Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen

Christian Lemke[1,2] Kai-Uwe Sattler[2] Franz Färber[1]
[1]SAP AG, Walldorf, Germany
[2]FG Datenbanken & Informationssysteme, TU Ilmenau, Germany

Abstract: BI-Accelerator-Lösungen wie SAP's TREX ermöglichen durch die Kombination von spaltenorientierter Datenorganisation, hauptspeicherbasierter Verarbeitung und skalierbarer Multiserver-Architektur eine deutliche Beschleunigung bei der Verarbeitung komplexer OLAP-Anfragen in riesigen Data Warehouses. Durch den Einsatz von Datenkompressionstechniken lässt sich der Speicherbedarf der Spalten und damit auch die Verarbeitungszeit weiter reduzieren. In diesem Beitrag untersuchen wir daher Verfahren zur Spaltenkompression und deren Implementierung in TREX. Da für eine effektive Kompression eine Sortierung der Werte pro Spalte notwendig ist, stellen wir weiterhin Strategien zur Optimierung der Spaltenreihenfolge für die Sortierung vor.

1 Einführung

In den vergangenen Jahren sind die Anforderungen an Data-Warehouse- und OLAP-Technologien ständig gestiegen. Heutzutage müssen Giga-, Tera- oder Petabytes an Daten verwaltet und komplexe analytische Anfragen von mehreren hundert Nutzern gleichzeitig beantwortet werden können. Weiterhin wächst der Bedarf von Kunden nach Ad-hoc- bzw. Realtime-Auswertungen, die eine Vorberechnung von Reports unmöglich machen.

Betrachtet man klassische relationale Datenbanksysteme, die noch immer die dominierende Technologie im Backend-Bereich darstellen, so stellt der externe Speicherzugriff den limitierenden Faktor bei der Erfüllung dieser Anforderungen dar. Trotz aller Forschritte bei Festplattenspeicher klafft immer noch eine große Lücke zwischen den Zugriffszeiten im Hauptspeicher und auf Externspeicher. Zur Überwindung dieser Zugriffslücke werden aktuell drei wesentliche Ansätze verfolgt:

1. Die vom Externspeicher zu lesenden Daten werden auf das nötigste beschränkt und so schnell wie möglich gelesen. Neben geeigneten Indexstrukturen besitzen insbesondere spaltenorientierte Speicherorganisationsformen (so genannte *Column-Stores*) großes Potential [SAB+05, BZN05].

2. Durch das Halten und Verarbeiten der Daten im *Hauptspeicher* lassen sich I/O-Zugriffe komplett vermeiden [DKO+84].

3. Schließlich kann die Verarbeitungskapazität einer großen Zahl günstiger Serversysteme (Blades) durch *parallele Verarbeitung* der Anfragen ausgenutzt werden.

Im Bereich von Business-Intelligence-Technologien werden diese Ideen gegenwärtig im Rahmen so genannter BI-Accelerator-Lösungen kombiniert. Ein Beispiel hierfür ist SAP's NetWeaver BI Accelerator TREX [LLR06]. TREX basiert auf einer skalierbaren Multiserver-Architektur aus kostengünstigen Blades. Die Verarbeitung findet komplett im Hauptspeicher statt, wobei Fakten- und Dimensionstabellen spaltenorientiert organisiert (vertikal partitioniert) und die Spalten wiederum horizontal über die einzelnen Knoten partitioniert sind. Ungenutzte Spalten werden mittels einer LRU-Strategie aus dem Speicher verdrängt und können ggf. später wieder vom Externspeicher nachgeladen werden. Die Hauptspeicher- und parallele Verarbeitung erlaubt die interaktive Beantwortung von OLAP-Anfragen ohne Voraggregation.

Ein einfache Verwaltung der Daten im Hauptspeicher stößt jedoch speziell bei sehr großen Warehouse-Installationen schnell an ihre Grenzen, da RAM im Vergleich zu Festplattenspeicher teuer ist und nicht beliebig vergrößert werden kann. Daher werden geeignete Kompressionstechniken benötigt, die das Datenvolumen verringern und damit auch die Ausnutzung von CPU-Caches verbessern.

In diesem Beitrag diskutieren wir geeignete Kompressionstechniken für spaltenorientierte Speicherorganisation im Kontext des BI-Accelerators TREX. Da eine effektive Komprimierung die Sortierung der Spalten erfordert, hat die Reihenfolge der Spaltensortierung einen großen Einfluss auf die erzielbare Kompressionsrate. Wir untersuchen daher weiterhin das Problem der optimalen Spaltenreihenfolge und diskutieren geeignete Heuristiken. Die vorgestellten Techniken werden anschließend einer Evaluierung unterzogen.

