Modularisierung leichtgewichtiger Kompressionsalgorithmen

Juliana Hildebrandt, Dirk Habich, Patrick Damme, Wolfgang Lehner
Technische Universität Dresden
Database Systems Group
01189 Dresden, Germany
firstname.lastname@tu-dresden.de

ABSTRACT

Im Kontext von In-Memory Datenbanksystemen nehmen leichtgewichtige Kompressionsalgorithmen eine entscheidende Rolle ein, um eine effiziente Speicherung und Verarbeitung großer Datenmengen im Hauptspeicher zu realisieren. Verglichen mit klassischen Komprimierungstechniken wie z.B. Huffman erzielen leichtgewichtige Kompressionsalgorithmen vergleichbare Kompressionsraten aufgrund der Einbeziehung von Kontextwissen und erlauben eine schnellere Kompression und Dekompression. Die Vielfalt der leichtgewichtigen Kompressionsalgorithmen hat in den letzten Jahren zugenommen, da ein großes Optimierungspotential über die Einbeziehung des Kontextwissens besteht. Um diese Vielfalt zu bewältigen, haben wir uns mit der Modularisierung von leichtgewichtigen Kompressionsalgorithmen beschäftigt und ein allgemeines Kompressionsschema entwickelt. Durch den Austausch einzelner Module oder auch nur eingehender Parameter lassen sich verschiedene Algorithmen einfach realisieren.

1. EINFÜHRUNG

Die Bedeutung von In-Memory Datenbanksystemen steigt zunehmend sowohl im wissenschaftlichen als auch im kommerziellen Kontext. In-Memory Datenbanksysteme verfolgen einen hauptspeicherzentrischen Architekturansatz, der sich dadurch auszeichnet, dass alle performancekritischen Operationen und internen Datenstrukturen für den Zugriff der Hauptspeicherhierarchie (z.B. effiziente Nutzung der Cachehierarchie etc.) ausgelegt sind. Üblicherweise gehen In-Memory Datenbanksysteme davon aus, dass alle relevanten Datenbestände auch vollständig in den Hauptspeicher eines Rechners oder eines Rechnerverbundes abgelegt werden können. Die Optimierung der internen Datenrepräsentationen wird damit extrem wichtig, da jeder Zugriff auf ein Zwischenergebnis genau so teuer ist wie ein Zugriff auf die Basisdaten [13].

Für In-Memory Datenbanksysteme haben sich zwei orthogonale Optimierungstechniken für die effiziente Behandlung der Zwischenergebnisse etabliert. Auf der einen Seite sollten Zwischenergebnisse nicht mehr zum Beispiel durch entsprechend angepasste Code-Generierung [19] oder durch den Einsatz zusammengefügter Operatoren [16] produziert werden. Auf der anderen Seite sollten Zwischenergebnisse (wenn sie beispielsweise nicht vermeidbar sind) so organisiert werden, dass eine effiziente Weiterverarbeitung ermöglicht wird. Im Rahmen unserer aktuellen Forschung greifen wir uns den optimierten Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse in hauptspeicherzentrischen Datenbankarchitekturen heraus und haben zum Ziel, eine *ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* zu entwickeln [13]. Mit der expliziten Kompression aller Zwischenergebnisse soll (i) die Effizienz einzelner Datenbankanfragen bzw. der Durchsatz einer Menge an Datenbankanfragen erhöht werden, da der Hauptspeicherbedarf für Zwischenergebnisse reduziert und der Mehraufwand zur Generierung der komprimierten Form möglichst gering gehalten wird und (ii) die durchgängige Betrachtung der Kompression von den Basisdaten bis hin zur Anfrageverarbeitung etabliert wird.

Im Forschungsbereich der klassischen Kompression existiert eine Vielzahl an wissenschaftlichen Publikationen. Klassische Kompressionsverfahren, wie zum Beispiel arithmetisches Kodieren [26], Huffman [15] und Lempel-Ziv [30], erzielen hohe Kompressionsraten, sind jedoch rechenintensiv und werden deshalb oft als schwergewichtige Kompressionsverfahren bezeichnet. Speziell für den Einsatz in In-Memory Datenbanksystemen wurden leichtgewichtige Kompressionsalgorithmen entwickelt, die verglichen mit klassischen Verfahren aufgrund der Einbeziehung von Kontextwissen ähnliche Kompressionsraten erzielen, aber sowohl eine viel schnellere Kompression als auch Dekompression erlauben. Beispiele für leichtgewichtige Kompressionsalgorithmen sind unter anderem Domain Kodierung (DC) [24], Wörterbuch-basierte Kompression (Dict) [5, 8, 17], reihenfolgeerhaltende Kodierungen [5, 28], Lauflängenkodierung (RLE) [7, 21], Frame-of-Reference (FOR) [12, 31] und verschiedene Arten von Nullkomprimierung [1, 20, 21, 23]. Die Anzahl der leichtgewichtigen Kompressionsalgorithmen hat in den letzten Jahren zugenommen, da ein großes Optimierungspotential über die Einbeziehung von Kontextwissen besteht.

