FOM Hochschule für Oekonomie & Management Essen
Standort München
Berufsbegleitender Studiengang zum B.Sc. Wirtschaftsinformatik

Seminararbeit

Die In-Memory-Datenbank SAP HANA

Eingereicht von:

Oliver Kurmis

Matrikel-Nr: 328091

Betreuer: Prof. Dr. Klaus Wilderotter

Abgegeben am:

31. August 2016

Erarbeitet im:

7. Semester

Inhaltsverzeichnis

Abkürzungsverzeichnis **II**

Abbildungsverzeichnis **III**

Tabellenverzeichnis **III**

1 Einleitung **1**

2 Motivation und Entwicklung von HANA **1**

2.1 Problemstellung 1
2.2 Entwicklung moderner Hardware 2
2.3 Spaltenorientierung 3
2.4 Wörterbuch-Kodierung und Datenkompression 4
2.5 Parallelverarbeitung 6

3 Experimenteller Leistungs-Vergleich von zeilen- und spaltenorientierter Speicherung **7**

3.1 Testaufbau 7
3.2 OLAP - analytische Abfragen 9
3.3 OLTP - Transaktionale Abfragen 9
3.4 Diskussion 9

4 Fazit und Ausblick **11**

Literatur **13**

Anhang 1: Benchmark-Daten **14**

Abkürzungsverzeichnis

CS	Column-Store, spaltenorientierter Datenspeicher
CSV	Comma Separated Values, Plaintext-Datenformat
DBMS	Datenbankmanagementsystem
DRAM	Dynamischer RAM, 1 Transistor + 1 Kondensator pro Bit, langsamer als SRAM
ECC	SAP Enterprise Core Component
HDD	Hard Disk Drive, Laufwerk mit Magnetspeicher-Platten
HCP	HANA Cloud Platform
OLAP	Online Analytical Processing, Datenanalyse von Geschäftsdaten
OLTP	Online Transaction Processing, Online-Transaktionsverarbeitung
RAM	Random Access Memory, Arbeitsspeicher
RS	Row-Store, zeilenorientierter Datenspeicher
SSD	Solid State Drive, Halbleiterlaufwerk (Flash-EPROM-Speicher)
SRAM	Statischer RAM, 6 Transistoren pro Bit, schneller und teurer als DRAM

Abbildungsverzeichnis

1 Zugriffsmuster bei Row-Store und Column-Store 4
2 Prinzip der Wörterbuch-Kodierung 5
3 Parallelverarbeitung im Column-Store 6
4 Import von CSV-Daten mit Eclipse 8
5 Performance Column-Store / Row-Store 10

Tabellenverzeichnis

1 Einfluss von Indizes auf die Performance bei Row-Store und Column-Store 10

1 Einleitung

SAP HANA ist eine Datenbank mit In-Memory-Technik, die von der Firma SAP und dem Hasso-Plattner-Institut ab dem Jahr 2006 entwickelt wurde und seit 2011 kommerziell verfügbar ist. Inzwischen ist HANA bei bei mehreren tausend SAP-Kunden im Einsatz.
Die vorliegende Arbeit gibt eine Einführung in einige wesentlichen Techniken von HANA, und arbeitet die Vorteile der Spaltenorientierung anhand von experimentellen Messungen heraus.
Im zweiten Abschnitt wird kurz auf die Geschichte und Entwicklung von HANA eingegangen und einige Kernaspekte der In-Memory-Technologie näher vorgestellt, wie die spaltenorientierte Speicherung und Wörterbuch-Kodierung.
Im Abschnitt drei wird anhand von konkreten Benchmarks die Leistung von zeilenorientierter und spaltenorientierter Speicherung bei verschiedenen Einsatzszenarien verglichen.
Im letzten Abschnitt wird ein kritisches Fazit gezogen und ein Ausblick auf weitere mögliche Untersuchungen gegeben. Für die Erstellung dieser Arbeit wurde auf relevante Fachliteratur zurückgegriffen und im Internet recherchiert. Die Benchmark-Messungen wurden auf leistungsfähigen Server-Computern durchgeführt, die von SAP im Rahmen der SAP HANA Cloud Platform zur Verfügung gestellt werden.

2 Motivation und Entwicklung von HANA

2.1 Problemstellung

Seit dem Aufkommen der ersten Datenbank-Systeme in den 70er-Jahres ist eines der grossen Probleme die geringe Zugriffsgeschwindigkeit der Datenspeicher. Bei den hierfür eingesetzten Festplatten (HDD) liegt die Latenz für einen Datenzugriff um viele Grössenordnungen über der Latenz für Zugriffe auf den Arbeitsspeicher (RAM) des Computers. Im Laufe der Jahrzehnte hat sich diese Diskrepanz immer weiter verschärft. Seit den 70er-Jahren hat sich die Leistung von CPUs und des RAM millionenfach erhöht, aber die Zugriffsgeschwindigkeit von Festplatten hat sich nur von 100 ms auf ca. 10 ms verringert. In diesen 10 ms Wartezeit vergehen bei einer modernen CPU ca. 30 Mio. Takte, könnte eine Mehrkern-CPU hunderte Millionen Operationen ausführen und 1 GB Daten aus dem Arbeitsspeicher lesen.[1]
Seit jeher wird in der Datenbank- und Anwendungsentwicklung versucht, dieses Missverhältnis durch verschiedene technische Maßnahmen abzumildern, etwa durch Caching auf den mehreren Ebenen, durch Anlegen von Indizes mit Baumstruktur

[1] vgl. [Intel (2016)]

oder auch auf Seite der Anwendungsprogramme, z.B. durch aufwendiges Vorberechnen von diversen Datenaggregaten. All diese Zusatzmaßnahmen erhöhen die Komplexität der IT-Systeme, den Entwicklungsaufwand, die Fehleranfälligkeit und binden wertvolle Ressourcen in den verschiedensten Bereichen.

