Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse

Dirk Habich, Patrick Damme, Wolfgang Lehner

Technische Universität Dresden
Institut für Systemarchitektur
Professur für Datenbanken
Nöthnitzer Strasse 46
01187 Dresden
dirk.habich@tu-dresden.de
patrick.damme@mailbox.tu-dresden.de
wolfgang.lehner@tu-dresden.de

Abstract: In Hauptspeicher-zentrischen Architekturansätzen für Datenbanksysteme müssen für die Anfrageverarbeitung sowohl die Basisrelationen als auch die Zwischenergebnisse im Hauptspeicher gehalten werden. Des Weiteren ist der Aufwand, um Zwischenergebnisse zu generieren, äquivalent zum Aufwand, um Änderungen an den Basisrelationen durchzuführen. Daher sollten zur effizienten Anfrageverarbeitung die Zwischenergebnisse so organisiert werden, dass eine effiziente Verarbeitung im Anfrageplan möglich ist. Für diesen Bereich schlagen wir den durchgängigen Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse vor und haben das Ziel, eine ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse zu entwickeln. Dieser Artikel gibt einen Überblick über unser Konzept und die wissenschaftlichen Herausforderungen. Darüber hinaus führen wir erste Ansätze zur Optimierung von leichtgewichtigen Kompressions- und Transformationsverfahren ein, die aus Sicht der Anfrageverarbeitung eventuell sinnvoll sind, um einen effizienten Wechsel des Kompressionsverfahrens durchzuführen.

1 Einführung

Die Bedeutung von In-Memory-Datenbanksystemen steigt zunehmend sowohl im wissenschaftlichen als auch im kommerziellen Kontext. In-Memory-Datenbanksysteme verfolgen einen Hauptspeicher-zentrischen 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 (Cluster-Konfiguration) abgelegt werden können.

Im Gegensatz dazu stehen (klassische) Festplatten-zentrische Architekturansätze, deren originäres Ziel es ist, den Zugriff auf den Externspeicher zu optimieren; effiziente Primär-

und Sekundärindexstrukturen werden gepflegt und der Anfrageoptimierer entscheidet für eine gegebene Anfrage, wie und in welcher Reihenfolge auf die auf Externspeicher abgelegten Daten zugegriffen wird. Die Optimierung der internen Datenrepräsentation ist in diesen Architekturansätzen nachgelagert, da Externspeicherzugriffe um Größenordnungen langsamer als direkte Zugriffe auf den Hauptspeicher sind. Mit der Entwicklung hin zu Hauptspeicher-zentrischen Architekturansätzen ändert sich dieser Aspekt jedoch komplett: Jeder Zugriff auf ein Zwischenergebnis ist genauso teuer wie ein Zugriff auf die Basisdaten; entsprechend ist auch der Aufwand, Zwischenergebnisse zu generieren (bis auf den zusätzlichen Aufwand der Protokollierung) ähnlich teuer wie Änderungen an Basisrelationen.

Auf Grundlage des veränderten Architekturverständnisses bieten sich zwei orthogonale Optimierungstechniken für die effiziente Behandlung der Zwischenergebnisse an. Auf der einen Seite sollten Zwischenergebnisse nicht mehr zum Beispiel durch entsprechend angepasste Code-Generierung [Neu11] oder durch den Einsatz kooperativer Operatoren [KSHL13] produziert werden. Auf der anderen Seite sollten Zwischenergebnisse (wenn sie beispielsweise nicht vermeidbar sind) so organisiert werden, dass eine effiziente Weiterverarbeitung ermöglicht wird. In diesem Kontext 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. 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. Diese Art der Anfrageoptimierung wurde bereits diskutiert [CGK01], jedoch nicht näher untersucht, da der Rechenaufwand für die Komprimierung den Nutzen der Transferkostenreduktion bisher überstieg. Durch die immer größer werdende Schere zwischen Rechenleistung und Speicherbandbreite bei modernen Mehrkernprozessoren [QRR+08] und der Weiterentwicklung insbesondere leichtgewichtiger Kompressionsverfahren [SGL10, WPB+09, ZHNB06] verliert dieses Argument jedoch zusehends seine Gültigkeit. Dennoch ist es gerade wegen der zusätzlichen Kosten von Kompressionsverfahren wichtig, dass eine *Balance* zwischen den reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten hergestellt wird, um die Gesamtausführungszeiten einer Anfrage zu minimieren.

Zur Erreichung unseres Ziels liefert dieser Artikel einen ersten aber nicht umfassenden Beitrag, wobei folgende Schwerpunkte betrachtet werden:

1. Einen ausführlichen Überblick über den Stand der Technik leichtgewichtiger Kompressionsverfahren und deren Einsatz in Datenbanksystemen (Abschnitt 2).
2. Eine strukturierte Formulierung unseres Ziels und der notwendigen wissenschaftlichen Herausforderungen, denen wir uns zur Zielerreichung stellen müssen (Abschnitt 3).
3. Aufbauend auf der Zielformulierung präsentieren wir den aktuellen Forschungs-

stand. Neben der effizienten Umsetzung von leichtgewichtigen Kompressionsverfahren haben wir uns mit einer neuen Klasse von Transformationsverfahren beschäftigt, um Daten von einem Kompressionsschema in ein anderes Schema effizient zu überführen. Derartige Transformationen sind im Rahmen einer Anfrageverarbeitung sinnvoll, da sich die Datencharakteristiken während der Verarbeitung in der Regel stark ändern können und damit Einfluss auf die Anwendbarkeit mit Blick auf Effizienz und Kompressionsrate der Kompressionsverfahren haben (Abschnitt 4).

4. Im Abschnitt 5 evaluieren wir unsere leichtgewichtigen Kompressions- und Transformationsverfahren und zeigen das Potential für eine Integration in die Anfrageverarbeitung auf.

2 Stand der Technik

Alle Arbeiten zu Kompression in relationalen Datenbanksystemen können in zwei große Bereiche eingeteilt werden. Der erste Bereich beschäftigt sich mit der Kompression von Relationen auf verschiedenen Granularitätsstufen: (1) Zeilenebene [GRS98, IW94, RHS95], (2) Spaltenebene [AMF06, MF04] und (3) Seiten- bzw. Blockebene [PP03]. Kompression auf Spalten- und Zeilenebene ermöglicht es unter bestimmten Voraussetzungen, Datenbankoperationen direkt auf den komprimierten Daten auszuführen, während komprimierte Seiten vor ihrer Benutzung vollständig entpackt werden müssen. Der zweite Bereich befasst sich mit der Kompression von Indexstrukturen, die oftmals durch ihre Anzahl in einem relationalen Datenbanksystem (pro Relation mehrere Indizes) ähnlich viel Speicherplatz wie die Relationen selbst belegen und daher auch großes Kompressionspotential bieten. Bezüglich der Granularität lässt sich die Kompression von Indexstrukturen auf der Ebene von Seiten [BB07, GRS98], Schlüsseln [BU77, BLM+09, Com79, SDF+00] und Tupel-IDs (TIDs) [BLM+09, Tro03] einteilen. TID-basierte Indexstrukturen mit vielen TIDs pro Schlüssel sind ein Spezialfall schlüsselbasierter Indexstrukturen und besitzen oft ein höheres Kompressionspotential.