2 Verwandte Arbeiten

Zur Speicherung und Organisation von Daten in Datenbanken existieren zwei grundlegende Konzepte: zeilenorientierte und spaltenorientierte Datenbanken (Row-Stores und Column-Stores). In letzter Zeit wurden mehrere Untersuchungen [HLAM06, HBND06, AMH08, HD08] durchgeführt, die beide Ansätze unter Berücksichtigung verschiedener Kriterien vergleichen und für das jeweilige Einsatzszenario Empfehlungen geben. Wird beispielsweise nur ein sehr kleiner Teil der vorhandenen Attribute in einer Anfrage verwendet und ändern sich die Daten relativ selten, dann ist ein Column-Store vorzuziehen, weil hier nur die benötigten Spalten geladen und verarbeitet werden müssen. Zwei bekannte Vertreter spaltenorientierter Datenbanken sind C-Store [SAB+05] und MonetDB/X100 [BZN05]. Im Gegensatz zu [HLAM06, HD08], wo die beiden Organisationsformen nur auf der Ebene des Plattenzugriffs und nicht hinsichtlich angepasster Operatoren verglichen werden, konzentriert sich [AMH08] auf die Anfrageausführung. Dahingehend wird in der Arbeit versucht, einen Column-Store unter Zuhilfenahme unterschiedlicher Herangehensweisen in einem Row-Store zu emulieren und herauszufinden, welche spaltenorientierten Optimierungen den größten Einfluss auf die Geschwindigkeit haben. Einen anderen Ansatz verfolgen [HBND06], die einen Row-Store auf lesenden Zugriff optimieren und in einem gemeinsamen Framework einer spaltenorientierten Datenbank gegenüberstellen.

Um auch mit großen Datenmengen effizient umgehen zu können, ist es notwendig die

Daten zu komprimieren und auf den komprimierten Daten zu arbeiten. Der Einsatz von Kompression in Datenbanken ist schon seit Anfang an Gegenstand der Forschung, wobei das Ziel früher Arbeiten [Als75, Cor85] hauptsächlich die Reduzierung des Speicherplatzbedarfs und des Datentransfers war. Spätere Veröffentlichungen [GS91, WKHM00] konzentrierten sich mehr auf die höhere Geschwindigkeit, die sich durch das Arbeiten auf komprimierten Daten erreichen lässt. So müssen bei Operationen wie Verbund oder Selektion nur noch kleinere Datenmengen verglichen werden und durch den zusätzlich verfügbaren Speicher können mehr häufig genutzte Seiten im Zwischenspeicher vorgehalten werden.

Neben vielen Arbeiten [Rv93, GRS98, CGK01, RS06, HRSD07, RSQ+08, SAB+05, ZHNB06] , die sich mit der Entwicklung neuer und der Zusammenfassung vorhandener Kompressionstechniken befassen, gibt es auch einige Veröffentlichungen [IW94, WKHM00, CGK01, AMF06], die auf das Problem der Integration von Kompression in Datenbanken eingehen. Obwohl die meisten neueren Publikationen [WKHM00, CGK01, SAB+05, AMF06, BZN05, ZHNB06] nur leichtgewichtige Kompressionstechniken vorschlagen, weil sonst die Kosten für die Dekompression den Gewinn durch den schnelleren Datentransfer übersteigen würden, setzen [RS06, HRSD07, RSQ+08] auf eine modifizierte Huffman-Kodierung [Huf52]. Sie zeigen, dass trotz einer schwergewichtigen Kodierung eine effiziente Anfragebearbeitung möglich ist.

Ein weiterer Aspekt, in dem sich die Techniken unterscheiden, besteht in der Granularität der Dekompression. Während erste Implementierungen in kommerziellen, relationalen Datenbanksystemen nur ganze Seiten entpacken konnten, was bei vielen ungleichmäßig verteilten Einzelzugriffen auf mehrere Seiten einen sehr großen Overhead darstellt, untersuchten [Cor85, IW94, GRS98] bereits die Vorteile einer zeilenweisen Dekompression. OLAP-Anfragen verwenden jedoch häufig nur einen kleinen Teil der Spalten, wodurch es nötig wird, auch effizient auf einzelne Werte zugreifen zu können. Da auch normale Datenbankanfragen davon profitieren, empfehlen besonders aktuelle Arbeiten [Als75, GS91, GRS98, WKHM00, CGK01] Kompressionstechniken, die einen feingranularen Zugriff unterstützen. Dazu gehören auch die neueren Publikationen [SAB+05, AMF06, ZHNB06, Aba07], die spezielle Kompressionsverfahren für eine spaltenorientierte Organisation präsentieren und teilweise auch deren Integration beschreiben. Neben der Kodierung von NULL-Werten, die abhängig von der Häufigkeit erfolgt, schlägt [Aba07] weitere Anwendungsszenarios für Column-Stores vor, die über Data Warehousing und OLAP hinausgehen.