Mit Blick auf unser Ziel der *ausgewogenen Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* wollen wir eine breite Vielfalt an leichtgewichtigen Kompressionsalgorithmen unterstützen, um die jeweiligen Vorteile der Algorithmen effizient ausnutzen zu können. Um dieses Ziel zu erreichen, haben wir uns mit der Modularisierung von

27^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 26.05.2015 - 29.05.2015, Magdeburg, Germany.

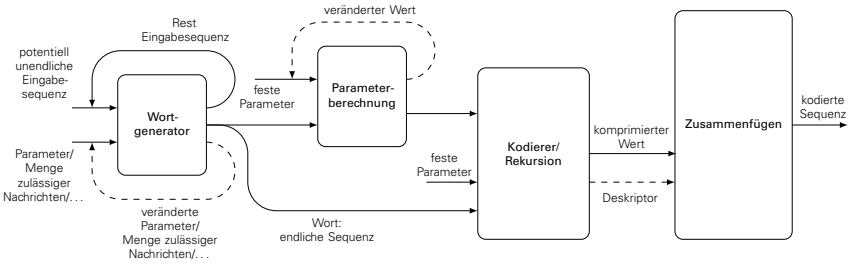

Abbildung 1: Allgemeines Schema für leichtgewichtige Komprimierungsalgorithmen.

Kompressionsalgorithmen beschäftigt. Für diese Modularisierung haben wir eine Vielzahl von leichtgewichtigen Kompressionsalgorithmen systematisch analysiert und ein allgemeines Kompressionsschema bestehend aus wohldefinierten Modulen abgeleitet. Durch den Austausch einzelner Module oder auch nur eingehender Parameter lassen sich häufig verschiedene Algorithmen einfach darstellen. Des Weiteren wird durch die Darstellung eines Algorithmus und durch die Unterteilung in verschiedene, möglichst unabhängige kleinere Module die Verständlichkeit erleichtert. Unsere entwickelte Strukturierung bildet eine gute Basis zur abstrakten und implementierungsunabhängigen Betrachtung von leichtgewichtigen Kompressionsalgorithmen.

Im Abschnitt 2 führen wir unser neues Kompressionsschema für leichtgewichtige Kompressionsverfahren bestehend aus vier Modulen ein. Dieses neue Kompressionsschema nutzen wir in Abschnitt 3, um bekannte Muster zu definieren. Im Anschluss daran gehen wir auf die Modularisierung konkreter Algorithmen exemplarisch im Abschnitt 4 ein. Der Artikel schließt dann mit einer Zusammenfassung und einem Ausblick im Abschnitt 5.

2. KOMPRESSIONSSCHEMA

Mit dem Paradigma der Datenkompression aus den 1980er Jahren [25] gibt es bereits eine eher allgemein gehaltene Modularisierung für die Kompression von Daten im Kontext der Datenübertragung. Diese unterteilt Kompressionsverfahren lediglich in ein *Datenmodell*, welches auf Grundlage bereits gelesener Daten erstellt und angepasst wird, und einen *Kodierer*, welcher eingehende Daten mithilfe des berechneten Datenmodells kodiert. Diese Modularisierung eignet sich für damals übliche adaptive Kompressionsmethoden; greift aber für viele Verfahren zu kurz. Die gesamte Zerlegung eines Eingabedatenstroms wird beispielsweise außen vor gelassen. Bei vielen aktuellen und gerade semiadaptiven Verfahren mit mehreren Pässen werden Daten mehrstufig zerlegt, um ein komplexes Datenmodell zu erzeugen. Auch das Zusammenfügen der Daten wird im Normalfall wesentlich diffiziler realisiert als mit einer einfachen Konkatenation komprimierter Daten. Unser aus vier Modulen bestehendes allgemeines Kompressionsschema in Abbildung 1 ist eine Erweiterung des bisherigen Paradigmas der Datenkompression, das eine wesentlich detailliertere und komplexere Abbildung einer Vielzahl von leichtgewichtigen Algorithmen erlaubt. Eingabe für ein Kompressionsverfahren ist hierbei immer eine potentiell unendliche Sequenz von Daten. Ausgabe ist ein Strom aus komprimierten Daten.