Der hier betrachtete Aspekt der Kompression von Zwischenergebnissen während der Anfrageverarbeitung wurde bisher kaum oder nur unvollständig betrachtet, da üblicherweise der Zugriff auf Externspeicher die Gesamtausführungskosten dominiert. Dabei ist dieser Ansatz unabhängig davon, ob ein Festplatten- oder Hauptspeicher-zentrischer Architekturansatz für Datenbanksysteme betrachtet wird, da Zwischenergebnisse (1) möglichst immer im Hauptspeicher gehalten werden sollten und (2) großes Kompressionspotential bieten. Bisher werden jedoch Zwischenergebnisse nur insofern komprimiert, als dass versucht wird, solange wie möglich Datenbankoperationen auf komprimiert vorliegenden Basisdaten durchzuführen [AMF06, Goy83, GS91, RS06]. Jedoch erlauben nicht alle Datenbankoperatoren die Verarbeitung komprimierter Daten, so dass die Datensätze beziehungsweise ganze Seiten während der Anfrageverarbeitung im Anfrageplan einmalig dekomprimiert und ab dieser Stelle weiter unkomprimiert verarbeitet werden müssen. Abbildung 1 fasst nochmals die beschriebenen Ansatzpunkte für Kompression in Datenbanken zusammen und zeigt die Einordnung unserer aktuellen Forschung in diesem Bereich.

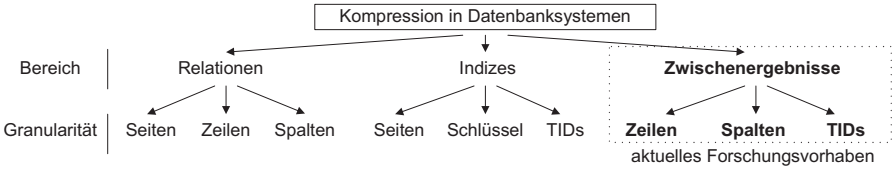


Abbildung 1: Kompression in relationalen Datenbanksystemen.

2.1 Arten von Kompressionsverfahren

Im Forschungsbereich der klassischen Kompression existiert eine Vielzahl an wissenschaftlichen Publikationen. Klassische Kompressionsalgorithmen, wie z.B. arithmetisches Kodieren [WNC87], Huffman [Huf52], Predictive Coding [CIW84] und Lempel-Ziv (LZW) [LZ76, ZL77], erzielen hohe Kompressionsraten, sind jedoch rechenintensiv und werden deshalb oft als *schwergewichtige* Kompressionsverfahren bezeichnet. Aufgrund der Rechenintensität werden die durch Kompression verringerten I/O-Transferzeiten wieder aufgehoben, so dass klassische Verfahren für die Kompression von Zwischenergebnissen im Hauptspeicher höchstwahrscheinlich nicht geeignet sind.

Speziell für den Einsatz in Datenbanksystemen wurden *leichtgewichtige* Kompressionsalgorithmen entwickelt, die verglichen mit klassischen Verfahren aufgrund der Einbeziehung von Kontextwissen ähnlich hohe und teilweise auch höhere Kompressionsraten erzielen, aber sowohl eine viel schnellere Kompression als auch Dekompression erlauben.

Beispiele für leichtgewichtige Kompressionsalgorithmen sind unter anderem Domain Kodierung (DC) [WPB+09], Wörterbuch-basierte Kompression (Dict) [ALM96, BMK99, LC86], reihenfolgeerhaltende Kodierungen [ALM96, ZIL93], Lauflängenkodierung (RLE) [Bas85, RH93], Frame-of-Reference (FOR) [GRS98, ZHNB06] und verschiedene Arten von Nullwertkomprimierung [AMF06, Reg81, RH93, WKHM00]. Abbildung 2 zeigt eine Klassifikation einer Auswahl relevanter leichtgewichtiger und schwergewichtiger Kompressionsalgorithmen.

Für die verschiedenen Datentypen eines Datenbanksystems werden unterschiedliche Kompressionsverfahren eingesetzt, die sich grob in Verfahren zur Kompression von (1) numerischen Werten und (2) Zeichenketten einteilen lassen. Für numerische Werte wie Ganzzahlen, Gleitkommazahlen oder Datumsangaben wurde eine Vielzahl von Kompressionsverfahren [CGK00, GRS98, Reg81, RH93] entwickelt. Da für die Kompression von Zeichenketten, wie z.B. CHAR, VARCHAR, BLOB, weitere Kontextinformationen zur Verfügung stehen, wurden für diese Datentypen speziell angepasste Verfahren [BHF09, BCE77, CGK01] entworfen.

Neben diesen Datentyp-spezifischen Kompressionsverfahren, existiert ebenfalls eine aktuelle Arbeit, die sich der Kompression von Verbundresultaten widmet [MLL13]. In diesem Ansatz wird ein Wörterbuch-basiertes Kompressionsverfahren vorgestellt, welches sowohl Informationen aus dem Anfrageplan als auch dem Datenbankschema nutzt, um Verbundergebnisse sehr kompakt zu repräsentieren. Ziel dieser Arbeit war, die Ergebnisübertragung in einem Client-Server-Umfeld zu optimieren.

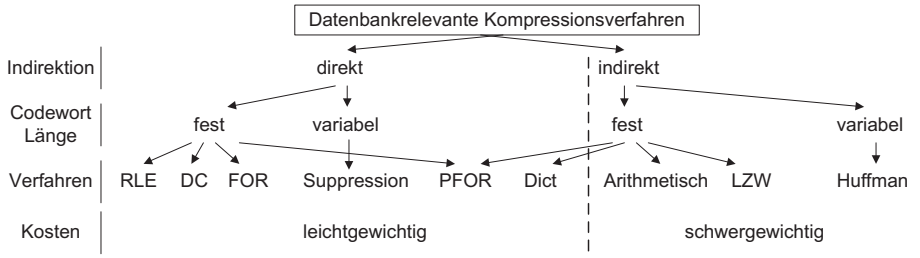


Abbildung 2: Klassifikation verschiedener Kompressionsverfahren.

2.2 Auswahl von Kompressionsverfahren

Ein wichtiger Aspekt bei der Kompression in Datenbanksystemen ist die Auswahl des besten Verfahrens für die vorliegenden Daten. Die Auswahl des Kompressionsverfahrens bzgl. der Datencharakteristiken von Relationen wird in unterschiedlichen Arbeiten adressiert. Dabei lassen sich diese Arbeiten nach ihrer Granularität klassifizieren. Zum einen existieren verschiedene Arbeiten [GRS98, GS91, WKHM00], die auf Ebene einzelner Attribute die Auswahl vornehmen, so dass für jeden Attributtyp einer Relation ein angepasstes Kompressionsverfahren genutzt wird. Ein Spezialfall bzw. eine Vereinfachung bildet dabei die Auswahl eines Kompressionsverfahrens für komplette Spalten [AMF06, MF04] in einem spaltenorientierten Datenbanksystem. Zum anderen existieren Arbeiten [HRSD07, RS06], die Kompressionsverfahren auf Ebene der Datensätze für Relationen auswählen. Die Auswahl ist dabei nicht auf einzelne Kompressionsverfahren beschränkt, sondern kann sich auch auf Kombinationen verschiedener Kompressionsverfahren [LLR06, RS06], wie beispielsweise die Kombination von Lauflängenkodierung und Domain-Kodierung, erstrecken. Als Grundlage für die Auswahl eines Kompressionsverfahrens [AMF06, CGK01, HRSD07] werden klassischerweise Angaben wie die Anzahl der Ausprägungen, die durchschnittliche Anzahl an Wiederholungen, die Lokalität der Daten und der geschätzte Kompressionsmehraufwand herangezogen.

