TU Dortmund
Fakultät für Informatik
Lehrstuhl 6

Bericht der Projektgruppe 589

Hauptspeicherdatenbanken auf moderner Hardware

SliceDB

Teilnehmer:
Maximilian Berens
Andreas Blume
Eric Fiege
Harun Kara
Philipp Krause
Milad Nayebi
Stefan Noll
Ersin Özdemir
Majuran Rajakanthan
Tristan Schäfer
Christian Schnieder
Jan Stallmann

Betreuer:
Sebastian Breß
Michael Moritz Kußmann
Prof. Dr. Jens Teubner

2. März 2016

Vorwort *(Stefan Noll)*

Diese Arbeit ist der abschließende Bericht der Projektgruppe 589 „Hauptspeicherdatenbanken auf moderner Hardware“. Bei der Projektgruppe 589 handelt es sich um eine Lehrveranstaltung des Lehrstuhls 6 der Fakultät für Informatik an der Technischen Universität Dortmund. Die Lehrveranstaltung fand während des Sommersemesters 2015 und Wintersemesters 2015/2016 statt. Betreuer der Projektgruppe waren Sebastian Breß (erstes Semester), Michael Moritz Kußmann (zweites Semester) und Prof. Dr. Jens Teubner.
Gemäß der Prüfungsordnungen für den Studiengang Master Informatik oder Angewandte Informatik an der Technischen Universität Dortmund ist die Teilnahme an einer Projektgruppe (PG) Voraussetzung für einen erfolgreichen Abschluss des Studiums. Eine Projektgruppe besteht dabei aus zehn bis zwölf Studierenden, die ausgehend von einer praktischen Problemstellung, ein Thema zunehmend selbstständig und in kleineren Untergruppen erarbeiten, die Realisierung planen und schließlich implementieren und dokumentieren. Das Ziel der PG wird von den Veranstaltern im Antrag der Projektgruppe formuliert und vorgegeben.
Die Projektgruppe startet mit einer Seminarphase, in der alle Teilnehmer einen Vortrag über ein Teilgebiet des Themas der PG halten. Nach diesem Einstieg arbeiten die Studierenden unter Aufsicht der Betreuer zwei Semester lang in Eigenverantwortung an der Erfüllung des Ziels der Projektgruppe. Das Ergebnis wird in Form eines Berichtes dokumentiert.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation und Hintergrund . . . 1
1.2 Aufgabe der PG . . . 3
1.2.1 Minimalziele . . . 4
1.3 Organisation der PG . . . 4
1.4 Aufbau des Berichts . . . 5

2 Grundlagen - Ergebnisse der Seminarphase **7**
2.1 Vectorwise . . . 7
2.1.1 Technologie . . . 8
2.1.2 Datenspeicherung . . . 8
2.1.3 Anfrageverarbeitung . . . 10
2.1.4 Star Schema Benchmark . . . 15
2.2 MonetDB/Ocelot . . . 16
2.2.1 Entwicklung und Geschichte . . . 16
2.2.2 Features und Aufbau . . . 18
2.2.3 Ocelot . . . 23
2.2.4 Performance . . . 24
2.2.5 Vergleich zu SliceDB . . . 25
2.3 HyPer . . . 26
2.3.1 Überblick . . . 26
2.3.2 OLTP-Anfragen . . . 27
2.3.3 OLAP-Anfragen . . . 27
2.3.4 Sicherung & Wiederherstellung . . . 29
2.3.5 Anfragen als Maschinencode . . . 29
2.3.6 Morsel-getriebener Parallelismus . . . 32
2.3.7 Performanz . . . 35
2.3.8 Fazit . . . 37
2.4 CoGaDB . . . 38
2.4.1 Architektur . . . 38
2.4.2 Operatoren . . . 40
2.4.3 HyPE . . . 44
2.4.4 Performance . . . 46
2.4.5 Zusammenfassung . . . 48
2.5 IBM DB2 BLU . . . 48
2.5.1 Blink . . . 48
2.5.2 DB2 BLU . . . 58
2.5.3 Was können wir von dem System lernen? . . . 65

2.6 SAP HANA . . . 66
2.6.1 Überblick . . . 66
2.6.2 Eingesetzte Techniken und Konzepte . . . 68

3 SliceDB **79**
3.1 Interface . . . 79
3.1.1 Bedienung . . . 79
3.1.2 Konfiguration & Parameter . . . 81
3.1.3 Logging . . . 82
3.1.4 Syntax . . . 83
3.1.5 Anfrageplan . . . 85
3.1.6 Schema Import . . . 86
3.1.7 Parser . . . 87
3.1.8 Automatische Optimierung . . . 90
3.1.9 Ausführung des Anfrageplans . . . 91
3.2 Datenhaltung . . . 94
3.2.1 Datentypen . . . 95
3.2.2 Repräsentation der Daten . . . 95
3.2.3 Speicherverwaltung . . . 98
3.2.4 Dictionary Compression . . . 103
3.2.5 Operationen auf Spalten . . . 106
3.2.6 Schemaverwaltung . . . 106
3.2.7 Histogramme . . . 107
3.2.8 Zwischenergebnisse . . . 108
3.2.9 Zusammenfassung . . . 111
3.3 Operatoren . . . 112
3.3.1 Selektion . . . 112
3.3.2 Gather . . . 117
3.3.3 Nested-Loop Join . . . 118
3.3.4 Hash-Join . . . 121
3.3.5 Operator für arithmetische Ausdrücke . . . 124
3.3.6 Sortieren . . . 132
3.3.7 Gruppierungen & Aggregation . . . 153
3.3.8 Invisible Join . . . 169
3.3.9 Bloom-Filter . . . 174
3.3.10 Insert . . . 181

4 Projektablauf **183**
4.1 Erstes Semester . . . 183
4.1.1 Verbesserungsvorschläge . . . 185
4.2 Zweites Semester . . . 186
4.2.1 Aufgabenverteilung . . . 186
4.2.2 Bewertung der Gruppenarbeit . . . 187

5 Zusammenfassung und Ausblick **189**
5.1 Zusammenfassung . . . 189
5.1.1 Benchmarks . . . 189
5.1.2 Korrektheit . . . 193
5.2 Fazit . . . 195
5.3 Ausblick . . . 196

Kapitel 1

Einleitung *(Stefan Noll)*

Dieses Kapitel führt in das grundlegende Thema und die Problemstellung der Projektgruppe ein. Hierfür wird das Thema zunächst motiviert und vorgestellt. Des weiteren wird die Bedeutung der Problemstellung im Kontext von Datenbanksystemen eingeordnet. Im Anschluss werden die zentrale Aufgabe der Projektgruppe sowie die daraus abgeleiteten Minimalziele hervorgehoben, die für die Lösung der Problemstellung erforderlich sind. Um die Struktur des Berichtes besser verstehen zu können, wird abschließend ein Überblick über den Aufbau und Inhalt gegeben.

1.1 Motivation und Hintergrund

Die folgenden Kapitel 1.1, 1.2 sowie 1.2.1 basieren zum Teil stark auf dem Projektgruppenantrag.

Relationale Datenbankmanagementsysteme (DBMS) sind ein weit verbreitetes, kommerziell verfügbares Werkzeug, das in vielfältigen Anwendungen zur Verwaltung und Verarbeitung von Daten und der damit ausgedrückten Information eingesetzt wird. In den letzten zwanzig Jahren wurden Datenbankmanagementsysteme jedoch nicht nur für operative Zwecke eingesetzt, sondern auch für die Analyse großer integrierter Datenbestände, den sogenannten *Data Warehouses* genutzt. Die Anfragen an Data Warehouse-Datenbanken dauern in klassischen DBMS sehr lange, da unter anderem sehr große Datenmengen verarbeitet werden müssen.
Im Allgemeinen müssen Datenbankmanagementsysteme in der Praxis eine Vielzahl an Aufgaben erfüllen, wobei die wichtigsten Aufgaben die persistente, redundanzfreie Speicherung von Daten und die effiziente Abfrage solcher Datenbestände durch eine deklarative Anfragesprache sind. Die einzelnen Komponenten eines DBMS sind in Abbildung 1.1 schematisch dargestellt. Die hier visualisierte klassische Datenbankarchitektur gilt nahezu für alle heute auf dem Markt verfügbaren Systeme.
Ganz oben in der Hierarchie steht die Benutzerschnittstelle. Diese nimmt Anfragen an und erzeugt aus diesen einen Anfrageplan, eine systeminterne Repräsentation aller für die Anfrage notwendigen Operationen und Informationen (*Parser*). Ein Anfrageoptimierer versucht daraufhin, durch die Anwendung von Trans-

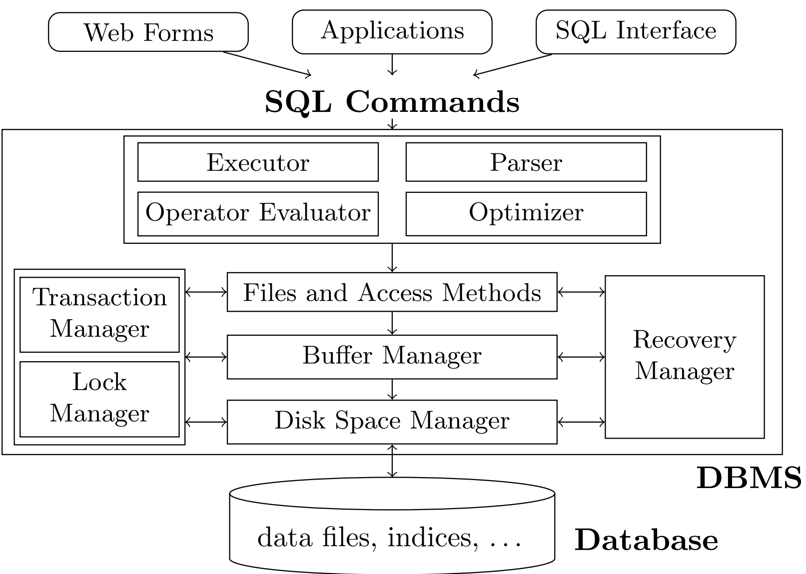


Abbildung 1.1: klassische Architekur eines DBMS

formationsregeln und kostenbasierten Analysen einen effizienten Anfrageplan zu bestimmen (*Optimizer*). Eine Ausführungseinheit (*Executor*) führt schließlich den optimierten Anfrageplan aus, indem die im Anfrageplan enthaltenen Operatoren ausgeführt werden (*Operator Evaluator*).
Die Operatoren fordern ihre Eingabedaten von einer Persistenzschicht an, die Datenbankobjekte auf Blöcken fester Größe (sogenannten Seiten) speichert (*Files and Access Methods* → *Buffer Manager* → *Disk Space Manager*). Die Persistenzschicht entscheidet transparent zum übrigen Teil des Systems, welche Daten im Hauptspeicher gehalten und welche Daten nur auf einem Massenspeicher gehalten werden. Bei einem Hauptspeicherdatenbanksystem liegen die (meisten) Daten im flüchtigen Hauptspeicher, was im klassischen Modell in etwa dem *Buffer Manager* entspricht. Im Fall eines Systemausfalls sollte der ursprüngliche Systemzustand mit Hilfe der Daten auf der Festplatte wieder hergestellt werden können (*Recovery Manager*).
Darüber hinaus wird in der Regel auch ein Mehrbenutzerbetrieb unterstützt. Anfragen unterschiedlicher Benutzer werden von einander abgeschirmt und Ressourcen geschützt (*Lock Manager*). Unterschiedliche Anfragen müssen verwaltet werden und können jederzeit abgebrochen sowie rückgängig gemacht werden (*Transaction Manager*).

Aktuelle Entwicklungen

Mit zunehmender Größe von Hauptspeichern in modernen Servern (Terabytes und mehr) ist es möglich, den gesamten Datenbestand (oder zumindest einen großen Teil) im Hauptspeicher zu halten. Dadurch verschiebt sich der klassische Flaschenhals vom Festplatten-I/O zu Hauptspeicherzugriffen.
In diesem Zusammenhang ergeben sich viele neue Probleme, die im klassischem Fall bei der Nutzung von Festplatten zur Datenhaltung eine viel geringere Bedeu-

tung hatten. Eines dieser Probleme, die zunehmende Geschwindigkeitsdifferenz zwischen Prozessor und Hauptspeicher, bezeichnet man als sogenannte *Memory Wall* [69].

Die Geschwindigkeit von sowohl Prozessoren als auch von Hauptspeicher wächst durch neue Entwicklungen zwar exponentiell, jedoch ist der Wachstumsfaktor bei CPUs sehr viel größer als bei DRAM. Da die Differenz zwischen den exponentiell wachsenden Geschwindigkeiten ebenfalls exponentiell wächst, wird der Geschwindigkeitsunterschied zwischen Prozessor und Hauptspeicher im Laufe der Zeit immer größer und damit zunehmend zum Problem.

Konkret bedeutet dies, dass die Anzahl der Taktzyklen, die die CPU braucht um auf den DRAM Speicher, der sich sich nicht direkt auf dem Chip befindet, zuzugreifen, um einen Faktor von 100 größer ist als bei Recheninstruktionen. Prozessoren verschwenden also viel Rechenzeit durch Warten auf den Hauptspeicher, sodass die Performanz eines System vollständig durch die Geschwindigkeit des verbauten Speichers dominiert wird.

Dieses inhärente Problem der Hardware wird von den Hardwareherstellern im wesentlichen durch die Verwendung von Caches versucht abzuschwächen. In heutigen Systemen findet man daher überlichweise drei unterschiedliche Arten von Caches: *instruction cache*, *data cache* sowie *translation lookaside buffer (TLB)*. Der Cache für die Daten wird außerdem in mehreren Hierarchien von unterschiedlich schnellem SRAM Speicher angeordnet (L1, L2, etc.), der direkt mit der CPU auf dem Chip verbaut wird.

In solchen Szenarien kann das Potential der vorhandenen Hardware nur ausgenutzt werden, wenn Algorithmen die Eigenschaften von Prozessoren, Caches und Hauptspeicher gezielt und gewinnbringend ausnutzen. In der Literatur wurde zum Beispiel intensiv die Cache-Effizienz von Algorithmen diskutiert [7, 11, 12]. Hier stellt sich für analytische Datenbankanfragen heraus, dass eine spaltenbasierte Datenrepräsentation zu einer wesentlich höheren Cache-Effizienz führt als die klassische zeilenbasierte Repräsentation.

Auch das Ausführungsmodell der Ausführungseinheit spielt eine entscheidende Rolle in der Performanz eines Datenbankmanagementsystems. Das in klassischen Datenbankmanagementsystemen genutzte Volcano-Modell [28] erweist sich in Hautpspeicherdatenbanken als nicht optimal. Hier wurden *Operator-at-a-Time* und *Vector-at-a-Time* [17, 73] als Anfrageauswertungsstrategien vorgeschlagen.

1.2 Aufgabe der PG

Im Rahmen dieser Projektgruppe soll ein relationales Datenbankmanagementsystem auf Basis von hauptspeicheroptimierten Datenbanktechniken entworfen, prototypisch implementiert und die umgesetzten Techniken evaluiert werden. Dabei sollen insbesondere vorgeschlagene Ansätze aus der Literatur aufgearbeitet und ihr Zusammenspiel in einem Gesamtsystem untersucht werden. Am Lehrstuhl für Datenbanken und Informationssysteme stehen entsprechende Rechnersysteme mit modernen Multi-Core-Prozessoren und großem Hauptspeicher zur Verfügung. Diese sollen der Projektgruppe als Experimentier- und Evaluationsplattform dienen.

Ziele

Im Detail sollen die folgende Teilziele durch die Projektgruppe erreicht werden:

1. Einarbeitung in den Aufbau moderner Hauptspeicherdatenbanken.
   (a) Literaturrecherche bezüglich cacheoptimierter Algorithmen für Datenbankoperatoren und Integration ausgewählter Algorithmen im Systementwurf.
   (b) Literaturrecherche von Techniken für die effiziente Datenverarbeitung im Hauptspeicher (z.B. *Vektor-at-a-Time* Anfrageauswertung oder leichtgewichtige Kompression).
2. Ausarbeitung eines Entwurfs für eine Hauptspeicherdatenbank. Die klassische Datenbankarchitektur (Abbildung 1.1) soll dabei als Orientierungspunkt dienen.
   - Angestrebt wird, dass das resultierende System in der Lage ist, alle Anfragen des Star Schema Benchmark [51] zu verarbeiten.
   - Techniken aus 1a und 1b sollen möglichst im Systementwurf berücksichtigt werden.
3. Implementierung der Entwürfe.
4. Experimentelle Auswertung und Bewertung der implementierten Techniken und Algorithmen anhand eines aussagekräftigem OLAP-Benchmarks.

1.2.1 Minimalziele

Folgende Ziele sollen im Rahmen der Projektgruppe mindestens erreicht werden:

- Entwurf und Implementierung eines lauffähigen Hauptspeicherdatenbanksystem. Dieses sollte mindestens in der Lage sein, die Anfragen des Star Schema Benchmark zu verarbeiten. Die Realisierung einzelner Techniken im Stil der Punkte 1a und 1b wäre wünschenswert, ist jedoch nicht Teil der Minimalziele.
- Die gewonnenen Ergebnisse müssen mindestens derart dokumentiert werden, dass sie als Ausgangspunkt für weiterführende Abschlussarbeiten verwendbar sind.
- Erstellung eines Endberichts
- PG-Vortrag

1.3 Organisation der PG

Während der Zeit im ersten Semester wurden drei Untergruppen aus allen Teilnehmern der Projektgruppe gebildet, um die Aufgaben strukturiert sowie parallel bearbeiten zu können. Außerdem wird so die Möglichkeit geschaffen, dass sich Gruppen und einzelne Teilnehmer stärker auf Teilaufgaben konzentrieren können. Jeder Teilnehmer der Projektgruppe ist dadurch in der Lage sich in einem Gebiet zu spezialisieren und bei Fragen anderen als Ansprechpartner zur Verfügung zu stehen. Konkret wurden folgende Untergruppen gebildet:

1. **Planinterface & Optimierer**

   Aufgabe: Planung und Implementierung eines Interfaces (mit Anfrageplan), Entwicklung einer Algebra als Anfragesprache und Integration durch einen Parser, Implemetierung des Volcano Interfaces zur Ausführung von Anfragen, Implemetierung erster Optimierungsansätze (Transformation des Anfrageplans, Push-Down-Selection, späte Materialisierung).

   Mitglieder: Christian Schnieder, Maximilian Berens, Stefan Noll

2. **Datenhaltung**

   Aufgabe: Planung und Implementierung der Datenhaltung und Persistenz, Verwaltung von Datenbankschema und Metadaten, Entwicklung eines polymorphen Typsystems, effiziente Speicherverwaltung im Hauptspeicher.

   Mitglieder: Harun Kara, Jan Stallmann, Majuran Rajakanthan, Milad Nayebi, Philipp Krause

3. **Opertoren**

   Aufgabe: Planung und Implementierung von (optimierten) Operatoren zur effizienten Anfrageverarbeitung.

   Mitglieder: Andreas Blume, Eric Fiege, Ersin Özdemir, Tristan Schäfer

Im zweiten Semester der Projektgruppe wurde eine andere Form der Arbeitsaufteilung gewählt. Die Gruppen wurde aufgelöst, sodass jeder einzeln an einem bestimmten Teil des Datenbanksystems mögliche Optimierungen implementieren konnte.
Während der gesamten Zeit der Projektgruppe wurde die Projektmanagementsoftware *Redmine* [35] zur Aufgabenverteilung und Kommunikation genutzt. Über ein Ticketsytem wurde die Verwaltung von Teilaufgaben durchgeführt, mögliche Fristen festgehalten und themenspezifisch diskutiert. In einem Wiki wurden zentrale Informationen sowie Beschlüsse der Gruppe festgehalten und dokumentiert. Ein Forum bzw. eine eingerichtete Mailingliste wurde darüber hinaus ebenfalls für die Kommunikation untereinander genutzt. Außerdem gab es in der Regel auch immer wöchentliche Treffen.

1.4 Aufbau des Berichts

Zunächst werden in Kapitel 2 die wesentlichen Erkenntnisse aus der Seminarphase über aktuell verfügbare Datenbankmanagementsysteme zusammengefasst und dort verwendete Techniken und Konzepte für eine mögliche Nutzung bewertet. Anschließend wird in Kapitel 3 das entwickelte Datenbanksystem *SliceDB* vorgestellt. Dabei werden die wesentlichen Konzepte und Techniken präsentiert sowie viele Details zur Implementierung beschrieben. Insbesondere wird auf das Interface, die Datenhaltung sowie die implementierten Operatoren eingegangen. In Kapitel 4 wird dann näher auf den gesamten Projektablauf während des ersten und zweiten Semesters eingegangen. Außerdem wird auch die Gruppenarbeit in der Projekgruppe insgesamt bewertet. Zum Schluss wird in Kapitel 5 auf die

erreichten bzw. nicht erreichten Ziele der Projektgruppe eingegangen sowie die zentralen Ergebnisse der gesamten Arbeit zusammengefasst. Darüber hinaus wird auch das entwickelte Datenbanksystem *SliceDB* anhand des *Star Schema Benchmark* mit anderen Datenbanksystemen abschließend verglichen. Als letztes wird noch ein Ausblick gegeben, indem denkbare Ansätze zu nachfolgenden Untersuchungen oder möglichen Erweiterungen der Datenbank vorgeschlagen werden.

Kapitel 2

Grundlagen - Ergebnisse der Seminarphase

(Majuran Rajakanthan)

In diesem Kapitel werden folgende Hauptspeicher-Datenbanksysteme, die in der Seminarphase von den einzelnen Gruppenmitgliedern untersucht wurden, beschrieben:

- Vectorwise
- MonetDB/Ocelot
- HyPer
- CoGaDB
- IBM DB2 BLU
- SAP HANA

2.1 Vectorwise

Vectorwise ist ein relationales Hauptspeicherdatenbanksystem, das von Actian Corporation für hohe Performance auf moderner Hardware und Analyse großer Datenmengen entwickelt wurde. Dadurch ist es insbesondere für *Online Analytical Processing (OLAP)* geeignet. Im Gegensatz zum *Online Transaction Processing (OLTP)* werden hier sehr große Datenmengen analysiert und ausgewertet. Es stehen somit keine transaktionalen *Queries*, sondern hauptsächlich lesende Anfragen, im Vordergrund. Seit 2011 befindet sich Vectorwise in den *Top Ten Perfomance Results (Non-Clustered)* des TPC-H Benchmark für Datenbankgrößen von 100 GB, 300 GB und 1 TB [66].
Im Folgenden wird zunächst die verwendete Technologie in Vectorwise vorgestellt. Im darauf folgenden Abschnitt wird auf die Datenspeicherung und Kompressionsverfahren eingegangen. Anschließend wird die Anfrageverarbeitung dargestellt. Im letzten Abschnitt werden die Ergebnisse des Star Schema Benchmarks präsentiert.

2.1.1 Technologie

Im Gegensatz zu traditionellen Datenbanksystemen, wo die Speicherung der Daten zeilenorientiert stattfindet, wird in Vectorwise eine spaltenorientierte Speicherung der Daten verwendet. Die zusätzliche Kompression der Daten sorgt für einen hohen I/O Durchsatz.

Vectorwise macht bei der Anfrageverarbeitung von der sogenannten *Vectorized Query Execution* Gebrauch, um gleiche Operationen auf mehrere Daten parallel auszuführen. Dafür wird *Vector processing* und *Single Instruction, Multiple Data (SIMD)* verwendet (s. Abschnitt 2.1.2). Dadurch wird die Leistung moderner CPUs für eine effiziente Anfrageverarbeitung ausgenutzt.

2.1.2 Datenspeicherung

Vectorwise verwendet ein hybrides spalten-/zeilenorientiertes Speicherverfahren, welches auf dem *Partition Attributes Across (PAX)* Layout basiert. Abb. 2.1 zeigt das in traditionellen Datenbanksystemen oft genutzte *N-ary storage model (NSM)* und im Vergleich dazu PAX.

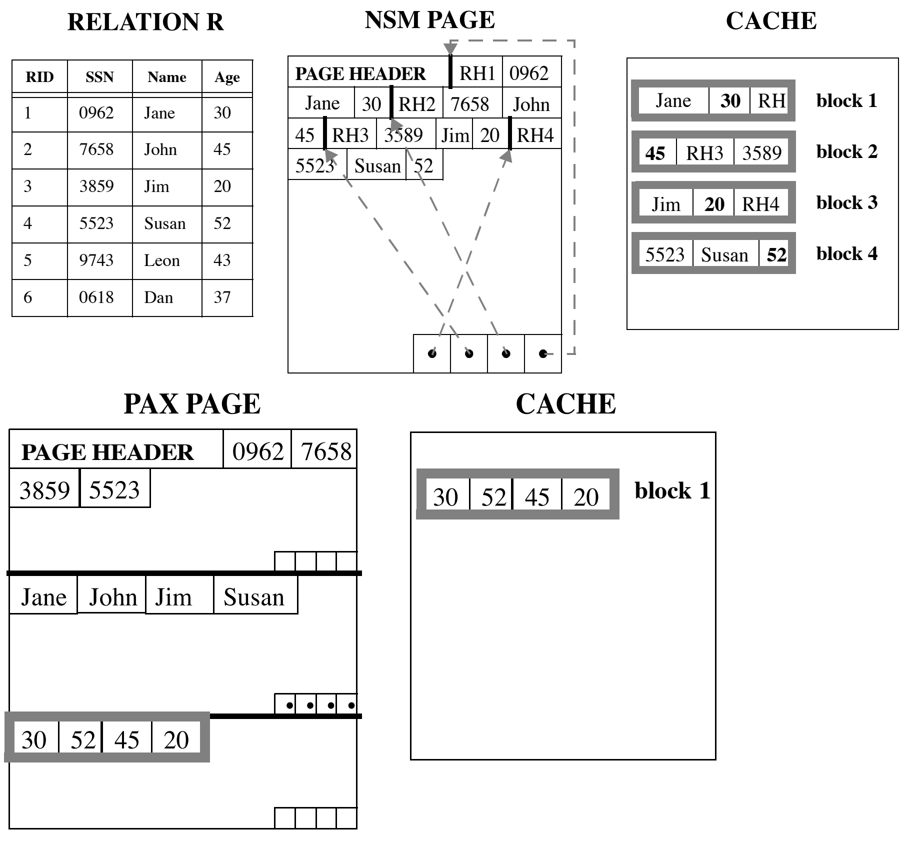


Abbildung 2.1: NSM und PAX Layout [6].

PAX

NSM speichert Daten sequenziell in einer Page mit einem Offset am Ende der Page, die auf den jeweiligen Beginn der Daten zeigen. Viele Queries greifen nur auf einen begrenzten Datenbereich zu. Durch dieses Speichermodell werden im Cache allerdings auch Daten gehalten, die irrelevant für die Query sind und zur ineffizienten Speicherverwaltung führen kann (s. Abb. 2.1). PAX speichert die gleichen Daten wie NSM auf einer Page. Allerdings, werden innerhalb einer Page die Daten nach dem jeweiligen Spaltentyp gruppiert in eine sogenannte *Minipage* gespeichert (s. Abb. 2.1). Dadurch wird der Cache voll ausgenutzt, da sich für die Query notwendige Daten darin befinden. [72]

Durch die spaltenorientierte Speicherung werden in Vectorwise die I/O Kosten reduziert und die Effizienz somit verbessert. Dafür verwaltet Vectorwise automatisch für jede Spalte sogenannte *MinMax* Indizes. Dadurch werden bei einem Table Scan nicht gesuchte Wertebereiche schnell eliminiert. Diese Indizes sind im RAM gespeichert. [4]

Positional Delta Trees

Ein sogenannter *Positional Delta Tree (PDT)* ist eine Baumstruktur im RAM. Da kleine Insert-, Update- oder Deleteoperationen auf spaltenorientierten Datenbanken teuer sind, hat Vectorwise PDTs eingeführt. Diese speichern die Position und die jeweilige Veränderung (delta) an diesen. Dabei wird von den PDTs ein bestimmter Speicheranteil im RAM genutzt. Bis dieser aufgebraucht ist, werden die Änderungen im RAM gehalten und erst danach auf die Platte geschrieben. Dadurch werden unnötig viele I/O Zugriffe auf die Platte vermieden. Um die ACID Eigenschaft zu gewährleisten, nutzt Vectorwise *Write Ahead Log (WAL)* für *commited PDTs*.

Kompression

Vectorwise komprimiert Daten spaltenweise unter Benutzung von verschiedenen Algorithmen. Durch die spaltenorientierte Speicherung können die Daten effizienter komprimiert und dekomprimiert werden: Für jeden Datentyp gibt es einen geeigneteren Algorithmus, diesen zu komprimieren. Da jede Spalte einen bestimmten Datentyp besitzt ist es einfacher, als bei der traditionellen zeilenorientierten Speicherung, eine geeignete Komprimierung zu finden.
Vectorwise komprimiert Daten automatisch anhand einer geeigneten Methode für jede Spalte und deren Datentyp. Es werden dabei Kompressionen wie beispielsweise Run Length Encoding oder Dictionary Encoding genutzt, die eine sehr schnelle Dekomprimierung bieten. Durch den sehr geringen Overhead für die Dekomprimierung können sich Daten komprimiert im Buffer Pool befinden und kurz vor der Anfrageverarbeitung dekomprimiert werden. Daher kann die Größe des Buffer Pools erhöht werden, wodurch gleichzeitig die I/O Kosten gesenkt werden.

2.1.3 Anfrageverarbeitung

(Philipp Krause)

Moderne CPU's können enorme Mengen an Daten pro Sekunde verarbeiten, sofern diese unabhängig voneinander sind und die CPU somit parallel auf den Daten arbeiten kann. Im Falle von z.B. OLAP oder Data-Mining ist genau das der Fall, weshalb angenommen werden könnte, dass die IPC (Instructions Per Cycle) nahezu ideal sein sollten. Trotz der Unabhängigkeiten der Daten weisen viele Datenbanksysteme eine schlechte IPC-Effizienz auf [72]. Dies ist mit der Architektur vieler DBMS zu begründen, da die Architektur der DBMS den Compiler an seinen Optimierungstechniken hindert. Ein Beispiel dafür bietet die *tuple-at-a-time-execution*. Dabei wird immer nur ein Tupel gleichzeitig und vollständig verarbeitet, bevor die Bearbeitung des nächsten Tupels erfolgt [72]. Hierbei findet folglich keine parallele Ausführung statt.
In *MonetDB* wird eine sog. *column-at-a-time-execution* verwendet. Dabei wird direkt eine vollständige Spalte einer Tabelle verarbeitet. Bei dieser Technik entsteht zwar nicht das Problem der fehlenden parallelen Ausführung wie bei der tuple-at-a-time-execution, aber es können - abhängig von der Spaltengröße - sehr große Datenströme entstehen, weshalb dieses Verfahren stark durch die Speicherbandbreite limitiert wird und zu einer schlechteren CPU-Effizienz führen kann [72].

X100

Das *X100* stellt ein neu konzipiertes Anfragesystem dar. Es kombiniert die spaltenweise Ausführung von MonetDB mit *volcano-style-pipelining*, das die gleichzeitige Bearbeitung von Vektoren durch Operatoren ermöglicht. Vektoren, die von einem Ausdruck verwendet wurden, können - während die Daten im CPU-Cache sind - als Input für den nächsten Ausdruck dienen. X100 verwendet weitesgehend die standardisierte relationale Algebra und hat die Bearbeitung von großen Datenmengen mit hoher CPU-Effizienz als Ziel [72].

Volcano Iterator Model

Bei der Anfrageverarbeitung müssen mehrere Algorithmen miteinander kombiniert werden. Die naive Vorgehensweise (Abb. 2.2) dafür wäre, dass jeder Algorithmus seine jeweilige Eingabe zunächst liest und anschließend wieder auf die Festplatte schreibt. Diese Vorgehensweise hat jedoch zwei Nachteile: Zum einen werden hohe I/O-Kosten durch das Schreiben verursacht und zum anderen kann keine parallele Bearbeitung durchgeführt werden [33].

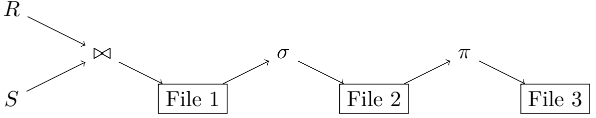


Abbildung 2.2: Naive Kombination relationaler Operatoren [33]

Beim Volcano Iterator Modell arbeiten die Algorithmen parallel, d.h., die Ausgabe eines Algorithmus wird direkt an den nächsten Algorithmus weitergeleitet. Dieses

Vorgehen wird auch als *Pipelining* bezeichnet. Die Auswertung der einzelnen Ausdrücke erfolgt dabei von oben nach unten (*Top-Down*). Jeder Operator implementiert dazu ein uniformes Interface, das über folgende drei Methoden verfügt [33]:

- **open():** interne Datenstrukturen werden initialisiert
- **next():** das nächste Tupel der Ergebnismenge wird ermittelt (<eof>, wenn Menge vollständig)
- **close():** interne Ressourcen werden aufgeräumt (Locks etc.)

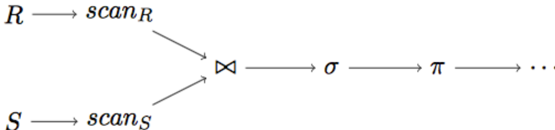


Abbildung 2.3: Pipelining relationaler Operatoren [33]

Eine Anwendung der Methoden nach Abb. 2.3 würde demzufolge wie folgt aussehen:

Vectorized Iterator Model

Die Operatoren von Anfragen besitzen - analog zum Volcano Iterator Model - eine Baumstruktur. Der Unterschied zu diesem Modell liegt darin, dass die Operatoren keine einzelnen Tupel, sondern Vektoren für den Datentransfer verwenden. Vektoren stellen die Basiseinheit für die Datenmanipulation und den Datentransfer dar. Jeder Vektor enthält ein eindimensionales Array, das typischerweise aus ca. 100-1000 Tupeln besteht [72]. Die Vektoren sind *kurzlebig*, d.h., dass sie aufgrund der Iterationen von Operatoren verschiedene Daten enthalten.
Der Vorteil des Prinzips von Vektoren mit vertikal zerlegten Daten ist das triviale Löschen bzw. Hinzufügen von Spalten. Ein Kopieren der Daten entfällt somit. Zudem bietet diese Variante einen besseren sequentiellen Speicherzugriff und fördert SIMD-Instruktionen (Single Instruction, Multiple Data) [72].

Query Language

Listing 2.1: TPC-H Query 1 mit SQL Syntax (vereinfacht) [72]

```
1: SELECT   sum(l_extendedprice * (1 - l_discount))
2:            AS sum_disc_price
3: FROM     lineitem
4: WHERE    l_shipdate < date("1998-09-02")
5: GROUP BY l_returnflag
```

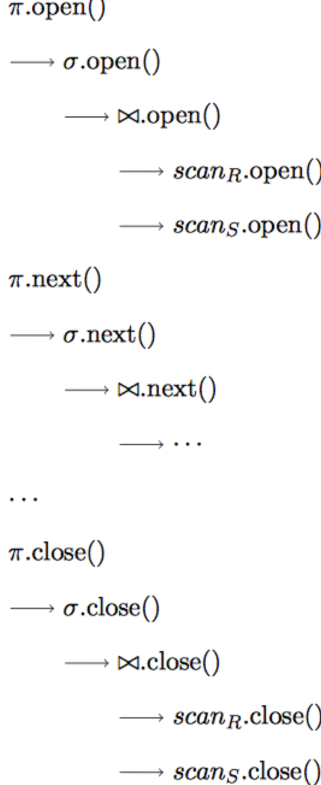


Abbildung 2.4: Uniforme Schnittstelle des Pipelinings [33]

Listing 2.2: TPC-H Query 1 mit X100 Syntax (vereinfacht) [72]

```
1: Aggr (
2:   Project (
3:     Select (
4:       Scan (Table (lineitem),
5:     < (shipdate , date('1998-09-03'))),
6:   [ discountprice = * ( -( flt('1.0'), discount),
7:                           extendedprice) ]),
8: [ returnflag ],
9: sum_disc_price = sum(discountprice) ])
```

Die Ausführung (Abb. 2.4) erfolgt nach dem Volcano-Pipelining-Prinzip über vier Operatoren [72]:

- **Scan-Operator**: Ruft Daten nach *vector-at-a-time* ab.
- **Select-Operator**: Legt einen *selection-vector* mit den Positionen passender Tupel an.
- **Project-Operator**: Berechnet die Ausdrücke für die Aggregation. Die Ergbnisse werden im Output-Vektor an die gleiche Position wie im Input-Vektor geschrieben und der selection-vector wird weiterpropagiert.
- **Aggregation-Operator**: Für jedes Tupel wird die Position im Hash-Table berechnet und damit die aggregierten Ergebnisse upgedatet. Das Anfrageergebnis ist verfübar, sobald von den darunterliegenden Operatoren keine weiteren Vektoren erzeugt werden.

Operatoren können beliebig viele Input-Attribute entgegennehmen und akzeptieren zwei Datentypen. *Dataflows* enthalten Tupel, die in Pipeline-Form übermittelt werden. *Tables* sind eine spezielle Form von Dataflows, bei der der gesamte Input in einem Iterationsschritt zur Verfügung steht. Dadurch wird den Operatoren u.a. ein zufälliger Zugriff (*random-access*) ermöglicht. Die Berechnungen werden als Funktionen, die auf *Primitive* abgebildet werden, in Prefix-Notation ausgedrückt [72].

Primitive

Primitive führen die Operationen auf den *eigentlichen* Daten aus. Jedes Primtiv ist auf eine bestimmte Aufgabe mit einer speziellen Typenkombination spezialisiert [72].

Listing 2.3: Beispiel: Hinzufügen eines Integer-Vectors zu einer Konstanten [72]

```
1: int map_add_sint_col_sint_val(int n, sint *result,
2:                               sint *param1, sint *param2) {
3:     for (int i = 0; i < n; i++)
4:         result[i] = param1[i] + *param2;
5:     return n;
6: }
```

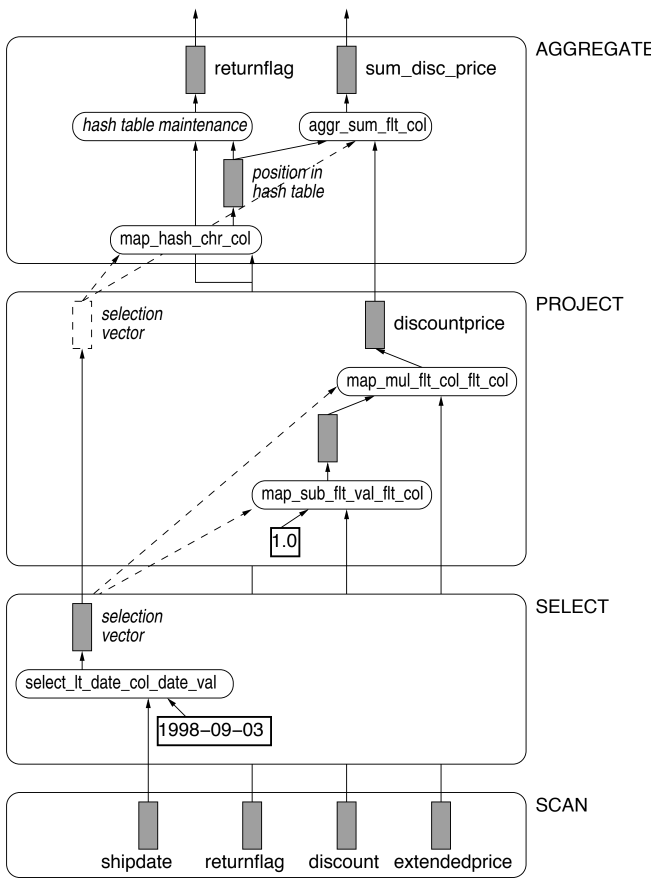


Abbildung 2.5: Ausführungsschema [72]

Eine Einteilung nach Funktionalität der Primtive ergibt drei Gruppen [72].

- **map-Primitive:** geben einen Wert für jedes Input-Tupel zurück
- **select-Primitive:** erzeugen einen zum Prädikat passenden selection-vector
- **aggr-Primitive:** legt die Art der Primitive fest, die für die Operation nötig sind

2.1.4 Star Schema Benchmark

(Majuran Rajakanthan)

Vectorwise wird in diesem Abschnitt mit Hilfe des *Star Schema Benchmarks (SSBM)* getestet. SSBM basiert auf dem TPC-H Benchmark und dient für *Data Warehouse* und *OLAP*-Anwendungen. Hier werden mehrere große Tabellen aus dem TPC-H Benchmark zu einer Faktentabelle zusammengefasst. SSBM hat eine *lineorder* Faktentabelle und vier Dimensionstabellen: customer, supplier, part, date (Abb. 2.6).

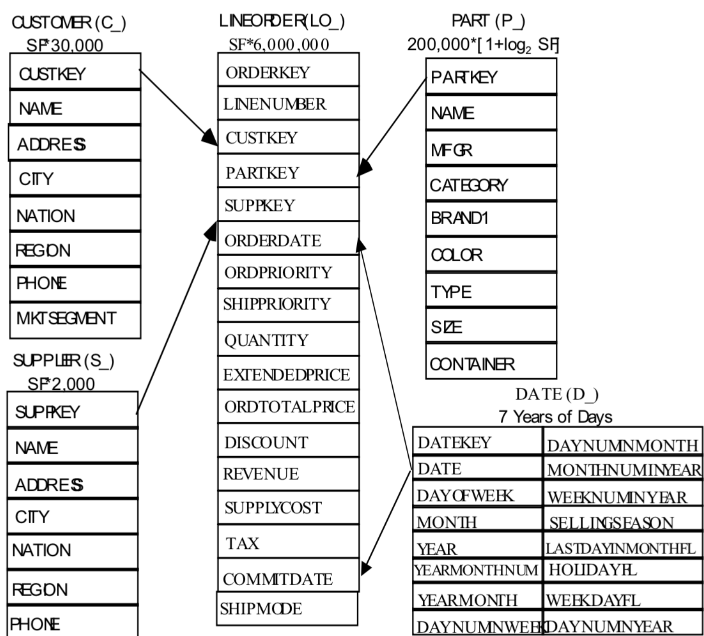


Abbildung 2.6: Datenschema des Star Schema Benchmarks [53].

Dafür wurde folgende Testumgebung verwendet:

DBMS:	Actian Vector 3.5
OS:	Ubuntu 12.04.5 LTS 64-bit (Linux Kernel 3.13.0)
Hardware:	Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz (64GB RAM)

Die Erzeugung der Daten für den Benchmark geschieht mit dem SSB Data Generator [60]. Für die Datenerzeugung werden die *Scale Faktoren (SF)* 1, 5,

10 und 15 verwendet. Nach dem Laden der Tabellen in die Datenbank, werden die 13 Queries des SSBM ausgeführt. Im Folgenden sind die Ergebnisse des Benchmarks tabellarisch (Tab. 2.1) und als Diagramm (Abb. 2.8) dargestellt.

Ausführungszeit der Queries							
SF	**Q1.1**	**Q1.2**	**Q1.3**	**Q2.1**	**Q2.2**	**Q2.3**	**Q3.1**
1	0,13	0,12	0,12	0,13	0,11	0,13	0,17
5	0,22	0,16	0,17	0,22	0,18	0,17	0,59
10	0,30	0,17	0,13	0,39	0,26	0,25	1,18
15	0,35	0,20	0,18	0,37	0,32	0,39	1,40

SF	**Q3.2**	**Q3.3**	**Q3.4**	**Q4.1**	**Q4.2**	**Q4.3**	**Total**
1	0,13	0,13	0,13	0,15	0,14	0,14	0,71
5	0,42	0,30	0,16	0,31	0,27	0,24	2,44
10	0,65	0,41	0,21	0,54	0,42	0,36	4,28
15	0,89	0,60	0,28	0,87	0,65	0,41	5,89

Tabelle 2.1: Ausführungszeit der SSBM-Queries in Sekunden

2.2 MonetDB/Ocelot

Nach einer kurzen Einführung, welche die historischen Aspekte beleuchtet, wird auf die Features und den Aufbau von MonetDB eingegangen. Danach wird eine kurze Übersicht über Ocelot gegeben, eine Erweiterung, welche GPU Unterstützung für MonetDB implementiert. Abgeschlossen wird die Sektion mit einem Einblick über die Performance dieses Columnstore-Systems und eines kurzen Querbezugs zu SliceDB.

Author: Maximilian Berens

2.2.1 Entwicklung und Geschichte von MonetDB *(Andreas Blume)*

MonetDB [62] ist ein hauptspeicherbasiertes Datenbankmanagementsystem (DBMS), dass am Centrum voor Wiskunde en Informatica (CWI) in Amsterdam (Niederlande) entstanden ist und heute als Open-Source-Projekt weiter gepflegt wird.
Die Anfänge des „Column Store Pioneers“ gehen in das Jahr 1993 zurück. In dieser Zeit war Hauptspeicher (RAM) teuer und eine knappe Ressource. Dennoch war der RAM (wie heute auch noch) deutlich schneller als eine Festplatte und sollte daher die Grundlage für ein schnelleres DBMS darstellen.
Nach über 10 Jahren interner Entwicklung wurde 2004 MonetDB 4 unter einer Open-Source Lizenz veröffentlicht. In den folgenden Jahren wurde das DBMS durch viele Erweiterungen verbessert und die Codegröße nahm immer weiter zu. Deshalb wurde zwischen 2011 und 2013 der komplette Kernel von MonetDB neu geschrieben und das Ergebnis als MonetDB 5 veröffentlicht.
Heutzutage wird MonetDB in vielen High-Performance-Anwendungen wie zum Beispiel DataMining und Online Analytical Processing (OLAP) eingesetzt. Um

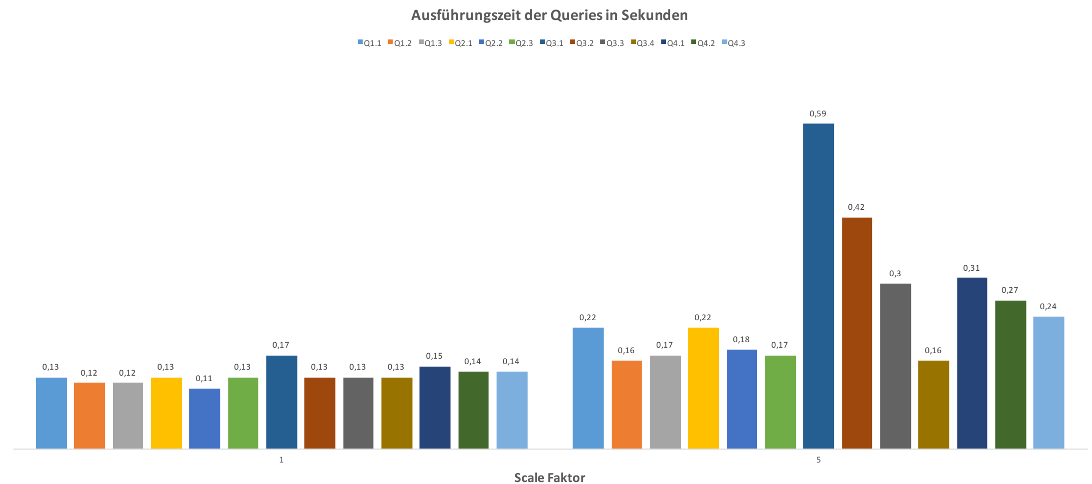


Abbildung 2.7: SSBM Query Ausführungszeit für SF 1 und 5

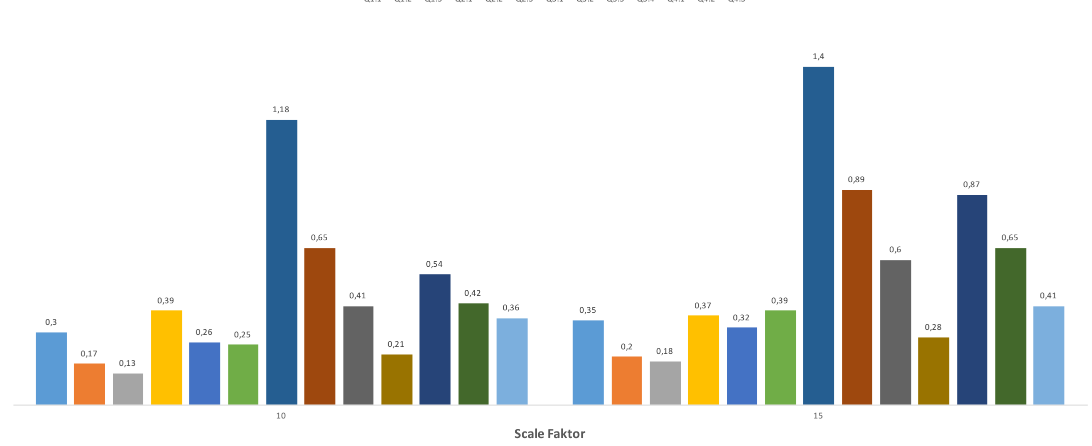


Abbildung 2.8: SSBM Query Ausführungszeit für SF 10 und 15

die Verbreitung, speziell im kommerzielen Bereich, weiter voranzutreiben wurde 2013 die Firma „MonetDB Solutions“ [63] gegründet, die einen kompletten Support-Service für MonetDB anbietet.

2.2.2 Features und Aufbau

In dieser Sektion wird detaillierter auf die Features und den Aufbau von MonetDB eingegangen.

Architektur und Column Store *(Maximilian Berens)*

MonetDB verwendet zur Darstellung von relationalen Daten ein so genanntes Decomposed Storage Model (DSM), welches die Daten vertikal fragmentiert abspeichert („Column Store“). Die Spalten werden dazu in zweispaltige Tabellen der Form *<oid, value>* bzw *<Head, Tail>* abgelegt, Binary Association Tables (BAT's) genannt. Auch Zwischen und Endergebnisse sind lediglich verknüpfte BAT‘s. Auf diesen Tabellen wird eine einfache Algebra definiert, die BAT Algebra. MonetDB kann als eine virtuelle Maschine für diese Algebra betrachtet werden, vergl. 2.9.
Datentypen mit variabler Länge werden in zwei Arrays geteilt, eines hält einen Offset, das Andere die aneinander gehängten, restlichen Daten. Persistenz wird mittels *memory mapped files* ermöglicht. Um Zugriffe und Speicher zu optimieren, kann der *Head* einer BAT weg gelassen werden, da es sich dabei in der Regel lediglich um eine fortlaufende Nummer handelt. Lookups mittels *oid*'s reduzieren sich so auf einen schnellen Indexdurchlauf. Dadurch werden durch die MMU (Memory Management Unit, eine Komponente der CPU) implementierte Hardwaremechanismen ausgenutzt, die einen Zugriff in konstanter Zeit (d.h. $\mathcal{O}(1)$) erlauben.
Viele DBMSe verwenden ein Vulcano-Iterator Modell, in dem jeder Operator iterativ implementiert und Daten-Tuple wie durch eine Pipeline „durch gereicht“ werden (*Tuple-At-A-Time* Modell). Dies kann zum Einen dazu führen, dass es mit vielen aufeinander folgenden Instruktionen auch viele Misses im Instruction-Cache gibt. Außerdem stellen die für das Iterator Modell notwendigen Funktionsaufrufen (wie zum Beispiel *next()*) einen nicht unerheblichen Interpretations-Overhead dar. In MonetDB hat man sich daher entschieden, das *Operator-At-A-Time* Modell um zu setzen. Da dies bedeutet, dass ein Ergebnis nach der Berechnung eines Operators vollständig materialisiert wird, muss genug Hauptspeicher zur Verfügung stehen. Man bedenke, das (Zwischen-)Ergebnisse insbesondere durch Joins schnell groß werden können. Ein in diesem Falle notwendiges Auslagern von Teilergebnissen auf die Festplatte verlangsamt die Bearbeitung der Anfrage signifikant. In [34] wird dies als Tradeoff zwischen Cacheeffizienz und Speicherbedarf (RAM) bezeichnet. Hier wird ebenfalls darauf hingewiesen, dass MonetDB starken Gebrauch von *structure sharing* macht und demnach häufig auf das Erstellen von Kopien verzichten kann. Dies drückt den Speicherverbrauch und auch den Performanceoverhead, welcher durch die häufige Materialisierung zu Stande kommt.
Komplexe Operatoren werden durch die MAL (MonetDB Assembly Language) umgesetzt, welche aus einfachen Instruktionen besteht. Diese Operationen bieten keine Möglichkeit komplexe Instruktionen als Parameter zu übergeben, sie führen lediglich eine Operation auf ganzen Spalten aus („Bulkprocessing“).

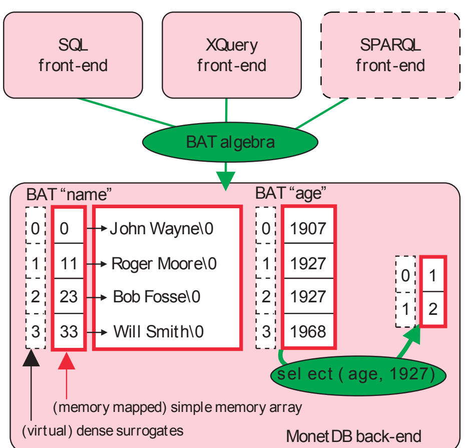


Abbildung 2.9: Struktur von MonetDB [44]

MonetDB benötigt keine Interpretationsengine, da die Anweisungen auf einfache Array Operationen abgebildet werden. Eine Konsequenz daraus ist, dass diese mit „engen“ For-Schleifen implementiert werden können, welche vom Compiler bzw. der CPU selbst optimierbar sind und durch Instruktionslokalität weniger Cache-Misses hervorrufen können.
Dieser Abschnitt wurde aus dem Architekturkapitel in [44] entnommen.

MonetDB Assembly Language *(Maximilian Berens)*

Die *MonetDB Assembly Language (MAL)* stellt ein einfaches Interface für den Zugriff auf die Kernelfunktionen dar. Aus der MAL lassen sich Anfragepläne generieren, welche optimiert und ausgegeben werden können. Ein Anfrageplan in der MAL kann zum Beispiel so aussehen:

```
function user.s1 2(A0:date,A1:date,A2:int,A3:str):void;
  X5 := sql.bind("sys","lineitem","l returnflag",0);
  X11 := algebra.uselect(X5,A3);
  X14 := algebra.markT(X11,0@0);
  X15 := bat.reverse(X14);
  X16 := sql.bindIdxbat("sys","lineitem","l orderkey fkey");
  X18 := algebra.join(X15,X16);
  X19 := sql.bind("sys","orders","o orderdate",0);
  X25 := mtime.addmonths(A1,A2);
  X26 := algebra.select(X19,A0,X25,true,false);
  X30 := algebra.markT(X26,0@0);
  X31 := bat.reverse(X30);
  X32 := sql.bind("sys","orders","o orderkey",0);
  X34 := bat.mirror(X32);
  X35 := algebra.join(X31,X34);
  X36 := bat.reverse(X35);
  X37 := algebra.join(X18,X36);
  X38 := bat.reverse(X37);
  X40 := algebra.markT(X38,0@0);
```

```
X41 := bat.reverse(X40);
X45 := algebra.join(X31,X32);
X46 := algebra.join(X41,X45);
X49 := algebra.selectNotNil(X46);
X50 := bat.reverse(X49);
X51 := algebra.kunique(X50);
X52 := bat.reverse(X51);
X53 := aggr.count(X52);
sql.exportValue(1,"sys.orders","L1","wrd",32,0,6,X53);
end s1 2;
```

In SQL formuliert entspricht dies der folgenden Anfrage:

```
select count(distinct o_orderkey)
from orders, lineitem
where l_orderkey = o_orderkey
    and o_orderdate >= date '1996-07-01'
    and o_orderdate < date '1996-07-01'
    + interval '3' month
    and l_returnflag = 'R';
```

Dieses Beispiel wurde aus [34] entnommen und stellt eine Anfrage des TPC-H Benchmarks dar. Zu erkennen ist, dass ein MAL-Anfrageplan linear ist und dementsprechend schrittweise von oben nach unten abgearbeitet wird (vergl. Operator-At-A-Time Modell). Eine vollständige Übersicht über die Syntax der MAL ist hier zu finden [1].

Cache-bewusste Algorithmen *(Andreas Blume)*

Cache-bewusste Algorithmen sind ein wichtiger Bestandteil von MonetDB. Sie werden benötigt, um die Memory Wall zu überwinden und so die komplette Rechenleistung der heutigen Computersysteme vollständig nutzen zu können. Die Grundlage dieser Algorithmen bilden der effiziente Einsatz von Caches und ein optimales Zugriffsmuster auf den Speicher. Beide Dinge werden beispielsweise beim Radix-Cluster Algorithmus (einem optimierten equi-Join Algorithmus) eingesetzt, um den teuersten Operator der relationalen Algebra, den Join von zwei Relationen L und R, zu beschleunigen.
Der Radix-Cluster Algorithmus [45] beginnt mit der Aufteilung der beiden Relation in jeweils H Cluster. Für die Clusterung werden die B kleinsten Bits vom Hashwert (= Integer) einer Spalte verwendet, um die Tupel einer Relation aufzuteilen. In jedem sequentiellen Schritt (engl. pass) P werden dazu B_P Bits (beginnend mit dem linkesten Bit) genutzt, um die einzelnen Tupel in Cluster aufzuteilen. So enstehen im ersten Schritt $H_1 = 2^{B_1}$ Cluster. Der nächste Schritt teilt nun jedes Cluster in $H_2 = 2^{B_2}$ Neue, so dass wir nach dem zweiten Schritt insgesamt $H = H_1 * H_2$ Cluster erhalten. Durch die weitere Durchführung der Prozedur kann die Clusteranzahl gesteigert und die Clustergröße gesenkt werden. Das Ziel sollte es dabei sein, dass wir die Clusteranzahl ($H_x = 2^{B_x}$) kleiner als die Anzahl an Cachelines und TLB-Einträge halten und dadurch sowohl TLB- als Cache-Misses komplett verhindern.

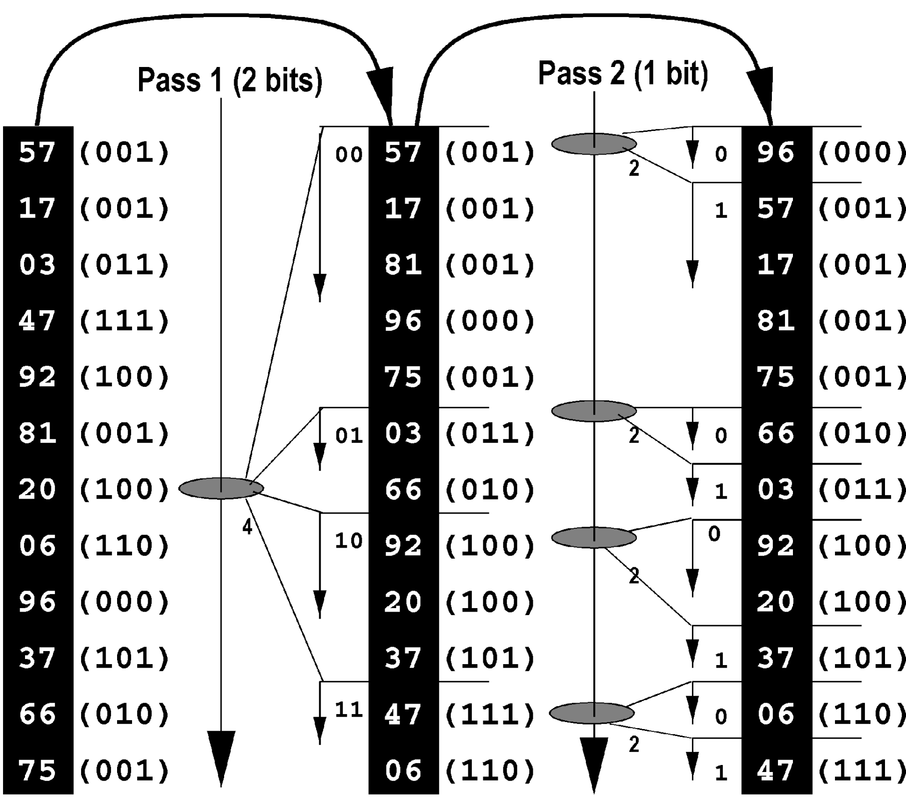


Abbildung 2.10: Radix-Cluster Algorithmus [45]

In Abbildung 2.10 ist das Beispiel einer zweistufigen Clusterung für eine Relation zu sehen. Die Clusterung wird hier mit Hilfe der drei kleinsten Bits von jeder Zahl (in Klammern hinter jeder Zahl angegeben) durchgeführt. So werden im ersten Schritt das vorvorletzte und das vorletzte Bit verwendet, um $H_1 = 2^2 = 4$ Cluster aus der Ausgangsrelation L zu erzeugen. Das Ergebnis dieses Schrittes ist in der Mitte zu sehen. Das *01*-Cluster enthält beispielsweise die Werte „3“ und „66“. Nun schließt sich der zweite Schritt an, indem jedes Cluster mit Hilfe des letzten Bits aufgeteilt wird. Da die „3“ und „66“ eine unterschiedliches letztes Bit besitzen, werden sie nun getrennt und gehören ab jetzt zu unterschiedlichen Clustern. Nach diesem Schritt wächst die Clusteranzahl auf $H_1 * H_2 = 8$ (mit $H_2 = 2^1 = 2$).