Der *Wortgenerator* als erstes Modul zerlegt die Sequenz in endliche Teilsequenzen resp. einzelne Werte. Mit einem Datenstrom als Eingabe gibt ein Wortgenerator einen endlichen Anfang aus und verarbeitet den Rest der Eingabe ebenso, *rekursiv*. Ist die Eingabe des Wortgenerators endlich, kann ihre Zerlegung stattdessen auch *nicht rekursiv*, z.B. ein Optimierungsproblem sein. Wortgeneratoren erhalten als zweite Eingabe die Information, wie die Eingabesequenz zerlegt werden soll, als Berechnungsvorschrift. Entweder wird *datenunabhängig* eine Anzahl von Werten ausgegeben (z.B. immer 128 Werte oder immer ein Wert) oder *datenabhängig* aufgrund inhaltlicher Merkmale die Länge der auszugebenden Teilsequenz bestimmt. Möglich ist eine *adaptive* Zerlegung, so dass sich die Berechnungsvorschrift nach jeder Ausgabe einer Teilsequenz ändert. Als optionaler Datenfluss ist dies im Kompressionsschema durch eine unterbrochene Linie dargestellt.

Das Datenmodell des Paradigmas der Datenkompression wird durch das Modul der *Parameterberechnung* ersetzt. So können bei semiadaptiven Verfahren für endliche Sequenzen statistische Werte berechnet werden, wie zum Beispiel ein Referenzwert für Frame-of-Reference-Verfahren (FOR) oder eine gemeinsame Bitweite, mit der alle Werte der endlichen Sequenz kodiert werden können. Möglicherweise gibt es feste Eingabeparameter, beispielsweise eine Auswahl an erlaubten Bitweiten. Eine *adaptive* Parameterberechnung zeichnet sich durch einen Startwert als festen Eingabeparameter aus und eine Ausgabe, die im nächsten Schritt wieder als Eingabe und so dem Modul der Parameterberechnung als Gedächtnis dient. Beispielsweise benötigen Differenzkodierungsverfahren eine adaptive Parameterberechnung.

Der *Kodierer* erhält einen atomaren Eingabewert sowie möglicherweise berechnete oder feste Parameter, die für die Kodierung des Eingabewertes benötigt werden. Solche Parameter können z.B. Referenzwerte, Bitweiten oder gar Mappings sein, die die Abbildung einzelner Werte in einen Code definieren. Ein Kodierer bildet einen Eingabewert eineindeutig auf einen anderen Wert ab, was für die Dekodierbarkeit notwendig ist. Ausgabe eines Kodierers ist ein komprimierter Wert, der sich möglicherweise durch einen Deskriptor wie einer Längenangabe bei einer Abbildung in einen variablen Code auszeichnet, um die Dekodierbarkeit zu gewährleisten. Soll eine endliche Sequenz, die der Wortgenerator ausgibt, noch weiter zerlegt werden, kann das gesamte Schema noch einmal mit einer *Rekursion* aufgerufen werden. Dabei geht eine endliche Sequenz wieder in einen Wortgenerator ein und wird dabei zerlegt, weiterverarbeitet und als komprimierte Sequenz wieder zusammengefügt. Diese komprimierte Sequenz ist Ausgabe der Rekursion.

Das letzte Modul des *Zusammenfügens* erhält als Eingabe einen komprimierten Datenstrom, nämlich die Ausga-

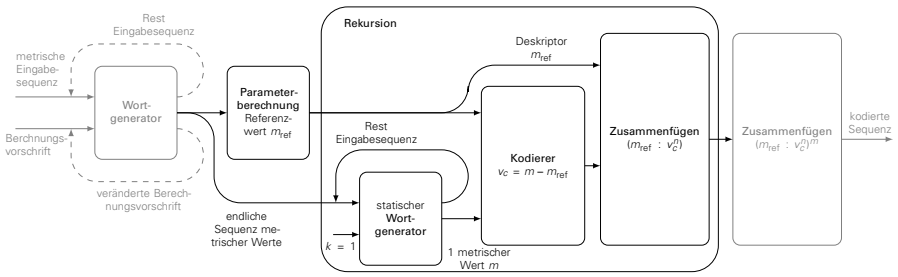

Abbildung 2: Modularisierung semiadaptiver Frame-of-Reference-Verfahren.