2.3 Integration von Kompression in Datenbanksysteme

Komprimierte Relationen werden in klassischen, zeilenbasierten und Festplatten-zentrischen Datenbanksystemen genau wie unkomprimierte Relationen in Seiten auf dem externen Speicher organisiert. Hierbei unterscheidet man die Ablage in unabhängig komprimierte Seiten [GRS98, PP03], bei denen alle nötigen Informationen zur Dekompression (z.B. ein Wörterbuch) innerhalb einer Seite vorhanden sind, Relationen-komprimierte Seiten [MF04] und Relationenübergreifend-komprimierte Seiten [LLR06] (z.B. mit globalen Wörterbüchern). Aufgrund der sehr hohen Kosten für den Datentransfer zwischen Externspeicher und Hauptspeicher sind auch komplexere bzw. schwergewichtige Kompressionsverfahren (wie z.B. Lempel-Ziv [AMF06] oder Huffman [HRSD07, RS06, RSQ$^+$08]) für die Kompression der auf dem Externspeicher abgelegten Daten sinnvoll.

Wie bereits angesprochen, werden Kompressionstechniken auch auf Ebene der Datenbank-Indexstrukturen eingesetzt. Hierbei werden zwei Klassen unterschieden: Zum einen existieren Indexstrukturen, die Kompression inhärent besitzen, wie z.B. Bitmap-Indizes [Joh99], bei denen ein Schlüssel je nach Anzahl der Ausprägungen durch wenige Bits ersetzt wird. Als weiteres Beispiel gelten blockbasierte B+-Bäume [Sto88], bei denen nur ein Schlüssel für eine Menge von Sätzen genutzt wird. Zum anderen existieren Indexstrukturen, bei denen Kompression explizit genutzt wird. Beispiele hierfür sind B-Bäume [BU77, Com79, GRS98, HCL+90], B+-Bäume [BU77, BLM+09] und Präfixbäume [BHF09] mit komprimierten Schlüsseln sowie komprimierte Bitmap-Indizes [Joh99, MZ92, WOS06].

Für die Anfrageausführung existieren zur Zeit drei verschiedene Strategien, die sich in Zeitpunkt und Art der Datendekompression unterscheiden und sich auf den klassischen Festplatten-zentrischen Architekturansatz beziehen. Bei *eager decompression* [IW94] werden die Daten vom externen Speicher dekomprimiert, sobald diese in die Seiten des DB-Puffermanagers geladen werden. Der DB-Puffermanager hält deshalb nur Seiten mit dekomprimierten Daten. Der Hauptvorteil dieser Strategie ist, dass die Anfrageverarbeitung (bzw. das Verarbeitungsmodell) nicht angepasst werden muss, da nur auf dekomprimierten Daten gearbeitet wird. Bei *lazy decompression* [AMF06, GRS98, Goy83, GS91, HRSD07, RS06, RHS95, WKHM00, ZHNB06] wird die Eigenschaft ausgenutzt, dass bestimmte Operationen wie beispielsweise Projektion, Selektion und Gleichverbund direkt auf komprimierten Daten durchgeführt werden können und somit die Dekompression der Daten möglichst spät bzw. erst bei Bedarf im Anfrageplan durchgeführt wird. Bei der dritten Strategie, *transient decompression* [CGK01], werden die Daten vor der Ausführung der relationalen Operatoren, die nicht auf komprimierten Daten arbeiten können, dekomprimiert; als Ausgabe der Operatoren wird jedoch die komprimierte Repräsentation genutzt, die temporär für jeden Datensatz vorgehalten wird.

3 Vision: Ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse

Unser zentrales Ziel bzw. unsere Vision ist die Untersuchung und Realisierung einer *durchgängigen* Kompression von Zwischenergebnissen zur effizienten Anfrageverarbeitung. Ohne Beschränkung der Allgemeinheit konzentrieren wir uns auf den Bereich der Hauptspeicher-zentrischen Datenbanksysteme, um eine umfassende Behandlung der Kompression in derartigen System zu etablieren. Die gewonnenen Erkenntnisse können bzw. sollen konzeptionell auf die klassischen Festplatten-zentrischen Datenbanksysteme übertragen werden.

In [CGK01] ist die Optimierung der Anfrageverarbeitung mittels komprimierter Zwischenergebnisse bereits diskutiert worden, jedoch nicht näher untersucht. Durch die immer größer werdende Schere zwischen Rechenleistung und Speicherbandbreite bei modernen Mehrkernprozessoren [QRR+08] und der Weiterentwicklung insbesondere leichtgewichtiger Kompressionsverfahren [SGL10, WPB+09, ZHNB06] verliert dieses Argument je-

doch zusehends seine Gültigkeit. Dennoch ist es gerade wegen der zusätzlichen Kosten wichtig, dass eine *Balance* zwischen den reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten hergestellt wird, um die Gesamtausführungszeiten einer Anfrage zu minimieren und damit ergibt sich unser Ziel der ausgewogenen Anfrageverarbeitung. Um dieses Gleichgewicht zu erzielen, muss nicht nur die Anfrageverarbeitung sondern auch der dazu notwendige Teil der Anfrageoptimierung adressiert werden. Ziel ist es, die Kompression von Zwischenergebnissen als parametrierbaren Bestandteil in die Anfrageoptimierung zu integrieren.

Unsere Betrachtung erstreckt sich dabei von der Untersuchung und Optimierung leichtgewichtiger Kompressionsansätze (struktureller Aspekt) über die Möglichkeit, Datenbankoperatoren direkt und soweit wie möglich auf einer komprimierten Datenform auszuführen (operationaler Aspekt) bis hin zur Integration in die Optimierungskomponente (Optimierungsaspekt), um situationsabhängig entscheiden zu können, wann und welche Verfahren für eine gegebene Datenbankanfrage eingesetzt werden sollen. Diese drei Aspekte stellen im Wesentlichen auch die wissenschaftlichen Herausforderungen dar, denen wir uns zur Zielerreichung stellen müssen. Nachfolgend sind die drei Herausforderungen etwas näher erläutert.

3.1 Struktureller Aspekt

Wie bereits in Abschnitt 2 dargelegt ist, existiert eine Vielzahl von leichtgewichtigen Kompressionsverfahren, so dass wir kein neues Verfahren entwickeln, sondern uns um effiziente Implementierungen kümmern wollen. Insbesondere wollen wir erörtern, wie die einzelnen Kompressionsverfahren parallelisiert werden können und somit von den verschiedenen Arten von Parallelität moderner Prozessoren (Pipeling, SIMD, Many-Core) profitieren. Die effiziente Umsetzung der Algorithmen ist ein wichtiger Schwerpunkt, um eine Balance zwischen reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten erzielen zu können.

Neben der Anwendung klassischer leichtgewichtiger Kompressionsverfahren beziehungsweise einer Kombination von Kompressionsverfahren ist die Transformation von komprimierten Daten ohne vorherige vollständige Dekompression im Bereich der Anfrageverarbeitung von enormer Bedeutung. Dieser Bereich ist bisher ungenügend bis kaum bearbeitet worden. Aus Sicht der Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse ist es oft sinnvoll, für die Eingabedaten und Ausgabedaten eines einzelnen Operators unterschiedliche Kompressionsverfahren einzusetzen oder zumindest unterschiedliche Parameter eines Kompressionsverfahrens zu nutzen. So ist bei einem Selektionsoperator häufig das Kompressionspotential für die Ausgabedaten höher als für die Eingabedaten, da die Anzahl der Ausprägungen in einer Spalte oder mehreren Spalten reduziert wird. Wird nun zum Beispiel Domainkodierung mit 24-bit Codewörtern für die Eingabedaten eingesetzt, dann können 16-bit Codewörter für die Ausgabedaten ausreichend sein. Die Transformation von 24-bit zu 16-bit Codewörtern sollte dabei ohne die aufwändige und vollständige Dekompression durchgeführt werden. Dies könnte zum Beispiel erreicht werden, indem Techniken (Permutation von Bytes mittels SIMD Instruktionen) aus [SGL10] eingesetzt

werden. Ähnliche Techniken auf Basis neuer Instruktionen (z.B. PEXT) in modernen Prozessoren (wie z.B. Haswell) erlauben zudem die Transformation von Codewörtern, die nicht an Byte-Grenzen (byte aligned) ausgerichtet sind oder auch Transformationen zwischen unterschiedlichen Kompressionsverfahren wie z.B. Nullwertkomprimierung und Frame-of-Reference.

