators speichert dieser den letzten Zustand des Zählers. Der Rückgabewert entspricht dem letzten Zustand des *Count*-Operators.

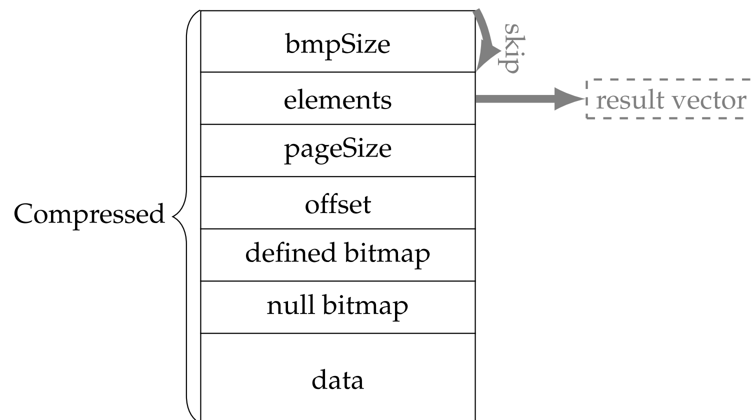


Abbildung 5.8: Funktionsschema des Count-Operators. Liefert die Anzahl aller Werte einer Page zurück

Aggregationsoperatoren können in *expressions* verwendet werden. Sie sind in der Lage, einen Wert aus einer durchgeführten Operation zu speichern. Der gespeicherte Wert kann in einem weiteren Aufruf des Operators wieder verwendet werden. Ein weiteres Beispiel für einen Aggregationsoperator ist der *Sum*-Operator.

Der *Sum*-Operator akkumuliert alle Werte, die an ihn übergeben werden. Zusätzlich wird dem *Sum*-Operator ein Multiplikator mitgegeben, der die Anzahl der Vorkommen des gegebenen Werts entspricht. Dies ist insbesondere in Kombination mit dem Kompressionsalgorithmus Run-Length-Encoding von Vorteil (siehe Abschnitt 3.3 und 6.4.2), da die Run-Length dem Multiplikator entspricht. Der Standardwert des Multiplikators ist 1. Der Operator addiert den gegebenen Wert, multipliziert mit dem Multiplikator, auf den bisherigen Wert. Der Rückgabewert des *Sum*-Operators entspricht dem aktuellen Akkumulationswert.

5.5.2 Junktoren

Die Funktionsweise der Junktoren innerhalb von Ausdrücken wird in Abbildung 5.9 schematisch dargestellt. Für die Verwendung von Verknüpfungsoperatoren wird die *expression* in ihre Teilausdrücke aufgeteilt. Die Teilausdrücke werden separat einer *compressedoperation* übergeben und auf der Page, die der jeweiligen *compressedvalue*-Instanz übergeben wurde, ausgeführt. Die Ergebnisvektoren der *compressedoperation* enthalten Paare, bestehend aus dem Ergebniswert und der jeweiligen Position des Wertes innerhalb der Page.

Zum Beispiel wird der Ausdruck $x > 100\ \&\&\ y < 200$ in die Teilausdrücke $x > 100$ und $y < 200$ aufgeteilt (siehe Abbildung 5.10). Die Variablen x und y bilden `CompressedValue`-Instanzen, denen jeweils ein Pointer zu einer Page bei der Konstruktion übergeben wurde. Somit stellen x und y innerhalb ihrer Teilausdrücke Werte ihrer jeweiligen Page dar. Die Teilausdrücke werden separat auf ihren Pages ausgeführt und das Ergebnis mit der jeweiligen Position in Vektoren gespeichert. Auf den Wert-Positions-Paaren der Ergebnisvektoren wird der &&-Operator angewendet. Dieser überprüft die Positionen der Werte auf Gleichheit und ob beide Teilausdrücke ein valides Ergebnis zurückgeliefert haben. Sind beide Bedingungen erfüllt, werden die Paare einem Ergebnisvektor hinzugefügt.

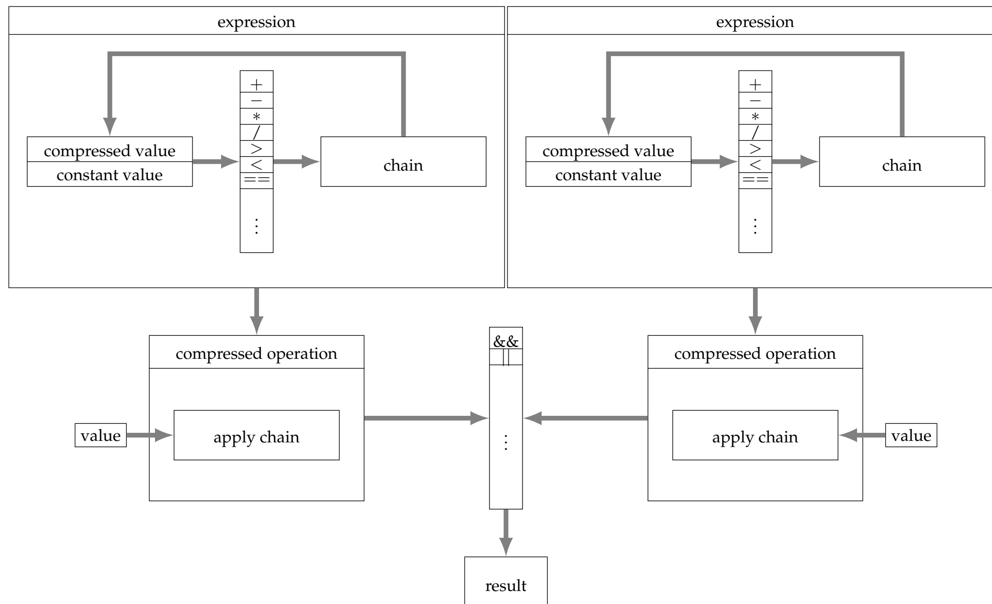


Abbildung 5.9: Funktionsschema der Junktionsoperatoren

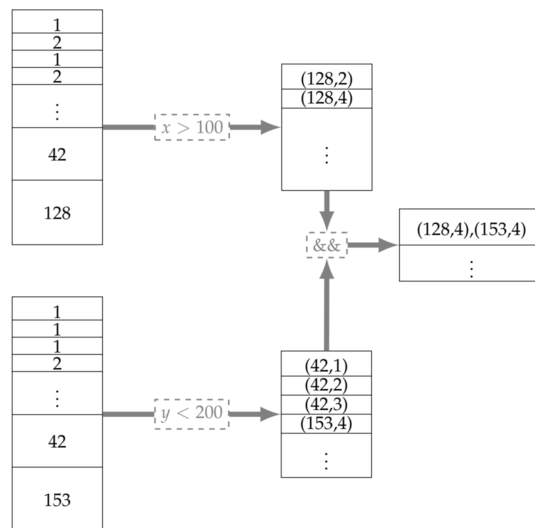


Abbildung 5.10: Beispielanwendung der Junktionsoperatoren anhand mit Dictionary-Encoding komprimierten Pages

6 EVALUATION

Für die Evaluation der Kompressionsalgorithmen und der Kompressionsoperatoren wurden Datengeneratoren implementiert, die Zahlenfolgen mit einer gegebenen Anzahl an Werten erzeugen können. Die Arten der Zahlenfolgen sind in Tabelle 6.1 dargestellt. Für die Generatoren *AlternatingNumSeries* und *HalfRangeAlternatingNumSeries* wurden für die Erstellung der Zahlenfolgen standardmäßig die Werte 42 und 128 verwendet.

Die erstellten Diagramme zeigen die Verarbeitung einer Zahlenfolge im unkomprimierten Zustand und im Vergleich dazu auch in ihren komprimierten Zuständen. Dafür wurden die in dieser Arbeit verwendeten Kompressionsalgorithmen SIMD-FastPFOR, Run-Length-Encoding und Dictionary-Encoding verwendet.

Name	Definition
OrderedNumSeries	$1, 2, \dots, n$
AlternatingNumSeries	$42_1, 128_2, \dots, 42_{n-1}, 128_n$
HalfRangeAlternatingNumSeries	$42_1, 42_2, \dots, 42_{n/2}, 128_{n/2+1}, \dots, 128_n$
RandomNumSeries	Zufällige Folge von n Werten (Mersenne-Twister 19937 Generator, 32Bit unsigned int, gleichverteilt) [12]

Tabelle 6.1: Verwendete synthetische Zahlenfolgen

Die Evaluation wurde auf einem 64-Bit Linux-System mit einer i7-4600U CPU (4 Kerne, davon 2 virtuell, 4MB Cache, 2.1GHz Grundtakt, max. 3.3GHz) und 8GB RAM ausgeführt.

In Abschnitt 6.1 werden Messungen zu den erreichten Kompressionsgrößen gezeigt. Ebenfalls werden Ergebnisse zu den erreichten Kompressionsgrößen mit Kompressionsketten (Abschnitt 6.1.1) und die adaptive Auswahl eines Kompressionsalgorithmus evaluiert (Abschnitt 6.1.2). Die Messungen der Ausführungszeiten für die Kompression und Dekompression wird in Abschnitt 6.2 gezeigt. Die Messungen der Ausführungszeiten für den Page-Packing-Algorithmus und für die Kompressionsoperatoren werden in den Abschnitten 6.3 und 6.4 vorgestellt.

6.1 KOMPRESSIONSRATEN

Die Diagramme in Abbildung 6.1 vergleichen die Größe der Columns für verschiedene Zahlenfolgen mit 948 bis 30825 Werten im unkomprimierten Zustand und nach Anwendung der Kompressionsalgorithmen. Die Anzahl an Werten entspricht der Kapazität einer Page. Das bedeutet, dass sich in der ersten Page 948 Werte befinden, während in der letzten Page 30825 Werte sind.

Kompressionsalgorithmen zeigen je nach Eigenschaften der gegebenen Daten eine Verbesserung der Kompressionsgrößen. Ununterbrochene Läufe eines Wertes sind prädestiniert für Run-Length-Encoding. Die Zahlenfolge *HalfRangeAlternatingNumSeries* besitzt diese Eigenschaft. Während sich die Kompressionsalgorithmen SIMD-FastPFOR mit 83% bis 25% und Dictionary-Encoding mit 25% in ihren Kompressionsraten mit wachsender Wertanzahl angleichen, zeigt Run-Length-Encoding mit 2.7% bis 0.1% eine signifikante Verbesserung. Da die gegebene Zahlenfolge aus zwei Läufen besteht, kodiert Run-Length-Encoding diese in zwei Wertpaare. Da alle anderen Zahlenfolgen keine Wertläufe haben, erreicht Run-Length-Encoding keine Verbesserung durch die Kompression. Die Pages werden in ihrem Originalzustand belassen.

Alternierende Werte (*AlternatingNumSeries*) lassen sich mit Dictionary-Encoding mit einer Kompressionsrate von 25% am besten komprimieren. Das Dictionary muss in diesem Fall mit nur zwei Werten abgespeichert werden. Da die verwendeten Werte (42, 128) sehr klein sind, zeigt SIMD-FastPFOR eine Kompressionsrate von 83% bis 27%.