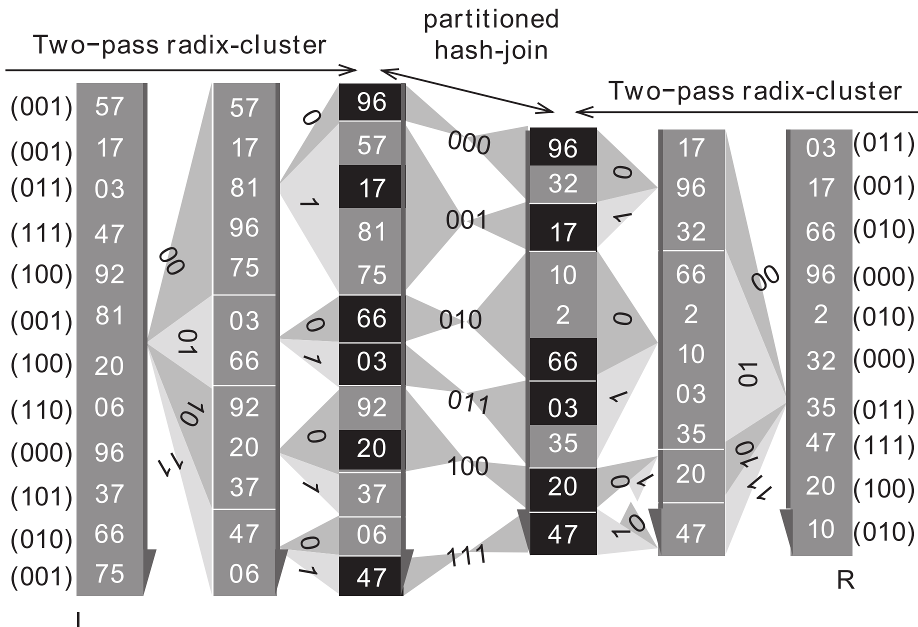


Abbildung 2.11: Radix-Cluster Algorithmus mit Partitioned Join [15]

Nachdem wir beide Relationen geclustert haben, kann nun der eigentlich Join durchgeführt werden [15]. Dazu müssen die Cluster der beiden Relationen L und R, die die gleiche Eigenschaft besitzen (im Beispiel: die letzten 3 Bits aller Tupel eines Clusters sind gleich), gejoint werden. Insgesamt müssen somit maximal H (parallele) Teiljoins durchgeführt werden, um das komplette Join-Ergebnis zu erhalten (vgl. Abbildung 2.11). In dieser Abbildung ist zunächst zu sehen, dass die Relationen L mit dem oben beschriebenen Verfahren in $H_L = 8$ Cluster aufgeteilt wurde. Die Relation R wurde mit dem gleichen Algorithmus aufgeteilt und es entstehen hier $H_R = 6$ Clustern. Da $H_R < H_L$ müssen in diesem Beispiel nur 6 Teiljoins durchgeführt werden. Dieser Vorgang ist in der Mitte zu sehen, wobei die weißen Zahlen auf schwarzem Hintergrund das Ergebnis des kompletten Joins darstellen.

Kostenmodell für Speicherzugriffe *(Maximilian Berens)*

Um Verfahren wie den Radix-Cluster-Algorithmus effizient um zu setzen, müssen die Parameter, wie die Clustergröße möglichst geschickt gewählt werden. Zu diesem Zweck wird versucht, die Speicherzugriffskosten für einen Algorithmus zu bewerten. Die Summe über die Anzahl der Cachemisses M_i und die damit verbundene Latenz l_i modelliert hierbei die Kosten, wobei jedes Cachelevel i getrennt voneinander, wenn auch gleich wie die Anderen behandelt wird:

$$T_{Mem} = \sum_{i=1}^{N} (M_i^s \cdot l_i^s + M_i^r \cdot l_i^r)$$

Mit r und s unterscheidet man die Kosten für *random* bzw *sequential* Zugriffe. Die Schwierigkeit, eine möglichst akkurate Kostenfunktion zu erhalten, besteht darin, die Anzahl und Art der Cachemisses möglichst genau vorher zu sagen. Hierfür abstrahiert man die Datenstrukturen als *data regions* und setzt komplexe Zugriffsmuster aus einfachen *basic access patterns* zusammen. *Data regions* werden über die Anzahl und Größe ihrer Elemente definiert, an *basic access patterns* existieren *single/repitative sequential/random traversels* sowie *interleaved* und *random access*. Für eine detaillierter Erläuterung wird auf [15] sowie die ursprünglichen Arbeiten von Manegold [42], [43] verwiesen.

2.2.3 Ocelot *(Andreas Blume)*

Bei Ocelot handelt es sich um eine leichtgewichtige Erweiterung für das Datenbankmanagementsystem MonetDB. Durch die Erweiterung kann MonetDB das komplette Potential von modernen, parallelen Prozessorarchitekturen (z.B. Multicore CPUs und GPUs mit gemeinsamen Speicher) ausnutzen. Dazu werden alle Operatoren (Selektion, Projektion, usw.) mit Hilfe von OpenCL (Open Computing Language) implementiert und können so ohne manuelle Anpassung auf den unterschiedlichsten Computerarchitekturen ausgeführt werden.

OpenCL [47, 57] ist das erste Framework, dass speziell für die Programmierung von heterogenen Systemen mit einem gemeinsamen Speicher (= Computer, die aus CPUs, GPUs und anderen Prozessoren bestehen können) entwickelt wird. Die Erstellung und Verwaltung erfolgt und dem Dach der Khronos Group[1], der über 100 Mitglieder angehören. Unter anderem die großen IT-Firmen: AMD, Intel, Nvidia, Google, Samsung Electronics und Apple. Das Framework basiert auf der Programmiersprache C und unterstützt sowohl die Daten- als auch die Taskparallelität.
Jedes OpenCL-Programm besteht aus einem Host, der für die die Interaktion mit dem Benutzer und die I/O verantwortlich ist, und mehreren OpenCL-Devices (auch „compute devices“ genannt), die die eigentlichen Berechnungen durchführen. Daher werden für eine OpenCL-Applikation ein Host-Programm und eine endliche Menge von Kernels in der Programmiersprache OpenCL-C benötigt. Jeder Kernel wird vom Host aufgerufen und schließlich von den Devices berechnet.
In der Abbildung 2.12 ist die Integration von Ocelot in MonetDB zu sehen. Ocelot besteht aus insgesamt 4 Teilen: [31]

[1] siehe: `https://www.khronos.org/opencl`

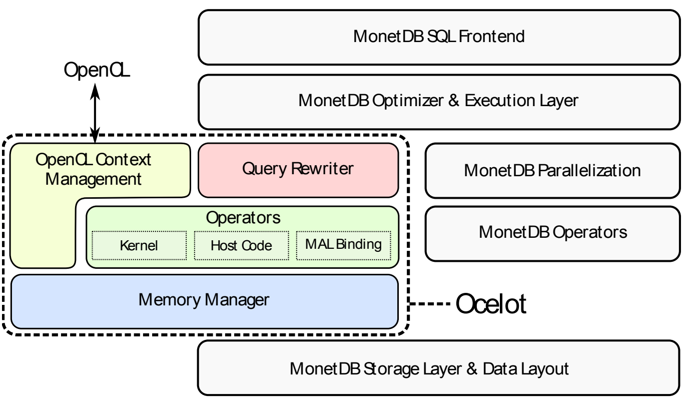


Abbildung 2.12: Architektur von MonetDB mit der Erweiterung Ocelot [31]

- Der **Memory Manager** verwaltet den Speicher vom jeweiligen Device. Er sorgt also für die Umwandlung der Binary Association Tables (BATs) in *cl_mem* Buffer (nur mit diesem können OpenCL Devices rechnen) und kümmert sich um den Transfer der Buffer vom Host zum Device und umgekehrt.

- Die Operatoren (engl. **Operators**) sind zentraler Bestandteil von Ocelot. Jeder Operator ist mit OpenCL umgesetzt und ersetzt den jeweiligen MonetDB-Operator. Folgende Operatoren wurden mit Hilfe von OpenCL-Kerneln implementiert: Selektion, Projektion, Sortierung, Hashing, Join, Group By und Aggregation.

- Um die Ocelot-Operatoren in MonetDB nutzen zu können, müssen die von MonetDB generierten Query-Pläne durch den **Query Rewriter** angepasst werden. Dabei werden die Operatoren von MonetDB durch die entsprechende Ocelot-Implementierung ersetzt.

- Für die Ausführung der in OpenCL geschriebenen Ocelot-Operatoren wird der **OpenCL Context Manager** benötigt. Er initialisiert zunächst die OpenCL-Laufzeitumgebung und steuert die Compilierung der OpenCL-Kernel. Dazu kommt die Verwaltung der Kernelplanung und -ausführung sowie der Zugriff auf die wichtigen OpenCL-Datenstrukturen.

2.2.4 Die Performance von MonetDB *(Maximilian Berens)*

MonetDB's Features begünstigen anfragelastige Workloads, zu sehen ist dies zum Beispiel in [61]. Hier werde drei DBMSe (MonetDB, PostgreSQL und eine Erweiterung von PostgreSQL namens cstore_fdw) auf dem TPC-H Benchmark verglichen. Die beiden Column-store Systeme (MonetDB und cstore_fdw) können sich dabei deutlich von ihrem zeilenorientierten Konkurrenten abgrenzen (cold, siehe 2.13). Jedoch kann MonetDB sich dabei in den meisten Anfragen auch gegen cstore_fdw durchsetzen, insbesondere mit 'warmen' Caches. Die Ausführungszeit liegt hier im Bereich von teils unter 1 Sekunde (siehe 2.14). Eine effiziente Nutzung der Caches könnte hierbei eine Erklärung sein. Allerdings ist auch zu erwähnen, dass die Testmaschine mit 16 GB genug Ram für die Querys hatte,

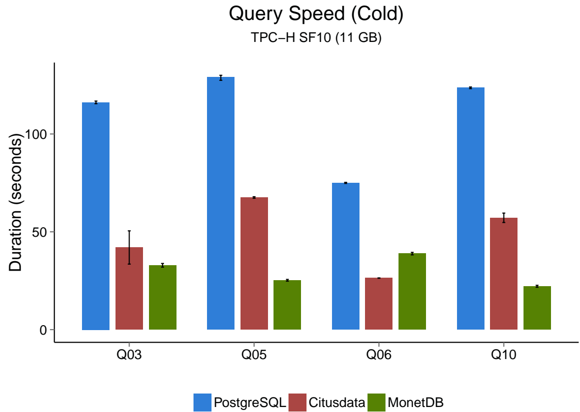


Abbildung 2.13: TPC-H Benchmark (cold caches), Scalefactor 10 [61]

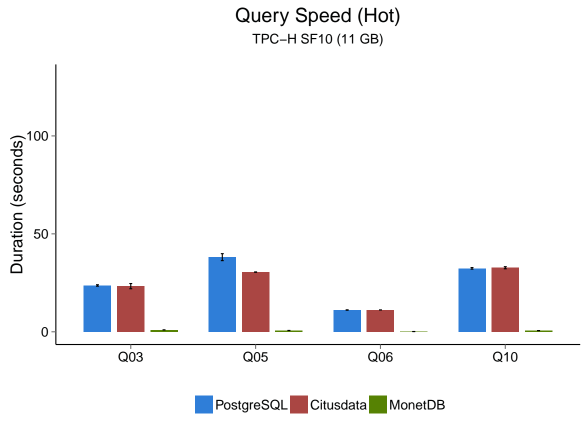


Abbildung 2.14: TPC-H Benchmark (hot caches), Scalefactor 10 [61]

größere Anfragen könnten zu enormen Performanceeinbußen führen, da zu große Zwischenergebnisse auf die Festplatte ausgelagert werden müssten.

2.2.5 SliceDB im Vergleich mit MonetDB *(Maximilian Berens)*

Die Ansätze, welche in SliceDB verfolgt werden, unterscheiden sich von dem Vorgehen MonetDBs und sind eher mit denen von Vectorwise zu vergleichen: SliceDB implementiert Vector-at-a-time im herkömmlichen Vulcano Iterator Model, MonetDB verwendet Operator-at-a-time. Außerdem wurden auch viele nutzbare Verbesserungen (zum Beispiel Cache-bewusste Algorithmen) aus MonetDB in SliceDB noch nicht umgesetzt, daher sind direkte Bezüge schwer zu ziehen. Ein Kostenmodell zur Queryplanoptimierung ist allerdings angedacht und diverse Vorteile des Operator-at-a-time werden durch die Vektorisierung erreicht.

2.3 HyPer *Ersin Özdemir & Eric Fiege*

HyPer [36] ist eine für moderne Hardware ausgelegte Hauptspeicherdatenbank, die an der TU München entwickelt wurde. Sie hat das Ziel, sowohl *Online Transactional Processing (OLTP)* Anfragen, als auch *Online Analytical Processing (OLAP)* Anfragen beantworten zu können. Dies hat den Vorteil, dass die Daten für die Analyse immer auf dem aktuellen Stand sind und nicht jeweils ein System für den OLTP und OLAP Workload gewartet werden muss. HyPer setzt die vollen ACID Eingeschaften für OLTP Anfragen um. Außerdem können mehrere OLAP Anfragen parallel laufen gelassen werden. Die Tabellen werden Vector-basiert und somit auf physischer Ebene als Columnstore abgelegt, wobei bei Bedarf auf Rowstore umgeschaltet werden kann. Anfragen können in SQL oder in einer PL/SQL ähnlichen Skriptsprache bearbeitet werden.

2.3.1 Überblick

Zunächst soll der Aufbau und die Funktionsweise von HyPer skizziert werden. Abbildung 2.15 zeigt einen Überblick der Funktionsweise von HyPer.

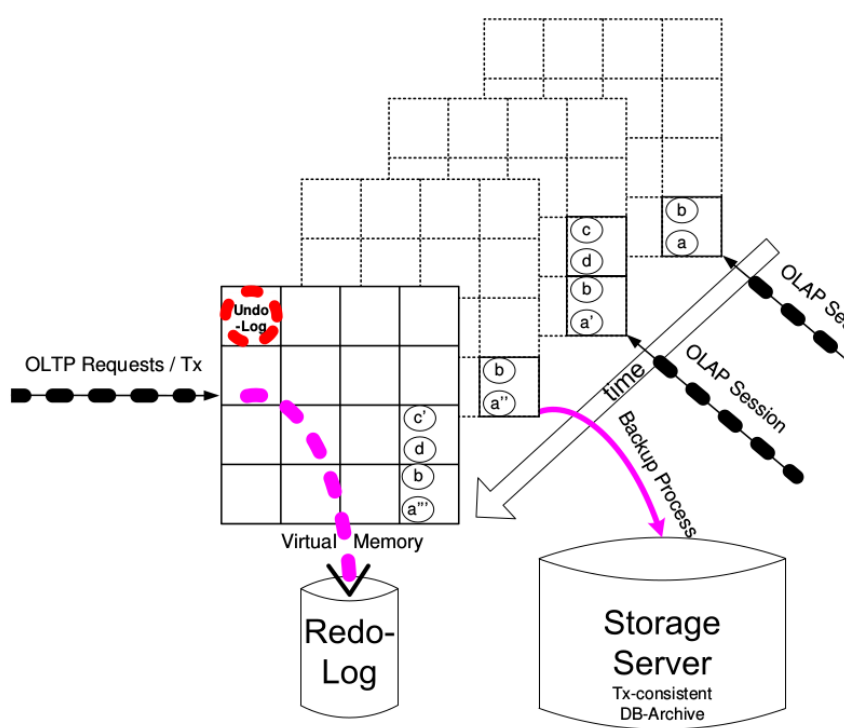


Abbildung 2.15: Übersicht zu HyPer.[36]

Es werden folgende **Annahmen** getroffen:

- OLTP und OLAP Anfragen behindern sich nicht gegenseitig bei zeitgleicher Verwendung.

- Sämtliche Daten passen in den Hauptspeicher. Transaktionen blockieren daher nicht wegen I/O.

- Nicht blockierende Transaktions-Isolation soll durch copy-on-write im virtuellen Speicher gewährleistet sein.

- Die CPU-Kosten müssen minimal gehalten werden.

OLTP Anfragen werden in HyPer sequentiell abgearbeitet. Dies hat den Vorteil, dass kein Latching und Locking benötigt wird. OLAP Anfragen werden hingegen nicht sequentiell abgearbeitet, da sie in der Zeit, in der sie ausgeführt werden, die OLTP Anfragen blockieren würden. Aus diesem Grund können zur Ausführung von OLAP Anfragen neue *OLAP Sessions* erstellt werden. Dazu wird ein Kind-Prozess mittels *fork()* erstellt. Dabei wird eine Hardware-unterstützte (virtuelle) Kopie des Speicherbereichs angelegt (auch *Snapshot* genannt), auf der der Prozess arbeiten kann. Beim Erstellen von neuen OLAP Sessions muss beachtet werden, dass der *fork()* zwischen zwei Anfragen geschieht, da sonst an dieser Stelle mit einem inkonsistenten Zustand weitergearbeitet werden könnte.

2.3.2 OLTP-Anfragen

OLTP Anfragen können auch parallel ausgeführt werden. Die parallele Ausführung kann mit zwei Methoden umgesetzt werden:

- Bei read-only OLTP Anfragen wird in den parallelen Modus umgeschaltet. Sobald eine Anfrage ein Update enthält, wird wieder in den sequentiellen Modus umgeschaltet.

- Die Daten werden in Partitionen und einen *Shared-Memory* unterteilt. Für jede Partition wird ein Thread erstellt, der auf seinen eigenen Daten arbeiten kann. Solange die Threads auf ihren eigenen Partitionen arbeiten, können alle parallel laufen. Änderungen an dem Shared-Memory oder ein partitionsübergreifendes Lesen erfordern Synchronisation. Transaktionen, die mehrere Partitionen nutzen, fordern dann den alleinigen Zugriff auf alle Daten an. Die Transaktionen werden daher in zwei Gruppen unterteilt:

  - *partition-constrained*: Sind die Transaktionen, die nur auf ihrer eigenen Partition arbeiten. Bei diesen Transaktionen kann eine Anfrage pro Partition parallel abgearbeitet werden.
  - *partition-crossing*: Sind die Transaktionen, die auf dem Shared-Memory oder auf unterschiedlichen Partitionen arbeiten. Diese Transaktionen müssen warten bis alle anderen Transaktionen, die gerade laufen, terminiert sind. Danach haben sie den alleinigen Zugriff auf alle Daten.

2.3.3 OLAP-Anfragen

OLAP Anfragen können ebenfalls parallel ausgeführt werden. Dabei sollte beachtet werden, dass der OLTP Prozess einen bestimmten Prozessor zugewiesen bekommt, um weiterhin die schnelle Abarbeitung von OLTP Anfragen zu gewährleisten. Die parallele Ausführung kann mit zwei Methoden umgesetzt werden:

- Da OLAP Anfragen read-only sind, können die einzelnen Anfragen einfach in mehreren Threads ausgeführt werden, die sich einen gemeinsamen Adressraum teilen.
- Es können Kind-Prozesse mit *fork()* und somit mehrere OLAP Sessions erstellt werden.

Beim Erstellen eines Snapshots wird in Wirklichkeit nur das Nutzen eines gemeinsamen Adressraums durch die Hardware ermöglicht (Prozessoren mit *hardware transactional memory (HTM)*). Die Daten werden dabei nicht kopiert. Physikalisch sind die Kopien also gleich, virtuell jedoch unterschiedlich. Der virtuelle Speicher ist dabei auf die Hauptspeicher-Größe angepasst, um das Austauschen von Seiten durch das Betriebssystem zu vermeiden. Da OLAP Sessions parallel zu OLTP Anfragen laufen, kann es passieren, dass eine OLTP Anfrage mit einem Update ausgeführt wird und Daten verändert werden. Für die OLAP Sessions ist dies allerdings nicht erwünscht, da der Stand der Daten mit dem Zeitpunkt des Erstellens nicht mehr übereinstimmen würde. Um dies zu vermeiden, werden sämtliche Änderungen in einem sogenannten Delta abgelegt, sodass laufende OLAP Sessions keine Änderungen an den Daten erfahren. Wird also eine gemeinsame Seite bearbeitet, wird erst dann eine echte Kopie der Seite erstellt (*page shadowing*). Dadurch wird eine konstengünstige Transaktions-Isolation erreicht. Das Delta bleibt solange bestehen, bis alle laufenden OLAP Sessions abgearbeitet und anschließend terminiert sind. Erst, wenn keine OLAP Sessions mehr laufen, werden die Änderungen aus dem Delta in die Datenbank übernommen.
Es kann der Fall eintreten, dass zunächst eine OLAP Session erstellt wird, anschließend eine OLTP Anfrage eine Änderung an den Daten vornimmt und zuletzt ein weiterer Snapshot erstellt wird. Die letztere der beiden Sessions soll jedoch den aktuelleren Stand der Daten repräsentieren. Dies wird ermöglicht, indem wie bei den anderen Sessions der Adressraum geteilt wird, die Änderungen aus dem Delta jedoch mit übernommen werden. Weitere nachfolgende Veränderungen an den Daten müssen erneut in einem Delta abgelegt werden. Somit steigt der zusätzliche Speicherbedarf mit der Anzahl der Änderungen, die eintreffen, während OLAP Sessions aktiv sind.

In Abbildung 2.15 wird beispielhaft eine zeitliche Abfolge von erstellten Snapshots und eingetroffenen Änderungen dargestellt. Der hinterste und somit erste Snapshot zeigt den ältesten Stand der Datenbank. Bevor der zweite Snapshot erstellt wurde, gab es eine Änderung an a, sodass eine Kopie der Seite erstellt worden ist. Außerdem wurden c und d in einer anderen Seite eingefügt. Der zweite Snapshot sieht demnach aus wie der erste, beinhaltet jedoch die Änderungen aus dem Delta, welches die besagten zwei Kopien der Seiten beinhaltet. Bevor der dritte und letzte Snapshot erstellt wurde, gab es eine weitere Änderung an dem ursprünglichen a, hier also a', sodass eine Kopie der Seite mit a' und b inklusive der Änderungen im Delta abgelegt und für den letzten Snapshot verwendet wurde. Im Gegensatz zu dem, was man aus der Grafik schließen könnte, wurde die Seite mit c und d zwischenzeitlich nicht entfernt, sondern lediglich nicht verändert worden. Der letzte Snapshot beinhaltet also sämtliche Seiten aus dem zweiten Snapshot inklusive den Änderungen auf der besagten Seite. Nach dem letzten Snapshot wurden beide abgebildeten Seiten nochmals verändert, jedoch kein weiterer Snapshot mehr erstellt.

2.3.4 Sicherung & Wiederherstellung

Eine Datenbank kann aus den unterschiedlichsten Gründen ausfallen und muss anschließend neugestartet werden. Um die Daten persistent zu Speichern und nach einem Ausfall wiederherzustellen, können die Snapshots der OLAP Sessions genutzt werden. Die Daten eines Backups in HyPer sind stets konsistent. Das liegt daran, dass ein *fork()* stets nur zwischen zwei OLTP Transaktionen stattfindet und diese die ACID Eigenschaften aufweisen. Dabei steht ACID für:

- **Atomicity**: Bei einer fehlerhaften Transaktion wird der vorherige Stand wiederhergestellt.
- **Consistency**: Der Stand vor und nach einer Transaktion muss in sich stimmig sein, sonst wird die Transaktion abgebrochen.
- **Isolation**: Jede Transaktion muss unabhängig zu den zur gleichen Zeit ausgeführten Transaktionen sein.
- **Durability**: Jede erfolgreiche Transaktion muss bei System- oder Netzwerkfehlern unangetastet bleiben.

Erfolgreiche Transaktionen werden im Redo-Log protokolliert, welches sich auf einem nicht-flüchtigem Speichermedium befindet. Wichtig hierbei ist die korrekte Reihenfolge von partitionsgebundenen und partitionsübergreifenden Transaktionen. Einträge werden gruppenweise auf das Medium geschrieben (*group-commit*), um die Performanz zu steigern, da das separate Schreiben eines jeden Eintrags (*write-ahead-logging*) zum Flaschenhals werden kann.
Um bei einer fehlerhaften Transaktion den vorherigen Stand wiederherzustellen, existiert das *Undo-Log*. Dieses befindet sich ebenfalls im Hauptspeicher und wird lediglich für aktive Transaktionen benötigt, da nur Transaktions-konsistente Zustände gesichert werden. Dabei kann das Undo-Log verwendet werden, um einen solchen herzustellen.
Nach einem Ausfall der Datenbank wird die letzte Sicherung geladen und anschließend das Redo-Log angewendet. Dadurch wird stets der aktuellste, konsistente Stand der Datenbank wiederhergestellt. Bei einer stündlichen Sicherung einer 100 GB kann die Wiederherstellung in einer annehmbaren Zeit durchgeführt werden.

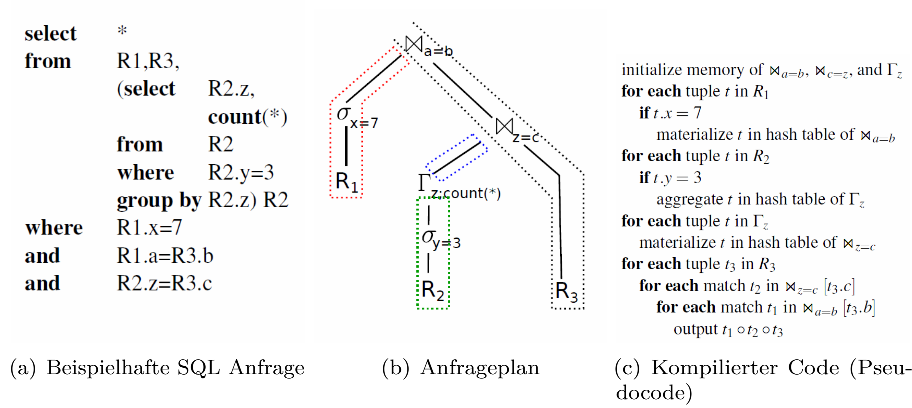
2.3.5 Anfragen als Maschinencode

In modernen Hauptspeicherdatenbanken wird der Zugriff auf externe Speichermedien größtenteils vermieden, sodass der Zugriff auf den Hauptspeicher der neue Flaschenhals ist und somit optimiert werden sollte. Die Idee von HyPer ist es, die Performanz durch Ausnutzung von Lokalitäten zu steigern, indem die Anfragen zunächst in Maschinencode kompiliert werden.

Traditionelle Datenbanken nutzen üblicherweise das *Volcano-Iterator-Model*, um Anfragen zu bearbeiten. Dabei werden Tupel von Operator zu Operator weitergereicht und ausgewertet. In Zeiten, wo die Performanz dieser Datenbanken durch I/O Zugriffe bestimmt wurde, mag dies eine gute Idee gewesen sein. Heutzutage lassen sich einige Probleme erkennen. Zum einen führt das Model zu überschüssigen **Funktionsaufrufen**, zum anderen werden **Datenlokalitäten**

nicht ausgenutzt.

HyPer verfolgt an dieser Stelle einen **datenzentrischen Ansatz** [49]. Eine Anfrage wird wie auch in anderen Datenbanken zunächst geparst, anschließend in einen Algebraausdruck umgeformt und zuletzt optimiert. Dann wird der Algebraausdruck jedoch *nicht* interpretiert, sondern in ein iteratives Programm *kompiliert*. Um Maschinencode zu generieren, muss zunächst Vorarbeit geleistet werden. Dazu werden aus dem Anfrageplan sogenannte *Pipelines* gebildet, die dann in Code Fragmente umgewandelt werden sollen. Abbildung 2.16 (b) zeigt eine beispielhafte Aufteilung eines Anfrageplans in vier Pipelines. Die Tupel werden pro Pipeline in die Register der CPU geladen und alle Operatoren aus der jeweiligen Pipeline auf die Tupel angewendet. Dadurch werden unnötige Tupel Materialisierungen vermieden und der Zugriff auf den Hauptspeicher reduziert. Im Gegensatz zum Volcano-Iterator-Model werden die Tupel also stets weitergereicht (*push*) und nicht vom vorherigen Operator angefordert (*pull*). Erst wenn ein *Pipeline-Breaker* erreicht ist, müssen die Tupel materialisiert werden. Ein Pipeline-Breaker ist ein *blockierender* Operator wie beispielsweise der Hash-Join oder das Sortieren, die sämtliche relevanten Daten zur Ausführung benötigen.



(a) Beispielhafte SQL Anfrage (b) Anfrageplan (c) Kompilierter Code (Pseudocode)

Abbildung 2.16: Generierung von Maschinencode in HyPer. [49]

Sind die Pipelines definiert, müssen diese noch in Code Fragmente umgewandelt werden. Dazu wird *zur Kompilierungszeit* das eingebundene *produce/consume-Interface* eines jeden Operators genutzt. Diese Methoden existieren *nicht* zur Laufzeit des Programms. Wichtig hierbei zu verstehen ist, dass Code verwendet wird, um anderen Code zu generieren. Dabei erzeugt *produce* Code, das die Resultat-Tupel eines Operators berechnet, währenddessen *consume* Code erzeugt, welches ein Tupel verarbeitet. Die Fragmente bestehen jeweils aus kompakten Schleifen, wodurch nochmals die Performanz gesteigert wird. Abbildung 2.17 zeigt eine simple Veranschaulichung des Interfaces, woraus der Code in Abbildung 2.16 (c) erzeugt werden kann.

Sind die Code Fragmente erst einmal erzeugt, können diese in Maschinencode umgewandelt werden. In HyPer kommt die **Low Level Virtual Machine**

```
scan.produce():
    print "for each tuple in relation"
    scan.parent.consume(attributes,scan)
σ.produce:
    σ.input.produce()
σ.consume(a,s):
    print "if "+σ.condition
    σ.parent.consume(attr,σ)

⋈.produce():
    ⋈.left.produce()
    ⋈.right.produce()
⋈.consume(a,s):
    if (s==⋈.left)
        print "materialize tuple in hash table"
    else
        print "for each match in hashtable[" +a.joinattr+"]"
        ⋈.parent.consume(a+new attributes)
```

Abbildung 2.17: Ein simples Übersetzungsschema, um das *produce/consume-Interface* zu veranschaulichen.[49]

(LLVM) [48] zum Einsatz. Dieses bietet ein ausgereiftes und effizientes Backend, um Maschinencode zu erzeugen. Es besitzt eine C++ API, womit beispielsweise auch direkt auf Register zugegriffen werden kann. Das Interface von LLVM ist zwar sehr nahe an der Hardware, allerdings sind an manchen Stellen des Kompilierens vom Code komplexere Funktionalitäten erwünscht. Daher verwendet HyPer eine Mischung aus vorkompiliertem C++ Code sowie LLVM. Abbildung 2.18 veranschaulicht die Interaktion zwischen beiden.

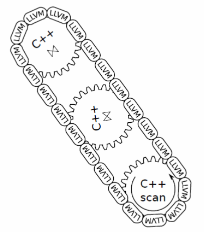


Abbildung 2.18: Interaktion zwischen LLVM und C++. [48]

Um eine optimale Performanz zu gewährleisten, wird LLVM eine fast vollständige Kontrolle verliehen. Der C++ Anteil wird nur zwischenzeitlich aufgerufen, wenn es nötig ist. Im Folgenden soll kurz aufgezeigt werden, welche Aufgaben wie verteilt werden.

- **C++**:
  - Datenstruktur-Verwaltung
  - Auslagerung auf externe Speichermedien
  - Bestimmung der Scan-Reihenfolge
  - u.a.
- **LLVM**:
  - Tupel-Zugriffe
  - Materialisierung in einer Hash-Tabelle
  - Filtern
  - u.a.

2.3.6 Morsel-getriebener Parallelismus

Heutzutage existieren viele Multi-/Manycore Systeme. HyPer setzt sich das Ziel, solche Systeme möglichst gut auszulasten. Dazu soll die Arbeit gleichmäßig auf alle Kerne aufgeteilt werden. Dabei bietet der *Morsel*-getriebene Ansatz [39] eine feingranulare Aufteilung der Last unter Berücksichtigung von *NUMA-Systemen* (siehe Abbildung 2.19). So ist der Grad des Parallelismus dynamisch und kann zur Laufzeit angepasst werden.

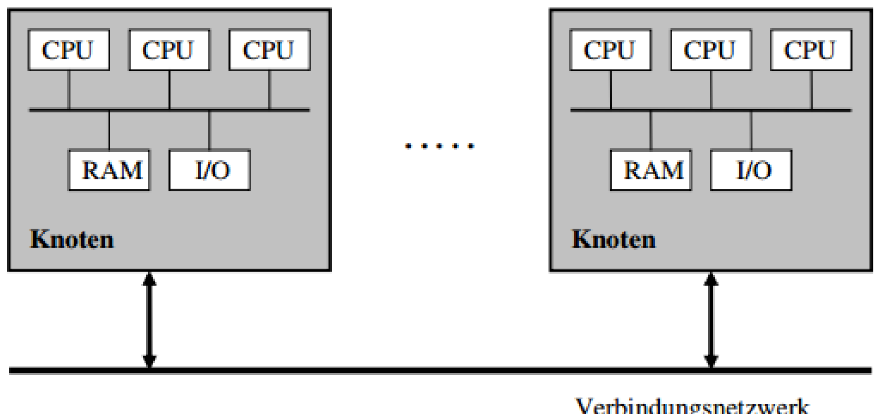


Abbildung 2.19: Aufbau eines NUMA-Systems. [21]

Abbildung 2.20 soll die Idee hinter dem Ansatz genauer verdeutlichen. Einzelne Teilstücke der relevanten Datenmenge - auch **Morsel** genannt - werden gleichmäßig auf die Knoten zugeteilt. Tests haben ergeben, dass eine Größe von ungefähr 100.000 Tupel eine gute Performanz liefern.

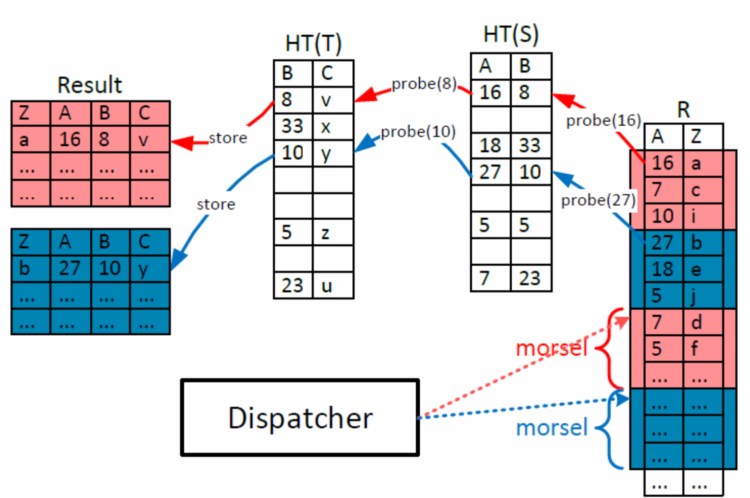


Abbildung 2.20: Idee des Morsel-getriebenen Parallelismus. [39]

In der Abbildung wird beispielhaft ein Hash-Join zwischen den Tabellen R, S und T durchgeführt. Dazu weist der *Dispatcher* zunächst die Arbeit auf die Knoten zu, welche hier farblich gekennzeichnet sind. Ist ein Knoten mit der Arbeit fertig, kann dieser sich das nächste Morsel greifen und weiterarbeiten.
Doch nicht nur innerhalb einer Pipeline, sondern auch die Pipelines selbst werden parallel abgearbeitet. Abbildung 2.21 zeigt die Parallelisierung des besagten Hash-Joins.

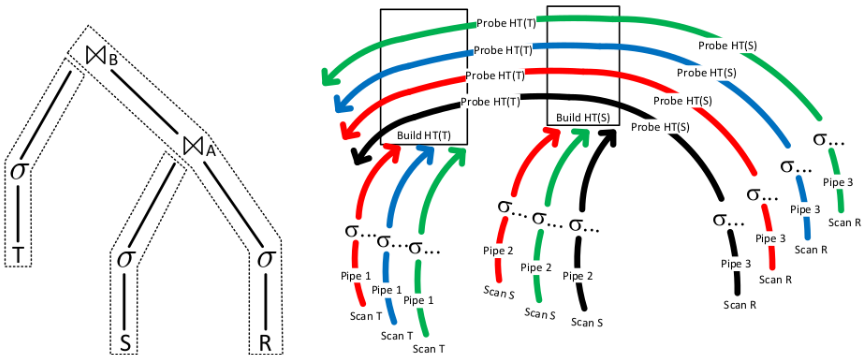


Abbildung 2.21: Parallelisierung von drei Pipelines des Anfrageplans. [39]

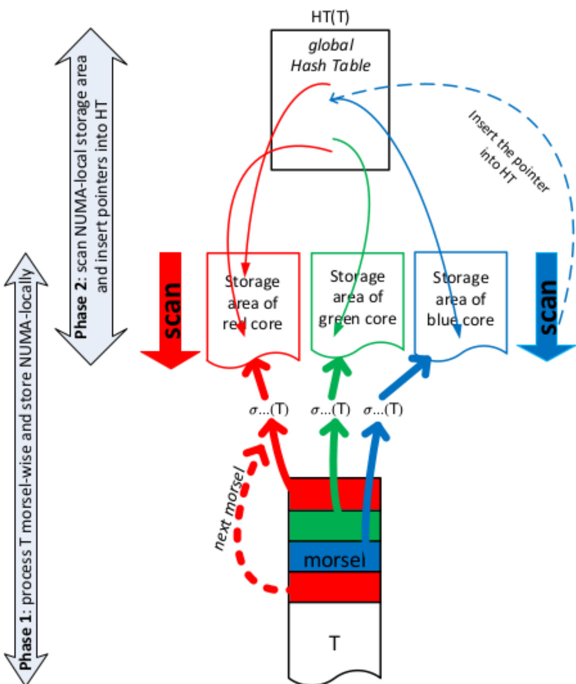


Abbildung 2.22: Auswertung der Anfrage unter Berücksichtigung des NUMA-Systems. [39]

Die Zwischenergebnisse werden bis zu einem Pipeline-Breaker lokal behalten. Ist ein Knoten mit der Arbeit fertig, kann dieser die Arbeit von anderen übernehmen, um einen Lastausgleich zu erreichen. Ein Speicher übergreifender Zugriff ist zwar möglich, soll jedoch so weit wie möglich vermieden und eher lokal gearbeitet werden. In Abbildung 2.22 wird das Aufbauen einer Hash-Tabelle von T demonstriert. Nachdem die Tabelle morselweise abgearbeitet wurde, werden in einer globalen Hash-Tabelle lediglich Zeiger auf die lokalen Speicherbereiche gesetzt. Die Zwischenergebnisse werden dann auch nochmals in gleich große Morsel eingeteilt, sodass eine weitere Verarbeitung dieser optimal verläuft.

Die Zuweisung der Arbeit übernehmen das sogenannte *QEPObject* und der *Dispatcher*. Ersteres kennt den Anfrageplan und die daraus resultierenden Pipelines. Das QEPObject übergibt dem Dispatcher diejenigen *Pipeline-Jobs*, deren Voraussetzungen erfüllt sind. So kann beispielsweise ein Hash-Join erst ausgeführt werden, wenn die nötigen Hash-Tabellen zuvor erzeugt worden sind. Zu beachten ist, dass der Dispatcher eine lockfreie Datenstruktur sein muss, da möglicherweise mehrere Knoten, die ihre Arbeit bereits erledigt haben, parallel neue anfordern können. Abbildung 2.23 zeigt eine solche Anreihung von Jobs und deren Zuweisung zu den Knoten im Dispatcher.

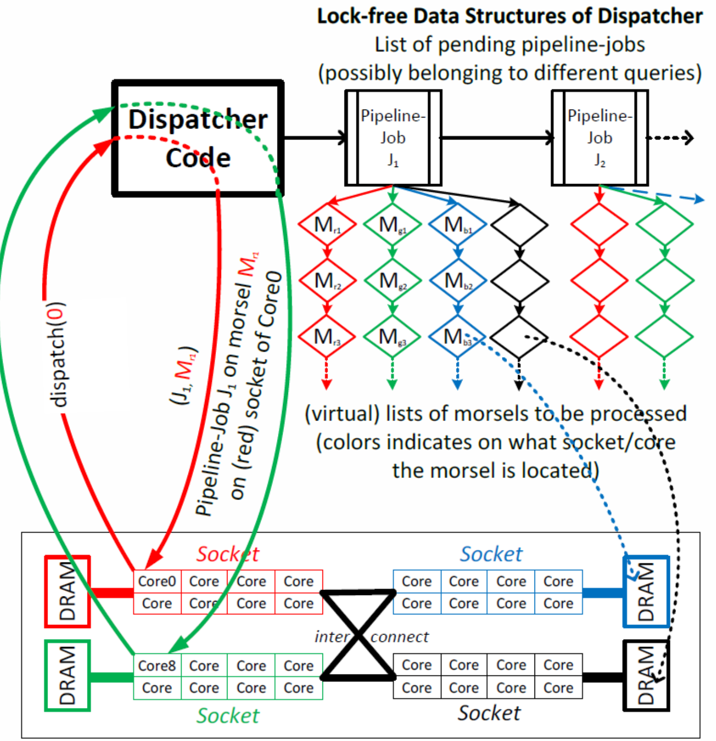


Abbildung 2.23: Die Zuweisung von Pipeline-Jobs durch den Dispatcher. [39]

2.3.7 Performanz

Um die Performanz zu analysieren, wurden der TPC-C und der TPC-H Bechmark zu TPC-CH kombiniert. Um HyPer mit anderen Systeme vergleichen zu können, wurden ebenfalls Anpassungen vorgenommen, wenn diese für das jeweilige System nötig waren.

	HyPer configurations						**MonetDB**	**VoltDB**
	one query session (stream) single threaded OLTP		8 query sessions (streams) single threaded OLTP		3 query sessions (streams) 5 OLTP threads		no OLTP 1 query stream	no OLAP only OLTP
Query No.	OLTP throughput	Query resp. times (ms)	OLTP throughput	Query resp. times (ms)	OLTP throughput	Query resp. times (ms)	Query resp. times (ms)	results from [18]
Q1	new order: 56961 tps; total: **126576** tps	67	new order: 29359 tps; total: **65269** tps	71	new order: 171384 tps; total: **380868** tps	71	63	**55000** tps on single node; **300000** tps on 6 nodes
Q2		163		233		212	210	
Q3		66		78		73	75	
Q4		194		257		226	6003	
Q5		1276		1768		1564	5930	
Q6		9		19		17	123	
Q7		1151		1611		1466	1713	
Q8		399		680		593	172	
Q9		206		269		249	208	
Q10		1871		2490		2260	6209	
Q11		33		38		35	35	
Q12		156		195		170	192	
Q13		185		272		229	284	
Q14		122		210		156	722	
Q15		528		1002		792	533	
Q16		1353		1584		1500	3562	
Q17		159		171		168	342	
Q18		108		133		119	2505	
Q19		103		219		183	1698	
Q20		114		230		197	750	
Q21		46		50		50	329	
Q22		7		9		9	141	

Abbildung 2.24: Performanz Vergleich zwischen HyPer, MonetDB und VoltDB.[36]

Abbildung 2.24 zeigt die insgesamte Performanz von HyPer im Vergleich zu MonetDB und VoltDB. Dabei sei angemerkt, dass die Werte für VoltDB aus anderen Quellen übernommen worden sind, wo die Benchmarks auf ähnlicher, aber nicht derselben, Hardware laufen gelassen wurden.
Im Vergleich zu VoltDB ist zu erkennen, dass der Durchsatz für OLTP Anfragen auf einem einzigen Server fast doppelt so hoch ist, obwohl parallel dazu eine OLAP Session läuft. Selbst bei acht OLAP Sessions und einem OLTP Thread ist der Durchsatz vergleichbar mit dem von VoltDB. Die Performanz könne hier sogar gesteigert werden, wenn dem OLTP Thread eine höhere Priorität zugewiesen werden würde.
Hinsichtlich der Performanz der OLAP Sessions, lassen sich im Vergleich zu MonetDB keine großen Unterschiede erkennen. Die Differenzen bei einigen Anfragen ergeben sich durch ein nicht perfekt angepasstes MonetDB, sodass sich die Anfragepläne bei HyPer teilweise unterschieden.

Abgesehen von der insgesamten Performanz des Systems, sind einige andere Statistiken ebenfalls interessant, um ein Fazit für SliceDB zu ziehen. So stellt sich die Frage, inwiefern die Optimierungen durch das Kompilieren der Anfragen in Maschinencode und die Verwendung von Morsels eine Steigerung der Performanz mit sich bringen und ob sich der Aufwand dafür lohnt.

Maschinencode Das Ausführen einer simplen Selektion mit einer verschiedenen Anzahl an Filterkriterien, führt zu den Ausführungszeiten, die in Abbildung

2.25 gezeigt werden. Dabei ist zu erkennen, dass der datenzentrische Ansatz eine deutlich bessere Performanz liefert als das Iterator-Model sowie eine blockweise Verarbeitung. Außerdem ist zu erkennen, dass das LLVM Backend ein optimierender Kompilierer ist, welcher bei einfachen Anfragen (siehe Spalte 0) entsprechenden Code erzeugt.

	0	1	2	3
data-centric compilation	0.001	5.199	7.037	18.753
iterator model - compiled	3.283	16.009	28.185	40.475
iterator model - interpreted	3.279	30.317	58.701	90.299
block processing - compiled	10.97	13.129	20.001	26.292

Abbildung 2.25: Performanz von unterschiedlichen Ansätzen eines Mikrobenchmarks. [49]

Abbildung 2.26 veranschaulicht, wie sich die Laufzeiten des TPC-H Benchmarks in HyPer im Vergleich zu Vectorwise verhalten, wenn die Anfragen in Maschinencode kompiliert werden. Neben der Kompilierungszeit und der Codegröße, werden hier auch der prozentuale Anteil des Maschinencodes, welcher durch LLVM zur Laufzeit generiert wurde, und die in diesem Code verbrachte Ausführungszeit aufgelistet. Es ist zu erkennen, dass die Laufzeiten sehr ähnlich sind. Einige Anfragen sind in HyPer deutlich schneller, andere wiederum etwas langsamer. Im Durchschnitt soll HyPer allerdings schneller sein als die blockweise Verarbeitung von Vectorwise.

TPC-H #	compile	executed code	HyPer generated	time in generated	runtime	Vectorwise runtime
1	13ms	5.6KB	42%	98%	9.0s	33.4s
2	37ms	10.9KB	58%	86%	2.4s	2.7s
3	15ms	6.6KB	36%	89%	27.5s	25.6s
4	12ms	6.2KB	33%	93%	21.6s	22.4s
5	23ms	8.1KB	42%	86%	31.4s	29.7s
6	5ms	1.1KB	82%	96%	5.5s	7.1s
7	31ms	9.4KB	51%	88%	23.8s	28.2s
geo. mean (all 22)	19ms	6.8KB	47%	81%	16.2s	21.5s

Abbildung 2.26: Laufzeiten des TPC-H Benchmarks für HyPer und Vectorwise. [49]

Morsel In Abbildung 2.27 wurden die Laufzeiten für das TPC-H Benchmark unter verschiedenen Konfigurationen gemessen. Angegeben sind jeweils die Anzahl der verwendeten Kerne (33 bis 64 virtuell) und die daraus resultierende Performanz Steigerung. Das erste System ist hierbei Vectorwise. In fast allen Fällen ist zu erkennen, dass jegliche Konfiguration von HyPer eine bessere Performanz liefert. Dabei ist die nicht-adaptive Konfiguration die schlechteste. Sofern Kerne nämlich nicht die Arbeit von anderen übernehmen, wenn sie nichts mehr zu tun haben, wird eine Möglichkeit zur Parallelisierung verschwendet. Ist das

System adaptiv, berücksichtigt jedoch nicht die NUMA-Architektur, so geht ebenfalls Performanz verloren. Die beste Konfiguration ist demnach eine, die beide Optionen verwendet. Zwar sieht es so aus, als sei HyPer Vectorwise in diesem Punkt extrem überlegen, jedoch sei erwähnt, dass die Operatoren von HyPer sehr stark bezüglich der Datenlokalität optimiert sind.

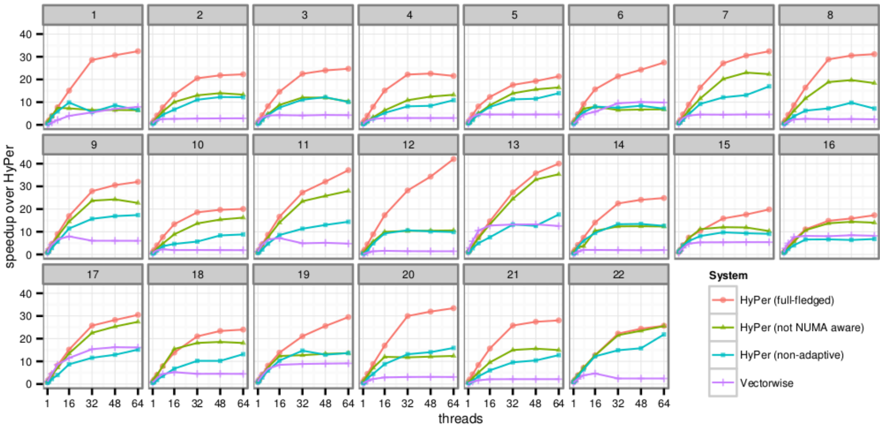


Abbildung 2.27: Skalierbarkeit des TPC-H Benchmarks unter Verwendung des Morsel-getriebenen Parallelismus. [39]

2.3.8 Fazit

HyPer stellt mit seinem hybriden System zwar eine geschickte Möglichkeit dar, mit Hilfe von Snapshots eine parallele Ausführung von sowohl OLTP als auch OLAP Anfragen zu ermöglichen, jedoch ist eine solche Anforderung für ein OLAP System wie SliceDB nicht nötig. Trotzdem ist es lohnenswert, sich die weiterführenden Techniken des Systems anzuschauen.
Der datenzentrische Ansatz, den HyPer durch das Kompilieren der Anfragen in Maschinencode verfolgt, ist eine davon. Die parallele Abarbeitung von Pipelines und die Maximierung der Datenlokalität führen zu einer Steigerung in der Performanz. Die Statistiken zeigen, dass HyPer eine durchschnittlich bessere Laufzeit für das TPC-H Benchmark erreicht als ein System wie Vectorwise, welches eine blockweise Abarbeitung der Daten vornimmt. Der Aufwand hierbei ist jedoch sehr hoch, da insbesondere die Operatoren hinsichtlich der Datenlokalität optimiert werden müssen. Außerdem wurde das LLVM Backend verwendet, das in seiner Grundform nicht ausreichend war und durch vorkompilierten Code ergänzt werden musste. Für SliceDB ist diese Technik zu fortgeschritten und zeitintensiv in der Implementierung, sodass sie für die Zielsetzung der Projektgruppe keine Priorität bekommt.
Eine andere Technik, die in HyPer verwendet wird, ist die morselweise Parallelisierung. Die dynamische Auslastung von Multicore Systemen ist eine wünschenswerte Eigenschaft. Doch auch hier ist eine Optimierung für NUMA Systeme zu komplex im Rahmen der Projektgruppe, sodass diese Technik ebenfalls außen vor gelassen und stattdessen eine einfache Parallelisierung der Operatoren selbst bevorzugt wird.

2.4 CoGaDB *(Christian Schnieder)*

CoGaDB[2] ist eine spaltenorientierte Hauptspeicherdatenbank mit eingebauter GPU-Beschleunigung, welche für OLAP-Anwendungen optimiert ist. Den Kern des Datenbanksystems bildet dabei das selbstoptimierende Framework *Hybrid Query Processing Engine (HyPE)*. Dieses verteilt die Arbeitslast optimal auf die verfügbaren Prozessoren, ohne die genaue Hardware zu kennen. Das ermöglicht eine optimale Nutzung von unterschiedlichen Algorithmen auf heterogenen Systemen, wie sie die Hardwarelandschaft heutzutage bietet. [18]
Mit der Verwendung von HyPE verfolgt CoGaDB das Ziel, zwei Basisprobleme für die Nutzung von zukünftiger Hardware abzudecken [18]:

1. die Einschätzung der Ausführungskosten bestimmter Datenbankoperatoren auf bestimmten Prozessoren, ohne die zugrunde liegende Hardware genau zu kennen und

2. eine effiziente Verteilung der Arbeitslast auf alle Prozessoren, wobei jedem heterogenen Prozessor die Aufgaben zugewiesen werden, für die er am geeignetsten ist.

Für eine effiziente Datenverarbeitung nutzt CoGaDB die parallelen Bibliotheken *Threading Building Blocks (TBB)* [3] von Intel auf der CPU und Thrust [4] auf der GPU. Für die Verwendung von Grafikprozessoren außerhalb ihres ursprünglichen Aufgabenbereichs wird das GPGPU-Framework CUDA von NVIDIA verwendet. [18]
Da die Entwickler von CoGaDB sich auf die Verarbeitung von OLAP-Anfragen konzentrieren, wird das *Star Schema Benchmark (SSBM)* für den Vergleich mit alternativen Datenbanksystemen verwendet. Die dafür hinreichenden Datentypen (32-Bit Integer, 32-Bit Float und Strings variabler Länge) werden entsprechend unterstützt. [18]

2.4.1 Architektur

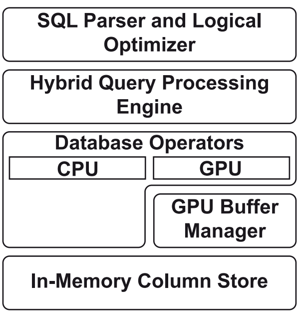


Abbildung 2.28: Architektur von CoGaDB [18].

Dieser Abschnitt gibt einen Überblick über die unterschiedlichen Komponenten von CoGaDB, welche in Abbildung 2.28 zu sehen sind.

[2] `http://cogadb.cs.tu-dortmund.de/wordpress/`
[3] `https://www.threadingbuildingblocks.org`
[4] `http://thrust.github.io`

Speicherverwaltung

Die Basis des DBMS bildet der *In-Memory Column Store*, welcher die Daten spaltenorientiert im Arbeitsspeicher hält. Durch das spaltenorientierte Datenlayout kann die Speicherhierarchie effizienter genutzt werden und durch separierte Speicherung der unterschiedlichen Datentypen können individuelle Kompressionsverfahren mit hohen Kompressionsraten angewandt werden. Die spaltenorientierte Speicherung der Daten eignet sich somit besonders gut für OLAP-Anwendungen. [18]
Für die effiziente Nutzung von GPUs ist es notwendig die Daten im Arbeitsspeicher anstatt auf der deutlich langsameren Festplatte liegen zu haben [29]. Daher hält CoGaDB alle Daten im Arbeitsspeicher. Die Verdrängung der Daten auf die Festplatte beim Überlaufen des Arbeitsspeichers übernimmt das Betriebssystem mit seiner virtuellen Speicherverwaltung. [18]

Verarbeitung und Operatormodell

Für die Verarbeitung von Anfragen verteilt CoGaDB die einzelnen Operatoren auf die verfügbaren Prozessoren. Das genaue Verfahren wird in Abschnitt 2.4.3 beschrieben. Um bei der Anfragenverarbeitung unnötigen Datentransfer zwischen verschiedenen Prozessoren einer Maschine zu vermeiden, muss jeder Prozessor durch Intra-Operator-Parallelität effizient genutzt werden. Bei verzweigten Anfrageplänen nutzt CoGaDB zusätzlich Inter-Operator-Parallelität, um unabhängige Teil-Anfragepläne parallel auszuführen. [18]
In Abschnitt 2.4.2 wird im Detail auf die Umsetzung der einzelnen Operatoren eingegangen.

Materialisierungsstrategie Um als Ergebnis einer Datenbankanfrage eine zeilenbasierte Darstellung zu erhalten, muss das spaltenbasierte (Zwischen-)Ergebnis umgewandelt werden. Aufgrund der hohen Selektivität bei OLAP-Anfragen verfolgt CoGaDB den Ansatz der *late materialization* und transformiert die Ergebnistabelle erst so spät wie möglich. [18]

Parallelität CoGaDB nutzt sowohl Intra-Operatorparallelität als auch Inter-Operatorparallelität. Der Schwerpunkt der Implementierung liegt jedoch bei der Inter-Operatorparallelität, da hohe Parallelität innerhalb von Operatoren bei stark verzweigten Anfrageplänen zur Überbeanspruchung von Prozessoren führen kann [18]. Für die parallele Ausführung von Threads auf der GPU kommt das CUDA-Framework zum Einsatz, welches Parallelität mithilfe von CUDA streams realisiert [50]. Im Gegensatz zu dieser aufwändigen Methode, stellt die parallele Ausführung von Threads auf CPUs kein Problem dar.

GPU-Buffer-Manager

Prozessoren mit eigenem Speicher müssen benötigte Spalten und Join-Indizes zunächst anfordern, bevor sie damit arbeiten können. Dies geschieht über den zentralen GPU-Buffer-Manager. Falls die angeforderten Daten nicht bereits im GPU-Speicher liegen, werden diese dorthin übertragen.
Für die Allokation von neuem Speicher verfolgt der Buffer-Manager die *allocate-as-needed*-Strategie, da diese im Gegensatz zur Vorabzuweisung keine eventuell

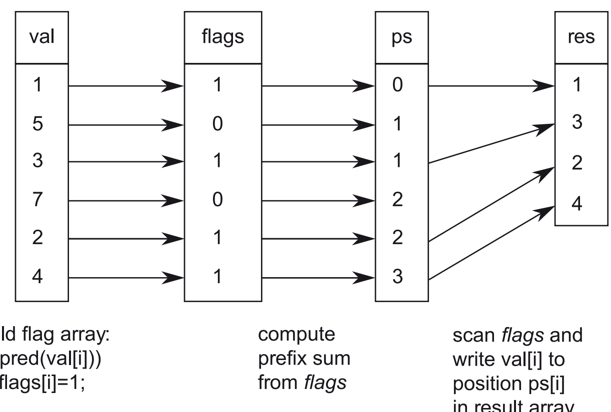


Abbildung 2.29: Parallele Selektion auf GPUs für das Prädikat $val < 5$ [18].

noch benötigten Daten aus dem Buffer wirft und der zu allozierende Speicher nicht aufwändig mit abgeschätzt werden muss. Wenn der Speicher durch die *allocate-as-needed*-Strategie voll läuft, müssen zwischengespeicherte Daten freigegeben werden. Da die erneute Allokation von Spalten günstiger ist als die von Join-Indizes, werden zunächst die Spalten aus dem Speicher entfernt und anschließend erst die Join-Indizes.
Es kann vorkommen, dass nach dem Aufräumen des Speichers durch schrittweises allozieren durch zwei Operatoren für beide Operatoren nicht mehr genug Speicher für den nächsten Prozessschritt verfügbar ist. In dem Fall wird ein GPU-Operator abgebrochen und auf der CPU neu gestartet. [18]

2.4.2 Operatoren

Dieser Abschnitt stellt die unterschiedlichen Operatoren vor, die in CoGaDB Anwendung finden.

Selektion

Die Selektion auf der CPU erfolgt mittels einer parallelen Version des SIMD-Scans von Zhou und Ross [71]. Dazu scannt jeder Thread seine eigene Partition der Input-Spalte und schreibt die passenden TIDs in den lokalen Ausgabepuffer. Anschließend werden die lokalen Ergebnisse zusammengefasst, indem über die lokalen Trefferzahlen die Größe des Gesamtergebnisses ermittelt wird und anschließend alle Threads parallel ihre Ergebnisse in zugewiesene Speicherregionen schreiben. [18]
Abbildung 2.29 veranschaulicht die parallele Selektion von He und weiteren [29], welche auf der GPU zum Einsatz kommt. Zunächst werden die Eingabespalten gescannt um anschließend ein Flag-Array zu berechnen. Dieses hat an den Stellen, wo das Prädikat auf die Reihe passt, den Wert eins, andernfalls den Wert null. Anschließend wird aus diesem Flag-Array ein Präfixsummen-Array berechnet, woraus sich die Ergebnisgröße und die Schreibpositionen von jedem Thread im Ausgabepuffer ablesen lassen. [18]

CoGaDB unterstützt Dictionary-Kompression für Strings, wodurch diese auch auf der GPU verarbeitet werden können. Da das Dictionary jedoch nicht sortiert ist, können Strings nur auf Gleichheit oder Ungleichheit geprüft werden. [18]

Komplexe Selektion

Die häufig verwendete Selektion nach komplexen Prädikaten über mehrere Spalten führt CoGaDB durch Verkettung von Operatoren durch. Dabei wird jedes Prädikat einzeln und unabhängig ausgewertet und die Ergebnisse anschließend verschmolzen. Die Auswertung von Prädikaten und Verschmelzung von Ergebnissen kann effizient auf verschiedenen GPU-Kernen laufen. Das Scannen unterschiedlicher Spalten kann von mehreren GPUs übernommen werden. Das Ergebnis der komplexen Selektion ist eine sortierte Liste der TIDs. [18]

Join

In CoGaDB wurden vier verschiedene Join-Algorithmen implementiert.

Generischer Join Generische Joins macht keine Annahmen über Eingabetabellen und sind daher immer anwendbar. Allerdings können sie zu einem Kreuzprodukt ausarten, welches nicht in den GPU-Speicher passt. Aus diesem Grund wurde in CoGaDB nur ein generischer CPU Hash-Join implementiert. [18]

PK-FK-Join Der *primary-key/foreign-key*-Join ist die häufigste Join-Art in OLAP-Anwendungen. Die Ergebnisgröße lässt sich leicht bestimmen, da das Ergebnis eines PK-FK-Joins genauso viele Zeilen hat, wie es *foreign-keys* in der FK-Tabelle gibt. In CoGaDB wird eine optimierte Version des Index Nested Loop Joins aus [29] für GPUs eingesetzt. Dieser Algorithmus sortiert zunächst die PK-Spalte und weist dann jedem Thread eine Anzahl von *foreign-keys* zu, welche mithilfe einer binären Suche aus den sortierten *primary-keys* ermittelt werden. Die Ergebnisse werden schließlich mittels eines CPU Hash-Joins, welcher die Hash-Tabelle fortlaufend generiert, zusammengeführt und parallel beschnitten. [18]

Fetch-Join Häufig lohnt es sich, Dimensionstabellen vor einem Join mit der Faktentabelle zu filtern. CoGaDB nutzt dazu einen vorberechneten Join-Index und extrahiert mit den passenden TIDs der Dimensionstabelle die passenden TIDs der Faktentabelle. Dieses Verfahren ist sowohl auf der CPU als auch auf der GPU sehr effizient. [18]

Parallele Star-Joins Bei einer Anfrage mit mehreren Joins zwischen Fakten- und Dimensionstabellen können die einzelnen Joins mit einem Star-Join parallel durchgeführt werden. CoGaDB implementiert eine Variante des *inivisible join* von Abadi und weiteren [3], welcher ursprünglich aus den folgenden drei Phasen besteht:
Zunächst werden die Prädikate wie in Abbildung 2.30 auf die Dimensionstabellen angewendet. Daraus ergibt sich für jede Dimensionstabelle eine Menge an Schlüsselwerten, welche in eine Hash-Tabelle übertragen werden.

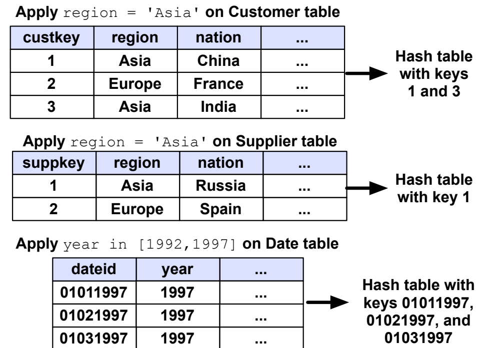


Abbildung 2.30: Star-Join: Anwendung der Prädikate auf Dimensionstabellen [3].