Für eine optimale Kompression in einem Column-Store spielt abgesehen von der Kompressionstechnik auch die Reihenfolge der Zeilen eine große Rolle. [OR86] ist eine frühe Arbeit, die sich mit dem Umsortieren von Daten beschäftigt, um eine bessere Kompressionsrate zu erzielen. Der Fokus liegt hier jedoch auf einer zeilenorientierten Organisation und auf der Kompression von Nullwerten. Wir hingegen gehen davon aus, dass es nicht nur einen, sondern mehrere häufige Werte in einer Spalte geben kann, die optimal angeordnet werden müssen. Im Kontext von Hybrid OLAP (HOLAP), einer Kombination von relationalem OLAP (ROLAP) und multidimensionalem OLAP (MOLAP), stellt [KL03] einige Heuristiken vor, um eine effiziente Organisation der Daten zu erzielen. In einem HOLAP System werden dichte Regionen in einem mehrdimensionalem Array gespeichert,

Dictionary

pos	value
0	Aachen
1	Karlsruhe
2	Leipzig
3	Münster

IndexVector

pos	value
0	0
1	0
2	0
3	1
4	0
5	0
6	2
7	3

Abbildung 1: Beispiel für das *domain coding*.

IndexVector (uncompressed)

pos	value
0	0
1	0
2	0
3	1
4	0
5	0
6	2
7	3

IndexVector (indirect coded, values)

pos	value
0/1/2	0
3	1
4	0
5	0
6	2
7	3

IndexVector (indirect coded, offsets) / Integer (start pos)

pos	value	pos
0	0	0
1	0	
2	0	
3	1	
0		2
1		
2		
3		

Abbildung 2: Beispiel für das *indirect coding* mit einer Blockgröße von 4 Werten.

um schnell auf sie zugreifen zu können, und Daten aus dünn besetzten Regionen in einem relationalen Schema abgelegt, um Speicherplatz zu sparen.

3 Techniken zur Spaltenkompression

Wie bereits erwähnt, liegt der Fokus dieser Arbeit auf leichtgewichtigen Kompressionstechniken für Column-Stores und deren Implementierung in SAP's TREX zur weitergehenden Evaluierung im Abschnitt 5. Die vorgestellten Verfahren erlauben den effizienten Zugriff sowohl auf einzelne Werte als auch auf Blöcke von Werten, wobei Letzteres durch optimierte Algorithmen besonders schnell durchgeführt werden kann. Als Basis für alle folgenden Techniken wird das *domain coding* [WKHM00, AMF06, ZHNB06] verwendet, bei dem die in einer Spalte vorkommenden Werte in einem Wörterbuch (Dictionary) sortiert abgelegt und dann nur noch bit-komprimierte Verweise (IndexVector) gespeichert werden. Dadurch benötigt die Kodierung mit minimalen Bitlängen von n Ganzzahlen und u unterschiedlichen Werten nur $n\lceil \log_2 u \rceil$ Bits . Abbildung 1 zeigt das domain coding anhand der Beispieldaten *'Aachen, Aachen, Aachen, Karlsruhe, Aachen, Aachen, Leipzig, Münster'*, wobei jeder Wert im IndexVector durch zwei Bits dargestellt wird und die kursiv dargestellten Spalten nur logisch vorhanden sind, da sie sich implizit durch den Kontext ergeben. Das Verwenden von Ganzzahlen statt den Originalwerten bringt darüber hinaus auch Geschwindigkeitsvorteile, weil das zu verarbeitende Datenvolumen kleiner ist und die Prozessoren auf diesen Datentyp optimiert sind. So passen beispielsweise mehr Daten in den viel schnelleren Cache und mehrere Werte können mit speziellen Prozessorbefehlen parallel bearbeitet werden (SIMD-Prinzip).

Die einfachste Kompressionstechnik ist *prefix coding*, bei der gleiche Werte am Anfang (Präfix p) weggelassen und stattdessen ein Wert und die Häufigkeit gespeichert werden. Für eine Spalte mit n Elementen und u_{col} unterschiedlichen Werten sind auf diese Weise $(n-p)\lceil \log_2 u_{col} \rceil + 64$ Bits erforderlich. In Abbildung 3 sieht man links den IndexVector der Beispieldaten unkomprimiert und rechts nach der Anwendung des prefix codings.

Wenn nun der häufigste Wert nicht nur am Anfang steht, sondern auch unregelmäßig zwischen den anderen Werten sehr oft vorkommt, dann lässt sich durch das *sparse co-*

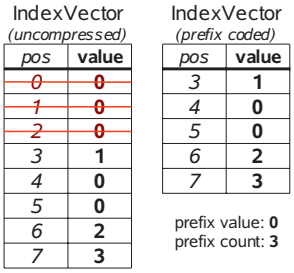

Abbildung 3: Example for prefix coding.

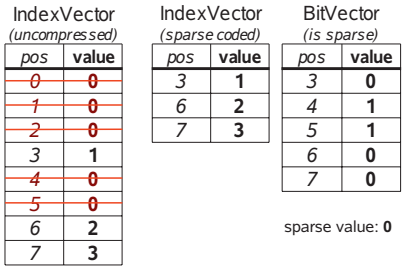

Abbildung 4: Example for sparse coding.

