Datenbank Spektrum (2011) 11: 43–46
DOI 10.1007/s13222-011-0047-5

FACHBEITRAG

Optimierungstechniken in Column Stores

David Fekete

Eingegangen: 18. Januar 2011 / Angenommen: 5. Februar 2011 / Online publiziert: 18. Februar 2011

Zusammenfassung Column Stores als spaltenbasierte Datenbankmanagementsysteme (DBMS) sind als Alternative zu zeilenbasierten DBMS für Anwendungen in den Bereichen Data Warehousing (DWH) und Business Intelligence (BI) in den Fokus gerückt. Mittlerweile sind mehrere Systeme am Markt vorhanden, die insbesondere das Leistungsverhalten der in DWH vorherrschenden Leseanfragen optimieren sollen. In diesem Beitrag wird gezeigt, welche Optimierungstechniken zusätzlich bei Columns Stores auf welchen Ebenen mit wie großem Effekt eingesetzt werden können. Einer Beschreibung und Analyse einer Auswahl dieser Techniken folgt die Präsentation einer empirischen Evaluation letzterer in einem kommerziellen Column-Store-DBMS anhand des auf DWH ausgelegten *Star Schema Benchmark* (SSB).

Schlüsselwörter Column store · Optimierung · Late materialization · Block iteration · Star schema benchmark · Kompression

1 Einleitung

Row Stores, als am Markt vorherrschende Datenbankmanagement systeme (DBMS) [1], waren ursprünglich in den 1970er Jahren für häufige Schreibzugriffe in OLTP-Szenarien (Online Transaction Processing) entworfen worden [1, 8]. Tupel werden dort zusammenhängend hintereinander (zeilenweise) auf den Datenträger gesichert.

Im Gegensatz dazu spezialisieren sich Column Stores auf die Optimierung von im On-Line Analytical Processing (OLAP) und somit Data Warehousing (DWH) vorherrschenden Anfragen. Diese zeichnen sich durch geringe Vo raussehbarkeit, hohe Dauer, Lese-Orientierung und Fokussierung auf einzelne Attribute aus [1]. Column Stores zerlegen jedes Tupel in dessen Attribute und speichern diese separat in mehreren Attribut-Blöcken auf einen physischen Datenträger. Man spricht hier von einer spaltenweisen Speicherung.

Neben diesem Unterschied auf der untersten Ebene der DBMS-Schichtenarchitektur [11] ergeben sich wichtige performanzrelevante Faktoren erst auf der Ebene der eigentlichen Anfragenausführung (bei dem sog. Query Executor) sowie auch auf der verwendeten Hardware des Systems selbst. Ein Query Executor muss sich der physischen Speicherung bewusst sein [8], um die Vorteile (vgl. Abschn. 2) von Column Stores möglichst optimal nutzen zu können. Durch die Leseorientierung ist nicht nur mehr der Datendurchsatz (I/O) wie bei OLTP entscheidend, sondern auch die CPU-Leistung aufgrund der komplexen analytischen Natur von OLAP-Anfragen [4, 7]. Auch sind Durchsätze von Puffern auf höheren Ebenen der Speicherhierarchie, wie Arbeitsspeicher und CPU-Cache, von größerer Bedeutung. Ziel dieser Arbeit ist es, die Frage zu beantworten, welche Optimierungstechniken bei einem speziellen, kommerziellen Column Store mit wie großem Effekt eingesetzt werden können.

Diese Arbeit fasst folgende Bachelorarbeit zusammen: Fekete D. (2010), Optimierungstechniken in Column Stores. Bachelorarbeit, Westfälische Wilhelms-Universität Münster, Münster.

D. Fekete (✉)
Lehrstuhl für Informatik, Institut für Wirtschaftsinformatik, Universität Münster, Leonardo-Campus 3, 48149 Münster, Deutschland
e-mail: david.fekete@uni-muenster.de

2 Optimierungstechniken

Das folgende Kapitel stellt einige Optimierungstechniken für Column Stores vor, die neben der physischen Speicherung die besondere Eignung von Column Stores für DWH verstärken oder gar erst ermöglichen.

