Hochschule für Technik, Wirtschaft und Kultur Leipzig

Oberseminar "Datenbanksysteme - Aktuelle Trends"
SS 2015

Spaltenorientierte Datenbanken

Marcel Gladbach
14 MIM
Juni 2015

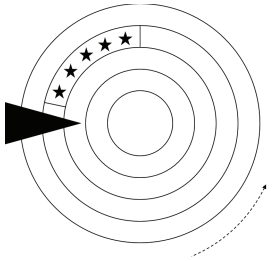
Inhaltsverzeichnis

Einführung

Online Analytical Processing (OLAP) in Data Warehouses (DWH) spielt eine immer größere Rolle in Unternehmen verschiedenster Art ein. Dabei lassen sich bei zunehmend wachsenden Datenbeständen Performance-Vorgaben für analytische Anfragen mit herkömmlichen Datenbankmanagementsystemen (DBMS) immer schwerer bewältigen. Ein Grund hierfür ist die Art der Datenspeicherung in den bestehenden zeilenorientierten DBMS.

Wie liest eine Festplatte Daten bei zeilenorientierten Datenbanken?

Auf persistenten Speichermedien (Festplatten) werden Daten unter dem Lesekopf nach Suche der Anfangsposition sequentiell eingelesen und zur Weiterverarbeitung bereitgestellt.



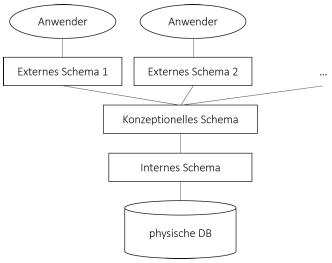
Eine typische analytische Anfrage zur Bestimmung der Personalkosten in der Mitarbeiter-Tabelle mit 10 Spalten könnte so aussehen:

```
SELECT SUM(Gehalt) FROM Mitarbeiter
```

Da in einer zeilenorientierten Datenbank alle Tupel einer Tabelle nacheinander abgespeichert werden, müssen für diese Anfrage hardwareseitig Attribute aller Spalten der Tabelle eingelesen werden (Scan), obwohl nur 1 von 10 Spalten für die Anfrage relevant ist. Die Anfragezeit entspricht somit der Zeit zum Lesen ALLER Daten der Tabelle und ist daher abhängig von der Gesamtgröße der Tabelle. Dies gilt für alle Anfragen, in welchen Scans über die gesamte Tabelle gemacht werden und Indizes entweder nicht vorhanden sind oder nicht verwendet werden können. [3]

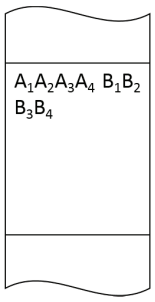
Es stellt sich also die Frage, wie man die Art der Datenspeicherung auf persistenten Medien verändern kann, um die nötige Optimierung der Ausführungszeiten analytischer Anfragen zu erzielen. Ein Ansatz sind spaltenorientierte Datenbanken (Column Stores), welche das Anforderungsprofil von OLAP meist besser abdecken können.

1 Überblick - Physische Datenmodelle



Wie bereits erwähnt, geht es bei Column Stores um die Art der Datenspeicherung auf dem persistenten Medium. Die folgenden Betrachtungen sind also dem internen Schema zuzuordnen, zu welchem die physischen Datenmodelle zählen. Die wichtigsten werden in diesem Kapitel erläutert. Auf konzeptioneller Ebene befindet sich bei Column Stores wie Row Stores das relationale Datenmodell und dementsprechend ist die Anfragesprache in den meisten Fällen SQL. Zur Veranschaulichung wird für jedes physische Datenmodell gezeigt, wie zwei Tupel A {A1,A2,A3,A4} und B {B1,B2,B3,B4} physisch gespeichert werden.