ben des Kodierers resp. der Rekursion. Es gibt verschiedene Möglichkeiten Daten zusammenzufügen. Im einfachsten Fall gibt es keine Deskriptoren, alle komprimierten Werte v_c werden nacheinander zusammengefügt (notiert als v_c^n). Gehört zu jedem komprimierten Wert ein Deskriptor d, so können beispielsweise immer Paare aus Deskriptoren und komprimierten Werten konkateniert werden (notiert als $(d : v_c)^n$) oder immer eine bestimmte Anzahl l von Deskriptoren, gefolgt von den zugehörigen komprimierten Werten (notiert als $(d^l : v_c^l)^n$). Gerade bei semiadaptiven Verfahren mit Rekursionen ist es möglich, dass gemeinsame Deskriptoren für mehrere Werte vom Modul der Parameterberechnung ausgegeben und mit gespeichert werden müssen, so dass verschiedene Anordnungen bei der Konkatenation aller Werte denkbar sind.

3. KOMPRESSIONSMUSTER

Bekannte Kompressionstechniken wie zum Beispiel Differenzkodierung, Frame-of-Reference (FOR), Wörterbuchkompression, Bitvektoren, Lauflängenkodierung (RLE) oder die Unterdrückung führender Nullen lassen sich mit dem allgemeinen Kompressionsschema als Muster ausdrücken. Das bedeutet, dass gewisse modulare Anordnungen und Inhalte einzelner Module durch die Begriffsdefinition der Techniken festgelegt, andere inhaltliche Aspekte sowie andere in Beziehung stehende Module hingegen nicht näher spezifiziert sind.

3.1 Muster Frame-of-Reference (FOR)

Für die allgemeine Definition des FOR muss die Eingabesequenz aus metrischen Werten bestehen, wie zum Beispiel aus natürlichen Zahlen. Diese werden als Differenz zum Referenzwert m_{ref} kodiert. Abbildung 3 zeigt das entsprechende Kompressionsschema. Der Wortgenerator gibt vom Anfang der Sequenz jeweils einen Integerwert m aus. Der Kodierer erhält neben den Eingabewerten den Referenzwert m_{ref} als

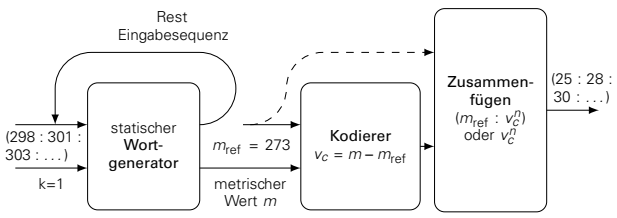

Abbildung 3: Modularisierung statischer Frame-of-Reference-Verfahren.

Parameter und berechnet die Differenz aus beiden Werten. Das Modul des Zusammenfügens konkateniert den Referenzwert mit allen kodierten Werten ($m_{ref} : v_c^n$). Notwendig ist die Speicherung des Referenzwertes aber nur, wenn dessen Kenntnis beim Dekodieren nicht vorausgesetzt werden kann. Dies ist durch einen optionalen Pfeil dargestellt. Im dargestellten Beispiel werden die Werte des Eingabedatenstroms als Differenz zum Referenzwert $m_{ref} = 273$ kodiert. Der Referenzwert sei beim Dekodieren aus dem Kontext bekannt.

Meist gehört es zum Selbstverständnis, dass der Referenzwert für eine endliche Sequenz wie in Abbildung 2 aus den gegebenen Daten berechnet und als Deskriptor gespeichert wird. Nach welchen Regeln der erste dargestellte Wortgenerator endliche Sequenzen ausgibt, ist dabei nicht spezifiziert. Das dargestellte Muster kann eine potentiell unendliche Sequenz als Eingabe erhalten. Es kann auch in einem größeren Zusammenhang mit anderen Modulen stehen und nur eine endliche Teilsequenz weiter zerlegen. Im Modul der Parameterberechnung wird aus der endlichen Sequenz, die der erste Wortgenerator ausgibt, der Referenzwert m_{ref} berechnet. Zum Beispiel kann als Referenzwert der kleinste Wert der endlichen Sequenz gewählt werden. Die endliche Sequenz geht in eine Rekursion ein. Arrangement und Inhalt der Module innerhalb der Rekursion entsprechen der Modularisierung statischer Frame-of-Reference-Verfahren (vgl. Abb. 3). Der Wortgenerator innerhalb der Rekursion gibt einzelne metrische Werte aus. Der Kodierer berechnet die Differenz aus Eingabe- und Referenzwert. Alle Werte werden gemeinsam mit dem Referenzwert konkateniert. Alle so komprimierten endlichen Sequenzen, die der erste dargestellte Wortgenerator ausgegeben hat, werden am Ende zusammengefügt, von Interesse sind für die allgemeinere Definition des FOR jedoch nur die Module innerhalb der Rekursion.