2.1 Kompression

Durch Kompression werden Daten in eine kürzere Form überführt, ohne Informationen zu verlieren. Während es auch für Row Stores bereits Möglichkeiten gibt, Kompression einzusetzen, ermöglichen Column Stores eine weitreichendere Nutzung der Kompression. Es kann mehr Speicherplatz und somit Transfervolumen auf dem Datenträger eingespart werden als bei Row-Store-Verfahren, da ähnliche Daten aufgrund der spaltenbasierten Speicherung physisch näher beieinander liegen und somit besser komprimiert werden können [5]. Weiterhin können für jede Spalte unterschiedliche Algorithmen verwendet werden. Auch kann ein geschickt agierender Query Executor, der direkt auf komprimierten Daten arbeitet, eine aufwendige Dekomprimierung herauszögern oder gar vermeiden. Dadurch wird CPU-Leistung eingespart.

Ein Beispiel für ein Kompressionsverfahren ist die Lauflängen kodierung (*Run-length Encoding*, RLE), bei der statt mehrfachen Wiederholungen von aufeinanderfolgenden Attribut-Werten lediglich der Attribut-Wert mitsamt der Wiederholungsanzahl (die *Lauflänge*) in einem Tupel vermerkt wird.

Für Attribute mit einer geringen Anzahl voneinander verschiedenen Werten (*unique values*) bietet sich auch die Enkodierung über sog. Bit-Vektoren (Bitmaps) an. Für jede einzigartige Wertaus prägung eines Attributs wird ein aus 0 und 1 bestehendes Array mit derselben Länge wie das Attribut angelegt. Für eine Wert-Ausprägung beinhaltet dieser Vektor an der Position im Array, an der das Attribut ebenfalls diesen Wert besitzt, eine 1 und ansonsten eine 0.

Weitere einfache Möglichkeiten sind die Unterdrückung von Null-Werten (Null-Suppression) oder Wörterverzeichnisse (Dictionary Encoding). Neben diesen einfach anzuwendenden Verfahren ist auch für bestimmte Daten eine Anwendung von komplexeren Kompressionsalgorithmen, wie etwa Lempel-Ziv oder gzip, möglich, obgleich diese einen höheren Aufwand für den Hauptprozessor bedeuten.

Insgesamt kann so eine Performanzsteigerung bis zum Faktor 10 erreicht werden, sofern auch der Query Executor eine zu frühe Dekomprimierung (vgl. Abschn. 2.3) vermeiden kann. Wird nur die Verringerung der Datenmenge betrachtet, ist höchstens eine Steigerung um den Faktor 3 möglich, selbst wenn die Datenmenge um das zehnfache verringert werden kann [1].

2.2 Block Iteration

Ein typisches Problem in Row Stores ist die Verarbeitung von einzelnen Tupeln innerhalb des Query Executor. Zunächst müssen benötigte Attribute aus diesen extrahiert, dann verarbeitet und ggf. projiziert werden. Zudem muss das gesamte Tupel eingelesen werden, was zu einer Verlangsamung durch unnötigen I/O führt. Zusätzlich entsteht ein Mehraufwand für den Hauptprozessor durch den Vorgang der Extraktion der Attribute selbst.

Column Stores hingegen lesen aufgrund der physischen Trennung nur wirklich für die Anfrage benötigte Attribute ein. Hier werden auch ganze Blöcke eingelesen und verarbeitet, ohne jedoch unnötige Daten mit einzulesen, um diese dann verwerfen zu müssen. Bei fester Attribut-Länge kann ein entsprechender Attribut-Block mittels Iteration durchlaufen und verarbeitet werden, vergleichbar mit dem Zugriff auf ein Array [2]. Solche Zugriffe (bzw. Arrays) ermöglichen es modernen Prozessoren, solchen Code besser zu optimieren und zu parallelisieren (sog. *loop pipelining*), um so mehr Instruktionen pro Taktzyklus ausführen zu können [2, 3].

2.3 Late Materialization

Entscheidend für die Performanz eines Column-Store-Systems kann es sein, wann die in Spalten gespeicherten Attribute beim Verarbeiten einer Anfrage für das Ergebnis einer selbigen wieder zu Tupeln zusammengesetzt werden. Man spricht auch von der sog. *Tuple materialization*.

Wird die Materialisierung direkt beim Zugriff auf die Datenstrukturen vorgenommen (*Early Materialization*, EM), kann die eigentliche Auswertung der Anfrage mit denselben Tupel-Operatoren wie in einem Row Store vorgenommen werden. Jedoch müssen hier komprimierte Spalten sofort dekomprimiert werden, und es werden wieder unnötige Daten eingelesen. Zu dem erhöhten Transfervolumen auf der Festplatte kommen noch die Hauptprozessor-Leistung sowie Arbeitsspeicher-Nutzung für das Zusammensetzen der Tupel und das Dekomprimieren hinzu.