2.2 Entwicklung moderner Hardware

In den 2000er-Jahren fanden zunehmend Multi-Core-Prozessoren Verbreitung, also CPUs mit mehreren Rechenkernen auf einem Chip. Gleichzeitig wurde Arbeitsspeicher so günstig, dass Systeme mit mehreren Gigabyte RAM erschwinglich wurden. So wurde es erstmals theoretisch möglich, eine ganze Unternehmensdatenbank kommplett im RAM vorzuhalten und auf den Festplattenspeicher als primären Datenspeicher zu verzichten. Ab 2006 wurde bei der SAP und dem Hasso-Plattner-Institut (HPI) an dieser Idee gearbeitet, das Potential der verfügbaren Hardware sollte optimal ausgeschöpft werden ohne durch langsame Festplatten ausgebremst zu werden.[2] Gleichzeitig würde sich die Komplexität der Systeme deutlich reduzieren und dadurch weitere Ressourcen frei werden lassen.
Zu diesem Zeitpunkt hatte man bei SAP bereits Erfahrung gesammelt mit *TREX*, einer In-Memory-Datenbank mit Column-Store zur Text-Analyse, mit *P*Time*, einer einer In-Memory-Datenbank mit Row-Store und mit *MaxDB*, einer konventionellen relationalen Enterprise-Datenbank auf der auch SAP ECC betrieben werden kann.[3] Das HPI erhielt Zugang zum Quelltext dieser Datenbank-Systeme und entwickelte einen Prototyp der In-Memory-Datenbank für den Unternehmenseinsatz mit dem Namen *SanssouciDB*. Dieser vielversprechende Prototyp wurde bei SAP zu einem marktfähiges Produkt weiterentwickelt, 2010 der Öffentlichkeit vorgestellt und als SAP HANA ab 2011 kommerziell vertrieben. Anfangs war HANA nur als Appliance erhältlich, also eine Kombination von genau aufeinander abgestimmter Software und Hardware. Inzwischen kann HANA aber auch auf zertifizierten Servern betrieben werden (Tailored Data Center Integration). Typische HANA-Server haben heute 128 GB bis 12 TB RAM und bis zu 8 Prozessoren mit insgesamt einigen Dutzend bis über 100 CPU-Cores.[4]
Um die Persistenz der Daten zu gewährleisten kann auch bei einer In-Memory-Datenbank nicht ganz auf herkömmliche Massenspeicher verzichtet werden. Festplatten oder SSD sind bei HANA aber nicht mehr der primäre Speicher sondern werden benötigt, um regelmäßig Snapshots der Datenbank anzulegen und um alle Schreibzugriffe in Log-Dateien abzuspeichern. So kann nach einem Stromausfall der Zustand der Datenbank aus dem letzten Snapshot und allen Logs seit diesem

[2] vgl. [Plattner, H., Leukert, B. (2015)] S.3
[3] vgl. [Plattner, H., Leukert, B. (2015)] S.5
[4] vgl. [SAP (2016)]

Zeitpunkt zuverlässig wieder hergestellt werden.
Mit dem Wegfall des relativ langsamen Massenspeichers ist nun der Arbeitsspeicher der neue Flaschenhals für den Datentransport. Im Vergleich zu den 10 ms einer Festplatte ist der Zugriff auf DRAM mit ca. 100 ns zwar um viele Grössenordnungen schneller, jedoch beträgt die Zykluszeit aktueller Prozessoren nur etwa 0,3 ns. Bei einem Zugriff auf den DRAM-Arbeitspeicher muss die CPU also immer noch mehrere hundert Takte auf das Ergebnis warten. Aus diesem Grund sind zwischen CPU und DRAM-Arbeitsspeicher mehrere Stufen von kleinen aber schnellen Cache-Speichern aus SRAM geschaltet. Von dem schnellsten Level-1-Cache (L1) kann bei einem Treffer (Cache-Hit), also wenn sich die Daten der angefragten RAM-Adresse im Cache befinden, mit 0,5 ns Latenz gelesen werden. Auf den grösseren Level-2-Cache kann noch mit 7 ns Verzögerung zugegriffen werden. [5] Meist kommt auch noch ein L3-Cache zum Einsatz, sodass pro CPU bis zu 60 MB Cache zur Verfügung stehen.[6] Bei einem Cache-Miss, also wenn sich die angeforderten Daten noch nicht im Cache befinden, muss die CPU aber mindsten 100 ns auf den DRAM warten, ein Cache-Miss sollte also möglichst vermieden werden.
Der Cache wird in Blöcken zu je 64 Byte, den sogenannten Cache-Lines, verwaltet. Diese Cache-Lines werden immer zusammen am Stück aus dem Arbeitsspeicher in den Cache geladen, zwar mit der entsprechenden Latenz, aber einer hohen Bandbreite. Daten innerhalb dieser Cache-Line können also von der CPU sehr schnell gelesen werden. In der Zwischenzeit kann aus dem Arbeitsspeicher schon die nächste Cache-Line geladen werden (Prefetching). Zusammenhängende Speicherbereiche können so mit einer Geschwindigkeit von 4 MB / ms von einem Core gelesen und gescannt werden. Bei einer 15-Kern-CPU ist das eine Scangeschwindigkeit 60 GB pro Sekunde, also bei einem 8-Sockel-System 480 GB pro Sekunde.[7]

2.3 Spaltenorientierung

Eine Tabelle in einer Datenbank ist eine theoretische, 2-dimensionale Struktur, die auf den linear addressierten Speicher eines Computers abgebildet werden muss. Diese Abbildung kann sinnvoll auf zwei Arten geschehen: zeilenorientiert und spaltenorientiert. Bei der zeilenorientierten Speicherung werden alle Attribute eines Datentupels hintereinander abgelegt, z.B. `[Max,Huber,m,81234,München,Müllerstraße,12]` und daran anschliessend das nächste Tupel. Diese Art der Speicherung orientiert sich im Prinzip noch an der Datenverarbeitung mit Lochkarten oder Magnetbändern, wo ein schneller wahlfreier Zugriff wie bei Festplatten nicht möglich war.
Bei der spaltenorientierten Speicherung hingegen wird ein Attribut aller Tupel zu-

