Einfluss von Materialisierungsstrategien auf GPU beschleunigte spaltenorientierte Datenbanken

René Hoyer
Otto-von-Guericke Universität Magdeburg
rene.hoyer@st.ovgu.de

Patrick Sulkowski
Otto-von-Guericke Universität Magdeburg
patrick.sulkowski@st.ovgu.de

***Abstract*—Diese Arbeit befasst sich mit der vergleichenden Betrachtung von zeilenorientierten und spaltenorientierten Datenbanken für eine GPU-gestützte Anfrageverarbeitung. Dabei wird untersucht, wie sich unterschiedliche Materialisierungsstrategien auf die Verarbeitung durch GPU's auswirken. Dazu wurde exemplarisch das Datenbankmanagementsystem CoGaDB um einen GPU Join und einer frühen Materialisierung erweitert.**

I. Einleitung

Bereits in den 80er Jahren wichen Forscher von zeilenorientierten Datenbanken ab und untersuchten eine spaltenweise Speicherung und Verarbeitung in Datenbanken [10]. Dieser Ansatz wird nach dem Millennium wieder aktueller und entwickelt sich zur performanten Alternative eines zeilenorientierten Datenbanksystems [1, 2, 3]. Spaltenorientierte Datenbanken besitzen gegenüber zeilenorientierten Datenbanken den Vorteil, dass sie (1) für Hauptspeicherdatenbanken auf Grund ihrer besseren Ausnutzung des *Central Processing Unit* (CPU) Caches besonders gut geeignet sind und (2) eine höhere Kompressionsrate halten, was dazu führt, dass mehr Daten im Hauptspeicher gehalten werden können. [2, 9]

Für die Datenverarbeitung im Hauptspeicher stellt die Speicherbandbreite den Flaschenhals dar [22]. Um die Performance der Anfrageverarbeitung weiter zu erhöhen, befassen sich neuere Ansätze mit einer Koprozessor gestützten Verarbeitung. Koprozessoren bieten den Vorteil, dass sie zum einen eine höhere Speicherbandbreite als CPU's aufweisen und eine parallele Verarbeitungsverteilung auf CPU und Koprozessor ermöglichen [4, 8, 18, 21]. Der prominenteste Vertreter unter den Koprozessoren ist die *Graphics Processing Unit* (GPU). GPUs können die Performance von Hauptspeicherdatenbanken signifikant beschleunigen [16].

Diese Arbeit evaluiert, wie sich eine spaltenorientierte gegenüber einer zeilenorientierten Datenbank verhält, wenn eine Verarbeitung auf einem Koprozessor stattfinden soll. Es gilt in dieser Arbeit die Frage "*Was ist für eine GPU beschleunigte Verarbeitung besser geeignet: RowStore vs. ColumnStore?*" zu beantworten.

Das Papier gliedert sich weiter in Grundlagen zu Spaltenorientierten Datenbanken, Materialisierungsstrategien und GPUs. Darauf aufbauend werden Materialisierungsstrategien und GPU Operatoren in dem verwendeten Prototypen CoGaDB diskutiert. Abschnitt fünf befasst sich mit dem Evaluierungskonzept und den Ergebnissen. Abschließend werden verwandte Arbeiten aufgeführt und eine Zusammenfassung sowie Ausblick gegeben.

Name	Vorname	Matrikel	Status
Hoyer	René	12345	Student
Sulkowski	Patrick	67890	Student
Breß	Sebastian	02468	Doktorand

a)

Name	Vorname	Matrikel	Status
Hoyer	René	12345	Student
Sulkowski	Patrick	67890	Student
Breß	Sebastian	02468	Doktorand

b)

Fig. 1. Zeilenorientierte (a) und Spaltenorientierte Darstellung (b)

II. Grundlagen

In diesem Abschnitt werden Grundlagen zu spaltenorientierten Datenbanken, dem Konzept der Materialisierung sowie der allgemeinen GPU-Architektur eingeführt. Den Abschluss bildet eine Einführung in das verwendete Datenbankmanagementsystem CoGaDB.

A. Spaltenorientierte Datenbanken

In *zeilenorientierten Datenbanken* (ZD) umfasst ein Eintrag auf einer Seite genau eine Zeile einer Datenbank-Tabelle. Der Eintrag ist dabei Byte-weise abgelegt, sodass die Daten im Speicher exakt den Daten auf der Festplatte entsprechen. In *spaltenorientierten Datenbanken* (SD) werden hingegen auf einer Seite nicht der Datensatz einer Zeile einer Tabelle hinterlegt, sondern eine gesamte Spalte dieser Tabelle. Daraus ergibt sich, dass (1) Aggregatsfunktionen direkt auf Spalten arbeiten können, was es wiederum ermöglicht, dass nur benötigte Daten geladen werden müssen und den I/O Overhead verringert. (2) Es kann eine spaltenweise Komprimierung erfolgen, welches eine erhöhte Kompressionsrate mit sich bringt, da die Daten einer Spalte sehr ähnlich sein können. Darüber hinaus kann durch (3) eine späte Materialisierung die Performance gegenüber einer ZD um das dreifache erhöht werden [2]. Abbildung 1 verdeutlicht den Unterschied an einem Beispiel.

Für einen tieferen Überblick über die Unterschiede von ZD und SD wird auf die Arbeit von Abadi, Madden und Hachem verwiesen [2].

B. Materialisierungsstrategien

Damit das Ergebnis einer Anfrage transparent von einer Anwendung verarbeitet werden kann (unabhängig von der Implementierung einer Datenbank), ist es für SD notwendig, eine spaltenorientierte Darstellung in eine zeilenorientierte Tabelle zu überführen. Dieser Vorgang wird als *Materialisierung* bezeichnet [3].

Der Zeitpunkt der Materialisierung ist für eine performante Anfrageverarbeitung entscheidend. Unterschieden wird dabei zwischen früher und später Materialisierung. Bei einer *frühen Materialisierung* wird beim ersten Zugriff auf einen Wert einer Spalte die entsprechende Zeile einer Tabelle materialisiert [3].

Die späte Materialisierung hingegen verzögert den Vorgang und wendet zunächst Datenbankoperationen (z.B. Selektion) auf einzelne Spalten an und hält als Ergebnis die entsprechenden TupelId's vor. Erst vor der Auslieferung des Anfrageresultats an die jeweilige Anwendung wird der Vorgang der Materialisierung auf die Zeilen einer Tabelle entsprechend der TuppelId's angewendet. Dadurch wird die Menge der zu materialisierenden Zeilen durch die eigentliche Datenbankanfrage eingeschränkt und verringert die Verarbeitungszeit bei SD.