Materialisiert man die Tupel jedoch erst so spät wie möglich, ist bei einigen Anfragen ein Performanz-Vorteil zu erzielen [1]. Bei der späten Materialisierung (*Late Materialization*, LM) werden bei der Selektion von Attributen lediglich Positionslisten gebildet, die beinhalten, welche Tupel Bestandteil des Ergebnisses sind. Erst im letzten Schritt werden anhand der Positionslisten Tupel mit den zu projizierenden Attributen materialisiert (vgl. Abb. 1).

Umsetzungsmöglichkeiten für Positionslisten sind beispielsweise einfache Listen (*Attribute 1*, *3 und 7*), Bereichsangaben (*Attribute 1 bis 4*) oder Bit-Vektoren (vgl. Abschn. 2.1, 3.2). Mehrere solchen Positionslisten können

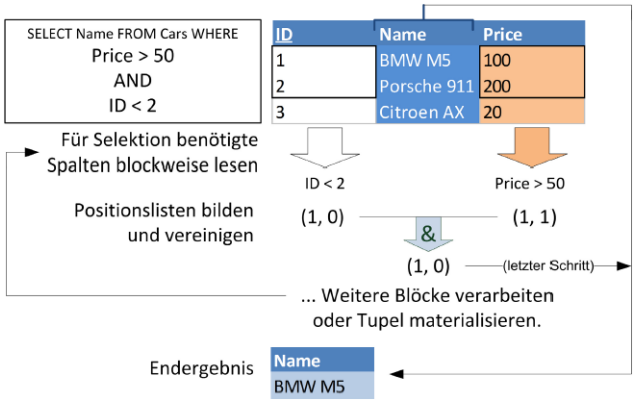


Abb. 1 Beispiel für Late Materialization (LM) in Column Stores

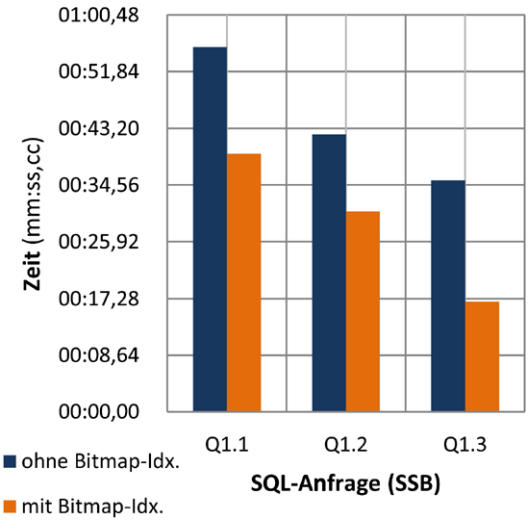


Abb. 2 Anfragezeiten in Flight 1 des SSB (SF 10)

dann mit vergleichsweise einfachen und schnellen Mengenoperationen, wie etwa einer Intersektion mittels *AND*, vereinigt werden.

Mit dieser Methode werden unnötige Transfers von der Festplatte eingespart. Auch kann die Dekomprimierung von Attributen vermieden (z.B. bei Aggregation) oder hinausgezögert (bei Projektion) werden, sodass mit LM die Vorteile der Kompression (vgl. Abschn. 2.1) voll genutzt werden können [2]. Allerdings muss der Zugriff auf eine Tabelle ggf. zweimal während der Verarbeitung erfolgen, was noch eigene Herausforderungen mit sich bringen kann (vgl. [1]).

3 Empirische Evaluation

In diesem Kapitel wird das Vorgehen zur praktischen Untersuchung von Optimierungstechniken in einem kommerziellen Column-Store-System erläutert. Anschließend werden einige Untersuchungsergebnisse dargelegt und erörtert.

3.1 Methode zur Leistungsmessung

Zur empirischen Evaluation der genannten Optimierungen wurde der speziell auf DWH ausgelegte *Star Schema Benchmark* (SSB, Rev. 3, Juni 2009) verwendet [6]. Der SSB enthält eine Datenbank im Sternschema sowie einen Datensatzgenerator und eine Sequenz von Anfragen, deren Ausführungszeiten zu messen sind. Zusätzlich ist die Zeit zum Einspielen der Datensätze zu erfassen.