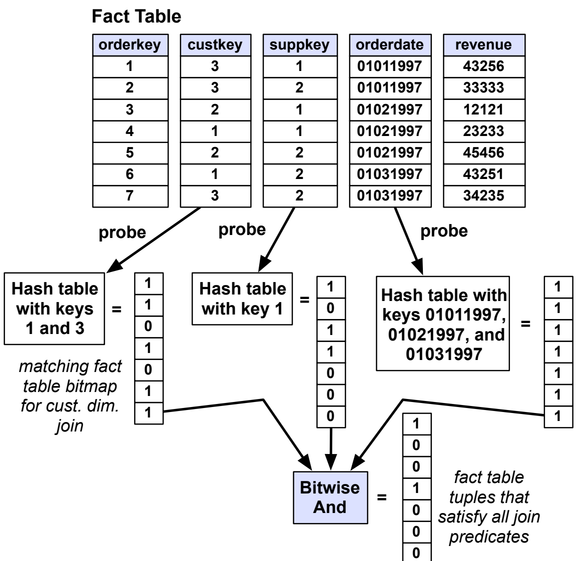


Abbildung 2.31: Star-Join: Produktion und Verundung von Bitmaps zu den Faktentabellen[3].

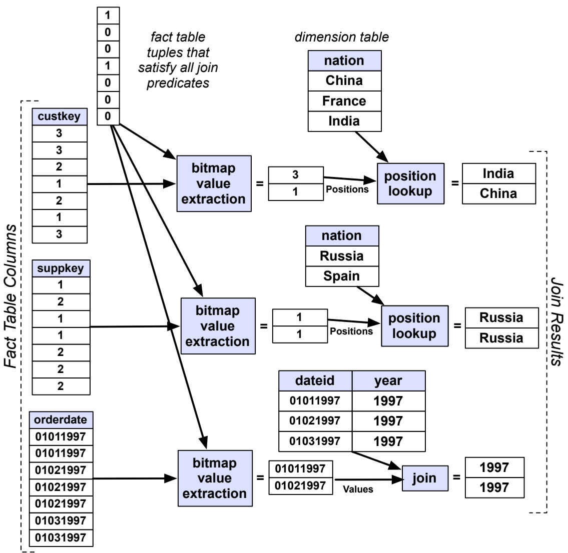


Abbildung 2.32: Star-Join: Extrahieren der relevanten Daten aus den Faktentabellen und zusammenführen der Ergebnisse [3].

Mit den erhaltenen Fremdschlüsseln der Faktentabelle werden in der zweiten Phase (Abbildung 2.31 für die unterschiedlichen Spalten Bitmaps produziert, die die Treffer der einzelnen Dimensionen enthalten. Diese Bitmaps lassen sich dann mit einer bitweisen UND-Operator kombinieren.
In der dritten Phase werden die passenden Zeilen in der in den Dimensionstabellen gesucht um das Ergebnis des Star-Joins zu konstruieren (Abbildung 2.32.
In der Variante von CoGaDB wird das Erzeugen der Hash-Tabelle und das Beschneiden der Faktentabelle durch einen Fetch-Join mit einem Join-Index ersetzt. Dadurch Verbessert sich die Performance bei großen Dimensionstabellen signifikant. Die daraus resultierenden Schlüssel des Join-Indexes werden in eine Bitmap konvertiert und bitweise mit einem UND kombiniert. Die kombinierte Bitmap wiederum wird in eine Positionsliste konvertiert, welche das von CoGaDBs Anfrageprozessor erwartete Ausgabeformat hat.
Jeder Schritt des Star-Join kann sowohl von der CPU als auch von der GPU bearbeitet werden. [18]

Sortierung

CoGaDB nutzt zum Sortieren die parallele Sortierfunktion aus Intels TBB-Bibliothek auf der CPU. Soll eine Tabelle nach den Spalten $A_1, ..., A_n$ sortiert werden, dann wird zunächst die Spalte A_n sortiert. Daraus resultiert eine TID-Liste, mit der die Werte der Spalte A_{n-1} geholt werden und die sortierte Version A'_{n-1} berechnet wird. Dieser Schritt wird so lange wiederholt, bis auch die Spalte A_1 sortiert wurde. Dann enthält die finale Liste die TIDs in der korrekten Reihenfolge. Voraussetzung hierfür ist ein stabiler Sortieralgorithmus. Für die üblichen Ergebnisgrößen von nur einigen hundert Tupeln ist dieser Algorithmus performant und kann sowohl auf der CPU als auch auf der GPU ausgeführt werden. [18]

select sum(X) from tab group by A, B;

A	B
8	3
11	6
8	3
15	5

P	A	B	TIDs
0..00	1000	0011	0
0..00	1011	0110	1
0..00	1000	0011	2
0..00	1111	0101	3

32 Bit 32 Bit

P: Padding Bits (Zero)

Abbildung 2.33: Packen von Werten mehrere Spalten, um eine Tabelle nach den Spalten A und B in einem einzigen Schritt zu gruppieren [18].

Gruppierung

Die beiden Gruppierungsalgorithmen in CoGaDB basieren auf der Sortierung. Der erste ist ein generischer Algorithmus und nutzt eine Sortierung nach mehreren Spalten.

Der zweite ist ein spezialisierter Algorithmus, der mehrere Spalten zu einer zusammenfasst, wie es am Beispiel in Abbildung 2.33 zu sehen ist. Die Spalten A und B lassen sich aufgrund des kleinen Wertebereiches jeweils mit 4 Bit darstellen. Durch Auffüllen mit führenden Nullen erhalten wir Gruppenschlüssel als 32-Bit-Integer. Um nach dem Sortieren eine TID-Liste zu erhalten müssen diese zunächst an die Gruppenschlüssel anhängen und erhalten so eine Liste mit 64-Bit-Integer. Diese lässt sich mit nur einem Durchlauf sowohl auf der CPU als auch auf der GPU sortieren, anschließend müssen nur noch die hinteren 32 Bit als Ergebnis-TID-Liste extrahiert werden. [18]

Aggregation und Arithmetische Operationen

CoGaDB unterstützt die Aggregationsfunktionen SUM, COUNT, MIN und MAX. Die Aggregation basiert auf der Gruppierung und wird im Anschluss auf der CPU seriell oder auf der GPU mittels Thrust parallel reduziert. Auch komplexe Aggregationsfunktionen wie zum Beispiel SELECT SUM(A+B/C) werden mittels eines Operatorbaumes und verschiedenen implementierten Operatoren berechnet. [18]

2.4.3 HyPE

Wie bereits in Abschnitt 2.4.2 dargestellt, werden die meisten Operatoren sowohl für die CPU als auch für die GPU bereitgestellt. Darum, dass die anstehende Arbeitslast optimal verteilt wird und auch alle Ressourcen in einem heterogenen Prozessorsystem effizient genutzt werden, kümmert sich die *Hybrid Query Processing Engine*. Sie besteht aus drei Komponenten: der Schätzungskomponente, dem Algorithmen-Selektor und dem Hybriden Anfragenoptimierer (Abbildung 2.34).

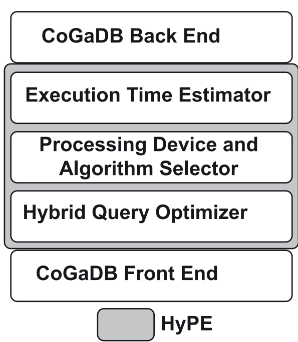


Abbildung 2.34: Architektur von HyPE [18].

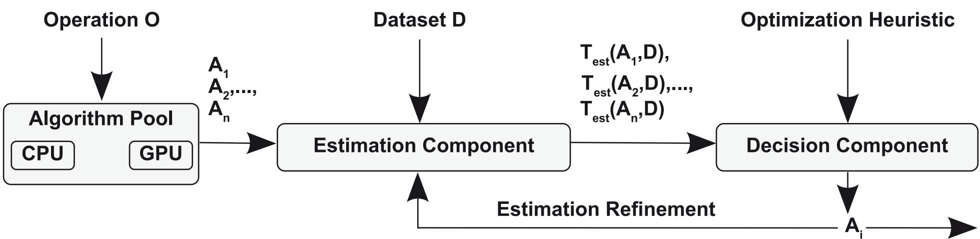


Abbildung 2.35: HyPE Entscheidungs-Modell [19].

Schätzungskomponente

Das Ziel der Schätzungskomponente ist es, ohne genaue Kenntnis über Hardware und Algorithmen, diese optimal einander zuzuordnen. Dabei schätzt sie das Performanceverhalten beim Zusammenspiel von verschiedenen Datenbankalgorithmen, Hardware und Eingabedaten. Die ersten Ausführungszeiten dienen dabei als Trainingsdaten, doch auch danach überwacht HyPE die Algorithmenlaufzeiten kontinuierlich und verfeinert mit neu gewonnenen Daten die Kostenmodelle. [18]

Prozessorzuordnung und Algorithmen-Selektor

Die Prozessorzuordnung und die Auswahl der Algorithmen basiert auf den Ergebnissen der Kosteneinschätzung. Abbildung 2.35 zeigt das Entscheidungsmodell, das HyPE zugrunde liegt. Soll eine Operation ausgeführt werden, steht HyPE ein Pool an Algorithmen zur Auswahl. Daraus werden zunächst für den gewünschten Operator alle passenden Algorithmen abgerufen. Anschließend berechnet die Einschätzungskomponente die erwarteten Ausführungszeiten. Mit diesen Informationen und einer benutzerspezifischen Heuristik kommt nun die Entscheidungskomponente zum Einsatz und wählt den passenden Algorithmus aus, welcher die Operation ausführt. Nach der Ausführung wird die Ausführungszeit an die Schätzungskomponente zurückgegeben.

Dadurch, dass jedem Algorithmus ein ein Prozessortyp zugewiesen ist, schafft HyPE es, mit diesem Verfahren in nur einem einzigen Schritt sowohl Algorithmus als auch Prozessor auszuwählen. [18]

Abbildung 2.36: HyPE Belastungs-Tracking [19].

Für den Fall von mehreren Einheiten löst HyPE das Problem so, dass es jedem Algorithmus eine eindeutige Kennung zuweist, die genau festlegt, welcher Prozessor zu welchem Algorithmus gehört. Der optimale Algorithmus wird nach einer Optimierungsstrategie gewählt, die bei HyPE standardmäßig *Waiting Time Aware Response Time (WTAR)* ist. In Abbildung 2.36 ist zu sehen, dass diese Planungsstrategie die Auslastung der einzelnen Prozessoren berücksichtigt, indem die geschätzte Ausführungszeit aller einem bestimmten Prozessor zugeordneten Algorithmen aufsummiert wird. Dem Prozessor, von dem die kleine Antwortzeit zu erwarten ist, wird der neue Operator zugewiesen. Dadurch verteilt sich die Arbeitslast gleichmäßig auf alle verfügbaren Prozessoren. [18]

Hybride Anfragenoptimierung

Der Anfragenoptimierer weist jedem Operator eines Anfrageplanes einen passenden Prozessor und Algorithmus zu. Dabei unterstützt HyPE zwei Modi: Im ersten wird beim Durchlaufen des Ausführungsplan die Operatorplatzierung vom Algorithmus-Selektor angefordert. Diese gierige Strategie berücksichtigt bei der Verwendung von WTAR auch die Auslastung der Prozessoren und unabhängige Teilpläne können auf verschiedenen Prozessoren laufen. Dadurch ergibt sich ein erheblicher Performancegewinn. Die zweite Modus erstellt verschiedene kandidierende Pläne und schätzt deren Ausführungszeiten. Der Nachteil beim letzteren Verfahren ist, dass hierfür Kardinalitätsannahmen getroffen werden müssen und diese nur schwer zu schätzen sind. [18]

2.4.4 Performance

Breß und weitere haben haben CoGaDB in [18] mit MonetDB, dem Pionier unter den spaltenbasierten Datenbankmanagementsystemen, verglichen. Dabei kam eine Machine mit einem Intel® Core™i7-4770 CPU mit 4 Kernen (@ 3.40 GHz) mit 24 GB Hauptspeicher und einer GeForce® GTX 660 GPU (@980 MHz) mit 1,5 GB Gerätespeicher zum Einsatz. Zum Vergleichen der beiden DBMS wurde das Star Schema Benchmark mit dem Scale-Faktor 15 verwendet. [18]

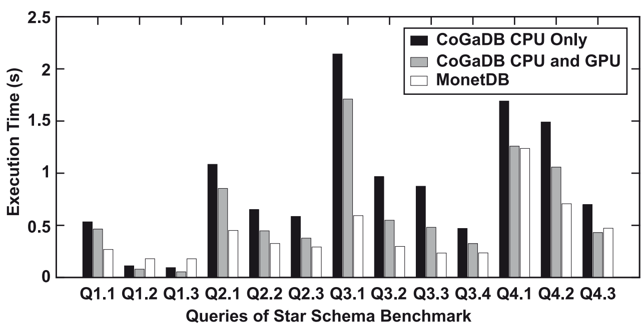


Abbildung 2.37: Antwortzeiten von CoGaDB und MonetDB für das SSBM mit dem Scale-Faktor 15 [18].

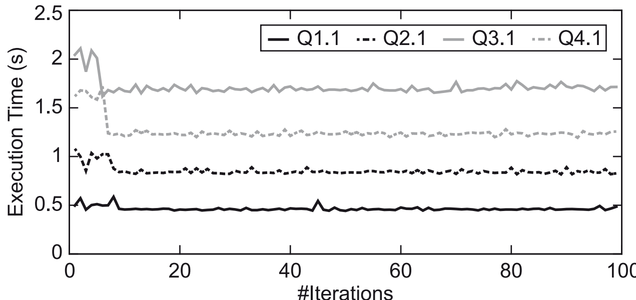


Abbildung 2.38: Antowrtzeiten von ausgewählten SSBM-Queries in CoGaDB über 100 Ausführungen [18].

In Abbildung 2.37 ist ein Vergleich der Antwortzeiten beider Systeme zu sehen. Man kann direkt erkennen, dass bei jeder einzelnen Abfrage in CoGaDB die GPU-Beschleunigung greift und zum Teil sogar sehr große Vorteile bei der Reaktionszeit im Vergleich zur ausschließlichen Nutzung der CPU zeigt.

MonetDB ist bei fast allen Anfragen deutlich schneller als CoGaDB - selbst als die GPU-beschleunigte Version. Allerdings muss man hierbei beachten, dass MonetDB ein seit 2002 existierendes und inzwischen ausgereiftes System ist, welches bereits einen langen Entwicklungsprozess mit vielen Optimierungen durch gemacht hat. CoGaDB hingegen ist von 2013 und hatte seinen Schwerpunkt bisher nicht in der Optimierung der einzelnen Operationen. Auch würde ein aktuellerer Vergleich (die Ergebnisse wurden im August 2014 veröffentlicht) sicherlich anders aussehen.

Betrachtet man CoGaDB unabhängig von anderen Datenbanksystemen über mehrere Iterationen der Ausführung auf einem frisch aufgesetzten System, kann man in Abbildung 2.38 sehen, dass die ersten 10 Anfragen noch sehr langsam sind, die Antwortzeiten sich danach aber konstant auf einem niedrigen Level befinden. Das ist dadurch begründet, dass die Laufzeit vom System während der ersten Anfragen noch nicht ausreichend präzise geschätzt, diese durch die Sammlung von Trainingsdaten aber immer stabiler wird. So kann man sehen, dass das Konzept aufgeht und sich das System in einem kurzen Lernprozess auf neue Hardware einstellen kann.

2.4.5 Zusammenfassung

CoGaDB ist ein Datenbankmanagementsystem, welches mit der Nutzung von HyPE einen interessanten Ansatz verfolgt. Durch das selbstoptimierende Framework kann das DBMS sich innerhalb von kurzer Zeit auf eine neue Hardwareumgebung einstellen und die vorhandenen Komponenten sehr effizient nutzen. Auch die Nutzung von GPUs bringt hierbei einen erheblichen Vorteil in der Performance und es bleiben keine Prozessoren ungenutzt.

2.5 IBM DB2 BLU

2.5.1 Blink *(Harun Kara)*

Das System Blink wurde vom IBM Almaden Research Center entwickelt und in mehreren IBM Produkten, wie z.B. dem System DB2 mit BLU (**B**link **U**ltimate) Acceleration, integriert. Das von den Entwicklern selbst als ämbitioniert"bezeichnete Ziel war es, mit dem Blink System alle Business Intelligence Anfragen in konstanter Zeit beantworten zu können, ohne Berücksichtigung der Datenbankgröße und ohne Definition eines Performance Layers. Blink wurde von Grund auf entwickelt, um die Vorteile von modernen Multicore Prozessoren sowie von billigem DRAM (Haupt-)Speicher durch Engineering von hardwarebewussten Algorithmen auszunutzen. Die hier vorgestellte Version von Blink (2008) ist überholt und implementiert nur Funktionalitäten, die notwendig sind, um die Kernerneuerungen des Systems gegenüber klassischen Systemen zu präsentieren. [13]

Überblick

Blink ist dazu gedacht, BI-Anfragen effizient zu verarbeiten, wie z.B das Star Schema Benchmark. Blink hält eine Kopie der gesamten Datenbank im Hauptspeicher und somit auch die große Faktentabelle eines Star Schemas, mit ihren Millionen von Tupeln. Der Flaschenhals verschiebt sich vom Disk-I/O (Disk <-> Memory) zur Speicherbandbreite (Memory <-> CPU). Da Blink für alle Anfragen möglichst gleich-effizient laufen soll, verzichtet man bei diesem System auf ein Performance Layer, bei dem für bestimmte Workloads die Datenbank durch Indizes, Materialized Views usw. getuned wird. Stattdessen führt jede Anfrage eine Full Table Scan durch. Dieser Scan ist sehr anfällig für den Flaschenhals *Speicherbandbreite*. An dieser Stelle setzt Blink an und versucht durch eine intelligente Kompression namens Frequency Partitioning, die Speicherbandbreite möglichst effizient zu nutzen. Dabei werden die Daten zunächst anhand ihrer Häufigkeit partitioniert. Danach wird jede Partition separat dictionary-codiert, was durch den vorigen Schritt ein gutes Kompressions-Ergebnis liefert. Anschließend wird nochmal jede Partition separat delta-codiert. Weiterhin wird der Scan beschleunigt, indem Equality und Range Prädikate parallel, mit wenigen Instruktionen in der SIMD-Einheit der CPU ausgeführt werden, anstatt skalar mit teuren Loops und Branches. Des Weiteren geht es um effizientes Hash Grouping, bei dem für jede Partition, abhängig von der Anzahl an verschiedenen Gruppen innerhalb dieser Partition und der erwarteten Größe der Hash Tabelle, eine Hashing-Variante gewählt wird, damit die Hash Tabelle in den L2-Cache

passt und somit L2-Cache-Misses verhindert werden. Zum Schluss werden einige Ergebnisse vorgestellt, welche experimentell ermittelt wurden. [55]

Kompression und Speicherung

Ein Tupelcode ist eine Konkatenation von Dictionary-Codes für jedes Feld eines Tupels. Um auf den Feldcode für das i-te Feld eines Tupels zuzugreifen, muss das System die Feldcodes 1 bis i-1 parsen, um deren Codelänge zu bestimmen. Dies erzeugt Kontroll- und Datenabhängigkeiten, die die Performance von modernen Prozessoren schwer beeinträchtigen. Die Kosten dieser Decodierung bemessen sich ungefähr auf 2 Nanosekunden pro variabel großem Feld pro Tupel. Die Grundidee von **Frequency Partitioning** ist es, die Berechnung von Codelängen durch das Zusammengruppieren von Tupeln mit dem selben Muster von Feldcodelängen, zu amortisieren. Um dies zu erreichen, werden Tupel durch die Auftrittshäufigkeit ihrer Spaltenwerte dicht (viele Tupel in einer Partition) partitioniert, und Codes fester Länge werden innerhalb von Partitionen zugewiesen. Abbildung 2.39 illustriert dies auf einer Tabelle mit zwei Spalten. Blink startet mit der Teilung der unterschiedlichen Werte in jeder Spalte zu disjunkten Spaltenpartitionen, gemäß ihrer Auftrittshäufigkeiten. Diese sind C1a, C1b für Spalte 1 und C2a, C2b für Spalte 2. Jede Kombination von Spaltenpartitionen (z.B. C1a, C2b) formt eine Tabellen-Partition, die *Zelle* genannt wird. Dann erzeugt Blink ein separates Dictionary von Werten für jede Partition von C1 und C2, und weist ihnen Codes fester Länge zu. Diese Dictionaries werden benutzt, um die Tupel der Tabelle zu codieren. Somit wird garantiert, dass jedes Tupel in derselben Zelle das selbe Muster von Feldcodelängen besitzt. [55]

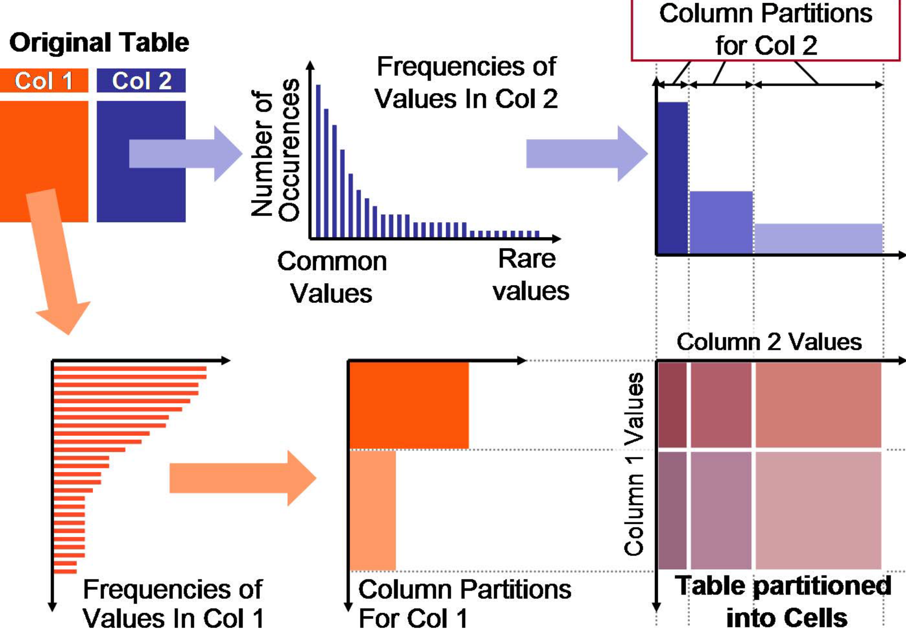


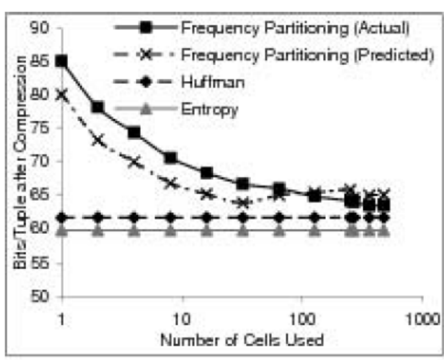
Abbildung 2.39: Partitionierung einer Tabelle mit zwei Spalten. [55]

Kompressionsprozess Der erste Schritt bei der Kompression einer Tabelle ist das Analysieren der Spalten-Verteilungen und entsprechend die Bestimmung des besten Wegs (siehe: Auswahl der Partitionen) bei der Partitionierung der Tabelle mit Frequency Partitioning. Danach wird ein separates Dictionary von Werten für jede Partition erzeugt. Dann werden diese Dictionaries benutzt, um letztendlich die Tupel (der Tabelle) vom Input in die geeignete Zelle einzuordnen, und um die Felder jedes Tupels zu codieren und zu einem Tupelcode zu konkatenieren. Am Ende werden die Tupelcodes innerhalb jeder Zelle sortiert und nochmal delta-codiert. Als nächstes wird kurz beschrieben, wie die Analyse der Spalten-Verteilungen und das entsprechende Frequency Partitioning erfolgen. [55]

Auswahl der Partitionen Das Erreichen einer Spitzen-Kompression erfordert eine Spaltenpartitionierung, welche die durchschnittliche Tupelcodelänge minimiert. Dies kann dadurch erreicht werden, dass sich Werte mit einer ähnlichen Häufigkeit in derselben Partition befinden, und dass häufige Werte zu Partitionen mit kurzen Codelängen zugeordnet und weniger häufige Werte zu Partitionen mit längeren Codelängen zugeordnet werden. Die wichtigste Beschränkung bei dieser Optimierung ist die Anzahl der resultierenden Zellen. Wie bereits erwähnt, ist der Hauptzweck von Frequency Partitioning das Amortisieren der Codelängenberechnung über alle Tupel hinweg in einer Zelle. Wenn es zu viele Zellen gibt, wird jede Zelle nur wenige Tupel beherbergen und diese Amortisierung schlägt fehl.
Das Problem der Auswahl von Partitionen für eine Spalte und über mehrere Spalten hinweg, ist ein Optimierungs-Problem und zur Lösung werden Dynamische Programmierung und ein Greedy-Algorithmus eingesetzt. Die Details sind für diese Arbeit nicht von Relevanz und bei Interesse in [55] nachzulesen. [55]

Speicherung Das hier betrachtete Blink-System wurde 2008 als eine klassische Row-Store Datenbank in [55] vorgestellt. Später jedoch, im Jahre 2012, wurde eine weiterentwickelte Version von Blink in [13] vorgestellt, mit einem Speicherformat namens *Banked-Layout*. Im Prinzip werden bei diesem Layout mehrere Spalten einer Tabelle zu einer Bank zusammengefasst gespeichert. Diese Banks sind bitorientiert, d.h. sie nehmen nur die Größen 8, 16, 32 oder 64 Bit ein, notfalls mit Padding. Das hat den Effekt, dass mehrere solcher *Tuplets* später bei der Ausführung in der ALU zusammengefasst und parallel ausgeführt werden können. [55]

Kompressionseffizienz Die Frage nach der Effizienz, ist die Frage danach, wie nah die Kompression der Entropie kommt, daher dem mittleren Informationsgehalt. Falls jede einzelne Zelle bis zur Entropie komprimiert wird, dann komprimiert auch die Gesamtmenge der Daten nah zur Entropie. Blink komprimiert aber nicht jede Zelle bis zur Entropie, weil es Codes fester Länge in jeder Zelle benutzt (kleinere Codes werden mit Nullen aufgefüllt). Falls in einer genug feinen Granularität partitioniert wird, daher genug Zellen benutzt werden, kann der Unterschied bei den Häufigkeiten der versch. Werte innerhalb der Zelle unbedeutend gemacht werden, und die Codierung in fester Länge wäre ausreichend für eine gute Kompression (bis zur Entropie). Jedoch ist die Anzahl der Zellen die benutzt werden dürfen, beschränkt, wie oben erwähnt. Als nächstes werden die Auswirkungen dieser Abwägung untersucht.



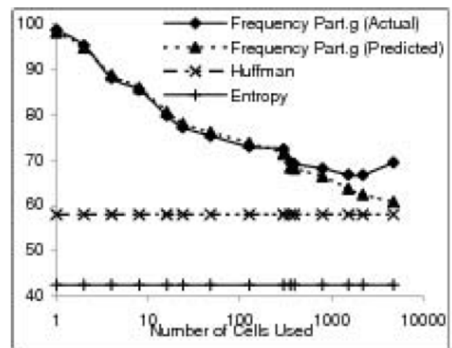


Abbildung 2.40: Kompressions-Ergebnisse für TPCH (links) und Census (rechts). [55]

Abbildung 2.40 zeigt die Anzahl der Bits pro Tupel bei wachsender Anzahl von benutzten Zellen für zwei Datensätze: Der erste Datensatz ist eine unkorrelierte, aber verschrägte Version von TPCH. Eine genauere Beschreibung des Datensatzes steht in [55]. Der zweite Datensatz stammt von census.gov, genauere Beschreibung auch in [55]. Wir können beobachten, dass in beiden Fällen eine Partitionierung zu ein paar Tausend Zellen, nahezu dieselbe Kompression liefert wie Huffman-Codes. Wir können auch beobachten, dass das Benutzen von mehr Zellen generell die Kompression verbessert, obwohl ganz am Ende des Census-Datensatzes, die Kompression nicht mehr besser wird, aufgrund des Overheads, der durch die Benutzung von zu vielen Zellen ausgelöst wurde. [55]

Anfrageverarbeitung

Der Kern von Blinks Anfrageverarbeitung ist der generalisierte Scan-Operator. Ein Modul, das die Funktionalität von Scan, Selektion, Grouping und Aggregation kombiniert. Abbildung 2.41 zeigt einen Überblick. Der Scan erhält einen Puffer (work queue) aufrecht, in dem jeder Eintrag einen Block komprimierter Tupel aus einer einzigen Zelle vom Input darstellt. Ein Pool von Threads (worker threads) konsumiert diese Blocks und produziert partielle Anfrage-Ergebnisse, welche am Ende zu einem kompletten Ergebnis gemerged werden.
Jeder Thread startet mit der Ermittlung der *Codelängen* von jedem Feld und des jeweiligen *Dictionaries*, welche für die Codierung der Felder benutzt wird. Dies wird nur einmal pro Zelle durchgeführt. Anschließend werden für alle Tupel der Zelle folgende drei Schritte durchgeführt: Auflösung des Deltacodes (Undo Delta Coding), Selektion, Grouped-Aggregation.
Das Decodierung des Deltacodes ist trivial und hier nicht von Interesse. Die nächsten beiden Schritte sind interessanter und mehr von Bedeutung für diese Arbeit.

Selektion: Parallele Equality- und Rangeprädikate Die meisten modernen Prozessoren besitzen 128-Bit Register, auf welche man bitweise Operationen (AND, OR) oder auch SIMD-Versionen arithmetischer Operationen anwenden kann. Nach der Kompression passen Tupelcodes generell in eine kleine Anzahl

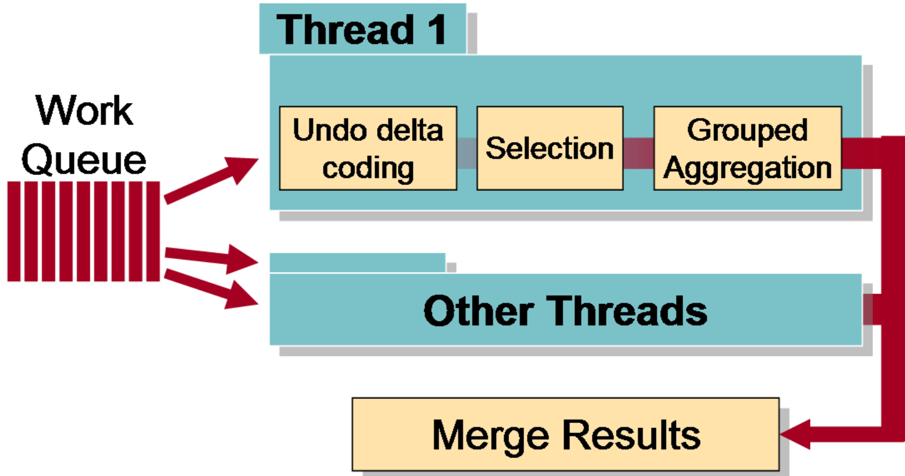


Abbildung 2.41: Der Scan-Operator bekommt einen Puffer von Zellen komprimierter Tupel, und verarbeitet diesen mit einem Pool von Threads. [55]

dieser Register. Laut [55] sind diese Codes in der Regel kleiner als eine Registerbreite, dürfen jedoch maximal zwei Register (256 Bit) in Anspruch nehmen. Ab hier wird für eine vereinfachte Präsentation angenommen, dass sich der Tupelcode in genau einem Register befindet.
Die Evaluierung einer Verundung von Equalityprädikaten auf einem Tupelcode, welcher komplett in einen Register passt, ist einfach. Extrahiere zuerst die benötigten Felder durch eine passende Maske (ein bitweises AND). Vergleiche dann das Ergebnis mit einer zweiten Maske, die die erwarteten Feldcodes an den entsprechenden Feldoffsets enthält. Beachte, dass diese Masken nur genau einmal pro Zelle berechnet werden müssen. Rangeprädikate sind schwieriger parallel zu evaluieren, weil sie nicht zu einem einzigen Vergleich konvertiert werden können. Betrachte eine Konjunktion von Prädikaten der Form $col > literal$. Man könnte die benötigten Felder mittels eines AND extrahieren und dann eine Maske subtrahieren, die die Literale an den richtigen Offsets enthält. Wenn das Ergebnis auf jedem Feld positiv ist (überprüft mittels einer Maske auf den Bits der höherwertigen Bits), dann erfüllt das Tupel alle Prädikate. Unglücklicherweise funktioniert dieser naive Ansatz nicht. Wenn eine Subtraktion ein Feld negativ werden lässt, dann leiht der Prozessor von den höherwertigen Bits des Registers, und verändert dabei die Werte von anderen Feldern. Dieses Bit wird in [55] als *Borrow* bezeichnet. Die Lösung ist es, ein Borrow-Stopper-Bit links von jedem Feld zu platzieren, das in dem Prädikat involviert ist. Um Platz für diese Borrow-Stopper freizumachen, werden die ungerade nummerierten Felder separat von den gerade nummerierten Feldern verarbeitet. Wenn die geraden Felder verarbeitet werden, wird der Platz der ungeraden Felder benutzt, um die Borrows abzufangen, und andersherum. Das linke Bit wird frei gelassen, um die Borrows des linken Felds abzufangen. In [55] gibt es ein Beispiel dazu. [55]
Für andere Prädikate als Equality und Range war die Standardlösung, erst zu

decodieren und dann das Prädikat anzuwenden. Blink benutzt eine alternative Implementierung, die keine Decodierung beinhaltet. Betrachte z.B. ein LIKE-Prädikat auf Spalte C. Die Idee ist, das Dictionary auf C als eine implizite Dimension-Table zu betrachten. Am Anfang der Anfrage wird das Prädikat über das Dictionary evaluiert, die betreffenden Codes werden identifiziert und in einer Hash-Tabelle platziert. Während des Scans wird das LIKE-Prädikat durch Hashing auf dem Tupelcode evaluiert, nach dem Wegmaskieren aller Spalten außer C. [55]

Grouping und Aggregation Der nächste Schritt ist die Anwendung von Group-By durch das Updaten eines geeigneten laufenden Aggregats für jedes Tupel, welches alle Prädikate erfüllt. Die Aggregat-Werte in dem aktuellen Tupel werden durch die Extrahierung und Decodierung der Feldcodes von Aggregations-Feldern geformt. Die Feldcodes für alle Group-By-Spalten werden extrahiert und zu einem verpackten *Groupcode* konkateniert (durch ANDs und Shifts).
Informell werden die laufenden Aggregate in einer Hash-Tabelle aufrechterhalten. Somit umfasst ein Update:

$$aggTable[hash(group)] += aggregatevalue(s) \tag{2.1}$$

für einen geeignet definierten +=-Operator.
Insbesondere müssen zwei Aspekte beachtet werden, um dies effizient zu tun:

- Die Hash-Tabelle muss klein gehalten werden, so dass es in den L2-Cache passt.
- Für einen schnellen Hash-Tabellen-Lookup, muss der wahlfreie Zugriff und das bedingte Springen (Branching) verhindert werden.

In Blink werden drei Hashing-Techniken benutzt, welche in verschiedenen Situationen anwendbar sind, abhängig von der Anzahl an verschiedenen Gruppen. Diese Hashing-Techniken sind, im Gegensatz zur Granularität der Gruppierung, für diese Arbeit nicht relevant. [55]

Drawers als Granularität für die Gruppierung Angenommen eine Anfrage gruppiert auf Spalten H und G, welche die Spaltenpartitionen G1, G2, H1 und H2 besitzen. Die Tabelle könnte zahlreiche Zellen besitzen, von dem Kreuzprodukt der Partitionen aller Spalten (nicht nur G und H). Dann existieren drei Granularitäten, in denen gruppiert werden kann.
Erstens, kann man eine separate *AggTable* für jede Zelle aufrechterhalten. Dies resultiert in einer kleinen AggTable, weil es nur wenige verschiedene Gruppen innerhalb einer Zelle gibt. Da die Hash-Tabellen nun pro Zelle vorliegen, müssen diese kombiniert werden, um ein Endergebnis über alle Daten zu ermitteln.
Zweitens, könnte man eine einzige AggTable für die gesamte Anfrage aufrechterhalten. Dies resultiert aber in einer sehr großen Hash-Tabelle (so groß, wie viele verschiedene Gruppen es in der gesamten Tabelle gibt), welche nicht in das L2-Cache passen könnte. Dieser Ansatz wäre auch schwierig zu parallelisieren, weil die Zugriffe aller Threads auf die Hash-Tabelle synchronisieren werden müssten. Die Entwickler von Blink haben sich für eine dritte Option namens *Drawer* entschieden, die zwischen diesen Extremen liegt. Ein Drawer ist definiert durch die Partitionierung entlang der Group-By-Spalten:

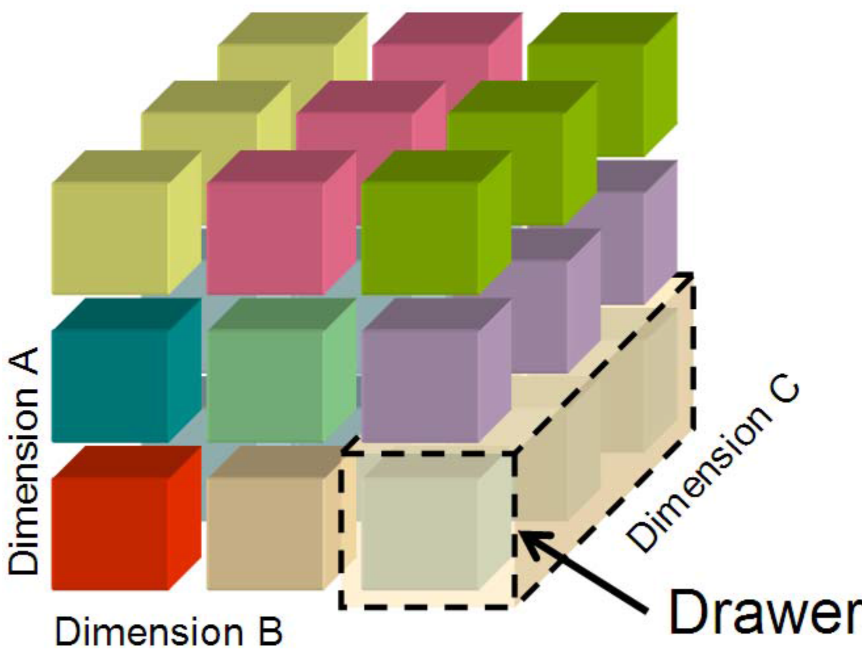


Abbildung 2.42: Beziehung zwischen Drawers und Zellen. [55]

Ein Drawer ist eine Kollektion von Zellen, die von einer einzigen Partition der Group-By-Spalten kommen.
Abbildung 2.42 zeigt eine Tabelle, die auf den Spalten A, B und C frequency-partitioniert wurde. Das gestrichelte Rechteck zeigt einen Drawer für eine Anfrage, die auf A und B gruppiert. Beachte, dass anders als Zellen, Drawers für eine bestimmte Anfrage definiert sind. Drawers haben zwei wichtige Eigenschaften, welche sie zu der richtigen Granularität machen, um AggTables auf ihnen zu berechnen:

1. Für alle Zellen innerhalb eines Drawers werden die Feldcodes für jede Grouping-Splate von einer einzigen Dictionary gezogen. Weil Dictionaries eine 1-zu-1-Abbildung von Werten zu Feldcodes sind bedeutet dies, dass Gleichung 2.1 ersetzt werden kann durch:

$$aggTable[hash(groupcode)] += aggregate \tag{2.2}$$

2. Jede Gruppe kann nur in einem einzigen Drawer vorkommen. Somit kann die aggTable drawer-spezifisch sein und muss nur so viele Einträge halten, wie viele Gruppen in einem einzigen Drawer sind. Diese Anzahl ist oft viel kleiner als die Gesamtzahl der verschiedenen Gruppen in der Tabelle.

Für jeden Drawer werden AggTables unabhängig voneinander berechnet. Am Ende werden die Listen von *(groupcode,aggregate)*-Paaren vereinigt. Weil jede Gruppe nur in einem Drawer vorkommt, ist dies eine triviale Operation.
Verschiedene Drawers können eine weitgehend voneinander verschiedene Anzahl

von Gruppen enthalten, weil die Spalten durch ihre Häufigkeit partitioniert werden. Weil eine separate aggTable für jeden Drawer benutzt wird, kann auch die Hashing-Technik für jeden Drawer unabhängig gewählt werden, gemäß ihrer Anzahl an verschiedenen Gruppen, abgeschätzt anhand der Dictionary. [55]

Performance

Ein Prototyp von Blink implementiert alle Merkmale und Eigenschaften, die in dieser Arbeit vorgestellt wurden. Blink komprimiert Daten durch *Frequency*-Partitionierung und Deltakodierung innerhalb von Zellen. Blink hält diese Daten im Hauptspeicher, um Anfragen auszuführen. Blink unterstützt Single-Block SQL-Anfragen mit Equality-, Range- und In-List-Prädikaten, SUM- und COUNT-Aggregaten und Gruppierung. Im Folgenden werden bestimmte Aspekte von Blink experimentell evaluiert. Der verwendete Datensatz basiert auf TPC-H und ist in [55] genauer beschrieben. [55]

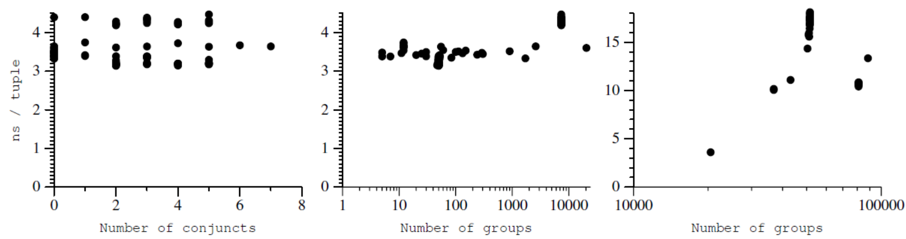


Abbildung 2.43: Scan-Geschwindigkeit für beliebige Single-Block Anfragen (ca. 100%-ige Selektivität der Prädikate). [55]

Variabilität der Antwortzeiten für Anfragen Bei den Experimenten in [55] wurden 150 Anfrage der Form

SELECT SUM(revenue) FROM denormalized **table**
WHERE <conjunction of predicates>
GROUP BY <**column** list>

durchgeführt. Wie man erkennen kann, unterscheiden sich die Anfragen nur in den Prädikaten, die benutzt wurden und in den Spalten, auf denen gruppiert wird. Eine genauere Beschreibung befindet sich in [55]. In Abbildung 2.43 zeigt die Performance für Anfragen mit 0 bis 7 Prädikaten und die Performance für Anfragen, die bis zu 20.000 Gruppen produzieren (bzw. 100.000 ganz rechts). Die Antwortzeiten bleiben bei 3.1 bis 4.5 Nanosekunden pro Tupel, sowohl bei steigender Anzahl der Prädikate, als auch bei steigender Anzahl der Gruppen. Nur ab ca. 20.000 Gruppen degradiert die Performance rapide auf bis zu ca. 18 Nanosekunden pro Tupel, weil die laufenden Aggregate nicht mehr in den L2-Cache passen.
Abbildung 2.44 zeigt die Laufzeiten für verschiedene Zell-Selektivitäten. Dies ist deshalb wichtig zu sehen, weil Blink einen Filter-Effekt auf Zellebene hat: Ganze Zellen können vom Scan eliminiert werden, wenn kein Wert in des jeweiligen Dictionaries das Prädikat erfüllt. Auch bei verschiedenen Zell-Selektivitäten bleibt die Ausführungszeit unter 4,5 Nanosekunden pro Tupel und es ist definitiv

zu erkennen, dass die Geschwindigkeit beschleunigt, wenn die Selektivität sinkt. [55]

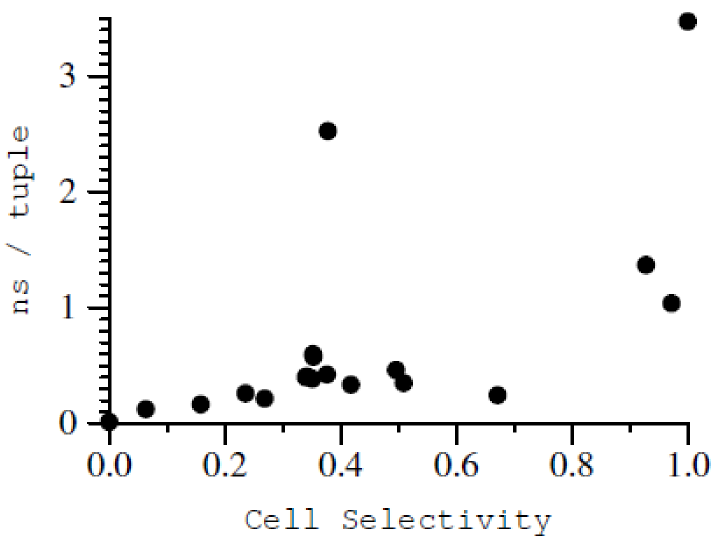


Abbildung 2.44: Scan-Geschwindigkeit für selektive Anfragen. [55]

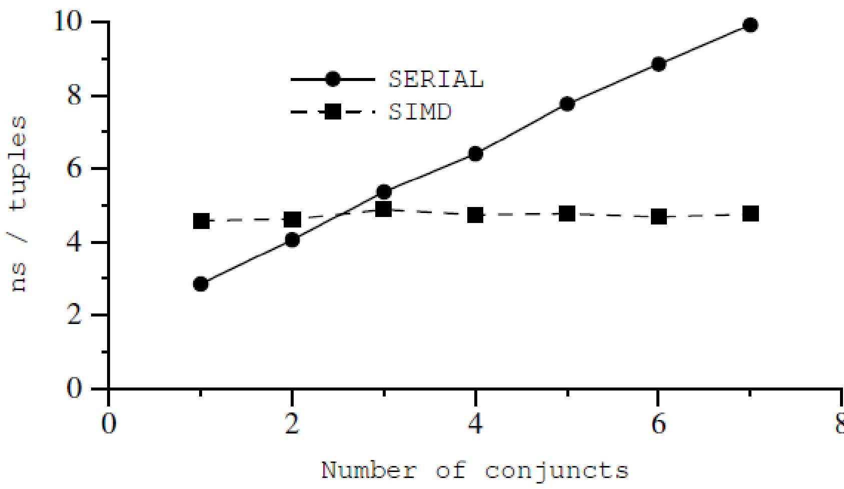


Abbildung 2.45: Parallele Prädikat-Evaluation. [55]

Ersparnisse durch SIMD-Prädikate Bei diesem Experiment werden 7 Versionen der folgenden Anfrage ausgeführt, wobei bei jedem (Ausführungs-) Schritt ein Prädikat aus der Konjunktion der Prädikate entfernt wird.

```
SELECT SUM(revenue) FROM denormalized table
WHERE QTY < 50 AND month ≤11
AND l_price > 1 AND year ≥ 1995 AND l_partkey ≥ 1
AND revenue ≥ 1.0 AND week ≥ 1
GROUP BY month
```

Abbildung 2.45 zeigt die Ausführungszeit für die Anzahl der Prädikate in der Konjunktion. *SIMD* ist die standard Blink-Implementierung, welche alle Prädikate parallel evaluiert. SERIAL ist eine typische Implementierung, welche die Prädikate in einer typischen for-Schleife evaluiert. Auffällig ist, dass die Ausführungszeit für die parallele Implementierung fast konstant ist, während die Ausführungszeit für die serielle Implementierung linear steigt. [55]

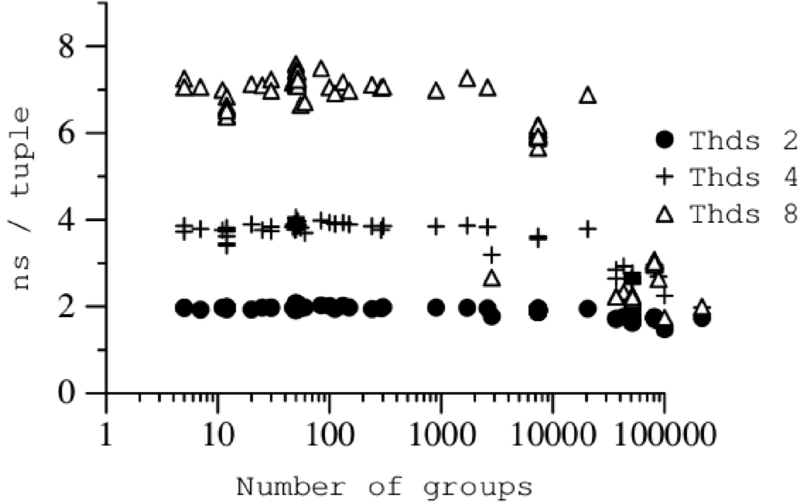


Abbildung 2.46: Multicore Beschleunigung. [55]

Das letzte Experiment untersucht die Multicore-Skalierbarkeit von Blink, wobei dafür die ursprünglichen 150 Anfragen benutzt wurden. Abbildung 2.46 zeigt die Beschleunigung, erreicht durch die Aufteilung in 1, 2, 4, und 8 Threads. Die Speedups für 2 und 4 Threads sind fast ideal und die meisten Anfragen erreichen mit 8 Threads einen Speedup von 7x. [55]

Evaluation

Frequency-Partitioning an sich wäre nicht sehr kompliziert umzusetzen, zumal die ganze Komplexität ja auch im Parsing steckt, was nur einmal ganz am Anfang durchgeführt werden muss, bevor man ganze Tupel als Zellen abspeichert und dann wieder zur Verwendung laden kann. Allerdings fängt es an kompliziert zu werden, wenn man Update-, bzw. Delete/Insert-Operationen hinzu implementieren will. Das Blink-System, oder zumindest das Paper [55], behandelt nur lesende Zugriffe. Die häufigkeitenbasierte Partitionierung (Frequency-Partitioning) soll eine gute Kompression liefern, um die Speicherbandbreite möglichst effizient ausnutzen zu können, und das tut sie auch, wie die Performance von Blink zeigt. Das Problem ist jedoch, dass wir in dieser Projektgruppe, nicht von Anfang an

ein hoch ausgeklügeltes System entwickeln können, da es uns an Erfahrung in dem Bereich Datenbankentwicklung fehlt. Daher wollen wir in zwei Schritten vorgehen, und erst ein funktionierendes und weniger ausgeklügeltes Datenbanksystem entwerfen und im Nachhinein das System optimieren. Aus diesem Grund haben wir uns nicht für ein Konzept, wie das Frequency-Partitioning entschieden. Allerdings werden wir in der Optimierungsphase der PG, mindestens eine Kompressionstechnik implementieren. Dictionary-Kodierung ist aufgrund ihrer Einfachheit sehr attraktiv für uns und wird in der zweiten Phase der PG mit großer Wahrscheinlichkeit implementiert. Ein weiteres Ziel von Frequency-Partitioning ist es, das Berechnen von Feldcodelängen zu amortisieren. Wie man weiter unten lesen wird, haben wir uns bei dieser PG für eine spaltenorientierte Speicherung und vektorisierte Verarbeitung entschieden, weil es relativ einfach umzusetzen ist und trotzdem modern und effizient ist. Bei einem solchen System, fällt das Berechnen von Feldcodelängen weg, und somit ist das Konzept Frequency-Partitioning auch aus dieser Sicht komplett uninteressant für uns. Der Einsatz von SIMD-Technologie für das parallele Evaluieren von Equality- und Rangeprädikaten, ist im Grunde eine gute Idee, lässt sich jedoch in der oben vorgestellten Tupel orientierten Form, nicht für unser Projekt anwenden. Mitnehmen können wir, dass man per SIMD, bei der Auswertung einer einzigen Operation auf vielen Daten, die Ausführungszeit deutlich beschleunigen kann. Die Konzepte für Grouping und Aggregation, lassen sich auch nicht für unser Projekt anwenden, weil sie Tupel orientiert sind und auf Frequency-Partitioning basieren. Allgemein, kann man aber mitnehmen, dass es sich lohnen kann, z.B. laufende Aggregate cache-bewusst zu implementieren und zu pflegen, und verschiedene Hashing-Varianten für verschiedene Anzahlen von Gruppen auszuwählen.

2.5.2 DB2 BLU *(Milad Nayebi)*

Überblick

DB2 bietet mit BLU viele Beschleunigungstechnologien, die für große Datenmengen geeignet sind. Terabyte-große Daten werden durch mehrfaches, intelligentes Sortieren und Komprimieren auf ein Minimum reduziert. Davon profitieren nicht nur massiv lesende Zugriffe, sondern auch typische Data-Warehouse-Anwendungen. Wenn Datenmengen sehr groß sind, benötigen sie neue Technologien. Traditionelle Methoden versagen an Terabyte und Petabyte von Daten. In der neuen Version des Datenbankservers DB2 mit BLU Acceleration verwendet IBM eine Technologie namens BLU (Blink Ultimate), um auf Daten von analytischen Systemen In-Memory zuzugreifen.
BLU geht auf das Projekt Blink des IBM-Forschungszentrums in Almaden zurück. BLU kombiniert viele Technologien:

- In-Memory-Computing
- Spaltenorientierte Datenbank-Tabellen
- Data Skipping
- SIMD (Single Instruction Multiple Data) und aus dem Supercomputing stammendes Vektorrechnen.

Mit diesen Methoden soll die Datenverarbeitung und -analyse deutlich beschleunigt werden.

Die Daten werden dabei nicht reduziert, sondern nur sortiert und komprimiert. "Die eigentliche Datenmenge bleibt auf dem ursprünglichen Speichersystem erhalten", sagt Holm Landrock von der Experton Group in einem Review des neuen IBM-Systems. "Der Clou ist die mehrfache Datenkomprimierung auf dem Weg zum SQL-Befehl. Die Datenkomprimierung findet im Speicher und in der Analyse statt."
BLUs Grundziele:

- Halten der Daten möglichst effizient im Hauptspeicher
- Spaltenorientiertes Abgelegen der Daten in den Tabellen und im Speicher

Data layout and Compression

Die Daten werden grundsätzlich spaltenorientiert im Hauptspeicher abgelegt. Damit ist es bei bestimmten Operationen deutlich schneller als die bisherigen zeilenorientierten DB2-Systeme. Wenn etwas in spaltenorientierten Systemen abfragen wird, müssen nicht ganze Zeilen, sondern nur die für die Auswertung notwendigen Spalten durchsucht werden. Dieses nur scheinbar nebensächliche Implementierungsdetail hat zur Folge, dass sich gewisse Arten von Abfragen effizienter abarbeiten lassen. Das sind im Wesentlichen Aggregationen von Werten über viele Zeilen hinweg, wie sie in Data-Warehouses und anderen analytischen Applikationen häufig vorkommen. Aufgrund eines einheitlichen Typs von Spaltendaten stehen zusätzlich in spaltenorientierten Systemen Komprimierungsoptionen zur Verfügung, die bei zeilenorientierten Daten nicht möglich waren. Wenn die Tabellen in Spalten konvertiert werden, ist BLU deshalb in der Lage, die Daten gleichzeitig per Bitverschlüsselung zu komprimieren.

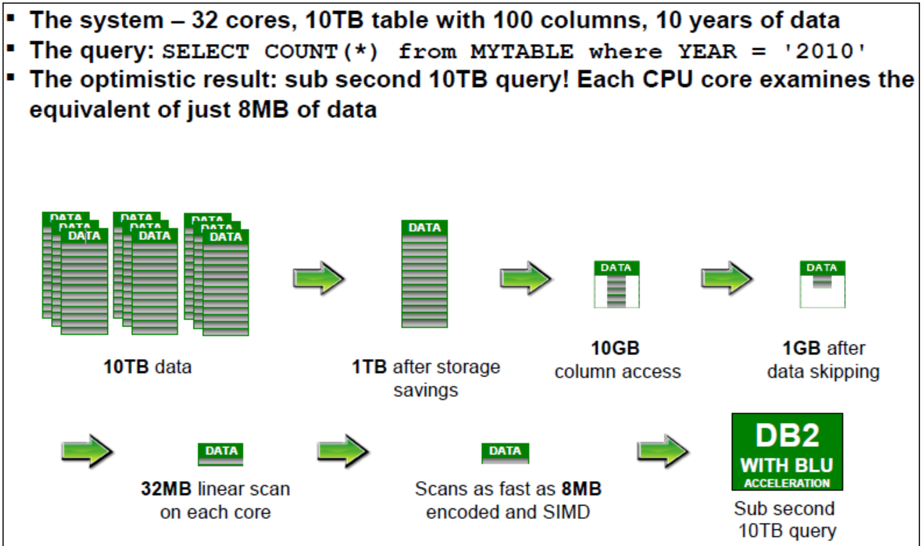


Abbildung 2.47: Das Beispiel zeigt, wie BLU-Technologien 10 TB große Daten auf 8 MB reduzieren können.

Die spaltenorientierte Speicherung der Daten in den Tabellen erzielt eine zusätzliche Komprimierung um etwa Faktor 5 bis 10. Laut IBM haben erste Erfahrungen

bei Kunden gezeigt, dass die Dauer der Ausführung von langlaufenden SQL Anfragen um Faktor 25 - 40 verkürzt werden konnte.

On-disk format Jede Spalte kann mehrere Kodierungsschemas haben (z.B. short-code-word für häufige Daten und larger-width code-words für seltene Werte. Entscheidend ist, dass code-words in einem Schema konstante Größe haben, siehe Figure 2.48).

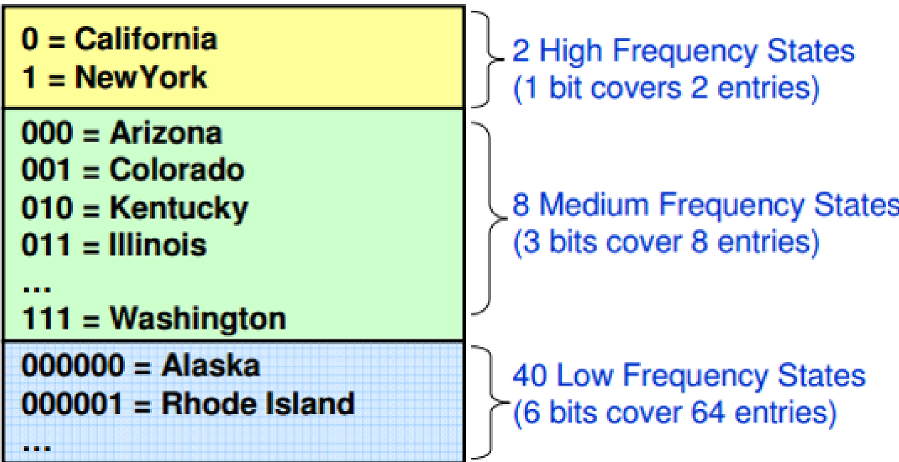


Abbildung 2.48: short code-word und larger-width code-words

Die Spalten werden in Spaltengruppen eingeteilt. Alle Werte aus der gleichen Spaltengruppe werden zusammen gespeichert. Spaltengruppen werden in Seiten gespeichert und eine Seite kann eine oder mehrere Regionen enthalten. Eine einzige Region hat ein konstantes Kompressionsschema für alle darin gespeicherten Spalten. Innerhalb einer Region werden die einzelnen Spalten in data-banks fester Größe gespeichert. Schließlich enthält jede Seite eine Tuple Map, mit der sich die Tupel den Regionen zuordnen lassen. Wenn es zwei Regionen in einer Seite gibt, nutzt die Tupel Map ein Bit pro Tupel (Tupel-Map: 101001000... die 1., 3. und 6. Einträge sind in der ersten Region). Jede Seite enthält eine zusammenhängende Reihe von "tuplets"(Projektion eines Tupels auf eine Spaltengruppe).
Alle Spaltengruppen werden durch die gleiche TSN "Tuple Sequence Number"geordnet(siehe Figure 2.49). Das Mapping von TSN zu den Seiten wird als B+-Tree gespeichert. Nullable Spalten werden mit einer zusätzlichen „1-Bit nullable“ Spalte behandelt. Jede Spalte hat eine Synopsis Tabelle, die für das page-Skipping verwendet wird, und in dem gleichen Format wie eine normale Datentabelle gespeichert ist.