Abadi u. a. konnten keinen eindeutigen Zeitpunkt feststellen, welcher allgemeingültig den optimalen Zeitpunkt für eine Materialisierung darstellt. Sie fanden heraus, dass eine gute Heuristik für eine späte Materialisierung darin besteht, ob die Ausgabedaten aggregiert sind, ob die Anfrage eine geringe Selektivität aufweist oder ob die Eingabedaten komprimiert vorliegen [3].

C. GPU

Die GPU kann der CPU zur Beschleunigung bestimmter Berechnungen dienen, was als *GPGPU* (General Purpose Computation on Graphics Processing Unit) bezeichnet wird. Über Programmierschnittstellen wie CUDA[1] oder OpenCL[2] können die ursprünglich für Grafikberechnungen gedachten Shadereinheiten für andere Berechnungen ausgenutzt werden. Bei der GPU-Architektur handelt es sich um eine skalierbare parallele SIMD-Architektur (Single Instruction, Multiple Data), welche vor allem für parallele Berechnungen geeignet ist - besonders auch für Datenbank-Anfrageverarbeitung [20][16][11][13]. Speziell Join-Algorithmen profitieren von der parallelen Verarbeitung [15][20].

Die parallele Verarbeitung auf der GPU wird von einem Host-Programm auf der CPU gesteuert. Dieses ruft Kernel-Programme für jeden einzelnen Threads der GPU auf. Diese Threads sind in Thread-Gruppen (Blöcke) organisiert, welche wiederum ein Block-Gitter bilden. Über bereitgestellte Indizes können einzelne Threads identifiziert werden, was vor allem den Zugriff auf Array-basierte Daten erleichtert. Wenn mit der GPU als Koprozessor gearbeitet wird, stehen zudem mehrere Adressräume zur Verfügung:

- Host-Speicher
- globaler Speicher
- lokaler Speicher
- privater Speicher
- Konstanten-Speicher
- und Textur-Speicher

Daten liegen meist im Host-Speicher vor, welcher zuerst auf den globalen Speicher der GPU übertragen werden muss, da dieser von der GPU nicht adressierbar ist. Ergebnisse müssen anschließend wieder vom GPU-Speicher in den Host-Speicher Als Schnittstelle dient hierbei PCI-E, welches aufgrund einer relativ niedrigen Bandbreite einen Flaschenhals darstellen kann [14]. Dadurch sollte dieser Datenaustausch auf ein Minimum reduziert werden. Der globale Speicher ist von den einzelnen GPU-Prozessoren direkt adressierbar. Koaleszierter Zugriff auf diesen Speicher kann die Verarbeitungszeit erheblich verkürzen [15]. Aus dem globalen Speicher können die Daten in den lokalen Speicher transferiert werden. Dieser dient als gemeinsamer Cache innerhalb eines Threadblocks; ist jedoch relativ klein. Jeder einzelne Thread erhält zudem einen privaten Speicher, welcher jedoch sehr klein ist. Der Konstanten-Speicher dient als schnelle Alternative zum globalen Speicher für Daten, welche sich zur Kernel-Laufzeit nicht ändern. Texturspeicher ist globaler Speicher, auf welchen über einen Cache, der für Texturen optimiert ist, gelesen wird.

He et al. haben mit Hilfe von Parallelprimitiven wie sort, scatter, reduce, etc. gezeigt, dass bestimmte Datenbankanfragen auf der GPU schneller verarbeitet werden können als auf der CPU [16]. Zudem wurde gezeigt, dass sich sowohl Nested-Loop-Joins, Hash-Joins und Sort-Merge-Joins auf der GPU sehr gut beschleunigen lassen [15][16].

D. CoGaDB

Diese Ausarbeitung setzt auf CoGaDB[3], einem spaltenorientiertem, GPU-beschleunigten Datenbankmanagementsystem auf. Das Ziel von CoGaDB ist es, komplexe OLAP Workloads mithilfe von (Ko-)Prozessoren zu beschleunigen, wobei die Datenbank selbst komplett im Hauptspeicher der CPU gehalten wird.

Der Grundgedanke der spaltenorientierten Implementierung in CoGaDB besteht darin, dass jeder Operator eine Positionsliste zurück gibt, welche das Ergebnis eindeutig bestimmt. Die Positionsliste kann zwischen CPU und GPU hin und her transferiert werden, was einen billigen Wechsel zwischen (Ko-)Prozessoren während der Anfrageverarbeitung ermöglicht. Eine Spalte beschränkt sich im aktuellen Implementierungsstand auf die Datentypen Integer, Float und Varchar.

[1] http://www.nvidia.com/object/cuda_home_new.html/
[2] http://www.khronos.org/opencl/

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/

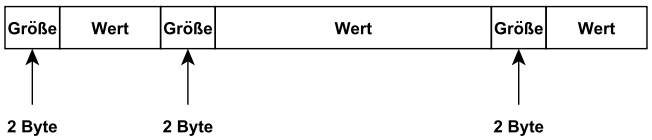

Fig. 2. Aufbau einer Seite vom Spaltentyp varchar

III. Materialisierungsstrategien in CoGaDB

CoGaDB unterstützt bereits eine späte Materialisierung unterstützt. Es wird davon ausgegangen, dass sich eine späte Materialisierung gegenüber einer frühen Materialisierung für eine GPU-beschleunigte Verarbeitung performanter verhält [7]. In dieser Arbeit soll die Frage "*Was ist für eine GPU beschleunigte Verarbeitung besser geeignet: RowStore vs. ColumnStore?*" experimentell bewiesen werden. Dazu wurde CoGaDB um eine frühe Materialisierung, als Repräsentant einer ZD, sowie um einen GPU Join[1] erweitert. Nachfolgend sind beide Ansätze kurz beschrieben.

A. Frühe Materialisierung

In diesem Abschnitt wird die Implementierung der frühen Materialisierung für CoGaDB diskutiert.

Die benötigten Spalten einer Tabelle werden binär gespeichert. Jede Binärdatei hält im Header ein führendes Status-Byte. Dieses Byte setzt sich aus einem Compressed-Bit (C) und zwei Bits für den Typ (T) zusammen. Ein gesetztes C-Bit signalisiert, dass die Daten dieser Datei komprimiert vorliegen und eine Dekomprimierung erfolgen muss. Die T-Bits ergeben zusammen den Datentyp dieser Spalte. Somit ergeben sich die relevanten Informationen für eine Spalte direkt aus ihrem Header.