[5] vgl. [Plattner/HPI (2015)] S.29
[6] vgl. [Intel (2016)]
[7] vgl. [Plattner/HPI (2015)] S.29

sammen in einem Attribut-Vektor abgelegt, z.B. `[Annett,Max,Stefan,Susanne]` und daran anschliessend die anderen Attributvektoren. Die hohe Leseleistung von zusammenhängenden Speicherbereichen, bzw. die schlechte Leistung bei verteilten Zugriffen sind der Grund, warum analytische Datenbank-Abfragen bei spaltenorientierter Speicherung (Column-Store) deutlich schneller sind als bei zeilenorientierter Speicherung (Row-Store). Bei dieser Art von Abfragen müssen üblicherweise große Mengen eines oder mehrer Attribute verarbeitet werden, z.B. mit Filter- oder Aggregationsfunktionen. Demgegenüber ist die zeilenorienterte Speicherung (Row-Store) im Vorteil, wenn viele oder alle Attribute (Spalten) eines oder mehrere Datensätze gelesen oder geschrieben werden (Row Operation). Abbildung 1 veranschaulicht die Speicherzugriffsmuster dieser beiden Betriebsmodi bei Row- und Column-Store.

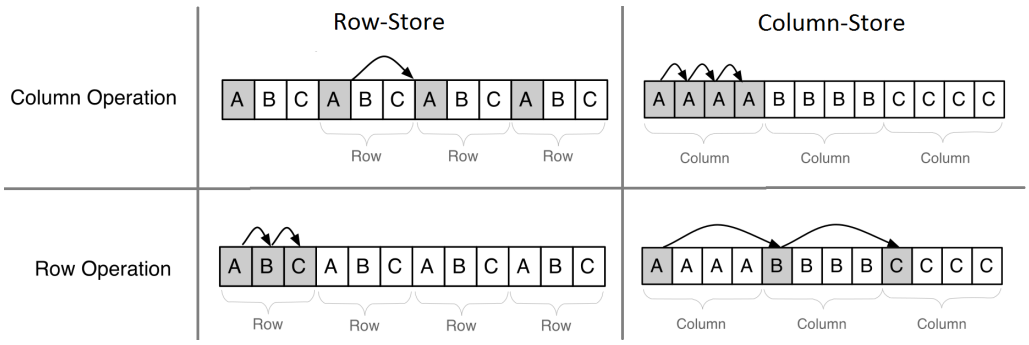

Abbildung 1: Zugriffsmuster bei Row-Store und Column-Store[8]

2.4 Wörterbuch-Kodierung und Datenkompression

Bei einer In-Memory-Datenbank wie SAP HANA ist wie bereits beschrieben nicht mehr die Festplatte (HDD oder SSD) der Flaschenhals sondern der Arbeitsspeicher (RAM). Aus dem Arbeitsspeicher können die Daten jedoch nicht so schnell geladen werden, wie der Prozessor (CPU) sie bearbeiten kann. Bei den heute üblichen Multicore-Prozessoren, mit vielen Rechenkernen pro Prozessor-Chip verstärkt sich dieser Effenkt noch einmal deutlich.

Die Datenbank arbeitet also um so langsamer, je mehr Daten vom Hauptspeicher gelesen (bzw. dorthin wieder geschrieben) werden müssen. Am schnellsten kann die CPU arbeiten, wenn die Daten aus dem internen Cache gelesen werden können, der jedoch in der Grösse beschränkt ist. Daher wird mit verschiedenen Techniken versucht, die Datenmenge zu reduzieren. Eine dieser Techniken ist die Wörterbuch-Kodierung (engl. dictionary encoding). Dabei wird in den Tabellenspalten nicht der Datenwert selbst gespeichert, sondern eine Integer-Zahl, die diesen Wert eindeutig repräsentiert. Zusätzlich gibt es für jede Spalte ein eigenes Wörterbuch (dictionary),

[8] Quelle: eigene Darstellung nach [Plattner/HPI (2015)] S.62

mit dem die Integer-Werte wieder dem eigentlichen Datenwert zugeordnet werden können. Sollen z.B. Personendaten wie Name und Adresse in einer Tabelle abgelegt werden, kommen viele Werte mehrfach vor, es gibt also Redundanzen im Datenbestand. Am Beispiel einer Spalte mit Vornamen wird in Abbildung 2 das Prinzip der Kodierung mit einem Wörterbuch verdeutlicht.

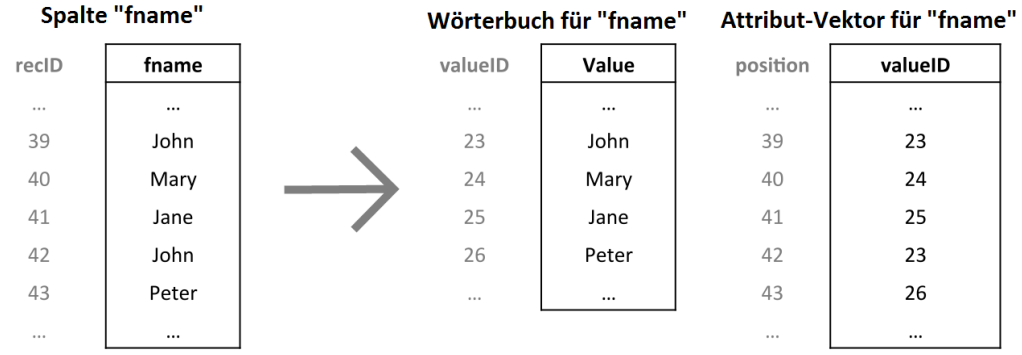

Abbildung 2: Prinzip der Wörterbuch-Kodierung[9]