Die Entropien der Zahlenfolgen *OrderedNumSeries* und *RandomNumSeries* verhindern eine effektive Komprimierung. Lediglich SIMD-FastPFOR komprimiert die ersten Werte der Zahlenfolge *OrderedNumSeries*, da diese ausreichend klein sind und erreicht eine Rate von 92% bis 50%. Für zufällige Werte erreicht SIMD-FastPFOR für die ersten beiden Pages keine Verbesserung, ab der dritten Page wird eine Kompressionsrate von 99% bis 98% erreicht.

6.1.1 Kompressionskette

Kompressionsalgorithmen fügen Meta-Informationen für die Dekompression hinzu. Die entstandenen komprimierten Daten bieten Eigenschaften, die durch weitere Kompressionsalgorithmen ausgenutzt werden können (siehe Abschnitt 4.3). Dadurch lässt sich eine Verbesserung der Kompressionsraten erreichen. Nachfolgend soll das für die Tripel-Kompressionsketten bestehend aus SIMD-FastPFOR, (kohärentes) Run-Length-Encoding und Dictionary-Encoding evaluiert werden.

Die beste Kompressionsrate für die Zahlfolge AlternatingNumSeries wurde mit dem Tripel Dictionary-Encoding → kohärentes Run-Length-Encoding → SIMD-FastPFOR mit 2%-0.05% erreicht. Für die Zahlenfolge HalfRangeAlternatingNumSeries wurde mit dem Tripel kohärentes Run-Length-Encoding → SIMD-FastPFOR → Dictionary-Encoding die beste Kompressionsrate mit 1.2%-0.05% erreicht. Die Folge OrderedNumSeries ließ sich nur mit SIMD-FastPFOR komprimieren. Es wurde eine Kompressionsrate von 92%-50% erreicht. Die Folge RandomNumSeries weist generell eine höhere Entropie als OrderedNumSeries auf und wurde daher nicht betrachtet.

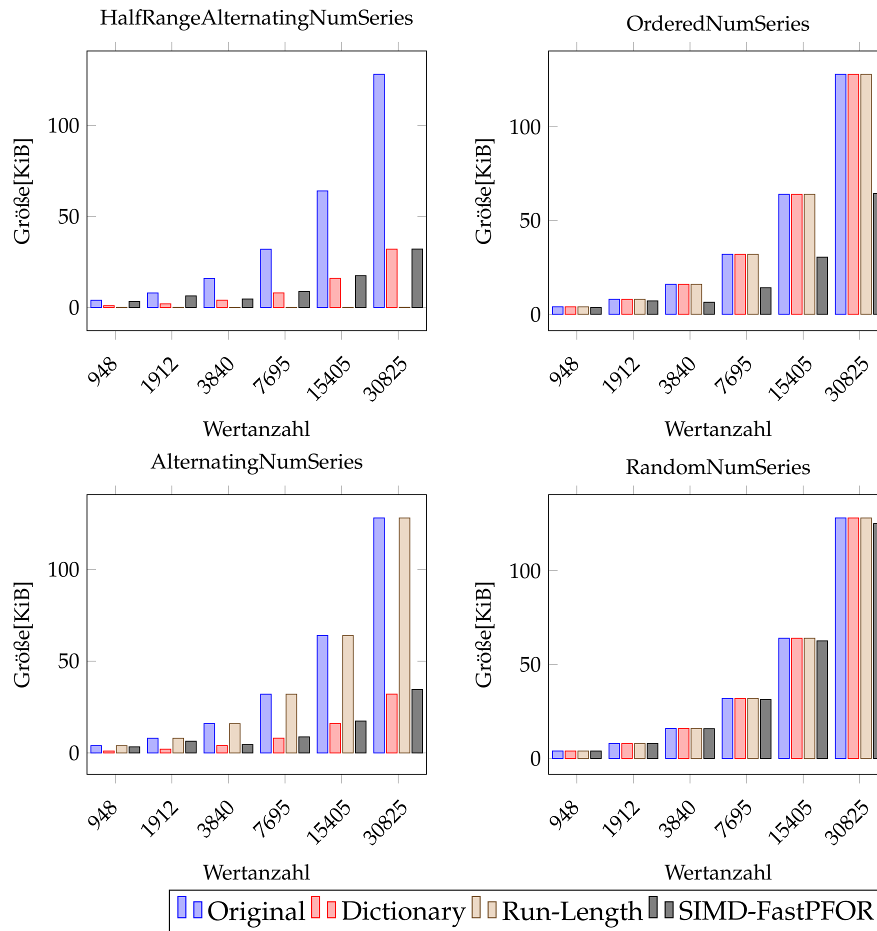


Abbildung 6.1: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen mit 948 bis 30825 4Byte-Werten

SIMD-FastPFOR, Run-Length-Encoding, Dictionary-Encoding

SIMD-FastPFOR entfernt von den zu kodierenden Werten einen Teil, der größer ist als die durchschnittliche Bitlänge der Werte eines Blocks (siehe Sektion 3.3). Da die Möglichkeiten an Bitvariationen aufgrund kleinerer Bitlängen verringert werden, können dadurch innerhalb der Blöcke gleiche aufeinander folgende Werte entstehen, die sich wiederum mit Run-Length-Encoding weiter komprimieren lassen. Da in dieser Arbeit ein Run-Length-Encoding Algorithmus für die Kodierung von 4Byte Werten verwendet wurde, müssen in den Blöcken von SIMD-FastPFOR gleiche aufeinander folgende 4Byte Werte existieren. Eine Anpassung des Run-Length-Encoding Algorithmus an andere Wertgrößen kann die Kompressionsrate nach einer SIMD-FastPFOR Kompression erhöhen.

Während mit alternierenden Werten keine Verbesserung zwischen SIMD-FastPFOR und Run-Length-Encoding auftreten, ergeben sich bei vorhandenen Läufen (siehe Diagramm *HalfRange-AlternatingNumSeries* in Abbildung 6.2) Verbesserungen der Kompressionsgrößen. Für alternierende Werte erreicht SIMD-FastPFOR Kompressionsraten von 83% bis 27%. Für die Zahlenfolge HalfRangeAlternatingNumSeries erreicht SIMD-FastPFOR Kompressionsraten zwischen 83% bis 25%. Run-Length-Encoding reduziert die erhaltenen Daten von SIMD-FastPFOR auf 43% bis

1.7%. Die darauffolgende Kompressionsrate von Dictionary-Encoding ist umso besser, je schlechter die Kompressionsrate von Run-Length-Encoding war. Während Run-Length-Encoding eine Kompressionsrate von 43% erreichte, konnte Dictionary-Encoding diese Daten nochmals auf 31% reduzieren. Wurden hingegen 1.7% erreicht, schaffte Dictionary-Encoding eine Rate von 74%. Die endgültige Kompressionsrate lag bei 11.7% bis 4.8%. Für die Zahlenfolge OrderedNumSeries wurde nur mit SIMD-FastPFOR eine Verbesserung von 92% bis 50% erreicht.

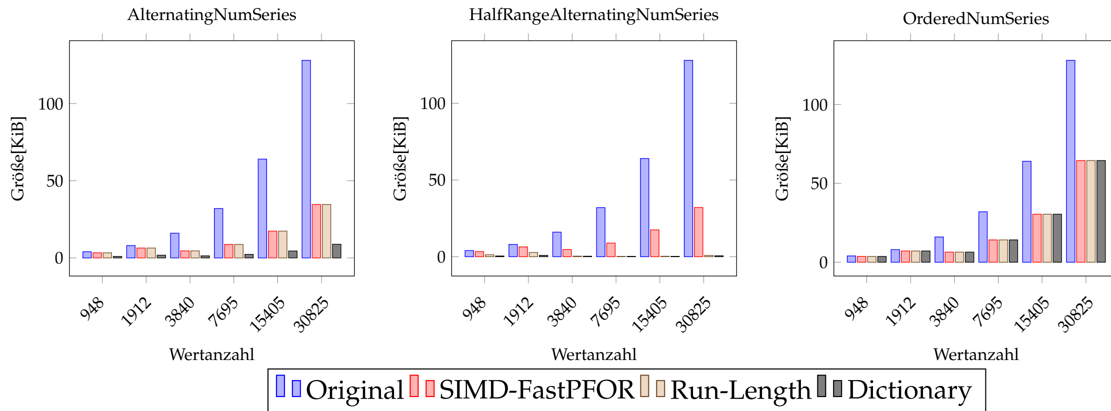


Abbildung 6.2: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (SIMD-FastPFOR→Run-Length-Encoding→Dictionary-Encoding) auf die ggf. bereits komprimierten Daten

(kohärentes) Run-Length-Encoding, SIMD-FastPFOR, Dictionary-Encoding

Die Ergebnisse der Evaluation der Kompressionskette kohärentes Run-Length-Encoding, SIMD-FastPFOR, Dictionary-Encoding sind in Abbildung 6.3 dargestellt.

Unter der Annahme, dass die zu kodierenden Werte möglichst klein sind oder die Läufe möglichst kurz sind, lässt sich kohärentes Run-Length-Encoding (siehe Abschnitt 3.3) mit anschließender SIMD-FastPFOR-Kompression verbessern. Während Run-Length-Encoding für alternierende Werte keine Kompression durchführt, erbringen stattdessen die anschließenden Algorithmen SIMD-FastPFOR mit 83% bis 27% und Dictionary-Encoding mit 29% bis 26% eine signifikante Verbesserung der Kompressionsgröße. Die endgültige Kompressionsrate beträgt 24.1% bis 6.9%.

Mit der Zahlenfolge HalfRangeAlternatingNumSeries verbessert SIMD-FastPFOR das Ergebnis von Run-Length-Encoding (2.7% bis 0.1%) nochmals auf 38% bis 40%. Die anschließende Verarbeitung mit Dictionary-Encoding zeigt eine weitere Verbesserung auf 75% bis 79%. Die endgültige Kompressionsgröße liegt bei 1.2% bis 0.05%.

Wie bereits in den vorherigen Evaluationsabschnitten gezeigt, erreicht Run-Length-Encoding bei geordneten Zahlenfolgen aufgrund fehlender Läufe keine Verbesserung. Lediglich die anschließende Kompression mit SIMD-FastPFOR zeigt eine Verringerung der Speichergröße der Daten auf 92% bis 50%.

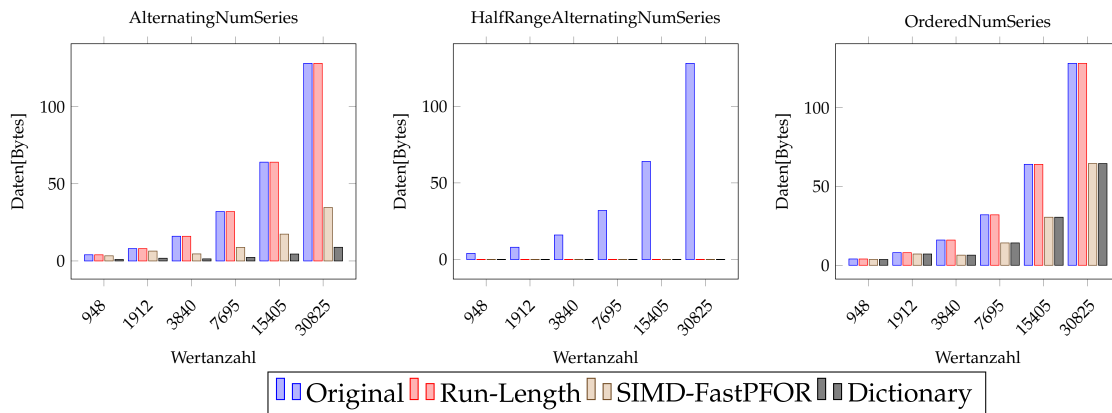


Abbildung 6.3: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (kohärentes Run-Length-Encoding→SIMD-FastPFOR→Dictionary-Encoding) auf die ggf. bereits komprimierten Daten

SIMD-FastPFOR, Dictionary-Encoding, Run-Length-Encoding

Die Ergebnisse der Evaluation der Kompressionskette SIMD-FastPFOR, Dictionary-Encoding, Run-Length-Encoding sind in Abbildung 6.4 dargestellt.