3.2 Operationaler Aspekt

Neben dem bisher beschriebenen strukturellen Aspekt, ist der operationale Aspekt für eine ausgewogene Anfrageverarbeitung ebenfalls ein zentraler Baustein, da physische Planoperatoren entworfen und umgesetzt werden müssen, die als Eingabe komprimierte Daten entgegennehmen und als Ergebnis ebenfalls wiederum komprimierte Daten bereitstellen, so dass möglichst alle Planoperatoren mit komprimierten Zwischenergebnissen arbeiten können. Die Herausforderung bei dieser Aufgabe besteht darin, dafür zu sorgen, dass der Aufwand für die Integration verschiedener Kombinationen von Kompressionsverfahren möglichst gering ist. Das bedeutet, dass die Anzahl der Planoperatoren linear mit der Anzahl der integrierten Kompressionstechniken skalieren soll.

Wie in Abschnitt 2 gezeigt worden ist, existieren drei unterschiedliche Integrationsmöglichkeiten von Kompressionstechniken in Planoperatoren, die wir ebenfalls betrachten werden. Insbesondere die Integrationsvariante *transient decompression* [CGK01], bei der Daten partiell und temporär vor der Ausführung der Operationen, die nicht auf komprimierten Daten arbeiten können, dekomprimiert werden, wobei als Ausgabe der Operation jedoch die komprimierte Repräsentation genutzt wird, ist für unsere *ausgewogene Anfrageverarbeitung* enorm wichtig. Dieses Verfahren bildet die Basis für unseren Ansatz, wobei der grundlegende Unterschied jedoch darin besteht, dass Zwischenergebnisse in verschiedene Repräsentationen unterschiedlicher Kompressionsverfahren im Anfrageplan überführt werden und somit wiederholt komprimiert als auch dekomprimiert werden müssen. Dies erlaubt den Wechsel des *optimalen* Kompressionsverfahrens im Ausführungsplan in Abhängigkeit der Operatoren und der Dateneigenschaften. So kann zum Beispiel ein selektiver Operator die Anzahl der Ausprägungen einer Relation so reduzieren, dass ein Kompressionsverfahren, das für die Basisdaten ungeeignet war, für die Kompression der Zwischenergebnisse nach dem Operator gut geeignet ist. Diese Art der Anfrageausführung wurde bisher ausschließlich für die Auswertung boolescher Ausdrücke auf komprimierten Bitmap-Indizes [AYJ00] genutzt. Wir wollen es für beliebige Attributdatentypen für Zwischenergebnisse der Anfrageverarbeitung mit beliebigen Operatoren unterstützen.

Die wiederholte Kompression von Zwischenergebnissen bzw. des Operatorzustands existiert bisher nur für einzelne Datenbankoperationen, die auf externen Speicher angewiesen sind. Ein Beispiel hierfür ist externes Sortieren variabel langer Zeichenketten [YZ03, YZ07]. Dabei werden durch Kompression des Operatorzustandes (sortierte Teilbereiche) die Transferkosten innerhalb der Speicherhierarchie gesenkt und somit die Verarbeitungszeit reduziert. Allerdings beschränken sich die bisherigen Arbeiten auf diesem Gebiet nur auf die Reduktion der Transferkosten zum externen Speicher, es wird nicht

auf die Besonderheiten Hauptspeicher-zentrischer Architekturen z.B. mit Direktzugriff auf bestimmte Datenbestände geachtet; insbesondere beschränken sich die verwandten Arbeiten nur auf das Sortieren variabel langer Zeichenketten.

3.3 Optimierungsaspekt

Nach Betrachtung des strukturellen und des operationalen Aspektes kann die eigentliche Anfrageoptimierung betrachtet werden. Auf dieser Ebene müssen sowohl reduzierte Transferkosten als auch der Mehraufwand für Kompression und Dekompression bei der Suche eines optimalen Ausführungsplanes mit einbezogen werden. Hierfür existieren bereits Erweiterungen für den Anfrageoptimierer [AYJ00, CGK01], die bei der Suche nach einem optimalen Plan die Entscheidung berücksichtigen, wann und wo dekomprimiert werden soll. Auch die Wahl des Kompressionsverfahrens für Zwischenergebnisse und die Auswahl von Operatoralternativen, die auf komprimierten Daten arbeiten können, sind für die Anfrageoptimierung [AYJ00] wichtige Einflussgrößen. Bisher wurden diese Optimierungen ausschließlich für die Auswertung boolescher Ausdrücke auf komprimierten Bitmap-Indizes [AYJ00] durchgeführt. Unser Ziel ist, ein allgemeines Verarbeitungsmodell zu entwerfen, bei dem Kompression in die Anfrageverarbeitung und Anfrageoptimierung mit einbezogen wird. Deshalb sind weitere Optimierungstechniken für die komprimierungssensitive Anfrageoptimierung zu entwickeln, welche einen großen Einfluss auf Verarbeitungszeiten von Anfragen haben können. Unsere Anfrageoptimierung soll auf Basis eines Kostenmodells erfolgen, wobei dieses Kostenmodell explizites Wissen über die leichtgewichtige Kompression und Transformation hat. Dieses Wissen soll über eine empirische Evaluierung der Verfahren erlangt werden.

4 Kompressions- und Transformationsverfahren

Wie im vorherigen Abschnitt beschrieben, ist für unser Ziel der *ausgewogenen Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* der strukturelle Aspekt ein extrem wichtiger Baustein, der am Anfang betrachtet werden muss. Zur umfassenden Bearbeitung dieses Aspektes bauen wir ein Repository mit leichtgewichtigen Kompressions- und Transformationsverfahren auf, wobei Abbildung 3 einen Überblick über unseren aktuellen Stand zeigt. Wir haben uns dabei zunächst auf drei bedeutende Klassen von Kompressionsverfahren konzentriert:

1. *Run Length Encoding (RLE)* mit den Algorithmen *RleSeq* und *RleSimd*
2. *Dictionary Encoding (Dict)* mit den Algorithmen *SimpleDictSeq* und *AnalyzingDictSeq*, wobei AnalyzingDict als OrderPreservingDict bezeichnet werden kann
3. *NullSuppression (NS)* mit den Algorithmen *4-Wise Null Suppression* und *4-Gamma Coding* [SGL10].

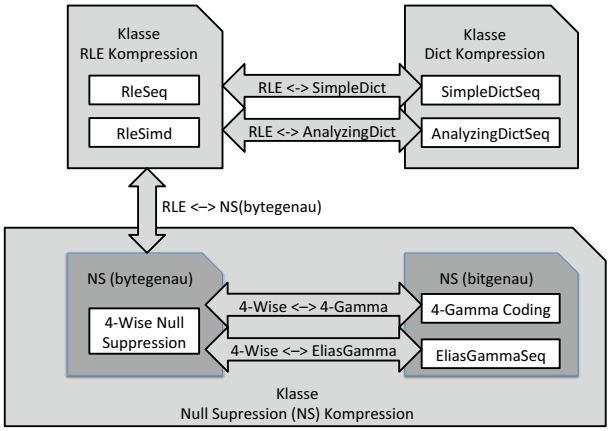


Abbildung 3: Überblick über den aktuellen Stand unserer Arbeit. Zu sehen sind Kompressionsverfahren (weiße Kästen) eingeordnet in Klassen (graue Kästen), sowie von uns entwickelte Transformationsverfahren (Pfeile). Die in diesem Artikel näher beschriebenen Verfahren sind *RleSimd* und die Transformation der Daten aus dem Kompressionsschema *AnalyzingDict* in das Schema *RLE*.