3.2 Muster Symbolunterdrückung

Unter dem Begriff Symbolunterdrückung werden sehr verschiedene Komprimierungsverfahren zusammengefasst, Präsenzbits sowie Lauflängenkodierung explizit für Nullen, die in potentiell unendliche Sequenzen auftauchen [6], Lauflängenkodierung von Nullen und Leerzeichen [21] oder auch die Eliminierung führender und damit redundanter Nullen bei binär kodierten Zahlen. Diese Methoden haben gemeinsam, dass es im Zeichenvorrat ein ausgezeichnetes Symbol s gibt, welches sich meist semantisch von allen anderen abhebt und öfter auftaucht als andere Werte. Das ausgezeichnete Symbol wird im Wortgenerator oder im Kodierer anders behandelt als andere Symbole. Im Falle von Präsenzbits ist dieses Symbol der NULL-Wert. Nullen sind das neutrale Element der Addition. Die genaue Anzahl führender Nullen beeinflusst

Additionsoperationen nicht und ist damit an sich schon eine redundante Information. Leerzeichen dienen in allen Sprachen dazu, Wörter voneinander zu separieren. Die Anzahl an Wiederholungen von Leerzeichen zwischen konkatenierten Wörtern besitzt auf semantischer Ebene keinerlei Bedeutung. Viele der Algorithmen, aber nicht alle, nutzen hierfür RLE-Kompressionen. Für Symbolunterdrückungen lässt sich allgemein keine Modularisierung darstellen, da es sich einfach nur durch die Sonderbehandlung eines Symbols auszeichnet. In Kombination mit einer Lauflängenkodierung gelingt aber eine Modularisierung mit unserem Schema.

Merkmal der Lauflängenkodierung ist das Vorhandensein von Läufen w^n, endlichen Sequenzen der Länge n aus ein und demselben Wert w. Werden wirklich einfach nur Läufe von Werten kodiert, so reicht das simple Kompressionsschema in Abbildung 4 aus. Der Wortgenerator unterteilt den

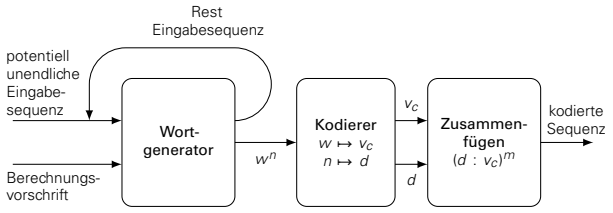

Abbildung 4: Modularisierung einer einfachen Lauflängenkodierung.

Eingabedatenstrom in Läufe. Im Kodierer werden dann der Wert w und die Lauflänge n kodiert. Läufe können auch in einer Sequenz zum Beispiel als führende Nullen eingebettet sein. Solche Fälle liegen im Schnittbereich zwischen Symbolunterdrückung und Lauflängenkodierung. Dies ist für statische Verfahren in Abbildung 5 dargestellt. Die Informationen über Sequenzlängen des ausgezeichneten Wertes s werden bei statischen Verfahren beim Zusammenfügen zum Beispiel in der Form $(d(n) : v_c)^m$ gespeichert. Dabei ist $d(n)$ eine eineindeutige Abbildung.

Sollen mehrere mit 32 Bits kodierte Werte mit geringerer, aber einheitlicher Bitweite gespeichert werden, wird für die Unterdrückung führender Nullen ein semiadaptives Schema benötigt (nicht dargestellt). Die gemeinsame Bitweite bw ist Ausgabe der Parameterberechnung und kann als Deskriptor angegeben werden, $bw = d(n)$ ist eine Funktion von n, der Anzahl der führenden Nullen, die entfernt wurden. Jeder

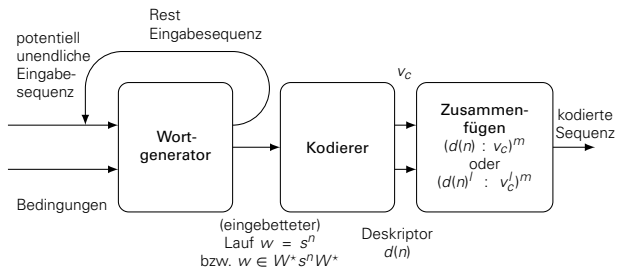

Abbildung 5: Modularisierung von symbolunterdrückenden Verfahren mit Lauflängenkodierung als statische Kompressionsverfahren.