Für die Zahlenfolge AlternatingNumSeries erreicht SIMD-FastPFOR eine Kompressionsrate von 83% bis 27%. Über Dictionary-Encoding werden die Daten auf 29% bis 26% reduziert. Run-Length-Encoding schafft anschließend eine Kompression auf 98% bis 8.4%. Die endgültige Kompressionsrate beträgt 24.1% bis 8.4%.

Für die Zahlenfolge HalfRangeAlternatingNumSeries erreicht SIMD-FastPFOR eine Kompressionsrate von 83% bis 25%. Über Dictionary-Encoding werden die Daten auf 28% bis 26% reduziert. Run-Length-Encoding schafft anschließend eine Kompression auf 98% bis 13%. Die endgültige Kompressionsrate beträgt 23.7% bis 0.6%.

Für die Zahlenfolge OrderedNumSeries erreicht nur SIMD-FastPFOR eine Kompressionsrate von 92% bis 50%.

Run-Length-Encoding, Dictionary-Encoding, SIMD-FastPFOR

Die Ergebnisse der Evaluation der Kompressionskette Run-Length-Encoding, Dictionary-Encoding, SIMD-FastPFOR sind in Abbildung 6.5 dargestellt.

Für die Zahlenfolge AlternatingNumSeries wurde mit Run-Length-Encoding keine Kompression erreicht. Über Dictionary-Encoding werden die Daten auf 26% bis 25% reduziert. SIMD-FastPFOR schafft anschließend eine Kompression auf 86% bis 80%. Die endgültige Kompressionsrate beträgt 22.7% bis 19.9%.

Für die Zahlenfolge HalfRangeAlternatingNumSeries wurde mit Run-Length-Encoding eine Kompressionsrate von 2.7% bis 0.1% erreicht. Über Dictionary-Encoding werden die Daten auf 69%

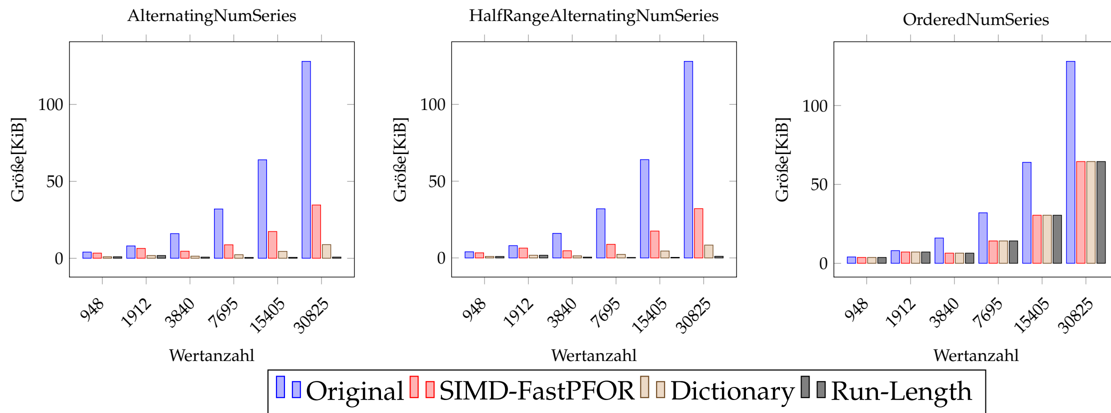


Abbildung 6.4: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (SIMD-FastPFOR→Dictionary-Encoding→Run-Length-Encoding) auf die ggf. bereits komprimierten Daten

bis 59% reduziert. SIMD-FastPFOR schafft anschließend eine Kompression auf 82% bis 57%. Die endgültige Kompressionsrate beträgt 1.5% bis 0.05%.

Für die Zahlenfolge OrderedNumSeries wurde mit Run-Length-Encoding und Dictionary-Encoding keine Verbesserung durch die Kompression erreicht. SIMD-FastPFOR reduziert die Größe der Seiten auf 92% bis 50%.

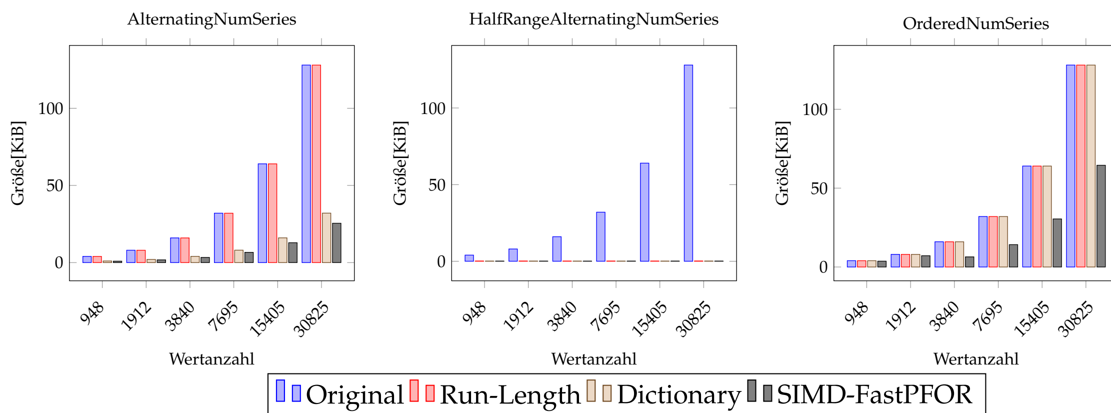


Abbildung 6.5: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (Run-Length-Encoding→Dictionary-Encoding→SIMD-FastPFOR) auf die ggf. bereits komprimierten Daten

Dictionary-Encoding, (kohärentes) Run-Length-Encoding, SIMD-FastPFOR

Die Ergebnisse der Evaluation der Kompressionskette Dictionary-Encoding, kohärentes Run-Length-Encoding, SIMD-FastPFOR sind in Abbildung 6.6 dargestellt.

Für die Zahlenfolge AlternatingNumSeries wurde mit Dictionary-Encoding eine Kompressionsrate von 26% bis 25% erreicht. Über Run-Length-Encoding werden die Daten auf 15% bis 0.4%

reduziert. SIMD-FastPFOR schafft anschließend eine Kompression auf 48% bis 41%. Die endgültige Kompressionsrate beträgt 2% bis 0.05%.

Für die Zahlenfolge HalfRangeAlternatingNumSeries wurde mit Dictionary-Encoding eine Kompressionsrate von 26% bis 25% erreicht. Über Run-Length-Encoding werden die Daten auf 14% bis 0.6% reduziert. SIMD-FastPFOR schafft anschließend eine Kompression auf 52% bis 42%. Die endgültige Kompressionsrate beträgt 1.9% bis 0.08%.

Für die Zahlenfolge OrderedNumSeries wurde mit Run-Length-Encoding und Dictionary-Encoding keine Verbesserung durch die Kompression erreicht. SIMD-FastPFOR reduziert die Größe der Seiten auf 92% bis 50%.

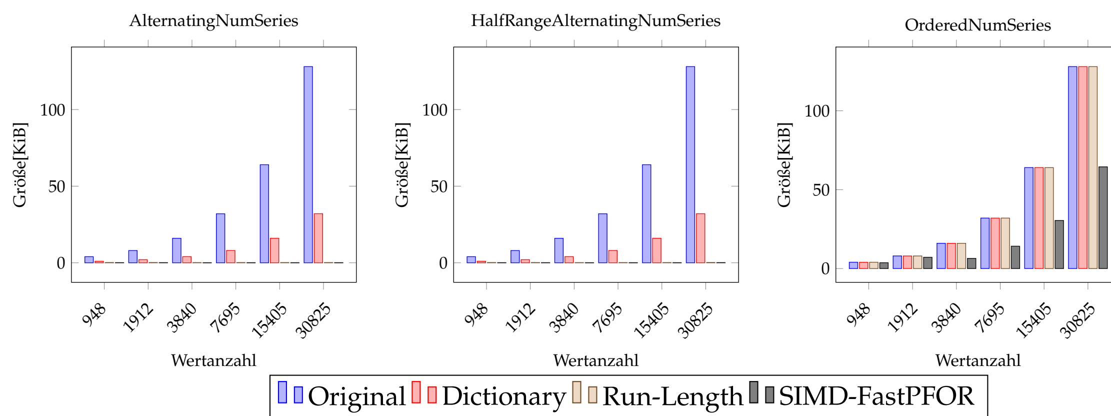


Abbildung 6.6: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (Dictionary-Encoding→kohärentes Run-Length-Encoding→SIMD-FastPFOR) auf die ggf. bereits komprimierten Daten

Dictionary-Encoding, SIMD-FastPFOR, Run-Length-Encoding

Die Ergebnisse der Evaluation der Kompressionskette Dictionary-Encoding, SIMD-FastPFOR, Run-Length-Encoding sind in Abbildung 6.7 dargestellt.

Für die Zahlenfolge AlternatingNumSeries wurde mit Dictionary-Encoding eine Kompressionsrate von 26% bis 25% erreicht. Über SIMD-FastPFOR werden die Daten auf 86% bis 80% reduziert. Run-Length-Encoding schafft anschließend eine Kompression auf 67% bis 51%. Die endgültige Kompressionsrate beträgt 15.4% bis 10.2%.

Für die Zahlenfolge HalfRangeAlternatingNumSeries wurde mit Dictionary-Encoding eine Kompressionsrate von 26% bis 25% erreicht. Über SIMD-FastPFOR werden die Daten auf 86% bis 80% reduziert. Run-Length-Encoding schafft anschließend eine Kompression auf 66% bis 51%. Die endgültige Kompressionsrate beträgt 15.2% bis 10.2%.

Für die Zahlenfolge OrderedNumSeries wurde mit Run-Length-Encoding und Dictionary-Encoding keine Verbesserung durch die Kompression erreicht. SIMD-FastPFOR reduziert die Größe der Seiten auf 92% bis 50%.