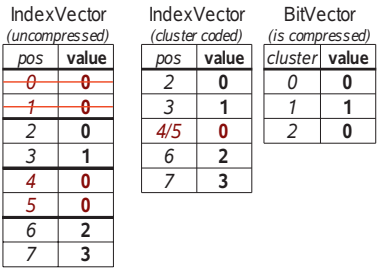

Abbildung 5: Example for cluster coding with a block size of two values.

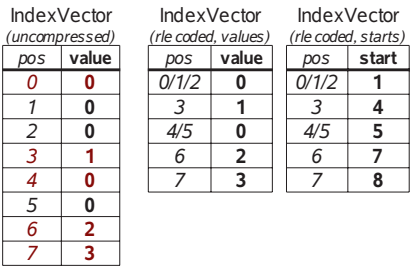

Abbildung 6: Example for run length coding.

ding (Abbildung 4) eine gute Kompression erreichen. Hier werden alle Vorkommen f des häufigsten Wertes entfernt und in einem Bitvektor die Positionen gespeichert. Weiterhin ist es möglich, für den Bitvektor das prefix coding anzuwenden, wodurch bei einem großen Präfix p noch zusätzlich viel eingespart werden kann. Bei dieser Technik beträgt der Speicherbedarf $(n-f)\lceil \log_2 u_{col} \rceil + (n-p) + 32$ Bits.

Alle im Folgenden beschriebenen Verfahren nutzen das Prinzip vom prefix coding und arbeiten auf Böcken von Daten, in denen möglichst wenig verschiedene Werte vorkommen dürfen, um eine gute Kompressionsrate zu erzielen. So werden beim *cluster coding* ausschließlich Blöcke mit einem unterschiedlichen Wert komprimiert, indem nur der auftretende Wert gespeichert wird. Weiterhin muss man sich in einem Bitvektor merken, welche Blöcke komprimiert wurden, um das Original wieder herstellen zu können. Die Bestimmung der optimalen Blöckgröße und deren Einfluss auf die Kompressionsrate und Geschwindigkeit wird in diesem Beitrag aus Zeigründen nicht weiter untersucht. Die Anzahl der Elemente sollte auf jeden Fall eine Zweierpotenz sein, weil dann statt Multiplikation und Modulo die schnellere Bitverschiebung und -verknüpfung verwendet werden kann. Abbildung 5 zeigt ein Beispiel, in dem ein Block aus zwei Werten besteht und die komprimierten Elemente dunkelrot dargestellt sind.

Enthalten die Datenblöcke mehr als einen aber trotzdem nur wenig unterschiedliche Werte, dann kommt *indirect coding* zum Einsatz. Hier wird für lohnenswerte Blöcke das *domain coding* verwendet, was eine weitere Indirektionsstufe einführt und für jeden Block ein ei-

genes Mini-Wörterbuch erforderlich macht. Um die Anzahl der Wörterbücher und somit den Speicherbedarf zu reduzieren, können aufeinanderfolgende Blöcke ein Wörterbuch weiter benutzen, falls sie durch neue Einträge nicht die Anzahl der zur Repräsentation benötigten Bits erhöhen. Bei einer Spalte mit u_{col} unterschiedlichen Werten wird ein Block mit k Einträgen und u_{block} unterschiedlichen Werten genau dann als lohnenswert bezeichnet, wenn die Größe des Wörterbuches und der Referenzen kleiner ist als die unkomprimierten Daten (nur *domain coding*):

$$u_{block}\lceil \log_2 u_{col} \rceil + k\lceil \log_2 u_{block} \rceil < k\lceil \log_2 u_{col} \rceil$$

Die in der Implementierung verwendeten Datenstrukturen sind anhand eines Beispiels in Abbildung 2 veranschaulicht, wobei ein Block aus vier Werten besteht und die komprimierten Elemente dunkelrot dargestellt sind. Die Wörterbücher und die unkomprimierten Daten werden in dem mittleren IndexVector (*values*) gespeichert und pro Block über einen Startposition (*start pos*) adressiert. Komprimierte Blöcken besitzen in der rechten Datenstruktur zusätzlich einen IndexVector (*offsets*), der die Verweise auf die einzelnen Werte enthält.

Die letzte hier vorgestellte Kompressionstechnik *run length coding* ist eine leicht modifizierte Variante vom *run-length encoding* [Gol66], die gleiche aufeinanderfolgende Werte zusammenfasst, indem sie nur noch einen Wert und die Anzahl des Auftretens speichert. Wenn nun zu einem Wert die Positionen des Auftretens ermittelt werden sollen, wäre es notwendig, die Häufigkeiten der vorhergehenden Werte aufzusummieren. Da dies jedoch recht aufwendig werden kann, haben wir uns entschlossen, auf Kosten der Kompressionsrate die Startposition und nicht die Anzahl zu merken. Für die Beispieldaten ist dies in Abbildung 6 illustriert.