Buffer Management and Caches

BLU(Die zweite Generation von Blink) arbeitet auch beim Storage deutlich effizienter:

- Data-Skipping
- Single-Instruction-Multiple-Data (SIMD)

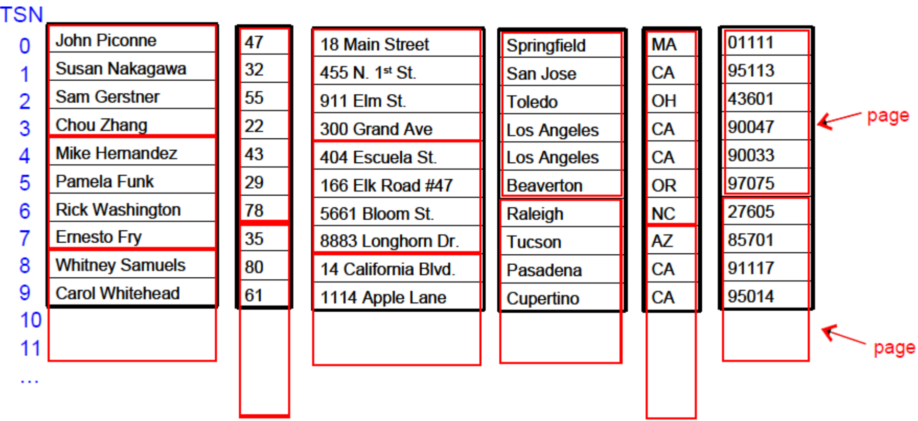


Abbildung 2.49: Die Grafik zeigt Tuple Sequence Number

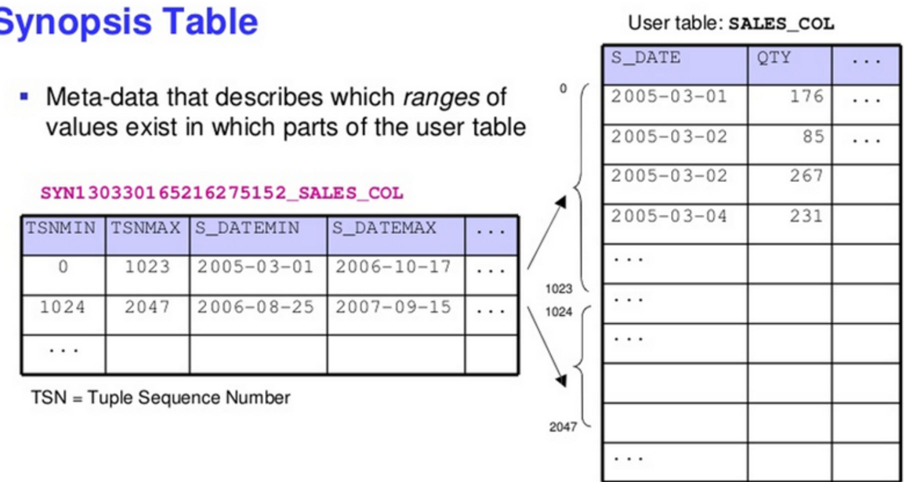


Abbildung 2.50: Synopsis Table

- Mittels Smart-Caching und einiger weiterer Funktionalitäten

erfolgt die Auswertung der Daten hardwarenah In Memory und speichereffizient(siehe Figure 2.51).
Beim Smart Caching werden die Daten in der bitorientierten, komprimierten Form im L1/L2-Cache gehalten, also im Pufferspeicher mit den schnellen Zugriffszeiten. Die CPU liest und verarbeitet dann die Daten direkt von diesem schnellen Speicher aus. Weil die Daten jetzt ganz nah am Prozessor gehalten werden, müssen keine Zugriffe in das weit entfernte RAM durchgeführt werden, und viel Zeit wird dadurch eingespart.
Mit der Einführung von SIMD-Units in den neueren x86- und Power-CPUs, wurden auch die neuen Beschleunigungs-Technologien von BLU ermöglicht. SIMD verarbeitet mit einem einzigen Maschinenbefehl mehrere Daten gleichzeitig in der

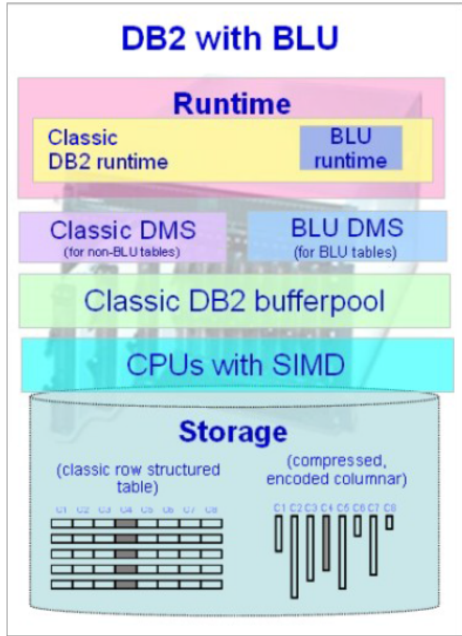


Abbildung 2.51: Smart Caching und SIMD werten Daten hardwarenah In Memory und speichereffizient aus

CPU. Die Performance von DB2 wird durch diese Abarbeitung einer Vielzahl von Daten mit einer einzelnen Instruktion weiter erhöht. BLU hat zudem einen neuen Bufferpool vorgestellt. Dieser neue Bufferpool hält die spaltenorientierten Daten komprimiert vor. Die Daten werden durch einen effizienten, neuen Algorithmus gecacht. Der Algorithmus cacht nur die für die Applikationen relevante Spalten, nicht die gesamte Tabellen. Als Beispiel:
SELECT columnname FROM tablename
Bei dieser Anfrage werden nur die Seiten gepuffert, die die Daten der Spalte columname halten (und nicht der gesamte Tabelleninhalt).
"Das ist ein sehr großer Unterschied im Vergleich zum Buffer-Pool-Zugriff bei den zeilenorientierten Tabellen", sagt Stefan Hummels, Senior IT Spezialist und DB2-Experte bei IBM. "DB2 hat mit BLU eine sehr effiziente Speicherung der Daten im Cache und kann ihn deutlich besser nutzen. Das ist auch ein Grund dafür, dass DB2 BLU nicht so viel Hauptspeicher braucht, wie die Datenbank groß ist."

Beschleunigung und Administration

Im Vergleich von DB2 BLU mit DB2 10.1. zeigten sich bei Kunden deutliche Performance-Steigerungen in den Datenbank-Zugriffen. Im Durchschnitt wurden die Zugriffe um das 10- bis 25-fache beschleunigt.
Die obige Grafik präsentiert das Schrumpfen der Tabellen bei der Umwandlung von Zeilen in Spalten. Dabei gilt grundsätzlich: Je größer die Tabelle, desto größer der Nutzen der Spaltenorientierung. Ein durchschnittlicher Komprimierungsfaktor von 10, wie es die Beispiele nahe legen, bedeutet, dass eine ursprünglich 1

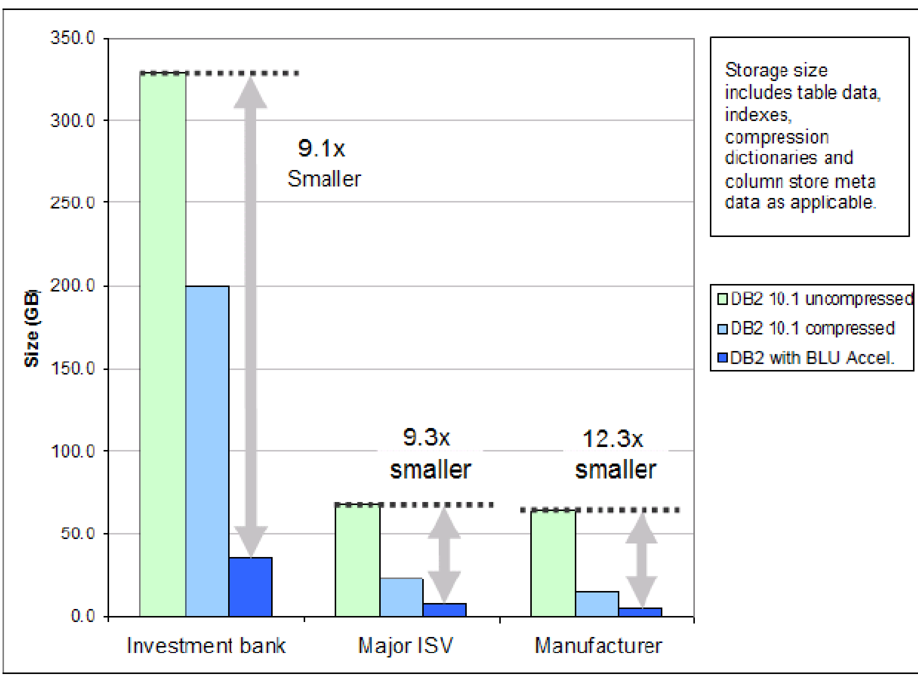


Abbildung 2.52: Schrumpfen der Tabellen bei der Umwandlung von Zeilen in Spalten

Terabyte große Tabelle auf 100 GB schrumpft. Dadurch ergeben sich entsprechend bessere Antwortzeiten. Die Arbeit für den Datenbankadministrator wird deutlich vereinfacht. Die Datenbankadministratoren müssen das Partitionieren der Tabellen und die Komprimierung der Daten in klassischen Data-Warehouse-Systemen festlegen. Sie müssen Tabellen anlegen, Daten laden und in einem immer wiederkehrenden Prozess Indizes verwalten, die Datenbank tunen und Statistiken erstellen. In BLU entfällt das gesamte Tuning und damit alles bis auf das Erzeugen von Tabellen und das Laden der Daten, denn es gibt in spaltenorientierten Tabellen keine Indizes. Insofern müssen die Datenbankadministratoren beim Datenbankdesign und Tuning nur über *Partition strategies, Select Compression Strategy, Create Auxiliary Performance Structures und Materialized views* entscheiden. Dagegen entfällt: *Create indexes, B+ indexes, Bitmap indexes, Tune memory, Tune I/O, Add Optimizer hints und Statistics collection.*

Der Workload Manager

Die Anfragen an den Datenbank-Server steuert in DB2 BLU, wie in früheren Versionen, der Workload Manager. Das Tool ist bei analytischem Workload immer automatisch eingeschaltet und arbeitet unter anderem die von den Applikationen an die Daten geschickten Anfragen in koordinierter Form ab.

Scans, Joins, Aggregationen

Scans Es gibt zwei Varianten des Zugriffs der Spalten, die auf der Festplatte abgespeichert sind. Die erste Variante, "LEAF"wertet Prädikate über Spalten in

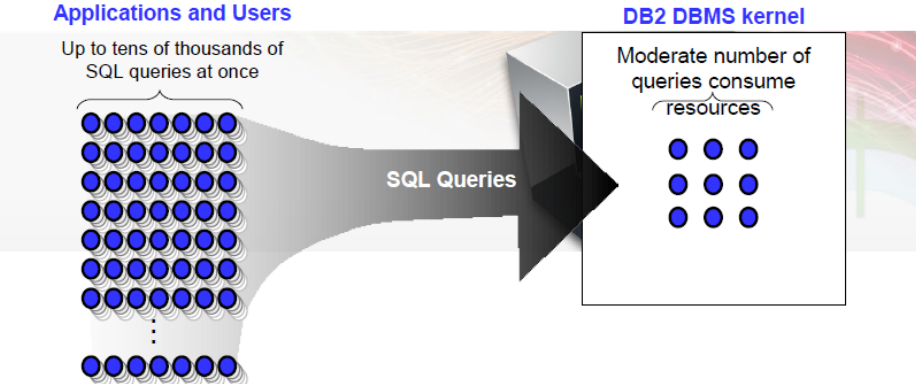


Abbildung 2.53: Der Workload Manager verteilt die Queries so auf die Cores, dass alle Kerne gut ausgelastet sind.

einer Spaltengruppe aus. Die Auswertung durch LEAF ist Region-By-Region. LEAF stellt eine Bitmap mit einer 1 für jedes Tupel her, welches die Auswertung für jede Region bestanden hat. Die Bitmaps werden anschließend mit etwas Bit-Twiddling-Magie verschachtelt, um eine Bitmap in TSN-Ordnung zu produzieren. Es ist nicht klar, ob der einzige Ausgang von LEAF die Validitäts-Bitmap ist. Die Auswertung der Prädikate kann über codierte Daten mit Hilfe der Verwendung von SIMD-effizienten Algorithmen durchgeführt werden. Der zweite Spaltenzugriffsoperator, "LCOL", lädt Spalten entweder in codierter oder uncodierter Form. Die codierten Werte können direkt in Joins und Aggregationen zugeführt werden (Dekomprimierung nicht zwingend notwendig). LCOL berücksichtigt ein Validitäts-Bitmap als Parameter und erzeugt eine Ausgabe, die nur dann gültige Tuplets enthält.

Joins Joins folgen einem typischen (Build-Probe) Zweiphasenmuster. Beide Phasen sind stark multithreaded. In der Build-Phase werden Join Keys partitioniert. Jeder Thread teilt eine separate Untergruppe der gescannten Zeilen (durch die zusammenhängende Tuple Sequence Number). Dann baut jeder Thread eine Hash-Tabelle von einer Partition. Partitionen werden nach Speicher bemessen (zum Halten einer Partition in einer einzigen Ebene der Speicherhierarchie). Joins werden auf kodierte Daten angewendet. Da die Join-Keys der äußeren und inneren Relation unterschiedlich kodiert sind, wird der innere Key auf das Kodierungsschema des äußeren Keys umkodiert.

Zur gleichen Zeit wird ein Bloom-Filter auf den Join Keys der inneren, und später auf der Join Keys der äußeren Relation erstellt. Das Gegenteil ist ebenfalls möglich, wenn beispielsweise die innere Relation sehr groß ist, wird die äußere einmal gescannt, um den Bloom-Filter zu berechnen. Dieser zusätzliche Scan soll einem SSpill-To-Disk"vorbeugen, wenn die innere Hash-Tabelle groß wird. Manchmal ist ßpillingünvermeidbar, in diesem Fall werden entweder nur einige Bereiche des inneren und äußeren gespilled, oder in Abhängigkeit von einem Kostenmodell nur die innere gespilled.

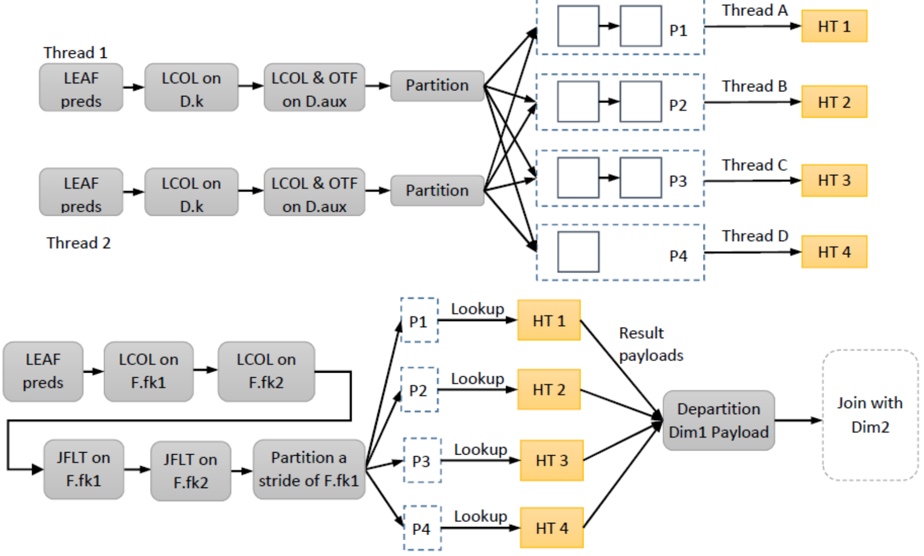


Abbildung 2.54: Top: Evaluator sequence for build of join with 4 partitions. Bottom: Evaluator sequence for probe of join. JFLT stands for join filter. Observe that join filters of all dimensions are applied before any joins are performed. [56]

Aggregationen Aggregationen bestehen wie die Joins aus Zwei-Phasen. Jeder Thread erstellt eine eigene lokale Aggregation Hash-Tabelle, die auf einem Teil des Eingangs basiert. In der zweiten Phase werden die Ergebnisse kombiniert: Jeder Thread nimmt einen Teil der kombinierten Hashtable und scannt jede lokale Hash-Tabelle, die in der ersten Phase hergestellt wurde. Auf diese Weise, schreibt jeder Thread ohne Konkurrenzsituation in den beiden Phasen auf einer lokalen Datenstruktur. Daher muss eine Barriere zwischen den beiden Phasen existieren. Diese Strategie vermeidet ein Aktualisieren einer lokalen Hash-Tabelle, während die Hash-Tabelle in der Phase 2 gelesen wird. Wenn eine lokale Hash-Tabelle zu groß wird, kann ein Thread Överflow-Bucketsërstellen, welche neue Gruppen enthalten. Sobald ein solcher Överflow-Buckets"voll ist, wird es zu einer globalen Liste von Överflow-Buckets"für eine bestimmte Partition veröffentlicht. Threads können ihre lokalen Hash-Tabellen und die entsprechenden Överflow-Buckets"reorganisieren, indem weniger häufige Gruppen zu den Överflow-Buckets"verschoben werden.

2.5.3 Was können wir von dem System lernen?

- Häufigkeitenbasierte Partitionierung der Columns
- Dictionary-kodierung der Werte
- Query Processing auf kodierten Daten
- Synopsis Table
- SIMD Operationen auf Vektoren komprimierter Daten
- Multithreading beim Query Processing

Abbildung 2.55: Overview of GROUP BY processing in DB2 BLU with two phases of local and global processing. [56]

- Reduzierung Speicherbandbreite durch „cache-conscious algorithms“

2.6 SAP HANA *(Stefan Noll)*

Bei dem von der SAP AG entwickelten Datenbankmanagementsystem SAP HANA [5] handelt es sich um eines der aktuell auf dem Markt erfolgreichsten Datenbanksysteme, welches vollständig auf eine Vorhaltung der Daten im Arbeitsspeicher setzt. Das wesentliche Verkaufsargument von SAP HANA ist dabei die Möglichkeit, die wachsende Anzahl an Daten und Abfragen von sowohl transaktionale als auch analytische Workloads mithilfe moderner Hardware hoch-performant zu verarbeiten.

2.6.1 Überblick *(Stefan Noll)*

Entwickelt wurde das Datenbanksystem am Hasso-Plattner-Institut in Kooperation mit der SAP AG. Es basiert auf verschiedenen anderen Projekten von SAP, wie z.B. SAPMaxDB, SAP TREX, SAP BI Accelerator oder P*Time. Im Jahr 2010 wurde das System auf der SAPPHIRE-Konferenz das erste Mal der Öffentlichkeit vorgestellt. Damit setzte SAP als eines der ersten großen Unternehmen auf die In-Memory Datenbanktechnologie, die bis dahin eher in der Forschung als der Wirtschaft eine größere Rolle spielte. Dabei wird der Arbeitsspeicher und die Caches der Hardware als primärer Speicher und zur Vorhaltung aller Daten genutzt. Herkömmlicher Festplattenspeicher wird hingegen nur als Backup eingesetzt. Heute ist das SAP HANA Datenbankmanagementsystem Teil der SAP HANA Appliance.
Ursprünglich wurde SAP HANA für die Analyse von Unternehmensdaten und das Erstellen von Berichten konzipiert und entwickelt. Außerdem können auch eigene Anwendungen für das System mittels einem Modellierungswerkzeug entwickelt werden, sodass SAP HANA weiterhin auch als Applikationsplattform einsetzbar ist (vgl. [24]).
Die ersten Systeme wurden zudem ausschließlich als Appliance, eine von SAP vorkonfigurierte Kombination aus Soft- und Hardware, verkauft. Mittlerweile kann SAP HANA auf unterschiedliche Art und Weise bezogen werden – beispielsweise auch in Kombination mit Hardware von IBM, Dell oder HP, oder aber auch als Cloudappliance von Amazon [8]. Die von Amazon vorgeschlagenen

Instanzen haben dabei alle die Größe *8xlarge* und verfügen somit z.B. bei der Konfiguration *c3.8xlarge* über mindestens 32 CPU-Kerne (Intel Xeon E5-2680 v2), 60 GB Arbeitsspeicher, sowie 320 GB SSD Speicher.

Architektur und Besonderheiten *(Stefan Noll)*

Im Unterschied zu herkömmlichen zeilenbasierten Datenbanksystemen, die häufig auf SQL als Benutzerschnittstelle beschränkt sind, ist SAP HANA eine Lösung für viele erdenklichen Anwendungen. So unterstützt SAP HANA verschieden Schnittstellen für Anwendungen und auch verschiedene physische Repräsentationen, um für verschiedene Anwendungen dessen Daten möglichst optimal zu speichern. Die unterschiedlichen Benutzerschnittstellen sowie die Architektur des Systems insgesamt sind in Abbildung 2.56 dargestellt.

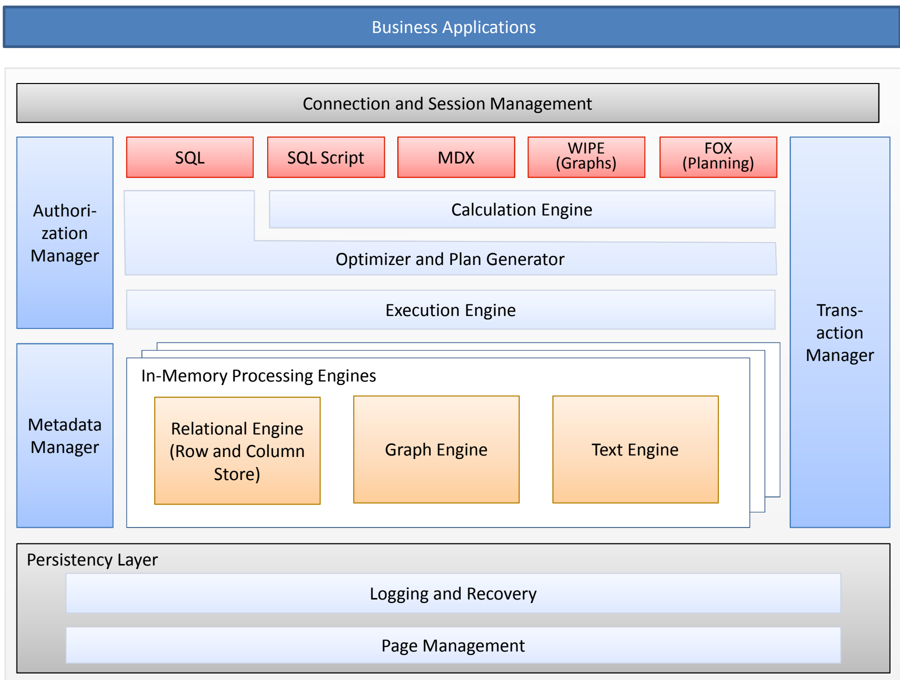


Abbildung 2.56: Architektur von SAP HANA [24]

So verfügt SAP HANA, neben einem SQL-Interface für relationale Datenbankabfragen, auch über eine Suchmaschine für Texte, um Anfragen auf halb oder nicht strukturierten Daten beantworten zu können. Darüber hinaus wird in Form einer speziellen Engine ebenfalls Graphalgorithmen unterstützt. Durch diese starke Verknüpfung mit der Anwendungssoftware, können innerhalb des Datenbankmanagementsystem spezifische semantische Informationen genutzt werden und so die Daten noch effizienter gespeichert und verarbeitet werden. Des Weiteren bietet SAP HANA die Möglichkeit anwendungsspezifische Operationen zu definieren, die dann für die Datenverarbeitung direkt in der Datenbank ausgeführt werden können (vgl. [24]).
Als Format für die Daten wird sowohl eine zeilenorientierte Darstellung als auch eine spaltenorientierte Anordnung der Daten unterstützt. Dabei werden bei

allen Anfragen jederzeit volle ACID Garantien gewährt. Um Nebenläufigkeit zu handhaben wird das Multiversion Concurrency Control (MVCC) Prinzip eingesetzt. Dadurch werden langlaufende Lese-Transaktionen nicht durch Update-Transaktionen blockiert. Im spaltenorientierten Format wird außerdem verstärkt auf Kompression gesetzt, um höhere Datentransferraten zu erreichen (vgl. [24]). Bei der Entwicklung von SAP HANA wurde außerdem großen Wert auf Parallelisierung und Skalierbarkeit gelegt. Dies reicht von Parallelisierung auf Thread oder CPU-Kern Ebene bis hin zu einen Cluster-betrieb, wo in einer verteilten Umgebung mehrere vernetzte Maschinen zusammen eingesetzt werden (vgl. [25]). Um Persistenz zu gewährleisten werden außerdem ein *Write-Ahead-Log*, *Shadows Pages* und *Savepoints* implementiert. Im Falle eines Neustart des Systems können so alle Daten vom Festplattenspeicher geladen werden und ein konsistenter Zustand gewährleistet werden (vgl. [24]).

2.6.2 Eingesetzte Techniken und Konzepte *(Stefan Noll)*

Im Datenbankmanagementsystem SAP HANA werden einige Techniken und Konzepte ein- bzw. umgesetzt, die für die Performanz des Systems eine große Rolle spielen. Einige dieser Besonderheiten sollen im Folgenden kurz vorgestellt und für die Nutzung in SliceDB evaluiert werden.

Speicherformate *(Jan Stallmann)*

Speicherlayout. Das zentrale Speicherformat des Datenbankmanagementsystems SAP HANA ist die spaltenorientierte Speicherung. Dieses Format wird typischerweise für analytische Workloads, wie z.B. in Data-Warehouse Anwendungen, eingesetzt, da sich eine große Menge von Daten in diesem Format sehr effizient verarbeiten lässt. In SAP HANA wird dieses Format sowohl für analytische als auch für transaktionale Workloads verwendet, obwohl sich für transaktionale Workloads eine zeilenorientierte Speicherung besser eignen würde. Trotzdem wird auch für diesen Anwendungsfall die spaltenorientierte Speicherung verwendet, da einige Annahmen getroffen werden können, sodass die Nachteile der spaltenorientierten Speicherung keine so starken Auswirkungen haben. So wird davon ausgegangen, dass in der Praxis die überwiegende Anzahl an Zugriffe lesende Zugriffe sind und die überwiegende Anzahl an Änderungen nur neue Daten Anfügen werden. Durch die effiziente spaltenorientierte Speicherung ist der Vorteil von Indices auf *Primary-Key* gering, sodass der Aufwand, diese zu führen, eingespart werden kann. Dazu kommen die Eigenschaften der SAP Anwendungen, für die das Datenbankmanagementsystem SAP HANA ausgelegt ist, dass es in deren Tabellen viele Spalten gibt, wovon in einer Anfrage nur wenige benötigt werden bzw. viele nur Default-Werte enthalten. Das bedeutet, dass mit der spaltenorientierten Speicherung der Aufwand für eine unkomprimierte Speicherung und vollständige Verarbeitung aller Spalten eingespart werden kann. Trotz dieser Argumente für die spaltenorientierten Speicherung, lässt sich in SAP HANA auch eine zeilenorientierte Speicherung nutzen und beide Format in einer Datenbank miteinander kombinieren (vgl. [25] und [24]).

Komprimierung. Durch den Einsatz von spaltenorientierter Speicherung lassen sich die Daten effizienter komprimieren. Ermöglicht wird dies durch die aufeinanderfolgende Speicherung von gleichartigen Daten bzw. Daten vom

gleichen Datentyp. Das Datenbankmanagementsystem SAP HANA setzt dafür auf eine sortierte *Dictionary*-Komprimierung. Dabei wird jedem eindeutigen zu speichernden Wert eine `valueID` zugeordnet, wobei die `valueIDs` nach der Sortierung der Werte vergeben werden. So muss neben dem *Dictionary*, das die Abbildung auf die eigentlichen Werte enthält, in der Spalte nur die `valueIDs` der entsprechenden Werte gespeichert werden. Da die Eigenschaften der `valueID` bekannt sind und meistens auch weniger Bits zur Repräsentation benötigen, als die original Werte, lassen sich diese effizient mittels *Bit-Packing* kodieren. Durch die Sortierung ist der überwiegende Teil der gespeicherten `valueIDs` aufeinanderfolgend, sodass sich der Speicherplatzbedarf durch Verfahren, wie z.B. *Run-Length Encoding*, weiter reduzieren lässt. Einen weiteren Einfluss auf den Speicherplatzbedarf hat aber auch das *Dictionary*, das jetzt die eigentlichen Daten enthält. Dieses wird zur Reduzierung des Speicherplatzbedarfs noch mal mittels verschiedener *Prefix-Encoding* Verfahren komprimiert (vgl. [24]).

Unified Table Structure. Um an den spaltenorientierten Daten effizient Änderungen vornehmen zu können, nutzt das Datenbankmanagementsystem SAP HANA die *Unified Table Structure*, veranschaulicht in Abbildung 2.57. Dabei werden zwei *Delta*-Strukturen verwendet, bevor die Daten in der eigentlichen Struktur gespeichert werden. Die erste *Delta*-Struktur ist `L1-delta`, die zeilenorientiert aufgebaut und so für schnelle schreibende Zugriffe optimiert ist. In dieser Struktur werden $\approx 10,000 - 100,000$ Zeilen gespeichert, bis die Daten in die nächste *Delta*-Struktur verschoben werden. Die nächste *Delta*-Struktur ist `L2-delta`, welche bereits spaltenorientiert aufgebaut ist und *Dictionary*-Komprimierung verwendet. Aus Performancegründen ist dieses *Dictionary* jedoch nicht sortiert. Das `L2-delta` kann bis zu 10 Millionen Zeilen speichern, bevor die Daten in den eigentlichen Speicher verschoben werden. Der eigentliche Speicher ist der `Main Store`, dieser nutzt das in den vorigen Abschnitten beschriebene Speicherlayout und Komprimierungsverfahren. Damit wird die Komprimierungsrate von *Delta*-Struktur zu *Delta*-Struktur immer größer. Durch die eingesetzten Verfahren ist diese Struktur besonders auf lesende Zugriffe optimiert und besitzt den geringsten Speicherplatzbedarf, um möglichst viele Daten im Hauptspeicher halten zu können (vgl. [59]).

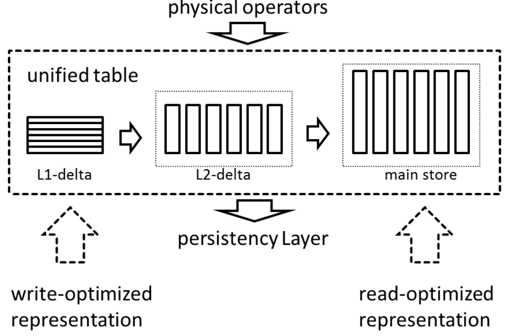


Abbildung 2.57: *Unifed Table Structure* [59]

Verwaltung der *Unified Table Structure.* Damit die Größenbeschränkungen der einzelnen Strukturen eingehalten werden und die Vorteile des `Main Store` zum Tragen kommen, werden die Daten während der Laufzeit auf die nächste Stufe verschoben. Dafür werden zwei verschiedene *Merge*-Algorithmen eingesetzt. Für das Verschieben der Daten von `L1-delta` in `L2-delta` wird `L1-to-L2-delta Merge` eingesetzt. Dabei werden die Daten einfach in die spaltenweise Darstellung überführt, fehlende Werte in das *Dictionary* übernommen, die entsprechenden `valueIDs` ermittelt und gespeichert. Das Verschieben der Daten von `L2-delta` in den `Main Store` mittels `L2-delta-to-main Merge` ist hingegen aufwändiger. Für dieses Verschieben gibt es mehrere Algorithmen, die aus einem `L2-delta` und einem `Main Store` atomar einen neuen `Main Store` berechnen. Für den `L2-delta-to-main Merge` werden hier zwei Algorithmen vorgestellt (vgl. [59]).

Classic Merge. In einem ersten Schritt des *Classic Merge* werden die beiden *Dictionarys* des `L2-delta` und des `Main Store` vereinigt, siehe Abbildung 2.58. Zu beachten ist dabei, dass zu den alten `valueIDs` die Einträge beider *Dictionarys* jeweils auch die neu berechneten `valueIDs` gespeichert werden. Mit diesen Informationen, lassen sich die `valueIDs` der Daten für das neue *Dictionary* umrechnen und in den neuen `Main Store` einfügen.
Dieser Algorithmus lässt sich optimieren, wenn bestimmte Annahmen über die beide *Dictionarys* getroffen werden können. Wenn z.B. das *Dictionary* des `L2-delta` eine Teilmenge des *Dictionary* des `Main Store` ist und die `valueIDs` übereinstimmen, ist das Vereinigen der *Dictionarys* nicht mehr notwendig und der aufwendigste Teil des Algorithmus ist nicht mehr nötig. Eine weitere hilfreiche Annahme ist, wenn alle `valueIDs` des `L2-delta` größer sind als die des `Main Store`, dann kann das *Dictionary* des `L2-delta` einfach angehängt werden und die Daten müssten nicht umgerechnet werden (vgl. [59]).

Partial Merge. Um die Optimierungen des *Classic Merge* auch ohne die Annahmen durchführen zu können, wird bei *Partial Merge* der `Main Store` noch einmal in zwei Teile aufgeteilt. Ein Teil ist das `Passiv Main`, dieses enthält die Daten und ein sortiertes *Dictionary*, die bei einem *Merge* mit dem `L2-delta` nicht verändert werden. Der andere Teil, der `Active Main`, enthält die restlichen Daten mit einem sortierten *Dictionary*, jedoch werden in diesem die Einträge aus dem *Dictionary* des `Passiv Main` nicht gespeichert und alle *valueIDs* der Einträge sind größer als alle des *Dictionarys* des `Passiv Main`. Dies ist veranschaulicht in Abbildung 2.59. Die Aufteilung hat zur Folge, dass bei dem *Merge* mit dem `L2-delta` nur ein Teil der Daten verändert werden muss und ein wesentlich kleineres *Dictionary* vereinigt werden muss. Damit das `Active Main` nicht zu groß wird, werden das `Active Main` und `Passiv Main` periodisch aufwendig zusammengeführt, jedoch bleiben die Daten des `Passiv Main` unverändert. Dieses Zusammenführen kann ausgeführt werden, wenn das System nicht ausgelastet ist und Rechenleistung bzw. Speicherbandbreite zur Verfügung steht (vgl. [59]).

Evaluation der Speicherformate *(Jan Stallmann)*

Das Datenbankmanagementsystem SAP HANA nutzt eine spaltenorientierte Speicherung mit zwei *Delta*-Strukturen und *Dictionary*-Komprimierung, um

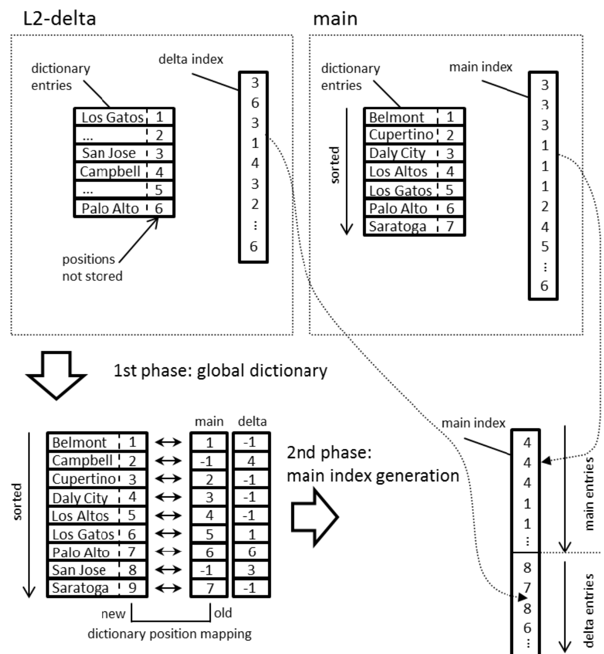


Abbildung 2.58: Details des L2-delta-to-main Merge [59]

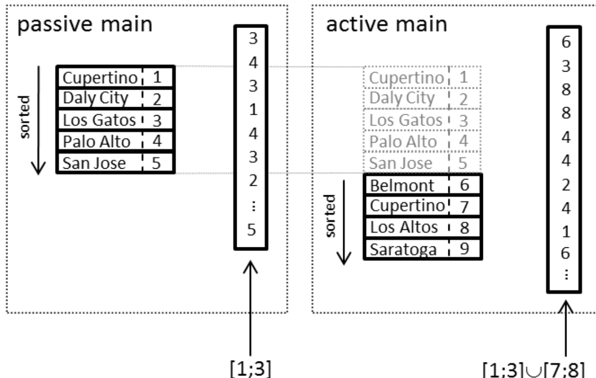


Abbildung 2.59: Bereichsabfrage im `Active Main` and `Passiv Main` [59]

schnelle lesenden Zugriffe, aber auch schreibende Zugriffe, zu ermöglichen, möglichst viele Daten im Hauptspeicher halten und dessen Bandbreite bestmöglich ausnutzen zu können.
Die mehrstufigen *Delta*-Strukturen bieten Potenzial für schnellere schreibende Zugriffe bzw. transaktionale Workloads, sind jedoch aufwendig zu implementieren. Nichtsdestotrotz lässt sich die Idee der *Delta*-Struktur vereinfachen, sodass man auch eine einstufige *Delta*-Struktur, in Form von *Ghostbits*, verwenden kann, um in dem hier betrachteten Workload selten auftretende schreibende Zugriffe noch einigermaßen effizient ausführen zu können.
Die *Dictionary*-Komprimierung bietet sich bei einer spaltenorientierte Speicherung an, um den Speicherplatz effizient auszunutzen. Dabei lassen sich auch verschieden Abstufungen implementieren, wie z.B. die Sortierung oder die komprimierte Abspeicherung der `valueIDs`. Somit wäre dies eine Technik, die sich eignen würde, um diese schrittweise in einer späteren Phase des Projektes umzusetzen.

SIMD *(Stefan Noll)*

Eine weitere Technik zur schnelleren Datenverarbeitung mit Prozessoren bietet SIMD (engl. Single Instruction Multiple Data). Dabei werden, wie in Abbildung 2.60 dargestellt, mehrere, gleiche Rechenoperationen parallel auf gleichartigen Daten ausgeführt. Dies führt je nach Anwendungsfall zu einer höheren Performanz.

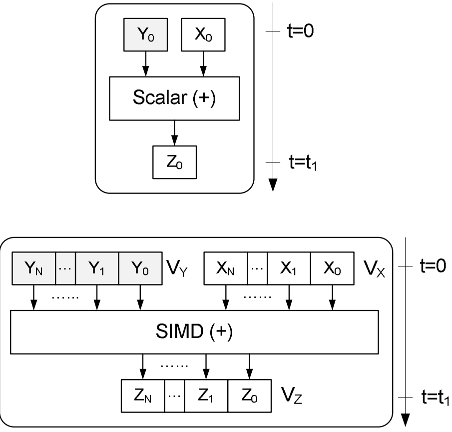


Abbildung 2.60: SIMD [68]

Ursprünglich eher bei der Multimedia-, Signal- oder Bildverarbeitung eingesetzt, lässt sich diese Technik auch bei der Datenverarbeitung in Datenbankmanagementsysteme anwenden. Praktisch unterstützt wird das Konzept durch verschiedene Architekturen und Befehlssätze wie beispielsweise *3DNow!* von AMD oder *MMX*, *SSE* oder *AVX* von Intel. Die im Folgende vorgestellten Algorithmen nutzen den *AVX2* Befehlssatz von Intel. Dabei handelt es sich um eine Erweiterung des x86-Befehlssatzes, der mit der Haswell Architektur bei den Prozessoren Core i3/i5/i7 4XXX eingeführt wurde.
Der AVX2 Befehlssatz arbeitet, wie in Tabelle 2.2 aufgelistet, auf 16 Prozessorregistern mit den Bezeichnern YMM{0-15}, die jeweils eine Größe von 256 Bit

haben. Alternativ können aus Gründen der Abwärtskompatibilität zu SSE auch 16 Register mit den Bezeichnern XMM{0-15} adressiert werden. Im letzteren Fall wird nur auf die unteren 128 Bit der Prozessorregister zugegriffen.

255 128	127 0
YMM0	XMM0
YMM1	XMM1
YMM2	XMM2
YMM3	XMM3
YMM4	XMM4
YMM5	XMM5
YMM6	XMM6
YMM7	XMM7
YMM8	XMM8
YMM9	XMM9
YMM10	XMM10
YMM11	XMM11
YMM12	XMM12
YMM13	XMM13
YMM14	XMM14
YMM15	XMM15

Tabelle 2.2: Register des AVX Befehlssatzes

Als Programmierer hat man mehrere Möglichkeiten den Befehlssatz einzusetzen. So bieten die meisten Compiler Unterstützung für `inline` Assemblercode in C oder C++. Alternativ können intrinsische Funktionen genutzt werden. Dies sind (praktisch) C Funktionen, die eins zu eins bestimmten Assemblerinstruktionen entsprechen. Als Beispiel sei hier ein Auszug aus dem *Intel Intrinsics Guide* (vgl. Abbildung 2.61) genannt.

__m256i _mm256_add_epi32 (__m256i a, __m256i b) vpaddd

Synopsis

__m256i _mm256_add_epi32 (__m256i a, __m256i b)
#include "immintrin.h"
Instruction: vpaddd ymm, ymm, ymm
CPUID Flags: AVX2

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 7
	i := j*32
	dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Abbildung 2.61: Auszug aus *Intel Intrinsics Guide* (319433-022)

Darüber hinaus gibt es auch angepasste Laufzeitbibliotheken, wie die standardmäßige Mathematikbibliothek von C, die die SIMD Befehlssätze direkt integriert haben.

SIMD Algorithmen *(Stefan Noll)*

Die oben genannten Konzepte werden für spezifische SIMD Algorithmen genutzt, um die Datenverarbeitung des Datenbanksystems SAP HANA zu beschleunigen. [25] Im weiteren Verlauf wird exemplarisch ein Algorithmus für einen *Full Table Scan* mit *Range Predicate* [67] vorgestellt.

Full Table Scan mit Range Predicate *(Stefan Noll)*

Bei einem Full Table Scan wird über alle Daten einer Spalte ein Prädikat ausgewertet und diejenige Zeilen zurück geliefert, die dieses Prädikat erfüllen. Im Falle eines Range Predicate geht es dabei um die Überprüfung, ob sich der Wert in der Spalte innerhalb eines vorgegeben Wertebereichs befindet. Insbesondere können dadurch natürlich auch normale Gleichheitsprädikate ausgewertet werden. Des weiteren erwartet der Algorithmus als Eingabe, dass die Daten komprimiert vorliegen. Genauer sollen alle Codewörter durch eine *Dictionnary* bzw. *Number Compression* eine feste Größe von z.B. 20 Bits haben.
Der Algorithmus lässt sich in drei verschiedene Phasen unterteilen:

1. Zunächst werden die Codewörter entpackt, d.h. aus dem Hauptspeicher in Prozessorregister geladen, dort neu angeordnet und in 32 Bit Zahlen transformiert. Die neue Ausrichtung ist notwendig, da für eine weitere Verarbeitung der Daten, die Codewörter mit der entsprechenden Maschinenwortbreite des Systems übereinstimmen müssen. Ein Beispiel für diesen Fall ist in Abbildung 2.62 dargestellt. Wie man leicht einsehen kann, müssen die Codewörter (hier $v1$ bis $v7$) nicht zwangsläufig mit der Breite eines Bytes übereinstimmen. Dies ist beispielsweise bei $v2$ der Fall. Hier liegt das 20 Bit breite Codewort nicht innerhalb eines 32 Bit Blocks.

2. Im darauf folgenden Schritt wird das Prädikat für einige Daten parallel mit Hilfe von AVX2 Instruktionen ausgewertet.

3. Als letztes werden die Ergebnisse konstruiert und zurückgegeben.

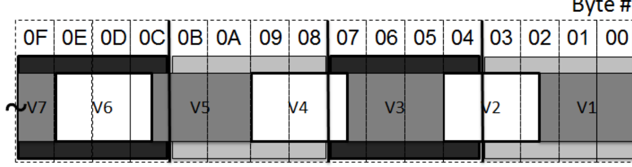


Abbildung 2.62: Codewörter vs. Maschinenwörter [67]

Der erste Schritt lässt sich als eigenständiger Algorithmus wie in Abbildung 2.65 formulieren. In Zeile 4 werden 8 Codewörter parallel in ein Register geladen. In den Zeilen 5 und 6 werden die Codewörter mit Hilfe einer Shuffle und Shift Operation an die Maschinenwortbreite ausgerichtet. Da sich die Positionen der

Codewörter alle 8 Wörter wiederholen, können die Operationen hier immer mit den gleichen Argumenten ausgeführt werden. In den Zeilen 7 und 8 werden die die ausgerichteten Daten nun noch mit einer AND-Operation mit 0 zu 32 Bit Zahlen transformiert und anschließend ausgegeben.

Algorithm 1 Unpacking algorithm

```
1: set k to 0
2: for i from 0 to max_index/128 do
3:    for j from 0 to 15 do
4:       parallel_load v from input[k * 16 + j * b]
5:       shuffle v using shuffle_mask(m_0, ..., m_31)
6:       parallel_shift v by (s_0, ..., s_7)
7:       parallel_and v by (a_0, ..., a_7)
8:       parallel_store v in output[i * 16 + j * 8]
9:    end for
10:   increase k by b
11: end for
```

Abbildung 2.63: Algorithmus zum Entpacken von Codewörtern [67]

Das gesamte Vorgehen das Algorithmus ist zudem noch einmal in Abbildung 2.64 anschaulich dargestellt.
Der eigentliche Algorithmus, der sowohl das Entpacken der Codewörter, als auch das Auswerten der Prädikate durchführt, ist hingegen in Abbildung 2.65 dargestellt. Das Ergebnis wird hier in Form eines Bitvektors zurückgegeben.
Zeilen 4 bis 7 des Algorithmus 2.65 entsprechen dabei dem zuvor genannten Algorithmus 2.63 zum Entpacken der Codewörter. In Zeile 8 wird das Range Predicate ausgewertet. Dafür wurden zuvor in Zeile 1 und 2 zwei Register mit den jeweiligen Intervallgrenzen des Prädikats initialisiert. Diese beiden Register werden nun für den parallelen Vergleich der Codewörter genutzt. Anschließend wird in Zeile 9 das Ergebnis des Vergleichs in eine 8 Bit Zahl überführt. Bei dieser Zahl handelt es sich um einen Bitvektor der mit einer 1 an der i-ten Stelle angibt, ob das i-te Codewort das Prädikat erfüllt, sonst 0.
Der Ablauf des Algorithmus ist noch einmal in Abbildung 2.66 graphisch veranschaulicht.

SIMD Algorithmen - mögliche Optimierungen *(Stefan Noll)*

Die allgemein gültigen Algorithmen 2.63 und 2.65 bieten darüber hinaus noch weiteren Spielraum für verschiedene Optimierungen. Einige dieser Optimierungen werden im Folgenden kurz angesprochen.

- Die Shift-Operation ist nicht notwendig bei 8, 16, 24, und 32 Bit Codewörtern, da diese schon einem Maschinenwort entsprechen.
- Es können 32 Codewörter innerhalb einer Iteration mit einem einzigen LOAD bei 1,2,4 und 8 Bit Codewörten dekomprimiert werden (siehe [67]).
- 16 Bit lange Codewörter können direkt in 32 Bit mit einem einzigen Befehl (*vpmovzxwd*) konvertiert werden.

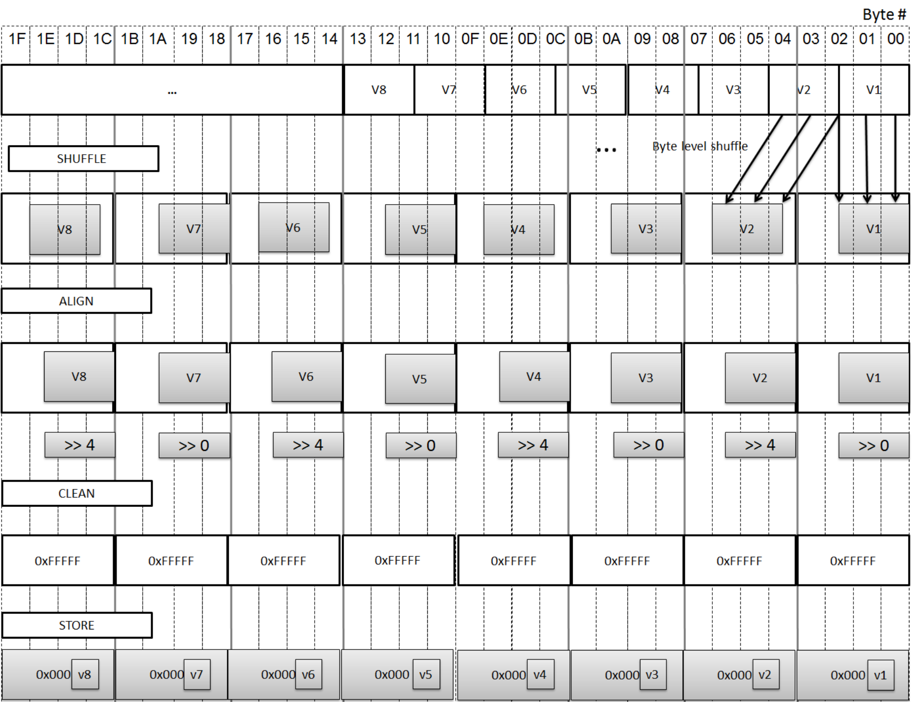


Abbildung 2.64: Beispiel für Algorithmus2.63 für 20 Bit Codewörter [67]

Algorithm 3 Range scan with bit vector result

1: parallel_shift $(min, min, min, min, min, min, min, min)$ by $(s_0, ..., s_7)$, store in min
2: parallel_shift $(max, max, max, max, max, max, max, max)$ by $(s_0, ..., s_7)$, store in max
3: set k to 0
4: **for** i from 0 to $max_index/128$ **do**
5: **for** j from 0 to 15 **do**
6: parallel_load v from input$[k * 16 + j * b]$
7: shuffle v using shuffle_mask$(m_0, ..., m_{31})$
8: parallel_compare v with $(min,\ max)$, store in t
9: convert t to 8-bit integer r
10: store r in output$[i * 16 + j]$
11: **end for**
12: increase k by b
13: **end for**

Abbildung 2.65: Algorithmus zum Auswerten eines Range Predicates mit Bitvektor als Rückgabe [67]

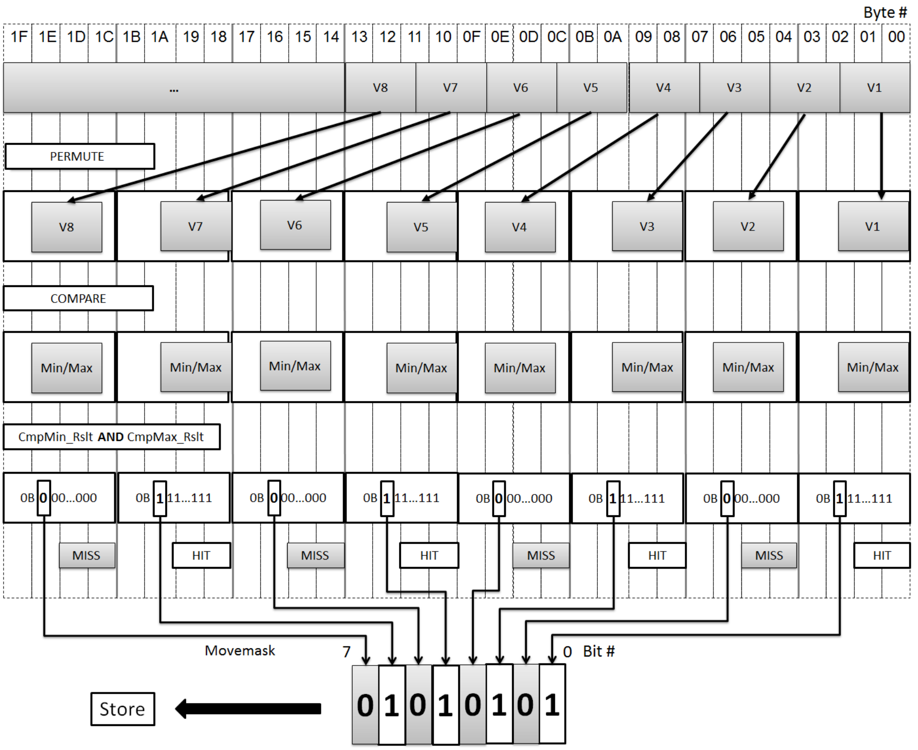


Abbildung 2.66: Beispiel für Algorithmus 2.66 für 20 Bit Codewörter [67]

- Die Bitvektorkonstruktion kann übersprungen werden, wenn alle Vergleiche Null ergeben. Dies kann mit dem Befehl *vptest* durchgeführt werden.

Außerdem sind noch viele weitere Optimierungen möglich, wie man [67] entnehmen kann. Insgesamt wichtig festzuhalten ist jedoch die Tatsache, dass dadurch für jede mögliche Codewortgröße ein speziell angepasster Algorithmus möglich ist.

Evaluation der SIMD Algorithmen *(Stefan Noll)*

Die im vorherigen Abschnitt vorgestellten SIMD Algorithmen werden in SAP HANA eingesetzt um möglichst effizient komprimierte Daten zu verarbeiten. Kompression spielt in dem Datenbankmanagementsystem von SAP eine große Rolle, da Anwender viel mit Zeichenketten arbeiteten und sich lange Zeichenketten teils stark komprimieren lassen.
Wir verwenden als erstes Anwendungsbeispiel für unsere entwickelte Datenbank den Star Schema Benchmark [51]. Dessen Anfragen und Daten konzentrieren sich allerdings nicht auf die schnelle Verarbeitung von langen Zeichenketten. Daher gibt es schon alleine aus der Anwendungsperspektive entscheide Unterschiede was die Anforderungen an das Datenbanksystem betrifft.
Nichtsdestotrotz bieten SIMD Techniken viel Potenzial für Optimierungen in einer späteren Phase des Projektes. Denn jeder halbwegs aktuelle Intel Prozessor bietet in Form von SSE Unterstützung für SIMD Befehle. Wenn diese vollständig ungenutzt bleiben würden, hätte man Ressourcen, die ohne zusätzliche Hardware praktisch umsonst verfügbar sind, nicht ausreichend ausgeschöpft. Allein schon für eine bessere Ressourcenausnutzung sollte man die Nutzung von SIMD Befehlen in Betracht ziehen.
Doch wie auch die zuletzt angesprochen Optimierungsmöglichkeiten der beschriebenen Algorithmen zeigen, kann eine effiziente Nutzung von SIMD Befehlen im Detail sehr aufwendig sein, da viele unterschiedliche Fälle, wie die verschiedenen Längen von Codewörtern, betrachtet werden müssen. Hier sollte man sich in jedem Fall die Frage stellen, ob der erhöhte Mehraufwand für spezialisierte Algorithmen am Ende einen möglichen Performanzgewinn rechtfertigt.

Kapitel 3

SliceDB

3.1 Interface

Nachfolgend wird der Aufbau von SliceDB im Bezug auf die Eingabe, die Darstellung und die Abarbeitung einer Anfrage beschrieben. Dazu wird hier auf die Konfiguration, das Logging und die speziell für SliceDB entworfene Anfragesprache eingegangen. Außerdem werden die grobe Struktur und die Abarbeitung von Anfrageplänen – welche aus der einer Anfrage in der zuvor genannten Sprache geparst werden – erläutert.

3.1.1 Bedienung *(Jan Stallmann)*

SliceDB stellt für die Bedienung des Datenbankmanagementsystems ein einfaches textbasiertes Interface zur Verfügung. Das Interface arbeitet immer maximal mit einer Datenbank. Nach dem Start ist keine Datenbank geladen und es wird die folgende Eingabeaufforderung angezeigt.

```
Welcome to SliceDB!
SliceDB>
```

Laden einer Datenbank. Eine Datenbank lässt sich laden, indem eine neue Datenbank erstellt oder eine bestehende Datenbank geladen wird.

`new_database [name]`
: Mit diesem Befehl lässt sich eine Datenbank erstellen. Wobei `[name]` durch den Namen der neuen Datenbank ersetzt werden muss. Es wird ein Ordner für die Datenbank angelegt und diese anschließend geladen.

`load_database [name]`
: Mit diesem Befehl lässt sich eine bereits bestehende Datenbank laden. Wobei `[name]` den Namen der zu ladenden Datenbank angibt.

Freigeben und Löschen einer Datenbank. Wenn eine Datenbank geladen ist, kann diese gelöscht oder auch wieder freigegeben werden.

`delete_database`
Mit diesem Befehl lässt sich die aktuell geladene Datenbank von der Festplatte löschen. Nach der Ausführung dieses Befehls ist keine Datenbank geladen.

`unload_database`
Mit diesem Befehl lässt sich die aktuell geladene Datenbank freigeben. Nach der Ausführung dieses Befehls ist keine Datenbank geladen.

Arbeiten mit einer Datenbank. Nachdem eine Datenbank geladen wurde, lässt sich mit dieser arbeiten. Es lassen sich Abfragen auf der Datenbank ausführen und Daten in die Datenbank einfügen.

`algebra [input]`
Mit diesem Befehl lässt sich eine Abfrage ausführen. Wobei die Abfrage durch `[input]` in unserer Anfragesprache (siehe Kapitel 3.1.4) angegeben und mit einem Semikolon bestätigt werden muss.

`algebrafile [path]`
Mit diesem Befehl lässt sich eine Abfrage ausführen. Wobei die Abfrage durch die Datei mit dem Pfad `[path]` in unserer Anfragesprache (siehe Kapitel 3.1.4) gegeben werden muss.

Ausführen des Star Schema Benchmarks. Um den Star Schema Benchmark [53] in SliceDB auszuführen, werden spezielle Befehle bereitgestellt. Diese erstellen die benötigten Tabellen und Spalten bzw. führen die vorgegebenen Abfragen aus.

`import_ssb [path]`
Mit diesem Befehl lassen sich die für den Star Schema Benchmark benötigten Tabellen und Spalten in der aktuell geladenen Datenbank erstellt und die Daten aus dem Ordner `[path]` importieren.

`ssbqX_Y`
Mit diesem Befehl lässt sich die Abfrage `X.Y` des Star Schema Benchmarks ausführen.

`ssbqX_Y_ij`
Mit diesem Befehl lässt sich die Abfrage `X.Y` des Star Schema Benchmarks ausführen, wobei der Invisible-Join verwendet wird (ausgenommen Abfrage 1.1, 1.2 und 1.3).

`ssbqX_Y_bf`
Mit diesem Befehl lässt sich die Abfrage `X.Y` des Star Schema Benchmarks ausführen, wobei die Anfragepläne Bloom-Filter einsetzen (siehe Kapitel 3.3.9).

Weitere Befehle. Das Interface von SliceDB unterstützt noch weitere Befehle. Ein Ausschnitt der wichtigsten Befehle bilden die folgenden.

`help`
Mit diesem Befehl lassen sich alle unterstützten Befehle ausgeben.

`exit`
Mit diesem Befehl lässt sich SliceDB beenden.

`about`
Mit diesem Befehl lassen sich allgemeine Informationen über SliceDB ausgeben.

3.1.2 Konfiguration & Parameter *(Maximilian Berens)*

Um die Konfiguration von SliceDB abseits der Kommandozeile zu ermöglichen, wurde ein Interface implementiert, welches die Definition und den Zugriff von Optionen ermöglicht. Das Interface wurde mit dem `Boost::Options` umgesetzt und erlaubt die Angabe der Optionen über eine Konfigurationsdatei und Kommandozeilen-Parameter.

Definition der Optionen

Um eine einheitliche und übersichtliche Definition zu ermöglichen, sind die Projektmitglieder angehalten, dass in `interface/configuration.h` per `Define`-Makro einheitlich der Name, Datentyp und die Kurzbeschreibung definiert wird. Dies soll eine zentrale Verwaltung der Optionen ermöglichen. Anschließend muss im Konstruktor der `Configuration`-Klasse die entsprechende Option registiert werden. An dieser Stelle werden die zuvor definierten Makros verwendet. Weiter ist es auch möglich, hier einen Default-Wert an zu geben, auf den zurückgegriffen wird, sollte die Option während des Startes von SliceDB nicht spezifiziert werden.

Zugriff von Optionen

Auf die per Kommandozeile oder Konfigurationsdatei spezifizierten Werte oder ihre Defaults wird mittels `OPTION`-Makro zugegriffen. Da dieses Makro auf ein `public static`-Member von `Configuration` zugreift, muss die Instanz dieser Klasse am Verwendungsort nicht bekannt sein. Das Makro verlangt als Parameter den Namen und den Typ der Option.
Zu beachten ist, dass für den Zugriff auf die Optionen die Klasse `Configuration` bereits instanziiert worden sein muss, dies geschieht aber in der `CMDHandler::init()` Methode und ist damit im Regelfall gegeben.

Übersicht der Optionen

Die SliceDB-Executable kann mittels `--help` gestartet werden, um alle verfügbaren Argumente und ihre Beschreibung an zu zeigen. Der Pfad zu einer Konfigurationsdatei kann mit dem Parameter `--config_path /path/to/file.cfg` (Default Argument: `./slicedb.cfg`) angegeben werden. Eine vollständige Übersicht ist nachfolgend zu finden.

Option	Beschreibung
`--help`	Zeigt alle Optionen an
`--version`	Zeigt die Version von SliceDB an
`--config_path`	Pfad zur Konfigurationsdatei
`--database_path`	Pfad zum Datenbankverzeichnis
`--benchmark`	Starte SliceDB als Benchmark
`--general.debug.console_sev_level`	Regelt das Print-Level in die Konsole
`--general.debug.all_logs_to_console`	Alle Channel in die Konsole printen
`--general.debug.file_sev_level`	Regelt das Print-Level in Logfiles

3.1.3 Ausgabe von Debugnachrichten *(Maximilian Berens)*

Um die geordnete Ausgabe von Debug- und Benutzerinformationen zu ermöglichen, wurde ein Loggingframework mittels `boost::log` implementiert. Die orthogonale Aufteilung von Nachrichten auf *Channels* und *Level* schafft die Rahmenbedingung für eine präzise Selektion der Informationen. Jeder Channel verfügt über eine eigene Logdatei, so können beispielsweise Debugginginformationen des SliceDB Schemaimporters getrennt von der Ausgabe einzelner Operatoren einer Query betrachtet werden. Weiter können die Lognachrichten über ihr Log-Level gefiltert werden. Dazu wird grob nach der Wichtigkeit unterschieden. Eine Liste der Log-Level ist nachstehend zu finden.

LogLevel	Beschreibung
FATALERROR	Fataler Fehler, führt zur Beendigung von SliceDB
INFO	Information für den Nutzer
ERROR	Fehler, der SliceDB nicht beendet
WARNING	Warnung
DEBUG	Debuglevel 0
DEBUG1	Debuglevel 1
DEBUG2	Debuglevel 2

SliceDB erlaubt die Definition der Loglevel (getrennt für jeweils die Ausgabe in Dateien und in die Konsole), um nur Nachrichten oberhalb des definierten Levels an zu zeigen. Ein Loglevel von 2 würde demnach `FATALERROR`s, `INFO`s und `ERROR`s anzeigen.
Über die entsprechende Option kann SliceDB auch alle Logging-Channel in der Konsole ausgeben, Standard ist dies allerdings nur für den *User-Channel*, welcher benutzt werden soll um den Endnutzer über den Ablauf von SliceDB zu informieren.
Das Verzeichnis, in dem die Log-Dateien zu finden sind, ist standardmäßig `./logs`. Ausgaben des Logger werden an bereits bestehende Dateien angehängt.

Ausgabe von Lognachrichten

Der Zugriff auf das Logginginterface wird mit Hilfe einiger Makros gewährleistet. Das wesentliche Makro dabei ist `LOG`, alle Weiteren sind als Spezialfall dieses Makros zu sehen und werden nur aus praktischen Gründen zur Verfügung gestellt. Der nachfolgende Code stellt die Verwendung exemplarisch dar. Zu beachten ist dabei die Verwendung des Standard Streaming-Operators «, welche die von `std::ostream` bekannte Funktionalität bietet:

```
LOG("irgend_ein_channel",
  ERROR,
  "Dies ist eine Logging Nachricht, die in die Datei
    \'logs/irgend_ein_channel.log\' ausgegeben wird.\n
    Eine Zahl: " << 42)
```

`ERROR` ist ein Enum, welches über `log.h` (wie auch das Makro) bekannt ist.

Zu beachten ist, dass der Programmierer im Falle eines `FATALERROR` die Beendigung von SliceDB selbst handhaben muss!