Mittelpunkt des Sternschemas, welches sich auch in typischen DWH wiederfindet [11], ist eine sog. Faktentabelle (*facts table*). Um diese Faktentabelle herum sind Dimensionstabellen (*dimension table*) angeordnet. Dimensionstabellen sind über Fremdschlüssel in der Faktentabelle mit dieser verbunden. In einem Entity-Relationship-Modell (ERM) würden typischerweise Dimensionstabellen durch Entity-Typen und die Faktentabelle durch Relationship- oder uminterpretierte Relationship-Typen abgebildet.

Als Test-DBMS kam ein kommerzielles Column-Store-System, im Folgenden mit *XYZ* bezeichnet, zum Einsatz. Das Testsystem besteht aus einer virtuellen Maschine (VM) mit Windows 7 Professional x64 auf einem Intel Core i7-860 mit 1,5 GB RAM für die VM sowie einer 250 GB Festplatte der Spinpoint-Serie von Samsung. Gemessen wurden die Skalierungsfaktoren 1, 10 und 25 des SSB. Je nach Skalierungsfaktor enthalten die Faktentabelle und die Dimensionstabellen eine unterschiedliche Anzahl an Datensätzen, wobei die Faktentabelle immer die meisten Zeilen enthält und sich deren Anzahl auch am stärksten mit steigendem Skalierungsfaktor erhöht. Alle Tests zur Zeiterfassung wurden dreimal durchgeführt, um die Aussagequalität der Resultate zur erhöhen.

In der zugrunde liegenden Arbeit wurde der Einfluss auf die Ergebnisse des SSB durch verschiedene Indexe und Konfigurationsgrößen (z. B. bestimmte Puffergrößen), die *XYZ* anbietet, untersucht. Nachfolgend werden einige Teile des Gesamtergebnisses im Kontext erörtert.

3.2 Ergebnisse der Leistungsmessung

Insbesondere bei Verwendung von speziellen, zusätzlichen Indexen von *XYZ* lässt sich die Wirkung der gezeigten Optimierungstechniken demonstrieren. Im Skalierungsfaktor 10 erwirken Bitmap-Indexe (vgl. Abschn. 2.1) auf zwei Spalten der Faktentabelle mit ca. 60 Millionen Zeilen bei einer Selektionen mit Attributen dieser eine Leistungssteigerung um ca. 30 bis 50 % (vgl. Abb. 2). Es handelt sich um drei Anfragen des ersten Satzes aus den vier durchzuführenden Sätzen (sog. *Flights*) des SSB. In diesen wird die Faktentabelle vor dem Join an eine Dimensionstabelle mit bis zu zwei Spalten der selbigen, die jeweils eine geringe Anzahl voneinander verschiedener Werte besitzen, eingeschränkt (vgl. Anfragen 1 bis 3 aus Flight 1 des SSB).

Für solche Spalten sind Bitmap-Indexe zur Kompression gut geeignet. Diese ermöglichen insbesondere die Positionslistenbildung für die späte Materialisierung, da die Bit-Vektoren direkt als Positionsliste verwendet werden können (vgl. Abschn. 2.3). Hier wird auch die optimale Nutzung der Kompression ermöglicht, da die betreffenden Spalten nur zur Selektion verwendet werden und nicht später bei einer Projektion noch dekomprimiert werden müssen.

Hingegen konnten zusätzliche Indexe auf den Dimensionstabellen, aufgrund ihrer geringer Größe im Vergleich zur Faktentabelle, in dieser Untersuchung nur geringfügigen Einfluss auf die Ausführungszeit zeigen.

Weiterhin bietet *XYZ* eine Option zur proaktiven Kompression aller Spalten. Die Verwendung dieser Option bestätigt ebenfalls den oben und zuvor (vgl. Abschn. 2.1) genannten positiven Effekt auf die Performanz bei der Ausführung der SSB-Anfragen. Alle Anfragen wurden durch Aktivieren dieser Option beschleunigt. Das Beladen der Tabellen mit den Test-Daten erfordert jedoch mit dieser Option mehr Ressourcen (insb. Arbeitsspeicher), sodass sich das Beladen der Datenbank unter Ressourcen-Knappheit verlangsamen kann. Im Testsystem war dies beim Skalierungsfaktor 25 auch zu beobachten, wo das Beladen der Tabellen mit aktivierter Option messbar langsamer war als ohne diese Option.