Nachdem von der Pufferverwaltung das Status-Byte ausgelesen und verarbeitet wurde, werden die Daten sequentiell von der Platte auf Seiten der Pufferverwaltung geschrieben. Da für eine Seite 16 (bzw. 8) Bit für die Länge des Inhalts reserviert wurde, ergibt sich eine maximale Größe von 64 Kb (bzw. 256 Byte) für jede Seite. Für den Inhalt einer Seite wird angenommen, dass ein Wert komplett auf eine Seite passt. Daraus ergibt sich, dass (1) nur Nicht-Spannsätze verwendet werden und (2) eine maximale Länge für varchar existiert, welche sich an der maximalen Größe einer Seite richtet. Auf Grund der spaltenorientierten Verarbeitung können nur Werte einer Spalte und somit Werte eines Datentyps auf einer Seite existieren. Daraus ergibt sich, dass Integer-, Float- und Boolean-Werte ohne Zusatzinformationen hintereinander weg auf die Seite geschrieben werden können. Dies resultiert unmittelbar aus der feststehenden Anzahl an Bytes für den jeweiligen Datentyp (z.B. vier Byte für Int und Float-Spalten und ein Byte für Boolean-Spalten [19]). Für varchar ist es zusätzlich notwendig, die Anzahl an Bytes niederzuschreiben, damit beim Auslesen die Position erkannt werden kann, an der ein varchar beginnt und endet. Abbildung 2 verdeutlicht den Aufbau für eine varchar-Spalte. Aus der Abbildung wird ersichtlich, dass der Aufbau einer Seite einer varchar-Spalte durch ein Tupel von Größe und Wert gekennzeichnet ist.

[1]Dieser ist der einzige Operator, welcher noch nicht in CoGaDB vorhanden war

Fig. 3. Aufbau einer materialisierten Tabelle mit drei Spalten

Liegen die Daten in der Pufferverwaltung vor, werden bei der Materialisierung die Werte der einzelnen Spalten zusammengeführt, sodass eine zeilenorientierte Darstellung vorliegt. Dabei wird der i-te Wert einer Seite in der Pufferverwaltung der i-ten Zeile in der materialisierten Tabelle zugeordnet. Das Resultat der Materialisierung ist eine Tabelle, welche aus RowPages besteht. Auf den RowPages sind die materialisierten Zeilen als Bytes hinterlegt, um eine Konvertierung in den entsprechenden Datentypen an dieser Stelle zu vermeiden und somit die Performance der Materialisierung zu erhöhen. Für eine Byte-weise Darstellung ist es notwendig, dass zunächst die Anzahl an Spalten festgehalten wird (um die Daten später auf die Festplatte niederzuschreiben und auf gleichem Wege wieder auslesen zu können) und anschließend jeden Attributwert der Zeile ein Tupel von "Anzahl an Bytes und dem eigentlichen Wert" zugewiesen wird. Abbildung 3 zeigt die i-te Zeile einer materialisierten Tabelle mit drei Spalten. Aus dieser Darstellung heraus kann auf den Wert der j-ten Spalte zugegriffen werden, indem der j-te Wert der Zeile i ausgelesen wird. Um einen indexbasierten Zugriff zu gewährleisten, kennt jede Spalte einer Tabelle ihre Position im Schema (Wert j) und benötigt somit nur die Zeilennummer (Tupel-Id; Wert i) für den Zugriff übermittelt bekommen. Die Materialisierung ist somit abgeschlossen und die erzeugte Tabelle steht für einen indexbasierten Zugriff über die Tupel-Id zur Verfügung.

In CoGaDB sind alle Operatoren für Spalten entwickelt. Dadurch erfolgt der oben beschrieben Zugriff durch die RowValueColumn. Diese Spalten kennen neben ihrer Position im Schema auch den internen Aufbau der Tabelle, wodurch ein transparenter Zugriff auf die materialisierte Tabelle erfolgt. Abbildung 4 zeigt schematisch den Zugriff auf das i-te Attribut einer Spalte. Eine RowValueColumn greift auf ihren i-ten Wert zu, indem zuerst ermittelt wird, auf welcher RowPage sich die entsprechende Zeile i befindet und dann der Wert der RowValueColumn aus der Zeile gelesen wird.

IV. GPU Join

In diesem Abschnitt wird die Implementierung eines für Primär- und Fremdschlüssel optimierten und durch die GPU beschleunigten Join-Algorithmus in CUDA vorgestellt. Es handelt sich dabei um eine angepasste Version des parallelen Sort-Merge-Join Algorithmus, welcher

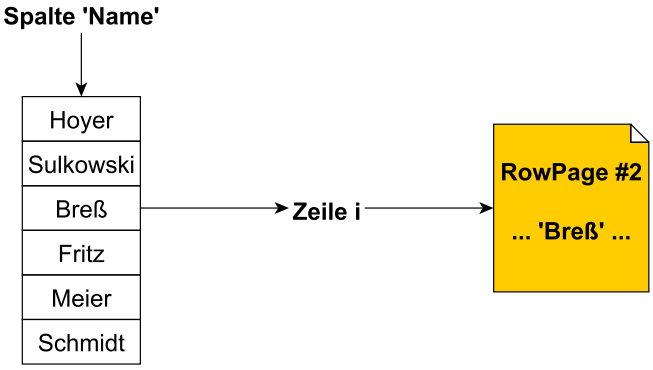

Fig. 4. Zugriff auf den Wert der i-ten Zeile einer Spalte (hier 'Breß'). In diesem Beispiel wird davon ausgegangen, dass zwei Zeilen auf eine RowPage Platz finden.

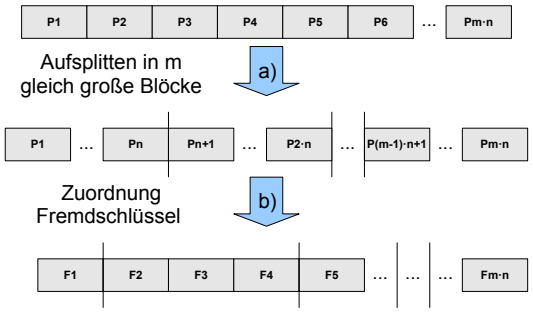

Fig. 5. Aufsplittung der Primärschlüsselspalte in m Blöcke (a) und Zuordnung von m Fremdschlüsselblöcken über binäre Suche (b)

von He et al. vorgestellt wurde. Ähnlich einem sequenziellem Sort-Merge-Join werden die zu joinenden Spalten zuerst sortiert und anschließend zusammengeführt. Hierdurch tut sich ein Vorteil des Sort-Merge-Joins auf: Liegen die Spalten durch frühere Query-Prozesse bereits sortiert vor, fällt die Sortierung weg. Im Gegensatz zu Hash-Joins skalieren Sort-Merge-Joins außerdem sehr gut bei steigendem Parallelisierungsgrad [20]. Für die parallele Ausführung auf der GPU werden zusätzliche Schritte wie die Datenübertragung vom und zum GPU-Speicher und der Erzeugung von Tupel-IDs benötigt. Der aus diesen Überlegungen entstandene Algorithmus läuft wie folgt ab:

1) Spalten vom Host- in GPU-Speicher transferieren
2) Erzeugung einer Indexsequenz (Tupel-IDs)
3) Parallele Sortierung
4) Aufsplitten der Primärschlüsselspalte in gleich große Blöcke
5) Paralleles Merging
6) Tupel-IDs- vom GPU- in den Host-Speicher transferieren