So gibt es z.B. in Deutschland 11116 unterschiedliche Gemeinden. [10] Mit einer Wörterbuch-Kodierung kann jede Gemeinde mit einer 14-Bit-Zahl dargestellt werden, denn $2^{14} = 16384$. Wenn die Ortsnamen als normaler Text abgespeichert würden, so müssten dafür 32 Byte vorgesehen werden (längster Ortsname in Deutschland). Bei einer Kundendatei von 20 Mio. Personen wäre das Datenvolumen der Spalte Ort 20 Mio. * 32 Byte = 640 MB.
Mit Wörterbuch-Kodierung hingegen werden nur 20 Mio. * 14 Bit = 280 MBit = 35 MB benötigt. Hinzu kommt noch das Wörterbuch mit maximal 11116 * 32 Byte = 355712 Byte = 0,35 MB. Der Komprimierungsfaktor in diesem Beispiel wäre also 640MB/35,35MB = 18,1.
Allgemein kann für geschäftliche Daten wie in einem SAP-System von einem Komprimierungsfaktor von etwa 1:5 ausgegangen werden. Duch den Wegfall von Indizes und vorberechneten Daten-Aggregaten werden etwa weitere 50% Datenvolumen eingespart. Ingesamt ist das Datenvolumen also nur ca. 1/10 einer konventionellen relationalen Datenbank wie Oracle, MaxDB oder MS-SQL-Server. [11]
Mit klassischen Methoden zur Datenkompression kann das benötigte Datenvolumen der kodierten Spalten noch weiter reduziert und somit die Geschwindikeit erhöht werden. Bei HANA kommen dazu leichtgewichtige Algorithmen zum Einsatz wie die Lauflängenkodierung (run-length encoding). [12] Komplexe Algorithmen wie zip lohnen sich nicht, da der Aufwand zur Dekomprimierung grösser ist als die erzielte

[9] Quelle: eigene Darstellung nach [Plattner/HPI (2015)] S.39
[10] vgl. [Statista (2014)]
[11] vgl. [Plattner, H., Leukert, B. (2015)] S.22
[12] vgl. [Plattner/HPI (2015)] S.47

Bandbreiteneinsparung. Für manche Kompressionsmethoden muss eine Spalte sortiert sein, wobei eine Tabelle natürlich immer nur nach einer Spalte sortiert sein kann. Der zusätzliche Vorteil einer sortierten Spalte ist, daß der Zugriff auf ein bestimmtes Element daraus mittels binärer Suche in logarithmischer Zeit (Komplexität O(log n)) erfolgen kann. Es ist also weder ein zusätzlicher Index von ein voller Tablescan erforderlich.

2.5 Parallelverarbeitung

Moderne Prozessoren enthalten normalerweise mehrere Rechenkerne (z.B. aktuelle Intel Xeon bis zu 24 physische Kerne)[13] und Server-Computer können mehrere Prozessoren besitzen[14]. Server mit 80 Rechernkernen und mehr sind daher heute nicht unüblich[15]. SAP HANA kann das Potential dieser Rechenleistung voll ausschöpfen, da es von Beginn an mit Fokus auf Parallelverarbeitung entwickelt wurde. Wenn z.B. bei einer SQL-Abfrage mehrere Spalten ausgewertet werden müssen, so kann jeweils ein Kern eine Spalte bearbeiten. Aber auch wenn nur eine Spalte bearbeitet wird können mehrere Kerne gleichzeitig daran arbeiten und z.B. die Summe berechnen. Die Einzelergebnisse werden danach zum Endergebnis zusammengeführt (Abbildung 3). Sehr grosse Tabellen im Terrabyte-Bereich können sogar über mehrere Server

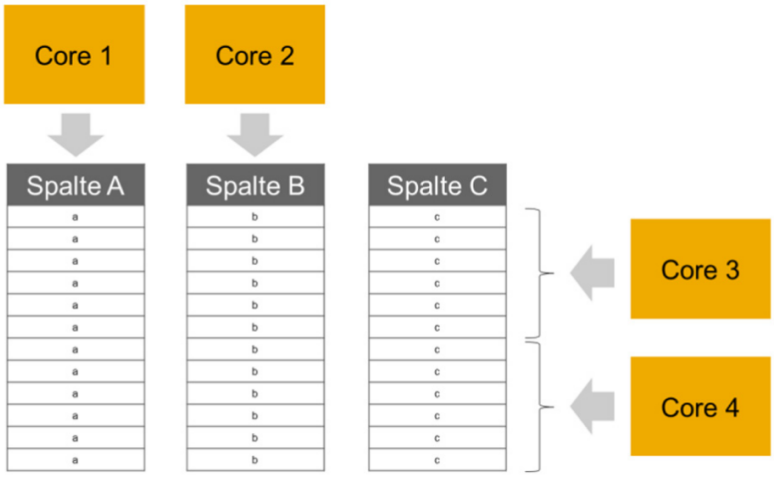

Abbildung 3: Parallelverarbeitung im Column-Store[16]

(sog. Nodes) verteilt werden (Scale-out). Diese sind mit einem Hochgeschwindigkeits-Netzwerk verbunden und tauschen untereinander Abfragen, Daten und Ergebnisse aus. Die Aufteilung kann spaltenweise oder zeilenweise (Partitionierung) erfolgen. Auch eine gemischte Aufteilung ist möglich: z.B. Spalten U, V und W sind klein