Wert w einer endlichen Sequenz hat eine komprimierte Form v_c und eine Lauflänge n. Beide werden entweder zusammen oder in der Gruppe aus Deskriptoren und einer Gruppe aus komprimierten Werten gespeichert.

4. ALGORITHMEN-MODULARISIERUNG

Die vorgestellten und auch weitere Muster finden sich in verschiedenen Kompressionsalgorithmen, oft auch kombiniert oder auf mehreren Rekursionsebenen miteinander verwoben, wieder. Sich ähnelnde Algorithmen unterscheiden sich meist nur geringfügig in manchen Modulen oder sogar nur in Parametern, die in ein Modul eingehen. Beispielsweise ähneln sich die Algorithmen varint-PU und varint-SU [22] - letzterer ist besser bekannt als VByte [11, 10, 9] - sehr. VByte kodiert 32-Bit-Integerwerte mit ein bis 5 Bytes, wobei ein Byte aus einem Deskriptorbit und 7 Datenbits besteht. Ebenso ist dies bei varint-PU der Fall, beide Algorithmen unterscheiden sich nur in der Anordnung der Daten- und Deskriptorbits. Während bei VByte ein Bit pro zusammenhängendem Byte als Deskriptor dient, steht bei varint-PU der gesamte Deskriptor an einem Ende des komprimierten Integerwertes. Ein Beispiel zeigt Abbildung 6. Nicht belegte Bits bedeuten, dass diese im Beispiel nicht benötigt und weggelassen werden. Der Integerwert wird in komprimierter Form mit 3 statt 4 Bytes kodiert.

Beide Formate haben den gleichen modularen Aufbau (siehe Abbildung 7). Der rekursive statische Wortgenerator gibt immer eine Zahl aus. Da die Kodierung der Eingabe bei diesen Algorithmen soweit spezifiziert ist, dass die Zahlen als 32-Bit-Integerwerte kodiert sind, ist die ausgegebene Zahl ein eingebetteter Lauf von der Form $0^l 1 w_1 \dots w_{31-l}$ (bzw. 0^{32} für den Wert 0). Der Deskriptor $bw_{/7}$ gibt die Anzahl der für die Datenbits benötigten 7-Bit-Einheiten an. Allein aus dem komprimierten Wert ohne Deskriptor ist die Lauflänge der bei der Kodierung unterschlagenen Nullen nicht ermittelbar, schon weil eine Folge von Werten nicht mehr dekodierbar ist. Die Lauflänge ist allein aus dem Deskriptor (und dem Wissen, dass es sich um 32-Bit-Integerwerte handelt) ersichtlich. Somit ist das Lauflängenmuster bei varint-SU und varint-PU begründbar. Da das Zeichen 0 eine Sonderstellung einnimmt, weil führende Nullen als Lauf betrachtet werden, findet sich hier, wie bei allen varint-Algorithmen, auch eine Symbolunterdrückung. Beide Algorithmen unterscheiden sich im Kompressionsschema nur im Modul des *Zusammenfügens*. Das Symbol $:$ wird hier als Konkatenationssymbol für abzählbar viele Werte verwendet.

Ein weiteres Beispiel für einen modularisierten Algorithmus ist FOR mit Binary Packing (nicht dargestellt), der sich durch marginale Veränderungen und weitere Definitionen aus dem Kompressionsschema für semiadaptive FOR-Verfahren (Abb. 2) ergibt. Beim Binary Packing wird für eine endliche Sequenz von n binär kodierten Integerwerten z.B. zu 32 Bits eine gemeinsame Bitweite bw berechnet, mit der alle n Werte kodiert werden können. Die erste Änderung im Kompressionsschema betrifft die Parameterberechnung. Zusätzlich zum Referenzwert für eine endliche Sequenz, die der erste Wortgenerator ausgibt, muss die gemeinsame Bitweite bw berechnet und ausgegeben werden. Die zweite Änderung betrifft den Kodierer innerhalb der Rekursion. Nach der Berechnung der Differenz aus Eingabe- und Referenzwert wird der so erhaltene Werte mit Bitweite bw binär kodiert. Im Modul des Zusammenfügens muss dann bw als ein weiterer gemeinsamer Deskriptor zum Beispiel in der Form