Sind die zu joinenden Spalten im ersten Schritt vom Host-Speicher in den GPU-Speicher transferiert wurden, wird im folgendem Schritt je Spalte eine Index-Sequenz über die enthaltenden Elemente gelegt. Im Anschluss werden die beiden Spalten samt Indizes sortiert. Hierbei wird auf die in CUDA enthaltene Thrust-Bibliothek[1] zurückgegriffen, welche hochoptimierte Sortieralgorithmen bereitstellt. Die anschließende Merge-Phase wird parallelisiert, indem die sortierten Primärschlüsselspalten in gleich große Blöcke zerlegt und die korrespondierenden Blöcke der Fremdschlüsselspalte durch eine binäre Suche ermittelt wird. Dabei werden die Datensätze in Blöcke der Mindestgröße acht zerlegt. Bei weniger als 512 oder gleich 512 Datenblöcken wird nur ein Threadblock in CUDA verwendet, um Zugriff auf den schnellen lokalen Speicher eines solchen Threadblocks mit maximal 512 Threads zu gewährleisten. Mit der binären Suche wird das letzte Element eines Teilblocks in der Fremdschlüsselspalte ermittelt, dessen Index die obere Grenze des korrespondierenden Blocks darstellt. Die untere Grenze eines Blockes wird durch die obere Grenze des Vorgängerblocks gesetzt. In Abbildung 5 sind diese beiden Zuordnungsphasen noch einmal zusammengefasst. He et al. haben gezeigt, dass eine parallele binäre Suche signifikant schneller als eine parallele sequenzielle Suche ist [15]. Sind die Teilspalten nun den einzelnen Threads zugeordnet, werden diese dort nun unabhängig voneinander mit einem Merge-Algorithmus zusammengeführt. Da es sich bei den Spalten um eine Primär-Fremdschlüsselbeziehung handelt, entstehen maximal so viele Zuordnungen, wie es Fremgschlüsseleinträge gibt. Dadurch entsteht keinerlei Overhead um die Anzahl der Join-Ergebnisse in den einzelnen Threads zu ermitteln. Das Ergebnis - zugeordnete ID-Paare, dessen korrespondierende Elemente gleich sind - wird im Anschluss wieder in den Host-Speicher zurücktransferiert. Die nun den Tupeln der Ausgangstabellen für den Join zugeordeten IDs können dazu verwendet werden, um das Join-Ergebnis zu materialisieren. Dies bedeutet, dass mit Hilfe der IDs die korrespondierenden Tupel zu einer neuen Tabelle verschmolzen werden. Andererseits können die IDs auch statisch im Programm hinterlegt werden (als sogenannte Lookup-Tabelle), wodurch der Zugriff auf zusammengehörige Tupel der beiden Ausgangstabellen über die IDs gewährleistet wird. Dadurch können im Anschluss weitere Datenbankanfragen auf dem Join-Ergebnis erfolgen, ohne dass zusätzlicher Speicher für eine neue Tabelle durch die Materialisierung belegt wird.

V. Evaluierung

Um die Forschungsfrage "*Was ist für eine GPU beschleunigte Verarbeitung besser geeignet: RowStore vs. ColumnStore?*" beantworten zu können, wurden Experimente unter verschiedenen Gesichtspunkten durchgeführt. Nachfolgend erfolgt eine Präsentation und Auswertung der ermittelten Ergebnisse, sowie Hinweise auf die Aussagekraft der Resultate. Dazu wird einführend eine Übersicht über die durchgeführten Experimente gegeben.

Das Testsystem umfasst: Intel® Core™ i5-2500 CPU

[1] https://code.google.com/p/thrust/

@3.30 GHz mit vier Kernen und acht GB RAM sowie einer NVIDIA® GeForce® GT 640 GPU mit zwei GB Speicher. Als Betriebssystem dient Ubuntu in der 64bit-Version.

A. Übersicht der Experimente

Für eine Betrachtung des Mehraufwands einer frühen Materialisierung werden Experimente zur Materialisierungszeit durchgeführt. Diese betrachten den Einfluss von Tabellengröße sowie die Anzahl der Spalten auf die Materialisierungszeit. Dafür wird einer der beiden Parameter festgesetzt und der jeweils andere variiert.

Anschließend erfolgen Experimente zur Zugriffszeit auf ein Element einer Tabelle. Als unabhängige Variablen dienen die Tabellengröße, die Position des Elements (Beginn, Mitte, Ende) und der Datentyp, welcher abgefragt wird. Anhand der Zugriffszeit kann später analysiert werden, wie die Verhältnisse zwischen einem SD und einem ZD beim Bereitstellen der Daten für eine GPU-basierte Verarbeitung ausfallen.

Darauf aufbauend wurde der TPCH-Benchmark herangezogen und die gleichen Messungen auf der Tabelle Lineitem im Skalierungsfaktor 1 durchgeführt. Daraus lässt sich ein allgemeiner Vergleich zu anderen Implementierungen und DBMS ableiten.

Weiterhin findet ein Experiment statt, welches die Materialisierung einer einzelnen Spalte misst. Dies ist notwendig da nur eine materialisierte Spalte an die GPU transferiert werden kann.

Die ersten Tests des GPU-Joins umfassen Experimente, in denen der GPU-Join gegen den in CoGaDB integrierten Hash-Join antritt. Dieser Hash-Join ist Single-Threaded um die CPU nicht komplett auszulasten, was zu Ungunsten anderer Operationen wäre. Zudem ist die Speicherbandbreite ein nicht zu vernachlässigender Flaschenhals, wodurch mehr Prozessorkerne nicht unbedingt eine Leistungssteigerung bedeuten. Der GPU-Join dagegen kann die volle Leistung der Grafikkarte ausschöpfen. Als erstes wurde die Skalierbarkeit der beiden Join-Algorithmen bei steigender Spaltengröße verglichen. Dafür wurde jeweils die Zeit des gesamten Join-Vorgangs protokolliert. Die Messung erfolgte dabei sowohl für gleich große Primär- und Fremdschlüsselspalten, als bei unterschiedlichen Größenverhältnissen dieser. Zudem wurden der Join mit unterschiedlichen Spaltentypen (Integer und Float) durchgeführt. Weiterhin wurden die einzelnen Komponenten unseres Sort-Merge-Joins genauer aufgeschlüsselt. Das Zeitverhältnis der Datenübertragung von Host- in Grafikspeicher vor und nach dem Join zur reinen Berechnung des Joins auf der GPU wurde dafür ermittelt. Die Sortierungsphase des Sort-Merge-Joins wurde zusätzlich im Verhältnis zur Mergephase gemessen. Für die GPU-Join-Tests von N Primärschlüsseln mit M Fremdschlüsseln wurden als Werte für die Primärschlüssel jeweils Sequenzen von 0 bis N erzeugt, welche anschließend zufällig durchmischt wurden. Die Fremdschlüsseltabelle wurde aus M zufällig und gleichverteilt gewählten Werten der Primärschlüsseltabelle aufgebaut. Da die beiden Spalten vor dem Join aus der Tabelle extrahiert werden und nur Lookup-Tabellen als Join-Ergebnis erzeugt werden, um die Materialisierungszeit nicht mit einfließen zu lassen, ist es irrelevant, wie viele zusätzliche Spalten die Tabellen aufweist. Die Join-Experimente wurden so gewählt, dass die Daten vollständig auf den GPU-Speicher passen.