3.1.4 Die Anfragesprache *(Maximilian Berens)*

Um die Darstellung und das Parsen zu erleichtern, wurde eine eigene Anfragesprache für SliceDB entworfen. Die Syntax erlaubt eine Schachtelung von Operatoren, welche direkt die Struktur des später erstellten Plans widerspiegelt. Operatoren werden als Funktionen dargestellt, welche als Argument andere Operatoren sowie Tabellen und/oder andere Parameter erhalten können.
Die Anfragesprache ist mächtig genug, um alle Anfragen des Star Schema Benchmarks [51] darstellen zu können. Eine vollständige Übersicht wird nachfolgend gegeben.
Argumente der Operatoren:

```
T = Table
Operation = {ADD, SUB, MUL, DIV}
Constant = Zeichenketten, Ganzzahlen und Dezimalzahlen
Aggregator = {SUM, COUNT}
Comparator = {EQU, UNEQU, LESS, GREA, LEQU, GEQU}
Combinator = {AND, OR}
Order = {ASC, DESC}
```

Handelt es sich bei einer Konstante um eine Dezimalzahl, so wird ein Punkt als Dezimaltrennzeichen verwendet. Die unterstützten Operationen und möglichen Werte können bei Bedarf erweitert werden.

Operatoren (Rückgabewert: T) und Prädikate (Rückgabewert BOOL):

```
T : AGGREGATE(Aggregator, 'column', T)
T : COLUMN-OPERATION(Operation, 'column1', 'column2', T)
T : COLUMN-OPERATION(Operation, 'column1', Constant, T)
T : FILTER(BOOL, T)
T : GROUPBY(Aggregator, 'column', ['column1', ...], T)
T : JOIN(BOOL, T1, T2)
T : ORDERBY( [('column1', Order), ...], T)
T : PROJECTION( ['column1', ..., ('columnX',"NewNameFor_columnX"), ...], T)

BOOL : COMPARISON(Comparator, 'column1', 'column2')
BOOL : COMPARISON(Comparator, 'column1', Constant)
BOOL : COMBINATION(Combinator, BOOL, BOOL)
```

Weitere Komponenten der Syntax:

```
"" = Kennzeichnung von Strings
'' = Kennzeichnung von Spalten- und Tabellennamen
(,) = Tupel aus genau zwei Elementen
[,] = Liste, beliebig lang
```

Anmerkung: Der Projection-Operator dient auch als Rename-Operator, dazu wird in der Liste ein Tupel mit Spalte und neuem Namen angegeben!
Es folgt eine kurze Beispielanfrage auf den Star Schema Benchmark Daten, welche die Spalte *lo_quantity* aus der Tabelle *lineorder* auswählt und das Attribut *lo_quantity* nach Werten kleiner 25 filtert:

```
PROJECTION( ['lo_quantity'],
  FILTER( COMPARISON( LESS, 'lo_quantity', 25),
    'lineorder'
  )
)
```

Als etwas komplexeres Beispiel wird nun die Anfrage 1.1 aus dem Star Schema Benchmark dargestellt:

```
PROJECTION([('SUM(lo_price*lo_discount)',"revenue")],
    AGGREGATE( SUM,
        'lo_price*lo_discount',
        COLUMN-OPERATION( MUL,
            'lo_price',
            'lo_discount',
            JOIN( COMPARISON(EQU, 'lo_orderdate', 'd_datekey'),
                FILTER( COMPARISON(EQU, 'd_year', 1993),
                    'date'),
                FILTER( COMBINATION( AND,
                            COMBINATION( AND,
                                COMPARISON(GEQU, 'lo_discount', 1),
                                COMPARISON(LEQU, 'lo_discount', 3)),
                            COMPARISON(LESS, 'lo_quantity', 25)),
```

```
                    'lineorder')
            )
        )
    )
)
```

In SQL formuliert:

```
select sum(lo_extendedprice*lo_discount) as revenue
 from lineorder, date
 where lo_orderdate = d_datekey
  and d_year = 1993
  and lo_discount between1 and 3
  and lo_quantity < 25;
```

Table (T) Parameter werden immer als Letztes in der Argumentliste eines Operators angegeben. Auf eine Vereinfachung der Schreibweise (beispielsweise *SELECT (*)* für *select all columns*) wurde bisher verzichtet.

3.1.5 Aufbau des Anfrageplans *(Maximilian Berens)*

Der Anfrageplan wurde als Baum implementiert. Alle Knoten repräsentieren dabei Operationen (“Operator-Knoten”), mit Ausnahme der Blätter, welche eine Tabelle und damit die Verbindung zur Datenschicht darstellen (“Input-Knoten”).
Der Plan orientiert sich dabei direkt an der Syntax:
Ein Operator der als Argument für einen anderen Operator dient, ist dessen Kind. Die Wurzel stellt demnach den zuletzt aus zu führenden Operator dar (in der Regel ist dies eine Projektion).
Als Knoten-Klasse dient die Klasse *Node*, welche eine Implementation eines klassischen Baumes mit entsprechenden Eltern-Kind Relationen darstellt. Die Anzahl der Kinder ist dabei flexibel und wird durch den Typ des Operators bestimmt (zum Beispiel *Join* mit zwei Kindern, *Aggregation* mit nur einem Kind, *Input* ohne Kinder). Funktionen zum umhängen, hinzufügen und entfernen von Kindern, der eindeutigen Identifikation von Knoten sowie diverse Debug-Print Methoden ermöglichen eine Umgestaltung des Baumes für die spätere automatische Optimierung.
Neben den Komponenten, die sich mit der Baumstruktur befassen, können auch operator- und datenspezifische Zusatzinformationen an den Knoten gespeichert und ausgewertet werden.
Um eine Verbindung zu den tatsächlichen Daten der Datenbank her zu stellen, werden Pointervariablen auf diese (Columns/Tables) in den Komponenten der Input-Knoten nach der Erstellung des Planes gesetzt.
Erwähnenswert ist außerdem ein spezieller Knotentyp, der *Materialize*-Knoten. Dieser wird nicht direkt vom Benutzer mit angegeben, sondern automatisch hinzugefügt und ermöglicht eine Materialisierung der Zwischenergebnisse für Operationen, welche nicht auf *RIDS* arbeiten können. Dazu wird ein Flag an den Operator-Knoten ausgewertet und zwischen den Kindern und dem Operator, der eine Materialisierung benötigt entsprechend Materialisierungs-Knoten eingefügt.
Zusätzlich bietet die Node-Klasse auch (u. a. abstrakte) Methoden zur Implementierung des Vulcano-Interfaces.

3.1.6 Schema Import *(Maxmilian Berens)*

Alternativ zu `import_ssb` exitiert auch noch eine weitere Möglichkeit Daten in SliceDB zu importieren: `import_data`. Im Gegensatz zu der `import_ssb` Variante, wobei der Import für den Star Schema Benchmark hard gecoded ist, lässt sich hierbei auch das Schema angeben. Zu diesem Zweck wird der Pfad zu einem *.xml* Dokument angeben:

`import_data [pfad]`

Dieser Befehl importiert ein per `.xml` definiertes Schema. Die Schemadefinition enthält neben den Tabellendefinitionen auch eine Pfadangabe zu den Daten. Sollte aktuell keine Datenbank geladen sein, so wird eine mit spezifiziertem Namen erstellt.

Anhand des XMLs kann eine Datenbank komplett spezifiziert werden. Als Input wird je Tabelle ein CSV-Style Dokument geladen, wobei das Trennzeichen einstellbar ist. Die Tabellen mit ihren Spalten werden unter Angabe des Datentyps automatisch angelegt und korrespondierende Daten hineingeladen. Es folgt nun die Definition des SSB Schemas als XML.

```
1: <schema name="StarSchemaBenchmark">
2:     <table name="lineorder" path="path/to/lineorder.tbl"
3:        delimiter_char="|">
4:         <column name="lo_orderkey" type="Storage::RID"/>
5:         <column name="lo_linenumber" type="Storage::RID"/>
6:         <column name="lo_custkey" type="Storage::RID"/>
7:         <column name="lo_partkey" type="Storage::RID"/>
8:         <column name="lo_suppkey" type="Storage::RID"/>
9:         <column name="lo_orderdate" type="Storage::RID"/>
10:        <column name="lo_orderpriority" type="Storage::String"/>
11:        <column name="lo_shippriority" type="Storage::String"/>
12:        <column name="lo_quantity" type="Storage::RID"/>
13:        <column name="lo_extendedprice" type="Storage::RID"/>
14:        <column name="lo_ordtotalprice" type="Storage::RID"/>
15:        <column name="lo_discount" type="Storage::RID"/>
16:        <column name="lo_revenue" type="Storage::RID"/>
17:        <column name="lo_supplycost" type="Storage::RID"/>
18:        <column name="lo_tax" type="Storage::RID"/>
19:        <column name="lo_commitdate" type="Storage::RID"/>
20:        <column name="lo_shipmode" type="Storage::String"/>
21:    </table>
22:    <table name="part" path="path/to/part.tbl"
23:       delimiter_char="|">
24:        <column name="p_partkey" type="Storage::RID"/>
25:        <column name="p_name" type="Storage::String"/>
26:        <column name="p_mfgr" type="Storage::String"/>
27:        <column name="p_category" type="Storage::String"/>
28:        <column name="p_brand1" type="Storage::String"/>
29:        <column name="p_color" type="Storage::String"/>
30:        <column name="p_type" type="Storage::String"/>
31:        <column name="p_size" type="Storage::RID"/>
32:        <column name="p_container" type="Storage::String"/>
33:    </table>
```

```
34:      <table name="supplier" path="path/to/supplier.tbl"
35:         delimiter_char="|">
36:          <column name="s_suppkey" type="Storage::RID"/>
37:          <column name="s_name" type="Storage::String"/>
38:          <column name="s_adress" type="Storage::String"/>
39:          <column name="s_city" type="Storage::String"/>
40:          <column name="s_nation" type="Storage::String"/>
41:          <column name="s_region" type="Storage::String"/>
42:          <column name="s_phone" type="Storage::String"/>
43:      </table>
44:      <table name="customer" path="path/to/customer.tbl"
45:         delimiter_char="|">
46:          <column name="c_custkey" type="Storage::RID"/>
47:          <column name="c_name" type="Storage::String"/>
48:          <column name="c_adress" type="Storage::String"/>
49:          <column name="c_city" type="Storage::String"/>
50:          <column name="c_nation" type="Storage::String"/>
51:          <column name="c_region" type="Storage::String"/>
52:          <column name="c_phone" type="Storage::String"/>
53:          <column name="c_mktsegment" type="Storage::String"/>
54:      </table>
55:      <table name="date" path="path/to/date.tbl"
56:         delimiter_char="|">
57:          <column name="d_datekey" type="Storage::RID"/>
58:          <column name="d_date" type="Storage::String"/>
59:          <column name="d_dayofweek" type="Storage::String"/>
60:          <column name="d_month" type="Storage::String"/>
61:          <column name="d_year" type="Storage::RID"/>
62:          <column name="d_yearmonthnum" type="Storage::RID"/>
63:          <column name="d_yearmonth" type="Storage::String"/>
64:          <column name="d_daynuminweek" type="Storage::RID"/>
65:          <column name="d_daynuminmonth" type="Storage::RID"/>
66:          <column name="d_daynuminyear" type="Storage::RID"/>
67:          <column name="d_monthnuminweek" type="Storage::RID"/>
68:          <column name="d_weeknuminyear" type="Storage::RID"/>
69:          <column name="d_sellingseason" type="Storage::String"/>
70:          <column name="d_lastdayinweekfl" type="Storage::RID"/>
71:          <column name="d_lastdayinmonthfl" type="Storage::RID"/>
72:          <column name="d_holidayfl" type="Storage::RID"/>
73:          <column name="d_weekdayfl" type="Storage::RID"/>
74:      </table>
75:  </schema>
```

3.1.7 Parser *(Christian Schnieder)*

Um eine dynamische Anfrage zu verarbeiten und auszuführen, ist es notwenig, die Anfragesprache aus Abschnitt 3.1.4 zu parsen und in einen ausführbaren Anfrageplan zu überführen.

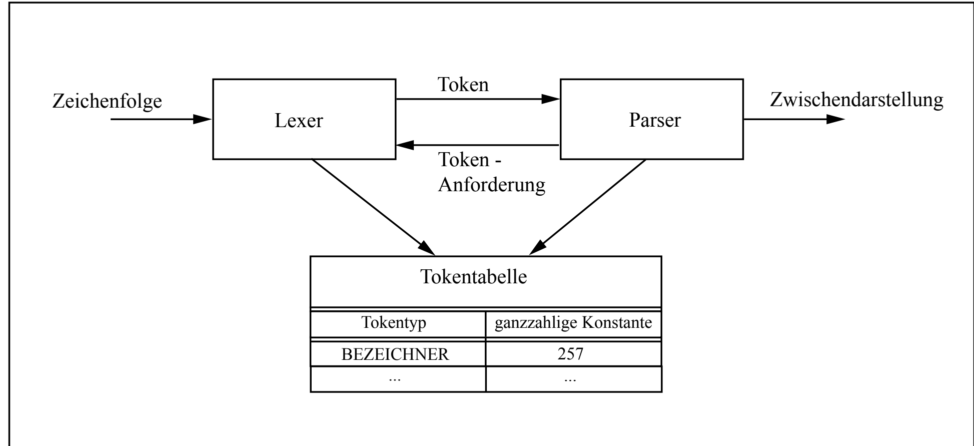


Abbildung 3.1: Prinzip der Zusammenarbeit eines Parsers und eines Lexers [32].

Für diesen Zweck nutzt SliceDB den Parsergenerator *GNU Bison*[1] in Kombination mit dem lexikalischen Scannergenerator *Flex*[2].

Funktionsweise

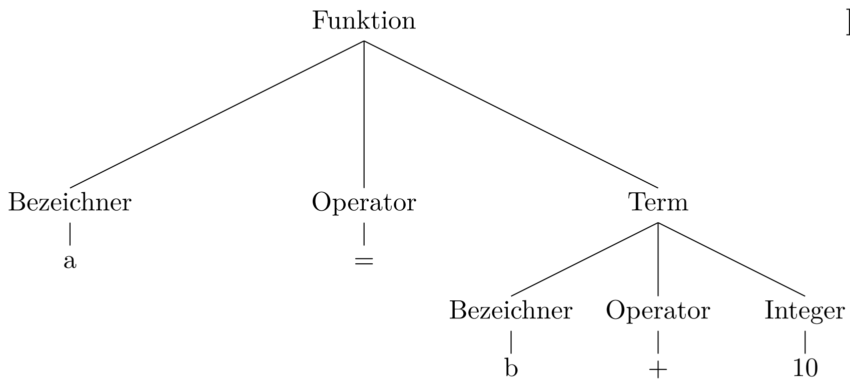
Mit Flex wird ein Scanner generiert, welcher – wie in Abbildung 3.1 dargestellt – aus einer Eingabezeichenfolge Tokens erkennt. Bison erstellt aus einer kontextfreien Grammatikbeschreibung einen Parser, der die Tokens von Flex anfordert und verarbeiten kann. Sowohl der Scanner als auch der Parser greifen dabei auf eine gemeinsame Tokentabelle zurück.

Beispielsweise könnte Flex aus der Eingabe $a = b + 10$ die folgenden Tokens erkennen:

Token	Wert
Bezeichner	a
Operator	=
Bezeichner	b
Operator	+
Integer	10

Würde Bison diese Tokens nun parsen könnte der folgende Baum entstehen:

[1] `http://www.gnu.org/software/bison/`
[2] `http://flex.sourceforge.net/`



Anwendung in SliceDB

Der Parser in SliceDB wurde so umgesetzt, dass er die dafür definierte Anfragesprache aus Abschnitt 3.1.4 versteht. Die benötigten Tokens, sowie die Grammatik wurden in den entsprechenden Dateien `parser.l` (Flex) und `parser.y` definiert. Die `parser.y` enthält dabei die Definition der Tokens, welche beim Aufruf von Bison in die `file.tab.h` übertragen werden. Diese Datei nutzt Flex anschließend beim Erzeugen des Scanners, wie es auch in Abbildung 3.2 dargestellt ist.

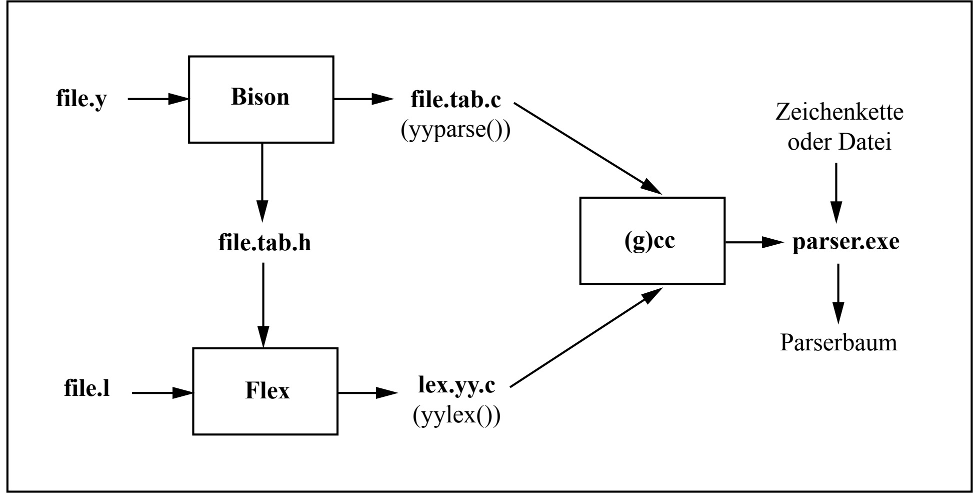


Abbildung 3.2: Erstellung eines Parsers mit Flex und Bison [32].

In Abbildung 3.3 ist dargestellt, wie die Zusammenarbeit zwischen Scanner und Parser intern funktioniert. Da die Standardimplementierungen der Funktionen `yyparse()` und `yylex()` die von SliceDB benötigte Funktionalität abdecken, werden für die die Funktionen keine eigenen Implementierungen verwendet. In der `parser.y` wird die Grammatik der Anfragesprache genau definiert und es werden an den entsprechenden Stellen die unterschiedlichen Knoten des Anfrageplanes aus den über die Anfrage angekommenen Informationen erstellt und zu einer Baumstruktur zusammengefügt.
Das Ergebnis ist ein einfacher Anfrageplan, welcher der Struktur des Anfragestrings entspricht.

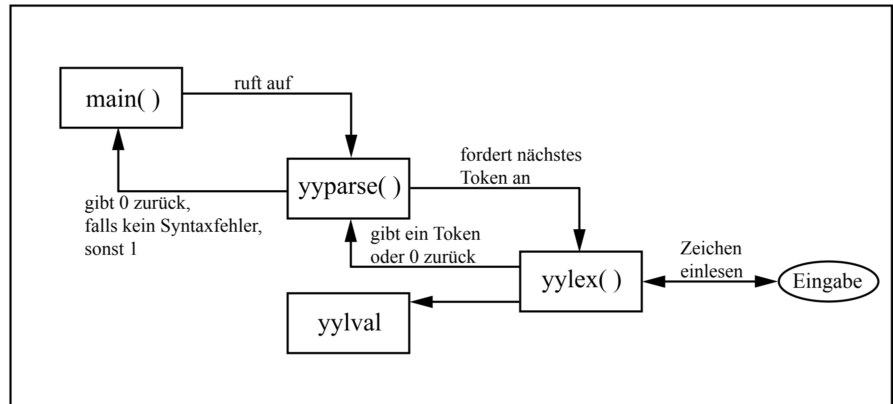


Abbildung 3.3: Interner Ablauf des Parserns einer Zeichenkette [32].

Ausführbarkeit des geparsten Anfrageplanes

Der durch den Parser erzeugten Anfrageplan ist noch nicht ausführbar, da er keine Referenzen auf die Daten enthält, auf denen eine Anfrage laufen soll.
Die Verknüpfung mit den entsprechenden Datenspalten im Storage-Layer erfolgt nach dem Parsen mittels Durchlaufen des Anfrageplanes bis zu den *Input-Knoten*, der Auswertung der Knoten und einer anschließenden Erstellung der benötigten Verknüpfung zu den existierenden Spalten im Storage-Layer. Im Anschluss daran werden die in Abschnitt 3.1.5 beschriebenen *Materialize-Knoten* über Baumoperationen hinzugefügt.
Mithilfe dieser zwei Anpassungen erhalten wir einen ausführbaren Anfrageplan.

3.1.8 Automatische Optimierung *(Christian Schnieder)*

Der in Abschnitt 3.1.7 beschriebene, vom Parser erstelle Anfrageplan enthält bisher keinerlei Optimierungen. Er stellt die Struktur des Anfragebaumes (abgesehen von den ergänzten *Materialize-Knoten*) als eine genaue Abbildung des Anfragestrings dar. So lässt sich beispielsweise durch die Anfrage festlegen, in welcher Reihenfolge die Operatoren ausgeführt werden sollen. Die Operator-Knoten werden als Standardoperatoren ohne jede Optimierung umgesetzt, so wird beispielsweise für alle *Join-Knoten* ein *Hashjoin* verwendet.
Für eine automatische Optimierung muss der Anfrageplan ausgewertet und mittels Baumoperationen (Umstrukturieren des Baumes, Austauschen/Ergänzen/Entfernen von Knoten, ...) verbessert werden.
Ein einfaches Beispiel einer Optimierung ist die Reduzierung der im Baum verwendeten Spalten auf die für die Anfrage benötigten Spalten. Hierzu müssen für jeden Knoten zunächst die benötigten und die ggf. durch den Knoten produzierten Spalten (z. B. durch eine Aggregation) ermittelt werden. Daraus lässt sich rekursiv berechnen, welche Spalten zurückgegeben werden und welche Spalten *Input-Knoten* anbieten müssen. Eine entsprechende Methode zur Berechnung der benötigten Spalten wurde auch umgesetzt, da während der Entwicklungsphase ein Join-Operator Probleme hatte, mit nicht benötigten Spalten umzugehen.
Die automatische Ermittlung von Optimierungspotential im Anfrageplan ist zum Teil sehr komplex, weshalb der Schwerpunkt bei der Entwicklung von

SliceDB bei der Implementierung von optimierten Operatoren und nicht bei der automatischen Integrierung in den Anfrageplan lag.
Beispielsweise fehlen für die automatische Verwendung des Bloom-Filters (Abschnitt 3.3.9) detaillierte Informationen über die Datenbank, wie z. B. die Größen der Tabellen oder ob es sich bei einer Tabelle um eine Fakten- oder Dimensionstabelle handelt. Ohne diese und weitere Informationen ist es auch nicht möglich, automatisiert Operationen durch andere zu ersetzen, welche unter bestimmten Bedingungen besser geeignet sind.

3.1.9 Ausführung des Anfrageplans *(Stefan Noll)*

Für die Ausführung des Anfrageplans in SliceDB wird für jeden Operator das *Open-Next-Close Interface*, auch *Volcano Iterator Model* [28] genannt, implementiert. In SliceDB werden die Daten aber nicht Tupel für Tupel, sondern vektorisiert verarbeitet. Daher wird das klassische *Volcano Iterator Model*, wie in [17] vorgeschlagen, entsprechend erweitert und angepasst. Beide eingesetzte Techniken sollen im Folgenden kurz erläutert werden. Zuerst wird das *Open-Next-Close Interface* vorgestellt und im Anschluss auf die Erweiterung zur vektorisierten Ausführung eines Anfrageplans eingegangen.

Open-Next-Close Interface *(Stefan Noll)*

Um das *Open-Next-Close Interface* im Anfrageplan zu nutzen, müssen alle Operatoren (d.h. Knoten des Anfrageplans) als sogenannte *Iteratoren* implementiert werden. Ein Iterator verfügt dabei über drei grundlegende Funktionen: `open`, `next` und `close`. Darüber hinaus speichert jede Operator seinen eigenen Zustand. Dazu zählt bei einem Hashjoin beispielsweise die Speicherung einer Hashtabelle oder bei einer Selektion die Speicherung der aktuellen Position innerhalb der Daten.
Dadurch ist es unter anderem möglich, dass ein Operator völlig unabhängig, mehrmals im Plan verwendet werden kann. Jeder Operator verfügt über gerade genug Informationen und Ressourcen, um die Daten eigenständig zu verarbeiten. Außerdem verarbeitet jeder Operator den ankommenden Datenstrom ohne dabei andere Operatoren des Anfrageplans zu kennen. Insbesondere benötigt ein Knoten im Anfragebaum also kein Wissen darüber, wo seine Eingabe genau herkommt oder wie diese konkret aussieht. Die jeweiligen Eingabedaten können so zum Beispiel das Ergebnis einer komplexen Anfrage sein oder aber auch nur das Ergebnis einer einfachen Selektion. Als Beispiel ist in Abbildung 3.4 ein möglicher Anfrageplan der Anfrage Q1.1 des Star Schema Benchmark schematisch dargestellt. Die Eingabetabellen sind dabei als Rechteckt, die Operatoren als abgerundete Rechtecke und die Ausgabe als Ellipse visualisiert.
Die Ausführung eines Anfrageplans mit Hilfe des *Open-Next-Close Interface* läuft nun wie folgt ab: Zu aller erst wird auf dem Wurzelknoten des Baums die Funktion `open` aufgerufen. Dies würde in Abbildung 3.4 beispielsweise dem Knoten „Aggregation“ entsprechen, da die Ausgabe eigentlich nicht Teil des Anfrageplans ist. Dadurch wird auch die Funktion `open` rekursiv auf allen direkten Kindknoten des Operators aufgerufen. Außerdem wird der Zustand des aufrufenden Operators anschließend selbst initialisiert. Letztendlich werden somit rekursiv alle Operatoren im Plan (von unten nach oben) initialisiert und für die eigentliche Datenverarbeitung vorbereitet.

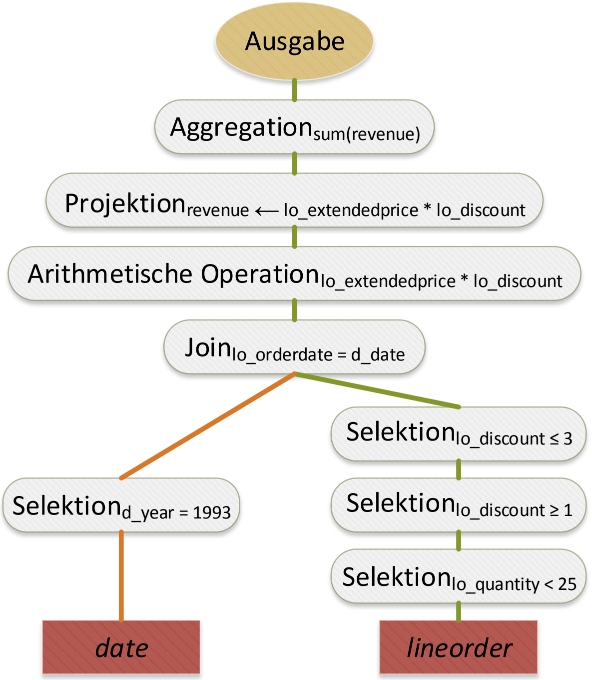


Abbildung 3.4: Schematischer Anfrageplan der Anfrage Q1.1 des SSB

Nach der Initialisierung aller Knoten wird auf dem Wurzelknoten die Funktion `next` aufgerufen. Hier geschieht die eigentlichen Verarbeitung der Daten bzw. die Ausführung des jeweiligen Operators. Zunächst wird auf allen direkten Kindknoten die Funktion `next` aufgerufen, um die verarbeiteten Daten der darunterliegenden Operationen zu erhalten. Anschließend wird auf den erhaltenen Daten die jeweilige Operation ausgeführt und das Ergebnis an den Aufrufer zurückgeliefert. Insgesamt wird die Funktion `next` solange aufgerufen bis der Datenstrom zu Ende ist. Dies ist genau dann der Fall, wenn alle Daten von den Blättern des Baums, der die Anfrage abbildet, aus verarbeitet wurden und durch alle Operatoren gewandert sind. Das Ergebnis wird schließlich oben an der Wurzel des Anfrageplan extrahiert. In Abbildung 3.4 würde dann beispielsweise beim *Volcano Iterator Model* ein Tupel in die Ausgabe geschrieben und dem Benutzer angezeigt. Außerdem erfordern einige Operatoren, sogenannten *blockierende* Operatoren, dass die Eingabedaten der Kindknoten zunächst zwischengespeichert werden müssen, bevor mit der Verarbeitung begonnen werden kann.

Sind keine Daten mehr vorhanden, die noch verarbeitet werden müssen, wird auf dem Wurzel Knoten `close` aufgerufen. Dies führt dazu, dass die jeweilige Operation ihren gespeicherten Zustand verwirft und alle gehaltene Datenstrukturen wieder freigibt. Im Anschluss wird die Funktion `close` rekursiv auf allen Kindknoten aufgerufen. Infolgedessen werden alle Operationen des Anfrageplans, die mit der Datenverarbeitung zu diesem Zeitpunkt fertig sind, beendet und alle gehaltenen Ressourcen wieder freigegeben.

Blockierende Operationen. Ein Beispiel für einen blockierenden Operator ist der Hash-Join in Abbildung 3.4. Hier wird bei dem ersten `next`-Aufruf auf dem Knoten zunächst die Eingabe des linken Kindknotens vollständig zwischengespeichert (orangefarbener Datenfluss): Es wird solange auf dem linken Kindknoten `next` aufgerufen bis alle Ergebnisse gespeichert wurden. Diese werden dann für die Konstruktion einer Hashtabelle verwendet. Anschließend, sowie in allen folgenden Aufrufen, wird nur auf dem rechten Kindknoten `next` aufgerufen, der Join mit Hilfe der Hashtabelle berechnet und das Ergebnis weiter an den Aufrufer gereicht (grüner Datenfluss).
Andere blockierende Operationen sind beispielsweise das Sortieren oder Gruppieren. Diese Operatoren besitzen im Gegensatz zum Join nur einen Kindknoten im Anfrageplan. Die Ergebnisse des Kindknotens müssen aber vollständig zwischengespeichert werden, bevor die Operation ausgeführt und das erste Ergebnis des `next`-Aufruf zurückgegeben werden kann. Eine weitere Besonderheit ist außerdem, dass das vollständige Ergebnis der Operation ebenfalls zwischengespeichert wird, um in folgenden `next`-Aufrufen die Daten Tupel für Tupel bzw. Vektor für Vektor weiter an den Aufrufer zu reichen.

Vorteile einer vektorisierte Datenverarbeitung *(Stefan Noll)*

Wie zu Beginn bereits erwähnt, werden die Datenströme im Anfrageplan bei SliceDB nicht Tupel für Tupel sondern Vektor für Vektor verarbeitet. Die Verarbeitung der Daten geschieht dabei analog zu der Anfrageverarbeitung von *Vectorwise* (siehe Abschnitt 2.1.2). Dafür wird zwischen den einzelnen Operatoren im Anfrageplan immer ein ganzer Block (hier Vektor oder auch Zwischenergebnis genannt) an Daten ausgetauscht. Solch ein Block besteht aus einer festen Anzahl von Tupeln. Eine möglich Größe wäre zum Beispiel 4000 Tupel pro Vektor. Wie ein solches Zwischenergebnis im Detail aufgebaut ist, wird in Abschnitt 3.2.8 detaillierter beschrieben.
Bei der tupelweisen Verarbeitung der Daten im klassischen *Volcano Iterator Model* gibt es einige Probleme, die sich negativ auf die Performanz des Systems auswirken können und insbesondere bei Hauptspeicherdatenbanken stärker ins Gewicht fallen. So entsteht insgesamt ein großer Overhead bei der Ausführung der Anfrage, wenn für jedes der vielen Tupel eine ganze Reihe von Funktionen bzw. Operatoren nacheinander aufgerufen wird. Es lässt sich in solch einen Fall nämlich beobachten, dass die eigentliche Ausführungzeit bei Betrachtung des Programmcodes nicht innerhalb der Funktionen für die Verarbeitung der Daten aufgewendet wird, sondern direkt durch die aufeinander folgenden Funktionsaufrufe ein Großteil der Ausführungszeit verbraucht wird ohne die eigentlichen Daten zu verarbeiten.
Des weiteren wirkt sich auch die Auflösung des Datentyps einer Spalte negativ auf die Performanz der Datenbank aus. Hier wird die üblicherweise generische Implementierung der Operatoren bzw. der Anwendung von Polymorphie zum Problem. Denn um beispielsweise eine arithmetische Operation auszuführen, wird zur Laufzeit der jeweils vorliegenden Datentyp einer Spalte bestimmt und davon abhängig im Falle einer arithmetischen Operation, wie der Division, beispielsweise eine Division von Integern oder von Gleitkommazahlen durchgeführt. Wenn der Datentyp allerdings für jede Spalte von jedem Tupel im Datenstrom immer wieder neu ermittelt werden muss, führt dies zu einem großen Overhead und dadurch auch zu einer schlechteren Perfomanz bei der Anfrageauswertung.

Bei einer vektorisierten Verarbeitung der Daten werden die zuvor angesprochenen Probleme umgangen. Dadurch, dass Funktionen für einen ganzen Block von Daten aufgerufen werden, machen entsprechende Kosten sich deutlich weniger bemerkbar. Gleiches gilt auch für den Overhead durch die Auflösung der Datentypen. Durch eine blockweise Verarbeitung der Daten werden die auftretenden Kosten pro Tupel amortisiert.
Darüber hinaus gibt es noch weitere Probleme bei der tupelweisen Verarbeitung im klassischen *Volcano Iterator Model*. So erreichen die eingesetzten Algorithmen inhärent eine geringere Cacheausnutzung. Dadurch, dass innerhalb eines Funktionsaufrufs nur ein Tupel verarbeitet wird, entsteht kaum eine Lokalität der Daten. Dies ist ein Grund warum Datencaches weniger gut genutzt werden können, da es zu vielen Fehlzugriffen kommt. Neben vielen Fehlzugriffen bei den Datencaches kommt es auch zu vielen Fehlzugriffen bei den Caches, die die Maschineninstruktionen vorhalten. Dies hängt eng mit der zuvor erwähnten Problematik von vielen Funktionsaufrufen zusammen. Denn so kommt es zu einer geringen Lokalität im auszuführenden Programmcode, wodurch sich identische Instruktionen, die im Cache gehalten werden könnten, viel seltener wiederholen. Außerdem bietet eine Tupel für Tupel Verarbeitung weniger Spielraum für Compileroptimierungen wie beispielsweise das Abrollen von Programmschleifen. Denn gerade Schleifen kommen bei einer Verarbeitung von Tupeln eher weniger zum Einsatz, weil dafür schlichtweg nicht genug Daten vorhanden sind, um über diese in einer Schleife zu iterieren. Im Allgemeinen führt dies insgesamt dazu, dass ein relativ schlechter IPC (*instructions per cycle*) Wert erreicht wird.
All diese angesprochenen Probleme können mit einer vektorisierten Verarbeitung der Daten angegangen werden. Denn hier können kompakte Programmschleifen benutzt sowie die Anzahl der Funktionsaufrufe reduziert werden. Unterm Strich wird so ein höherer IPC Wert, eine effizientere CPU Nutzung und auch eine bessere Nutzung der Caches erreicht.
Im Vergleich zu einer *column-at-a-time* Verarbeitung, mit der man ebenfalls versucht viele dieser Probleme in den Griff zu bekommen, ist bei einer vektorisierten Verarbeitung zudem auch die Speicherauslastung geringer. Denn hier müssen (mit Ausnahme der blockierenden Operatoren) nur Zwischenergebnisse und nicht ganze Spalten materialisiert und im Speicher gehalten werden. Des weiteren erreicht eine vektorisierte Verarbeitung im Vergleich zu einer spaltenweisen Verarbeitung auch eine geringere Latenz, da finale Zwischenergebnisse direkt angezeigt werden können. Es muss nicht erst das gesamte Ergebnis fertig erstellt worden sein. [17]

3.2 Datenhaltung *(Jan Stallmann)*

Eine zentrale Aufgabe eines Datenbankmanagementsystems ist die Speicherung und Verwaltung von Daten. Diese Aufgabe wird in SliceDB von dem *Storage-Layer* übernommen. Der *Storage-Layer* übernimmt damit das Vorhalten der Daten, die Verwaltung des Hauptspeichers und das Laden und Speichern der Daten. Aufgrund des gewählten Umgangs mit verschiedenen Datentypen, übernimmt dieser auch das Delegieren der Aufrufe der Operationen an die Funktionen für den entsprechenden Datentyp. Zusätzlich stellt der *Storage-Layer* verschiedene Datenstrukturen bereit, wie z. B. die Klasse `Slice`, mit der ein Zwischenergebnis für den Datenaustausch gespeichert werden kann.

3.2.1 Datentypen *(Milad Nayebi)*

Wie zu Beginn bereits erwähnt, wird die Verwaltung des Hauptspeichers und das Laden und Speichern der Daten von dem *Storage-Layer* übernommen. Die geladenen und gespeicherten Daten unterscheiden sich in Datentypen, die ihren speziellen Eigenschaften haben. Diese Eigenschaften unterscheiden jeden Datentyp von anderen Datentypen. Das Einsatzgebiet des Datentyps wird auch durch diese Eigenschaften definiert. Die Datentypen stellen unterschiedliche Wertebereiche zur Verfügung. Die Programmiersprachen enthalten einen festen Bestandteil, der die elementare Datentypen sind. Also die elementare Datentypen besitzen jeweils auch ein eigenes Schlüsselwort. Als Beispiel kann man *int, double, float, char etc.* nennen. Die Art der Verwendung der elementaren Datentypen sind unterschiedlich, z.B. Zeichnen, Zahlen oder Wahrheitswert. Sie unterscheiden sich auch in ihrem Wertebereich.
SliceDB unterstützt RID, Bool, Integer, Float und String. Bool speichert Werte, die zwei Zustände sein können, entweder True oder False. Integer repräsentiert die ganzen zahlen. Also Integer ist ein Datentyp, der ganzzahlige Werte speichert. Für das Unterstützen von Integer benutzen wir int64_t. Also int64_t ist ein Integer-Typ, der genau 64bits breit ist. RID repsäntiert std::size_t, in dem Längen von Strings und anderen Speicherbereichen angegeben werden sollen. Außerdem kann std::size_t die maximale Größe eines theoretisch möglichen Datentyps speichern. Der nächste von SliceDB unterstützte Datentyp ist Float, das Gleitkommazahlen mit Vorzeichen speichert. Für das Speichern von Char-Sequenzen wird String(std::array<char, 32>) von SliceDB unterstützt. Also *array* definiert eine Klasse Vorlage zum Speichern fester Größe Sequenzen von Objekten. Eine Instanz von Array <T, N> speichert N Elemente vom Typ T, dass size() == N invariant ist(Die Größe von N muss ein Vielfaches von 8 sein). Da für die Ein- und Ausgabe std::string verwendet wird, wandeln die beiden Funktionen

- *String_to_StringArray*
- *StringArray_to_String*

diese Datentypen entsprechend um.

3.2.2 Repräsentation der Daten *(Jan Stallmann)*

Mit der Zielsetzung ein Datenbankmanagementsystem zu entwickeln, das die Daten im Hauptspeicher hält und für analytische Workloads optimiert ist, fiel schnell die Entscheidung für ein spaltenorientiertes Speicherformat. Dieses Format wird auch von den meisten, in der Seminarphase betrachteten, Datenbankmanagementsysteme verwendet. Ein spaltenorientiertes Speicherformat hat gegenüber einem klassischen zeilenorientiertem Speicherformat (*n-ary Storage Model*) den Vorteil, dass lesende Zugriffe von einzelnen Spalten bzw. sehr vielen Tupeln einer Tabelle, vor allem im Hauptspeicher, sehr effizient möglich sind. Dies resultiert aus der besseren Cache-Nutzung durch die konsekutive Speicherung der Elemente einer Spalte, die auch vorteilhaft für den Einsatz von Kompressionsverfahren ist. Der Nachteil eines spaltenorientierten Speicherformats ist, dass ändernde Zugriffe einzelner Tupel aufwendig sind, da die einzelnen Elemente an verschiedenen Speicherstellen gespeichert sind. Dies hat jedoch keinen Einfluss auf die hier

betrachteten analytischen Workloads, da diese ausschließlich aus vielen lesenden Zugriffen auf die gesamten Tabellen bestehen.
Es hat sich aber noch die Frage gestellt, ob ein einfaches *Decomposition Storage Model* oder das *Partition Attributes Accross* Format eingesetzt werden soll. Bei dem *Decomposition Storage Model* werden jeweils alle Elemente einer Spalte konsekutiv gespeichert. Das *Partition Attributes Accross* Format ist eine Kombination aus einem zeilen- und spaltenorientierten Speicherformat. Dabei wird jeweils eine Teilmenge von Tupeln zusammen gespeichert, wobei die Elemente einer Spalte wieder konsekutiv gespeichert werden. Mit diesem Format wird versucht, die verschiedenen Vorteile der beiden Formate zu vereinen. Dies ist jedoch nicht vollständig möglich, da sich die Vorteile der beiden Formate gegenläufig verhalten. So hat man sich gegen das *Partition Attributes Accross* Format entschieden, da dieses aufwendiger zu implementieren ist und der Vorteil für die hier betrachteten Workloads nicht ausschlaggebend ist.
Damit verwendet der *Storage-Layer* ein spaltenorientiertes Speicherformat mittels *Decomposition Storage Model*, da dieses einfacher zu implementieren ist, eine gute Performance verspricht und die Möglichkeit bietet, in Zukunft einfach um Kompressionsverfahren erweitert zu werden.

Typeverwaltung. Zu Beginn der Umsetzung des spaltenoerientierten Speicherformats, ist das Problem aufgetreten, dass Spalten mit verschiedenen Datentypen in einer Struktur gespeichert werden sollten. Um dies möglich zu machen, wurde eine Spalten-Repräsentation eingeführt, die unabhängig von dem Typ der Daten ist. Diese wird im weiteren als `BaseColumn` Klasse bezeichnet. Als Spezialisierung dieser Klasse wurde die `TypedColumn` Klasse eingeführt, um jeweils Spalten mit einem Datentyp zu repräsentieren. Dafür ist die Klasse `TypedColumn` ein *Template* und wird für alle unterstützten Datentypen explizit spezifiziert. Mit dieser Idee geht das Ziel einher, dass der Datentyp der Daten einer Spalte hinter der `BaseColumn` Klasse versteckt wird. Dies wird umgesetzt, indem die `Base Column` Klasse die abstrakte Definition der Operatoren auf einer Spalte enthält und die Deklaration in der `TypedColumn` Klasse bzw. einer Spezialisierung dieser vorhanden ist. Damit wird der Polymorphie-Mechanismus von C++ genutzt, um für den Aufruf eines Operators, die Funktion für den entsprechenden Datentypen auszuführen. Dies funktioniert nur vollständig für Operatoren, die nur eine Spalte als Eingabe haben, reduziert aber auch den Aufwand für Operatoren, wie z. B. *Joins*, die mehrere Spalten als Eingaben haben.
Um eine reibungslose Zusammenarbeit zu ermöglichen, werden die Operatoren von der `TypedColumn` Klasse an die Klasse `Executor` delegiert, damit dort zusammengefasst die Operatoren deklariert werden können. Veranschaulicht wird die Idee der Typeverwaltung in Abbildung 3.5.

Vektorisierte Datenverarbeitung. Die beiden vorgestellten Klassen `Base Column` und `TypedColumn` sind abstrakte Klassen. Das bedeutet, dass diese keine Informationen über das eigentliche Speicherformat der Daten enthalten. Für die eigentliche Speicherung der Daten verwendet der *Storage-Layer* die Klassen `Page Column` und `ArrayColumn`, die die Klasse `TypedColumn` spezialisieren. Dabei ist zu beachten, dass diese wie die Klasse `TypedColumn` *Templates* sind und für alle unterstützten Datentypen explizit spezifiziert werden. Mit Objekten der Klasse `PageColumn` werden jeweils die Spalten der Tabellen der aktuell geladenen

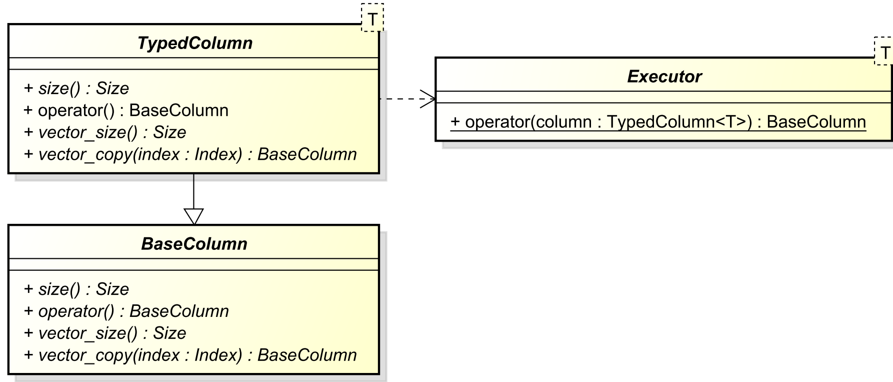


Abbildung 3.5: Ausschnitt aus dem Klassendiagramm zur abstrakten Spalten-Repräsentation.

Datenbank im Hauptspeicher repräsentiert. Die Funktionsweise dieser Klasse wird im späteren Kapitel 3.2.3 genauer beschrieben.

Für die vektorisierte Datenverarbeitung wird eine Repräsentation eines Vektors benötigt. Um den Aufwand gering zu halten, lassen sich die bereits beschriebenen Klassen wiederverwenden, indem ein Vektor als eine Spalte (Objekt der Klasse `BaseColumn`) repräsentiert wird. Damit für die Vektoren eine einfachere Datenstruktur, als für die permanente Speicherung der eigentlichen Spalten, zur Verfügung steht, ist für diese die Klasse `ArrayColumn` vorgesehen. In dieser Klasse werden die Daten einer Spalte bzw. eines Vektors in einem einfachen C-Array gespeichert.

Um die vektorisierte Datenverarbeitung zu ermöglichen, lassen sich aus einer Spalte Vektoren extrahieren. Die Anzahl der Vektoren, in die eine Spalte geteilt werden kann, gibt die jeweilige verwendete Implementierung vor. So lässt sich diese Anzahl mit der Funktion `vector_size()` ermitteln. Die einzelnen Vektoren in Form eines Objekts der Klasse `BaseColumn` lassen sich dann mit der Funktion `vector_copy(index : Index)` extrahieren, wobei `index` die Position des entsprechenden Vektors angibt. Diese Funktionen sind zusammen mit den in diesem Abschnitt vorgestellten Klassen in Abbildung 3.6 veranschaulicht. Des Weiteren hält die Klasse `TypedColumn` einen *Offset*, der für einen Vektor angibt, an welcher Position der zugrunde liegenden Spalte dieser beginnt. Für die zugrunde liegenden Spalten, die also die eigentlichen Spalten einer Tabelle repräsentieren, ist der *Offset* gleich 0. Zur Vereinfachung ist der *Offset* nicht in der Abbildung dargestellt.

Mit der Modellierung der Spalten und Vektoren als Objekte der Klasse `Base Column` ist es damit möglich, die Operatoren sowohl für eine *Column-At-A-Time* Verarbeitung auf der kompletten Spalte einer Tabelle auszuführen als auch für eine *Vector-At-A-Time* Verarbeitung auf einzelne Vektoren auszuführen. Dies bietet also eine große Flexibilität für das auf den Daten ausgeführte Verarbeitungsmodell.

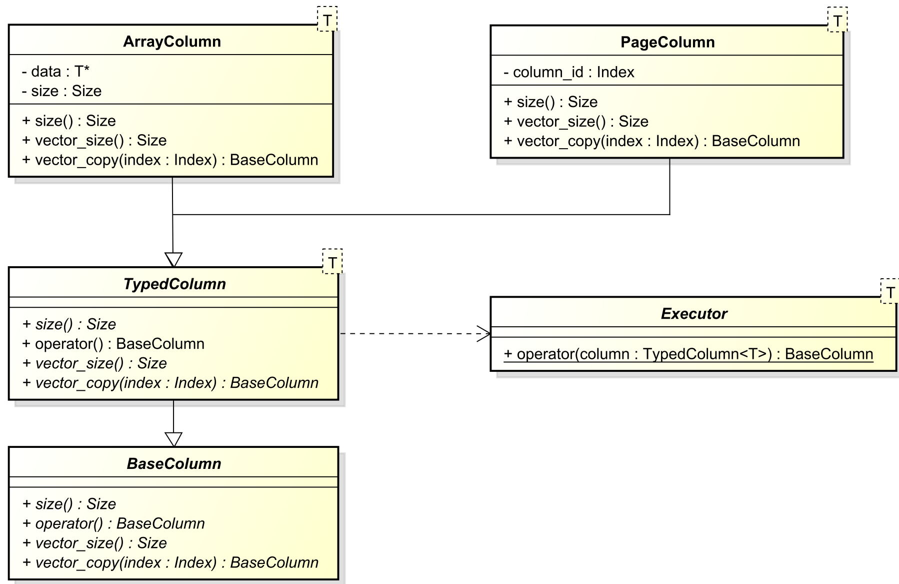


Abbildung 3.6: Ausschnitt aus dem Klassendiagramm zur eigentlichen Spalten-Repräsentation.

3.2.3 Speicherverwaltung

(Majuran Rajakanthan)

Zu Beginn hat sich die Frage gestellt, ob *pages* oder *memory mapped files* für die Speicherverwaltung verwendet werden sollen. Durch memory mapped files kann mit den Daten auf der Festplatte so gearbeitet werden, als wären diese im Hauptspeicher, wodurch I/O Zugriffe enorm reduziert werden. Der Vorteil von memory mapped files ist der geringe Implementierungssaufwand. Das Betriebssystem verwaltet automatisch den I/O. Daraus leitet sich auch automatisch der Nachteil von memory mapped files ab, da somit keine Kontrolle der Speicherverwaltung mehr möglich ist. Die Freiheit, wie bestimmte Pages auf die Festplatte geschrieben und anschließend gelesen werden geht verloren, da das Betriebssystem selbst entscheidet in welche Reihenfolge und was überhaupt geschrieben werden soll. Im Gegensatz hierzu kann mit den Standard I/O Methoden vorhersehbar geschrieben werden. Aus diesen Gründen wurde für eine eigene Speicherverwaltung, durch einen Buffer Manager, mit pages entschieden.

Der Buffer Manager hat die Aufgabe Werte aus einer Binärdatei von der Festplatte, in *Pages* in den Buffer Pool zu laden und diese für die Prozesse im Hauptspeicher verfügbar zu machen. Dabei implementiert der Buffer Manager eine Page-Ersetzungsstrategie, um einerseits die Festplattenzugriffe zu minimieren und andererseits den vorhandenen Hauptspeicher zu nutzen. Wenn eine Page angefragt wird und diese im Hauptspeicher existiert, soll der Buffer Manager die Adresse dieser übergeben. Ist diese Page noch nicht im Hauptspeicher, wird diese bei vorhandenem Platz im Buffer Pool, in den Hauptspeicher geladen und anschließend übergeben. Ansonsten, wird mit Hilfe der Ersetzungsstrategie

eine Page im Buffer Pool ersetzt. Des Weiteren ist der Buffer Manager für die Persistenz der Pages zuständig.

Aufbau des Buffer Manager

Die Klasse *BufferManager* verwaltet den Buffer Pool und ist für die interne Zuordnung der Spalten und Tabellen zu den Pages verantwortlich. Es existieren *Getter*-Methoden, über welche Informationen zu den Pages erhältlich sind.

- Rückgabe der *Default*-Größe einer Page (*default_size()*)
- Rückgabe der Anzahl der Pages einer Spalte (*page_count()*)
- Rückgabe eines *Boolean* Wertes, ob die Page voll ist (*page_full()*)
- Rückgabe der Anzahl der Elemente einer Page (*page_size()*)

Buffer Pool

Der Buffer Pool ist eine Sammlung aus *BufferFrame* Objekten, wobei jedes Frame-Objekt eine Page verwaltet. Zusätzlich hat jeder Frame die Attribute *PageNo*, *pinCount* und *accBit*, um jede Page einem Frame zuordnen zu können. Die *PageNo* identifiziert dabei eine Page, währenddessen der *pinCount* und *accBit* für die Ersetzungsstrategie benötigt werden. Dies wird im späteren Abschnitt 3.2.3 näher beschrieben. Diese Buffer Frames haben einen *Previous* und *Next Pointer* auf die anderen Frame-Objekte. Damit werden diese als verkettete Liste gespeichert (s. Abb. 3.7). Aufgrund der spaltenorientierten Speicherung, stellt eine Liste eine Spalte einer Tabelle dar, deren Elemente nacheinander auf den Pages liegen.

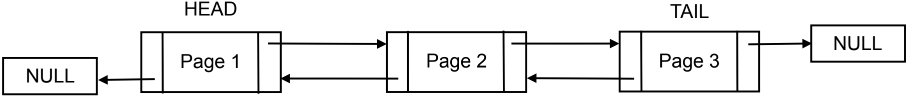


Abbildung 3.7: 3 Frame-Objekte mit Pages

Buffer Manager

Die BufferManager Klasse besitzt einen *struct column_property* genannt, welche folgende Attribute einer Spalte besitzt:

- Einen *String*, welcher den Pfad zu einer Spalte einer Tabelle enthält
- Das Anfangs- und Endobjekt der verketteten Liste
- Die Anzahl der Pages im Buffer Pool
- Die Anzahl der Pages in der binären Datei

Mit Hilfe einer *Map*-Funktion können diese Eigenschaften einer Spalte, einer einzigartigen *columnID*, welche eine Spalte identifiziert, zugeordnet werden. Diese columnIDs werden in der *init_column()* Methode erzeugt und als Rückgabewert dieser zurückgegeben.
Die Methode legt außerdem leere Binärdateien für jede Spalte an. Alle grundlegenden Funktionen des Buffer Manager werden mit dieser columnID als Parameter aufgerufen.

- *create_page()*: Erstellt eine leere Page mit einer zuvor definierten Page Größe und ein Frame-Objekt zu dieser.
- *free_column()*: Löscht alle Frame-Objekte im Speicher.
- *get_page()*: Neben der columnID wird eine *pageID* als Parameter mitgegeben. Anhand dieser werden die Frames durchsucht und die passende Page zurückgegeben. Falls die Page nicht im Speicher existiert, wird diese von der Festplatte aus der Binärdatei geladen. Anschließend wird durch den Page-Ersetzungsalgorithmus ein Frame-Objekt ausgewählt, die dort vorhande Page mit dieser ersetzt und zurückgegeben.
- *delete_page()*: Neben der columnID wird eine pageID als Parameter mitgegeben. Die zu löschende Page wird gesucht und samt dem Frame-Objekt gelöscht.
- *delete_column()*: Die Binärdatei der Spalte wird mit einer leeren Binärdatei überschrieben.
- *load_column()*: Eine Spalte wird in Pages geladen.

Persistenz der Pages

Die Methode *writeToFile* ist für die Persistenz der Pages zuständig. Dabei existiert diese Methode zweimal mit verschiedenen Parametern. Einmal nur mit einer columnID, um alle Pages einer Spalte zu speichern. Diese Methode wird durch die öffentliche Methode *persistence()* des Buffer Managers aufgerufen. Ruft man die Methode mit einer columnID und pageID auf, wird nur die betroffene Page persistent gemacht. Beides sind private Methoden.
Jede Binärdatei einer Spalte hat einen *Header* am Anfang mit der Anzahl der Pages, die sich in der Datei befinden. Mit Hilfe dieser Information wird bei der Persistenz einer Page, an die richtige Stelle in der Binärdatei gesprungen und anschließend der Inhalt der Page an diese Stelle geschrieben. Der Header wird immer beim Schreiben einer neuen Page aktualisiert. Neben dem Header am Anfang der Datei, existiert vor jeder Page in der Datei ein weiterer Header. In diesem steht die Anzahl der Elemente einer Page. Abbildung 3.8 zeigt eine solche Binärdatei mit einer Page, die zehn Elemente enthält.

Beispiel: Eine Binärdatei enthält 4 Pages. Der Header hat also den Wert 4. Vor dem Schreiben einer neuen Page, wird dieser Wert ersetzt. Nun kann anhand der zu schreibenden pageID, z.B 2 mit `_offset + (2 * (_offset + PAGE_SIZE_BYTES))` an die richtige Stelle gesprungen und dort der Inhalt geschrieben werden. Das erste *_offset* steht dabei für den Platz, den der Header beansprucht und das zweite *_offset* für den Header unmittelbar vor der Page.

```
maju@ubuntu:~/workspace/in-pg589/slicedb$ hexdump -C lo_shipmode.bin
00000000  01 00 00 00 00 00 00 00  0a 00 00 00 00 00 00 00  |................|
00000010  54 52 55 43 4b 00 00 00  80 83 9f b7 fc 7f 00 00  |TRUCK...........|
00000020  20 83 9f b7 fc 7f 00 00  2e 4d 41 49 4c 00 00 00  | ........MAIL...|
00000030  00 80 83 9f b7 fc 7f 00  00 20 83 9f b7 fc 7f 00  |......... ......|
00000040  00 2e 52 45 47 20 41 49  52 00 80 83 9f b7 fc 7f  |..REG AIR.......|
00000050  00 00 20 83 9f b7 fc 7f  00 00 2e 41 49 52 00 00  |.. ........AIR..|
00000060  00 00 00 80 83 9f b7 fc  7f 00 00 20 83 9f b7 fc  |........... ....|
00000070  7f 00 00 2e 52 41 49 4c  00 00 00 00 80 83 9f b7  |....RAIL........|
00000080  fc 7f 00 00 20 83 9f b7  fc 7f 00 00 2e 46 4f 42  |.... ........FOB|
00000090  00 00 00 00 00 80 83 9f  b7 fc 7f 00 00 20 83 9f  |............. ..|
000000a0  b7 fc 7f 00 00 2e 53 48  49 50 00 00 00 00 80 83  |......SHIP......|
000000b0  9f b7 fc 7f 00 00 20 83  9f b7 fc 7f 00 00 2e 41  |...... ........A|
000000c0  49 52 00 00 00 00 00 80  83 9f b7 fc 7f 00 00 20  |IR............. |
000000d0  83 9f b7 fc 7f 00 00 2e  41 49 52 00 00 00 00 00  |........AIR.....|
000000e0  80 83 9f b7 fc 7f 00 00  20 83 9f b7 fc 7f 00 00  |........ .......|
000000f0  2e 52 41 49 4c 00 00 00  00 80 83 9f b7 fc 7f 00  |.RAIL...........|
00000100  00 20 83 9f b7 fc 7f 00  00 2e                    |. ........|
```

Abbildung 3.8: Inhalt einer Binärdatei mit einer Page (erste Markierung), die zehn Elemente enthält (zweite Markierung in Hexadezimal)

Ersetzungsstrategie

(Philipp Krause)

Als Ersetzungsstrategie wurde der *CLOCK-Algorithmus* gewählt, da dieser versucht das *LRU-Verhalten* (Last Recently Used) nachzubilden, jedoch - im Vergleich zu diesem - einfacher zu implementieren ist [22].
Bei der LRU-Strategie wird die Seite, die am längsten nicht referenziert wurde, ausgelagert. Die Idee ist demnach, dass Seiten, die häufig verwendet werden, im Cache verbleiben während Seiten mit geringem Nutzungsgrad verdrängt werden. Die Implementierung erweist sich in sofern als schwierig, als dass Seiten mittels Warteschlange und Zeitstempel gemäß des Referenzzeitpunktes, der immer aktualisiert wird, verwaltet werden müssen [41].
Bei der Clock-Strategie werden die Seiten in einer Ringliste verwaltet, wobei ein Pointer immer auf die aktuelle Seite zeigt. Wenn ein Seitenfehler auftritt, wird bei dieser Seite überprüft, ob das Referenzbit der Seite auf wahr oder falsch gesetzt ist. Bei jedem Seitenzugriff wird das Bit auf wahr gesetzt. Ist es falsch, wird die Seite ausgelagert und die neue Seite an der gleichen Position eingefügt. Anschließend wird der Zeiger auf die nächstfolgende Seite verschoben. Ist das Referenzbit wahr, wird es auf falsch gesetzt und der Zeiger wird um eine Position verschoben. Dieser Prozess wird solange wiederholt, bis eine Seite gefunden wird, dessen Referenzbit falsch ist und die Seite aufgrund dessen ausgelagert wird. Der Vorteil dieser Vorgehensweise besteht darin, dass häufig benutzte Seiten seltender ausgelagert werden [22].
Der Clock-Algorithmus wird in Abb. 3.9 veranschaulicht. Dieser wird in der Methode get_page(), in der eine Seite angefragt wird, aufgerufen, sofern der BufferManager über keinen freien Speicherplatz verfügt und daher eine Seite für das Einlagern der angefragten, neuen Seite verdrängen muss. Die zu verdrängende Seite wird dazu auf die Festplatte geschrieben und die angefragte Seite wird von der Festplatte geladen und erhält den BufferFrame der ausgelagerten Seite. Die genaue Funktionsweise der Methode *clockAlgo()* soll im Folgenden näher

beschrieben werden.

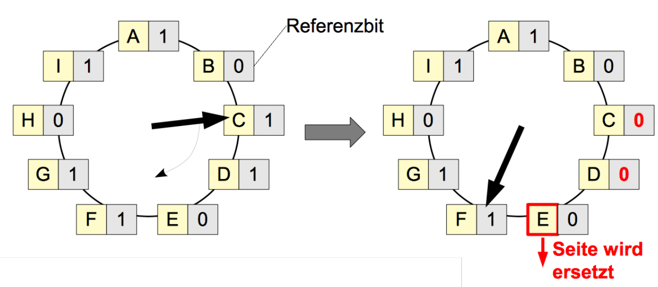


Abbildung 3.9: Clock-Algorithmus [23]

Die Methode clockAlgo() wird aufgerufen, sobald das Programm den festgesetzten Wert der Speicherbelegung überschritten hat. Die Methode liefert als Rückgabewert einen Zeiger auf einen BufferFrame, dem anschließend eine neue Seite zugeordnet wird.
Der Algorithmus iteriert durch die BufferFrames aller Spalten und überprüft für jeden BufferFrame, ob dessen pinCount=0 und das accBit entweder wahr oder falsch ist.
Die Variable pinCount wird in der Klasse pagecolumn vor jedem Seitenzugriff durch die Methode pin_page() erhöht und erst nach dem Zugriff der jeweiligen Funktion durch die Methode unpin_page() wieder herabgesetzt. Damit wird vehindert, dass Seiten verdrängt werden können, auf die derzeit noch zugegriffen wird.
Das accBit kann als *zweite Chance* angesehen werden, da es einer Seite ermöglicht im Cache bestehen zu bleiben, falls sie während des Zeigerumlaufs erneut referenziert wird. Bei jedem Seitenzugriff wird dieses in pin_page() auf wahr gesetzt. Bei der zyklischen Suche in clockAlgo() nach dem BufferFrame, dessen Seite verdrängt werden soll, wird das accBit auf falsch gesetzt, wenn es wahr ist und der Zeiger wird auf das nächste BufferFrame-Element gesetzt. Die Seite, bei der das accBit des zugehörigen BufferFrames falsch ist und pinCount=0 beträgt, wird verdrängt.