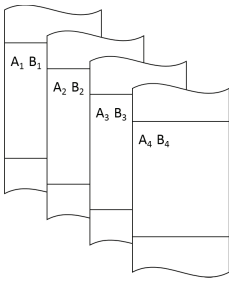
1.1 Zeilenorientierte Speicherung (Row Store)



Das in fast allen heutigen DBMS verwendete n-äre Speichermodell (NSM) beruht auf der zeilenweisen Speicherung der Tupel. Die Datensätze (Tupel) werden dabei direkt hintereinander in der gleichen Datei abgelegt. Eine Trennung der Tupel erfolgt nur, wenn die Seitenlänge des Festspeichers überschritten wird.

Die Vorteile des NSM sind das Lesen von kompletten Tupels mit nur einem Seitenzugriff und die leichte Änderbarkeit einzelner Attributwerte. Es ergeben sich aber wie in der Einführung gezeigt erhebliche Performance-Nachteile, da bei Scans immer alle Attributwerte gelesen werden müssen. [1] [2]

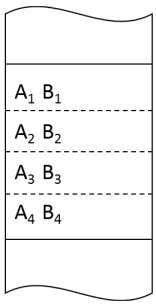
1.2 Spaltenorientierte Speicherung (Column Store)



Im Gegensatz dazu werden bei Column Stores mit dem Decomposition Storage Model (DSM) die Tupel spaltenweise abgespeichert. Dabei wird für jede Spalte eine neue Datei angelegt. Die Adressierung der Attributwerte erfolgt dann in den meisten Fällen über die Position (d.h. der erste Wert in jeder Datei gehört zum ersten Tupel). Manchmal werden auch binäre Relationen mit einem künstlichem Primärschlüssel (= Surrogat) angelegt.

Durch die spaltenweise Speicherung können die Daten besser komprimiert werden (Kapitel 3) und Scanoperationen sind wesentlich effizienter. Allerdings sind INSERTs und UPDATEs komplizierter und das Lesen aller Spalten etwas aufwendiger. [1] [2]

1.3 PAX (Partition Attributes Across)



Ein drittes Modell, welches hier nur kurz erwähnt werden soll, ist PAX. Hier werden zwar alle Tupel in derselben Datei gespeichert, jedoch spaltenweise getrennt durch die Zeilengrenzen des CPU-Cache. Die vertikale Partitionierung erfolgt also auf höherer Ebene der Speicherhierachie.

PAX stellt einen Kompromiss zwischen NSM und DSM dar. Es ist ein einfacherer Zugriff auf komplette Datensätze als im DSM möglich, dennoch werden die Vorteile der Spaltenorientierung genutzt, wenn die Daten sich im RAM befinden. Allerdings bringt dies keine Ersparnis beim Lesen der Daten vom Festspeichermedium. [1] [2]

2 Row Store vs Column Store

2.1 Beispiele

In diesem Abschnitt werden anhand konkreter SQL-Statements die Vor- und Nachteile von Column Stores und Row Stores gezeigt. [8] [7]

Veränderung der Tabellenstruktur

```
ALTER TABLE Mitarbeiter ADD Geburtsdatum date
```

RS: - Verschieben jedes Tupels (Shift) und Erneuern der Tabelle (Rebuild), enormer Aufwand je nach Datenmenge in Tabelle

CS: + einfaches Hinzufügen einer neuen Datei

Einfügen eines neuen Datensätze

(Gleiches gilt für das Aktualisieren eines kompletten Datensatzes.)

```
INSERT INTO Mitarbeiter (PNr, Nachname, Vorname, ..., Gehalt)
VALUES (51, 'Maier', 'Karl-Heinz', ..., 42000)
```

RS: + Anhängen des neuen Tupels ans Ende der Tabelle (evtl. Index-Aktualisierung)

CS: - Öffnen und Schreiben jeder Datei

Als Ausweg kann mit sogenannten Bulk Loads (Stapeln) gearbeitet werden. Dabei werden INSERTs gesammelt und dann auf einmal ausgeführt. Dies ist auch bei der Verwendung von Indizes bei Row Stores sinnvoll.