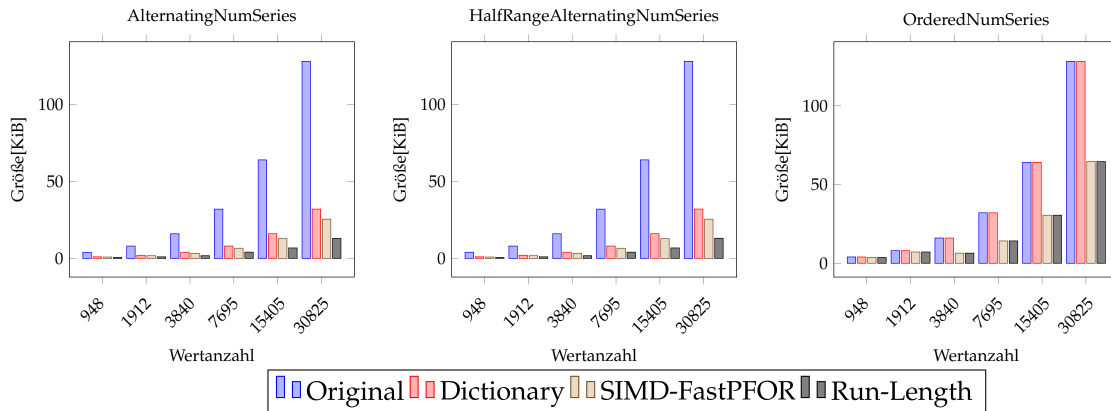


Abbildung 6.7: Vergleich der Kompressionsgrößen zur unkomprimierten Größe mit verschiedenen Zahlenfolgen unterschiedlicher Größe nach Anwendung eines Kompressionsverfahrens der Kompressionskette (Dictionary-Encoding→SIMD-FastPFOR→Run-Length-Encoding) auf die ggf. bereits komprimierten Daten

6.1.2 Auto-Kompression

Die adaptive Wahl eines Kompressionsalgorithmus geschieht auf Grundlage der eingefügten Werte. Anhand der Zahlenfolgen aus Tabelle 6.1 wurde die Auswahl einer Kompression durch die Auto-Kompression (siehe Abschnitt 4.4) untersucht. Die Ergebnisse sind in Tabelle 6.2 zusammengefasst.

Kandidaten & Heuristik-Ergebnisse (i.D.)			**Folge**	**Auswahl**
Null-Suppr.	Run-Length-Enc.	Dict.-Enc.	Random	SIMD-FastPFOR (1. & 2. Page unkomprimiert)
0.98	1.99	1.5		
Null-Suppr.	Run-Length-Enc.	Dict.-Enc.	Ordered	SIMD-FastPFOR
0.43	2.0	1.5		
Null-Suppr.	Run-Length-Enc.	Dict.-Enc.	HalfRangeAlternating	Run-Length-Encoding
0.24	0.0	0.25		
Null-Suppr.	Run-Length-Enc.	Dict.-Enc.	Alternating	SIMD-FastPFOR
0.22	2.0	0.25		
Null-Suppr.	Run-Length-Enc.	Dict.-Enc.	Alternating (30Bit Werte)	Dictionary-Encoding
0.94	2.0	0.25		

Tabelle 6.2: Ergebnisse der Auto-Kompression für unterschiedliche Zahlenfolgen mit 100000 Werten. Heuristik-Ergebnisse der einzelnen Kompressionsalgorithmen sind im Durchschnitt angegeben.

Für eine zufällige Zahlenfolge wurde der Kompressionsalgorithmus SIMD-FastPFOR mit einer erwarteten Kompressionsrate von 0.98 gewählt. Anhand der Ergebnisse aus Abschnitt 6.1 für zufällige Zahlenfolgen (siehe auch Abbildung 6.1) konnte mit SIMD-FastPFOR die Größe der Pages auf durchschnittlich 98,7% reduziert werden. Somit ist SIMD-FastPFOR unter alleiniger Betrachtung der Kompressionsrate die korrekte Wahl. Für die erste und zweite Page mit jeweils 948 und 1912 Werten wurde jedoch keine Verbesserung der Speichergröße durch die Kompression erwartet. Somit blieben diese Pages unkomprimiert. Die Heuristik-Werte für Run-Length-Encoding und Dictionary-Encoding sind jeweils größer als eins und erwarten damit keinen Kompressionserfolg bei ihrer Anwendung.

Für die Folge *OrderedNumSeries* wurde der Kompressionsalgorithmus SIMD-FastPFOR mit einer erwarteten durchschnittlichen Reduzierung der Größe der Pages auf 43% gewählt. Nach den Ergebnissen aus Abschnitt 6.1 und Abbildung 6.1 wurde für die Folge *OrderedNumSeries* eine durchschnittliche Kompressiongröße von 60.2% mit SIMD-FastPFOR ermittelt.

Anhand der ermittelten Ergebnisse aus Abschnitt 6.1 wurde für die Folge *HalfRangeAlternatingNumSeries* eine durchschnittliche Reduzierung der Kompressionsgröße auf 45.4% für SIMD-FastPFOR, 0.01% für Run-Length-Encoding und 25.3% für Dictionary-Encoding ermittelt. Dementsprechend wurde Run-Length-Encoding mit einer erwarteten durchschnittlichen Kompressionsgröße von unter 0.01% korrekt ausgewählt.

Für die Folge *AlternatingNumSeries* wurde aus den Evaluationsergebnissen eine durchschnittliche Kompressionsgröße von 45.5% mit SIMD-FastPFOR, $<$ 0.1% mit Run-Length-Encoding und 25.3% mit Dictionary-Encoding ermittelt. Da SIMD-FastPFOR eine bessere Komprensionsrate erwartet je kleiner die Werte sind (in diesem Fall 128 und 42), wurde SIMD-FastPFOR mit einer erwarteten Kompressionsrate von 22% ausgewählt. Dictionary-Encoding wäre mit einer erwarteten Kompressionsrate von 25% die korrekte Wahl gewesen. Werden die Werte der Folge auf 30Bit vergrößert, erwartet SIMD-FastPFOR nur noch eine Kompressionsrate von 94%, während der gewählte Kompressionsalgorithmus Dictionary-Encoding weiterhin 25% erwartet.

6.2 KOMPRESSIONSPERFORMANCE

Performante Kompressionsalgorithmen mit geringen Ausführungszeiten steigern die Effizienz eines Systems. In diesem Abschnitt soll daher untersucht werden, inwiefern sich die Kompression auf die Ausführungszeiten für das Einfügen und Auslesen von Daten im Vergleich zur unkomprimierten Variante auswirkt.

Die Messung der Ausführungszeiten bezieht sich auf das Einfügen und Auslesen der Daten. Die benötigten Zeiten sind unter Verwendung der Kompressionsalgorithmen höher, da beim Einfügen der Daten die Kompression und beim Auslesen die Dekompression ausgeführt wird. Die benötigte Ausführungszeit für die Einfüge-Operation unter Verwendung von 4 AEUs ist im Durchschnitt um 33.7% höher als bei Verwendung von nur einer AEU.

Mit Ausnahme von Dictionary-Encoding, mit 87ms für 1 Million Werte, zeigen die Ausführungszeiten für alternierende Werte keine großen Unterschiede (siehe Abbildung 6.8) beim Scan an. Da Run-Length-Encoding bei alternierenden Werten die Daten unkomprimiert belässt, sind die Scan-Zeiten für Run-Length-Encoding und im Fall ohne Kompressionsalgorithmen mit rund 35ms für 1 Mio. Werte nahezu gleich. Die Dekomprimierung der Werte mit Null-Suppression ist mit 39ms annähernd so schnell wie im nicht komprimierten Fall. Dictionary-Encoding erreicht eine etwas bessere Kompressionsrate als SIMD-FastPFOR (siehe Abschnitt 6.1), benötigt jedoch eine wesentlich längere Zeit für die Dekompression. Demnach ist in diesem Fall SIMD-FastPFOR den anderen Kompressionsalgorithmen vorzuziehen.

Eine Besonderheit ist bei der Verwendung von vier AEUs zu erkennen (siehe Abb. 6.9). Run-Length-Encoding zeigt wesentlich höhere Ausführungszeiten mit 134ms für 1 Mio. Werte als im vorherigen Fall mit nur einer AEU. Der Grund hierfür findet sich im Load-Balancer von ERIS,

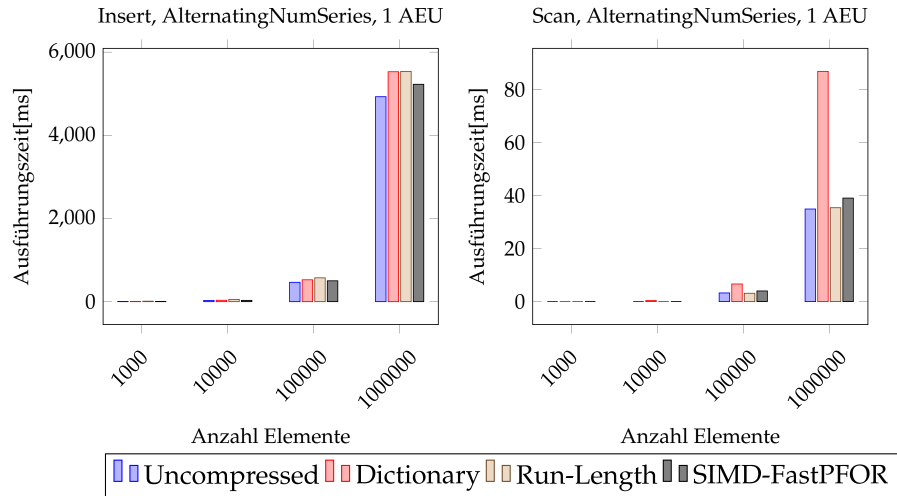


Abbildung 6.8: Insert/Scan Ausführungszeiten (1 AEU)

der Daten beim Einfügen Round-Robin auf die vorhandenen AEUs verteilt. Bei Zahlenfolgen mit alternierenden Werten entsteht dadurch eine implizite Sortierung der Werte, womit für Run-Length-Encoding eine Grundlage zur Kompression gegeben ist. Die Ausführungszeit des Scans im unkomprimierten Fall beträgt 22ms für 1 Million Werte. Mit Dictionary-Encoding werden 69ms und mit SIMD-FastPFOR 28ms für 1 Mio. Werte erreicht.

Die Ausführungszeiten für das Einfügen von Daten sind um durchschnittlich 33.7% höher als im Fall mit einer AEU. Dagegen fallen die Scan-Zeiten geringer aus, da jede AEU eine kleinere Datenmenge verarbeiten muss.

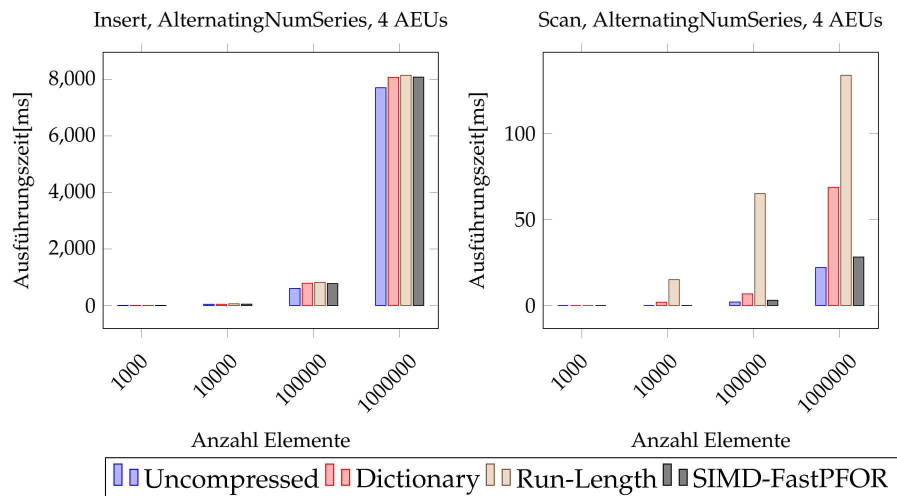


Abbildung 6.9: Insert/Scan Ausführungszeiten (4 AEUs)

Die Ausführungszeiten für das Einfügen und Auslesen der Zahlenfolge *HalfRangeAlternatingNum-Series* mit einer AEU ist in Abbildung 6.10 und mit vier AEUs in Abbildung 6.11 dargestellt. Die benötigten Zeiten für das Einfügen weisen einen Unterschied zwischen beiden Fällen auf (1 AEU: durchschnittlich 5.3s, 4 AEUs: durchschnittlich 8.2s, 1 Mio. Werte).