In der folgenden Tabelle wird noch einmal zusammengefasst, in welchen Fällen welche Verfahren eingesetzt werden können:

Fall (Werteverteilung)	**Kompressionstechniken**
ein sehr häufiger Wert *am Anfang*	prefix coding
ein sehr häufiger Wert *weit verstreut*	sparse coding
viele Blöcke mit nur *einem* Wert	cluster, run length coding
viele Blöcke mit *wenig verschiedenen* Werten	indirect coding
wenig verschiedene, *zusammenhängende* Werte	run length coding
sehr viele verschiedene Werte	nur domain coding

4 Optimierung der Spaltenreihenfolge

Alle im vorherigen Abschnitt vorgestellten Techniken können nur dann eine optimale Kompressionsrate erzielen, wenn die Daten nach bestimmten Kriterien sortiert sind. So profitieren bis auf das run length coding alle Verfahren davon, die Vorkommen des häufigsten Wertes an den Anfang zu verschieben und diese somit nicht mitkodieren zu müssen. Für cluster, indirect und run length coding ist es weiterhin notwendig die Daten

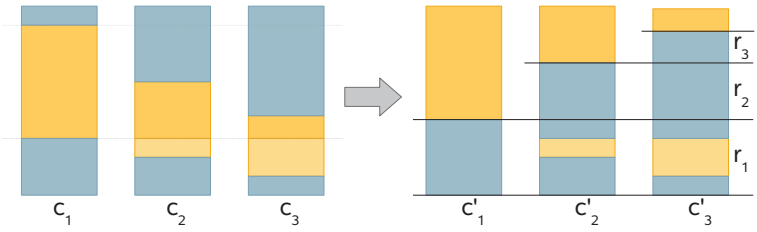

Abbildung 7: Beispiel für das Verschieben des häufigsten Wertes an den Anfang.

so anzuordnen, dass Blöcke mit möglichst wenig verschiedenen Werte entstehen. Aufgrund des mit der Anzahl der Spalten exponentiell anwachsenden Lösungsraumes ist das Suchen nach einer optimalen Lösung für dieses Optimierungsproblem praktisch undurchführbar. Bereits [Als75] hat zeigt, dass das Sortieren unter Verwendung von RLE ein NP-vollständiges Problem ist. Als Konsequenz wurden im Zuge dieser Arbeit mehrere Greedy-Heuristiken entwickelt, die einen Kompromiss zwischen der Laufzeit der Sortierung und der erzielten Speichereinsparung darstellen.

Der gesamte Optimierungsprozess einer Tabelle läuft in vier Schritten ab:

1. Überprüfung, ob eine Sortierung notwendig ist
2. Bestimmung lohnenswerter Spalten (Kandidaten)
3. Anwendung einer Heuristik
4. Bestimmung der besten Kompressionstechnik für jeden Kandidaten

Im ersten Schritt wird überprüft, ob sich genügend Daten verändert haben und damit eine neue Sortierung die Kompressionsrate verbessern kann. Danach werden alle Spalten bestimmt, die mehr als einen aber nicht zu viele unterschiedliche Werte besitzen, da sonst eine Umordnung keine Verbesserungen mit sich bringen würde. Nachdem eine der gleich vorgestellten Heuristiken mit der Optimierung fertig ist, wird für jeden Kandidaten die beste Kompressionstechnik bestimmt und angewendet.

Die einfachste Heuristik wird vom *FValueOptimizer* implementiert und verschiebt den häufigsten Wert jeder Spalte an den Anfang, wobei die Abhängigkeit der Spalten untereinander beachtet werden muss. Die Spalten können nicht unabhängig voneinander sortiert werden, da die Position der Werte die Zugehörigkeit zu einer Zeile ausdrückt und im Falle einer freien Sortierung ein teures Mapping von Spaltenwerten zu Zeilen erforderlich wäre. Abbildung 7 zeigt ein kleines Beispiel mit drei Spalten gleicher Bitbreite (c_1, c_2, c_3), in dem die Spalte 1 zuerst sortiert wird, weil dort im Vergleich zu den anderen Spalten die Häufigkeit des häufigsten Wertes (gelber Bereich) und somit die Einsparung am größten ist. Die Bereiche (r_1, r_2, r_3), die durch das Verschieben entstehen, werden im weiteren Verlauf als Restbereiche bezeichnet und können durch fortgeschrittene Verfahren weiter sortiert werden, ohne dass die bestehende Ordnung zerstört wird.