Tabellenzeilen	Zugriff ZD in µs	Zugriff SD in µs
10^1	2,048	0,256
10^2	2,048	0,256
10^3	2,048	0,256
10^4	2,048	0,256
10^5	2,048	0,256
10^6	2,048	0,256
10^7	2,048	0,256

Fig. 6. Vergleich der Zugriffszeiten auf die mittlere Zeile einer Tabelle durch eine varchar-Spalte (durchschnittliche varchar-Länge von 10) für ZD und SD in einer Tabelle mit drei Spalten

B. Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Untersuchung zusammengefasst.

Zuerst konnte festgestellt werden, dass die Materialisierung einen linearen Zeitaufwand in Abhängigkeit der Tabellengröße aufweist. Je mehr Daten zusammengefügt werden müssen, um so mehr Zeit nimmt der Vorgang mit einem konstantem Zeitanstieg in Anspruch. Aus Abbildung 7 wird ersichtlich, dass (1) für die Gesamtzeit einer Materialisierung maßgeblich die varchar-Spalte und deren Inhaltsgröße relevant ist. Je mehr Spalten dieses Typs bzw. je länger deren einzelne Werte sind, um so länger benötigt das Materialisieren. Es konnte gemessen werden, dass (2) varchar-Spalten den Großteil der Materialisierungszeit ausmachen. Das geht unmittelbar aus Messungen mit einem Tabellenschema von drei Spalten hervor. Dabei war jede Spalte von einem unterschiedlichen Datentyp und die durchschnittliche varchar-Länge betrug 10 Zeichen. Aus dieser Messung resultierte, dass die varchar-Spalte ca. 60% der Gesamtmaterialiserungszeit ausmacht. Weiterhin ist zu erkennen, dass (3) sowohl Float, als auch Int-Spalten die gleiche Zeit zur Materialisierung benötigen, da beide intern vier Byte zur Darstellung benötigen.

Für die Evaluierung der Zugriffszeit auf ZD (materialisierte Tabelle) und SD konnte ermittelt werden, dass die Zugriffszeit bei beiden konstant ist. Tabelle 6 zeigt eine Gegenüberstellung der Messergebnisse für den Zugriff auf das mittlere Element einer varchar-Spalte. Die durchschnittliche varchar-Länge betrug 10 und das Tabellenschema umfasste drei Spalten unterschiedlichen Datentyps (Int, Float, Varchar). Abbildung 13 zeigt die Werte im graphischen Vergleich und verdeutlicht die Unterschiede in der Zugriffszeit. Es wird ersichtlich, dass SD um den Faktor 8 schneller im Zugriff ist, als eine ZD. Experimente mit einem Zugriff auf unterschiedlichen Spaltentypen in einer ZD haben bestätigt, dass die Zugriffszeit unabhängig des

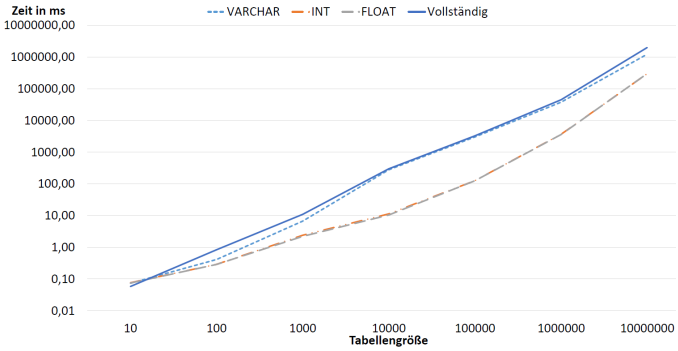

Fig. 7. Zeitmessung für die Materialisierung für unterschiedliche Tabellengrößen. Die durchschnittliche varchar-Länge beträgt 10 Zeichen.

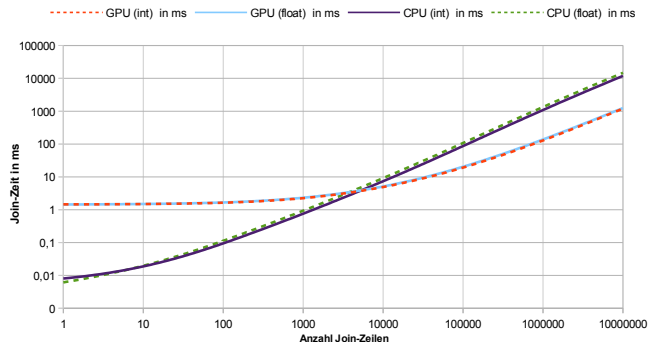

Fig. 8. Zeitmessung des GPU- und GPU-Joins für Float- und Integer-Spalten bei unterschiedlichen Zeilenanzahlen

Spaltentyps ist. Aus diesem Grund findet diese Betrachtung in den Diagrammen keine weitere Bedeutung.

Das TPCH-Benchmark Experiment der Tabelle Lineitem hat folgendes ergeben: der Zugriff auf das mittlere Element der Spalte COMMENT umfasst ebenfalls konstante 2,048 µs [1].

Die Experimente zur Materialisierung einer einzelnen Spalte (Vorbereitung für den Transfer zur GPU) erfolgt in lineare Zeit in Abhängigkeit der Spaltengröße. Beinhaltet eine Spalte zum Beispiel 10 Werte und der Zugriff auf ein Wert beträgt konstante 2,048 µs, dann ist die Zeit, welche zur Materialisierung dieser Spalte aufgebracht werden muss 10 * 2,048 µs = 20,48 µs. Das bedeutet, dass für eine GPU-gestützte Verarbeitung die Spaltengröße entscheidend für den Beginn der GPU-Berechnung ist.