4 Fazit

Abschließend lässt sich festhalten, dass für Column Stores vielfältige Optimierungstechniken bestehen und diese in der Lage sind, Anfragen in DWH-Anwendungen vielfach zu beschleunigen.

Techniken wie Kompression, blockweise Verarbeitung und späte Materialisierung erhöhen die Leistung von Column Stores um ein Vielfaches, wenn sie vollständig in diese integriert werden. Neben den eigentlichen Vorzügen der spaltenbasierten Speicherung sind es diese Techniken, die den vielfach festgestellten, enormen Geschwindigkeitsvorteil von Column Stores ermöglichen.

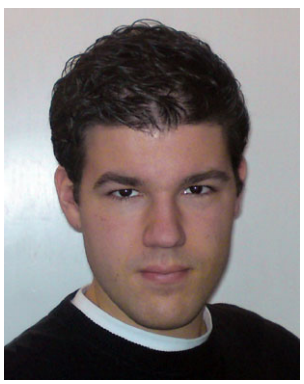
Die empirische Evaluation zeigt Möglichkeiten auf, wie die Leistung innerhalb von *XYZ* in einem typischen DWH-Anwendungsfall optimiert werden kann. Indexstrukturen und weitere Optionen können das Leistungsverhalten positiv beeinflussen, indem sie die Eigenschaften von Column Stores ausnutzen.

Ferner könnten bereits formulierte Ansätze, wie der von ABADI vorgestellte Invisible Join [1], die Performanz, insbesondere in DWH-Anwendungen mit Sternschema, steigern.

In der Zukunft könnten auch neuartige Speichermedien wie Flash-Datenträger (SSD) noch weiter in den Anwendungsfokus rücken. Wie verschiedene Entwicklungen zeigen, kann mittels SSDs bereits die Leistung in Row Stores erhöht werden [9, 10]. Um den Vorteil in DWH-Anwendungen auch zukünftig zu bewahren oder auszubauen, sind auch hier weitere Forschungen für Optimierungstechniken in Column Stores erforderlich.

Literatur

1. Abadi DJ (2008) Query execution in column-oriented database systems. Dissertation, Massachusetts Institute of Technology, Cambridge
2. Abadi DJ, Madden SR, Hachem N (2008) Column-stores vs. row-stores: how different are they really? In: Proc of the ACM SIGMOD int conf on management of data, Vancouver, 2008
3. Boncz PA, Zukowski M, Nes N (2005) MonetDB/X100: hyper-pipelining query execution. In: 2nd bienn int conf on innovative data Systs Res (CIDR), Asilomar
4. Luo Q, Ross KA (2008) In: 4th int workshop on data management on new hardw, Vancouver
5. MacNicol R, French B (2004) Sybase IQ multiplex—designed for analytics. In: Proc of the 30st VLDB conf, Toronto
6. O'Neil P, O'Neil E, Chen X (2009) The star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.pdf. Zuletzt abgerufen am 16.01.2011
7. Plattner H (2009) A common database approach for OLTP and OLAP using an inmemory column database. In: Proc of the 35th SIGMOD int conf on management of data, Providence
8. Stonebraker M, Abadi DJ, Batkin A et al (2005) C-Store: a column-oriented DBMS. In: Proc of the 31st VLDB conf, Trondheim
9. Shah MA, Harziopoulos S, Wiener JL, Graefe G (2008) Fast scans and joins using flash drives. In: Proc of the 4th int workshop on data management on new hardw, Vancouver
10. Tsirogiannis D, Harizopoulos S, Shah MA, Wiener JL, Graefe G (2009) Query processing techniques for solid state drives. In: Proc of the 35th SIGMOD int conf on management of data, Providence
11. Vossen G (2008) Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme. Oldenbourg Wissenschaftsverlag, Oldenbourg



David Fekete geboren 1987 in Szolnok (Ungarn), ist Master-Student im Studiengang Information Systems an der Westfälischen Wilhelms-Universität Münster. Seine Schwerpunkte sind Business Intelligence sowie Information Management. Seit 2010 besitzt Herr Fekete den akademischen Grad Bachelor of Science (B.Sc.) im Fach Wirtschaftsinformatik. Derzeit ist er als wissenschaftliche Hilfskraft am Lehrstuhl für Informatik am Institut für Wirtschaftsinformatik der Universität Münster tätig.