Aktualisieren eines Wertes

```
UPDATE Mitarbeiter SET Gehalt = 23000 WHERE PNr = 16
```

RS: Ändern des Wertes (evtl. Shift bei Überlänge)

CS: Öffnen, Suche und Schreiben in entsprechender Datei

Auch hier ist das Stapeln von UPDATEs sinnvoll, da vor jedem UPDATE ein Suchvorgang ausgeführt wird.

Aktualisieren einer Spalte

```
UPDATE Mitarbeiter SET Gehalt = Gehalt * 1,5
```

RS: - Suchen und Ändern der Spalte in jedem Tupel

CS: + Aktualisierung einer Datei

Auslesen einzelner kompletter Datensätze

```
SELECT * FROM Mitarbeiter WHERE PNr = 8
```

RS: + Auslesen des gesamten Tupels mit einem Zugriff (Performance-Vorteile mit Index), ABER: Scan über gesamte Tabelle, wenn Index nicht vorhanden oder nicht verwendet

CS: - Suche in jeder Datei

Aggregation / Auslesen vieler Datensätze (unter Verwendung weniger Spalten)

```
SELECT SUM(Gehalt) FROM Mitarbeiter
```

RS: - Lesen gesamter Tabelle

CS: + nur Lesen der Datei(en) für relevante Spalte(n)

2.2 Fazit

OLTP (Online Transactional Processing)	Row Store	Column Store
Einzelnes INSERT	↗	↘
Einzelnes UPDATE	↗ / →	↘ / →
SELECT über wenige, aber komplette Tupel	↗	↘

OLAP (Online Analytical Processing)	Row Store	Column Store
Bulk Load	→	→
UPDATE über komplette Spalte	↘	↗
SELECT über viele Tupel (unter Verwendung weniger Spalten)	↘	↗

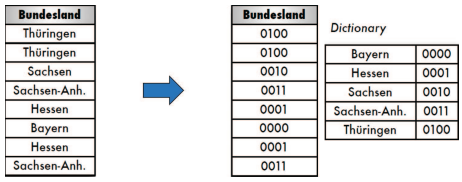
Column Stores sollten nicht für OLTP verwendet werden, bieten aber einen enormen Performance-Vorteil für OLAP.

3 Kompression

Neben der Reduzierung des Speicherplatzes, geht es bei der Kompression auch darum den Datendurchsatz zu erhöhen und so z.B. mehr Daten im Datenbankpuffer zu haben. Die Kompressionstechniken (welche im Folgenden vorgestellt werden) müssen dabei verlustfrei sein, d.h. es dürfen keine Daten verloren gehen. Weiterhin sollte die Dekompression leichtgewichtig sein und im besten Fall können relationale Operationen direkt auf den komprimierten Daten ausgeführt werden. Column Stores sind sehr gut für Kompression geeignet, da in einer Spalte oft gleichartige Daten vorkommen und diese hier physisch nah beieinander liegen. [2] [9]

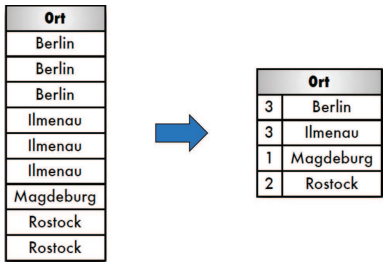
3.1 Dictionary Encoding (Wörterbuch)

Bei der Wörterbuchkompression werden kurze Binär-Codes statt der vollständign Attribute abgespeichert. Zur Übersetzung zurück in den Volltext wird ein Wörterbuch angelegt. Diese Technik ist vor allem mehrfach vorkommenden und langen Werten anwendbar. [2] [9]



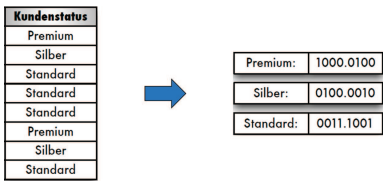
3.2 Run Length Encoding (Lauflänge)

Die Lauflängencodierung eignet sich zur Kompression von Sequenzen identischer Werte. Unterstützt durch die Sortierung der Daten werden dabei alle Werte nur einmalig mit der Häufigkeit ihrer Wiederholung gespeichert. Eine Weiterverarbeitung (Aggregation, z.B. Summierung) der Daten in komprimierter Form ist sehr gut möglich. [2] [9]