In Abbildung 8 ist die absolute Zeit eines Join-Vorgangs abgebildet - je einmal für die Datentypen Integer und Float für den GPU- als auch den CPU-Join. Im Gegensatz zum CPU-Join steigt die Dauer des Joins beim GPU-Join wenig an, was mit der Latenz des Datentransfers von Host- zu Grafikspeicher und zurück erklärbar ist. Dadurch ist der GPU-Join bei wenig Einträgen (unter 1000) klar unterlegen. Im Bereich von 1000 bis 10000 Elementen nähern sich die

[1]COMMENT ist eine varchar-Spalte, welche große Daten beinhaltet und für Performance-Messungen gut geeignet ist

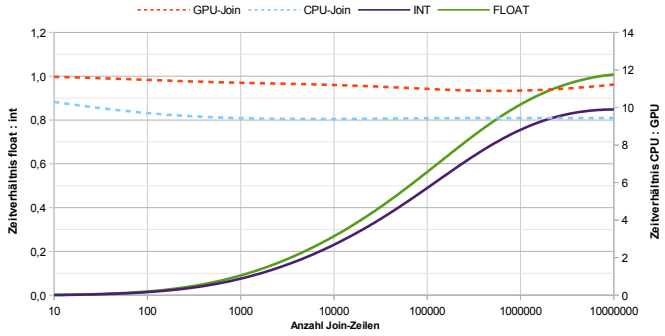

Fig. 9. Zeitverhältnis von Joins auf Integer- sowie Float-Spalten jeweils für GPU- und CPU-Join; Zeitverhältnis von CPU-Join zu GPU-Join bei Integer- sowie Float-Spalten

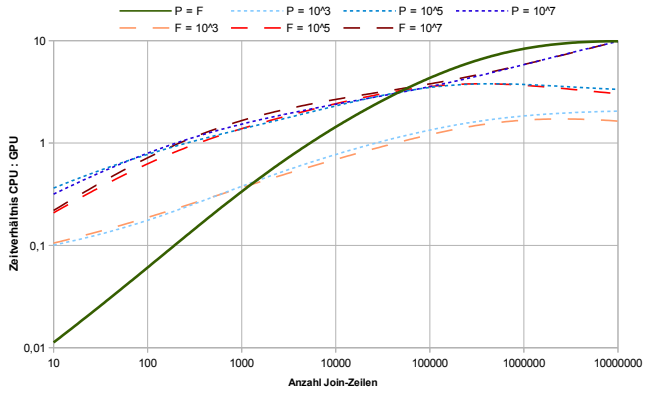

Fig. 10. Zeitverhältnisse des CPU- zum GPU-Join bei fester Anzahl an Primärschlüsseleinträgen (P) bzw. fester Anzahl an Fremdschlüsseleinträgen (F) bei gleichzeitig variabler Größe der jeweils anderen Join-Spalte; zum Vergleich: gleiche Größe beider Spalten (P = F)

Kurven an, bis der GPU-Join bei mehr als 10000 Einträgen klar die Oberhand gewinnt. Durch die begrenzte Anzahl an Prozessoren der Grafikkarte verlaufen die Kurven bei zunehmender Anzahl an Einträgen annähernd parallel. Diese Erkenntnis ist in Abbildung 9 noch deutlicher zu sehen. Dort ist das Verhältnis des CPU- zum GPU-Join abgebildet. Ist dieses anfangs noch unter 1 - d.h. der CPU-Join arbeitet schneller - nimmt dieses Verhältnis bei steigender Anzahl an Einträgen zu, bis es sich aufgrund fehlender weiterer Prozessoren für höhere Parallelität im Bereich um den Wert 10 (Integer) bzw. 12 (Float) sättigt. Außerdem in Abbildung 9 zu sehen: Das Verhältnis von Joins zweier Integer-Spalten zu Joins zweier Float-Spalten ist bei dem CPU-Join (bis 1 : 0,8) viel schlechter als bei der GPU-Variante (bis 1 : 0,93).

Abbildung 10 zeigt deutlich, dass sich der Zeitpunkt des Eintretens von beschleunigter Abarbeitung des Joins auf der GPU gegenüber der CPU bei festen Größen der Primär- oder Fremdschlüssel verschiebt. Sind einige Spalten schon von vornherein groß genug (10^5 Einträge), dass die Bearbeitung auf mehr Threads der GPU aufgeteilt werden kann, tritt die besagte Beschleunigung schon bei

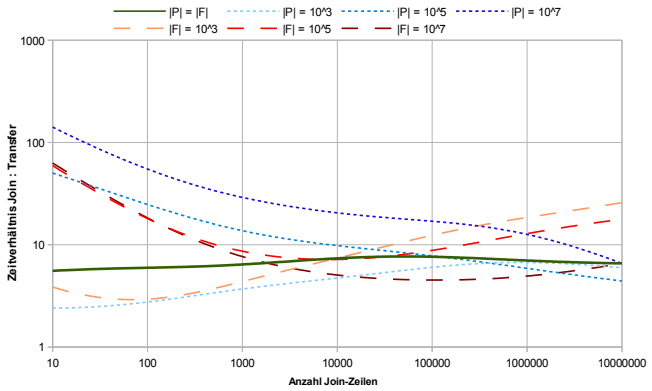

Fig. 11. Zeitverhältnisse des Joins auf der GPU zum Datentransfer zwischen Host- und Grafikspeicher bei fester Anzahl an Primärschlüsseleinträgen (P) bzw. fester Anzahl an Fremdschlüsseleinträgen (F) bei gleichzeitig variabler Größe der jeweils anderen Join-Spalte; zum Vergleich: gleiche Größe beider Spalten (P = F)

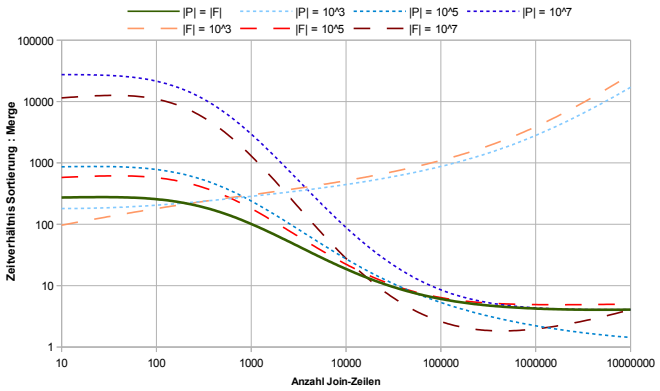

Fig. 12. Zeitverhältnisse der Sortierungs- zu den Mergephasen bei fester Anzahl an Primärschlüsseleinträgen (P) bzw. fester Anzahl an Fremdschlüsseleinträgen (F) bei gleichzeitig variabler Größe der jeweils anderen Join-Spalte; zum Vergleich: gleiche Größe beider Spalten (P = F)

einem Faktor 10 eher auf. Umgekehrt - bei konstant kleinen Spaltengrößen - tritt dieses Ereignis erst später ein. Außerdem ist aus Abbildung 10 zu schließen, dass bei umgekehrtem Größenverhältnis von Primärschlüsselspalte zu Fremdschlüsselspalte kaum ein nennenswerter Unterschiede zum nicht umgekehrtem Fall erkennbar ist. Sowohl CPU- als auch GPU-Join-Algorithmus skalieren demnach bei gleichen Größenverhältnissen von Primär- zu Fremdschlüsselspalte in etwa gleich. Das Verhältnis entscheidet aber darüber, ab wann der GPU-Join schneller als der CPU-Join ist.