PageColumn Klasse

(Majuran Rajakanthan)

Die *PageColumn*-Klasse repräsentiert die Spalten einer Tabelle im Hauptspeicher und ist zudem für die Speicherung der Daten über den Buffer Manager zuständig. Die Klasse ist dabei ein *Template*, welches die Klasse *TypedColumn* spezialisiert. Um ein PageColumn Objekt zu erzeugen, wird der Konstruktor mit dem Pfad zu der Binärdatei der Spalte als Parameter aufgerufen. Dabei wird eine Instanz des Buffer Managers angefordert und eine columnID generiert. Dadurch können nun alle Anfragen aus der höheren Schicht nach unten an den Buffer Manager delegiert werden.
Durch einen Aufruf der Klassenmethoden wird eine Operation auf einer kompletten Spalte durchgeführt. Dabei werden die Methodenaufrufe aus der überlie-

genden Schicht von der PageColumn Klasse nach unten zum Buffer Manager delegiert, welcher verschiedene Operationen auf den Pages durchführt und das jeweils angeforderte Ergebnis zurückgibt. Ein Beispiel dieser Vorgehensweise in der Methode für die Rückgabe der Anzahl aller Elemente, die auf allen Pages einer Spalte liegen, wird in Abb. 3.1 dargestellt.

Listing 3.1: Beispielcode für die Rückgabe der Anzahl aller Elemente einer Spalte

```
 1: template<typename T>
 2: Size PageColumn<T>::size() const {
 3:   Size sum = 0;
 4:   Index i;
 5:   for (i = 0; i < _page_count; i++) {
 6:     sum += BufferManager::getInstance()
 7:            .page_size<T>(_column_id, i);
 8:   }
 9:   return sum;
10: }
```

Darüber hinaus ist PageColumn für die *Pin* und *Unpin* Aufrufe auf pages zuständig, wenn mit den Elementen einer Page gearbeitet wird, um ein eventuelles Ersetzen der Page durch die Ersetzungsstrategie zu verhindern. Dies ist zum Beispiel in der Methode *vector_copy()* erforderlich. Dabei kopiert *vector_copy()* die Daten aus dem PageColumn in ein *ArrayColumn*, welche Vektoren repräsentiert, die anschließend für die vektorisierte Datenverarbeitung benötigt wird.
Eine weitere wichtige Methode für die Hauptfunktionalität der PageColumn ist *append()*. Mit dieser werden, über den Buffer Manager, einer Page Werte hinzugefügt, wenn diese noch nicht voll ist. Ansonsten wird eine neue Page erstellt.

3.2.4 Dictionary Compression

(Philipp Krause)

Als Kompressionsalgorithmus wurde der Lempel-Ziv-Welch-Algorithmus (LZW), der eine verbesserte Variante hinblicklich der Laufzeit des LZ77- und LZ78-Algorithmus darstellt, gewählt. Im Folgenden soll zunächst die allgemeine Funktionsweise des Algorithmus und anschließend die Implementation im BufferManager erläutert werden.

Funktionsweise

Das Wörterbuch der Standardimplementation umfasst 4069 Einträge und eine Codelänge von 12 Bit. Wie auch bei den Vorgängern LZ77 und LZ78 wird das Wörterbuch zur Laufzeit, d.h. während der Komprimierungs- bzw. Dekomprimierungsphase, aufgebaut. Ausgenommen davon sind die ersten 256 Einträge, die bereits sämtliche, einzelne Zeichen enthalten, um eine Übertragung dieser zu verhindern. Der Algorithmus sucht zur Laufzeit nach Mustern in den Zeichenketten. Sind diese Muster noch nicht im Wörterbuch vertreten, werden sie diesem hinzugefügt. Andernfalls wird anstelle des Musters der entsprechende Index

zurückgegeben. Dieses Vorgehen wird in Abb. 3.10 und 3.11 veranschaulicht. In

Kodierung:

Eingabe	Erkanntes Muster	Neuer Wörterbucheintrag
LZWLZ78LZ77LZCLZMWLZAP	L	LZ (=256)
ZWLZ78LZ77LZCLZMWLZAP	Z	ZW (=257)
WLZ78LZ77LZCLZMWLZAP	W	WL (=258)
LZ78LZ77LZCLZMWLZAP	LZ	LZ7 (=259)
78LZ77LZCLZMWLZAP	7	78 (=260)
8LZ77LZCLZMWLZAP	8	8L (=261)
LZ77LZCLZMWLZAP	LZ7	LZ77 (=262)
7LZCLZMWLZAP	7	7L (=263)
LZCLZMWLZAP	LZ	LZC (=264)
CLZMWLZAP	C	CL (=265)
LZMWLZAP	LZ	LZM (=266)
MWLZAP	M	MW (=267)
WLZAP	WL	WLZ (=268)
ZAP	Z	ZA (=269)
AP	A	AP (=270)
P	P	

Ausgabe: L Z W <256> 78 <259> 7 <256> C <256> M <258> Z A P

Abbildung 3.10: Beispiel: Kodierung des LZW-Algorithmus

jedem Schritt der Kodierung des Beispiels ist links der Inhalt der Zeichenkette angegeben. In der Mitte steht das längste Muster, das in dem jeweiligen Schritt gefunden wurde. Rechts werden die neuen Wörterbucheinträge mit den dazugehörigen Indizes abgebildet.
Im ersten Schritt besteht das längste Muster aufgrund der Vorbelegung der Einzelzeichen mit den Codewerten 0 bis 255 aus dem ersten Buchstaben (L) der Zeichenkette. Zusätzlich wird ein weiteres Zeichen (Z) betrachtet und an das vorangegangenen Zeichen angehängt. Das neue Muster (LZ) erhält nun den neuen Index 256. Dieses Vorgehen wiederholt sich entsprechend bis die gesamte Zeichenkette kodiert ist.
Auch bei der Dekodierung ist das Wörterbuch mit den Einträgen von 0 bis 255 vorinitialisiert. Erneut ist L zunächst das längste, gefundene Muster, das zurückgegeben wird (s. Spalte p). Das zweite Zeichen stellt Z dar und wird gespeichert (s. Spalte C). Beide Zeichen werden anschließend konkateniert und in das Wörterbuch mit dem Index 256 aufgenommen. Der Index ist somit bei der Komprimierung und Dekomprimierung identisch. Das Verfahren wird ebenfalls so lange analog wiederholt, bis die Zeichenkette vollständig dekomprimiert ist.

Implementation

Für die Komprimierung bzw. Dekomprimierung der Seiten gibt es folgende zwei Funktionen:

vector<CodeType> compress_lzw1(char* uncompressed, Size length)
const char* decompress_lzw1(vector<CodeType> compressed)

Die Komprimierungsmethode nimmt als Parameter ein char-Array und dessen

Dekodierung:

Eingabezeichen	c	Neuer Wörterbucheintrag	p
L			L
Z	Z	LZ (=256)	Z
W	W	ZW (=257)	W
<256>	L	WL (=258)	LZ
7	7	LZ7 (=259)	7
8	8	78 (=260)	8
<259>	L	8L (=261)	LZ7
7	7	LZ77 (=262)	7
<256>	L	7L (=263)	LZ
C	C	LZC (=264)	C
<256>	L	CL (=265)	LZ
M	M	LZM (=266)	M
<258>	W	MW (=267)	WL
Z	Z	WLZ (=268)	Z
A	A	ZA (=269)	A
P	P	AP (=270)	P

Ausgabestring: L Z W LZ 7 8 LZ7 7 LZ C LZ M WL W A P

Abbildung 3.11: Beispiel: Dekodierung des LZW-Algorithmus

Länge entgegen und liefert einen Vektor des Typs *CodeType* zurück. CodeType entspricht dabei einem vorzeichenlosem 16 Bit-Integer. Das char-Array erhält das Array einer Seite, in dem die Daten gespeichert sind. Der Vektor enthält nach dem in Abschnitt 3.2.4 beschriebenen Verfahren die kodierten Werte. Der Dekomprimierungsfunktion wird dieser Vektor übergeben und von dieser entschlüsselt bis das dekomprimierte char-Array anschließend zurückgegeben wird.
Um die Daten kodiert in den Binärdateien ablegen zu können, wurde der in Abschnitt 3.2.3 beschriebene Aufbau der Binärdateien um einen weiteren Header erweitert. Dieser Header enthält die *kodierte* Größe einer Seite, die für das Anlegen des Vektors zur Dekomprimierung nach dem Lesen einer Binärdatei notwendig ist. Der Aufbau wird in Abb. 3.12 dargestellt. Zudem wird nun anstelle des unkomprimierten char-Arrays der Vektor, der die komprimierten Daten der Seite enthält, in der Binärdatei gespeichert. Die Komprimierung und das Schreiben des Vektors wurde in die Methode *writeToFile(...)* integriert. Das Lesen sowie die Dekomprimierung ist in der Methode *getPage(...)* implementiert.

Header #Pages	**Header** Size Page 1 (compr.)	**Header** Size Page 1 (uncomp.)	**Page 1** (compr.)	**Header** Size Page 2 (compr.)	**Header** Size Page 2 (uncomp.)	**Page 2** (compr.)

Abbildung 3.12: Aufbau der Binärdateien

In dem aktuellen Stand des Projekts musste die Dictionary Compression deaktiviert werden, da das Auslesen der Seite nicht zu korrekten Ergebnissen führte. Die Ursache dafür konnte trotz diverser Debug-Maßnahmen nicht gefunden werden, weshalb die separate Datei *buffermanagerDictionary.cpp* und die dazugehörige Header-Datei in das Projekt eingefügt, aber nicht eingebunden wurden. Die Kor-

rektheit der Komprimierungs- sowie Dekomprimierungsmethode konnte anhand von manuell eingefügten Zeichenketten - im Gegensatz zu der Verwendung von Seiten - nachgewiesen werden.

3.2.5 Operationen auf Spalten *(Harun Kara)*

Bei den Operationen auf Spalten kann man grob zwischen Funktionalitäten für einfache Spalten und Operatoren für die Ausführung von Plänen auf Vektoren unterscheiden. Im Folgenden sind die wichtigsten Funktionalitäten für einfache Spalten mit einer kurzen Erklärung aufgelistet.

- `size()`: Gibt die Anzahl der Elemente in einer Spalte zurück.
- `append(value)`: Fügt zum Zeitpunkt des Auffüllens der Spalte, das Element *value* an dessen Ende hinzu.
- `insert(values)`: Fügt zu einem beliebigen Zeitpunkt die Elemente *values* in die Spalte ein.
- `reserve(size)`: Alloziert neuen Speicher, um size Elemente in die Spalte einzufügen.
- `shrink_to_fit()`: Gibt unbenötigten Speicher wieder frei.
- `vector_size()`: Gibt die Anzahl der möglichen Vektoren zurück.
- `vector(index)`: Erzeugt einen Vektor, der auf dieselben Daten zeigt.
- `vector_copy(index)`: Erzeugt einen Vektor mit einer Kopie der Daten.
- `partition_copy(begin, end)`: Erzeugt einen Vektor mit einer Kopie der Daten von einem Bereich.
- `persistence()`: Macht die Daten persistent.
- `remove_persistence()`: Entfernt die persistenten Daten vom Dateisystem.

Die Operatoren für die Ausführung von Plänen sind kompliziert und die Aufrufe hier delegieren die Berechnung nur an den richtigen Operator über den *Executor*. Die Klasse *Executor* stellt eine Schnittstelle zwischen den Spalten und den Operatoren dar. Die Operationen in der Spalte für das Delegieren werden nur von den *ArrayColumns* implementiert, da solche Operatoren nur auf Vektoren aufgerufen werden. Die einzelnen Operatoren werden hier nicht aufgelistet, da sie in Abschnitt 3.3 im Detail behandelt werden.

3.2.6 Schemaverwaltung *(Harun Kara)*

Ein Schema wird bei SliceDB über die Klassen *Database*, *Table*, *BaseColumn* und ihren Implementierungen repräsentiert. Die Klasse *Database* repräsentiert eine Datenbank und befindet sich ganz oben in der Klassenhierarchie eines Schemas. Über Database kann man zu den einzelnen Tabellen (repräsentiert durch Table-Objekte) navigieren, die in Form einer Liste in Database gespeichert sind. Alle Tabellen sind eindeutig durch einen Namen und ein Index zu unterscheiden. Man kann die Anzahl der Tabellen, und ihren Namen abfragen. Das Anhängen einer

neuen Tabelle und das zurückgeben von Pointern zu Tabellen ist möglich.
Eine Tabelle wird durch die Klasse *Table* repräsentiert. Ein Table-Objekt kennt eine Menge von Spalten vom Typ BaseColumn. Welchen konkreten Typ diese Spalten und ihre Daten besitzen, ist dem Tabellen-Objekt nicht bekannt. Es bietet Funktionen an, um die Größe der Tabelle abzufragen (Anzahl Spalten), eine neue Spalte einzufügen, eine leere Spalte zu erzeugen und einzufügen, eine Spalte zurückzuliefern und um den Namen einer Spalte zurückzuliefern. Table implementiert die Funktion *import_csv*(...), die eine Tabelle aus einer csv-Datei parsed und als Spalten in Table einfügt. Außerdem ruft Table mit einem Befehl auf allen seinen Spalten die $persistence()$-Funktion auf, wodurch die Spalten ihre Daten persistieren (mittels BufferManager). Die Klasse *DatabaseManager* ist in SliceDB die zentrale Organisationseinheit, wenn es um die Schemaverwaltung geht. Der DatabaseManager ist als Singleton implementiert und erlaubt dem Interface, die Schemata effizient verwalten zu können. Man kann eine Datenbank entweder selbst erstellen und dem DatabaseManager übergeben, oder man kann gleich dem DatabaseManager den Befehl geben, intern eine Datenbank mit einem beliebigen Namen zu erstellen. In beiden Fällen verwaltet jede Instanz des DatabaseManager, eindeutig die gerade erstellte Datenbank, bis sie explizit vom Benutzer verworfen wird. Dies ist z.B. der Fall, wenn eine neue/andere Datenbank geladen wird, was nur über den DatabaseManager geschieht. Das Serialisieren von Datenbanken ist auch ausschließlich über den DatabaseManager möglich, wobei nur das Schema serialisiert wird. Spalten greifen auf ihre Daten ausschließlich über den BufferManager zu, der sich auch um das Laden in den Hauptspeicher und das persistieren auf der Festplatte der Daten kümmert. Hierbei sollte man zur Kenntnis nehmen, dass das Persistieren der Daten durch den BufferManager effizienter ist, weil der BufferManager, z.B. bei der Organisation der Daten in Pages, nur die schmutzigen Pages wirklich auf die Festplatte zurückschreibt. Schemata werden in Form von XML-Dateien auf der Festplatte repräsentiert, weil XML-Dateien besser lesbar sind, was das Debugging bequemer macht. Dies bringt keine Performance-Nachteile mit sich, z.B. gegenüber Binary-Files, da für die Repräsention eines Schemas, sowieso nur eine sehr begrenzte Anzahl von Klassen nötig ist. Unterschieden werden die Schemata auf der Platte, bzw. die XML-Dateien, nur anhand ihrer Namen, weswegen die Namen immer eindeutig gewählt werden sollten.

3.2.7 Histogramme *(Harun Kara)*

Für den Planoptimierer von Datenbankmanagementsystemen sind Selektivitätsabschätzungen von Prädikaten oft von großer Bedeutung, z.B. wenn es darum geht, ob ein Index eingesetzt werden soll oder nicht. Histogramme werden häufig eingesetzt, weil sie die nötigen Statistiken liefern, um Selektivitäten abschätzen zu können. Allerdings sind Selektivitätsabschätzungen für den Star Schema Benchmark, für den wir SliceDB optimieren, nicht sehr erfolgversprechend, da eine Gleichverteilung der Datensätze zu erwarten ist.
In der Literatur wird zw. zwei grundlegenden Histogrammtypen unterschieden. Diese sind Equiwidth- und Equidepth-Histogramme. Bei Equiwidth-Histogrammen wird der Wertebereich (X-Achse) in gleichgroße Teilbereiche eingeteilt, sog. Buckets. Die Annahme ist, dass die Tupel innerhalb eines Buckets, gleichverteilt auf den Wertebereich sind. Ausreißer werden somit nicht ausreichend berücksichtigt. In einem Equidepth-Histogramm hingegen werden die Teilbereiche so

gewählt, dass die Anzahl der Tupel in jedem Bucket die Selbe ist. Diese Variante liefert bessere Abschätzungen, da die Buckets mit sehr häufig vorkommenden Werten, kleiner sind und sehr viel weniger (unterschiedliche) Werte enthalten, und damit die Annahme der Gleichverteilung auf einen kleineren Wertebereich angewendet wird, als bei Equiwidth-Histogrammen. Somit fällt die Selektivitätsabschätzung insbesondere von Ausreißern bzw. häufigeren Werten, sehr viel besser aus.
Es können nur für numerische Spalten Histogramme angelegt werden. Grundsätzlich könnte man auch für Spalten vom Datentyp String Histogramme berechnen. Für Strings sind, wenn überhaupt, nur einfache relative Häufigkeitsverteilungen berechenbar (ausgenommen, wenn eine Ordnung auf den Strings definiert ist). Für numerische Datentypen hingegen können viel kompliziertere Histogramme berechnet und abgespeichert werden. Je nach Typ der Spalte, wird während der Übersetzungszeit histogrammspezifischer Code in der Spalte mitübersetzt oder nicht. Dies wird mit der C++-Programmiertechnik *SFINAE* realisiert. Somit wird der Zugriff auf Histogramme über Spalten ermöglicht.
Die Berechnung der Buckets und der Selektivitäten machen ein Histogramm im wesentlichen aus. Hierfür haben wir eine etwas ältere Technik von Gregory Piatetsky-Shapiro aus [54] herangezogen, was unseren Anforderungen genügt. Bei Equi-Width Histogrammen ist es sehr einfach die Bucket-Grenzen festzulegen, nachdem man die (einheitliche) Bucket-Breite bestimmt hat. Bei Equi-Depth Histogrammen müssen aber die Bucket-Tiefen gleich groß sein, sodass die Bucket-Breiten unterschiedlich lang werden. Wir haben dafür nicht einfach den Wertebereich in gleichbreite Buckets eingeteilt und die zugehörigen relativen Häufigkeiten angegeben, sondern alle Werte in der Spalte aufsteigend sortiert und dann die Bucket-Grenzen ähnlich wie bei Equi-Width Histogrammen in gleichbreiten Abständen festgelegt. Somit ist gesichert, dass alle Buckets dieselbe Anzahl an Werten enthalten. Bei Histogrammen ist es am schwierigsten, die Bucket-Breite, bzw. in diesem Fall die Bucket-Tiefe so zu wählen, dass die Histogramme genügend viele Buckets haben, um ihren Sinn und Zweck zu erfüllen, aber trotzdem noch übersichtlich klein bleiben. Wir haben uns aus Zeit- und Komplexitätsgründen für die einfache Faustformel $\sqrt{n}$ (wobei $n =$ Anzahl Werte in der Spalte) entschieden, die unter anderem von Excel für Histogramme benutzt wird [2]. Für die Selektivitätschätzungen werden in [54] zwei Techniken vorgeschlagen, einmal mit dem kleinsten Worst-Case Error und einmal mit dem niedrigsten Average-Case Error. Wir haben die Variante mit dem niedrigsten Average-Case Error implementiert, da laut [54] die Minimierung des Average-Case Error wichtiger für Datenbanksysteme ist.

3.2.8 Zwischenergebnisse *(Jan Stallmann)*

Für das Volcano-Iterator Modell wird bei einem spaltenorientierten Speicherformat eine spezielle Datenstruktur für den Austausch von Zwischenergebnissen benötigt. Der *Storage-Layer* stellt dafür die Klasse `Slice` bereit.
Bei der Verwendung von einer Kombination aus dem Volcano-Iterator Modell und einem spaltenorientierten Speicherformat, müssen bei einem Datenaustausch zwischen Operationen im Anfrageplan mehrere Spalten übergeben werden können, um alle Elemente eines Tupels zusammen betrachten zu können. Nur so ist es möglich die Ergebnis-Tupel iterativ auszugeben. Bei einem zeilenorientierten Speicherformat werden hingegen immer gesamte Tupel bzw. alle benötigten

Elemente eines Tupels zwischen Operationen ausgetauscht, sodass direkt alle Elemente für ein Ergebnis-Tupel zur Verfügung stehen. Dazu gibt es Operationen, wie z. B. die *Filter*-Operation (*Selektion*) für eine Spalte, die nach diesem Modell auf alle Spalten angewendet werden muss, aber die direkt nur von den Daten einer Spalte abhängig ist. Vereinfachen lässt sich dies, indem diese Operation auf die Spalte, die von dem Prädikat verwendet wird, angewendet wird und die Positionen aller Elemente, die das Prädikat erfüllen, zurückgibt. Eine Position wird im weiteren als `RID` bezeichnet. Mit der Liste der `RIDs` lässt sich dann das Ergebnis für die anderen Spalten einfach berechnen, indem die Elemente an den gegebenen Positionen zurückgegeben werden. Dies wird im weiteren als *Gather*-Operation bezeichnet.
Diese Liste von `RIDs` bezieht sich auf die gesamte Tabelle und ist damit für alle Spalten dieser Tabelle identisch. Somit lässt sich der Austausch von Zwischenergebnissen vereinfachen, indem für die Spalten einer Tabelle nur die Liste der `RIDs` und eine Referenz auf die Tabelle gespeichert werden. Mit diesen Informationen lassen sich dann die eigentlichen Tupel bzw. Elemente von diesen berechnen. Um diese berechneten Daten weitergeben zu können, muss auch die Möglichkeit geschaffen werden, diese in einem Zwischenergebnis speichern zu können. Gesondert behandelt müssen bei diesem Konzept berechnete Spalten. Damit diese mit dem Vorgehen vereinbar sind, müssen die berechneten Spalten in einer temporären Tabelle gespeichert werden, damit eine Referenz auf die eigentlichen Daten gespeichert werden kann. Diese kann dann von weiteren Operationen, analog zu nicht berechneten Spalten, verwendet werden.

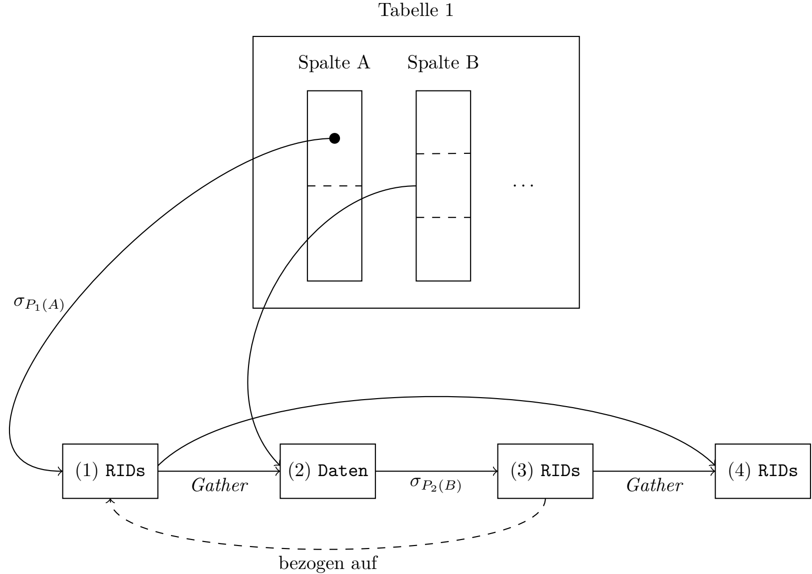


Abbildung 3.13: Beispiel für Zwischenergebnisse mit zwei *Filter*-Operationen.

Beispiel. Um die verschiedenen Szenarien, die die Klasse `Slice` zum Austausch von Zwischenergebnissen unterstützen muss, zu veranschaulichen, werden im Folgenden einige anhand des Beispiels in Abbildung 3.13 erklärt. In der Abbildung ist die Tabelle 1 mit den beiden Spalten A und B, auf die zugegriffen wird,

dargestellt. Es soll gezeigt werden, wie auf diese beiden Spalten erst die *Selektion* mit dem Prädikat P_1 und anschließend die *Selektion* mit dem Prädikat P_2 angewendet wird.
Nach der vektorisierten Verarbeitung wird als erstes ein Vektor aus der Spalte A extrahiert und auf diesen die *Filter*-Operation mit dem Prädikat P_1 ausgeführt. Um die Auswertung eines Prädikats einfach zu halten, werden o. B. d. A. nur Prädikate mit einer logischen Operation betrachtet. Da das Prädikat A in diesem Beispiel nur von der Spalte A abhängig ist, reicht als Eingabe der Vektor von Spalte A aus. Das Ergebnis dieser Operation ist der Vektor (1) mit der Liste von `RIDs` der Elemente, die das Prädikat P_1 erfüllen. Damit auf diesem Ergebnis die *Filter*-Operation mit dem Prädikat P_2, das von der Spalte B abhängig ist, ausgeführt werden kann, werden die Elemente der `RIDs` aus der Spalte B benötigt. Dafür wird die *Gather*-Operation auf die Liste der `RIDs` ausgeführt, wobei mithilfe des *Offsets* die entsprechenden Elemente aus der Spalte B geholt werden.
Auf diesen Elementen, veranschaulicht in Vektor (2), kann dann die *Filter*-Operation ausgeführt werden. Dabei ist zu beachten, dass die resultierenden `RIDs` in Vektor (3) sich nicht auf die eigentliche Tabelle beziehen, sondern auf die Daten in Vektor (2) und somit auf die Liste der `RIDs` in Vektor (1). Um die `RIDs` zu berechnen, die wieder bezogen sind auf die eigentliche Tabelle, lässt sich mit diesen auf der Liste der `RIDs` aus Vektor (1) die *Gather*-Operation ausführen. Damit beziehen sich die `RIDs` in Vektor (4) auf die eigentliche Tabelle und es lassen sich mit diesen die Elemente des Endergebnisses berechnen (nicht in der Abbildung dargestellt).
Um die Tupel für das Endergebnis zu berechnen, kann nun wieder für beide Spalten der Tabelle jeweils die *Gather*-Operation mit der Liste der `RIDs` in Vektor (4) auf der jeweiligen Spalte ausführen werden. Dieses Vorgehen wird für alle Vektoren, die aus der Spalte A extrahiert werden können, wiederholt.

Verallgemeinerung. Um alle Möglichkeiten von Zwischenergebnissen zu unterstützen, enthält ein Objekt der Klasse `Slice` für jede Tabelle bzw. berechnete Spalte, die in dem Zwischenergebnis vorkommt, folgende Attribute:

- `table`: Enthält einen Zeiger auf die Tabelle,
- `column`: Enthält einen Zeiger auf eine `BaseColumn`,
- `filter`: Enthält einen Zeiger auf eine `BaseColumn` mit `RIDs`,
- `name`: Enthält eine Zeichenkette zur Identifikation,
- `is_data`: Enthält ein *Flag*, ob das Attribute `column` Daten enthält oder nicht (`RIDs`).

Mit diesen Attributen lassen sich verschiedene Zwischenergebnisse repräsentieren. Die Bedeutung der Attribute ist abhängig von dem Wert des Attributs `is_data`. Da dieses eine binäre Variable ist, lassen sich für die restlichen Attribute jeweils zwei Bedeutungen unterschieden.

`is_data` = `false`: In diesem Fall enthält das Zwischenergebnis für diese Tabelle im Attribut `column` einen Zeiger auf eine `BaseColumn` mit `RIDs`, die angeben welche Tupel dieser Tabelle in dem Zwischenergebnis enthalten

sind. Wenn das Attribut `filter` nicht leer ist, beziehen sich die `RIDs` auf die Position in der `BaseColumn` von dem Zeiger aus dem Attribut `filter`. Um die eigentlichen Elemente der Tupel berechnen zu können, müssen die `RIDs` mit der *Gather*-Operation umgerechnet werden. In dem Beispiel aus Abbildung 3.13 trifft dieser Fall auf die Zwischenergebnisse der *Selektionen*, Vektor (1), (3) und (4), zu.

`is_data` = `true`: In diesem Fall enthält das Zwischenergebnis für diese Tabelle im Attribut `column` einen Zeiger auf eine `BaseColumn` mit Elementen einer Spalte der Tupel dieser Tabelle. Wenn das Attribut `filter` nicht leer ist, beziehen sich die Positionen der Elemente nicht auf die eigentliche Tabelle, sondern auf eine Teilmenge der Elemente, die durch die `Base Column` von dem Zeiger aus dem Attribut `filter` gegeben ist. In dem Beispiel aus Abbildung 3.13 trifft dieser Fall auf das Zwischenergebnis der *Gather*-Operation nach der ersten *Selektionen*, Vektor (2), zu.

Um Ergebnis-Tupel berechnen zu können, die Elemente aus Tupeln verschiedener Tabellen enthalten, werden die fünf beschriebenen Attribute für jede dieser Tabellen gespeichert. So lässt sich das Ergebnis eines *Joins* über zwei Tabellen mit zwei Listen von `RIDs` repräsentieren, die die entsprechenden Tupel aus der jeweiligen eigentlichen Tabelle bezeichnen. Mit diesen beiden gleichlangen Listen lassen sich dann die Ergebnis-Tupel konstruieren. Dafür wird für jede Position in diesen Listen, das Tupel aus der ersten Tabelle, bezeichnet durch die `RID` an der Position in der ersten Liste, und das Tupel aus der zweiten Tabelle, bezeichnet durch die `RID` an der Position in der zweiten Liste, konkateniert. Für weitere Operationen auf diesem Ergebnis müssen dann weiterhin diese Informationen für beide Tabellen betrachtet werden. Damit z. B. eine *Selektion* auf diesem Ergebnis ausgeführt werden kann, müssen erst die dafür benötigten Daten ermittelt werden. Diese Daten müssen dann zweimal in dem Zwischenergebnis gespeichert werden, damit die beiden Listen von `RIDs` jeweils als Bezug erhalten bleiben. Analog muss auch für berechnete bzw. sortierte Spalten verfahren werden. Wie bereits beschrieben, werden diese in temporären Tabellen gespeichert, so dass diese wie aus einer weiteren Tabelle behandelt werden können.
In der Version von SliceDB nach zwei Semestern werden diese Möglichkeiten jedoch nicht vollständig genutzt. Um die Verarbeitung der Zwischenergebnisse zu vereinfachen, wir das `filter` Attribut nicht verwendet. Das bedeutet, dass für jede Tabelle bzw. berechnete Spalte entweder eine `BaseColumn` mit Daten oder mit `RIDs` im `column` Attribut gespeichert wird. Für den Fall, dass diese `BaseColumn` `RIDs` enthält, wird dann vor der Weiterverarbeitung immer die *Gather*-Operation ausgeführt, um die Elemente des berechneten Zwischenergebnisses zu erhalten. Dadurch muss bei der Weiterverarbeitung das `filter` Attribut nicht beachtet werden.

3.2.9 Zusammenfassung *(Jan Stallmann)*

In diesem Abschnitt wurde der *Storage-Layer* von SliceDB vorgestellt und die grundlegenden Entscheidungen bei der Entwicklung aufgezeigt. Es wurde gezeigt wie eine Datenbank mit ihren Tabellen im Hauptspeicher und auf einer Festplatte gespeichert wird. Eine gesonderte Rolle spielen dabei die Spalten, in denen die eigentlichen Daten der Tabellen gespeichert werden. Für diese

wurde eine erweiterte Verwaltung vorgestellt, mit der der Hauptspeicher effizient genutzt und dafür gesorgt wird, dass häufig benötigte Daten im Hauptspeicher vorhanden sind. Des Weiteren wurde ein Modell vorgestellt, mit dem der Datentyp einer Spalte für die weitere Verarbeitung verdeckt wird und die Aufrufe der Operatoren an die entsprechenden Funktionen delegiert werden. Zum Schluss wurde noch eine Datenstruktur für das Volcano-Iterator Modell vorgestellt, mit der Zwischenergebnisse ausgetauscht werden können, die auf der verwendeten Spalte-Repräsentation basieren.

3.3 Operatoren *(Andreas Blume)*

Operatoren sind ein wichtiger Grundbaustein von allen Datenbankmanagementsystemen und deshalb auch ein fester Bestandteil von SliceDB. Hier werden sie von der Ausführungseinheit von SliceDB aufgerufen und arbeiten auf einzelnen Teilen einer Spalte (= Vektoren) oder auf ganzen Spalten einer Datenbank-Tabelle. Die unterschiedlichen Eingaben sind wichtig, da es in SliceDB blockierende und nicht blockierende Operatoren gibt. Zu den blockierenden Operatoren zählen u.a. Join, Sortieren und Gruppieren, welche jeweils eine vollständige Spalte als Input benötigen. Die nicht blockierenden Operatoren (z.B. Selektion, Gather und der Operator für arithmetische Ausdrücke) können hingegen mit einzelnen Vektoren arbeiten, ohne dass die Korrektheit beeinträchtigt wird.

Die folgenden Abschnitte erläutern die Funktionsweise und die Integration der einzelnen Operatoren in SliceDB.

3.3.1 Selektion *(Andreas Blume)*

In diesem Abschnitt wird zunächst die Selektion allgemein vorgestellt, um danach die Umsetzung in SliceDB zu präsentieren. Abschließend wird ein kurzes Fazit gezogen und einige Performanceverbesserungen vorgestellt.

Allgemeines

Die Selektion ist neben dem Join die wichtigste Operation eines DBMS. Der Operator wird verwendet, um eine Datenbank-Tabelle nach relevanten Tupeln zu filtern. Eine einfache Selektion besteht aus einer oder mehreren Selektionsbedingungen, die im Where-Statement einer SQL-Abfrage angegeben werden können:

SELECT
    spalte1, spalte2, ...
FROM
    tabelle
WHERE
    <selektionsbedingung/en>

Eine einzelne Bedingung kann u.a. ein Größenvergleich zweier Werte, eine Bereichs- (Between ... And ...) oder Ähnlichkeitsabfrage (Like), oder ein Vergleich mit einer Liste (In) sein (vgl. [20]). Beim Größenvergleich wird eine Spalte

mit Hilfe von Vergleichsoperatoren („=“, „<“, „>“, „<=“, „>=“, „<>“ (in einigen System auch „!=“)) mit einem Wert verglichen. Dabei müssen sowohl der Datentyp der Spalte als auch der Wert vergleichbar sein. Dies ist für Zahlen, Zeichenketten und Datumsangaben der Fall. Bei einer Bereichsabfrage (Spalte Between X and Y) werden alle Tupel zurückgeliefert, die zwischen den Bereichsgrenzen X und Y liegen. Abhängig vom DBMS gehören die Grenzwerte (im Beispiel: X und Y) zum Bereich oder nicht. Eine Bereichsabfrage ist wie der Größenvergleich für alle vergleichbaren Datentypen durchführbar. Die Ähnlichkeitsabfrage (Spalte Like ...) entspricht der „=“- (Like) bzw. „!=“-Operation (Not Like) des Größenvergleichs, der hier für den Vergleich von Zeichenketten mit Wildcards genutzt wird. Die Wildcards (der Unterstrich „_“ für ein beliebiges einzelnes Zeichen und das Prozentzeichen „%“ für eine beliebige Zeichenkette mit 0 oder mehr Zeichen) werden beispielsweise verwendet, um eine Spalte mit einem nicht genau bekannten Suchenbegriffe zu filtern. Der Vergleich mit einer Liste (Spalte In (a,b,...)) wird verwendet, um den Inhalt einer Spalte mit einer angegeben Liste, die einen beliebigen Datentyp hat, zu vergleichen.

Um die einzelnen Selektionsbedingungen verknüpfen zu können, werden logische Operationen benötigt. Neben dem *NOT* (als Negation), das oben bereits beim Like-Operator verwendet wurde, stellt SQL das *AND* (als Konjunktion) und das *OR* (als Adjunktion) zur Verfügung. Eine SQL-Abfrage mit mehreren Selektionsbedingungen kann also folgendermaßen angeben werden:

SELECT
  spalte1, spalte2, ...
FROM
  tabelle
WHERE
  (not) selektionsbedingungen1 and/or (not) selektionsbedingungen2 and/or ...

Umsetzung in SliceDB

Der Selektionsoperator wurde als erstes in SliceDB integriert, da an ihm das Zusammenspiel zwischen der Datenhaltung (Storage-Layer), der Generierung des Anfrageplans und der Operatoren getestet wurde. Deshalb wurde die Selektion zunächst sehr einfach umgesetzt und bietet so noch einige Möglichkeiten für Performanceverbesserungen.

Umgesetzte Selektionsbedingungen Aktuell unterstützt SliceDB alle Formen des Größenvergleichs und die Ausführung von Bereichsabfragen. Beim Größenvergleich müssen Spalte und Wert (= Konstante) den gleichen Datentyp besitzen. Ähnlich sieht es bei den Bereichsabfragen aus, hier müssen die Bereichsgrenzen und der Spalten-Typ übereinstimmen.

Ein- und Ausgabe Die Selektion wurde als separate Klasse mit dem Namen *Selection* implementiert. Sie enthält für jede Selektionsart eine einzelne, private Methode. Jede Methode benötigt folgende Eingabewerte:

- *T* column*
  Ein Array vom Typ T, dass die Werte des aktuellen Vektors enthält.
- *Storage::Size& num_elements*
  Die Länge des Vektors.
- Vergleichswerte:
  - Größenvergleich: *T& value*
    Ein Wert vom Typ T.
  - Bereichsabfrage: *T& value1* und *T& value2*
    Die beiden Bereichsgrenzen vom Typ T. Dabei entspricht *value1* der unteren und *value2* der oberen Grenze.
- *Storage::RID* result*
  Ein Array vom Typ RID (= Record-ID) in das später das Ergebnis eingetragen wird. Die Länge ist mit der *T* column* identisch, da maximal alle Tupel zum Ergebnis gehören können.

Nachdem eine Selektionsbedingung von der jeweiligen Methode ausgewertet wurde, werden folge Ausgaben zurückgeliefert:

- *Storage::RID* result*
  Das gefüllte Ergebnisarray vom Typ RID, dass die Record-ID aller Ergebnis-Tupel enthält.
- *Storage::Size& result_elements*
  Die Länge des Ergebnis.

Implementierung Jede Methode besteht aus einer einfachen for-Schleife, die über alle Werte eines Vektors iteriert und jeweils die Selektionsbedingung testet. Ist die Bedingung erfüllt, so wird die jeweilige Record-ID zum Ergebnis hinzugefügt. Die Implementierung der „=“-Bedingung sieht also wie folgt aus:

```
1: template <typename T>
2: Storage::Size select_equal(...) {
3:     Storage::Size resultsize = 0;
4:     for (Storage::Size i = 0; i < num_elements; i++) {
5:         if (column[i] == value)
6:             result[resultsize++] = i;
7:     }
8:     return resultsize;
9: }
```

Listing 3.2: Der *select_equal*-Algorithmus, der einen Vektor mit Hilfe der Selektionsbedinung „=“ filtert.

Template Um das Schreiben von redundantem Code zu vermeiden, wurde die *Selection*-Klasse mit Hilfe des Template-Mechanismus von C++ umgesetzt. Das Template unterstützt alle Datentypen, die der Storage-Layer erlaubt (vgl. Abschnitt 3.2.1 Datentypen). Dazu gehören laut aktuellem Stand: *Storage::Bool*, *Storage::Integer*, *Storage::Float* und *Storage::String*.

Einbettung der Selektion in SliceDB Damit die einzelnen Selektionsbedingungen von SliceDB genutzt werden können, stellt die Klasse *Selection* die public Methode *select(...)* zur Verfügung. Diese Methode bekommt neben den eigentlich Eingabewerten (siehe oben) zusätzlich die geforderte Selektionsbedingung übergeben und ruft anschließend die jeweilige private Methode auf. Eine Ausnahme bilden *select_between(...)* und *select_not_between(...)*, die direkt als public Methoden zur Verfügung stehen.

Die drei Methoden (select(...), select_between(...) und select_not_between(...)) werden nun im *Executer*, sowie in den Klassen *BaseColumn*, *PageColumn*, *PlainColumn* und *TypedColumn* eingebettet. Dadurch kann jede *Column* nach bestimmten Werten gefiltert werden.

Test Nach der Fertigstellung einer einfachen Implementierung der Selektion wurden Tests geschrieben, in denen die korrekte Ausführung des Operators überprüft wurde. Die Test verifizieren die Korrektheit von jeder Selektionsbedingung für Datentypen *Storage::Integer* und *Storage::Float*.

Die Tests werden später genutzt, um nach Performanceverbesserungen das korrekte Verhalten der Selektionsoperation zu validieren.

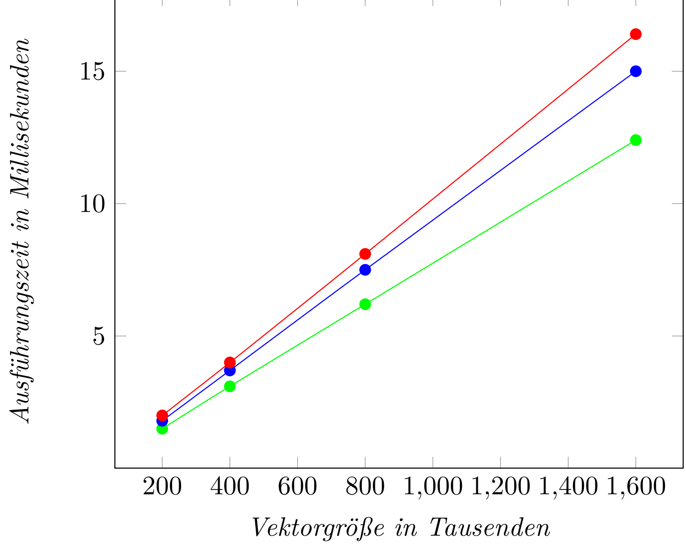


Abbildung 3.14: Laufzeit des *select_equal*-Algorithmus (ohne Erstellung der Result-Column) bei unterschiedlichen Selektionsgraden. Die grüne Kurve entspricht 10%, die Blaue 50% und die Rote 90%.

Performance Um spätere Laufzeitverbesserung der Selektionsoperation in Zahlen ausdrücken zu können, wurde die aktuelle Geschwindigkeit gemessen. Dazu wurden Vektoren (unterschiedlicher Länge) mit zufälligen Werten generiert, die einen Selektionsgrad von 10%, 50% und 90% besitzen. Die Ausführungszeit

wurde dann mit Hilfe von *boost::chrono* bestimmt.

In Abbildung 3.14 ist die Ausführungszeit des *select_equal*-Algorithmus für Vektoren vom Typ *Storage::Integer*, die aus 200-, 400-, 800- und 1600-Tausend Werten bestehen. Die grüne Kurve entspricht einem Selektionsgrad von 10%, die Blaue von 50% und die Rote von 90%. Bei allen drei Kurve zeigt sich, dass die Laufzeit linear mit der Spaltengröße wächst. Außerdem tritt der vermutete Effekt ein, dass der *select_equal*-Algorithmus bei einem Selektionsgrad von 10% deutlich schneller ist als bei 90%. Dies ist auf das Phänomen Branch Prediction, das durch die if-Anweisung entsteht, zurückzuführen.

Fazit und Ausblick

Wie bereits zuvor erwähnt, wurde die Selektionsoperation zunächst sehr einfach umgesetzt und enthält bisher nicht alle möglichen Selektionsbedingungen. Deshalb sollten zunächst die fehlenden Bedingungen (z.B. IN) implementiert werden, damit die volle Funktionalität des DBMS gewährleistet ist. Darüber hinaus sollten die bereits existierenden Methoden so aktualisiert werden, dass eine Selektionsbedingung auch mit unterschiedlichen Datentypen (z.B. zwei Spalten mit unterschiedlichem Datentypen vergleichen) arbeiten kann.

Ein weiterer großer Punkt sind Performanceverbesserungen. Deshalb werden im Folgenden einige Ideen vorgestellt.

Branch Prediction Eine einfache Methode, die die Ausführungszeit verbessern kann, ist Branch Prediction. Dabei wird die if-Anweisung in der for-Schleife durch einen Ausdruck ersetzt, der in jeder Schleifeniteration ausgeführt wird. Dadurch wird eine Ausführungszeit erreicht, die unabhängig vom Selektionsgrad ist.
Für die „=“-Bedingung (vgl. jetzige Implementation in Listing 3.2) würde eine Implementierung ohne Branch (eine Ausnahme ist die for-Schleife) wie folgt aussehen:

```
1: for (Storage::Size i = 0; i < num_elements; i++) {
2:     result[resultsize] = i;
3:     resutlsize += (column[i] == value);
4: }
```

Listing 3.3: Der *select_equal*-Algorithmus mit Branch Prediction.

Hier werden spätere Tests zeigen müssen, in wie weit Branch Prediction die Ausführungszeit der Selektionsoperation beschleunigen kann. Dazu muss genau geprüft werden bis zu welchem Selektionsgrad eine branchlose Implementierung schneller ist bzw. ab wann sie langsam wird.

Loop Unrolling Auch Loop Unrolling kann zu einer geringeren Ausführungszeit führen. Wie der Name schon verrät, wird hier die for-Schleife ausgerollt und so die Anzahl der Schleifenkopf-Ausführungen gesenkt. Eine Anwendung

auf die "="-Bedingung (vgl. jetzige Implementation in Listing 3.2) könnte also beispielsweise für eine gerade *Column*-Größe folgendermaßen implementiert werden:

```
1: for (Storage::Size i = 0; i < num_elements; i+=2) {
2:     if (column[i] == value) {
3:         result[resultsize++] = i;
4:     }
5:     if (column[i+1] == value) {
6:         result[resultsize++] = i+1;
7:     }
8: }
```

Listing 3.4: Der *select_equal*-Algorithmus mit Loop unrolling.

Im Listing 3.4 sehen wir, dass nun pro Schleifeniteration zwei Column-Einträge getestet werden. Dadurch verringert sich also die Anzahl der Schleifenkopf-Ausführungen um den Faktor 2. Natürlich können auch noch weitere Schleifeniterationen, z.B. k, ausgerollt werden, wobei das Beste k mit Hilfe von Tests bestimmt werden muss.

Parallelität Viele Algorithmen profitieren von einer parallelen Implementierung und werden dadurch deutlich schneller. Ähnlich sieht es bei der Selektion aus, bei der sich die for-Schleife beispielsweise durch parallele Threads beschleunigen lässt. Dabei ist sowohl eine parallele Implementierung für Multicore CPUs als auch für GPUs vorstellbar.

Die Parallelisierung kann dabei in mehreren Schritte erfolgen, wobei nach jedem Schritt mit Hilfe von Tests geprüft wird, um wie viel Prozent die Ausführung schneller wird und in welchen Szenarien die Verwendung der CPU oder der GPU von Vorteil ist. Neben den Laufzeitmessungen sollte regelmäßig die Korrektheit der parallelen Selektion validiert werden, da nur so die richtige Ausführung der Selektion garantiert werden kann.

3.3.2 Gather *(Tristan Schäfer)*

Der Gather Operator beinhaltet eine Methode, welche ein Datenarray in Verbindung mit Record IDs (RIDs) entgegennimmt und anhand dessen ein Ergebnisarray mit Werten füllt. Auf diese Weise werden die durch die RID-Liste adressierten Werte aufgelöst, so dass das Ergebnis für weitere Berechnungen genutzt werden kann.

Ein Gather ist jeweils zwischen dem Aufruf verschiedener Operatoren erforderlich. Bei mehreren Selektionen nacheinander werden alle Tupel gelesen und ein Zwischenergebnis materialisiert, was für diesen Anwendungsfall unnötigen Overhead bedeutet. Nach Rücksprache mit der Projektleitung ist der Einfluss auf die Performance durch dieses Vorgehen jedoch zum jetzigen Zeitpunkt vernachlässigbar.

Die komplette Funktion wird in Listing 3.5 gezeigt. Um alle verwendeten Datentypen zu unterstützen, werden Templates genutzt. In einer ersten Implementierungsphase wurde die Arraygröße als gegeben vorausgesetzt, so dass innerhalb der Funktion keine Allokation von Speicher erfolgt. Die Methode wird für eine Column aus dem Executor heraus aufgerufen, welcher aus diesen Werten einen BaseColumnPtr generiert und zurückgibt.

Ein Optimierungspotential für die Gather Funktion existiert nur begrenzt. Es könnte Random Access vermeiden werden, indem die Spalten sortiert zur Verfügung gestellt werden. Ebenso könnte Loop-Unrolling in Betracht gezogen werden, um die Performance zu verbessern.

```
1: template <typename T>
2: void Gather<T>::gatherValues(const T* column,
3:     const size_t num_elements,
4:     const Storage::RID* positions,
5:     const size_t num_positions, T* result) {
6:   for (size_t i = 0; i < num_positions; i++) {
7:     result[i] = column[positions[i]];
8:   }
9: }
```

Listing 3.5: Gather Funktion

3.3.3 Nested-Loop Join *(Tristan Schäfer)*

Ein Join fügt zwei verschiedene Entitäten oder Columns zusammen. Listing 3.6 enthält eine SQL Statement, welches einen Equijoin dargestellt. Bei einem derartigen Join müssen die Attribute R.A und S.E gleich sein, um die Joinbedingung zu erfüllen.

```
1: SELECT *
2: FROM R, S
3: WHERE R.A = S.E
```

Listing 3.6: SQL Query Join

Für die Anwendung wurden weitere Join Arten, wie beispielsweise Semijoins oder Not Joins, nicht beachtet, da diese für die Queries des Star Schema Benchmarks keine Relevanz haben. Für die Durchführung eines Equijoins existieren verschiedene Join-Algorithmen, welche sich beispielsweise in Rechenzeit, Speicherbedarf oder Anzahl der I/O Operationen unterscheiden. Zunächst wurde mit dem Nested Loop Join Algorithmus eine Herangehensweise gewählt, welche gleichermaßen intuitiv und einfach zu implementieren ist. Dabei wird über verschachtelte Schleifen jedes Element aus R mit jedem Element aus S verglichen. Sollten die Elemente gleich sein, wird das jeweilige Paar in das Ergebnis übernommen. Listing 3.7 zeigt einen entsprechenden Pseudo Code.

```
1: foreach r in R do
2:   foreach s in S do
3:     if r.a = s.a do
4:       Fuege Paar r,s zur Ergebniscolumn hinzu
5:     endif
6:   end foreach
7: end foreach
```

Listing 3.7: Nested Loop Join Pseudocode

Performance

Weil alle Elemente jeweils mit der kompletten gegenüberstehenden Relation verglichen werden, entspricht die Laufzeit dem kartesischen Produkt mit O(N*M). Durch die großen Datenmengen können zudem Thrashing Effekte eintreten (vgl. [30]). Hierbei werden die ersten hinterlegten Datenelemente aus dem Cache oder Hauptspeicher von nachfolgenden Elementen der gleichen Relation verdrängt, weil nicht ausreichend Platz für die kompletten Daten existiert. Thrashing hat nicht nur Einfluss auf die Cache-Effizienz, sondern kann zu vermehrten Festplattenzugriffen führen. Dies tritt insbesondere dann auf, wenn die Daten größer sind als der zur Verfügung stehende Hauptspeicher. Aus diesem Grund zeigt Nested Loop Join zwar Nachteile für große Datenmengen, ist jedoch besonders gut für kleine Tabellen geeignet, welche komplett im Cache gehalten werden können.

Anwendungsbeispiele

Aufgrund der genannten Besonderheiten ergeben sich Auswirkungen auf die Nutzbarkeit des Algorithmus. Er eignet nicht ausschließlich für kleine Tabellen, sondern auch für große Tabellen, bei denen zunächst nur die ersten Ergebnisse genutzt werden. Als Beispiel für eine solche Verarbeitung ist die Anfrageauswertung in einem Database Management Tool zu nennen. Im Oracle SQL Developer wird dem Nutzer auch bei großen Queries zunächst nur eine konfigurierbare Anzahl an Tupeln angezeigt (z.B. erste 1000 Ergebnisse). Erst bei entsprechender Nutzereingabe wird die Ergebnismenge inkrementell vervollständigt. Als weiterer Anwendungsfall wäre ein Extrembeispiel mit 2 Tabellen zu nennen, bei der die Join Menge annähernd dem kartesischen Produkts entspräche (z.B. 2 Boolean Tabellen, welche komplett mit true gefüllt sind). In diesem Fall würde ein Nested Loop Join eine bessere Performance als der Hash-Join zeigen. Für reelle Daten ist ein Auftreten dieses Anwendungsfalls allerdings sehr unwahrscheinlich. Weitere Vorteile des Algorithmus liegen in der einfachen Implementierung, Korrektheit und der Robustheit. Aus diesem Grund kann der Nested Loop Join als Referenzimplementierung für vergleichende Unit-Tests verwendet werden (siehe Abschnitt Hash-Join).

Optimierungspotential

Weitere Varianten des Nested Loop Joins sind der Block Nested Loop Join und der Index Nested Loop Join. Während eine Umsetzung des Index Nested Loop Join in der aktuellen Version aufgrund fehlender Indizexstrukturen nicht

in Frage kommt, könnte ein Block Nested Loop Join prinzipiell durch durch eine Anpassung der Volcano Iterator Implementierung erreicht werden. Ein Vektor könnte dabei jeweils einen Block darstellen. Diese Implementierung wird voraussichtlich eine schlechtere Performance als der Hashjoin zeigen, daher wird dieser Join Algorithmus nicht weiter verfolgt.

Implementierung

Die Methode nl_join ist in der Klasse Operators::Join als statische Methode hinterlegt und erwartet als Parameter zwei Pointer auf die Startpositionen der Plain Arrays und die jeweilige Arraygröße. Als Output generiert der Algorithmus ein Paar aus BaseColumnPointern, welche die entsprechenden RID Lists enthalten. Der Prototyp der Funktion ist in Listing Listing 3.8 dargestellt.

```
1: static std::pair<Storage::BaseColumnPtr,
2:   Storage::BaseColumnPtr> nl_join(
3:   const T* array1, const Storage::Size &size1,
4:   const T* array2, const Storage::Size &size2);
```

Listing 3.8: Funktions-Prototyp Nested Loop Join

Wie bei übrigen Operatoren wird die Polymorphie durch die Verwendung von Templates gewährleistet. Es wird vorausgesetzt, dass die Arrays den gleichen Datentyp aufweisen. Durch die Verwendung einer weiteren Templatevariable wäre es zwar möglich, auch Columns unterschiedlicher Typen zu joinen (beispielsweise int mit double). Hier müsste aber ggf. beachtet werden, dass der Compare Operator der Datentypen entsprechend angepasst wird. Eine Anforderung zum Join verschiedenartiger Columns entsteht durch die TCP-H Queries nicht, insofern wurde von einer solchen Implementierung abgesehen. Um zwei Columns zu joinen, ist eine entsprechende Methode der Column aufrufbar, welche eine Referenz auf eine weitere Column als Parameter erwartet. Der Executor einer Spalte agiert als Wrapper zum primitiven Operator und stellt die benötigten Daten in der korrekten Form zur Verfügung. Listing Listing 3.9 zeigt den entsprechenden Code.

```
1: template<typename T>
2: std::pair<BaseColumnPtr, BaseColumnPtr>
3:   Executor<T>::nl_join(const PlainColumn<T>* column1,
4:   const PlainColumn<T>* column2) {
5:   return Operators::Join<T>::nl_join(
6:     column1->data(), column1->size(),
7:     column2->data(), column2->size());
8: }
```

Listing 3.9: Executor Anbindung Nested Loop Join

3.3.4 Hash-Join *(Tristan Schäfer)*

Neben dem Nested Loop Join wurde ein hashbasierter Join Algorithmus gewählt. Ein Hashjoin setzt sich aus zwei Phasen zusammen. Beim Aufbau der Hashtabelle in der Buildphase werden alle Elemente der Relation R mit Hilfe einer Hashfunktion in einem entsprechenden Bucket abgelegt. In der anschließenden Probephase wird unter Verwendung der gleichen Hashfunktion geprüft, ob ein Bucket für den zu prüfenden Key aus S existiert. Falls ein solcher Bucket vorhanden ist, werden alle hier enthaltenen Werte aus R mit dem aktuellen Element aus S dem Ergebnis hinzugefügt. In Listing 3.10 ist ein entsprechender Pheudocode dargestellt.

```
 1: Für alle Elemente s aus S wiederhole
 2:   Hashe über das Join Attribut s.key;
 3:   Füge RID mit entsprechendem Key Hashtabelle ein
 4: end
 5: Für alle Elemente r aus R wiederhole
 6:   Hashe über den Join Attributen r.key;
 7:   if hash(r) verweist nicht-leeres Bucket
 8:     if r = s
 9:       Füge Paar r,s zur Ergebniscolumn hinzu
10:     end
11:   end
12: end
```

Listing 3.10: Hash Join Pseudocode

Performance

Unter Annahme gleich großer Relationen R,S ist die Laufzeit des Hash-Joins mit O(N) niedriger, als die des Nested Loop Joins [70]. Darüber hinaus gibt es verschiedene Faktoren, welche einen Einfluss auf die Performance des Algorithmus haben. Unter anderem spielt die Vektorgröße eine Rolle und insbesondere, ob diese sich komplett im Cache bzw. Hauptspeicher ablegen lassen. Hier ist die spaltenorientierte Speicherung der Daten von Vorteil. Diese hat zur Folge, dass nicht komplette Tupel, sondern immer nur einzelne Columns verbunden werden, was wiederum zu einer geringeren Bandbreitennutzung und höherer Cache Effizienz führt.

Bei Hashtabellen spielt zudem insbesondere der Translation Lookaside Buffer (TLB) eine große Rolle. Ein Bucket wird als eigene Page hinterlegt, so dass für einen Datenzugriff zunächst eine Übersetzung einer logischen in eine physische Adresse erfolgen muss. Für diese Aufgabe stehen dem TLB je nach Prozessorarchitektur eine begrenzte Anzahl von TLB-Einträgen zur Verfügung. Falls die Anzahl der Hashbuckets größer als die Anzahl der TLB-Einträge ist, entstehen Trashing Effekte, welche wiederum TLB-Misses zur Folge haben. Im Falle eines TLB-Misses werden Pagewalks notwendig. Zur Ermittlung der physischen Adresse sind dann mehrere Zugriffe auf den Hauptspeicher notwendig, was die Laufzeit stark beeinträchtigt.

Implementierung

Die als statische Funktion hinterlegte Hash-Join Implementierung nimmt zwei Arrays entgegen und liefert als Rückgabewert ein Paar der gefundenen RIDs. Sowohl der Funktionsaufruf als auch die Anbindung zum Storage Layer erfolgt exakt wie beim Nested Loop Join (siehe Abschnitt 3.3.3).

Als Hashtable Implementierung wurde eine Unordered Multimap der Boost Libraries gewählt. Im Gegensatz zur Unordered Map lassen sich hier mehrere Werte durch einen Key identifizieren. Somit kann die Datenstruktur genutzt werden, um Werte zu bearbeiten, die mehrmals in einer Relation enthalten sind. In der Probe-Phase werden über die Funktion equal_range(key) der Unordered Multimap Iteratoren ermittelt, welche auf das erste und letzte Element im Bucket verweisen. In einer for-Schleife werden die Iteratoren anschließend genutzt, um alle hinterlegten RIDs zusammen mit der RID des zu prüfenden Keys dem Ergebnis hinzuzufügen.

In der ersten Implementierung wurden in einer statischen Methode sowohl der Build- als auch Probevorgang durchgeführt. Dies hat zur Folge, dass die Build-Phase jeweils für jeden Vektor vollzogen wird und somit die Performance beeinträchtigt. In einer zweiten Variante wurden diese Arbeitsschritte getrennt. Dazu sind nicht nur Ergänzungen im primitiven Operator, sondern auch auf Ebene des Queryplans notwendig. Die open() Implementierung des Operators wird genutzt, um die Buildcolumn zunächst komplett einzulesen und anschließend eine Map durch Aufruf der build() Methode zu generieren. Eine Referenz der resultierenden Variable wird als Feldvariable im Queryplan Operator hinterlegt. Während des next() Aufrufs wird diese Referenz an die probe() Methode übergeben.

Tests, Typed Tests, Random Columns

Die grundlegende Idee zum Testen verschiedener Join-Algorithmen ist die Nutzung einer einfachen Join-Implementierung als Referenz, gegen die die Ergebnisse weiterer Join-Algorithmen geprüft werden können. Als simple und robuste Implementierung bietet sich der Nested Loop Join an, da bei dieser Variante Fehler schnell erkennbar sind und die Implementierung vergleichsweise übersichtlich ist. Um den Test automatisiert mit verschiedenen Datentypen durchzuführen, wurde ein Typed Test des Google Test Frameworks genutzt. Über die in Listing Listing 3.11 dargestellte Typedefinition lassen sich alle zu berücksichtigenden Typen festlegen.

```
1: typedef ::testing::Types<std::size_t, uint32_t, int32_t,
2:     double, bool, Storage::String> MyTypes;
3: TYPED_TEST_CASE(FooTest, MyTypes);
4:
5: TYPED_TEST(FooTest, join) {
6:   [...]
7: }
```

Listing 3.11: Definition der zu testenden Typen

Anhand dieser Typdefinition generiert das Framework zur Compile-Zeit verschiedene Tests, welche nacheinander durchlaufen werden. Aktuell werden die Joins für die von SliceDB unterstützen Datentypen size_t, uint32_t, int32_t, double, boolean und Storage::String getestet. Um Columns mit randomisierten Werten nutzen zu können, wurde im Util Namespace die Klasse Datageneration mittels vollständiger Templatespezialisierung dahingehend erweitert, dass auch für Strings und boolean Spalten sinnvolle Werte generiert werden. Die randomisierten R und S Relationen werden als Input für den Nested Loop Join und Hash-Join verwendet. Anschließend werden die Ergebnisse verglichen. Ein Test ist nur dann erfolgreich, wenn beide Join Algorithmen das gleiche Ergebnis liefern. Für die Vergleichbarkeit ist in Hinblick auf die Nutzung einer Hashtabelle eine Besonderheit zu beachten. Im Gegensatz zum Nested Loop Join spiegelt sich die ursprüngliche Reihenfolge der Werte nicht im Ergebnis wider. Entweder muss also die im Test verwendete Vergleichsfunktion angepasst werden, oder das Ergebnis zunächst sortiert und dann verglichen werden. Die Bedingungen für einen erfolgreichen Test lassen sich wie folgt zusammenfassen:

- Die Größen der ermittelten Ergebnisse stimmen überein
- Die Anzahl der Ergebnisse ist jeweils kleiner als das kartesische Produkt der Eingangsrelationen
- Die Inhalte der Ergebnisse stimmen komplett überein

Benchmark Ergebnisse

Die Abbildung 3.15 zeigt, dass die Laufzeit des überarbeiteten Hash Joins deutlich über den Zeiten der bestehenden Implementierung liegt. Bei näherer Betrachtung der Laufzeit einzelner Operationen fällt auf, dass insbesondere die Probephase einen Großteil der Berechnungszeit für sich beansprucht. Der Probe-Vorgang der beiden Implementierungen ähnelt sich stark und unterscheidet sich in erster Linie in der genutzten Datenstruktur. Während bisher der Template Datentyp als Key verwendet wurde, liegt nun durch die explizite Berechnung des Hashwertes ein Keys vom Typ uint64_t vor. Da die Eingabe- und Ergebnismenge bei beiden Vorgehen jeweils gleich sind, bleibt als mögliche Ursache für die Ergebnisse der Suchvorgang der Tabelle. Abweichungen in dieser Größenordnung lassen sich nicht auf die reine Berechnungszeit der alternativen Hashfunktion zurückführen. Die schlechte Skalierung deutet auf eine niedrige Look-up-Performance der Unordered Map hin, welche laut Library Referenz im Worst Case eine linear mit Größe des Containers wachsende Laufzeit aufweist. Unter Verwendung einer adäquaten Datenstruktur sollten sich bessere Ergebnisse erzielen lassen.