Bei den Verfahren, deren Name mit *„Seq“* endet, handelt es sich um sequenzielle Algorithmen. Analog dazu deutet das Suffix *„Simd“* auf einen mit Hilfe von SIMD-Instruktionen parallelisierten Algorithmus hin. Zwischen den betrachteten Kompressionsverfahren haben wir bereits mehrere Transformationsverfahren entwickelt, wie es in Abbildung 3 dargestellt ist. Dabei ist zwischen allen bisher betrachteten Paaren von Kompressionsverfahren eine Transformation in beide Richtungen möglich. Im Folgenden sollen exemplarisch unsere effiziente Implementierung von *RleSimd* sowie das Transformationsverfahren *Rle2AnalyzingDict* näher vorgestellt werden.

4.1 RLE-Kompression

Die grundlegende Idee von Run Length Encoding (RLE) besteht darin, die unkomprimierten Eingabedaten in *Läufe* einzuteilen. Ein Lauf ist eine ununterbrochene Aufeinanderfolge gleicher Werte, die durch einen Laufwert und eine Lauflänge charakterisiert ist. Jeder Lauf gibt somit an, welcher Wert wie oft wiederholt wird. Im Folgenden beschreiben wir das Format der komprimierten Daten sowie unseren vektorisierten Algorithmus *RleSimd*.

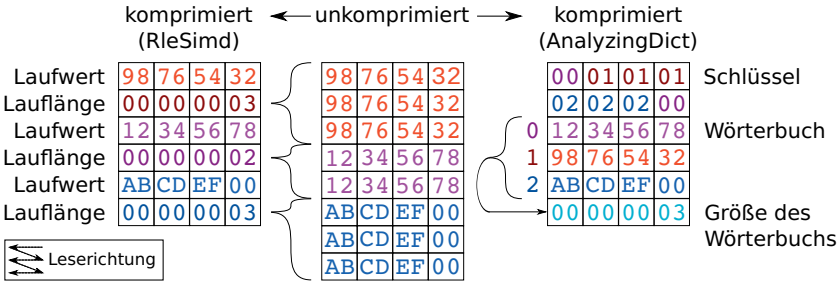


Abbildung 4: Eine beispielhafte Darstellung von unkomprimierten Daten und zugehörigen komprimierten Daten der Kompressionsverfahren RleSimd (links) und AnalyzingDict (rechts). Die Darstellung der Werte erfolgt in hexadezimaler Notation. Speicheradressen wachsen von rechts nach links und von oben nach unten, somit steht bei den unkomprimierten Daten in jeder Zeile ein 32-Bit-Integer im Little-Endian-Format

Komprimiertes Datenformat von RleSimd

Die mittels RleSimd komprimierten Daten sind eine Folge von 32-Bit-Integers, wie ebenfalls beim sequenziellen Verfahren. An allen geradzahligen Positionen (beginnend mit 0) befindet sich ein Laufwert, an allen ungeradzahligen Positionen eine Lauflänge. Die Reihenfolge der Läufe in den Originaldaten bleibt in den komprimierten Daten erhalten. Die linke Seite der Abbildung 4 zeigt ein Beispiel für mit RleSimd komprimierte Daten.

Algorithmus RleSimd

Kompression: Ein sequenzieller Algorithmus für RLE ist trivial implementierbar. Um jedoch die Performanz der Kompression zu optimieren, ist es notwendig, von den Möglichkeiten moderner Prozessoren bewusst Gebrauch zu machen. Dabei stellt insbesondere die Vektorisierung mittels SIMD-Instruktionen eine vielversprechende Möglichkeit dar, wobei wir intrinsische Funktionen in der Implementierung verwenden, um SIMD-Instruktionen zu nutzen. Unser vektorisierter Algorithmus übertrifft klar die Kompressionsgeschwindigkeit einer sequenziellen Implementierung, wie wir in Abschnitt 5 zeigen.

Am Anfang des vektorisierten Algorithmus' steht der Eingabezeiger auf dem erstem Wert der unkomprimierten Eingabedaten. Der Algorithmus geht in den folgenden Schritten vor (an Abbildung 5 können ausgewählte Schritte des Algorithmus nachvollzogen werden):

(a) Aktuellen Eingabewert (32-Bit-Integer) laden. Dieser ist der aktuelle Laufwert und wird mittels der Funktion `_mm_set1_epi32()` auf die vier 32-Bit-Elemente eines 128-Bit-Vektorregisters verteilt (Abbildung 5 a). Wir gehen im Moment von einer Größe des Vektorregisters von 128-Bit. Einer Vergrößerung ist jederzeit möglich.

(b) Die nächsten vier Werte der Eingabedaten mittels der Funktion `_mm_loadu_si128()` in ein zweites Vektorregister laden (Abbildung 5 b).

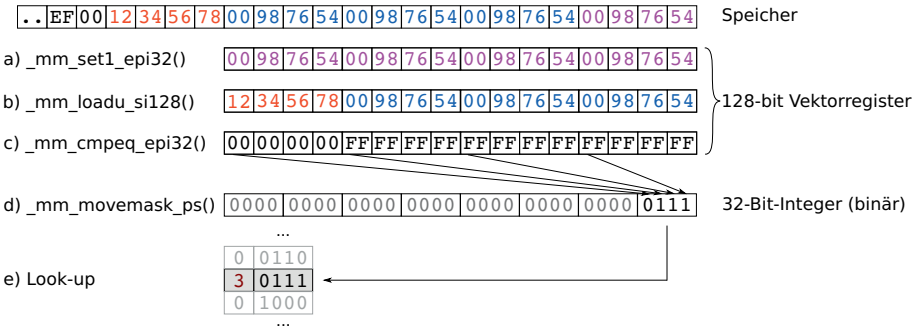


Abbildung 5: Darstellung ausgewählter Schritte des Kompressionsalgorithmus RleSimd an einem Beispiel. Alle Werte außer dem letzten sind in hexadezimaler Notation dargestellt. Adressen wachsen von rechts nach links. Oben: Ein Auszug des Speichers, der den Beginn der unkomprimierten Eingabedaten zeigt. Darunter: Registerinhalte nach Zwischenschritten des Algorithmus.

(c) Diese vier Werte mittels der Funktion `_mm_cmpeq_epi32()` parallel mit dem aktuellen Laufwert vergleichen (Abbildung 5 c). Das Ergebnis des parallelen Vergleichs liegt nun in einem Vektorregister vor. In jedem 32-Bit-Element dieses Vektorregisters sind entweder alle Bits gesetzt oder alle Bits nicht gesetzt, je nachdem, ob die korrespondierenden Elemente gleich waren oder nicht.

(d) Um dieses Ergebnis nutzen zu können, werden die höchstwertigen Bits jedes 32-Bit-Elements extrahiert und zu einer 4-Bit-Maske zusammengefügt (Abbildung 5 d). Dazu verwenden wir die Funktion `_mm_movemask_ps()`. Diese interpretiert die Elemente eines Vektorregisters jedoch als Single-Precision-Gleitkommazahlen, weshalb zunächst eine Typkonvertierung mittels `_mm_castsi128_ps()` erforderlich ist. Diese erzeugt allerdings keine Maschineninstruktionen und wirkt sich somit nicht auf die Performanz aus.

(e) Die entstandene Maske wird genutzt, um in einer zuvor erstellten Tabelle nachzuschlagen, um wie viele Elemente der aktuelle Lauf fortgesetzt wird.