[13] vgl. [Intel (2016)]
[14] vgl. [Intel (2014)]
[15] vgl. [Plattner/HPI (2015)]
[16] Quelle: Prassol, P. (2015)

genug und passen auf Server 1, Spalte X ist gross und liegt auf Server 2 und Spalte Y ist so gross, dass sie via Partitionierung auf Server 3 bis 10 verteilt wird. So können ohne Weiteres 1000 Rechenkerne eine Abfrage gleichzeitig bearbeiten die Verarbeitungsleistung steigt nahezu linear.[17]

3 Experimenteller Leistungs-Vergleich von zeilen- und spaltenorientierter Speicherung

3.1 Testaufbau

Für den Geschwindigkeisvergleich von Row-Store und Column-Store wurde die frei zugängliche Entwicklungs- und Testumgebung der SAP HANA Cloud Platform (HCP) eingesetzt.[18] Hier kann eine bis zu 1 GB grosse HANA-Datenbank auf leistungsfähigen Server-Computern der SAP betrieben werden. Für die Tests wurden mit einem Ruby-Script [19] genau 5 Mio. Personendatensätze mit Zufallsdaten generiert und in einer CSV-Datei abgespeichert. Die Datensätze bestehen aus Name, Anschrift, Email, Größe(cm), Gewicht(kg) und Geburtsdatum, wobei Größe und Gewicht jeweils normal-verteilt sind und miteinander korrelieren:

```
1;Fatma;Maier;Violastr.;1;19760;Mainburg;fatma_maier@greenfelderdamore.us;155;57;1974-06-17
2;Marc;Elsner;Ferdinandstr.;53;57315;Alzey;marc_elsner@anderson.biz;183;88;1931-02-10
3;Noelle;Hahn;Beckstr.;6;57109;Freilassing;hahn_noelle@kreiger.co.uk;168;44;1982-01-28
4;Mona;Wagner;Connerstr.;14;48256;Leichlingen;mona.wagner@emardpadberg.name;171;73;1942-09-13
```

Für den Zugriff auf die Datenbank kam die Entwicklungsumgebung *Eclipse* in Verbindung mit den *SAP HANA Cloud Platform Tools* zum Einsatz. [20]
In der Datenbank wurden 2 Tabellen für die Personendaten angelegt, eine Tabelle mit Row-Store (RPERSON) und eine Tabelle mit Column-Store (CPERSON):

```
CREATE COLUMN TABLE "CPERSON" (
    "ID" INTEGER CS_INT NOT NULL ,
    "FIRST_NAME" NVARCHAR(20) NOT NULL ,
    "LAST_NAME" NVARCHAR(20) NOT NULL ,
    "STREET" NVARCHAR(20) NOT NULL ,
    "STR_NUM" SMALLINT CS_INT NOT NULL ,
    "ZIP" VARCHAR(5) NOT NULL ,
    "CITY" NVARCHAR(30) NOT NULL ,
    "EMAIL" NVARCHAR(60) NOT NULL ,
```

[17] vgl. [SAP (2014)]
[18] https://hcp.sap.com
[19] https://www.kurmis.com/csv-faker
[20] https://tools.hana.ondemand.com

```
    "HEIGHT" TINYINT CS_INT NOT NULL ,
    "WEIGHT" TINYINT CS_INT NOT NULL ,
    "BIRTHDAY" DAYDATE CS_DAYDATE NOT NULL ,
    PRIMARY KEY ("ID")) UNLOAD PRIORITY 5 AUTO MERGE
;
```

Die Tabelle mit Row-Store wurde analog mit `CREATE ROW TABLE` angelegt (`CS_INT` und `CS_DAYDATE` können auch weggelassen werden, diese Datentypen werden von HANA beim Erstellen der Tabellen automatisch verwendet). Als Tabellenbezeichner wird im Folgenden nur `PERS` verwendet, und soll als Platzhalter für CPERSON und RPERSON verstanden werden.
Die CSV-Datei mit den 5 Mio. Test-Datensätze wurde mit Hilfe von *Eclipse* und den *SAP HANA Cloud Platform Tools* in eine der beiden Tabellen übertragen (Abbildung 4) und anschließend von dort mittels SQL in die zweite Tabelle kopiert, so dass beide Tabellen exakt die gleichen Daten enthalten.

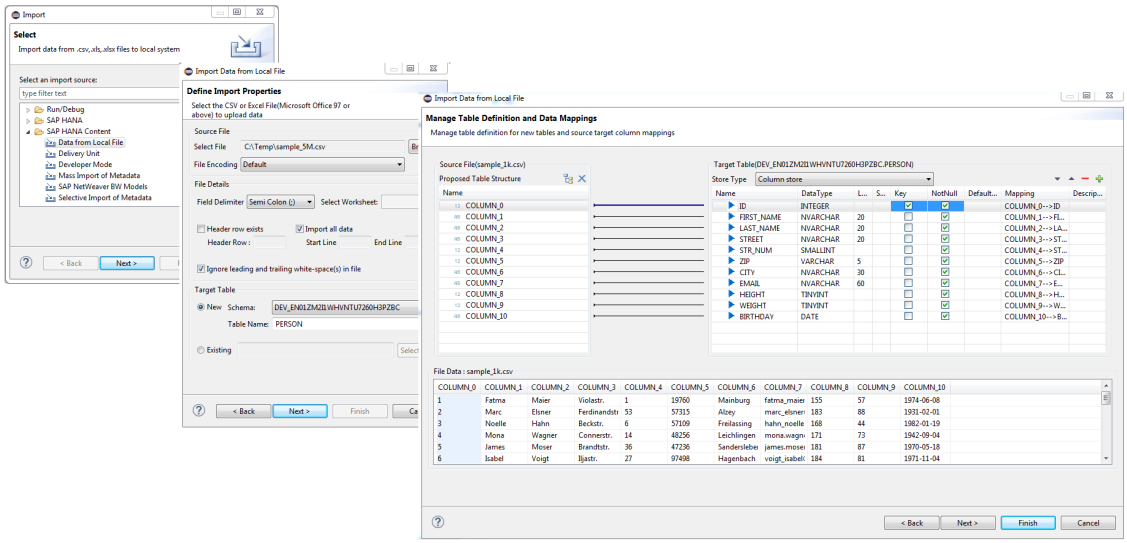

Abbildung 4: Import von CSV-Daten mit Eclipse[21]