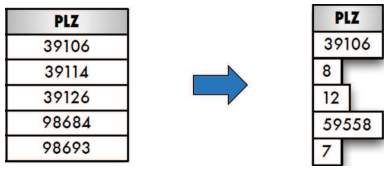
3.3 Bit-Vector Encoding

Bei einer insgesamt kleinen Anzahl mehrfach vorkommender Werte eignet sich auch das Bit-Vector Encoding. Hier wird pro unterschiedlichem Wert ein Bitstring gespeichert. Darin steht eine 1, wenn der Wert an der entsprechenden Position vorkommt, sonst eine 0. Die Länge des Bitstrings entspricht also Anzahl der Tupel in der Spalte. [2]



3.4 Delta Coding

Beim Delta Coding wird statt kompletter Werte nur jeweils die Differenz zum Vorgänger abgespeichert. Diese Technik eignet sich insbesondere bei Sequenzen aufeinanderfolgender ähnlicher ganzzahliger Werte. Minimale Differenzen werden bei vorab sortierten Spalten erreicht. [2] [9]

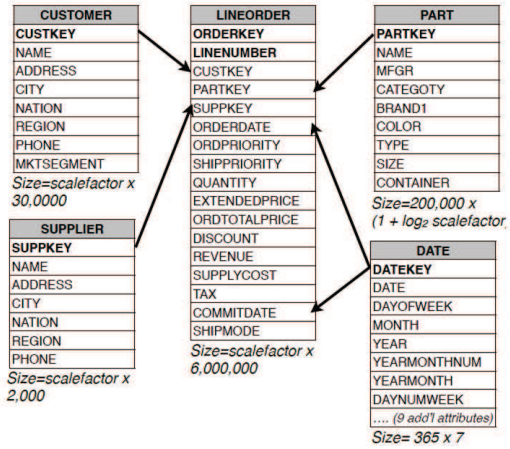


3.5 Fazit

Die verwendete Methode zur Kompression muss immer an der Datenlage ausgerichtet werden. Eine Wörterbuchkompression wäre beispielsweise für Postleitzahlen nicht sinnvoll. Die Speicherplatzersparnis ist bei allen Techniken enorm (meist weit über 50%, manchmal sogar über 90%). [9]

4 Performance

4.1 Star Schema Benchmark



Alle folgenden Perfomance-Tests wurden anhand des sogenannten “Star Schema Benchmark” durchgeführt. Dieser gilt als sehr aussagekräftig im Bereich DWH. Grundlage sind 5 Tabellen im Starschema (Faktentabelle „lineorder“ und 4 Dimensionstabellen). In den Tests wurde der scalefactor=10 verwendet, d.h. in der Faktentabelle befinden sich 60 Mio. Tupel. Zur Analyse der Performance werden 4 „Query Flights“ mit jeweils 3 bis 4 Einzelabfragen durchgeführt. Dabei werden typische DWH-Anwendungen abgedeckt. In den folgenden Diagrammen sind jeweils die durchschnittlichen Ausführungszeiten aller Queries dargestellt. Das für die Tests verwendete Column Store DBMS war C-Store (Abschnitt 5.1), das nicht benannte Row Store DBMS war “a very recent product release of System X”. [4]

4.2 Optimierung in Column Stores

Zur vollen Ausnutzung der Spaltenorientierung muss die Art der physischen Speicherung bei der Query-Ausführung berücksichtigt werden. Dazu werden folgende Optimierungstechniken angewendet.