(f) Falls der Lauf um weitere vier Elemente fortgesetzt wird, so ist das Ende des Laufs noch nicht bekannt. Der Eingabezeiger wird um vier Elemente weiter geschoben. Die Abarbeitung fährt mit Schritt (b) fort.
Falls der Lauf um weniger als vier Elemente fortgesetzt wird, so ist das Ende des Laufs bekannt. Der Laufwert und die Lauflänge werden an die Ausgabedaten angehängt. Der Eingabezeiger wird um so viele Elemente weiter geschoben, wie der aktuelle Lauf soeben fortgesetzt wurde. Damit steht er nun auf dem ersten Element des nächsten Laufs. Die Abarbeitung fährt mit Schritt (a) fort.

Kurz vor Erreichen des Endes der Eingabedaten sind unter Umständen nicht mehr ausreichend viele (d.h. weniger als vier) Eingabewerte für eine parallele Verarbeitung übrig. Diese restlichen Werte werden dann mit einem trivialen sequenziellen Algorithmus verarbeitet.

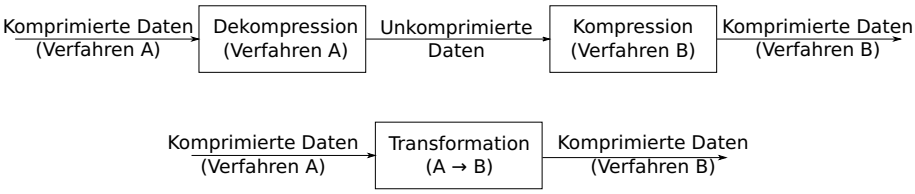


Abbildung 6: Oben: Klassischer Ansatz für einen Wechsel des Kompressionsverfahrens: Vollständige Dekompression gefolgt von vollständiger Rekompression. Unten: Unser Ansatz: Direkte Transformation ohne zwischenzeitliche vollständige Dekompression.

Initialisierung: In der Initialisierungsphase wird einmalig die Lookup-Tabelle für Schritt (e) des Kompressionsalgorithmus' aufgestellt. Diese Tabelle enthält 16 Einträge und ist mit den 4-Bit-Masken indiziert. Die Fortsetzungslänge zu einer Maske `m` ist die Anzahl der *„trailing ones"* in `m`. Diese kann berechnet werden als `table[m] = __builtin_ctz(~m)`, d.h. als die Anzahl der abschließenden Nullen der negierten Maske.

Dekompression: Für die Dekompression kommt ein einfacher, rein sequenzieller Algorithmus zum Einsatz: Bis das Ende der komprimierten Eingabedaten erreicht ist, werden jeweils Laufwert und Lauflänge des nächsten Laufs gelesen und der Laufwert entsprechend oft an das Ende der Ausgabedaten angehängt.

4.2 Transformation von RLE zu AnalyzingDict

Im Folgenden widmen wir uns der Transformation von komprimierten Daten eines Kompressionsverfahrens *A* in die eines anderen Verfahrens *B*. Im klassischen Fall ist dafür eine vollständige Dekompression gemäß Algorithmus *A* gefolgt von einer vollständigen Kompression gemäß Algorithmus *B* erforderlich. Unser Ziel besteht darin, die Transformation ohne zwischenzeitliche vollständige Dekompression der Daten zu ermöglichen. Stattdessen schlagen wir eine eng verzahnte De- und Rekompression vor, wie es beim *transient decompression*-Ansatz [CGK01] bereits für Operatoren gemacht wird. Idealerweise kann dabei ein Schreiben von unkomprimierten Daten in den Speicher vermieden werden. Abbildung 6 verdeutlicht den Unterschied zwischen beiden Alternativen.

Wir stellen *Rle2AnalyzingDict* als ein Beispiel, der von uns bereits konzipierten Transformationsverfahren (siehe Abbildung 3), vor. Dieses Verfahren erwartet Eingabedaten im komprimierten Format von RleSimd (siehe Abschnitt 4.1). Die Ausgabendaten haben das komprimierte Format des Kompressionsalgorithmus' *AnalyzingDictSeq*, welcher nun auch kurz vorgestellt wird.

Komprimiertes Datenformat von AnalyzingDict

Bei AnalyzingDict handelt es sich um ein wörterbuchbasiertes Verfahren. Das heißt, jedem unterschiedlichen Wert der unkomprimierten Daten wird ein Schlüssel zugeordnet, welcher der Position des zugehörigen Wertes im Wörterbuch entspricht. Die komprimierten Daten von AnalyzingDict lassen sich in zwei Abschnitte gliedern: (i) eine Folge von Wörterbuchschlüsseln (ein Schlüssel pro Originalwert) und (ii) das Wörterbuch selbst.

Schlüssel werden mit 0 beginnend fortlaufend vergeben, so dass sie bei d unterschiedlichen Werten im Intervall $[0, d-1]$ liegen. Beim Speichern der Schlüssel werden führende Null-Bytes weggelassen. Jedoch werden alle Schlüssel mit derselben Anzahl Bytes gespeichert. Bei d unterschiedlichen Werten ergeben sich $4 - (\_\_builtin\_clz((d-1)|1)/8)$ Bytes pro Schlüssel.

Das Wörterbuch enthält alle unterschiedlichen Werte der unkomprimierten Eingabedaten in aufsteigender Reihenfolge sowie die Anzahl der unterschiedlichen Werte als 32-Bit-Integers. Wir erwarten, dass die dadurch entstehende reihenfolgeerhaltende Kompression das Arbeiten auf den komprimierten Daten besonders begünstigt. Ein Beispiel für in diesem Format komprimierte Daten findet sich in der rechten Seite der Abbildung 4.

Kompressionsverfahren AnalyzingDictSeq

Der Kompressionsalgorithmus ist sequenziell und läuft in zwei Phasen ab. In der ersten Phase wird das Wörterbuch aufgestellt. Dazu wird einmal komplett über die Eingabedaten iteriert und jeder bis dahin nicht gesehene Wert dem Wörterbuch hinzugefügt. Anschließend wird das Wörterbuch sortiert und die Anzahl der notwendigen Bytes pro Schlüssel ermittelt. Während der Kompression dient das Wörterbuch dem Nachschlagen des Schlüssels zu einem Wert.

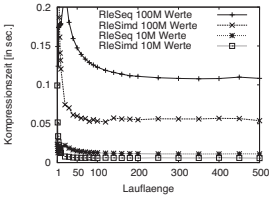
In der zweiten Phase findet die eigentliche Kompression der Eingabedaten statt. Nun wird erneut über die Eingabedaten iteriert. Dabei wird zu jedem Wert der zugehörige Schlüssel im Wörterbuch nachgeschlagen und mit der festgelegten Anzahl Bytes den Ausgabedaten angehängt. Zuletzt werden das Wörterbuch und dessen Größe an das Ende der Ausgabe geschrieben.

Im Zuge der Dekompression werden die Schlüssel wieder durch die zugehörigen Werte ersetzt. Wir verzichten an dieser Stelle auf eine ausführliche Beschreibung des Algorithmus', da dieser für die Betrachtung von Rle2AnalyzingDict nicht relevant ist.

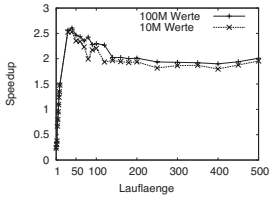
Transformationsverfahren Rle2AnalyzingDict

Der Transformationsalgorithmus *Rle2AnalyzingDict* ist weitgehend identisch mit dem Kompressionsalgorithmus von *AnalyzingDictSeq*. Der Ablauf gliedert sich in dieselben zwei Phasen, wobei hier auf eine vollständige Dekompression der Daten verzichtet werden kann.