Ausblick

Die bisherige Implementierung weist Nachteile sowohl in Hinblick auf Parallelisierbarkeit als auch Cache Effizienz auf. Als Alternativen könnten ein Partitioned Hash Join (vgl. [58]) oder Radix Join (vgl. [16]) genutzt werden.

Beim Partitioned Hash Join werden die zwei Phasen Build und Probe um eine weitere Phase erweitert, welche vor der Build Phase stattfindet. In dieser Partition Phase werden die Relationen mit einer Hashfunktion in Partitionen

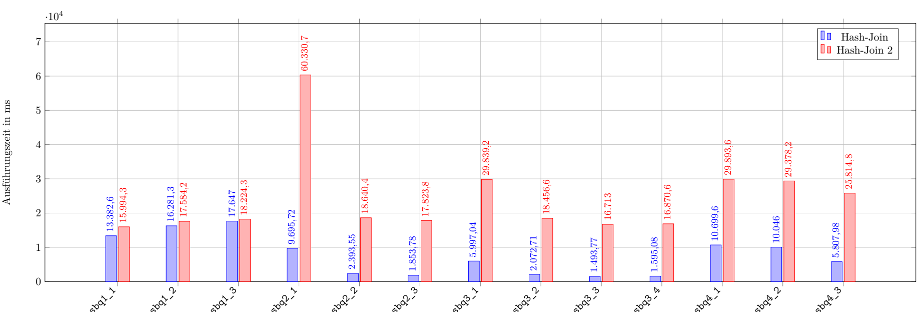


Abbildung 3.15: Gegenüberstellung der Hash-Join Implementierungen, Scalefactor 10

unterteilt, welche sich aufgrund ihrer geringen Größe in der folgenden Build Phase cache-effizient nutzen lassen. Ebenso lässt sich der Buildvorgang auf diese Weise einfach parallelisieren, indem die Partitionen unterschiedlichen Threads zugewiesen werden. Zwar ließe sich mit einem ähnlichen Vorgehen auch ein einfacher Hashjoin parallelisieren, allerdings müsste hier zur Vermeidung von Race Conditions der Zugriff auf die Hashtabelle synchronisiert stattfinden. Das entsprechende Locking führt wiederum zu Overhead.

Der Radix Join folgt dem gleichen Prinzip wie der Partitioned Hash Join. Allerdings wird hier die Partition Phase in verschiedene Schritte unterteilt, indem nur bestimmte Bits der zu hashenden Keys von der Hashfunktion berücksichtigt werden. Auf diese Weise lässt sich unter anderem die TLB Problematik beschränken, weil die Anzahl der benötigten TLB Einträge für jeden Partitionsschritt je nach Wahl der zu berücksichtigenden Bits gezielt steuern lässt.

Weiterführend ließe sich die Hashfunktion unter Einbeziehung von SIMD Operationen parallelisieren. Da die Bottlenecks allerdings im Datentransfer zwischen Cache und Hauptspeicher liegen und nicht in der eigentlichen Berechnung der Hashfunktion, ist zu prüfen, ob sich durch dieses Vorgehen große Vorteile ergeben.

3.3.5 Operator für arithmetische Ausdrücke *(Andreas Blume)*

In diesem Abschnitt wird zunächst vorgestellt, was arithmetische Ausdrücke sind und wie sie in SQL verwendet werden können. Danach wird eine einfache Implementierung vorgestellt und die Integration in *SliceDB* erläutert. Abschließend werden einige Performanceverbesserungen präsentiert und die Auswirkungen auf die Ausführungszeit des *Star Schema Benchmarks* [51] betrachtet.

Allgemeines

Arithmetische Ausdrücke sind ein fester Bestandteil von SQL. Sie folgen den Gesetzen der Mathematik und bestehen „aus einer Folge von Operanden [u.a. konstante Zahlen, Variablen oder ganze Spalten], die durch Operatoren und Klammern voneinander getrennt (oder miteinander verknüpft) sind“.[65] Angegeben werden sie im Select-Befehl einer SQL-Abfrage:

SELECT
    <arithmetischer Ausdruck1>, <arithmetischer Ausdruck2>, . . .
[FROM
    tabelle
WHERE
    . . .]

Wie in der Mathematik werden in SQL drei Gruppen von Operatoren (vgl. [10]) unterschieden. Zunächst gibt es die *unären Operatoren*, die direkt vor einem Ausdruck stehen. Zu nennen sind hier das Plus („+“) und das Minus („-“, auch Negation genannt), die das Vorzeichen eines Ausdrucks ändern. So wandelt das Plus den nachfolgenden Ausdruck in einen positiven Ausdruck um und das Minus in einen Negativen. Dazu kommt die bitweise Negation („∼“).

Die zweite Gruppe umfasst die *arithmetischen Operatoren*. Dazu gehören die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division sowie die Modulo-Operation. Diese Art von Operatoren steht immer zwischen zwei Ausdrücken.

Neben den arithmetischen Operatoren bietet SQL die Möglichkeit *bitweise Operatoren* zu verwenden. Hierzu zählen das bitweise Und („&“), das bitweise Oder („|“) und das bitweise Exklusiv-Oder („ˆ“). Bitweise Operatoren stehen immer zwischen zwei ganzen Zahlen (Integer) und erzeugen eine neue Ganzzahl.

Umsetzung ohne Optimierung in SliceDB

Da die Konzentration bei der Entwicklung von *SliceDB* zunächst auf der korrekten Ausführung aller Queries des *Star Schema Benchmarks* [51] lag, wurden alle *arithmetischen Operatoren* zunächst nur sehr einfach und ohne Optimierungen umgesetzt. Dabei können alle arithmetischen Operatoren zwei gleichlange Spalten oder eine Spalte und eine konstante Zahl verarbeiten.

Ein- und Ausgabe Der Operator für arithmetische Ausdrücke wurden als eine eigenständige Klasse mit dem Namen *ArithmeticOperation* implementiert. Die Klasse enthält für jede mögliche arithmetische und bitweise Operation sowie die Negation eine einzelne, private Methode. Jede Methode erwartet folgende Eingabewerte:

- *T* column*
  Ein Array vom Typ *T*, dass die Werte eines Vektors enthält.

- *Storage::Size& size*
  Die Länge des Vektors.

Dazu kommen bei den arithmetischen und bitweisen Operationen der folgende Input:

- Spalte <Operation> Spalte2: *T* column2*
  Ein Array vom Typ *T*, dass die Werte eines zweiten Vektors enthält. Das Array besitzt ebenfalls die Länge *size*.

- Spalte <Operation> konstante Zahl: *T& value*
  Eine konstante Zahle vom Typ *T*.

Nach der Ausführung der jeweiligen Methoden wird ein *BaseColumnPtr*, der die Länge *size* besitzt, zurückgeben.

Implementierung Um das Schreiben von redundanten Code zu vermeiden, wurde die Klasse *ArithmeticOperation* mit Hilfe des Template-Mechanismus von C++ umgesetzt und unterstützt so alle Datentypen, die laut Storage-Layer erlaubt sind (siehe Abschnitt 3.2.1 Datentypen). Die einzelnen Methoden bestehen aus einer einfachen for-Schleife in der die jeweilige Operation durchgeführt wird. Das Addieren einer konstanten Zahl *value* und einem Array *column* sieht also wie folgt aus:

```
1: template <typename T>
2: Storage::BaseColumnPtr addition_value(...) {
3:     ArrayColumn result mit Länge size erstellen
4:     for (Storage::Size i = 0; i < size; i++) {
5:         result->append(column[i] + value);
6:     }
7:     return result;
8: }
```

Listing 3.12: Die Addition von einer konstanten Zahl und einem Array.

Zunächst wird eine ArrayColumn (*result*) mit der Größe *size* erstellt, in der später das Ergebnis eingetragen wird. Danach startet die zuvor erwähnte for-Schleife, bei der in jeder Iteration die konstante Zahl *value* und der jeweilige *Column*-Eintrag addiert werden. Abschließend wird das Ergebnis zum *result* hinzugefügt. Ist das komplette Ergebnis berechnet, so wird das *result* in Form eines BaseColumnPtr's zurückgegeben.

Einbettung des Operators in SliceDB Damit die einzelnen Operationen von *SliceDB* genutzt werden können, bietet die Klasse *ArithmeticOperation* die public Methoden *column_sign_change(...)*, für die Negation, sowie *column_value(...)* und *column_column(...)*, für die arithmetischen und die bitweisen Operationen, als Schnittstelle nach außen an. Alle drei Methoden bekommen zunächst die bereits zuvor beschriebenen Eingabewerte. Darüber hinaus benötigen die Schnittstellen zu den arithmetischen und bitweisen Operationen die Angabe der auszuführenden Operation.

Um nun die einzelnen Operationen auf einer Column durchführen zu können, müssen die zuvor benannten Methoden (column_sign_change(...), column_value(...) und column_column(...)) im *Executer*, sowie in den Klassen *BaseColumn*, *PageColumn*, *PlainColumn* und *TypedColumn* eingebettet werden. Dies ermöglicht den Aufruf durch die Ausführungeinheit von *SliceDB*.

Test Nach der Implementation der Klasse *ArithmeticOperation* wurden mit Hilfe von *Google Test* [27] Tests geschrieben, um die Korrektheit der einzelnen

Methoden zu validieren. Die Tests beschränken sich auf die Datentypen *Storage::Integer* und *Storage::Float* (vgl. Abschnitt 3.2.1 Datentypen).

Die Tests werden später eine wichtige Rolle spielen, wenn nach Performanceverbesserung das korrekte Verhalten der Methoden geprüft werden soll.

Performanceverbesserungen

Nachdem der Opterator vollständig implemetiert und in *SliceDB* integriert war, konnte die Optimierung beginnen. Dabei lag die Konzentration zunächst auf einer sequentielle Optimierung mit Hilfe von **Loop Unrolling**, weil alle Operationen auf einer for-Schleife basieren. Beim Loop Unrolling wird die Schleifenbedingung seltener überprüft und ermöglicht so eine schnellere Ausführung der Schleife. Unterschieden werden das vollständige und teilweise Entrollen einer Schleife. Bei der Implementierung wurde das teilweise Entrollen (vgl. Listing 3.13) gewählt, da die Anzahl der Schleifendurchläufe vorher nicht bekannt ist und die Anzahl der Iterationen in den meisten Fällen zu hoch ist, um die Schleife vollständige zu entrollen.

```
1: // alte Implementierung:
2: for (Storage::Size i=0; i<N; i++) {
3:     result[i] = column[i] OPERATION value;
4: }
5:
6: // teilweises Entrollen mit Abrollfaktor k:
7: for (Storage::Size i=0; i<N; i+=k) {
8:     result[i]   = column[i]   OPERATION value;
9:     result[i+1] = column[i+1] OPERATION value;
10:    ...
11:    result[i+(k-2)] = column[i+(k-2)] OPERATION value;
12:    result[i+(k-1)] = column[i+(k-1)] OPERATION value;
13: }
```

Listing 3.13: Loop Unrolling Variante: teilweises Entrollen mit Abrollfaktor k.

Da nicht alle Arrays (*columns*) eine Länge besitzen, die sich durch einen konstanten Abrollfaktor k teilen lässt, musste anschließend eine Möglichkeit gefunden werden wie alle Arraylängen verarbeiten werden können. Dazu muss zunächst eine neue Obergrenze (entspricht der größten Zahl u, mit $0 <= u <= N$ und $u\%k = 0$) gefunden werden und anschließend geprüft werden, ob $u < N$ (N entspricht der Länge des *column*-Arrays) ist. Ist die Obergrenze u kleiner als N so muss für die noch fehlenden Arraywerte u, $u + 1$ bis $N - 1$ das Ergebnis berechnet und in das Array *result* geschrieben werden. Für die Berechnung der fehlen Werte wird eine weitere for-Schleife genutzt (vgl. Listing 3.14).

```
1: Storage::Size Obergrenze = (N / k) * k;
2: for(Storage::Size i=0; i<Obergrenze; i+=k) {
```

```
 3:     result[i]    = column[i]    OPERATION value;
 4:     result[i+1] = column[i+1] OPERATION value;
 5:     ...
 6:     result[i+(k-2)] = column[i+(k-2)] OPERATION value;
 7:     result[i+(k-1)] = column[i+(k-1)] OPERATION value;
 8: }
 9:
10: for(Storage::Size i=Obergrenze; i<N; i++)
11:     result[i] = column[i] OPERATION value;
```

Listing 3.14: Teilweises Entrollen mit Abrollfaktor k für alle *column*-Längen N.

Zu guter Letzt musste noch der beste Abrollfaktor k bestimmt werden. Um die Ausführungszeit aller Operationen für einige Abrollfaktoren (k = 2; 4; 6; 8 und 10) zu messen, wurde die Benchmark Bibliothek *Google Benchmark* [27] verwendet. Google Benchmark funktioniert ähnlich wie der bereits in *SliceDB* verwendete *Google Test* [26] (zur Prüfung der korrekten Ausführung eines Operators), misst aber die durchschnittliche Ausführungszeit einer Funktion. Dazu wird die Funktion, in unserem Fall der Operator, wiederholt von Google Benchmark ausgeführt und daraus die durchschnittliche Ausführungszeit über alle Wiederholungen bestimmt.

Der Benchmark wurde anschließend auf zwei unterschiedlichen Systemen getestet:

Prozessor	Cores	Taktrate	Cachegröße	DRAM
Intel Core i5-4258U	2	2.4GHz	3MB	8GB
Dual Socket Intel Xeon E5-2690	16	2.9GHz	20MB	64GB

Tabelle 3.1: Testsysteme für den Operator für arithmetische Ausdrücke.

Das Ergebnis der Additions- und der Vorzeichenwechseloperation für eine *column*-Länge von $N = 10.000$ ist in Abbildung 3.16 zu sehen. Beide Diagramme zeigen, dass Loop Unrolling die Ausführungszeit verringern kann. Beim Intel Core i5-4258U System wird die schnellste Ausführungszeit für die Vorzeichwechseloperation beim Abrollfaktor 10 erreicht und bei Additionsoperation beim Abrollfaktor 4. Beim Intel Xeon E5-2690 System hingegen wäre der Abrollfaktor 8 die beste Wahl. Bei den restlichen arithmetischen Operatoren liegt der Abrollfaktor k zwischen 2 und 6. Es gibt also keinen einheitlichen Abrollfaktor, der bei jedem Operator das beste Ergebnis liefert. Somit muss jeder Operator individuell eingestellt werden und ein Kompromiss gefunden werden, der bei vielen Systemen ein gutes bis sehr gutes Ergebnis erreicht.

Insgesamt zeigt sich, dass Loop Unrolling nur zu einer geringen Laufzeitverbesserung führt, die auf den beiden gemessenen Systemen zwischen 5 und 15% liegt.

Eine zweite Idee zur Optimierung des Operators ist die **Parallelisierung**. Für die Umsetzung der Parallelilität wurde die Programmierschnittstelle *OpenMP* [52] (Open Multi-Processing) gewählt, da sich damit eine for-Schleife gut beschleunigen lässt. Dabei konnte (bei einer *column*-Länge von $N = 10.000$) auf beiden

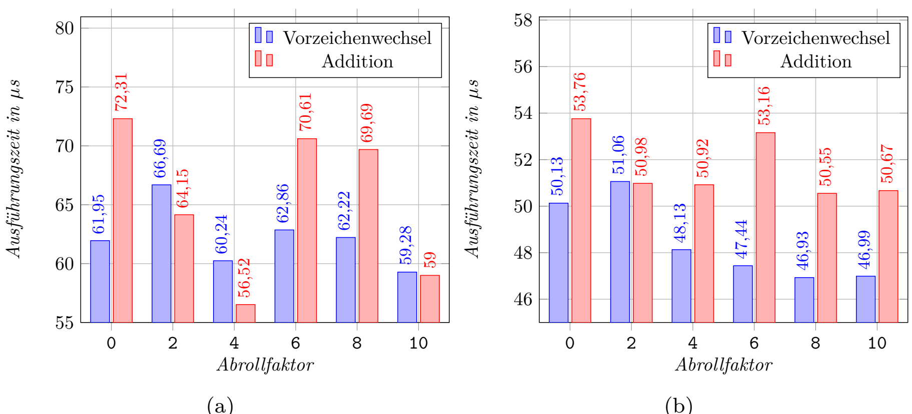


Abbildung 3.16: Ausführungszeit der Vorzeichenwechsel- und der Additionsoperation auf dem Intel Core i5-4258U (a) und dem Intel Xeon E5-2690 (b) System. Dargestellt ist jeweils der Mittelwert aus 50 Ausführungen der *column*-Länge $N = 10.000$ (zufällige Werte).

Systemen eine Ausführungszeit für die Addition erreicht werden, die ca. um den Faktor 9 (Intel Core i5-4258U System) bzw. 4,5 (Intel Xeon E5-2690 System) schneller ist (vgl. Abbildung 3.17). Bei den anderen Operatoren liegt der Faktor zwischen 3 und 9.

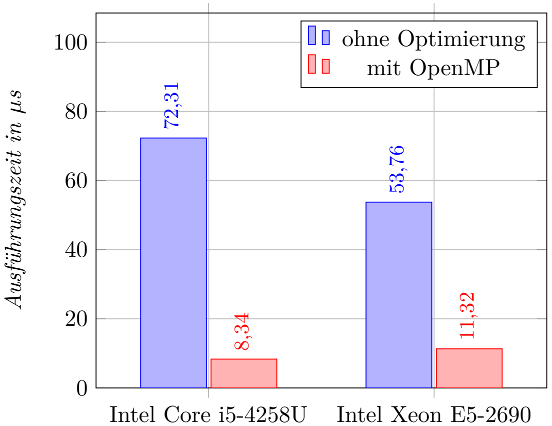


Abbildung 3.17: Ausführungszeit der parallelen und der nicht optimierten Version des Additionsoperators. Bei beiden Systemen ist jeweils der Mittelwert aus 50 Ausführungen für die *column*-Länge $N = 10.000$ (zufällige Werte) dargestellt.

Abschließend wurde für alle Operationen die Kombination aus OpenMP und Loop Unrolling implementiert und getestet. Dabei zeigt sich, dass die Verknüpfung der beiden Optimierungen eine gute Idee ist, wenn beim Loop Unrolling der beste Abrollfaktor (unterscheidet sich in den meisten Fällen vom besten Abrollfaktor für den sequentiellen Fall) gewählt wird. In Abbildung 3.18 ist der Vergleich zwischen

der nicht optimierten Version, der besten Loop Unrolling Variante, das Ergebnis der OpenMP Variante und die beste Kombination zwischen OpenMP und Loop Unrolling für die Operationen: Vorzeichenwechsel, Addition und Division auf den beiden oben beschrieben Systemen zu sehen. Dabei beträgt die *column*-Länge jeweils $N = 10.000$.
Auf den ersten Blick wird sofort deutlich, dass die Parallelisierung mit Hilfe von OpenMP den größten Anteil an der Performanceverbesserung bei allen Operationen einnimmt. Außerdem ist zu sehen, dass die Kombination beider Optimierungen, wie erwartet, die beste Ausführungszeit erreicht. Der Loop Unrolling Anteil ist dabei mal mehr und mal weniger stark sichtbar.

Auswirkungen der Optimierungen auf den Star Schema Benchmark

Nachdem der Operator für arithmetische Ausdrücke bisher nur mit Zufallszahlen getestet und gemessen wurde, sollen in diesem Abschnitt die Auswirkungen auf dem *Star Schema Benchmark* betrachtet werden. Dazu wurden die Querys, die den Operator für arithmetische Ausdrücke benötigen, 20 Mal ausgeführt und die mittlere Ausführungszeit des Operators bestimmt. Das Ergebnis ist in Abbildung 3.19 dargestellt.

Bei allen Querys ist zu sehen, dass die Ausführungszeit verringert wurde. Der größte Performancegewinn wird bei den Querys Q1.1, Q4.1 und Q4.2 sichtbar. Hier ist die optimierte Variante im Schnitt 1.5x schneller. Bei den anderen Querys (Q1.2, Q1.3 und Q4.3) ist die Beschleunigung nur sehr gering, da bereits die Variante ohne Optimierung weniger als 1ms benötigt. Insgesamt lässt sich jedoch sagen, dass alleine die Optimierung des Operators für arithmetische Ausdrücke keinen großen Einfluß auf die gesamte Ausführungszeit der einzelnen Querys hat, da im Moment die Querys des *Star Schema Benchmarks* zwischen 1.000 und 4.000ms benötigen (vgl. Abschnitt 5.1.1 Benchmarks).

Ausblick

Durch die Optimierung des Operators für arithmetische Ausdrücke konnte die Ausführungszeit gesenkt werden. Dies zeigten sowohl die Messungen des Operators an zufällig generierten Daten als auch die Auswirkungen der Optimierungen auf den *Star Schema Benchmark*. Dennoch könnte der Operator noch weiter verbessert werden. Deshalb werden im Folgenden einige Ideen vorgestellt.

Implementierung für GPUs Eine parallelel Implementierung für GPUs könnte unter Umstände die Ausführungszeit für sehr große Eingabe weiter senken. Dabei müsste natürlich bedacht werden, dass der Transfer von Daten zur und von der GPU eine lange Zeit in Anspruch nimmt und deshalb eine Art "Optimierer", den es bisher in *SliceDB* nicht gibt, die GPU- oder CPU-Version auswählt. Zudem müsste das verwendete System zu Beginn von *SliceDB* analysiert werden, ob eine GPU überhaupt vorhanden ist.

Kombination mit Aggregation Da beim Star Schema Benchmark einige Queries eine Kombination beider Operatoren benötigen, wäre ein Kombination der Operatoren denkbar. So könnte nicht nur die Ausführungszeit der Operationen

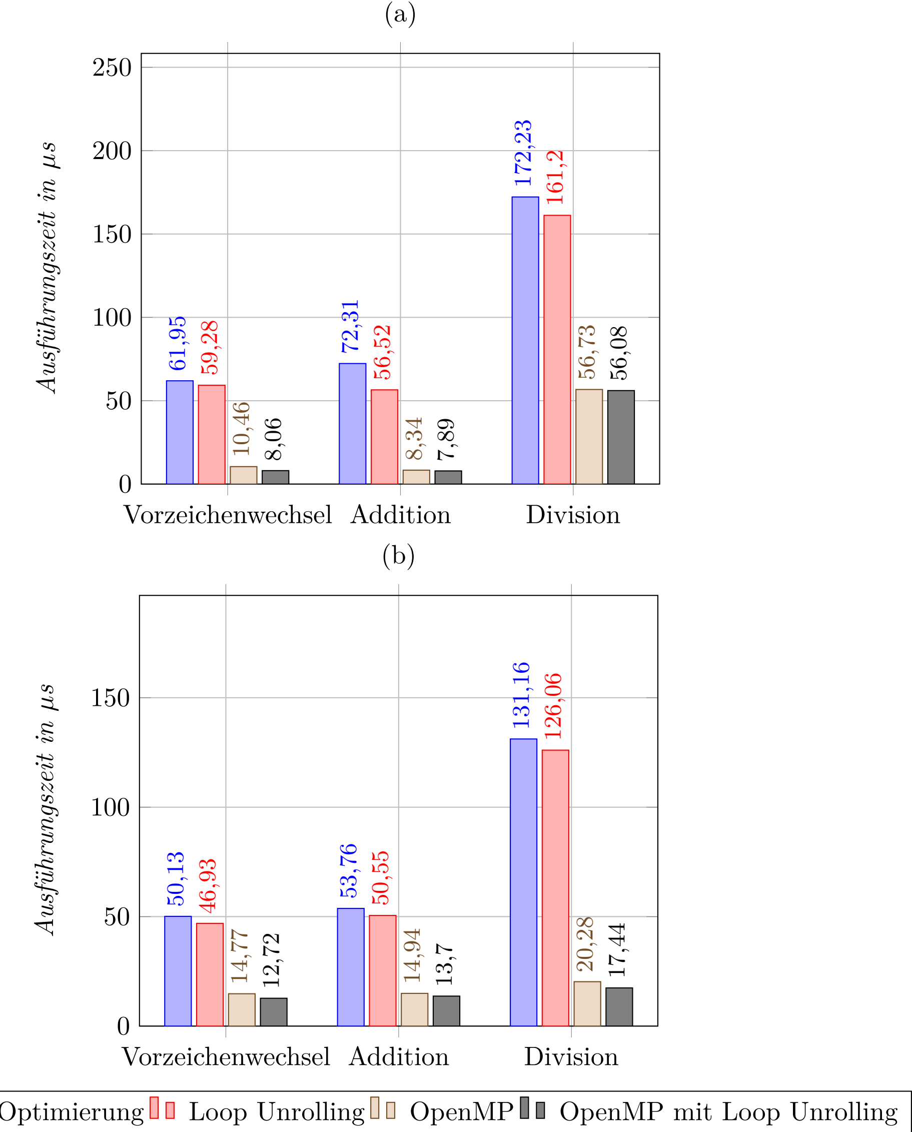


Abbildung 3.18: Vergleich der einzelnen Optimierungen für die Vorzeichwechsel-, Additions- und Divisionsoperation auf dem Intel Core i5-4258U (a) und dem Intel Xeon E5-2690 (b) System. Dargestellt ist jeweils der Mittelwert aus 50 Ausführungen der *column*-Länge $N = 10.000$ (zufällige Werte).

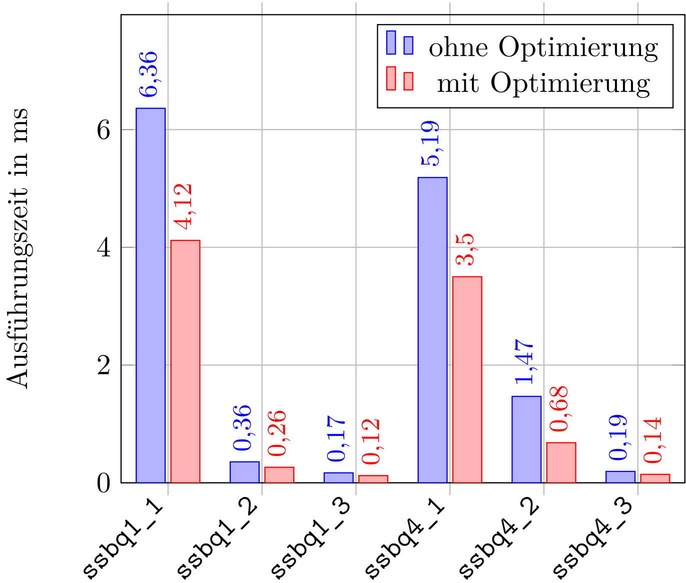


Abbildung 3.19: Ausführungszeiten des Operators für arithmetische Ausdrücke bei Ausführung der Anfragen Q1.□ und Q4.□ (nur diese Anfragen benötigen den Operator für arithmetische Ausdrücke) des Star Schema Benchmarks mit Skalierungsfaktor 10. Dargestellt ist der Mittelwert aus 20 Ausführungen für Variante mit und ohne Optimierungen. Hardware: Dual Socket Intel Xeon CPU E5-2690 (16 Cores@2.90GHz), 64GB DRAM.

an sich gesenkt werden, sondern zusätzlich auch ein Aufruf durch den Queryplan gespart werden.

3.3.6 Sortieren *(Ersin Özdemir)*

Der Sort-Operator dient zum Sortieren von Zeilen einer Tabelle. Dabei können je nach Bedarf eine oder mehrere Spalten individuell entweder aufsteigend (engl.: *ascending*, kurz: *asc*) oder absteigend (engl.: *descending*, kurz: *desc*) sortiert werden. In den gängigsten DMBS wird bei keiner Angabe der Sortierungsart eine aufsteigende Sortierung gewählt. Üblicherweise ist das Sortieren die letzte Operation, die bei einer Anfrage durchgeführt wird. Im Gegensatz zu anderen Operatoren, benötigt dieser Operator die vollständige Menge der relevanten Daten und kann nicht auf einzelnen Blöcken durchgeführt werden. Anders ausgedrückt: er ist *blockierend*.

Die Syntax für den Sort-Operator sieht wie folgt aus:

SELECT spalte1, spalte2, . . .
FROM tabelle
ORDER BY spalte1 ASC|DESC, spalte2 ASC|DESC, . . .

Listing 3.15 aus dem Kapitel 3.3.7 zeigt eine beispielhafte Anwendung des Operators innerhalb des Star-Schema-Benchmarks.

In diesem Kapitel soll der Operator anhand von Pseudocode erläutert werden. Im Folgenden unterscheiden wir **Singlesort** vom **Multisort**. Während Singlesort lediglich auf den Daten einer einzigen Spalte arbeitet, kann Multisort unter Verwendung des Singlesorts auf mehreren Spalten sortieren.

Singlesort

Daten können auf unterschiedlichste Arten sortiert werden. Ein DBMS, wie auch **SliceDB**, unterstützt normalerweise mehrere Datentypen. Abhängig vom vorliegenden Datentyp einer Spalte können die Daten *alphabetisch (a-z)*, *chronologisch (alt nach neu)* oder *numerisch (nach Zahlen)* sortiert werden. Dementsprechend verfügt die Sort-Klasse über ein Template, mit dem sich typisierte Objekte der Klasse erstellen lassen. Zusätzlich zum Datentyp wird auch die Information benötigt, in welcher *Reihenfolge (aufsteigend oder absteigend)* die Daten sortiert werden sollen.

In Hinblick auf Multisort müssen beim Singlesort gewisse **Voraussetzungen** erfüllt werden. Zum einen muss der Operator *Grenzen* unterstützen, sodass die gegebene Datenmenge von einem Startpunkt bis zu einem Endpunkt sortiert werden kann, ohne dass sich die Sortierreihenfolge der Daten außerhalb dieser Grenzen ändert. Zum anderen muss der Sortieralgorithmus die *Stabilität* bewahren.

Stabilität *Ein Sortieralgorithmus ist dann stabil, wenn nach der Sortierung die ursprüngliche Anordnung gleicher Elemente erhalten bleibt.* [38, Fußnote]

So bleibt beispielsweise eine alphabetisch sortierte Liste mit Personen und deren Zuweisung zu einer Raumnummer dann stabil, wenn nach einem erneuten Sortieren nach der Raumnummer die Personen mit der gleichen Raumnummer immer noch alphabetisch sortiert sind. Stabile und instabile Sortierverfahren weisen keine Unterschiede im Ergebnis auf, wenn es keine Duplikate unter den Schlüsseln gibt. Tabelle 3.2 zeigt eine beispielhafte Sortierung aller Teilnehmer der Projektgruppe.

Bekannte stabile Sortierverfahren wären u.a. *Bubblesort*, *Instertionsort* oder auch *Mergesort*. [64] In SliceDB kommen sowohl die *stabile* als auch die *bitonische Variante des Mergesort* zur Verwendung. Die letztere von beiden ist zwar nicht stabil, allerdings bietet sie die Eigenschaft, immer genau gleich viele Schritte zu brauchen, unabhängig davon, ob die Eingabedaten in richtiger Reihenfolge, in umgekehrter Reihenfolge oder zufällig sortiert sind. [38, Seite 5] Dadurch ist der Operator lediglich abhängig von der vorliegenden Hardware und der Datenmenge, jedoch nicht von den Daten selbst. Die Implementierung der bitonischen Variante war ursprünglich deshalb vorgesehen, um Optimierungen für die Operatoren besser auszuwerten. Die nicht-Stabilität der bitonischen Variante umgeht der Multisort-Operator jedoch durch eine *gruppenweise* Sortierung und stellt daher kein Problem für die *Korrektheit* der Sortierung dar.

Eingabe		nicht stabil		stabil	
Andreas	1	Stefan	1	Andreas	1
Christian	2	Jens	1	Eric	1
Eric	1	Eric	1	Jens	1
Ersin	3	Andreas	1	Stefan	1
Harun	4	Majuran	2	Christian	2
Jan	3	Christian	2	Majuran	2
Jens	1	Maximilian	2	Maximilian	2
Majuran	2	Ersin	3	Ersin	3
Maximilian	2	Jan	3	Jan	3
Milad	4	Philipp	3	Philipp	3
Philipp	3	Tristan	4	Harun	4
Sebastian	4	Sebastian	4	Milad	4
Stefan	1	Milad	4	Sebastian	4
Tristan	4	Harun	4	Tristan	4

Tabelle 3.2: Stabile Sortierung

Klasse *Sort* Die Klasse *Sort* beinhaltet zwei öffentliche Funktionen zur Sortierung:

- void **sort_ascending**
  (Size n, T* column, Index left_bound, Index right_bound, RID* rids)
- void **sort_descending**
  (Size n, T* column, Index left_bound, Index right_bound, RID* rids)

Beide Funktionen sortieren die Daten entsprechend aufsteigend oder absteigend. Als Eingabe wird ein Zeiger auf das Datenarray *column* erwartet, sowie die dazugehörige Größe bzw. Anzahl der Elemente *n*. Zwei Indizes *left_bound* und *right_bound* geben an, von wo bis wo sortiert werden soll. Soll die gesamte Datenmenge sortiert werden, muss beim Aufruf die linke Grenze auf *0* und die rechte Grenze auf *n-1* gesetzt werden. Nach Ausführung der Funktion befindet sich im Array *rids*, welches per Zeiger übergeben wird und ebenfalls die Größe *n* besitzt, die sortierte Positionsreihenfolge der Daten.

Als Pseudocode sehen die Funktionen wie folgt aus:

Algorithmus 1 : sort_ascending und **sort_descending**

1 Erstelle eine Kopie der Datenspalte
2 Reserviere Speicherplatz für temporäre Hilfsspalten
3 Befülle die noch leere RID-Liste von *0* bis *n-1*
4 Rufe Mergesort auf
5 Gib den Speicherplatz der Kopie wieder frei
6 Gib den Speicherplatz der temporären Hilfsspalten wieder frei

Zunächst wird eine Kopie der Daten erzeugt, um sicherzustellen, dass an den ursprünglichen Daten keine Veränderung vorgenommen wird. Außerdem wird Speicherplatz für globale, temporäre Hilfsspalten für die nachfolgende Sortierung

reserviert. Die RID-Liste wird anschließend mit den Werten von *0* bis *n-1* befüllt. Dann wird Mergesort aufgerufen und die RID-Liste dem Sortierkriterium entsprechend angeordnet.

Die Klasse besitzt drei private Funktionen, welche zur Sortierung verwendet werden:

- void **mergesort**
  (Index left_bound, Index right_bound, Size n, T* column, RID* rids, bool ascending, bool threaded)

- void **merge_bitonic**
  (Index left_bound, Index mid, Index right_bound, Size n, T* column, RID* rids, bool ascending)

- void **merge_stable**
  (Index left_bound, Index mid, Index right_bound, Size n, T* column, RID* rids, bool ascending)

Der Parameter *ascending* gibt an, ob die Daten aufsteigend oder absteigend sortiert werden sollen. Bei den restlichen Parametern werden lediglich die Werte der Parameter der öffentlichen Funktionen weitergeleitet.

Als Teil der Optimierungen kann für Mergesort mit dem Parameter *threaded* angegeben werden, ob dieser für nachfolgende, rekursive Aufrufe Threads zur parallelen Abarbeitung erzeugen soll. Eine Synchronisierung ist nicht benötigt, da die Threads auf unterschiedlichen, zueinander disjunkten Teilbereichen arbeiten. In **sort_ascending** und **sort_descending** wird dieser Parameter auf wahr gesetzt, bei den rekursiven Aufrufen auf falsch, damit der gesamte Sortierungsprozess auf maximal 2 Threads läuft. Die Optimierungs-Option **__SORT_PARALLEL__** kann verwendet werden, um Parallelität für den Operator gänzlich ein- oder auszuschalten. Die Optimierungs-Option **__SORT_STABLE__** entscheidet darüber, ob die bitonische oder stabile Variante von Merge verwendet wird.

Mergesort Mergesort ist ein vergleichsbasiertes Sortierverfahren. Eine sortierte Folge wird beim Mergesort durch das Verschmelzen von sortierten Teilstücken erreicht. Dabei kommt das Prinzip des *Divide-and-Conquer* zum Einsatz. Im ersten Schritt werden die Daten in zwei Hälften aufgeteilt (*divide*) und anschließend sortiert (*conquer*). Die dann in sich sortierten Hälften werden zum Schluss zu einer insgesamt sortierten Folge verschmolzen.

Besteht die Datenmenge innerhalb der Grenzen aus mehr als nur einem Element, wird die Mitte aus der linken und rechten Grenze ermittelt. Ist die Anzahl der Elemente ungerade, wird zur Bestimmung der Mitte abgerundet. Die Datenmenge wird nun in zwei Hälften aufgeteilt, für die jeweils **mergesort** mit den neuen Grenzen rekursiv aufgerufen wird.

Als Pseudocode sieht **mergesort** wie folgt aus:

Algorithmus 2 : mergesort

```
1  if linkeGrenze < rechteGrenze then
2  |  Bestimme Mitte (abgerundet)
3  |  Rufe mergesort für linke Hälfte auf
4  |  Rufe mergesort für rechte Hälfte auf
5  |_ Verschmelze beide Hälften durch merge miteinander
```

Abbildung 3.20 zeigt eine beispielhafte Aufteilung der Daten. Zum Schluss werden beide Hälften unter Aufrechterhaltung der Sortierung verschmolzen.

Als Pseudocode sieht **merge_bitonic** wie folgt aus:

Algorithmus 3 : merge_bitonic

```
1  Befülle die Hilfsspalten Kopie_Spalte und Kopie_RIDs innerhalb der
     Grenzen
2  Erstelle zwei Indizes i = linkeGrenze und j = rechteGrenze
3  while k = linkeGrenze ≤ rechteGrenze do
4  |  if Kopie_Spalte(i) ≥ Kopie_S(j) (oder entsprechend ≤) then
5  |  |  Spalte(k) = Kopie_Spalte(i)
6  |  |  RIDs(k) = Kopie_RIDs(i)
7  |  |  i = i + 1
8  |  else
9  |  |  Spalte(k) = Kopie_Spalte(j)
10 |  |  RIDs(k) = Kopie_RIDs(j)
11 |  |  j = j − 1
12 |_ k = k + 1
```

Als erstes kopiert sich **merge_bitonic** die Daten und RIDs, welche sich innerhalb der Grenzen befinden, in die Hilfsspalten. Die Daten der rechten Hälfte werden in umgekehrter Reihenfolge kopiert. Anschließend werden zwei Indizes i und j erstellt, mit Hilfe derer die Kopien durchlaufen werden. Dabei wird der Startpunkt von i auf die linke Grenze und der Startpunkt von j auf die rechte Grenze gesetzt. Während der Index i von links nach rechts läuft, läuft der Index j von rechts nach links. Die Indizes müssen beim Durchlaufen der Kopie nicht zwangsweise in „ihren Hälften“ verbleiben. Paarweise werden dann die Werte verglichen und mit Hilfe des Index k sortiert in die Datenspalte zurück kopiert. Liegen zwei gleiche Werte vor, wird der Wert aus der linken Hälfte übernommen. Das Verschmelzen der beiden Hälften ist abgeschlossen, sobald sich i und j kreuzen. In Abbildung 3.21 wird das Verfahren beispielhaft demonstriert.

Beim Erstellen der Kopien sei angemerkt, dass nur die Daten innerhalb der Grenzen kopiert werden. Es wird dafür Speicherplatz in der Größe von n in globalen Hilfsspalten reserviert. In der ersten Phase der Projektgruppe wurden diese Hilfsarrays unglücklicherweise lokal mit jedem Aufruf von **mergesort** allokiert, was zu einem merkbaren Performanzverlust geführt hatte, da immer

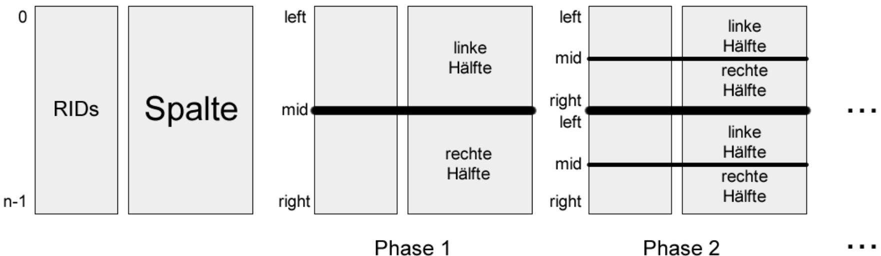


Abbildung 3.20: Ermittlung der Grenzen zum Verschmelzen beim Mergesort.

ursprüngliche Daten

RIDs	0	1	2	3
Spalte	d	c	b	a

Teilstücke bereits sortiert

k = left ⟶ mid right

RIDs	1	0	3	2
Spalte	c	d	a	b

linke Hälfte | rechte Hälfte

Hilfsspalten

i ⟶ ⟵ j

Kopie_R	1	0	2	3
Kopie_S	c	d	b	a

Ergebnis

RIDs	3	2	1	0
Spalte	a	b	c	d

Abbildung 3.21: Aufsteigendes Sortieren eines ursprünglich umgekehrt sortierten Datenarrays mit der bitonischen Variante vom Mergesort; Zwei 2-elementige Hälften werden miteinander verschmolzen.

wieder Speicherplatz reserviert, jedoch dann noch nicht einmal vollständig genutzt wurde.

Als Pseudocode sieht **merge_stable** wie folgt aus:

Algorithmus 4 : merge_stable

```
1  Befülle die Hilfsspalten Kopie_Spalte und Kopie_RIDs innerhalb der
   Grenzen
2  Erstelle drei Indizes i = linkeGrenze, j = rechteGrenze und
   k = linkeGrenze
3  while i ≤ Mitte & j ≤ rechteGrenze do
4      if Kopie_Spalte(i) ≥ Kopie_S(j) (oder entsprechend ≤) then
5          Spalte(k) = Kopie_Spalte(i)
6          RIDs(k) = Kopie_RIDs(i)
7          i = i + 1
8      else
9          Spalte(k) = Kopie_Spalte(j)
10         RIDs(k) = Kopie_RIDs(j)
11         j = j − 1
12     k = k + 1
13 while i ≤ Mitte do
14     Spalte(k) = Kopie_Spalte(i)
15     RIDs(k) = Kopie_RIDs(i)
16     k = k + 1
17     i = i + 1
```

Auch beim stabilen Mergesort werden die Daten zunächst innerhalb der Grenzen in die Hilfsspalten kopiert. Allerdings werden auch bei der rechten Hälfte die Daten in der ursprünglichen Reihenfolge und nicht verkehrt herum wie bei der bitonischen Variante eingefügt. Anschließend starten die zwei Indizes i und j entsprechend an der linken Grenze und ein Element weiter als in der Mitte. Beide Indizes laufen von links nach rechts und verbleiben in „ihren Hälften“. Durch dieses Verfahren ist auch sichergestellt, dass die Daten in ihrer ursprünglichen Reihenfolge abgearbeitet werden und die Stabilität bewahrt wird. Das Sortieren selbst verläuft wie in der bitonischen Variante. Das Verschmelzen ist dann abgeschlossen, wenn eine Hälfte abgearbeitet wurde. Die Daten der anderen Hälfte werden dann lediglich angehangen, da sie bereits in sich sortiert sind. Insgesamt ist diese Variante dadurch auch effizienter als die andere. In Abbildung 3.22 wird das Verfahren beispielhaft demonstriert.

Korrektheit Das **mergesort** teilt die Datenmenge soweit auf, bis jeweils nur noch ein Element vorhanden ist. Eine Menge, die aus nur einem Element besteht, ist in sich korrekt sortiert. Beim Verschmelzen zweier Hälften innerhalb von **merge_bitonic** laufen die Indizes i und j in gegensätzlicher Richtung innerhalb der definierten Grenzen aufeinander zu. Beim Verschmelzen zweier Hälften innerhalb von **merge_stable** laufen die Indizes i und j innerhalb der definierten Grenzen in die gleiche Richtung. Beim paarweisen Vergleichen der vorliegenden, in ihrer

ursprüngliche Daten	RIDs	0	1	2	3
	Spalte	d	c	b	a
		k = left ⟶		mid	right
Teilstücke bereits sortiert	RIDs	1	0	3	2
	Spalte	c	d	a	b
		linke Hälfte		rechte Hälfte	
		i ⟶		j ⟶	
Hilfsspalten	Kopie_R	1	0	3	2
	Kopie_S	c	d	a	b
Ergebnis	RIDs	3	2	1	0
	Spalte	a	b	c	d

Abbildung 3.22: Aufsteigendes Sortieren eines ursprünglich umgekehrt sortierten Datenarrays mit der stabilen Variante vom Mergesort; Zwei 2-elementige Hälften werden miteinander verschmolzen.

Hälfte sortierten Elemente wird jeweils das kleinere (oder entsprechend größere) Element zurück kopiert und der passende Index erhöht bzw. verringert. Dadurch wird kein Element bei einem einzigen Aufruf der Funktion zweimal verglichen oder kopiert. Kreuzen sich beide Indizes beim **merge_bitonic**, wurden alle Elemente verglichen und in der gewünschten Sortierreihenfolge zurück kopiert. Beim **merge_stable** wird eine Hälfte früher abgearbeitet als die andere, sodass die noch nicht abgearbeitete, jedoch in sich sortierte Hälfte, lediglich angehangen wird. Innerhalb der gleichen Rekursionsebene gibt es keine zwei Aufrufe der Funktion, die die gleichen Grenzen aufweisen. Dadurch kann ein bereits sortiertes Teilstück nicht nochmals umsortiert werden. Somit werden stets sortierte Teilstücke bis zum ersten Aufruf weitergeleitet und das gewünschte Ergebnis erzielt.

Laufzeit Besteht die Datenmenge aus nur einem Element, muss nicht sortiert werden. Sind jedoch mehr Elemente vorhanden, ruft das **mergesort** zweimal rekursiv das **merge** mit jeweils der Hälfte aller Daten auf. Das **merge** braucht maximal n Schritte, um eine Kopie der Daten zu erstellen. Genauso viele Schritte werden beim zurück kopieren benötigt. Demnach ergibt sich eine Laufzeit von

$T(n) \leq 2n + 2T(\frac{n}{2})$ und
$T(1) = 0$.

Insgesamt ergibt sich dadurch eine Laufzeit von

$T(n) <= 2n \cdot \log(n) \in \mathcal{O}(n \cdot \log(n))$.

Die untere Schranke für das Sortierproblem ist $\mathcal{O}(n \cdot \log(n))$. Daher ist die Laufzeit des Algorithmus optimal. [38, Seite 6] [64, Folie 9 ff.]

Klasse *Sort-Test* In der Test-Klasse für den Sort-Operator werden drei Datenarrays angelegt. Diese enthalten jeweils zufällige *Integer*-, *Float*- und *String*-Werte in beliebiger Reihenfolge. Im Test werden diese Daten mit Hilfe von entsprechend typisierten Sort-Objekten sortiert. Anschließend wird geprüft, ob die Werte in einer korrekt aufsteigenden bzw. absteigenden Reihenfolge vorliegen. Ist dies der Fall, war der Test erfolgreich, andernfalls nicht. Die Klasse beinhaltet ebenfalls einen Test, um die Stabilität des Sortierverfahrens zu testen. Außerdem existiert ein Test, um die Performanz bei einer festen Anzahl von zufällig generierten Daten zu überprüfen.

Anmerkung: Folgende Tests wurden auf einem Intel Xeon CPU E3-1230 v3 @ 3.30GHz durchgeführt.

Abbildung 3.23 zeigt die Ausführungszeit bei unterschiedlichen Spaltengrößen. Dabei ist eine lineare Steigung der benötigten Zeit zu erkennen. Das stabile Verfahren ist ein wenig schneller als die bitonische Variante des Merge. Eine deutliche Steigerung der Performanz ist zu erkennen, wenn das Sortieren parallel vonstatten geht.

Einbettung Damit der Sort-Operator verwendet werden kann, wurde er in den *Executor*, sowie in die Klassen *BaseColumn*, *PageColumn* und *TypedColumn* mit entsprechenden Funktionen und Weiterleitungen eingebettet. Somit kann jede *Column* ein typisiertes Sort-Objekt erstellen und die Sortierung auf den eigenen Daten aufrufen. Als Rückgabewert wird stets ein *BaseColumnPtr* zurückgegeben, welcher auf ein Array mit der sortierten Positionsliste zeigt.

Multisort

Die Sortierung nach einer einzigen Spalte ist manchmal nicht präzise genug. In den meisten Anwendungsfällen ist es interessant, eine Sortierung nach mehreren Kriterien durchzuführen. Beispielsweise könnte eine Kundenliste nach Nachnamen und anschließend abhängig vom Nachnamen nach dem Vornamen

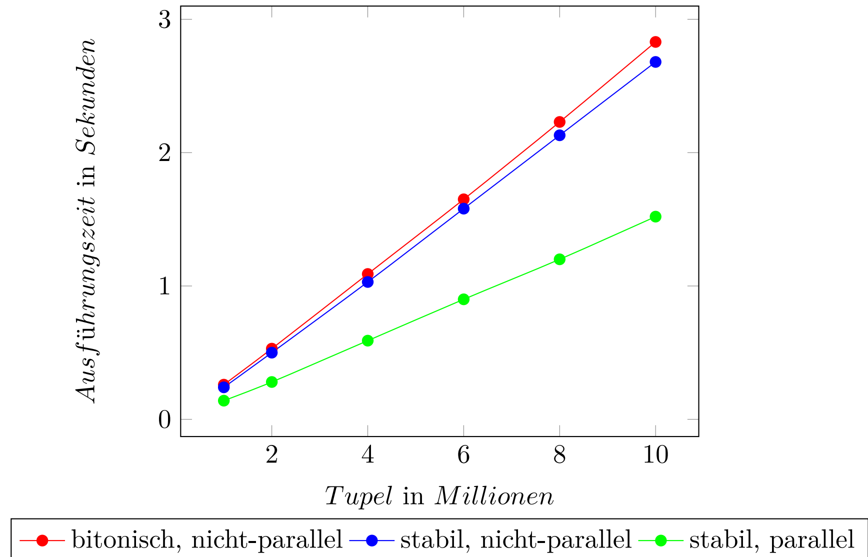


Abbildung 3.23: Laufzeit des Singlesort-Operators bei unterschiedlichen Spaltengrößen.

sortiert werden. Dabei darf natürlich die Sortierung vorangegangener Spalten nicht verloren gehen, wenn nachfolgende Spalten sortiert werden. Außerdem soll die Sortierung stabil sein (siehe Stabilität, Seite 133).

Klasse *Multisort* Der Multisort-Operator muss die unterschiedlichsten Datentypen, die in SliceDB zum Einsatz kommen, berücksichtigen. Hier liegt auch schon die erste Herausforderung, denn die Klasse Multisort darf keine Templates verwenden, um die Typinformation einer Spalte festzuhalten, da die Eingabe mehrere Spalten beinhalten kann, die unterschiedliche Datentypen aufweisen können. Würde hier ein Template zum Einsatz kommen, müssten alle Spalten, die der Funktion übergeben werden, vom gleichen Typ sein, was jedoch nicht erwünscht ist.

Stattdessen muss an dieser Stelle abstrahiert werden. Anstatt für jede Spalte ein Sort-Objekt zu erstellen, wird das Sortieren von der Spalte selbst übernommen (siehe Einbettung, Seite 140). Da es in SliceDB mehrere *Column*-Klassen gibt, alle jedoch von *BaseColumn* erben, wird eben eine Liste dieser als Eingabe erwartet.

Die Klasse *Multisort* beinhaltet eine öffentliche Funktion zur Sortierung:

- BaseColumnPtr **sort**
  (BaseColumnVectorPtr& columns, OrderVectorPtr orders)

In einem Vektor *columns* werden die Spalten übergeben, die sortiert werden sollen. Wie auch beim Singlesort benötigt die Funktion die Anzahl der Elemente n innerhalb der Spalten. Da die Größe der einzelnen Spalten stets gleich ist, kann hier beispielsweise die Größe der ersten Spalte verwendet werden. Die Anzahl der

Spalten *m* wird ebenfalls benötigt und kann aus der Größe des Vektors *columns* entnommen werden. Die Sortierungsart der Spalten kann unterschiedlich sein. So kann beispielsweise die erste Spalte aufsteigend, die letzte aber absteigend sortiert werden. Im Vektor *orders*, welches ebenfalls die Größe *m* besitzt, werden die Informationen zur Sortierungsart zu den Spalten erwartet. Die Reihenfolge der Spalten in *columns* (und *orders*) gibt zugleich die Reihenfolge an, in welcher diese sortiert werden. Ist die Funktion abgearbeitet, wird ein Zeiger auf eine *BaseColumn* zurückgegeben, in der sich die sortierte Positionsreihenfolge der Daten befindet.

Als Pseudocode sieht die Funktion wie folgt aus:

Algorithmus 5 : sort

1 Erstelle eine RID-Liste und befülle sie von *0* bis *n-1*
2 Erstelle eine Kopie der Datenspalten
3 Rufe Groupsort auf der ersten Spalte auf mit den Grenzen *0* und *n-1*
4 Gib die RID-Liste zurück

Im Vergleich zum Singlesort wird beim Multisort zunächst eine RID-Liste in Form eines *ArrayColumn* erstellt, auf das man mit einem *BaseColumnPtr* zugreifen kann. Der Grund hierfür liegt lediglich in der Verwendung des *Gather*-Operators, welche vom Groupsort gebraucht wird und ein manuelles *Gathern* verhindert. Außerdem soll das Allokieren der Ergebnisliste von Multisort übernommen werden und nicht wie beim Singlesort als Parameter überreicht werden. Beim Singlesort hatte diese Aufgabe die Spalte selbst übernommen.

Die Klasse besitzt eine private Funktion, welche zur Sortierung verwendet wird:

- void **group_sort**
  (Size n, Size m, BaseColumnPtr* cols, bool* ascending, BaseColumnPtr result, Size col_no, Index begin, Index end)

Neben den Informationen, die bereits beim Singlesort benötigt wurden, ist hier auch die Nummer der Spalte nötig, auf der sortiert werden soll. Dabei legen *begin* und *end* wieder die Grenzen fest.

Groupsort Um eine korrekte Sortierung zu ermöglichen, sortiert der Multisort-Operator gruppenweise. Die Idee dahinter ist, die Spalten nicht nacheinander *vollständig* zu sortieren, sondern *Gruppen* der vorherigen Spalte zu bilden und nur innerhalb der Gruppengrenzen zu sortieren. Da Gruppen in sich immer nur gleiche Werte aufweisen, ist die Korrektheit der Sortierung einer Gruppe unabhängig davon, ob der Singlesort ein stabiles oder nicht stabiles Verfahren nutzt. Der Aufruf von Groupsort wird rekursiv durch alle Spalten weitergeführt. Somit sortieren sich Gruppen innerhalb von Gruppen.

Als Pseudocode sieht **group_sort** wie folgt aus:

Algorithmus 6 : group_sort

```
1  Sortiere die angegebene Spalte innerhalb der vorgegebenen Grenzen
2  Gather die RID-Liste
3  if m ≠ 1 then
4      Gather über alle Spalten
5      if col_no ≠ m − 1 then
6          Erstelle eine noch leere Liste von Gruppen
7          Erstelle zwei Indizes für den Anfang und das Ende einer Gruppe
8          Setze die Grenzen der Gruppe zunächst auf den Startpunkt der
             Sortiergrenzen
9          while Gruppenanfang ≠ Ende do
10             Finde das Gruppenende
11             if Gruppenende > Ende then
12                 Setze das Gruppenende auf das Ende der Sortiergrenzen
13             Merke gefundene Gruppe in der Liste
14             Setze den Anfang der Gruppe auf das Ende der Gruppe
15         Rufe pro Gruppe Groupsort für die nächste Spalte mit den
             jeweiligen Gruppengrenzen auf
```

Der Aufruf von **group_sort** erfolgt zunächst mit der ersten Spalte und den Grenzen *0* und *n-1*. Die Spalte wird vollständig sortiert, um anschließend Gruppen zu finden. Bevor dies geschehen kann, müssen sowohl die RID-Liste als auch sämtliche Spalten *gegathert* werden, damit alles mit der aktuellen Sortierung übereinstimmt. Ist die erste Spalte sortiert, können die restlichen Spalten abhängig von der ersten (bzw. jeweils vorherigen) sortiert werden. Um die nötigen Gruppen zu bestimmen, wird die Spalte komplett durchlaufen. Solange kein Wert vorkommt, der sich von dem Wert davor unterscheidet, handelt es sich um dieselbe Gruppe. An dieser Stelle müssen Werte verglichen werden, jedoch fehlt wie bereits erwähnt die Typinformation der Spalte. Deshalb muss die Spalte selbst eine Funktion besitzen, die als Rückgabewert die Position liefert, an der ein neuer Wert vorkommt, beginnend bei einer Startposition.

Im *Executor* und in den Klassen *BaseColumn*, *PageColumn* und *TypedColumn* wird enstprechend folgende Funktion bereitgestellt:

- Index **get_tid_of_new_value** (Index begin)

Diese simple Funktion sieht als Pseudocode wie folgt aus:

Algorithmus 7 : get_tid_of_new_value

```
1  Erstelle einen Index end = begin
2  while Ende ≠ Spaltenende − 1 und Wert(end + 1) ≠ Wert(end) do
3      Erhöhe end um 1
4  Erhöhe end um 1
```

Ist das Ende gefunden, kann die Gruppe in die Liste der Gruppen der vorliegenden Spalte aufgenommen werden. Da die Funktion **get_tid_of_new_value** keine

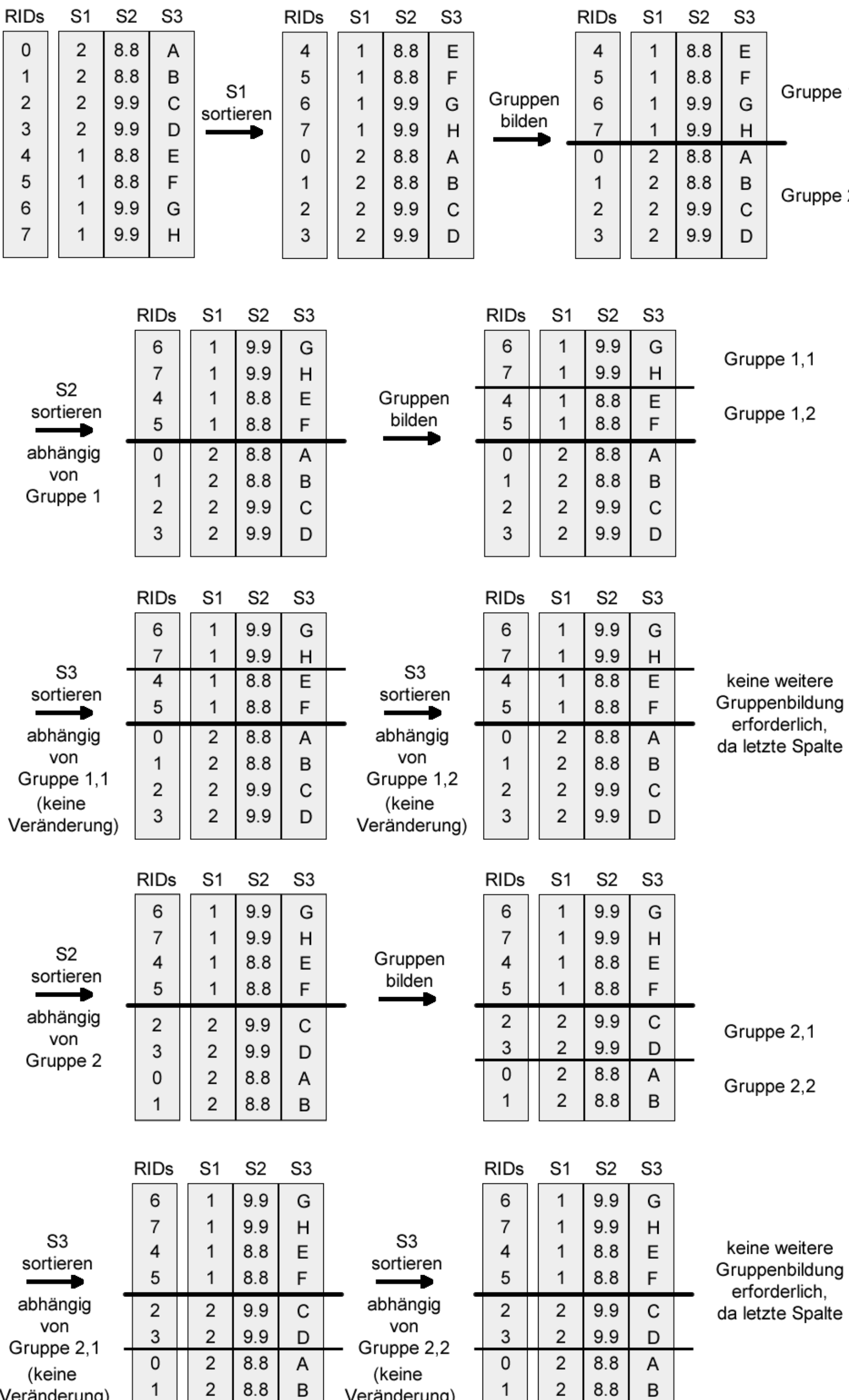


Abbildung 3.24: Beispielhafte Sortierung dreier Spalten mit unterschiedlichen Datentypen. Dabei wird die erste und letzte Spalte aufsteigend, die zweite Spalte absteigend sortiert.