Die eigentlichen Datenbank-Abfragen für den Benchmark wurden aus Eclipse heraus auf jeder Tabelle jeweils 4 mal ausgeführt und von deren Ausführungszeiten auf dem Server (server processing time) t1-t4 der Mittelwert Øt bestimmt.[22] Die gemessenen Werte lagen durchweg im Millisekunden-Bereich und waren mit geringen Schwankungen reproduzierbar. Es wurden verschiedene SQL-Abfragen vom analytischen Typ (OLAP) und vom transaktionalen Typ (OLTP) gestellt und gemessen, jedoch keine Schreibzugriffe wie INSERT oder UPDATE. Anschließend wurde jeweils noch ein Index zu den Spalten `HEIGHT` und `WEIGHT` angelegt (Abfragen Q und R) und ausgewählte OLAP- und OLTP-Abfragen noch einmal zum Vergleich ausgeführt.

[21] Quelle: eigener Screenshot
[22] siehe Anhang 1: Benchmark-Daten

3.2 OLAP - analytische Abfragen

Die analytischen Datenbank-Abfragen verwenden typischerweise Aggregationsfunktionen wie COUNT(), SUM(), AVG(), MIN(), MAX() und arbeiten nur auf wenigen ausgewählten Spalten, die Ergebnismenge ist in der Regel klein.
Mit Fokus auf diese Art von Abfragen wurde der Column-Store von SAP HANA entwickelt. Diese Abfragen auf die Column-Store-Tabelle werden dementsprechend auch schneller von dem DBMS bearbeitet als Abfragen auf die Row-Store-Tabelle.

3.3 OLTP - Transaktionale Abfragen

Transaktionale Abfragen lesen in der Regel viele Spalten oder alle Spalten einer Tabelle (SELECT *) und haben einen einzelnen Datensatz oder mehrere Zeilen als Ergebnismenge. Diese Art von Abfragen werden im laufenden Geschäftsbetrieb verwendet um z.B. Detaildaten zu einem Kunden oder eine Liste von Kunden anzuzeigen. Hier ist der Row-Store vorteilhaft, da die einzelnen Attribute eines Datensatzes im Speicher nah beieinander liegen und daher schnell darauf zugegriffen werden kann.
Die Selektion eines einzelnen Datensatzen erfolgt bei beiden Tabellen so schnell (<1ms), dass ein sinnvoller Vergleich nicht möglich ist. Deswegen wurden die Abfragen so gewählt, dass sie als Ergebnismenge 1000 Tupel zurückliefern, wie z.B. `select * from PERS where id <= 1000` (Abfragen O,P,U,V). Bei den verwendeten Abfragen erfolgt die Selektion über den Primärschlüssel `ID` und ist daher sehr schnell (Nutzung eines Index).

3.4 Diskussion

Die analytischen OLAP-Abfragen werden von dem Column-Store etwa eine Grössenordnung schneller bearbeitet als von dem Row-Store. Der Beschleunigungsfaktor variiert aber stark zwischen den einzelnen Abfragen und reicht von 1,4 (Abfrage E: `select sum(weight) from PERS`) bis zu Faktor 170 (Abfrage M: `select count(*) from PERS where height = 210 and weight=110`). Hier zeigt sich, dass die Spalten in einem Column-Store viel schneller gescannt werden können.
Bei Abfrage N (`select * from PERS where height=210 and weight=110`) werden über 300 Tupel mit allen Spalten als Ergebnis zurückgeliefert und trotzdem ist hier der Column-Store noch um Faktor 16 schneller als der Row-Store. Das liegt daran, dass hier die Selektion nicht über den Primärschlüssel erfolgt, sondern die Spalten `height` und `weight` gescannt werden müssen, was beim Column-Store deutlich schneller erfolgen kann.
Bemerkenswert ist die nur geringe Beschleunigung um Faktor 1,4 bei der Summen-

funktion `SUM()` bei Abfrage E. Bei Einsatz eines Column-Store ist die Funktion erstaunlicherweise nur halb so schnell wie `MIN()`, `MAX()` oder `AVG()`, im Row-Store jedoch ist die Summen-Funktion etwa um eine Größenordnung schneller als die anderen Aggregatsfunktionen. Diese Besonderheit sollte noch genauer untersucht werden, da SUM() eine wichtige SQL-Funktion ist die in realen Geschäftsanwendungen oft verwendet wird.

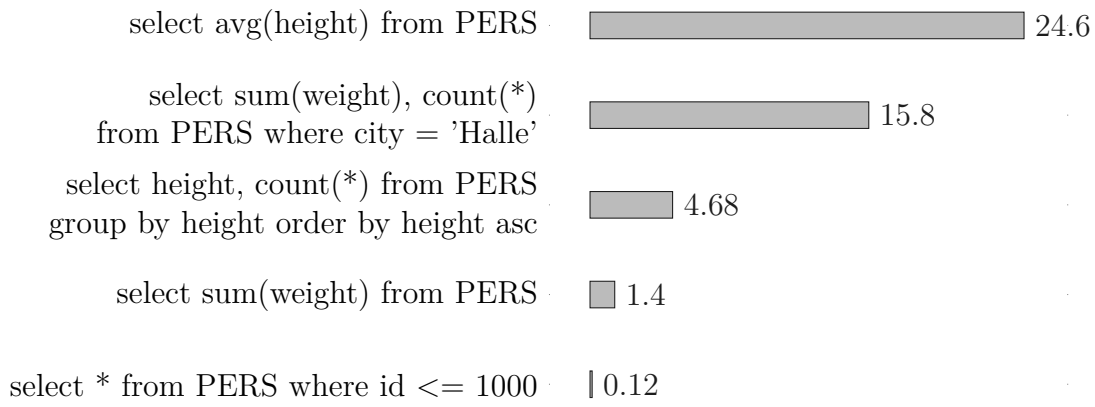

Abbildung 5: Performance Column-Store / Row-Store[23]