Alle Heuristiken, die auf dem FValueOptimizer aufbauen, sind durch den *FValueBlockOptimizer* realisiert und unterscheiden sich allein durch die Ermittlung der Reihenfolge, in der die Spalten sortiert werden. Um in den Restbereichen möglichst große, zusam-

menhängende Bereiche mit gleichen Werten zu erzeugen, erfolgt die Sortierung der Werte absteigend nach der Häufigkeit ihres Vorkommens. Falls diese Bereiche größer als die in den Kompressionstechniken verwendeten Blockgröße sind, werden sie zu der Liste der aktuellen Restbereiche hinzugefügt und in der nächsten Spalte mit berücksichtigt. Die folgende Tabelle fasst mögliche Strategien zusammen:

Strategie	**Sortierreihenfolge**
reversed	umgekehrte Verschiebereihenfolge
single value blocks	$\sim$ Anzahl der Blöcke mit einem unterschiedlichen Wert
average distinct values	$\sim \varnothing$ Anzahl unterschiedlicher Werte pro Block
valuable blocks	$\sim$ Anzahl lohnenswerter (komprimierbarer) Blöcke

Die *reversed* Strategie sortiert die Restbereiche entgegengesetzt der Reihenfolge, in welcher der häufigste Wert an den Anfang verschoben wird, da auf diese Weise die bis jetzt am wenigsten optimierten Spalten bevorzugt werden. Wenn nun eine Gleichverteilung und eine identische Anzahl unterschiedlicher Werte in allen Spalten angenommen wird, dann sinkt nach dem Entfernen eines großen Präfixes die Wahrscheinlichkeit große Bereiche mit gleichen Werten formen zu können und damit eine hohe Kompressionsrate zu erreichen. Während *single value blocks* (s. v. b.) versucht, die Kompression mit cluster coding zu verbessern, dienen die Maße bei *average distinct values* (a. d. v.) und *valuable blocks* (v. b.) dazu, eine gute Sortierreihenfolge für die Spalten beim Einsatz von indirect coding zu bestimmen. In den letzten drei Strategien wird wie beim FValueOptimizer die Anzahl der Bits eingerechnet, die zur Kodierung der Werte in den einzelnen Spalten notwendig sind, um die jeweilige Einsparung abzuschätzen.

Einen etwas anderen Ansatz verfolgen die durch den *BlockOptimizer* implementierten Heuristiken. Hier fängt die Sortierung unabhängig von dem häufigsten Wert in den einzelnen Spalten mit einem großen Restbereich an, der die gesamten Daten repräsentiert. Die Sortierreihenfolge bestimmt sich aus der Anzahl der unterschiedlichen Werte einer Spalte, wobei entweder mit dem Minimum (*min value count*) oder Maximum (*max value count*) angefangen wird.

In unserer Implementierung der vorgestellten Heuristiken werden bei der Sortierung einer Spalte nicht die Werte in allen Spalten getauscht, sondern nur die Einträge in einem Mapping, das die neue Zeilenreihenfolge bestimmt. Nach der Umordnung einer Spalte auf diese Weise kann man die Daten nicht mehr blockweise bearbeiten, sondern es werden viele Einzelzugriffe notwendig, die viel teuerer sind. Aus diesem Grund werden die häufigsten Werte im FValueOptimizer und die Maße im FValueBlockOptimizer nicht in jedem Schritt neu berechnet.

5 Evaluierung

Das Ziel dieses Abschnittes ist die Evaluierung der vorgestellten Heuristiken mit Hilfe der modifizierten TPC-H Daten und Anfragen aus [HRSD07]. Dort wird eine Projektion mit neun Spalten und 32 Millionen Zeilen aus den TPC-H Tabellen *customer*, *supplier*, *line-*

Heuristik	Größe in KB	Laufzeit in ms
keine	*321 245*	-
fvalue	262 527	1 886
fvalue[1]	285 170	-
fvalue[2]	262 500	7 647
reversed	136 042	6 322
single value blocks (s. v. b.)	160 898	8 329
single value blocks (s. v. b.)[3]	163 263	19 628
average distinct values (a. d. v.)	136 671	10 103
average distinct values (a. d. v.)[3]	150 540	19 204
valuable blocks (v. b.)	136 692	9 669
min value count (min v. c.)	204 609	211 264
max value count (max v. c.)	*108 090*	2 030

[1] für alle Attribute wurde *prefix coding* und nicht die beste Kompressionstechnik verwendet

[2] die häufigsten Werte wurden in jedem Schritt für den aktuellen Bereich neu bestimmt

[3] die Maße wurden in jedem Schritt für die aktuellen Restbereiche neu bestimmt

Tabelle 1: Erreichte Größe und für die Optimierung benötigte Zeit aller vorgestellten Heuristiken.

item und *order* erzeugt, wobei zusätzlich eine ungleichmäßige Datenverteilung gewählt wurde, um den Vorteil der Huffman-Kodierung zu zeigen. So entspricht die Verteilung der Länder von Lieferanten und Kunden dem kanadischen Handelsverkehr der Welthandelsorganisation. Darüber hinaus umfassen die Datumsangaben zu 99% Wochentage, wovon 40% auf Weihnachten und Muttertag fallen, und die Jahre von 3 bis 9999, wobei 99% im Bereich von 1995 und 2005 liegen.