4.2.1 Kompression

Die Kompression sollte spaltenspezifisch erfolgen, d.h. es werden unterschiedliche Techniken (Kapitel 3) pro Spalte angewandt. Um den Datendurchsatz so gering wie möglich zu halten, sollte die aufwendige Dekompression so spät wie möglich erfolgen und Operationen, wenn möglich, direkt auf komprimierten Daten ausgeführt werden. [4] [6]

4.2.2 Block Iteration

Block Iteration bedeutet, dass die Attribute der Spalten blockweise an Operatoren geleitet und so verarbeitet werden. Bei Attributen mit fester Länge kann das Durchlaufen der Spalten (Attributblöcke) durch Iteration erfolgen, ähnlich wie der Zugriff auf ein Array. Damit entstehen bessere Optimierungs- und Parallelisierungsmöglichkeiten für Prozessoren. [4] [6]

4.2.3 Late Materialization

Eine entscheidende Frage bei Column Stores ist: Wann werden die Attribute aus den Spalten wieder zu Tupeln (für das Ergebnis) zusammengesetzt? bzw. Wie lange wird mit den Daten in Spaltenform gearbeitet? Bei der „Early Materialization“ sind die Daten nur spaltenweise gespeichert und die Verarbeitung (Tupel-Operation) erfolgt wie im Row Store. Um den Vorteil der Column Stores auszunutzen ist es daher sehr wichtig, dass die Tupel so spät wie möglich materialisiert werden (Late Materialization). Damit werden keine Tupel umsonst materialisiert (wenn diese nicht im Ergebnis sind) und der Speicher und die Operationen werden nicht mit unnötigen Attributwerten belastet. Außerdem können sowohl die Kompression als auch die Block Iteration optimal genutzt werden. Der Performance-Vorteil der Late Materialization steigt mit der Selektivität der Daten und Anfragen. [4] [6]

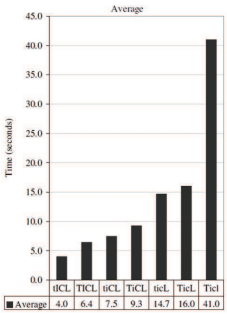
4.2.4 Invisible Join

Der in [4] entwickelte „Invisible Join“ ist ein alternativer Ausführungsplan für Queries mit Fremdschlüssel-Primärschlüssel-Joins in Column Stores mit Starschema. Statt bei mehreren Joins zu den Dimensionen ein Filterattribut in der WHERE-Klausel anzuwenden und danach die nächste Dimensionstabelle an der Ergebnis zu joinen, werden hier vorab mit den jeweiligen Filterattributen Hash-Tabellen aus jeder Dimensionstabelle erstellt, AND-verknüpft und auf die Faktentabelle angewandt. Somit ist kein echter Join zu den Dimensionstabellen nötig. Mit der gefilterten Faktentabelle werden die Tupel anschließend mit Hilfe der Fremdschlüssel zusammengesetzt. [4]

4.2.5 Performance

Um zu untersuchen, welchen Einfluss die einzelnen Optimierungstechniken auf die Performance des Column Store haben, wurden diese einzeln im System deaktiviert. Die Ergebnisse sind im Diagramm rechts dargestellt und erklären sich wie folgt:

Block Iteration und Invisible Join (T = tuple-at-a-time processing, t = block processing, I = invisible join enabled, i = invisible join disabled): Performance-Vorteil: ca. Faktor 1,5

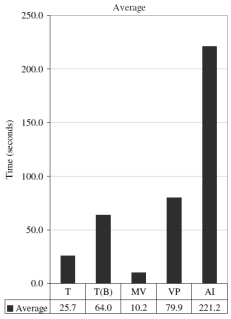


spaltenspezifische Kompression (C = compression enabled, c = compression disabled): Der Performance-Vorteil der Kompression variiert stark abhängig von den Begebenheiten. Durchschnittlich ist er um den Faktor 2, wenn möglich aber auch bis zu Faktor 10.