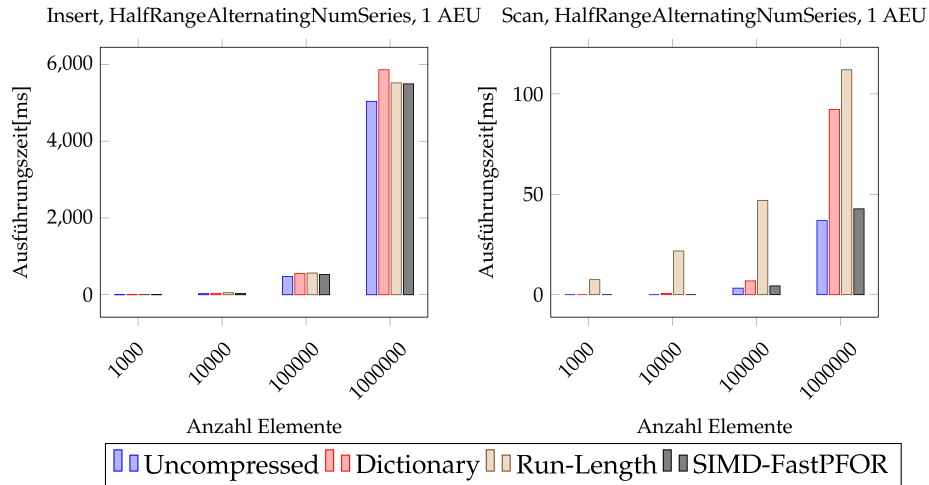


Abbildung 6.10: Insert/Scan Ausführungszeiten (1 AEU)

Da die besten Kompressionsraten mit dieser Zahlenfolge mit Run-Length-Encoding erreicht werden, spiegelt sich dies auch in den Ausführungszeiten für das Auslesen der Daten unter Verwendung mit einer AEU (109.7ms, 1 Mio. Werte) und vier AEUs (138ms, 1 Mio. Werte) wieder. Die Ausführungszeit für das Auslesen mit Run-Length-Encoding dauert im Vergleich zu den anderen Kompressionsalgorithmen am längsten. Während mit SIMD-FastPFOR und Dictionary-Encoding annähernd gleiche Kompressionsraten erreicht werden, ist die Ausführungszeit zum Auslesen mit einer AEU (39ms, 1 Mio. Werte) und vier AEUs (26ms, 1 Mio. Werte) mit SIMD-FastPFOR um 55.6% und 70.1% geringer als mit Dictionary-Encoding (1 AEU: 88ms, 4 AEUs: 87ms, 1 Mio. Werte). Dementsprechend ist die Verwendung von SIMD-FastPFOR den Kompressionsalgorithmen Run-Length-Encoding und Dictionary-Encoding vorzuziehen.

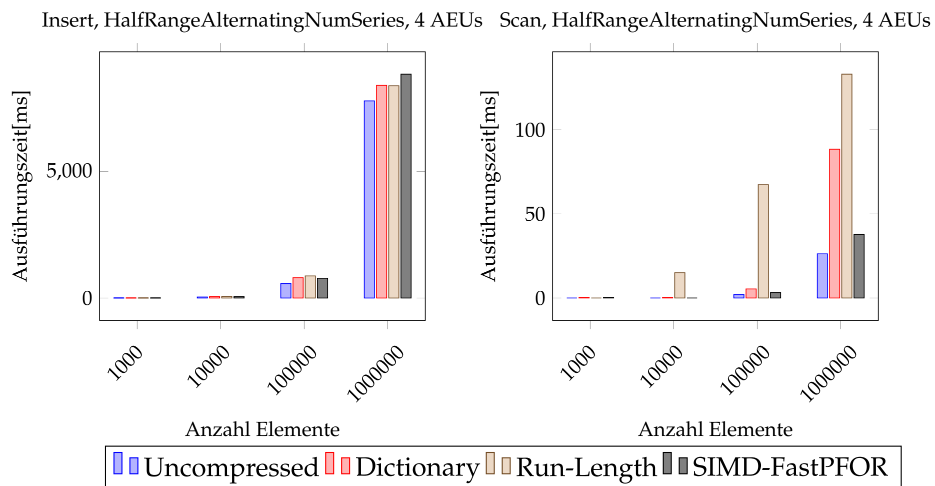


Abbildung 6.11: Insert/Scan Ausführungszeiten (4 AEUs)

6.3 PAGE-PACKING PERFORMANCE

In Abbildung 6.12 werden die Ausführungszeiten des Page-Packing Algorithmus für das Einfügen und Auslesen von Werten gezeigt. Es wurde die Zahlenfolge HalfRangeAlternatingNumSeries mit 1 AEU und 4 AEUs verwendet. Die Diagramme zeigen zum Vergleich die Ausführungszeiten im unkomprimierten Fall und mit den Kompressionsalgorithmen ohne aktiviertem Page-Packing-Algorithmus.

Die Größe der Tabellen (siehe Abschnitt 4.2) wurde auf jeweils 4 Einträge begrenzt. Da Pages, die deallokiert wurden, ihren Speicherplatz dem System wieder zur Verfügung stellen, müssen für weitere Pages weniger Speichersegmente neu allokiert werden. Durch die Wiederverwendung bereits allokierten Speichers, können sich die Ausführungszeiten für das Einfügen von Werten mit Page-Packing verbessern.

Die Ausführungszeiten für das Einfügen mit 4 AEUs für Dictionary-Encoding sind mit aktiviertem Page-Packing um durchschnittlich 0.8% schlechter, als ohne Page-Packing. Für Run-Length-Encoding mit Page-Packing ergab sich eine durchschnittliche Verbesserung um 1.4%. Für SIMD-FastPFOR mit Page-Packing wurden 9.4% bessere Ausführungszeiten erreicht als ohne Page-Packing. Mit 1 AEU ergab sich mit SIMD-FastPFOR mit Page-Packing eine durchschnittliche Verbesserung um 5.3%. Run-Length-Encoding ohne Page-Packing verschlechterte sich im Durchschnitt mit 1 AEU um 2.9%. Dictionary-Encoding mit Page-Packing verbesserte sich durchschnittlich um 1.2%.

Die Ausführungszeiten für das Auslesen von Werten können sich mit Page-Packing verbessern, wenn die Speicherzugriffe auf benötigte Pages in einem begrenzten Speicherbereich stattfinden. Auf den Speicher muss somit in geringerem Maße auf entfernt liegende Speicheradressen zugegriffen werden. Wurden die Pages jedoch nicht in ihrer Reihenfolge oder in entfernt liegende freie Speicherbereiche verschoben, können sich längere Ausführungszeiten ergeben. Die Ausführungszeiten für das Auslesen der Werte verbesserte sich insbesondere mit 4 AEUs für Run-Length-Encoding mit Page-Packing für 100000 Werte. Da Run-Length-Encoding die verwendete Zahlenfolge *HalfRangeAlternatingNumSeries* auf sehr geringe Größen komprimiert, sind die resultierenden Pages prädestiniert für die Verwendung mit dem Page-Packing-Algorithmus. Es wurde eine Verbesserung um 12.2% erreicht. Für 1 Million Werte ergab sich eine Verschlechterung um 3.6%. Mit SIMD-FastPFOR mit Page-Packing zeigte sich eine durchschnittliche Verbesserung um 18.7%. Dictionary-Encoding mit Page-Packing verschlechterte sich um durchschnittlich 9.8%. Ähnliche Werte mit geringeren Abständen zeigten sich unter der Verwendung von einer AEU.

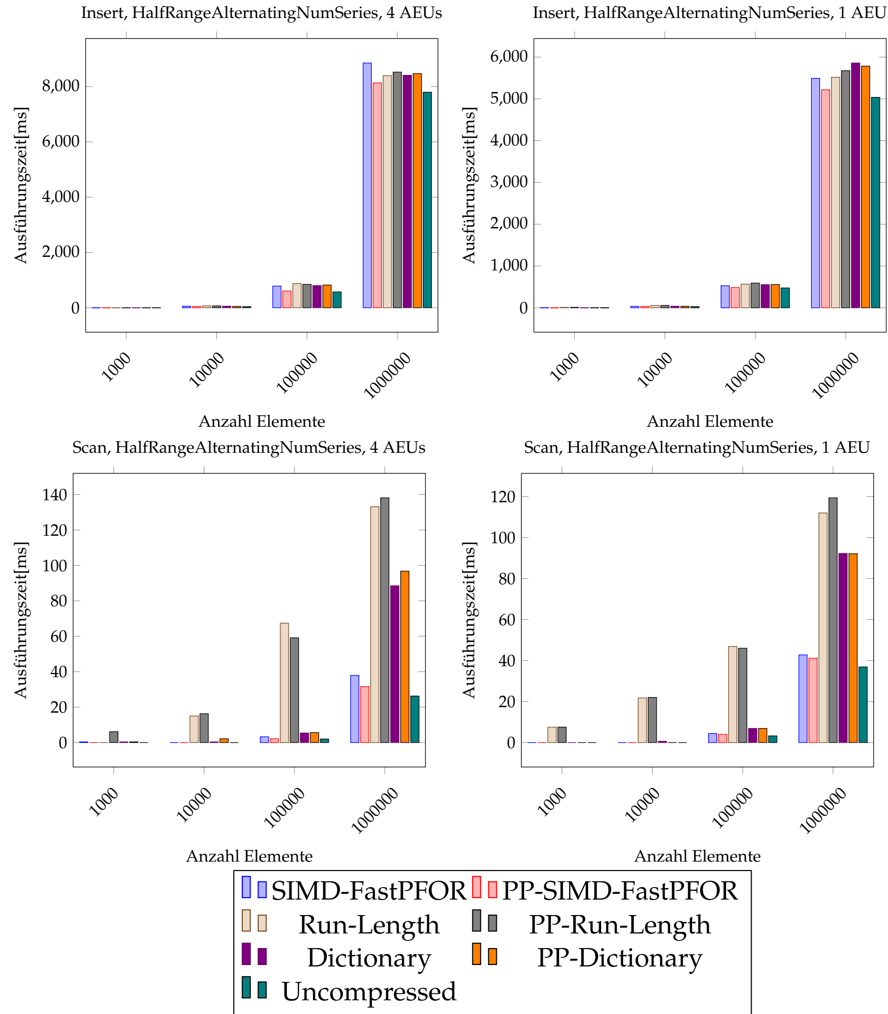


Abbildung 6.12: Insert/Scan Ausführungszeiten mit aktiviertem Page-Packing für die Zahlenfolge *HalfRangeAlternatingNumSeries* mit 1 AEU und 4 AEUs

6.4 KOMPRESSIONSOPERATORPERFORMANCE

Die Evaluation der Kompressionsoperatoren erfolgte für den Ausdruck $(x > 100) + 42$ und für die Aggregationsoperatoren *count* und *sum*. Die Diagramme vergleichen die Ausführungszeiten der Operatoren zwischen den verschiedenen Kompressionsalgorithmen und dem unkomprimierten Fall für die Zahlenfolgen *AlternatingNumSeries*, *HalfRangeAlternatingNumSeries* und *OrderedNumSeries*. Es wurden vier AEUs verwendet.