Der implementierte GPU-Join ist sehr rechenintensiv. Dies lässt sich aus Abbildung 11 entnehmen, welche die (addierte) Transferzeit der Join-Spalten von Host- zu Grafikspeicher und die der aus dem Join resultierenden ID-Listen von Grafik- zu Hostspeicher mit der eigentlichen Berechnung des Join-Ergebnisses auf der GPU ins Verhältnis setzt. Für den Fall, dass beide Spalten gleichermaßen wachsen (P = F) ist dieses Verhältnis annähernd konstant. Bei kleinen (festen) Spaltengrößen (P = 1000 oder F = 1000) ist das Verhältnis etwas aufgrund der fehlenden Parallelität verfälscht und dementsprechend gering. Bei größeren (festen) Werten begrenzt die Latenz des Grafikspeichers noch, wodurch die Transferzeit verhältnismäßig hoch ist. Bei wachsender Spaltengröße der jeweils anderen Tabelle nähernn sich alle Kurven dem konstanten Verhältnis bei gleich wachsenden Primär- und Fremdschlüsselspalten an. Bei sehr großen Primärschlüsselspalten und konstanten Fremdschlüsseleinträgen von weniger als dem Maximalwert von 10^7 steigt das Verhältnis. Bei allen Testfällen ist die Dauer des Joinvorgangs größer als die Dauer des Datentransfers.

Die beiden Hauptkomponenten des Sortierens und anschließenden Mergens werden in Abbildung 12 ins Verhältnis gesetzt um zu überprüfen, welcher Prozess zeitaufwändiger ist. Es ist zu erkennen, dass bei konstant kleinen Spalten ein verhältnismäßiger Zeitgewinn des Mergens

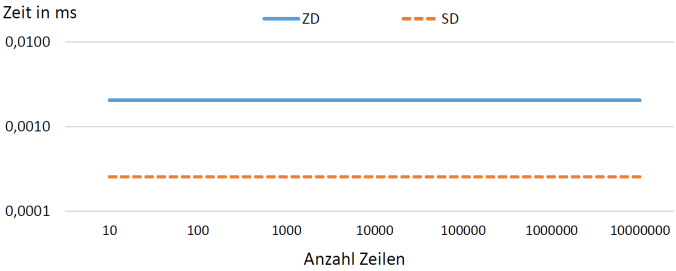

Fig. 13. Zeitmessung für den Zugriff auf das mittlere Element einer Spalte vom Typ varchar für zeilenorientierte (ZD) und spaltenorientierte Darstellungen (SD) in CoGaDB in Abhängigkeit der Tabellengröße

gegenüber des Sortiervorganges entsteht. Bei konstant großen Spalten sieht es dagegen umgekehrt aus. Dort sinkt das Zeitverhältnis rapide, bis es sich im Bereich von 1 bis 10 sättigt. Der Vorgang des Sortierens ist in jedem Testfall langsamer als der des Mergens.

C. Auswertung

Die Messungen zur Materialisierungszeit haben ergeben, dass die Größe des Wertes einer Spalte maßgeblich für dessen Materialisierungsdauer ist. Daraus resultiert, dass varchar-Spalten, welche Werte länger als vier Zeichen beinhalten, den meisten Aufwand der Materialisierung darstellen.

Die Zugriffszeit auf ein Attribut ist maßgeblich durch den Seitenaufbau bestimmt [12, 23]. Durch das Vorhalten der Offsets auf einer Seite erfolgt der Zugriff in konstanter Zeit. Jedoch liegen die Werte als Byte-Darstellung vor, sodass zunächst die Größe eines Wertes in Byte und dann der entsprechende Wert ausgelesen werden muss. Dadurch entsteht ein Performanceunterschied zwischen SD und ZD. Abbildung 13 zeigt anhand der Messergebnisse für den Zugriff auf das mittlere Element einer Spalte, dass eine SD um den Faktor 8 schneller ist, als eine ZD. Das resultiert

letztendlich auch daraus, dass die Byte-weise Darstellung in einen spezifischen Datentyp umgewandelt werden muss (z.B. Int oder Float). Diese Umwandlung ist in der SD nicht erforderlich und spart weitere Zeit ein.

Aus den Zeitmessungen der Joinalgorithmen lässt sich schließen, dass aufgrund der Latenz, die sich aus der Datenübertragung von Host- zu Grafikspeicher und zurück ergibt, eine Nutzung des GPU-Joins keine Vorteile bei kleinen Datensätzen ergibt. Für große Datensätze ist jedoch bei Beschleunigungsfaktoren von bis zu 10 (Integer-Join) bzw. 12 (Float-Join) ein sehr hoher Nutzwert für eine beschleunigte Berechnung des Joins auf der GPU gegenüber der CPU erkennbar.

Der Datentransfer von Host- zu GPU-Speicher und zurück ist bei dem GPU-Join nicht als allzu kritisch anzusehen, da das Verhältnis von Datentransferzeit zur Zeit der Join-Berechnung immer zu ungunsten der Join-Zeit ausfällt.

Zudem konnte gezeigt werden, dass das Verhältnis von Sortierzeit zu Mergezeit auf der GPU zu gunsten der Mergezeit ausfällt, was bedeutet, dass die Sortierung den Großteil der Berechnung ausmacht. Unter der Annahme, dass eine oder beide Tabellen bereits sortiert nach dem Join-Attribut vorliegen, kann die Join-Zeit daher erheblich verkürzt werden.

D. Aussagekraft

Es wurde während der Testzeit sichergestellt, dass die Experimente die einzige Lastquelle auf dem Testsystem waren. Alle Experimente wurden mehrfach durchgeführt. Die Experimente der Unabhängigen Variablen verliefen als Ceteris Paribus Experimente.

Durch die Micro-Benchmarks spiegeln unsere Experimente nicht das Performanceverhalten realer Datenbanksysteme wieder. Jedoch weisen unsere Experimente Szenarien auf, die im produktivem Einsatz üblich sind. Weiterhin sind diese Experimente für unsere Beantwortung der Fragestellung notwendig.

Eine optimierte Verwendung der SmartPointer-Architektur in CoGaDB kann die Materialisierung beschleunigen. Aktuell ist die Dauer der Materialisierung deutlich zu hoch, um sie mit anderen Systemen vergleichbar zu machen. Aus diesem Grund ist die Materialisierungszeit nicht repräsentativ.