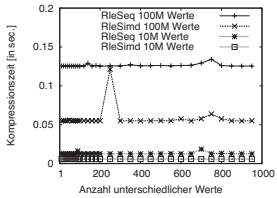
Bei der Erstellung des Wörterbuchs kann nun ein erheblicher Teil der Laufzeit eingespart werden. Während beim Arbeiten auf unkomprimierten Daten für *jeden Wert* geprüft wer-



(a) Lauflänge



(b) Speedup - Lauflänge



(c) Unterschiedliche Werte

Abbildung 7: Kompressionszeiten von *RleSeq* und *RleSimd* sowie Speedup von *RleSimd* gegenüber *RleSeq*. Die Ausgangsdaten in (a) und (b) setzen sich aus 10M bzw. 100M unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

den muss, ob dieser bereits im Wörterbuch enthalten ist, muss dies beim Arbeiten auf RLE-komprimierten Daten nur für *einen Wert pro Lauf* durchgeführt werden. Zudem müssen nur die Laufwerte betrachtet werden und die Lauflängen können übersprungen werden.

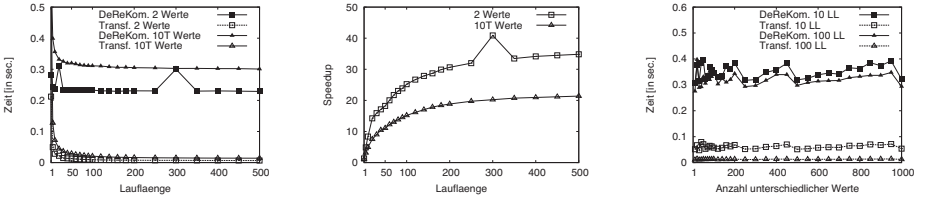
Auch die zweite Phase ist nun weniger aufwendig. Statt für *jeden Wert* der Eingabedaten den zugehörigen Schlüssel nachzuschlagen, muss dies nun wieder nur für *einen Wert pro Lauf* getan werden. Der erhaltene Schlüssel wird dann entsprechend der Lauflänge mehrmals an die Ausgabedaten angehängt.

5 Evaluierung

Unsere Algorithmen sind in C++ implementiert und benutzen SIMD-Instruktionen aus der Erweiterung SSE2 mittels intrinsischer Funktionen. Als Evaluierungssystem nutzen wir eine Intel Core i3-2350M CPU mit 4GB Hauptspeicher, wobei die Algorithmen mit gcc 4.7 und dem Optimierungsflag `-O3` kompiliert sind. Für die Experimente haben wir einen eigenen Datengenerator geschrieben, um 32-bit Integer-Werte mit vorgegebener Anzahl unterschiedlicher Werte und Lauflängen zu erzeugen. Die Verteilung der Daten erfolgt dabei gleichmäßig.

5.1 Sequenzielle vs. parallele RLE-Kompression

Als Erstes vergleichen wir die sequenzielle versus unsere SIMD-fähige RLE-Kompression. In Abbildung 7 ist dabei Folgendes zu erkennen: (1) dass sowohl für *RleSeq* als auch für *RleSimd* die benötigte Zeit mit steigender Lauflänge sinkt, (2) dass die Kompressionszeiten beider Algorithmen unabhängig von der Anzahl der unterschiedlichen Werte sind und (3) dass *RleSimd* unabhängig von der Größe der Ausgangsdaten ab einer Lauflänge von circa 8 schneller komprimiert als RleSeq. Die Speedups der Kompression mit *RleSimd* gegenüber der mit *RleSeq* sind in Abbildung 7 (b) dargestellt. Wie zu erwarten war, ist die parallele Variante effizienter als die sequenzielle Implementierung. Die Opti-



(a) Lauflänge (b) Speedup - Lauflänge (c) Unterschiedlicher Werte

Abbildung 8: Laufzeiten der Transformation (Transf.) und der vollständigen De- und Rekompression (DeReKom.) sowie Speedups der Transformation gegenüber der vollständigen De- und Rekompression. Die Ausgangsdaten in (a) und (b) setzen sich aus 2 bzw. 10.000 unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge (LL = Lauflänge) 10 bzw. 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

mierung der leichtgewichtigen Kompressionsverfahren durch Parallelisierung kann einen erheblichen Performanzgewinn bringen, der gerade mit dem Blick auf unsere anvisierte *ausgewogene Anfrageverarbeitung* in Datenbanksystemen besonders attraktiv ist.

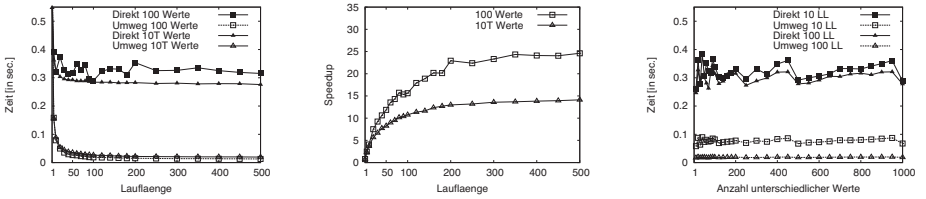
5.2 Transformation vs. vollständige De- und Rekompression

Als Nächstes widmen wir uns dem Wechsel des Kompressionsschemas, am Beispiel des Wechsels von RLE zu AnalyzingDict. Wir vergleichen die dafür benötigte Zeit bei Nutzung von (1) der direkten Transformation mit dem Algorithmus *Rle2AnalyzingDict* und (2) der vollständigen Dekompression mit *RleSimd* und Rekompression mit *AnalyzingDictSeq*. In Abbildung 8 ist klar zu erkennen, dass die direkte Transformation unabhängig von der Lauflänge und der Anzahl unterschiedlicher Werte deutlich weniger Zeit benötigt als die vollständige De- und Rekompression. Die erreichten Speedups der Transformation gegenüber der vollständigen De- und Rekompression sind in Abbildung 8 (b) dargestellt.

Wie aus diesem Experiment ersichtlich wird, ist der Transformationsansatz für die Änderung des Kompressionsschemas in diesem Szenario wesentlich effizienter als die vollständige Dekompression gefolgt von einer vollständigen Kompression. Für uns ist das ein Indikator dafür, dass derartige Transformationsverfahren sinnvoll sind und für die Anfrageverarbeitung betrachtet werden müssen. Dieser operationale Aspekt (siehe Abschnitt 3.2) wird als Nächstes von uns untersucht.

5.3 Transformations-beschleunigte Kompression

Eine weitere mögliche Anwendung der Transformationsalgorithmen sehen wir in der Beschleunigung der Kompression in ein Schema *B*, indem zunächst die Daten mittels eines Verfahrens *A* komprimiert und anschließend von dort nach *B* transformiert werden. Wir



(a) Lauflänge (b) Speedup - Lauflänge (c) Unterschiedlicher Werte

Abbildung 9: Laufzeiten der direkten Kompression (Direkt) nach *AnalyzingDict* und der Kompression über den Umweg *RleSimd* sowie Speedups der Kompression über den Umweg gegenüber der direkten Kompression. Die Ausgangsdaten in (a) und (b) setzen sich aus 100 bzw. 10.000 unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge (LL = Lauflänge) 10 bzw. 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

demonstrieren die prinzipielle Machbarkeit dieser Idee am Beispiel der Kompression in das Format *AnalyzingDictSeq*. Zum Einen wird *AnalyzingDictSeq* direkt ausgeführt und zum Anderen über den „Umweg“ *RleSimd* und *Rle2AnalyzingDict*.

Die Ergebnisse sind in der Abbildung 9 dargestellt. Wie zu erkennen ist, wird für alle Lauflängen außer 1 und beliebige Anzahlen unterschiedlicher Werte die Kompression über den Umweg erheblich schneller als auf dem direkten Weg. Die erreichten Speedups der Kompression über den Umweg gegenüber der direkten Kompression sind in der Abbildung 9(b) illustriert.