Abbildung 5 zeigt exemplarisch die relative Performance von Column-Store und Row-Store für einige SQL-Abfragen. Beispielsweise ist `select avg(height) from PERS` mit dem Column-Store 24,6 mal schneller als mit dem Row-Store.
Das Anlegen der Indizes auf der CS-Tabelle erfolgt sehr schnell (ca. 150ms), allerdings profitiert der Column-Store davon kaum, die Ausführungszeiten verbessern sich nicht signifikant. (s. Tabelle 1)

Req#	SQL	Colum-Store ohne/mit Index	Row-Store ohne/mit Index
N,S:	`select * from PERS where height=210 and weight=110`	3,75/ 2,18 ms	61,78/ 0,61 ms
A,T:	`select MIN(height) from PERS`	11,56/10,62 ms	185,81/185,56 ms

Tabelle 1: Einfluss von Indizes auf die Performance bei Row-Store und Column-Store

Bei der Abfrage T mit der Funktion MIN() findet der Index offenbar keine Verwendung im Ausführungsplan, die Ausführungzeiten haben sich durch den Index im Column-Store und im Row-Store nicht verändert.
Bei der RS-Tabelle dauert die Erstellung der Indizes deutlich länger (>6s), jedoch kann der Row-Store teilweise deutlich von den Indizes profitieren. Bei Abfrage S,

[23]Quelle: eigene Darstellung

welche 323 vollständige Tupel selektiert, kann die Ergebnismenge 100 mal schneller bestimmt und geliefert werden: in 0,61ms statt 61,8ms. Bei einem realen Anwendungsfall kommen solche Indizes normalerweise auch zur Anwendung, wenn über die entsprechenden Spalten häufig selektiert wird. Deswegen müssen für einen realistischen Geschwindigkeitsvergleich auch Indizes berücksichtigt werden.
Hier wird also deutlich, dass auch der Row-Store seine Vorteile hat, und zwar immer dann, wenn die Ergebnismenge viele Spalten oder sogar die kompletten Tupel enthält. Durch die Lokalität der Daten eines Tupels kann das ganze Tupel sehr schnell aus dem Speicher gelesen werden. Der Column-Store fällt hier zurück, weil die Attribute eines Tupels an relativ weit entfernten Speicherstellen abgelegt sind und erst wieder zu einem Datensatz zusammengefügt werden müssen (Tuple Reconstruction). Je mehr Zeilen die Ergebnismenge hat, umso deutlicher ist der Row-Store im Vorteil. Dieser kleine Geschwindigkeitsnachteil des Column-Store im OLTP-Bereich hat in der Praxis aber kaum Auswirkungen, da hier immer auch noch andere Latenzen in den nachgelagerten Systemen auftreten und die transaktionalen Abfragen insgesamt ja sehr kurz sind. Bei einem Mischbetrieb von OLAP und OLTP sollte also vorzugsweise mit einem Column-Store gearbeitet werden, denn die Geschwindigkeitsvorteile bei OLAP-Abfragen überwiegen deutlich.

4 Fazit und Ausblick

In der Arbeit wurde ein kurzer Überblick über die In-Memory-Datenbank SAP HANA gegeben. Es wurde gezeigt, wie und mit welcher Motivation diese neue Technologie bei der Firma SAP und dem Hasso-Plattner-Institut entwickelt wurde. Die beiden Schlüssel-Techniken Spaltenorientierung und Wörterbuch-Kodierung wurden kurz skizziert. Anhand von synthetischen Benchmarks wurden Row-Store und Column-Store gegenübergestellt und die jeweiligen Vor- und Nachteile herausgearbeitet. Dabei hat sich die Überlegenheit der spaltenorientierten Speicherung für analytische Datenbank-Abfragen klar gezeigt.
Die hier ermittelten Ergebnisse sind jedoch nur ein erster Anhaltspunkt. Für weitergehende systematische Untersuchungen von Row- und Column-Store sollten auch die Leistung von Schreibzugriffen gemessen werden sowie die Leistung bei Mischbetrieb von Lese- und Schreibzugriffen, wie er ja auch in der Praxis vorkommt.
Die in dieser Arbeit verwendete Datenmenge war mit 5 Mio. Datansätzen und 481MB Rohdaten recht klein. Die gemessenen Zeiten waren daher auch nur sehr kurz. Um aussagekräftigere Messwerte zu bekommen sollte die Datenmenge noch deutlich grösser sein. Interessant wäre auch ein Vergleich von SAP HANA mit anderen RDBM auf der gleichen Hardware, wie Oracle DB, MS SQL-Server, MaxDB, MySQL, Maria-DB oder PostgreSQL welche teilweise auch mit einem Column-Store

betrieben werden können, zum Teil werben deren Anbieter auch mit In-Memory-Technik. Idealerweise sollte Datenbank-Leistung mit einem standardisierten Test, wie z.B. TPC gemessen werden, um vergleichbar zu sein. Der Aufwand dafür sollte aber nicht unterschätzt werden.