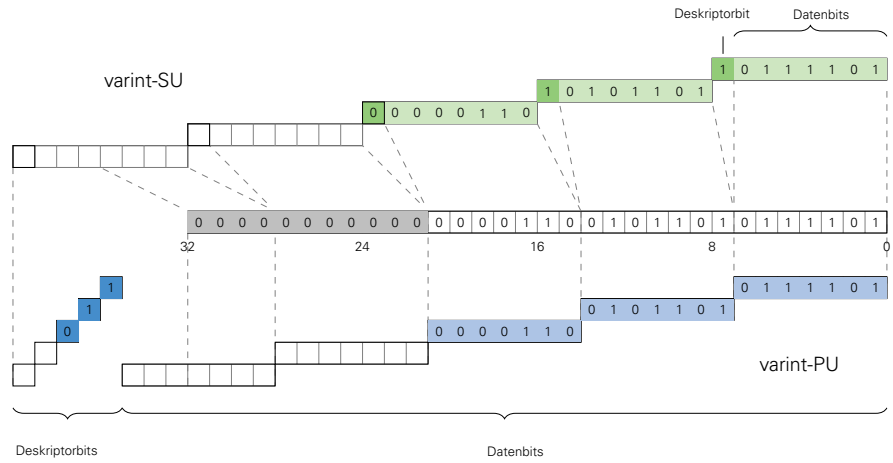

Abbildung 6: Datenformat varint-SU und varint-PU.

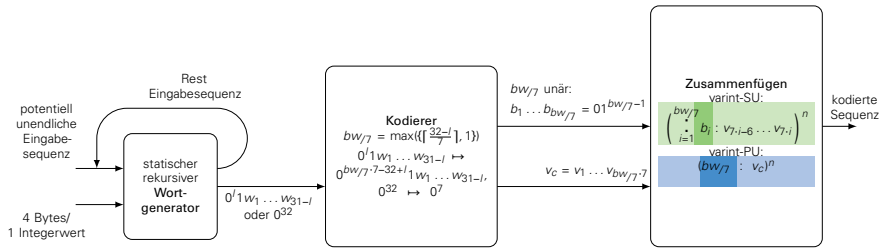

Abbildung 7: Modularisierung der Algorithmen varint-SU und varint-PU.

$(m_{\text{ref}} : bw : v_c^n)$ gespeichert werden. Die Modularisierung dieses Algorithmus zeichnet sich durch die Muster FOR, Lauflängenkodierung und Symbolunterdrückung aus.

Nicht für alle Algorithmen ist diese recht einfache Modularisierung ausreichend. Auf PFOR basierende Algorithmen [31, 29, 27, 18] kodieren die meisten Eingabewerte aus natürlichen Zahlen mit der gleichen Bitweite bw. Die größeren, die nicht mit der Bitweite bw kodierbar sind, werden allerdings als Ausnahme deklariert und auf andere Weise kodiert und an anderer Stelle gespeichert. Dafür benötigt das erweiterte Schema ein Splitmodul, welches Daten aufgrund inhaltlicher Merkmale in verschiedene Gruppen aufteilt und ausgibt. Für jede dieser Gruppen muss ein separater Kodierer verfügbar sein, wobei die kodierten Werte aller Gruppen am Ende gemeinsam zusammengefügt werden.

5. ZUSAMMENFASSUNG UND AUSBLICK

Unser entwickeltes Kompressionsschema bestehend aus vier Modulen ist durchaus geeignet, um eine Vielzahl verschiedener leichtgewichtiger Kompressionsalgorithmen gut zu modularisieren und systematisch darzustellen. Durch den Austausch einzelner Module oder auch nur eingehender Parameter lassen sich verschiedene Algorithmen mit dem gleichen Kompressionsschema darstellen. Einige Module und Modulgruppen tauchen in verschiedenen Algorithmen immer wieder auf, wie zum Beispiel die gesamte Rekursion, die das Binary Packing ausmacht, die sich in allen PFOR- und Simple-Algorithmen [2, 3, 29, 27, 4] findet. Die Verständlichkeit eines Algorithmus wird durch die Unterteilung in verschiedene, möglichst unabhängige kleinere Module, welche überschaubare Operationen ausführen, verbessert. Die Strukturierung durch das entwickelte Schema bildet aus unserer Sicht eine gute Basis zur abstrakten Betrachtung von leichtgewichtigen Kompressionsalgorithmen. Als Muster können nicht nur bestimmte Techniken, sondern auch andere Eigenschaften von Kompressionsalgorithmen dargestellt werden. Statische Verfahren wie z.B. varint-SU und varint-PU bestehen nur aus Wortgenerator, Kodierer und dem Modul des Zusammenfügens. Adaptive Verfahren haben einen adaptiven Wortgenerator, eine adaptive Parameterberechnung oder beides. Semiadaptive Verfahren zeichnen sich durch eine Parameterberechnung und eine Rekursion aus, in deren Wortgenerator oder Kodierer die Ausgabe der Parameterberechnung eingeht.