Late Materialization: (L = late materialization enabled, l = late materialization disabled): Die Late Materialization ist sehr entscheidend für die Performance von Column Stores. Vergleicht man die letzten beiden Balken ergibt sich hierfür ein Performance-Vorteil vom Faktor 3. [4] [6]

4.3 Simulation eines Column Stores in einem Row Store DBMS

Eine wesentliche Frage, die sich bei der Einführung von Column Stores stellt, ist, ob oder wie gut man einen Column Store in einem herkömmlichen Row Store DBMS nachbauen könnte. Dazu wurden in [4] verschiedene Ansätze gewählt und derern Performance betrachtet (T = traditional, T(B) = traditional (bitmap), MV = materialized views, VP = vertical partitioning, AI = all indexes). Dabei stellte sich heraus, dass nur der Ansatz der Materialized Views (MV) die Performance gegenüber dem nativen Row Store (T) verbessern konnte. [4]



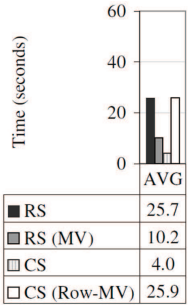
4.3.1 Vertikale Partitionierung

Hier wurden für jede Spalte zweispaltige Tabellen (Key, Value) angelegt, um die spaltenorientierte Speicherung nachzuahmen. Die schlechte Performance kommt aufgrund sehr großer Tuple Overheads zustande, da jeweils die Schlüssel der zweispaltigen Tabellen mitgeschleppt werden müssen. Ein Größenvergleich zeigt das Problem deutlich: Eine zweispaltige Tabelle (entspricht einer Saplte) ist hier ca. 1 GB groß. Im nativen Row Store ist die gesamte Fakten-Tabelle (17 Spalten) komprimiert nur 4 GB groß (im echten Column Store: 2,3 GB). Weiterhin müssen sehr viele Joins ausgeführt werden. [4]

4.3.2 Materialized Views

Hier wurden vorab für alle Queries materialsierte Sichten erstellt, welche nur die Spalten enthalten, die in der jeweiligen Anfrage verwendet werden. Dieser Ansatz ist vergleichbar mit einem Column Store mit Early Materialization und liefert aufgrund des minimalen Datendurchsatzes die beste Performance, da nur benötigte Spalten gelesen werden gelesen. Allerdings entsteht ein enormer zusätzlicher Speicherplatzbedarf für die Sichten und die Praxistauglichkeit ist stark zu bezweifeln, da jegliche Anfragen vorher bekannt sein müssen. [4]

4.4 Zusammenfassung



Die in Abschnitt 4.3 beschriebenen Tests zeigen, dass die Nachahmung eines Column Stores in einem Row Store DBMS allgemein sehr aufwendig ist und im besten (nicht praxistauglichen) Fall eine durchschnittliche Anfragezeit von 10s erreicht. Der native Row Store benötigt im Schnitt 25s. Diese Zahlen werden um Größenordnungen vom nativem Column Store mit 4s geschlagen, was erneut die Performance-Vorteile von Column Stores in DWH-Anwendungen deutlich zeigt. Bei Deaktivierung aller Optimierungstechniken erreicht auch der Column Store nur 41s (ca. Faktor 2 zu nativem Row Store wegen Tupel-Rekonstruktion), was deren Wichtigkeit unterstreicht. Einen Column Store macht also mehr aus als nur der reduzierte Datendurchsatz. Ein weiterer Test mit denormalisiertem Tabellenschema (d.h. nur 1 große Fakten-Tabelle mit direkt angehangenen Dimensionsattributen statt Fremdschlüsseln) zeigte, dass dies nur in Verbindung mit maximaler Kompression sinnvoll ist. [4]

5 Systeme

Fast alle großen DBMS-Hersteller werben mit Column Stores. Jedoch sind die Implementierungen sehr unterschiedlich. Oft ist der spaltenbezogenen Ansatz nur im Arbeitsspeicher und nicht auf der Festplatte realisiert (wie auch in Oracle 12c). Bei MS SQL Server gibt es bespielsweise den sogenannten „Columnstore-Index“, der auf eine Tabelle gelegt werden kann und eine „bis zu zehnfache Abfrageleistung“ und „bis zu siebenfache Datenkomprimierung“ verspricht. [12] Es gibt dagegen nur sehr wenige “echte” physische Column Stores. Führend sind hier Vertica (hp), SybaseIQ (SAP), LucidDB und mit etwas alternativem Ansatz monetDB. [1]