Literatur

[Plattner, H., Leukert, B. (2015)] *The In-Memory Revolution - How SAP HANA enables business of the future*, Springer, 2015

[Prassol, P. (2015)] Prassol2015 *SAP HANA als Anwendungsplattform für Real-Time Business*, Springer Fachmedien Wiesbaden, 2015

Internetquellen:

[Intel (2014)] *Scaling Data Capacity for SAP HANA with Fujitsu Servers* URL: `http://www.intel.de/content/www/de/de/big-data/big-data-xeon-e7-sap-hana-fujitsu-paper.html`, Abruf am 28.8.2016

[Intel (2016)] *Intel Xeon Processor E7-8890 v4* URL: `http://ark.intel.com/de/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz`, Abruf am 28.8.2016

[Plattner/HPI (2015)] *In-Memory Data Management 2015, Prof. Hasso Plattner* URL: `https://open.hpi.de/courses/imdb2015`, Abruf am 20.8.2016

[SAP (2014)] *SAP HANA SPS 09 - What's New? SAP HANA Scalability* URL: `https://hcp.sap.com/content/dam/website/saphana/en_us/Technology%20Documents/SPS09/SAP%20HANA%20SPS09%20-%20HANA%20Scalability.pdf`, Abruf am 30.8.2016

[SAP (2016)] *SAP HANA Tailored Data Center Integration - Overview* URL: `http://go.sap.com/documents/2016/05/827c26ba-717c-0010-82c7-eda71af511fa.html`, Abruf am 27.8.2016

[Statista (2014)] *Anzahl der Gemeinden in Deutschland nach Gemeindegrößenklassen* URL: `http://de.statista.com/statistik/daten/studie/1254/umfrage/anzahl-der-gemeinden-in-deutschland-nach-gemeindegroessenklassen/`, Abruf am 27.8.2016

Anhang 1: Benchmark-Daten

#	SQL	Result	Column-Store (server processing time in ms)					Row-Store (server processing time in ms)					Faktor
			t1	t2	t3	t4	Øt	t1	t2	t3	t4	Øt	
	select count(*) from PERS	5.000.000											
A	select min(height) from PERS	120	11,971	11,458	11,436	11,362	**11,557**	178,009	195,815	188,632	180,780	**185,809**	**16,08**
B	select max(height) from PERS	220	13,936	11,717	13,461	11,977	**12,773**	173,824	173,631	165,133	164,467	**169,264**	**13,25**
C	select avg(height) from PERS	174,45153	14,635	12,475	13,620	11,898	**13,157**	319,168	319,107	329,284	326,565	**323,531**	**24,59**
D	select avg(weight) from PERS	74,47353	13,131	13,005	11,052	11,685	**12,218**	307,321	313,798	321,857	318,898	**315,469**	**25,82**
E	select sum(weight) from PERS	372.367.652	25,331	26,046	27,026	24,547	**25,738**	37,734	36,873	35,294	34,587	**36,122**	**1,40**
F	select min(city) from PERS		75,720	82,810	82,377	73,704	**78,653**	499,917	499,335	506,296	508,155	**503,426**	**6,40**
G	select sum(weight), count(*) from PERS where city = 'Halle'	442.044 , 5.945	4,634	4,549	3,900	2,804	**3,972**	63,523	62,784	62,270	62,774	**62,838**	**15,82**
H	select avg(weight), count(*) from PERS where height BETWEEN 170 AND 181	74,981791 1.552.376	13,075	12,803	11,903	14,564	**13,086**	633,390	601,736	605,184	631,530	**617,960**	**47,22**
I	select avg(weight), count(*) from PERS where height = 180	79,5310053, 124.223	7,649	6,878	10,631	6,546	**7,926**	295,027	294,437	295,167	291,406	**294,009**	**37,09**
J	select avg(weight), count(*) from PERS where height = 210	109,64291 8.082	4,643	3,610	5,071	4,387	**4,428**	324,306	345,635	316,385	308,940	**323,817**	**73,13**
K	select weight, count(*) from PERS group by weight order by weight asc	weight_dist ribution.csv	14,354	15,184	14,679	14,858	**14,769**	65,579	64,785	65,547	67,343	**65,814**	**4,46**
L	select height, count(*) from PERS group by height order by height asc	height_distri bution.csv	13,203	15,709	14,250	13,732	**14,224**	67,414	68,156	65,098	65,569	**66,559**	**4,68**
M	select count(*) from PERS where height = 210 and weight=110	323	3,987	3,246	2,715	3,970	**3,480**	600,169	581,816	596,646	589,765	**592,099**	**170,17**
N	select * from PERS where height=210 and weight=110	resultset	3,498	3,947	3,679	3,859	**3,746**	61,491	63,754	59,003	62,885	**61,783**	**16,49**
	Transaktionale Abfragen:												
O	select * from PERS where id <= 1000	resultset	4,232	3,434	3,260	3,291	**3,554**	0,371	0,416	0,456	0,408	**0,413**	**0,12**
P	select * from PERS where id > 4999000	resultset	2,159	2,217	2,255	2,606	**2,309**	0,635	1,587	3,240	0,641	**1,526**	**0,66**
	Anlegen von Indizes:												
Q	create index "IX_PERS_HEIGHT" on "PERS" ("HEIGHT" ASC)		151,874				**151,874**	6353,000				**6353,000**	**41,83**
R	create index "IX_PERS_WEIGHT" on "PERS" ("WEIGHT" ASC)		146,646				**146,646**	6805,000				**6805,000**	**46,40**
	Abfragen mit angelegten Indizes:												
S	select * from PERS where height=210 and weight=110	resultset	1,936	2,199	2,368	2,230	**2,183**	0,532	0,628	0,638	0,644	**0,611**	**0,28**
T	select min(height) from PERS		9,501	11,475	10,904	10,609	**10,622**	181,281	178,986	204,408	177,598	**185,568**	**17,47**
U	select * from PERS where id <= 1000		3,123	3,171	4,627	3,276	**3,549**	0,170	0,327	0,334	0,362	**0,298**	**0,08**
V	select * from PERS where id > 4999000		2,066	2,292	2,999	2,426	**2,446**	0,246	0,307	0,293	0,511	**0,339**	**0,14**

Column-Store vs Row-Store

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

München, 31. August 2016

Oliver Kurmis