6.4.1 Ausführungszeiten Kompressionsoperatoren

Die Diagramme in Abbildung 6.13 zeigen die Ausführungszeiten für den Operationsausdruck $(x > 100) + 42$. Die Operation wählt alle Werte größer 100 aus und addiert sie mit der Zahl 42. Die Kompressionsoperatoren auf Basis von SIMD-FastPFOR benötigen für alle Zahlenfolgen die mit Abstand höchste Ausführungszeit (Alternating: 209ms, HalfRangeAlternating: 189ms, Orde-

red: 288ms, 1 Mio. Werte). Grund hierfür ist die Komplexität der Meta-Daten. Während bei Run-Length-Encoding die zu kodierenden Werte durch Wert-Run-Paare ersetzt werden und somit ein Zugriff auf entfernte Meta-Daten nicht notwendig ist, müssen für Null-Suppresion pro Block zunächst die Blockinformationen ausgelesen werden (siehe Abschnitt 3.3). Über das Attribut *maxb* wird entschieden, ob der momentane Block übersprungen werden kann oder die Operation auf den enthaltenen Werten ausgeführt werden muss (siehe Abschnitt 5.2).

Dictionary-Encoding benötigt pro Page ebenfalls Zugriff auf das Dictionary, um daraus das Ergebnis-Wörterbuch zu erstellen (siehe Abschnitt 5.4). Dennoch müssen im Gegensatz zu SIMD-FastPFOR lediglich einmalig pro Page die Meta-Daten ausgelesen werden. Dies zeigt sich ebenfalls in den geringeren Ausführungszeiten der Kompressionsoperatoren auf Basis von Dictionary-Encoding (Alternating: 38ms, HalfRangeAlternating: 51.5ms, Ordered: 26ms, 1 Mio. Werte).

Die Kompressionsoperatoren von Run-Length-Encoding weisen die geringsten Ausführungszeiten auf (Alternating: 25.5ms, HalfRangeAlternating: 17ms, Ordered: 26ms, 1 Mio. Werte). Da die Werte unkomprimiert in den Wert-Run-Paaren stehen, ist eine Vervielfältigung des Ergebnisses der Operation um die Run-Length ausreichend (siehe Abschnitt 5.3).

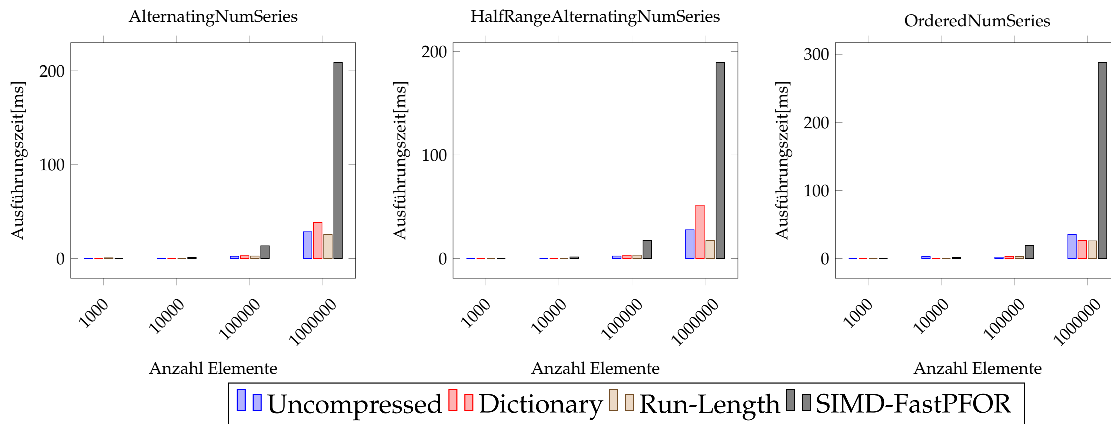


Abbildung 6.13: Ausführungszeiten für die Anwendung der Operation $(x > 100) + 42$ mit vier AEUs

Abbildung 6.14 zeigt die Ausführungszeiten der Kompressionsoperatoren im Vergleich zur konventionellen Verarbeitung der Operation $(x > 100) + 42$. Die jeweils letzte Säule im Diagramm zeigt zum Vergleich die Ausführungszeit im unkomprimierten Fall an (Alternating: 28.5ms, HalfRangeAlternating: 27.8ms, Ordered: 35ms, 1 Mio. Werte).

Die konventionellen Operatoren dekomprimieren die Pages vor der Ausführung ihrer Operation. Es ist zu erkennen, dass die Verarbeitung mit den Kompressionsoperatoren von Run-Length-Encoding (Alternating: 25.5ms, HalfRangeAlternating: 17ms, Ordered: 26ms, 1 Mio. Werte) und Dictionary-Encoding (Alternating: 38ms, HalfRangeAlternating: 51.5ms, Ordered: 26ms, 1 Mio. Werte) in einer kürzeren Zeit stattfindet als mit der konventionellen Variante. Mit der Dekompression mit Run-Length-Encoding werden für jeweils 1 Mio. Werte für die Zahlenfolge AlternatingNumSeries 141ms, für HalfRangeAlternatingNumSeries 135ms und für OrderedNumSeries 26ms benötigt. Mit Dictionary-Encoding werden für 1 Mio. Werte für AlternatingNumSeries 83ms, HalfRangeAlternatingNumSeries 89ms und OrderedNumSeries 35ms benötigt. Lediglich mit SIMD-FastPFOR benötigt die konventionelle Variante (Alternating: 40ms, HalfRangeAlternating: 28.8ms, Ordered: 29.8ms, 1 Mio. Werte) eine deutlich kürzere Ausführungszeit als mit der

Anwendung von Kompressionsoperatoren (Alternating: 209ms, HalfRangeAlternating: 189ms, Ordered: 288ms, 1 Mio. Werte).

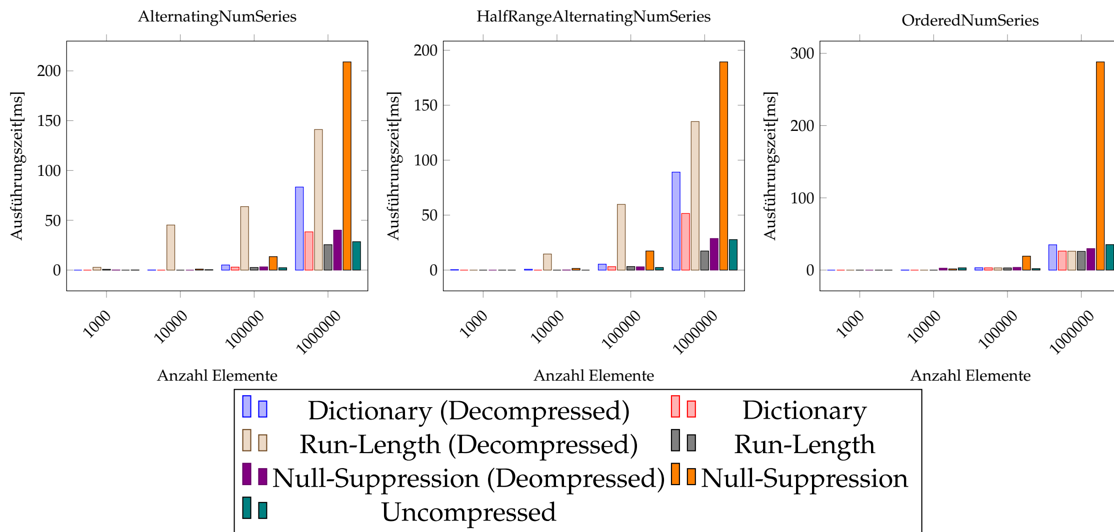


Abbildung 6.14: Vergleich der Ausführungszeiten für die Anwendung der Operation $(x > 100) + 42$ mit konventionellen Operatoren und Kompressionsoperatoren mit vier AEUs

6.4.2 Ausführungszeiten Aggregationsoperatoren

In Abbildung 6.15 sind die Ausführungzeiten für den *count*-Operator der Kompressionsalgorithmen im Vergleich zu einem improvisierten *count*-Operator im unkomprimierten Fall dargestellt. Es wurde die Zahlenfolge *HalfRangeAlternatingNumSeries* verwendet. Da über den Scan-Operator bisher kein Zugriff auf den Header der Pages erlangt werden kann, wurde der *count*-Operator durch Akkumulation der zurückgegebenen Anzahl an Werten improvisiert. Standardmäßig werden der Scan-Funktion 1024 Werte übergeben, bis alle Werte ausgelesen wurden. Es ist anzunehmen, dass über den direkten Zugriff auf das *elements*-Attribut des Headers eine deutlich kürzere Ausführungszeit zu erreichen ist als über den *count*-Operator der Kompressionsalgorithmen. Die Evaluation fand jeweils unter der Verwendung von einer AEU und vier AEUs statt.

Sofern keine *expression* übergeben wird, liefert der *count*-Operator die Gesamtzahl an Elementen zurück. Dafür wird das *elements*-Attribut des Headers ausgelesen (siehe Abschnitt 5.5.1). Auf einem 64-Bit System hat das *elements*-Attribut eine Größe von 8Byte. Die Kompressionsalgorithmen kodieren Werte der Größe 4Byte. Demnach muss das *elements*-Attribut aus zwei kodierten Werten zusammengesetzt werden. Während es mit einer AEU kaum Unterschiede in den Ausführungszeiten bei den Kompressionsalgorithmen gibt, zeigen sich bei der Benutzung von vier AEUs geringfügige Unterschiede. SIMD-FastPFOR benötigt für das Auslesen aufgrund des Zugriffs auf die Meta-Daten die längste Zeit.