5.1 C-Store / Vertica

C-Store wurde 2008 im Rahmen einer Dissertation [5] am MIT entwickelt. Das physische Datenmodell beruht auf überlappenden Projektionen (Teilmengen der Spalten einer Relation), welche unterschiedlich sortiert sind. Es erfolgt eine spaltenweise Speicherung und eine aggressive Komprimierung. Die Tupel werden durch paralleles Scannen mit Reißverschluss-Prinzip zusammengesetzt. Außerdem wird auf die lange Beibehaltung der Kompression geachtet. Änderungen werden in einer vorgelagerten schreiboptimierter Datenbank gespeichert und regelmäßig in den komprimierten Datenbestand synchronisiert.

Vertica ist die kommerzielle Weiterentwicklung von C-Store und wirbt auf der eigenen Website mit folgender Aussage: “A true column store, like Vertica, must have the following 4 features: Columnar Storage, Compression and Retrieval; Columnar on Disk, not just In-memory; Columnar Optimizer and Execution Engine; Optimized Loads and Transactions“. [10] [1]

Zusammenfassung

Wie Kapitel 1 zeigt, gibt es verschiedene physische Datenmodelle (zeilen- oder spaltenorientiert). Die Entscheidung, welches Modell zur Anwendung kommt sollte immer am Zweck der Datenbank ausgerichtet werden. Column Stores sollten dabei nicht für OLTP verwendet werden, bieten aber einen enormen Performance-Vorteil für OLAP bzw. DWH-Anwendungen.

Die Optimierung von Column Stores u.a. durch Kompression und Late Materialization ist sehr entscheidend für deren Performance, um die Vorteile der Spaltenorientierung voll auszunutzen. Ein Nachbau eines Column Stores in einem Row Store DBMS ist nicht zu empfehlen.

Aktuelle Systeme, die Column Stores implementiert haben, sind Vertica (C-Store) und SybaseIQ. Da das Thema insgesamt noch relativ jung ist sind für die Zukunft weitere Entwicklungen und Optimierungen zu erwarten

Literaturverzeichnis

[1] Daniel Bößwetter. Spaltenorientierte datenbanken. *Informatik-Spektrum*, 33/2010.

[2] Kai-Uwe Sattler and Gunter Saake. *Datenbank-Implementierungstechniken (Vorlesungsskript)*. TU Ilmenau FG Datenbanken & Informationssysteme, Universität Magdeburg Institut für Technische und Betriebliche Informationssysteme, 2012.

[3] Samuel R. Madden. Column oriented database (video from edx course: Mitx: 6.00x introduction to computer science and programming). https://www.youtube.com/watch?v=mRvkikVuojU, 2013.

[4] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Columnstores vs. rowstores: How different are they really? 2008.

[5] Daniel J. Abadi. Query execution in column-oriented database systems. *PhD Thesis (MIT)*, 2008.

[6] David Fekete. Optimierungstechniken in column stores. *Datenbank-Spektrum*, 11/2011.

[7] Spaltenorientierte Datenbank. Wikipedia. http://de.wikipedia.org/wiki/ Spaltenorientierte_Datenbank, 2015.

[8] June Tong. Demystifying columnar databases (introduction to columnar databases, and calpont infinidb). https://www.youtube.com/watch?v=wYl_YxqTof4, 2012.

[9] Olaf Herden. *Spaltenorientierte Speicherung.* in T. Kudraß (Hrsg): Taschenbuch Datenbanken, Carl Hanser Verlag, 2. Auflage, 2015.

[10] Vertica. Column store vs. column store. http://www.vertica.com/2010/04/22/column-store-vs-column-store/, 2010.

[11] IBM DB2. In spalten organisierte tabellen. https://www-01.ibm.com/support/know ledgecenter/SSEPGG_ 10.5.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0060592.html, 2015.

[12] Microsoft SQL Server 2014. Beschreibung von columnstore-indizes. https://msdn.microsoft.com/de-de/library/gg492088.aspx, 2015.