Weil in [HRSD07] weiterhin eine andere Architektur (Row-Store) verwendet wird, können die Ergebnisse nicht ohne weiteres verglichen werden, was jedoch auch nicht Gegenstand dieser Arbeit ist.

Die Tabelle 1 enthält für alle vorgestellten Heuristiken die erreichte Größe in *KB* und die für die Optimierung benötigte Zeit in *Millisekunden*. Bei dem *cluster coding* und *indirect coding* wurde eine Blockgröße von 1024 Elementen verwendet. Bereits das Verschieben des häufigsten Wertes an den Anfang und die Verwendung von *prefix coding* bringt eine Einsparung von 11% (36 075 KB). Mit den optimalen Kompressionstechniken sind es sogar 18% (weitere 22 643 KB). Die Messungen zeigen ebenfalls, dass der Mehraufwand, in jedem Schritt den häufigsten Wert neu zu bestimmen, sich nicht lohnt, da die Verbesserung in der Kompression vernachlässigbar ist.

Sehr gute Ergebnisse erreicht auch die sehr einfache *reversed* Heuristik, die den Speicherbedarf bei den Testdaten um 58% (185 203 KB) senkt! Wie bei dem FValueOptimizer wurde auch für den FValueBlockOptimizer die Bestimmung der Maße in jedem Schritt implementiert, um mögliche Verbesserungen oder Verschlechterungen analysieren zu können. Im schlimmsten Fall hat sich die Kompressionsrate um 4% (von 136 671 KB auf 150 540 KB) verschlechtert.

Die längste Laufzeit wurde für die *min value count* Heuristik gemessen, welche die meis-

Heuristik	nur domain coding	prefix coding	sparse coding	cluster coding	indirect coding	run length coding
keine	9	-	-	-	-	-
fvalue	5	2	1	-	1	-
reversed	2	2	1	-	1	3
s. v. b.	1	1	1	2	4	-
a. d. v.	2	1	1	1	1	3
v. b.	2	2	1	-	1	3
min v. c.	3	-	-	-	2	4
max v. c.	2	-	2	-	2	3

Tabelle 2: Anzahl der Attribute, welche die jeweilige Kompressionstechnik verwenden.

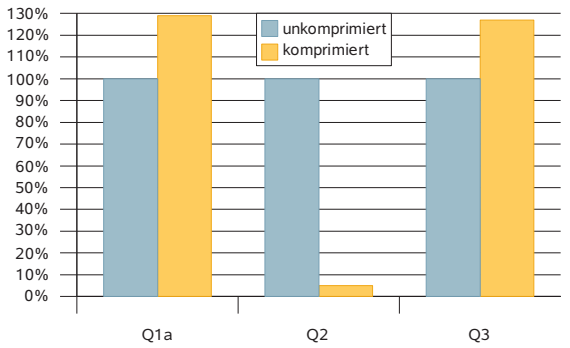

Abbildung 8: Vergleich von Anfragen aus [HRSD07] mit und ohne weitere Kompression.

ten Restbereiche bearbeitet und dafür viele teure Einzelzugriffe benötigt hat. Die beste Kompressionsrate (66%) hingegen wurde von *max value count* erzielt, wobei das Ergebnis relativ stark von einer bestehenden Ordnung abhängig ist, da nur relativ wenig Restbereiche und Werte sortiert wurden. So konnten mit einer guten Vorsortierung auch 74% (82 613 KB) erreicht werden.

Als Nächstes zeigt die Tabelle 2 wie viele Attribute die jeweilige Kompressionstechnik verwenden, nachdem die entsprechende Heuristik angewendet wurde. Hier ist deutlich zu sehen, dass *cluster coding* und *sparse coding* sehr selten automatisch ausgewählt wurden. Die Ursache liegt in dem Overhead, der beim ersten Verfahren für große Bereiche mit gleichen Werten auftritt, wenn für jeden Block der enthaltene Wert wiederholt gespeichert werden muss, was beispielsweise beim *run lenght coding* nur einmal für den gesamten Bereich erforderlich ist. Im Gegensatz dazu amortisiert sich der unter Umständen sehr große Bitvektor beim *sparse coding* erst, wenn ein Wert sehr häufig und verstreut vorkommt. Da für die meisten Attribute die Daten nach der Optimierung jedoch sortiert vorliegen, sind die anderen Kompressionsverfahren effektiver.

Zuletzt wird in Abbildung 8 die Geschwindigkeit von verschiedenen Anfragen mit und

ohne weitere Kompression verglichen, wobei die Laufzeit einer Anfrage unter ausschließlicher Verwendung von *domain coding* als 100% angegeben ist und die Daten mit der *min value count* Heuristik sortiert wurden, weil damit die höchste Kompressionsrate erzielt wurde. In der Anfrage *Q1a* wird eine Gruppierung und Aggregation durchgeführt, im Gegensatz zu den anderen beiden Anfragen, die außerdem zwei Gleichheitsprädikate (*Q2*) beziehungsweise ein Bereichsprädikat (*Q3*) anwenden. Es zeigt sich, dass die zusätzliche Kompression in den meisten Fällen die Anfrageverarbeitung kaum ausbremst, sondern sogar beschleunigen kann, wenn man die Eigenschaften einzelner Kompressionstechniken geschickt ausnutzt.