Je nachdem wie die Werte vor und nach dem *elements*-Attribut beschaffen sind, muss Run-Length-Encoding den gesuchten Wert aus einem oder zwei Wert-Run-Paaren auslesen. Dafür muss das erste Attribut mit einer Größe von 8Byte zunächst übersprungen werden, um anschließend die folgenden 8Byte auslesen zu können. Im Gegensatz dazu bietet Dictionary-Encoding die Möglichkeit das *elements*-Attribut direkt auszulesen. Da jeder Index-Wert einem unkomprimierten

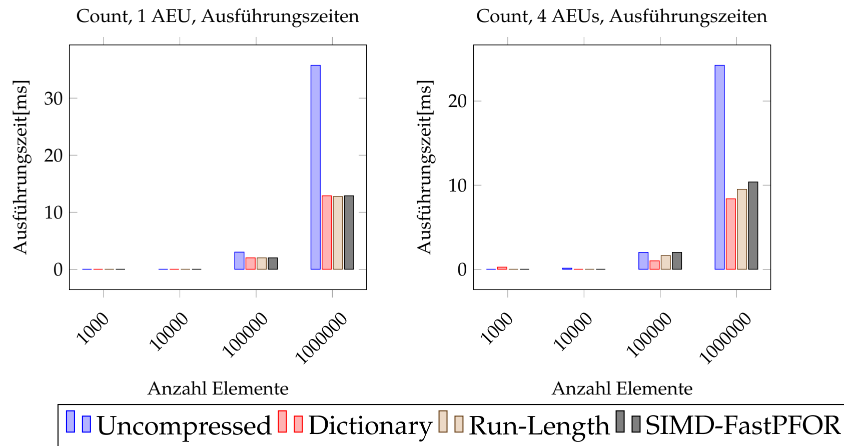


Abbildung 6.15: Ausführungszeiten des Aggregationsoperators *count* für 1 AEU und 4 AEUs (*HalfRangeAlternatingNumSeries*)

4Byte-Wert entspricht, lässt sich das *elements*-Attribut über den 3. und 4. Index-Wert auslesen. Der *count*-Operator des Dictionary-Encoding hat dadurch die geringste Ausführungszeit unter Verwendung von vier AEUs (Dict: 8.4ms, RLE: 9.5ms, FastPFOR: 10.4ms, Unkomprimiert: 24.25ms, 1 Mio. Werte, 4 AEUs).

In Abbildung 6.16 sind die Ausführungszeiten des *sum*-Operators für den unkomprimierten Fall im Vergleich zu den Kompressionsalgorithmen für eine AEU und vier AEUs dargestellt. Es wurde die Zahlenfolge *HalfRangeAlternatingNumSeries* verwendet. Die Nutzung von vier AEUs verbessert die Asuführungszeit des *sum*-Operators generell (Unkomprimiert: 38.8ms, FastPFOR: 61.8ms, RLE: 13.1ms, Dict: 40ms, 1 Mio. Werte). Während bei den Kompressionsoperatoren von Dictionary-Encoding und SIMD-FastPFOR ein Zugriff auf die Meta-Daten für die Akkumulation notwendig ist, wird bei Run-Length-Encoding dem *sum*-Operator das Wert-Run-Paar übergeben. Der *sum*-Operator akkumuliert das Ergebnis aus der Multiplikation des Wertes mit seiner jeweiligen gegebenen Anzahl. Dadurch erreicht der *sum*-Operator von Run-Length-Encoding die geringsten Ausführungszeiten mit 13.1ms für 1 Million Werte.

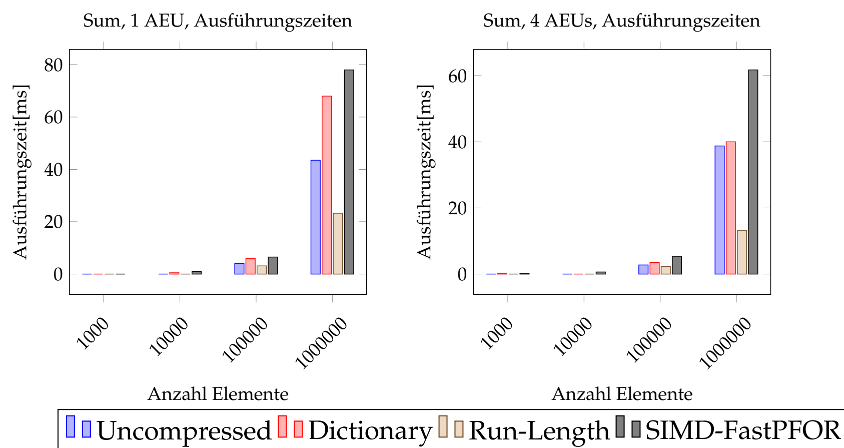


Abbildung 6.16: Ausführungszeiten des Aggregationsoperators *count* für 1 AEU und 4 AEUs (*HalfRangeAlternatingNumSeries*)

7 FAZIT

In dieser Arbeit wurde das Datenbanksystem ERIS um eine spaltenorientierte Speicherarchitektur und um die leichtgewichtigen Kompressionsalgorithmen SIMD-FastPFOR, Run-Length-Encoding und Dictionary-Encoding erweitert. Für die Anwendung der Kompressionsalgorithmen wurde eine allgemeine Schnittstelle entwickelt, die eine Verkettung von Kompressionsalgorithmen erlaubt. Die Schnittstelle ermöglicht es, weitere Kompressionsalgorithmen flexibel in das Datenbanksystem zu integrieren. Um die Speicherausnutzung zu optimieren, wurde ein Page-Packing-Algorithmus implementiert. Dieser kann im Anschluss an eine Kompression ausgeführt werden und verschiebt komprimierte Datenbereiche einer Page in den freien Speicherbereich einer anderen Page. Die ursprüngliche Page wird daraufhin deallokiert. Ebenso wurde mit der Auto-Kompression die adaptive Auswahl von Kompressionsalgorithmen zur Laufzeit ermöglicht.

Das Kompressions-Interface bietet die Möglichkeit, Kompressionsalgorithmen zu verketten. Dadurch kann durch geeignete Wahl der Reihenfolge der Algorithmen eine Verbesserung der Kompressionsgrößen erreicht werden. Empfehlenswert ist die Kombination Run-Length-Encoding (kohärent), SIMD-FastPFOR, Dictionary-Encoding und Dictionary-Encoding, Run-Length-Encoding (kohärent), SIMD-FastPFOR. Die Evaluation zeigte für diese Ketten eine Kompressionsrate für geeignete Werte von bis zu 0.05%.

Die Nutzung von Kompressionsalgorithmen verringert zwar den Speicherverbrauch der Daten. Jedoch stellen sie den frei gewordenen Speicherplatz dem System nicht wieder zur Verfügung. Erst durch die Nutzung des Page-Packing-Algorithmus werden nach der Verschiebung von Pages deren Speicherbereiche wieder dem System zur Verfügung gestellt. Page-Packing sollte daher in Verbindung mit den Kompressionsalgorithmen genutzt werden.

Da eine Dekompression die Ausführungszeit von konventionellen Datenbankoperatoren erhöht, wurden Kompressionsoperatoren für die direkte Ausführung auf komprimierten Daten entwickelt. Bisherige Kompressionsoperatoren beziehen sich überwiegend auf Run-Length-Encoding und Dictionary-Encoding. In dieser Arbeit wurden Kompressionsoperatoren basierend auf den Kompressionsalgorithmen SIMD-FastPFOR und FastPFOR von Lemire et al. (siehe Abschnitt 2.3) entwickelt.

Kompressionsoperatoren auf Basis von Run-Length-Encoding erreichten mitunter die geringsten Ausführungszeiten im Vergleich zur unkomprimierten Variante (Verbesserung um 25%) und zu den anderen Kompressionsalgorithmen, vorausgesetzt die Daten sind für eine Kompression mit Run-Length-Encoding geeignet gewesen (siehe Abschnitt 6.4.1). Insbesondere bei der Anwendung von Aggregationsoperatoren wurden deutliche Verbesserungen der Ausführungszeiten erreicht.

Die Kompressionsoperatoren von Dictonary-Encoding und Run-Length-Encoding weisen deutlich schnellere Ausführungszeiten auf im Vergleich zu einer vorherigen Dekompression mit diesen Algorithmen und der anschließenden Verarbeitung mit konventionellen Operatoren (siehe Abschnitt 6.4, Dictionary-Encoding: Verbesserung um 44,2%, Run-Length-Encoding: Verbesserung um 32%). Lediglich Operatoren auf Basis von SIMD-FastPFOR weisen höhere Ausführungszeiten auf als die konventionelle Variante (85% schneller, als mit SIMD-FastPFOR Kompressionsoperatoren). Dementsprechend sollte SIMD-FastPFOR eingesetzt werden, wenn für Datensätze mit kleinen verschiedenen Werten die Kompressionsrate entscheidend ist. Die Anwendung der Kompressionsoperatoren auf mit SIMD-FastPFOR komprimierten Daten eignet sich im Vergleich zu einer Dekompression und anschließenden konventionellen Verarbeitung nicht.

Sollte die Art der Daten unbekannt sein, ist die adaptive Auswahl von Kompressionsalgorithmen empfehlenswert. Definierte Kompressionsoperatoren unterliegen ebenfalls dieser Adaptivität und müssen nicht weiter angepasst werden.

8 ZUKÜNFTIGE ARBEITEN

Mit Verknüpfungsoperatoren (z. B. &&) werden Ausdrücke in Teilausdrücke unterteilt (siehe Abschnitt 5.5.2). Jeder Teilausdruck wird sequentiell auf seiner jeweils zugeordneten Page ausgeführt. Es ist zu untersuchen, inwieweit sich die Verteilung der Teilausdrücke auf die AEUs und die dortige Verarbeitung auf die Ausführungszeit auswirkt.

Die adaptive Wahl von Kompressionsalgorithmen bezieht momentan nicht die Auswirkungen von Kompressionsalgorithmen ein. Es wurde gezeigt, dass es Algorithmen geben kann (zum Beispiel SIMD-FastPFOR), bei denen eine konventionelle Datenverarbeitung der mit Kompressionsoperatoren vorzuziehen ist. Dementsprechend sollte auch die adaptive Kompressionswahl angepasst werden. Weiterhin ist die Auto-Kompression auch auf Kompressionsketten auszuweiten. Hierbei sollten die einzelnen Ergebnisse der Kompressionsalgorithmen innerhalb der Kette berücksichtigt werden.

LITERATURVERZEICHNIS

[1] Ingres vectorwise. http://www.ingres.com/products/vectorwise.

[2] C-store code release under bsd license. http://db.lcs.mit.edu/projects/cstore/, 2005.

[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[4] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[5] Patrick Damme. Lightweight techniques for compression and transformation. Master's thesis, TU Dresden, 2014.

[6] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10:33–58, 2012.

[7] Miguel C Ferreira. *Compression and query execution within column oriented databases*. PhD thesis, Massachusetts Institute of Technology, 2005.

[8] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. Eris: A numa-aware in-memory storage engine for analytical workloads. *Proceedings of the VLDB Endowment*, 7(14), 2014.

[9] D. Lemire. Fastpfor. https://github.com/lemire/FastPFor.

[10] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[11] Alicja Łuszczak, Marcin Zukowski, Peter Boncz, and Jacopo Urbani. Simple solutions for compressed execution in vectorized database system. 2011.

[12] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[13] MFXJ Oberhumer. The lempel-ziv-oberhumer data compression library.

[14] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[15] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[16] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.

[17] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

[18] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.