Um aussagekräftige Ergebnisse zu erreichen, wurde jeder Join-Test 10 mal durchgeführt und die durchschnittlichen Werte berechnet. Die durchschnittliche Abweichung der Testergebnisse vom Mittelwert lag bei allen Tests durchweg unter 0,1% - die maximale Abweichung bei unter 2%. Zudem wurde die Korrektheit der Join-Ergebnisse mit Hilfe von Wertevergleichen über die aus dem Join resultierenden ID-Paaren verifiziert. Jedem Fremdschlüssel muss dabei dem richtigen Primärschlüssel zugeordnet worden sein.

VI. Verwandte Arbeiten

Abadi zeigte, dass spaltenorientierte Datenbanken in ausgewählten Anfragen, wie zum Beispiel Aggregationsanfragen deutlich einer klassische, zeilenorientierte Datenbank überlegen ist [1]. Abadi u. a. weisen darauf hin, dass eine späte Materialisierung nicht immer besser geeignet ist als eine frühe Materialisierung [3].

Weitere Datenbankprototypen, wie zum Beispiel MonetDB [6] oder dessen Erweiterung Ocelot [17] befassen sich mit der parallelen Ausführung von Operationen auf unterschiedliche (Ko-)Prozessoren.

GPU Datenbanken wie Virginian [5] oder Parstream[1] fokussieren die Ausführung von Datenbankanfragen auf die GPU, teilweise mit dem Schwerpunkt, große Datenmengen in Echtzeit zu analysieren.

VII. Zusammenfassung

In dieser Arbeit wurde experimentell gezeigt, dass eine SD gegenüber einer ZD für eine GPU-beschleunigte Verarbeitung besser geeignet ist. Anhand der Zugriffszeiten konnte gezeigt werden, dass SD gegenüber den ZD deutlich im Vorteil sind. Wird zum Beispiel eine Selektion anhand einer Spalte auf eine ZD ausgeführt, so muss erst die eine Spalte aus der ZD extrahiert werden. Dieser Prozess wird durch SD komplett umgangen.

Weiterhin müssen Spalten aus einer ZD extrahiert werden. Es konnte gemessen werden, dass der Zeitaufwand in linearer Abhängigkeit zur Spaltengröße (Anzahl Elemente) steht. Das bedeutet, dass mit wachsender Tabellengröße die GPU-Verarbeitung immer weiter verzögert, da die Materialisierungszeit linear ansteigt.

Daraus resultiert, dass SD für eine GPU-beschleunigte Verarbeitung besser geeignet sind, sofern die Materialisierungszeit vernachlässigt wird und viele Elemente in den GPU-Speicher transferiert werden müssen.

VIII. Ausblick

In diesem Abschnitt wird dargelegt, in welche Richtung weitere Untersuchungen erfolgen und welche weiteren Arbeiten entstehen könnten.

Zunächst könnte untersucht werden, welchen Einfluss eine geringe Anzahl an Seiten (dafür aber sehr große Seiten) gegenüber einer großen Anzahl an Seiten (mit wenig Inhalt) auf die Zugriffszeit hat.

Abschließend ist zu empfehlen, eine gemeinsame Betrachtung von später Materialisierung einer SD und den Exrahierungsaufwand von Spalten einer ZD für ausgewählte Queries durchzuführen. Die Dauer der Operation auf der GPU ist für SD und ZD identisch, da die Operatoren auf den selben Daten arbeiten.

Literatur

[1] Daniel J. Abadi. “Query execution in column-oriented database systems”. Diss. Massachusetts Institute of Technology, 2008.

[1] http://www.parstream.com/

[2] Daniel J. Abadi, Samuel R. Madden und Nabil Hachem. “Column-Stores vs. Row-Stores: How different are they really?” In: *SIGMOD*. ACM, 2008, S. 967–980.

[3] Daniel J. Abadi u. a. “Materialization strategies in a column-oriented DBMS”. In: *ICDE*. IEEE, 2007, S. 466–475.

[4] Cédric Augonnet und Raymond Namyst. “A unified runtime system for heterogeneous multi-core architectures”. In: *Euro-Par 2008 Workshops-Parallel Processing*. Springer. 2009, S. 174–183.

[5] Peter Bakkum und Srimat Chakradhar. “Efficient Data Management for GPU Databases”. In: *ACM* (2012).

[6] Peter A. Boncz, Marcin Zukowski und Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *CIDR*. Bd. 5. 2005, S. 225–237.

[7] Sebastian Breß u. a. “Exploring the Design Space of a GPU-aware Database Architecture”. In: *ADBIS workshop on GPUs In Databases (GID)*. 2013.

[8] Sebastian Breß u. a. “Efficient Co-Processor Utilization in Database Query Processing”. In: *Information Systems* 38.8 (2013), S. 1084–1096.

[9] Nicolas Bruno. “Teaching an old elephant new tricks”. In: *CIDR* 4 (2009), S. 1–6.

[10] George P. Copeland und Setrag N. Khoshafian. “A decomposition storage model”. In: *ACM SIGMOD Record*. 1985, S. 268–279.

[11] Gregory Diamos u. a. *Efficient Relational Algebra Algorithms and Data Structures for GPU*. Techn. Ber. GIT-CERCS-12-01. CERCS, Georgia Institute of Technology, 2012.

[12] Wolfgang Effelsberg und Theo Haerder. “Principles of database buffer management”. In: *ACM Transactions on Database Systems (TODS)* 9.4 (1984), S. 560–595.

[13] Naga K. Govindaraju u. a. “Fast computation of database operations using graphics processors”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, S. 215–226.

[14] Chris Gregg und Kim Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer”. In: *Performance Analysis of Systems and Software, IEEE International Symmposium on* 0 (2011), S. 134–144.

[15] Bingsheng He u. a. “Relational joins on graphics processors”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD ’08. New York, NY, USA: ACM, 2008, S. 511–524.

[16] Bingsheng He u. a. “Relational query coprocessing on graphics processors”. In: *ACM Transactions on Database Systems (TODS)* 34.4 (2009), S. 1–39.

[17] Max Heimel. “Designing a database system for modern processing architectures”. In: *SIGMOD*. ACM. 2013, S. 13–18.

[18] E. Ilavarasan und P. Thambidurai. “Low complexity performance effective task scheduling algorithm for heterogeneous computing environments”. In: *Journal of Computer sciences* 3 (2007), S. 94–103.

[19] Rolf Isernhagen und Hartmut Helmke. *Softwaretechnik in C und C++: das Kompendium*. 4. Aufl. Hanser Verlag, 2004.

[20] Martin Krulis und Jakub Yaghob. “Revision of Relational Joins for Multi-Core and Many-Core Architectures”. In: *DATESO*. 2011, S. 229–240.

[21] Chi-Keung Luk, Sunpyo Hong und Hyesoon Kim. “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping”. In: *Microarchitecture*. 2009, S. 45–55.

[22] Stefan Manegold, Peter Boncz und Martin L. Kersten. “Database architecture optimized for the new bottleneck: Memory access”. In: *VLDB*. Bd. 99. 1999, S. 54–65.

[23] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp, 2011.