Durch die Möglichkeit Module sehr passend zusammenzustellen und mit Inhalt zu füllen, ergibt sich ein mächtiges Werkzeug für den automatisierten Bau von Algorithmen. Das Kompressionsschema bietet eine aus unserer Sicht fundierte Grundlage und eröffnet die Möglichkeit, für einen gegebenen Kontext sehr gezielt speziell zugeschnittene Algorithmen mit bestimmten Eigenschaften wie zum Beispiel der Art der Anpassbarkeit zusammenzubauen. Weiterhin können verschiedene Muster wie FOR, Differenzkodierung, Symbolunterdrückung oder Lauflängenkodierung an den Kontext angepasst eingesetzt werden und das auf verschiedensten Ebenen miteinander kombiniert.

Für die Fortführung dieses Gedankens ist es notwendig, einen noch stärkeren Zusammenhang zwischen Kontextwissen und passender Schemazusammenstellung sowie passen-

den Parametereingaben herzustellen. Des Weiteren wird gerade für das theoretische Grundkonzept eine passende praktische Umsetzung angegangen. Für die praktische Umsetzung wird ein Framework bestehend aus den eingeführten Modulen anvisiert, so dass der Zusammenbau leichtgewichtiger Kompressionsalgorithmen wie beschrieben realisiert werden kann. Die größte Herausforderung bei der praktischen Umsetzung wird die Effizienz der Algorithmen sein. Um eine vergleichbare Effizienz zu den bisherigen Implementierungen erzielen zu können, sind unterschiedliche Ansätze notwendig. Ein vielversprechender Ansatz dabei ist die Spezialisierung von generischen Code mit dem Einsatz spezieller Compilertechniken, wie wir es in [14] angesprochen haben. Über die Spezialisierung kann hochoptimierter Ausführungscode erzeugt werden, wobei das vorhandene Hintergrundwissen zur Codeoptimierung dem Compiler beigebracht werden muss.

Acknowledgments

Diese Arbeit ist im Rahmen des DFG-finanzierten Projektes "Leichtgewichtige Kompressionsverfahren zur Optimierung komplexer Datenbankanfragen"(LE-1416/26-1) entstanden.

6. LITERATUR

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, Jan. 2005.

[3] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):857–861, June 2006.

[4] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw. Pract. Exper.*, 40(2):131–147, Feb. 2010.

[5] G. Antoshenkov, D. B. Lomet, and J. Murray. Order preserving compression. In *ICDE*, pages 655–663, 1996.

[6] J. Aronson. Computer science and technology: data compression — a comparison of methods. NBS special publication 500-12, Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology, Washington, DC, USA, June 1977. ERIC Document Number: ED149732.

[7] M. A. Bassiouni. Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering*, 11(10):1047–1058, 1985.

[8] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[9] S. Büttcher, C. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[10] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[11] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In R. A. Baeza-Yates, P. Boldi, B. A. Ribeiro-Neto, and B. B. Cambazoglu, editors, *WSDM*, page 1. ACM, 2009.

[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE Conference*, pages 370–379, 1998.

[13] D. Habich, P. Damme, and W. Lehner. Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse. In *BTW 2015*, pages 259–278, 2015.

[14] C. Hänsch, T. Kissinger, D. Habich, and W. Lehner. Plan operator specialization using reflective compiler techniques. In *BTW 2015*, pages 363–382, 2015.

[15] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[16] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. QPPT: query processing on prefix trees. In *CIDR 2013*, 2013.

[17] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *SIGMOD Conference*, pages 239–250, 1986.

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

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[20] H. K. Reghbati. An overview of data compression techniques. *IEEE Computer*, 14(4):71–75, 1981.

[21] M. A. Roth and S. J. V. Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[22] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[23] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[24] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[25] R. N. Williams. *Adaptive Data Compression*. 1991.

[26] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications ACM*, 30(6):520–540, 1987.

[27] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[28] A. Zandi, B. Iyer, and G. Langdon. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, pages 330 –339, 1993.

[29] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[31] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.