6 Zusammenfassung

In Hauptspeicher-zentrischen Architekturansätzen für Datenbanksysteme müssen für die Anfrageverarbeitung sowohl die Basisrelationen als auch die Zwischenergebnisse im Hauptspeicher gehalten werden. Zudem ist der Aufwand, Zwischenergebnisse zu generieren äquivalent zum Aufwand, Änderungen an den Basisrelationen durchzuführen. Daher sollten die Zwischenergebnisse gesondert betrachtet werden, wobei sich zwei orthogonale Optimierungsansätze anbieten. Auf der einen Seite sollten Zwischenergebnisse nicht mehr z.B. durch entsprechend angepasste Code-Generierung [Neu11] oder durch den Einsatz kooperativer Operatoren [KSHL13] produziert werden. Auf der anderen Seite sollten Zwischenergebnisse (wenn sie z.B. nicht vermeidbar sind) so organisiert werden, dass eine effiziente Weiterverarbeitung ermöglicht wird. Für den letzten Bereich schlagen wir den durchgängigen Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse vor und haben unsere Vision einer *ausgewogenen Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* vorgestellt. Neben dem Überblick über unsere Vision haben wir ebenfalls die wissenschaftlichen Herausforderungen skizziert und den Stand der Technik umfassend aufgearbeitet. Darüber hinaus haben wir erste Ansätze für die Optimierung von leichtgewichtigen Kompressionsverfahren vorgestellt und erste Transformationsverfahren präsentiert, die aus Sicht der Anfrageverarbeitung eventuell sinnvoll sind, um einen

effizienten Wechsel des Kompressionsverfahrens durchzuführen. Des Weiteren hat unsere Evaluierung gezeigt, dass derartige Optimierungen einen erheblichen Speedup bringen.

Literatur

[ALM96] Gennady Antoshenkov, David B. Lomet und James Murray. Order Preserving Compression. In *ICDE Conference*, Seiten 655–663, 1996.

[AMF06] Daniel Abadi, Samuel Madden und Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, Seiten 671–682, 2006.

[AYJ00] Sihem Amer-Yahia und Theodore Johnson. Optimizing Queries on Compressed Bitmaps. In *VLDB Conference*, Seiten 329–338, 2000.

[Bas85] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. *IEEE Transactions on Software Engineering*, 11(10):1047–1058, 1985.

[BB07] Jeff Berger und Paolo Bruni. Index Compression with DB2 9 for z/OS. http://www.redbooks.ibm.com/redpapers/pdfs/redp4345.pdf, 2007.

[BCE77] Michael W. Blasgen, Richard G. Casey und Kapali P. Eswaran. An encoding method for multifield sorting and indexing. *Communications ACM*, 20(11):874–878, 1977.

[BHF09] Carsten Binnig, Stefan Hildenbrand und Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD Conference*, Seiten 283–296, 2009.

[BLM+09] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George A. Mihaila, Ken Ross, Sherman Lau, Cathy McCarthur, Zoltan Toth und Reza Sherkat. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2):1462–1473, 2009.

[BMK99] Peter A. Boncz, Stefan Manegold und Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB Conference*, Seiten 54–65, 1999.

[BU77] Rudolf Bayer und Karl Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[CGK00] Zhiyuan Chen, Johannes Gehrke und Flip Korn. Squeezing the Most Out of Relational Database Systems. In *ICDE Conference*, Seite 81, 2000.

[CGK01] Zhiyuan Chen, Johannes Gehrke und Flip Korn. Query optimization in compressed database systems. *SIGMOD Record*, 30(2):271–282, 2001.

[CIW84] John G. Cleary, Ian und Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Goy83] Pankaj Goyal. Coding methods for text string search on compressed databases. *Inf. Syst.*, 8(3):231–233, 1983.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan und Uri Shaft. Compressing Relations and Indexes. In *ICDE Conference*, Seiten 370–379, 1998.

[GS91] G. Graefe und L.D. Shapiro. Data compression and database performance. In *Applied Computing*, Seiten 22–27, Apr 1991.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey und E. Shekita. Starburst mid-flight: as the dust clears [database project]. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):143 –160, Marz 1990.

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart und David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD Conference*, Seiten 389–400, 2007.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[IW94] Balakrishna R. Iyer und David Wilhite. Data Compression Support in Databases. In *VLDB Conference*, Seiten 695–704, 1994.

[Joh99] Theodore Johnson. Performance Measurements of Compressed Bitmap Indices. In *VLDB Conference*, Seiten 278–289, 1999.

[KSHL13] Thomas Kissinger, Benjamin Schlegel, Dirk Habich und Wolfgang Lehner. QPPT: Query Processing on Prefix Trees. In *CIDR 2013*, 2013.

[LC86] Tobin J. Lehman und Michael J. Carey. Query processing in main memory database management systems. In *SIGMOD Conference*, Seiten 239–250, 1986.

[LLR06] Thomas Legler, Wolfgang Lehner und Andrew Ross. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB Conference*, Seiten 1059–1068, 2006.

[LZ76] A Lempel und J Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[MF04] Roger MacNicol und Blaine French. Sybase IQ multiplex - designed for analytics. In *VLDB Conference*, Seiten 1227–1230, 2004.

[MLL13] Christopher M. Mullins, Lipyeow Lim und Christian A. Lang. Query-aware Compression of Join Results. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT 2013)*, Seiten 29–40, 2013.

[MZ92] Alistair Moffat und Justin Zobel. Parameterised compression for sparse bitmaps. In *SIGIR Conference*, Seiten 274–285, 1992.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[PP03] Meikel Poess und Dmitry Potapov. Data compression in Oracle. In *VLDB Conference*, Seiten 937–947, 2003.

[QRR+08] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas und Guy M. Lohman. Main-memory scan sharing for multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[Reg81] Hassan K. Reghbati. An Overview of Data Compression Techniques. *IEEE Computer*, 14(4):71–75, 1981.

[RH93] Mark A. Roth und Scott J. Van Horn. Database Compression. *SIGMOD Record*, 22(3):31–39, 1993.

[RHS95] Gautam Ray, Jayant R. Haritsa und S. Seshadri. Database Compression: A Performance Enhancement Tool. In *COMAD Conference*, 1995.

[RS06] Vijayshankar Raman und Garret Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *VLDB Conference*, Seiten 858–869, 2006.

[RSQ+08] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang und Richard Sidle. Constant-Time Query Processing. In *ICDE Conference*, Seiten 60–69, 2008.

[SDF+00] Jagannathan Srinivasan, Souripriya Das, Chuck Freiwald, Eugene Inseok Chong, Mahesh Jagannath, Aravind Yalamanchi, Ramkumar Krishnan, Anh-Tuan Tran, Samuel DeFazio und Jayanta Banerjee. Oracle8i Index-Organized Table and Its Application to New Domains. In *VLDB Conference*, Seiten 285–296, 2000.

[SGL10] Benjamin Schlegel, Rainer Gemulla und Wolfgang Lehner. Fast Integer Compression using SIMD Instructions. In *DaMoN*, 2010.

[Sto88] James A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., New York, NY, USA, 1988.

[Tro03] Andrew Trotman. Compressing Inverted Files. *Inf. Retr.*, 6(1):5–19, 2003.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer und Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[WNC87] Ian H. Witten, Radford M. Neal und John G. Cleary. Arithmetic coding for data compression. *Communications ACM*, 30(6):520–540, 1987.

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

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier und Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[YZ03] John Yiannis und Justin Zobel. External sorting with on-the-fly compression. In *BNCOD Conference*, Seiten 115–130, 2003.

[YZ07] John Yiannis und Justin Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, 2007.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes und Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE Conference*, Seite 59, 2006.

[ZIL93] A. Zandi, B. Iyer und G. Langdon. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, Seiten 330 –339, 1993.

[ZL77] Jacob Ziv und Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.