ANHANG

	free	pos	originalPage
-	0	0	0
A	670	0x7fcfcba0cd62	0x7fcfcba0c000
B	1626	0x7fcfcba139a6	0x7fcfcba12000
C	11574	0x7fcfcba252ca	0x7fcfcba24000

Tabelle 1: Tabelle sortiert nach freiem Speicherplatz

	used	currentPage	pos	pageSize	parentPage	pagesIn	offset
-	0	0	0	0	0	0	0
B	6566	0x7fcfcba12000	0x7fcfcba139a6	8192	0x7fcfcba0c000	1	0
C	4810	0x7fcfcba24000	0x7fcfcba252ca	16384	0x7fcfcba12000	1	0
A	3426	0x7fcfcba0c000	0x7fcfcba0cd62	4096	0	1	0

Tabelle 2: Tabelle sortiert nach belegtem Speicherplatz

	free	pos	originalPage
-	0	0	0
B	0	0x7fcfcba139a6	0x7fcfcba12000
A	670	0x7fcfcba0cd62	0x7fcfcba0c000
C	5007	0x7fcfcba26c76	0x7fcfcba24000

Tabelle 3: Tabelle sortiert nach freiem Speicherplatz nach der Verschiebung

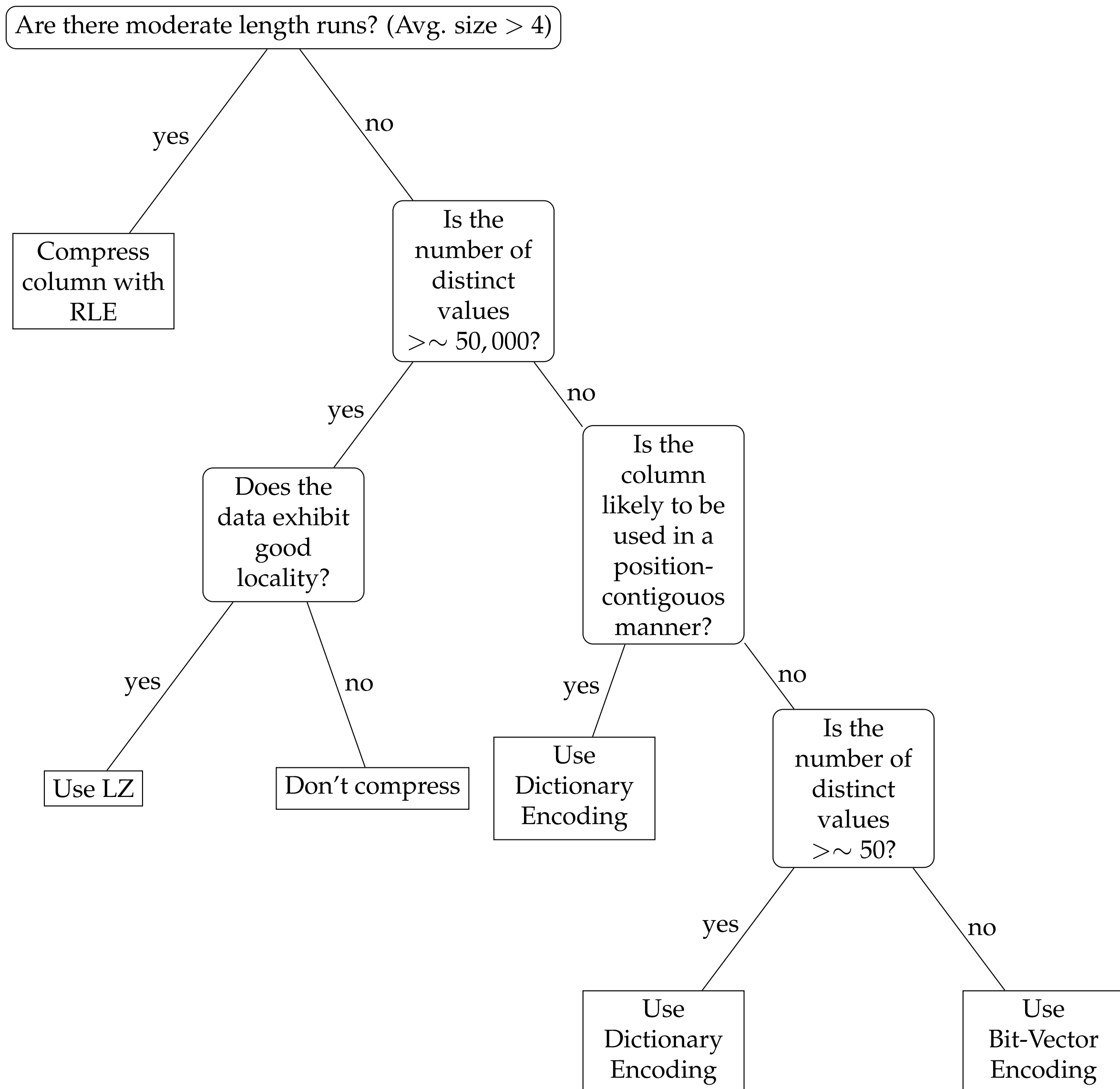


Abbildung 1: Entscheidungsbaum nach Abadi [3], um auf Grundlage der Daten eine geeignete Kompressionsmethode zu ermitteln

	used	currentPage	pos	pageSize	parentPage	pagesIn	offset
-	0	0	0	0	0	0	0
B	3	0x7fcfcba252ca	0x7fcfcba139a6	8192	0x7fcfcba0c000	1	6
C	11377	0x7fcfcba24000	0x7fcfcba26c76	16384	0x7fcfcba252d0	2	0
A	3426	0x7fcfcba0c000	0x7fcfcba0cd62	4096	0	1	0

Tabelle 4: Tabelle sortiert nach belegtem Speicherplatz nach der Verschiebung

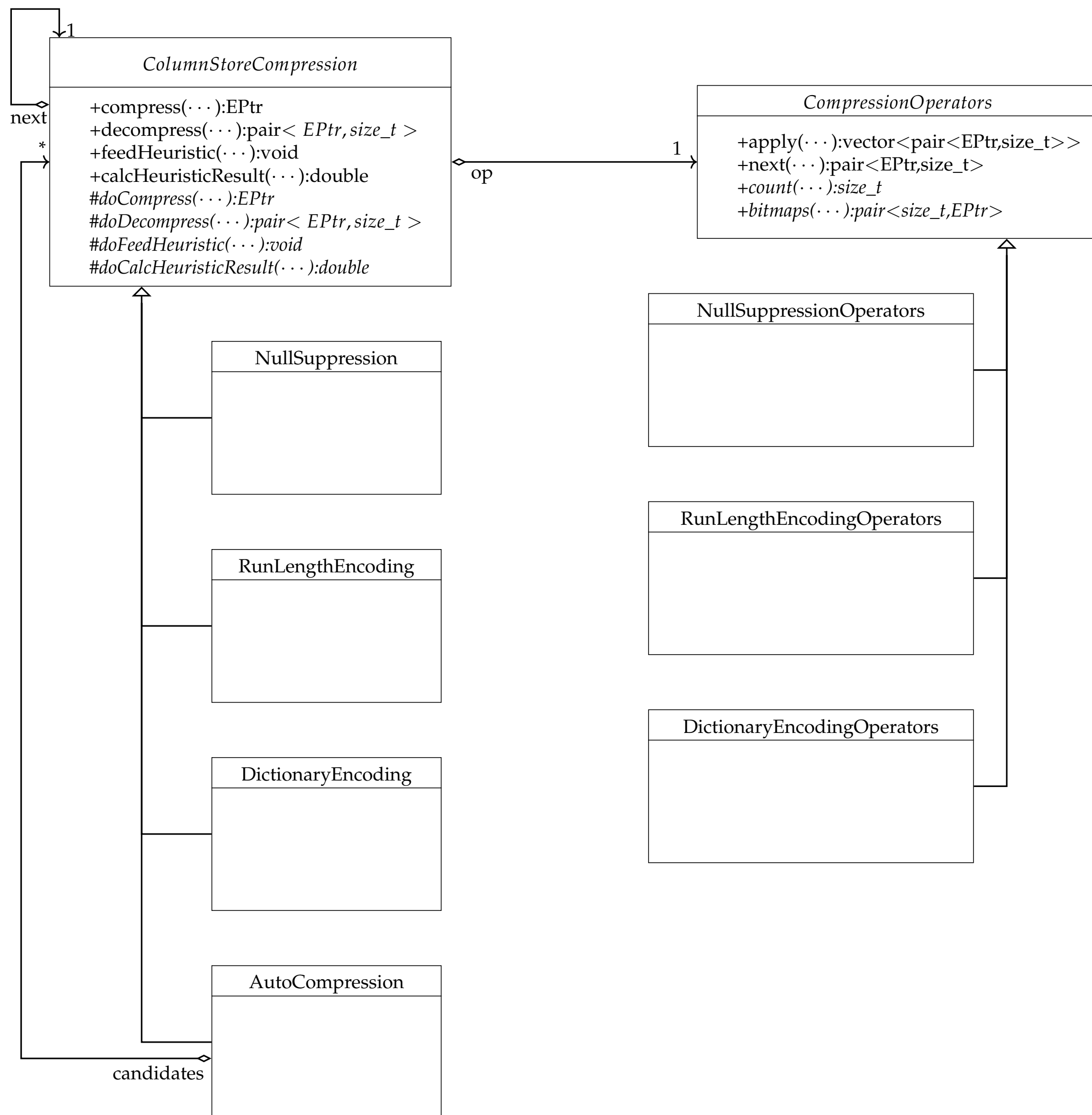


Abbildung 2: UML-Diagramm der Kompressionsklassen

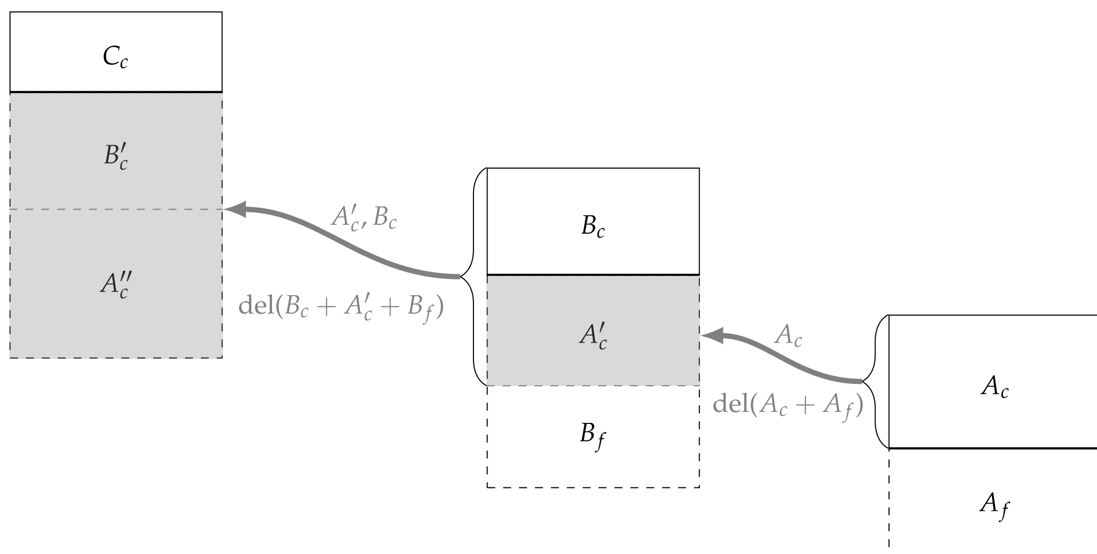


Abbildung 3: Page-Packing-Algorithmus (Ideal). Freier Speicher ist gestrichelt umrandet. Komprimierte Pages werden in den frei gewordenen Speicher anderer Pages verschoben (grau hinterlegt)