6 Zusammenfassung

In diesem Beitrag haben wir Techniken der Datenkompression für Column-Stores am Beispiel des BI-Accelerators TREX vorgestellt und evaluiert. Die auf der Idee des domain coding basierenden Verfahren ermöglichen eine effizienten Zugriff auf Datenblöcke wie auch auf einzelne Werte. Zur Erzielung einer hohen Kompressionsrate werden die Spaltenwerte zu Blöcken zusammengefasst, die nur wenige verschiedene Werte enthalten. Aus der dafür notwendigen Sortierung leitet sich die Aufgabe der Bestimmung einer optimalen Reihenfolge der Spalten bei der Sortierung ab. Da es sich hierbei um ein NP-vollständiges Problem handelt, haben wir in dieser Arbeit verschiedene Greedy-Lösungen vorgestellt. Die Ergebnisse der Evaluierung zeigen, dass mit unseren Heuristiken gute Kompressionsraten erzielt werden können und dass auch die Verarbeitung von Anfragen davon profitiert.

Literatur

[Aba07] D. J. Abadi. Column-Stores For Wide and Sparse Data. In *Proc. CIDR*, Seiten 292–297, 2007.

[Als75] P. A. Alsberg. Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring. *Proc. IEEE*, 63(8):1114–1122, 1975.

[AMF06] D. J. Abadi, S. R. Madden und M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. SIGMOD*, Seiten 671–682, 2006.

[AMH08] D. J. Abadi, S. R. Madden und N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proc. SIGMOD*, Seiten 967–980, 2008.

[BZN05] P. Boncz, M. Zukowski und N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Seiten 225–237, 2005.

[CGK01] Z. Chen, J. Gehrke und F. Korn. Query Optimization In Compressed Database Systems. *SIGMOD Rec.*, 30(2):271–282, 2001.

[Cor85] G. V. Cormack. Data Compression on a Database System. *Commun. ACM*, 28(12):1336–1342, 1985.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker und D. Wood. Implementation Techniques for Main Memory Database Systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[Gol66] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[GRS98] J. Goldstein, R. Ramakrishnan und U. Shaft. Compressing Relations and Indexes. In *Proc. 14th ICDE*, Seiten 370–379, 1998.

[GS91] G. Graefe und L. D. Shapiro. Data Compression and Database Performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Seiten 22–27, 1991.

[HBND06] A. Halverson, J. L. Beckmann, J. F. Naughton und D. J. DeWitt. A Comparison of C-Store and Row-Store in a Common Framework. Bericht 1570, University of Wisconsin-Madison, 2006.

[HD08] A. L. Holloway und D. J. DeWitt. Read-Optimized Databases, In Depth. In *Proc. 34th VLDB*, 2008.

[HLAM06] S. Harizopoulos, V. Liang, D. J. Abadi und S. R. Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proc. 32nd VLDB*, Seiten 487–498, 2006.

[HRSD07] A. L. Holloway, V. Raman, G. Swart und D. J. DeWitt. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *Proc. SIGMOD*, Seiten 389–400, 2007.

[Huf52] D. Huffman. A Method for Construction of Minimum-Redundancy Codes. *Proc. IRE*, 40(9):1098–1101, 1952.

[IW94] B. R. Iyer und D. Wilhite. Data Compression Support in Databases. In *Proc. 20th VLDB*, Seiten 695–704, 1994.

[KL03] O. Kaser und D. Lemire. Attribute Value Reordering For Efficient Hybrid OLAP. In *Proc. 6th DOLAP*, Seiten 1–8, 2003.

[LLR06] T. Legler, W. Lehner und A. Ross. Data Mining with the SAP NetWeaver BI Accelerator. In *Proc. 32nd VLDB*, Seiten 1059–1068, 2006.

[OR86] F. Olken und D. Rotem. Rearranging Data to Maximize the Efficiency of Compression. In *Proc. 5th PODS*, Seiten 78–90, 1986.

[RS06] V. Raman und G. Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proc. 32nd VLDB*, Seiten 858–869, 2006.

[RSQ+08] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang und R. Sidle. Constant-Time Query Processing. In *Proc. 24th ICDE*, Seiten 60–69, 2008.

[Rv93] M. A. Roth und S. J. van Horn. Database Compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[SAB+05] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. C. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran und S. Zdonik. C-Store: A Column-oriented DBMS. In *Proc. 31st VLDB*, Seiten 553–564, 2005.

[WKHM00] T. Westmann, D. Kossmann, S. Helmer und G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.*, 29(3):55–67, 2000.

[ZHNB06] M. Zukowski, S. Héman, N. Nes und P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. 22nd ICDE*, Seite 59, 2006.