Informationen über die Sortiergrenzen verfügt, sondern lediglich über das Ende der Spalte, muss das Gruppenende nachträglich korrigiert werden, falls es die Sortiergrenzen überschreitet. Sind alle Gruppen gefunden, kann die nachfolgende Spalte abhängig von diesen sortiert werden. Abbildung 3.24 verdeutlicht das Verfahren anhand von drei unterschiedlich getypten Spalten *S1*, *S2* und *S3*. Dabei werden *S1* und *S3* aufsteigend, *S2* jedoch absteigend sortiert.

Partielles Gathern Im Laufe der Optimierungsphase ist ein enormer Performanz-Verlust im Bereich des Gathern deutlich geworden. Das Problem liegt darin, dass Gruppen gebildet werden und nach der Sortierung einer solchen, alle Spalten gegathert werden müssen. Obwohl sich nur Teilbereiche durch ein Sortieren verändern, kopiert der Gather-Operator auch die restlichen, unveränderten Teilbereiche einer Spalte. Bei einer Spaltengröße von 200 Zeilen, in der in einer Teilsortierung lediglich 2 Zeilen vertauscht werden, wäre das ein unnötiger Kopiervorgang von 99% der Daten.

Um Kopiervorgänge einzusparen, werden bei dieser Optimierung lediglich Kopien der zu sortierenden Teilbereiche erstellt, die dann gegathert werden. Die Ergebnisse werden dann in die Teilbereiche der Spalten zurück geschrieben. Dadurch werden nur die relevanten Daten der Teilsortierung kopiert. In Abbildung 3.25 soll das Prinzip verdeutlicht werden.

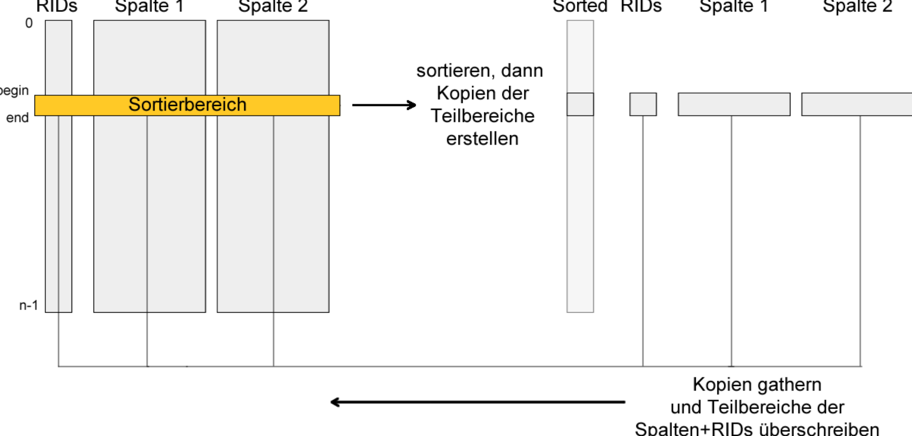


Abbildung 3.25: Prinzip des partiellen Gatherns.

Damit Kopien von Teilbereichen einer Spalte erstellt werden können, existiert folgende Funktion in den *Column*-Klassen, für die lediglich die Indizes für den Anfang und das Ende benötigt werden:

- BaseColumnPtr **partition_copy** (Index& begin, Index& end)

Um Bereiche einer Spalte zu überschreiben, existiert folgende Funktion, für die eine Datenspalte und deren Länge sowie der Startpunkt für das Überschreiben benötigt werden:

- BaseColumnPtr **override_data**
  (BaseColumnPtr& newData, Index& offset, Size& length)

Die Optimierungs-Option **__SORT_PARTIAL_GATHER__** kann verwendet werden, um das partielle Gathern für den Operator gänzlich ein- oder auszuschalten.

Limiterte Parallelität Nachdem die bereits erwähnten Optimierungen implementiert worden sind, ist beim Multisort-Operator ein Performanz-Verlust aufgefallen, sobald die Parallelität für den Singlesort-Operator eingeschaltet wurde, obwohl eine Verbesserung zu erwarten war. Das Problem liegt darin, dass bei großen Datenmengen tausende von Gruppen gebildet werden können, für die jeweils ein Singlesort ausgeführt wird. Läuft dieser parallel, werden tausende von Threads erzeugt und zerstört, was das System eher ausbremst als das Sortieren zu beschleunigen.

Als eine Schutzmaßnahme ist eine Abfrage eingefügt worden, die bei eingeschalteter Parallelisierung nur dann Threads erzeugt, wenn der zu sortierende Bereich größer ist als ein prozentualer Grenzwert (3%) der Datenmenge innerhalb der Spalte. Wird nur eine Spalte sortiert, ist diese Abfrage nicht von Bedeutung und behindert daher nicht die Tests zum Singlesort. Werden mehrere Spalten sortiert, greift der Mechanismus zu, sobald die zu sortierenden Teilbereiche zu klein werden.

Mit der Optimierungs-Option **__SORT_PARALLEL_LIMITER__** kann diese Schutzmaßnahme ein- oder ausgeschaltet werden.

Korrektheit Wenn nur eine Spalte sortiert werden soll, folgt die Korrektheit aus der Korrektheit des Singlesort. Erst bei einer Spaltenanzahl von zwei oder größer unterscheidet sich Multisort vom Singlesort. Ist die erste Spalte korrekt sortiert, werden Gruppen gebildet. Falls der Rückgabewert von **get_tid_of_new_value** die Sortiergrenzen überschreitet, wird dieser entsprechend auf das Ende gesetzt. Dieser Fall kann eintreten, wenn das Ende der letzten Gruppe gesucht wird, da die Funktion nur das Ende der Spalte, aber nicht das Ende der Sortiergrenzen kennt. Durch die zusätzliche Abfrage ist die Korrektheit der Grenzen für die Gruppen sichergestellt. Die Grenzen der Gruppen innerhalb einer Spalte überschneiden sich in keinem Fall und schließen in jedem Fall sämtliche Daten innerhalb der Sortiergrenzen ein. Der rekursive Aufruf für eine nachfolgende Spalte erfolgt nur innerhalb der Grenzen der jeweiligen Gruppen. Da eine Gruppe in jedem Fall nur gleiche Werte beinhaltet, macht es keinen Unterschied, ob Singlesort ein stabiles oder instabiles Verfahren nutzt. Die Korrektheit von Multisort kann jedoch nur garantiert werden, wenn nach jedem Sortieren einer Spalte, auch alle anderen Spalten sowie die RIDs mitsortiert werden. Da Singlesort außerhalb der Sortiergrenzen keine Veränderung vornimmt, wird beim Multisort stets nur innerhalb der Gruppengrenzen sortiert, sodass eine bereits zuvor vorgenommene Sortierung nicht durcheinander gebracht wird. Ist die letzte Spalte erreicht, werden keine Gruppen mehr gebildet, sodass die Korrektheit erneut aus der Korrektheit des Singlesort folgt.

Klasse *Multisort-Test* Die Test-Klasse für den Multisort-Operator sortiert drei Spalten mit unterschiedlichen Datentypen. Diese werden mit *Integer*-, *Double*- und *String*-Werten in zufälliger Reihenfolge befüllt. Ein Multisort-Objekt sortiert diese, wobei die erste und letzte Spalte aufsteigend und die zweite Spalte absteigend sortiert werden. Sind die Spalten nach dem Ausführen des Multisort-Operators entsprechend sortiert, war der Test erfolgreich, ansonsten nicht. Wie auch in der Testklasse zum Singlesort, existieren hier ebenfalls Tests zur Stabilität und zur Performanz.

Anmerkung: Folgende Tests wurden auf einem Intel Xeon CPU E3-1230 v3 @ 3.30GHz durchgeführt.

Abbildung 3.26 zeigt einen Vergleich zwischen dem Singlesort- und Multisort-Operator bei einer einzigen Spalte und der gleichen Anzahl an Tupeln. Parallelität und partielles Gathern ist für diesen Test ausgeschaltet. Dabei ist die Laufzeit des Multisort-Operators für eine Spalte höher, da dieser nach dem Sortieren die RID-Liste gathert. Hier ist also zu empfehlen, dass die Sortierung für eine Spalte mit dem Singlesort-Operator durchgeführt wird. Dieser Fall kommt im *Star Schema Benchmark* allerdings nicht vor und spielt für die Auswertung von *SliceDB* keine Rolle.

Abbildung 3.27 zeigt die Auswirkungen auf die Ausführungszeit des Operators, wenn partiell gegathert wird. Es sei angemerkt, dass die Anzahl der Tupel nicht mehr in Millionen, sondern in Tausend angegeben ist. Parallelität ist für diesen Test ausgeschaltet. Hierbei ist zu erkennen, dass diese Optimierung eine Beschleunigung von ungefähr 90% mit sich bringt.

Abbildung 3.28 zeigt die Auswirkung von Parallelität auf die Ausführungszeit des Operators. Dabei wird der Test für eine gewisse Anzahl an Tupel auf 3 Spalten nicht-parallel, parallel und limitiert-parallel ausgeführt. Partielles Gathern ist für diesen Test eingeschaltet. Die Abbildung gibt Aufschluss darüber, dass der Multisort-Operator Performanz einbüßen muss, sofern sämtliche Aufrufe von Singlesort parallel laufen, da das Erzeugen und Zerstören von Threads das System auslastet. Limitiert man diese, scheint zunächst kein Unterschied zur nicht-parallelen Variante vorhanden zu sein. Verringert man in den Tests allerdings die Gruppen und vergrößert zeitgleich die Anzahl der Elemente in derer, so lässt sich eine kleine Steigerung in der Performanz im Vergleich zur nicht-parallelen Variante feststellen, da bestimmte Gruppen groß genug werden, um diese parallel abzuarbeiten.

Abbildung 3.29 zeigt die Ausführungszeit des Operators bei unterschiedlichen Spaltengrößen und einer Anzahl von bis zu drei Spalten. Alle Optimierungen sind bei diesem Test eingeschaltet. Dabei ist erkennbar, dass zusätzliche Spalten eine deutliche Erhöhung in der Laufzeit verursachen. Die Ausführungszeit für nur eine Spalte erhöht sich ebenfalls mit steigender Anzahl an Tupeln, auch wenn dies in der Abbildung kaum sichtbar ist.

Einbettung Da der Multisort-Operator über keinerlei Typinformationen verfügt, kann dieser auch nicht im *Executor* eingebettet werden. Stattdessen greift

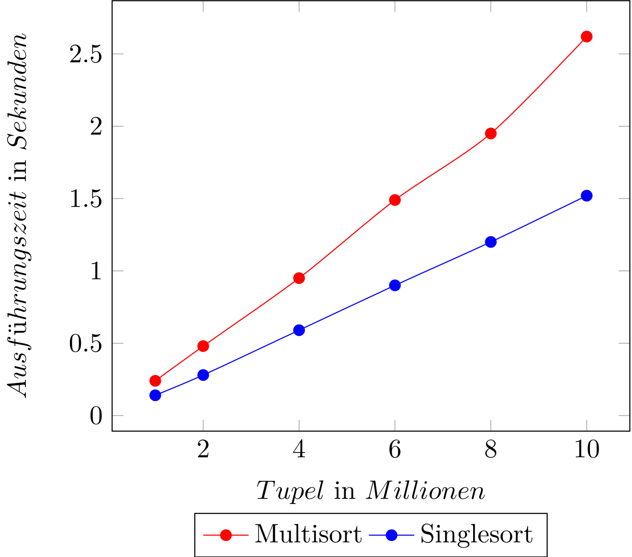


Abbildung 3.26: Vergleich der Laufzeiten der Sort-Operatoren bei einer einzigen Spalte und unterschiedlichen Spaltengrößen.

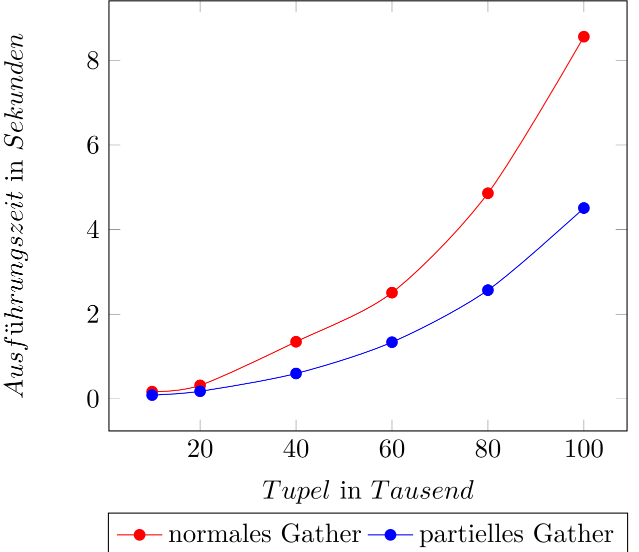


Abbildung 3.27: Laufzeit des Multisort-Operators bei 3 Spalten und verschiedenen Spaltengrößen unter Verwendung vom partiellen Gather.

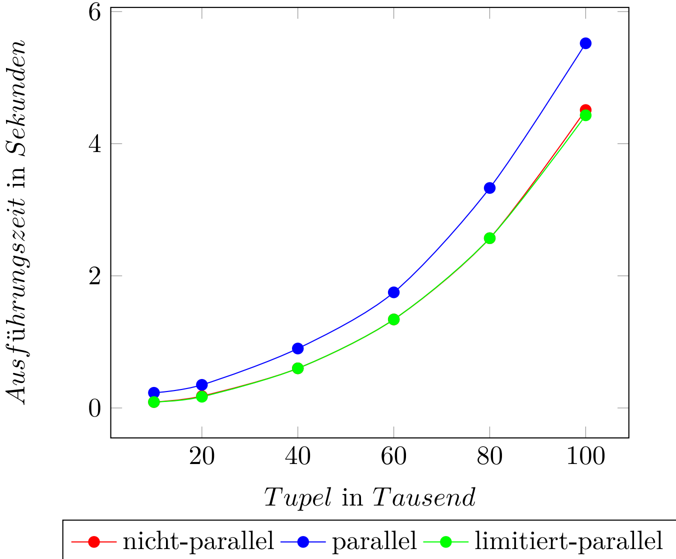


Abbildung 3.28: Laufzeit des Multisort-Operators bei 3 Spalten und verschiedenen Spaltengrößen unter Verwendung von Parallelität.

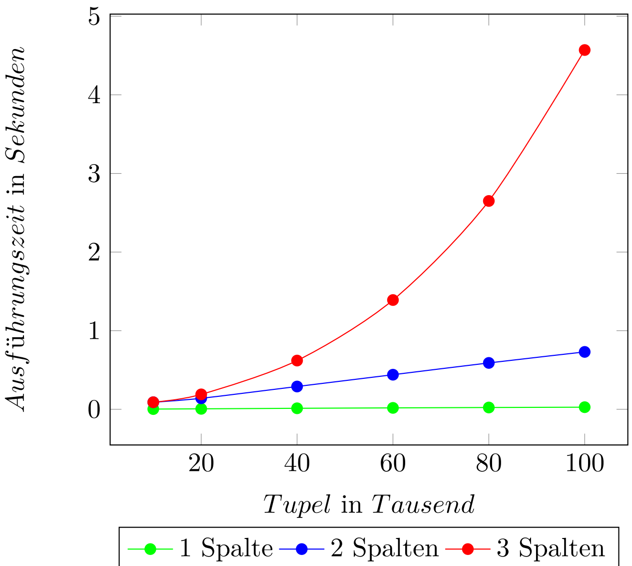


Abbildung 3.29: Laufzeit des Multisort-Operators bei einer unterschiedlichen Anzahl an Spalten und verschiedenen Spaltengrößen.

dieser auf der Plan-Ebene zu, wo normalerweise die Aufrufe für die jeweiligen Operatoren an den *Executor* weitergeleitet werden.

Fazit & Ausblick

Der Sort-Operator wurde in zwei einzelne Operatoren **Singlesort** und **Multisort** aufgeteilt, wobei letzteres auf dem ersteren beruht. Bei beiden Operatoren wurden aus Gründen der Implementation Kompromisse eingegangen und Möglichkeiten zur Optimierung untersucht.

Die nachfolgenden Tests zeigen die Auswirkung der Optimierungen des Sort-Operators auf die Performanz bei unterschiedlichen Skalierungsfaktoren innerhalb des *Star Schema Benchmarks* in **SliceDB** auf dem **Isengard** Server der Universität. Es stellte sich heraus, dass trotz einer Limitierung der Parallelität das System bei eingeschalteter Parallelisierung für das Benchmark deutlich langsamer lief, als wenn alle Optimierungen ausgeschaltet blieben. Daher ist die Parallelisierung für die nachfolgenden Tests ausgeschaltet und nur die restlichen Optimierungen werden in Betracht gezogen.

Abbildung 3.30 zeigt die Laufzeiten der verschiedenen Anfragen bei einem Skalierungsfaktor von 1. Dabei wird deutlich, dass sich lediglich bei 3.2 und 3.4 größere Unterschiede bemerkbar machen. Die restlichen Anfragen laufen durch die Optimierungen mal langsamer, mal schneller.

Wird der Skalierungsfaktor wie in Abbildung 3.31 auf 10 erhöht, so ist erkennbar, dass das partielle Gathern im Benchmark für jede Anfrage höhere Laufzeiten verursacht. Das System ist mit dem Erstellen der Teilkopien beim partiellen Gather deutlich länger beschäftigt, als wenn sämtliche Daten durch das normale Gather sortiert werden.

Abschließend können folgende Punkte zusammengefasst werden, die eine Übersicht und auch einen Ausblick über die implementierten Aspekte geben.

Speicherplatz Der Singlesort-Operator benötigt zum Sortieren einen zusätzlichen Speicherplatz in der Größe von n. Hilfsvariablen sollten stets global definiert werden, wenn Speicherplatz immer wieder benötigt wird, da das System sonst mit dem Allokieren derer ausgelastet wird.

Laufzeiten Die Laufzeit des Singlesort-Operators befindet sich bei der unteren Schranke von $\mathcal{O}(n \cdot \log(n))$. Die Implementation eines stabilen Sortierverfahrens hat die Laufzeit für beide Operatoren minimal verbessert. Der Multisort-Operator konnte durch ein partielles Gathern innerhalb der Tests für den Operator eine deutlich bessere Laufzeit erreichen, jedoch nicht beim *Star Schema Benchmark*.

Parallelität Der Singlesort-Operator konnte parallelisiert werden. Da der Multisort-Operator auf diesem beruht, kann die Optimierung auch für diesen genutzt werden. Jedoch stellte sich heraus, dass das Erzeugen und Zerstören von vielen Threads das System ausbremst und das Sortieren langsamer verläuft als ohne Parallelisierung. Um das Problem einzuschränken, wurde eine zusätzliche

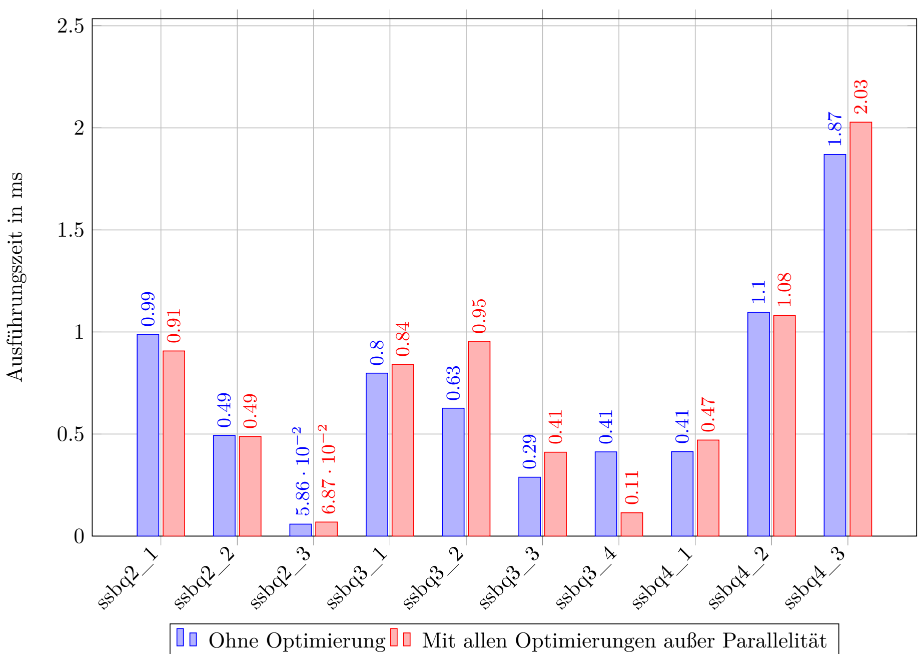


Abbildung 3.30: Ausführungszeiten der Anfragen des Star Schema Benchmarks für den Sort-Operator mit Skalierungsfaktor 1, zum einen ohne Optimierungen und zum anderen mit allen Optimierungen außer Parallelität.

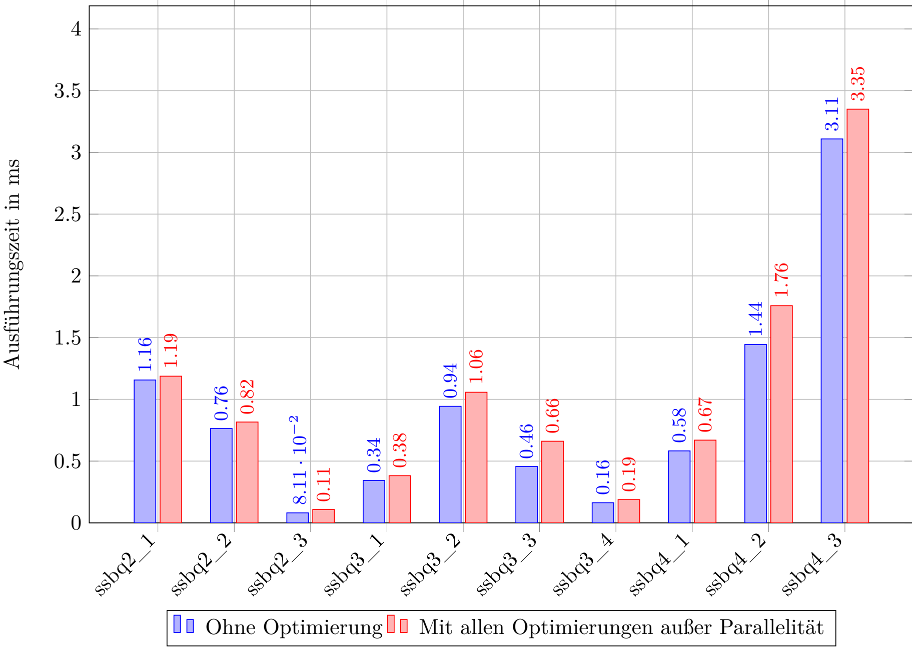


Abbildung 3.31: Ausführungszeiten der Anfragen des Star Schema Benchmarks für den Sort-Operator mit Skalierungsfaktor 10, zum einen ohne Optimierungen und zum anderen mit allen Optimierungen außer Parallelität.

Abfrage implementiert, die die parallele Abarbeitung limitiert und nur für große Teilsortierungen erlaubt. Die Laufzeit wurde dadurch für die Tests wieder besser, jedoch ergaben sich Probleme beim Ausführen des *Star Schema Benchmarks*. Daher wäre für eine zukünftige Weiterentwicklung von *SliceDB* eine völlig andere Herangehensweise, den Operator zu parallelisieren, wünschenswert. Hier könnten beispielsweise, abhängig vom vorliegenden System, die Gruppen selbst parallel abgearbeitet werden. Möglicherweise wäre eine völlig neue Implementierung des Multisort-Operators sinnvoller, da das partielle Gather ebenfalls für das Benchmark bei höheren Skalierungsfaktoren Probleme verursacht.

3.3.7 Gruppierungen & Aggregation *Eric Fiege*

Der GroupBy-Operator wird dazu verwendet, Werte zu gruppieren und basierend auf der Gruppierung weitere Spalten zu aggregieren. Dieses Kapitel beschreibt die Umsetzung des Operators für SliceDB. Dazu werden zunächst die Anfragen des Star-Schema-Benchmarks analysiert, um die Anforderungen an den Operator zu definieren. Aufbauend auf den Anforderungen wird dann beschrieben, wie der Operator für SliceDB umgesetzt wird. Es sollen zwei Varianten des Operators umgesetzt werden: Die erste Variante arbeitet mit im vorhinein sortierten Daten. Die zweite Variante arbeitet mit einer HashMap. Beide Varianten werden anhand von Pseudocode erläutert. Zum Schluss werden die Tests und die Performanz erläutert und durch einen Ausblick, weitere Möglichkeiten zur Verbesserungen aufgezeigt.

Anforderungen

Listing 3.15 zeigt die Anfrage 2.1 aus dem Star-Schema-Benchmark, an der die Verwendung des GroupBy-Operators im Star-Schema-Benchmark aufgezeigt werden soll. Andere Anfragen aus dem Benchmark verwenden den GroupBy-Operator auf die gleiche Art und Weise.

```
1: select
2:     sum(lo_revenue) as lo_revenue, d_year, p_brand1
3: from
4:     lineorder, dates, part, supplier
5: where
6:     lo_orderdate = d_datekey
7:     and lo_partkey = p_partkey
8:     and lo_suppkey = s_suppkey
9:     and p_category = 'MFGR#12'
10:    and s_region = 'AMERICA'
11: group by
12:    d_year, p_brand1
13: order by
14:    d_year, p_brand1;
```

Listing 3.15: Anfrage 2.1 des Star Schema Benchmarks

Die Zeilen 2, 11 und 12 aus Listing 3.15 sind für das GroupBy interessant. Sie zeigen, dass nach mehreren Spalten, nämlich *d_year* und *p_brand*1 gruppiert wird und eine weitere Spalte namens *lo_revenue* aggregiert werden soll. Das gleichzeitige Aggregieren mehrerer Spalten ist im Star-Schema-Benchmark nicht enthalten. Aus diesem Grund wird das GroupBy so gebaut, dass es nach mehreren Spalten gruppieren, jedoch nur eine Spalte aggregieren kann. Aggregieren auf mehr als einer Spalte kann jedoch durch den mehrmaligen Aufruf des GroupBy-Operators geschehen.

Der Star-Schema-Benchmark nutzt außerdem nur die Summe und keinen weiteren Aggregations-Typ wie Minimum oder Maximum. Da diese jedoch auch essentielle Operationen sind, soll nicht darauf verzichtet werden.

Es kann vorkommen, dass vor dem GroupBy-Operator der Sort-Operator (siehe Kapitel 3.15) ausgeführt wurde. Unter diesen Umständen kann sich ein Algorithmus, der sich die Sortierung zu nutze macht, stark davon profitieren. Aus diesem Grund soll es eine Variante des GroupBy-Operators geben, der auf nicht sortierten Daten arbeitet und eine zweite Variante implementiert werden, die mit vorsortierten Daten arbeitet. Zusammengefasst lauten die Anforderungen wie folgt:

- Es sollen mehrere Spalten gruppiert werden können.
- Mindestens eine Spalte muss aggregiert werden können.
- Als Aggregations-Typ wird mindestens die Summe umgesetzt. Minimum und Maximum sollen auch umgesetzt werden, sind jedoch nicht im Star-Schema-Benchmark enthalten.
- Bereits sortierte Daten sollen durch einen Algorithmus bearbeitet werden, der die Sortierung ausnutzt.
- Für unsortierte Daten wird ein anderer Algorithmus verwendet.

Umsetzung

Um eine lauffähige Variante von SliceDB zu erhalten, wird das GroupBy zunächst auf eine einfache Art und Weise umgesetzt. Sobald Anfragen, die auf dem GroupBy-Operator basieren, beantwortet werden können, soll die Performanz verbessert werden. Der GroupBy-Operator wurde im ersten Semester der Projektgruppe und als letzter der Operatoren in SliceDB integriert, um die Star-Schema-Benchmark Anfragen der zweiten Kategorie zu unterstützen.

Der Operator wird zunächst mit vollständigen Spalten anstatt mehrerer Vektoren arbeiten, da der letzte Vektor, der durch das Volcano Iterator Model in den Operator gelangt, z.B. auch noch den Wert einer Gruppe ändern könnte, die mit dem ersten Vektor erstellt wurde. Aus diesem Grund werden die Vektoren vor dem Aufruf des GroupBy-Operators an einer Stelle gesammelt und in ihrer Gesamtheit an den Operator übergeben.

Den Anforderungen entsprechend sollen zwei Varianten des GroupBy-Operators umgesetzt werden. Die erste Variante setzt das Gruppieren von Spalten unter Verwendung einer HashMap um. Die zweite Variante geht von einer bereits vorliegenden Sortierung aus und nutzt diese, um nach einer oder mehreren Spalten zu gruppieren. Beide Varianten haben einige Ähnlichkeiten, die in den folgenden Absätzen erläutert werden.

Template Beide Algorithmen des GroupBy-Operators werden mit dem Template Mechanismus von C++ umgesetzt, um redundanten Code zu vermeiden. Die Datentypen für das Template werden im Namespace Storage festgelegt. Zu beachten ist dabei, dass mit dem Datentyp `String` nicht alle Basis Operationen durchgeführt werden können. Diese werden jedoch gerade beim Aggregieren verwendet. Aus diesem Grund müssen die Funktionen für den Datentyp `String` explizit mit einer totalen Template-Spezialisierung angelegt werden.

Aggregation Beide Algorithmen müssen die Daten mindestens einer Spalte und in Abhängigkeit der vorhandenen Gruppen aggregieren. Im Star Schema Benchmark wird nur die Summe als Aggregations-Typ verwendet. Für SliceDB werden jedoch auch noch das Minimum, Maximum und, zu einem späteren Zeitpunkt, der Durchschnitt umgesetzt. Es gibt zwei Möglichkeiten die Aggregation einer Spalte im GroupBy-Operator umzusetzen:

- Die Werte werden unter Nutzung von Funktoren (siehe Kapitel 3.3.7), direkt wenn sie einer Gruppe zugewiesen werden, aggregiert.

- Es werden Tupel-IDs gesammelt und in Gruppen zusammengefasst. Die Tupel-IDs werden dann einer optimierten Aggregations-Funktion übergeben und gruppenweise berechnet.

Zunächst wird die erste Methode genutzt, um schneller eine funktionierende Version von SliceDB zu erhalten. Danach soll auch die zweite Methode umgesetzt und mit einem Test die performantere Version gefunden werden.

Funktoren Um die Ergebnisse des GroupBy-Operators zu berechnen werden zunächst Funktoren genutzt. Diese bestehen aus kleinen Klassen, die Operationen auf einem von der Klasse gehaltenen Wert ausführen. Für jeden Aggregations-Typen, das heißt zunächst Summe, Minimum und Maximum, wird deshalb eine neue Klasse erstellt. Die Klassen werden ebenfalls mit dem Template-Mechanismus von C++ umgesetzt und erben von der Klasse `Aggregator`. Sie implementieren folgende virtuelle Methode:

- `T aggregate(T pValue)`:
  Aggregiert den Wert des Parameters $pValue$ vom Datentyp T zu dem aktuell gespeicherten Ergebnis.

- `reset()`:
  Setzt den Wert des Funktors auf seinen Standardwert zurück.

- `T getResult()`:
  Gibt das Ergebnis der bisherigen Aggregation zurück.

- `setTo(T pValue)`:
  Setzt das Ergebnis des Funktors auf den Wert des Parameters $pValue$.

Die Algorithmen erwarten einen Pointer auf die `Aggregator` Klasse als Parameter für die Aggregat-Funktion. Der Funktor muss deshalb mit dem gleichen Typen, mit dem auch der GroupBy-Operator instanziiert wurde, instanziiert werden.

Bei der Verwendung innerhalb der Algorithmen speichert der Funktor den bisherigen Stand seiner Aggregation selbstständig. Falls eine neue Gruppe erstellt wird, wird der Funktor wieder auf seinen Basis-Wert zurückgesetzt. Wenn ein neuer Wert hinzugefügt werden soll, wird die `aggregate` Funktion des Funktors aufgerufen. Dieser berechnet dann den neuen Wert der Aggregation.

Ein- und Ausgabewerte Die Ein- und Ausgabewerte sind, mit Ausnahme der Sortierung, für beide Umsetzungen gleich. Der GroupBy-Operator benötigt folgende Eingabewerte:

- std::vector<std::shared_ptr<Storage::BaseColumn> >* pColumnVector
  Ein Vector mit Zeigern auf ungetypte Spalten: Enthält die Spalten nach denen gruppiert werden soll.
- T* pAggregateColumn
  Die Spalte, die aggregiert werden soll: Wird als Pointer auf einen spezifischen Datentyp, der durch das Template generiert wird, übergeben.
- Storage::Size &pSize
  Die Länge der Spalte: Wird für die Grenze zum Iterieren benötigt.
- Aggregator<T> *pFunctor
  Die Aggregations-Funktion: Ein Funktor oder ein Enum mit dem der Aggregations-Typ festgelegt wird.

Mit diesen Eingabewerten werden die Varianten des GroupBy-Operators aus dem Executor aufgerufen. Der Executor wird, anstatt direkt mit dem instanziierten Aggregator, mit einem Enum aufgerufen (siehe Kapitel 3.3.7). Das Ergebnis wird aus einem Paar von BaseColumnPtr gebildet und sieht folgendermaßen aus:

- std::pair<Storage::BaseColumnPtr, Storage::BaseColumnPtr>
  Ist der eigentliche Rückgabe-Wert des Operators und enthält die zwei nachfolgenden Werte, die als BaseColumnPtr übergeben werden.
- Storage::ArrayColumn<Storage::RID>* resultIDColumn
  Eine Spalte, die Tupel-IDs von gruppierten Werten enthält: Diese werden eventuell von nachfolgenden Operatoren genutzt.
- Storage::ArrayColumn<T>* resultAggregationColumn
  Eine Spalte, die die aggregierten Werte enthält.

Ein Beispiel für ein mögliches Ergebnis des GroupBy-Operators kann Tabelle 3.4 entnommen werden. Die Spalte TID enthält für jede Gruppe genau eine TID und dient dazu die Gruppe später anhand ihrer Werte identifizieren zu können.

Aufruf durch den Rest der Datenbank Der GroupBy-Operator wird letztendlich direkt aus dem Executor aufgerufen. Der Executor bekommt mit einer Ausnahme die gleichen Eingabeparameter wie das GroupBy. Anstatt des Funktors wird ein Enum übergeben, das die möglichen Werte Summe, Minimum und Maximum enthalten kann. Im Executor wird dann in Abhängigkeit vom übergebenen Enum ein Funktor erstellt. Der Executor wird widerum aus den Colum Klassen aufgerufen. Dazu gehören die Klassen BaseColumn, PageColumn, PlainColumn und TypedColumn. In diesen Klassen werden virtuelle Methoden mit dem Namen groupBy und groupBy_sort angelegt.

GroupBy mit Vorsortierunng

Diese Variante des GroupBy-Operators basiert darauf, dass die Spalten bereits sortiert sind, wenn der Operator sie erhält. Falls nach mehreren Spalten gruppiert werden sollen, müssen diese Spalten ebenfalls sortiert werden. Der Sort-Operator (siehe Kapitel 3.3.6) erlaubt es ebenfalls nach mehreren Spalten zu sortieren.

Ablauf Der Algorithmus iteriert über alle Tupel die er erhalten hat. Da die Werte vorsortiert sind, befinden sich die Gruppen ebenfalls sortiert in den Spalten. Der Algorithmus iteriert für jedes Tupel über alle Spalten, die gruppiert werden sollen. Dabei hängt er jeweils die String-Repräsentation der aktuellen Spalte an ein Ergebnis-String. Auf Basis dieses String wird entschieden, ob das aktuelle Tupel in die letzte Gruppe gehört. Falls sich der Ergebnis-String beim nächsten Tupel geändert hat, wird eine neue Gruppe erstellt. Folgender Abschnitt zeigt den Pseudocode, vom GroupBy mit vorsortierten Daten:

Algorithmus 8 : Pseudocode für Sort-GroupBy

```
1 for i < Anzahl der Tupel do
2     for Spalte s aus Spalten do
3         Hole die String-Represäntation des Wertes von s
4         Hänge ihn hinten an einen Ergebnis-String
5     if vorheriger Ergebnis-String == aktueller Ergebnis-String then
6         Aggregiere den Wert des aktuellen Tupels zur alten Gruppe
7     else
8         Lege eine neue Gruppe an
9         Setze den Funktor zurück
```

Beispiel Tabelle 3.3 zeigt ein Beispiel für eine Tabelle, die über zwei Spalten gruppiert werden soll. Damit Algorithmus 8 richtig funktioniert sind die Daten bereits nach Spalte A und Spalte B sortiert. Spalte C soll in diesem Beispiel aggregiert werden. Dazu wird die Summe als Aggregator verwendet. Tabelle 3.4 zeigt das Ergebnis, dass der GroupBy-Operator liefern soll. Die Spalte *Gruppe* wird dabei nicht zurückgegeben und soll lediglich zu einer besseren Übersicht beitragen.

Der Algorithmus arbeitet die Tupel nun der Reihe nach ab. Um den ersten Ergebnis-String zu erhalten, werden Spalte A und Spalte B konkateniert. Die erste Gruppe lautet demnach *AA*. In Zeile 5 wird dieser String nun mit dem vorherigen String verglichen. Da vorher noch keine Gruppe angelegt wurde, ist der vorherige Ergebnis-String leer und es wird eine neue Gruppe angelegt.
Das darauf folgende Tupel bildet ebenfalls *AA* als Ergebnis-String. Nun wird der Wert 1 mit dem Funktor zu dem vorherigen Wert der ersten Gruppe aggregiert. Das dritte Tupel bildet *AC* als Ergebnis-String. Verglichen mit alten String, der immernoch den Wert *AA* hat, ergibt sich nun also ein Unterschied und es wird eine neue Gruppe angelegt. Das Ergebnis wird gespeichert und der Funktor wird zurückgesetzt. Dann wird mit dem nächsten Tupel fortgefahren. Auf diese Weise werden auch alle folgenden Tupel gruppiert.

GroupBy mit Hash

Diese Variante des GroupBy-Operators basiert darauf, dass die Werte anhand einer HashMap in die richtigen Gruppen eingeteilt und aggregiert werden. Hierzu wird die unordered_map aus der C++ Standardbibliothek verwendet.

TID	Spalte A	Spalte B	Spalte C
1	A	A	3
2	A	A	1
3	A	C	4
4	A	C	7
5	B	B	2
6	B	C	3
7	C	A	6
8	C	A	3
9	C	B	4

Tabelle 3.3: GroupBy mit Sortierung Beispiel

TID	Ergebnis	Gruppe
1	4	A A
3	11	A C
5	2	B B
6	3	B C
7	9	C A
9	4	C B

Tabelle 3.4: GroupBy Ergebnis Beispiel

Ablauf Wie schon der vorherige Algorithmus iteriert auch diese Variante des GroupBy-Operators über alle Tupel. Dabei iteriert er ebenfalls für jedes Tupel über alle Spalten, die gruppiert werden sollen und hängt jeweils die String-Repräsentation der aktuellen Spalte an ein Ergebnis-String. Da die Daten nun jedoch nicht sortiert sind, befinden sich die Tupel einer Gruppe nicht mehr direkt hintereinander. Anstatt die Gruppen nacheinader abzuarbeiten, wird nun in der HashMap nachgeschaut, ob der Ergebnis-String bereits enthalten ist (siehe Algorithmus 9 Zeile 5). Falls der Wert bereits vorhanden war, existierte diese Gruppe bereits. Dann wird der Wert dieser Gruppe aus der HashMap geholt, dem Funktor zugewiesen und der neue Wert hinzu aggregiert. Falls der Ergebnis-String noch nicht in der HashMap vorhanden war, wird der Wert des Funktors zurückgesetzt und der neue Wert in die HashMap eingetragen. Auf diese Weise werden alle Gruppen korrekt aggregiert. Folgender Abschnitt zeigt den Pseudocode, der das GroupBy mit Hash beschreibt:

TID	Spalte A	Spalte B	Spalte C
1	B	C	3
2	A	A	1
3	A	A	3
4	A	C	7
5	C	A	6
6	C	B	4
7	B	B	2
8	A	C	4
9	C	A	3

Tabelle 3.5: GroupBy Beispiel

Algorithmus 9 : Pseudocode für Hash-GroupBy

```
1  for i < Anzahl der Tupel do
2      for Spalte s aus Spalten do
3          Hole die String-Represäntation des Wertes von s
4          Hänge ihn hinten an einen Ergebnis-String
5      Schaue in der HashMap nach dem Ergebnis-String
6      if Wert in HashMap then
7          Setze den Wert des Funktors auf den alten Wert der Gruppe
8          Aggregiere den Wert des aktuellen Tupels
9          Trage den neuen Wert in die HashMap ein
10     else
11         Setze den Funktor zurück
12         Trage den neuen Wert in die HashMap ein
```

Beispiel Tabelle 3.5 zeigt ein Beispiel für eine Tabelle, die über zwei Spalten gruppiert werden soll. Die Daten sind in diesem Beispiel nicht sortiert, weshalb der GroupBy-Operator mit der HashMap verwendet werden muss. Das Ergebnis ist jedoch gleich und kann ebenfalls Tabelle 3.4 entnommen werden.

Der Algorithmus bildet für jedes Tupel wieder den Ergebnis-String. Für das erste Tupel ist das in diesem Fall der Wert *BC*. Anstatt den Wert mit dem alten Wert zu vergleichen, wird nun jedoch in der HashMap nachgeschaut. Da der Wert noch nicht vorhanden ist, wird ein neuer Schlüssel angelegt und der erste Wert, in diesem Fall 3, gespeichert. Danach wird der Schlüssel mit dem Wert *AA* angelegt, da dieser ebenfalls noch nicht vorhanden ist. Bei dem dritten Tupel ist der Wert jedoch schon vorhanden. Aus diesem Grund wird der Wert des Funktors auf den alten Wert der Gruppe (hier 1) gesetzt. Danach wird der neue Wert hinzu aggregiert und wieder in der HashMap abgelegt. Dann wird der Algorithmus für den Rest der Tupel ausgeführt.
Auf diese Weise können auch unsortierte Daten gruppiert werden.

Test

Um bei fortlaufendem Fortschritt von SliceDB zu gewährleisten, dass der GroupBy-Operator richtig funktioniert, werden Tests für den Operator geschrieben. Die Tests sollen die Korrekheit des Operators verifizieren und werden für beide Varianten umgesetzt. Getestet werden Kombinationen aus verschiedenen Datentypen und der Anzahl der zu gruppierenden Spalten.
Die Tests für das GroupBy mit HashMap müssen etwas anders implementiert werden, da die HashMap ihre Werte, wenn über sie iteriert wird, in zufälliger Reihenfolge ausgibt. Das führt dazu, dass die Tests die Werte nicht der Reihe nach überprüfen können.

Performanz

Um später einen Überblick über die Performanz und mögliche Verbesserungen zu gewinnen, werden die GroupBy-Operatoren hinsichtlich ihrer Geschwindigkeit vom Bearbeiten der Tupel getestet. Dazu werden Spalten mit zufälligen Werten generiert. Danach wird der `progress_timer` aus der Boost-Bibliothek verwendet um die Ausführungszeit zu bestimmen. Diagramm 3.32 zeigt die Ausführungszeiten des GroupBy-Operators in Abhängigkeit von der Anzahl der Tupel. Um die Werte zu ermitteln wurde der Algorithmus, der die HashMap verwendet, genutzt. Als Eingabe wurde eine unterschiedliche Anzahl zu gruppierender Spalten übergeben. Aggregiert wurde jeweils immer eine Spalte vom Datentyp `double`. Um angepasste Bedingungen zu schaffen, wurden die Spalten, nach denen gruppiert werden soll, so generiert, dass die Anzahl der Gruppen 10% von der Anzahl der Tupel entspricht.

Aus dem Diagramm kann abgelesen werden, dass sich die Laufzeit in Abhängigkeit zur Anzahl der Tupel linear verhält. Pro hinzukommender Spalte wird der Faktor der linearen Funktion um einen festen Wert erhöht.

Fazit & Ausblick

Der GroupBy-Operator ist bisher auf eine einfache Art umgesetzt. Aus diesem Grund ergeben sich einige Punkte die verbessert werden können. Einige davon können und sollen im zweiten Teil der Projektgruppe umgesetzt werden.

Vector-At-A-Time Der Operator unterstützt bisher nur komplette Spalten, die vorher aus den Vektoren zusammengebaut wurden. Dies entspricht dem Operator-At-A-Time Prinzip. SliceDB soll jedoch nach dem Vector-At-A-Time Prinzip arbeiten. Um dies für den GroupBy-Operator zu erreichen, könnte das auf der Sortierung basierende GroupBy modifiziert werden. Die Schwierigkeit liegt dabei darin, dass die Gruppen auch Vektor-Übergreifend vorhanden sein können und nach der Bearbeitung eines Vektors nicht bestimmt werden kann, ob die letzte Gruppe vollständig bearbeitet wurde.

Vergleiche Bisher werden die Einteilungen in die Gruppen über Strings vorgenommen. Bei mehreren zu gruppierenden Spalten werden die Strings einfach

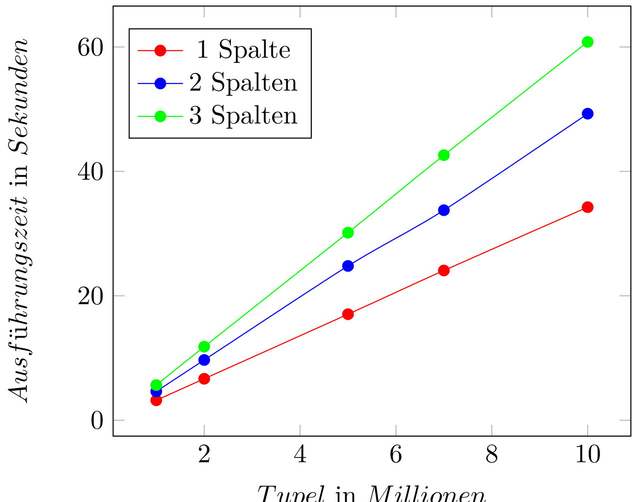


Abbildung 3.32: Ausführungszeiten des GroupBy-Operators mit unterschiedlicher Spaltenanzahl bei 1000 Gruppen auf einem AMD Phenom II x4 955 Prozessor.

hintereinander geschrieben. Um die Gruppen dann zu unterscheiden, werden die Strings miteinander verglichen. Diese Methode wurde verwendet, da sie zunächst am leichtesten zu implementieren war und eine gute Möglichkeit bot, mit verschiedenen Datentypen umzugehen. Eine bessere Variante ist es gleiche Datentypen direkt miteinander zu vergleichen.

Parallelität Parallelität kann zu einer schnelleren Bearbeitung der Tupel führen. Die Daten können auf mehrere Threads aufgeteilt werden. Für das GroupBy, das auf sortierten Daten arbeitet, kann es jedoch schwierig werden Parallelität zu erreichen. Wenn die Daten auf mehrere Threads aufgeteilt werden, die diese Variante ausführen, kann es sein, dass Gruppen mehrfach aufgeführt werden. Beim GroupBy mit der HashMap können die Daten jedoch gut aufgeteilt werden, ohne dass sich daraus Unstimmigkeiten in der Korrekheit ergeben. Dazu muss die HashMap jedoch als gemeinsam genutzte Datenstruktur verwendet werden.

Aggregation Die Aggregation wird bisher mit Funktoren vorgenommen. Diese speichern das aktuelle Ergebnis der Gruppe und aggregieren neue Werte hinzu. Die Aggregation findet direkt, nachdem die Tupel einer Gruppe zugewiesen werden, statt. Eine weitere Technik aggregiert die Werte erst, wenn die Gruppen schon feststehen. Dazu werden lediglich die TIDs der Tupel in Gruppen gesammelt und an einen optimierten Aggregations-Operator weitergeleitet.
Es ist im vorhinein nicht klar, welche der beiden Varianten die performantere Wahl ist. Aus diesem Grund wird letztere Technik ebenfalls implementiert. Anschließend wird mit dem Performanz-Test bestimmt welche Version scheller mit der Bearbeitung der Tupel fertig ist.

GroupBy-Verbesserungen

Überblick Im folgenden werden die Verbesserungen für den GroupBy-Operator beschrieben. Dabei wurde der auf Hashing basierende Operator optimiert. Der auf Sortierung basierende Operator wurde nicht weiter entwickelt, da es im SSB-Benchmark keine Anfrage gibt, die die Daten vor einer GroupBy Operation sortiert. Für die meisten Änderungen sind jeweils neue GroupBy-Methoden angelegt worden, sodass der Performanz-Gewinn nachgemessen werden kann. Im vorherigen Kapitel 12 wurden bereits einige Möglichkeiten zur Verbesserung des Operators beschrieben. Diese wurden zum größten Teil implementiert. Zunächst werden einige kleinere Änderungen an dem Operator beschrieben. Danach wird erläutert, wie der Operator parallelisiert wurde und die Performanz durch eine eigene HashMap nochmals gesteigert werden konnte. Zum Schluss werden die Endergebnisse mit Diagrammen aufgearbeitet und ein Ausblick für weitere Optimierungen gegeben.

Verbessertes Bilden der Hash-Keys Nach einigen Zeitmessungen viel auf, dass das Bilden der Schlüssel für die HashMap sehr lange dauert. Bisher wurde die String-Repräsentation für jede Spalte, die gruppiert werden soll, über einen `stringstream` aus der Spalte geholt. Der Schlüssel wurde dann mit einem `ostringstream` gebildet, indem die Strings jeder Spalte hintereinander gehängt wurden. Die Streams sind sehr langsam. Außerdem wird die String-Repräsentation für den Groupby-Operator nicht benötigt.

Um nicht länger von den Streams abhängig zu sein, wurde eine neue Methode für die Spalten-Klassen hinzugefügt. Diese gibt die Länge des Datentyps der Spalte und, mittels cast, einen Zeiger auf ein `char` Array des gewünschten Wertes zurück. Von allen Spalten einer Reihe, die gruppiert werden sollen, werden dann die Zeiger auf `char` Arrays abgerufen. Eine Hash-Funktion bildet aus diesen Arrays dann einen Schlüssel. Dieser wird dann in der HashMap verwendet.

Verbesserung der Aggregation In dem ersten Teil der Projektgruppe wurde eine Klasse namens `Aggregator` entwickelt, die das Aggregieren der Werte übernimmt. Diese Klasse wurde von dem GroupBy-Operator auf falsche Art und Weise verwendet. In der ersten Version wurden die Zwischenergebnisse in der `HashMap` gespeichert. Es wurde lediglich eine Instanz der `Aggregator` Klasse erstellt. Für jedes eingehende Tupel wurde das Zwischenergebnis dann in diese Klasse geladen und der aktuelle Wert aggregiert. Aufgrund der Zugriffe auf den Arbeitsspeicher ist dieses Verfahren natürlich sehr langsam.

In der neuen Version wird für jede gefundene Gruppe ein neuer `Aggregator` auf dem Heap erstellt. Danach wird ein Pointer auf den `Aggregator` in der `HashMap` gespeichert. Falls nun ein neuer Wert zu einer bestehenden Gruppe aggregiert werden soll, liegt das Zwischenergebnis bereits im Funktor. Da die `HashMap` den Pointer auf den `Aggregator` zurückgibt, kann der aktuelle Wert direkt aggregiert werden.

Paralleles GroupBy

Als nächstes wurde der GroupBy-Operator so angepasst, dass die Tupel parallel auf mehreren CPUs bearbeitet werden können. Folgende Punkte wurden dabei umgesetzt:

- In der GroupBy-Klasse wurde dazu eine HashMap vom Typ `unordered_map` aus der Standard-Bibliothek global angelegt, sodass mehrere Threads auf die Datenstruktur zugreifen können.
- Zur Parallelisierung wurden die Threads der Boost-Bibliothek verwendet. Die `unordered_map` kann ohne Synchronisation nur für lesende Zugriffe parallel genutzt werden. Parallele schreibende Zugriffe können zu *race conditions* führen und müssen daher mit einem `Mutex` ausgeschlossen werden. Dazu wurde ebenfalls die aus der *Boost* Bibliothek stammende Implementierung gewählt. Das Aggregieren in der `Aggregator` Klasse ist keine atomare Operation. Daher muss dieser Teil des Codes auch mit einem `Mutex` versehen werden.
- Es wurde eine neue Funktion erstellt, die zusätzlich zu den Parametern des alten GroupBy-Operators einen weiteren Parameter übergeben bekommt. Dieser legt die Anzahl der Threads fest, die von dem Operator genutzt werden. In der Standard-Bibliothek gibt es die Funktion `thread::hardware_concurrency` mit der die Anzahl der maximal verfügbaren Threads festgestellt werden kann. Diese wird in der `Executor` Klasse aufgerufen und an den GroupBy-Operator weitergegeben.

Listing 10 zeigt den Algorithmus für den parallelen Groupy-Operator. Dieser ist sehr übersichtlich geworden, da die eigentliche Arbeit in der Methode `groupByThread` (siehe 11) passiert. Der Algorithmus erstellt im ersten Schritt die Threads, die jeweils die `groupByThread` Methode ausführen. Der Methode wird dabei die gesamte Anzahl der Threads und die jeweilige Thread-Nummer als Parameter übergeben. Jeder Thread arbeitet auf der gleichen, global definierten `unordered_map` aus der `GroupBy` Klasse. Sobald alle Threads ihre Arbeit beendet haben, wird das Ergebnis aus der `unordered_map` geholt und in einer neuen Spalte an den `Executor` zurückgegeben.

Algorithmus 10 : Pseudocode für paralleles Hash-GroupBy

1 **for** $i <$ *Anzahl der Threads* **do**
2 ⎣ Starte Methode `groupByThread` mit Parameter i in Thread i
3 Warte bis alle Threads fertig sind
4 **for** $i <$ *Anzahl der Gruppen* **do**
5 ⎣ Füge der Ergebnis-Spalte das Ergebnis und die TID der Gruppe i hinzu

Listing 11 zeigt den Algorithmus, den die Threads des parallelen GroupBy-Operators ausführen. Die ersten Zeilen sorgen dafür, dass die Threads jeweils auf unterschiedlichen Tupeln arbeiten. Dazu werden der Methode die zwei Parameter `thread` und `thread_count` übergeben. `thread` enthält die Zahl des aktuellen Threads, `thread_count` enthält die Gesamtzahl an Threads. Ab Zeile 6 werden die Tupel dann in einer For-Schleife gruppiert. Dazu wird der Hash-Key über

char Arrays, wie im obigen Abschnitt beschrieben, gebildet. Als nächstes wird in der `HashMap` nach dem Schlüssel gesucht. Falls dieser noch nicht vorhanden ist, wird auf dem Heap ein neuer `Aggregator` angelegt, initialisiert und ein Pointer auf den neuen `Aggregator` in der `HashMap` gespeichert. Ebenfalls neu ist, dass die TID des aktuellen Tupels in der `Aggregator` Klasse abgelegt wird. Auf diese Weise muss keine zweite `HashMap` für die TIDs verwendet werden.

Algorithmus 11 : Pseudocode für paralleles Hash-GroupBy

```
1  slice_size = tupel_count / thread_count
2  start = thread · slice_size
3  end = start + slice_size
4  if thread == thread_count then
5      end = tupel_count
6  for i = start ; i < end ; i++ do
7      key = Bilde den Hash-Key für die aktuelle Zeile
8      _mutex.lock()
9      entry = unordered_map.find(key)
10     if entry == NULL then
11         Erstelle den Aggregator
12         Speichere die TID des aktuellen Tupels in dem Aggregator
13         Aggregiere den Wert der aktuellen Zeile
14         Füge einen Pointer auf den Aggregator in die HashMap ein
15     else
16         Aggregiere den Wert der aktuellen Zeile zum Aggregator aus entry
17     _mutex.unlock()
```

Diese Implementierung bringt zwar schon einen Performanz-Gewinn, scheitert jedoch daran, dass zu viele Locks gesetzt werden müssen, wenn es mehrere Threads gibt.

Locks minimieren Um die Locks zu reduzieren, bekommt jeder Thread seine eigene HashMap. Dazu wird in der `GroupBy` Klasse global ein Pointer auf ein Array von `unordered_maps` angelegt. Das Array wird nun in dem Konstruktor der `GroupBy` Klasse initialisiert. Dem Konstruktor wird die Anzahl der Threads übergeben und er erstellt, der Anzahl entsprechend viele, $HashMaps$. So wird gewährleistet, dass weder zu viele noch zu wenig $HashMaps$ erstellt werden. Da jeder Thread seine Zahl übergeben bekommt, kann diese genutzt werden um die richtige $HashMap$ zu nutzen.

Nachdem alle Threads mit der Bearbeitung fertig sind, muss das Endergebnis in einer `Merge` Phase berechnet werden. Algorithmus 12 zeigt den Ablauf der `Merge` Phase, die direkt, nachdem alle Threads ihre Arbeit beendet haben, beginnt. Die Idee dahinter ist es, alle HashMaps der anderen Threads in die HashMap des ersten Threads zu verschmelzen. Dazu werden die Ergebnisse (Keys und `Aggregator`) aus den anderen Threads geholt. Für jeden Eintrag aus dem Ergebnis wird dann in der HashMap des ersten Threads geschaut, ob es die Gruppe (hier der Key) schon gibt. Falls es sie gibt, werden die Ergebnisse

der beiden Threads aggregiert. Falls es sie nicht gibt, wird die Gruppe in die HashMap des ersten Threads eingefügt.

Algorithmus 12 : Pseudocode für die `Merge` Phase

```
1 for i = 1 ; i < thread_count ; i++ do
2     thread_result_groups = hash_maps[i].getResultVector()
3     for result : thread_result_groups do
4         first_thread_entry = hash_maps[0].findEntry(result.key)
5         if first_thread_entry != NULL then
6             first_thread_entry.aggregate(result.getResult())
7         else
8             hash_maps[0].insert(result.key, result.getResult())
```

HashMap

Nun liegt der Flaschenhals in der Implementierung der `unordered_map` der Standard-Bibliothek. Ein Wechsel zu der Implementierung der `unordered_map` von `Boost` verschlechterte die Performanz sogar noch etwas. Aus diesem Grund wurde nun eine eigene Implementierung einer HashMap erstellt.

Wie bei dem GroupBy-Operator wurde auch die HashMap zunächst mit einer einfachen Implementierung erstellt. Diese implementiert eine HashMap, bei der Kollisionen als einfach verkettete Liste verwaltet werden. Als Schlüssel kann bisher nur ein Datentyp gewählt werden, der Zahlen darstellt. Der assoziierte Wert hingegen kann über die Nuzung von Templates variabel gewählt werden. Zur Ablage der Schlüssel und ihrer Werte wurde ein `struct` erstellt. Dieses enthält ebenfalls einen Verweis auf kollidierende Einträge. Soll ein neuer Wert eingefügt werden, wird ein neues `struct` erstellt und in einem `Vector` gespeichert. Ein zweiter `Vector` speichert an der Stelle des Keys einen Verweis auf das `struct`. Überschreitet die Anzahl durch Kollisionen angehängter Einträge eine Konstante, wird eine Methode zur Vergrößerung der HashMap aufgerufen. Diese vergrößert den `Vector` und fügt alle bisher eingefügten Elemente noch einmal ein.

Mit dieser einfach HashMap konnte die Geschwindigkeit des GroupBy-Operators nochmal verbessert werden.

Hash-Funktionen Die Auswahl der Eigenschaften der Funktion, die den Hash-Key erstellt, hat großen Einfluss auf die Performanz des Operators. Es wurden Versuche mit dem Fowler-Noll-Vo (kurz: FNV) Verfahren sowie mit dem Murmur-Hash und verschiedenen Längen der Keys unternommen. Die besten Resultate wurden dabei mit dem FNV Verfahren und `uint` als Länge für die Keys erreicht. Die Funktionen wurden außerdem mit dem `inline` Keyword versehen.

Zeitmessungen & Auswahl der Optimierungen

Für den GroupBy-Operator wurden weitere Anpassungen für die Zeitmessungen und Auswahl der Optimierungen vorgenommen. Die Zeitmessungen werden in

der `Executor` Klasse, unter der Bedingung, dass ein Benchmark-Flag gesetzt wurde, aufgerufen. Die Optimierungen werden in der `Executor` Klasse mittels Präprozessor-Anweisungen ein- und ausgeschaltet. Um die Optimierungen an einer zentralen Stelle aktivieren zu können, wurden die entsprechenden Präprozessor Variablen in dem `util/optimizations.h` Header definiert. Folgende Varianten können ausgewählt werden:

- Nicht optimierter GroupBy-Operator
- Optimierter Groupy-Operator, der alle Verbesserungen bis auf die selbst implementierte `HashMap` enthält
- Optimierter Groupy-Operator, einschließlich der neuen `HashMap`

Fazit

Die letzte Version des GroupBy-Operators wurde auf dem Insengard Server getestet. Diagramm 3.33 zeigt die gemessenen Werte bei 1000 Gruppen und bei einer verschiedenen Anzahl an Threads. Die gemessenen Ausführungszeiten verbessern sich bis zu einer Anzahl von 8 Threads stark. Eine größere Anzahl verbessert die Zeiten jedoch nicht mehr viel.

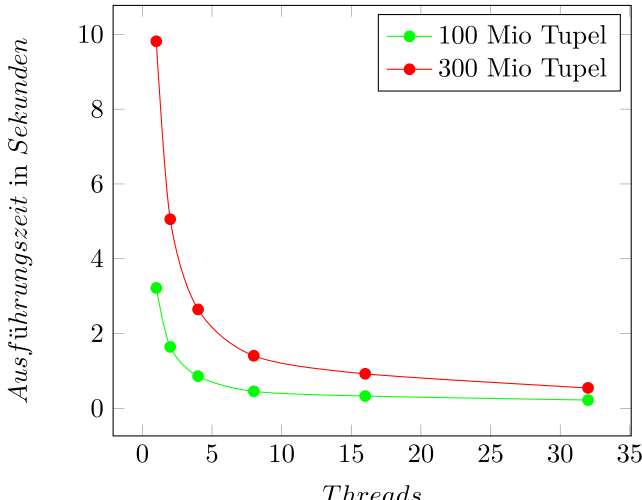


Abbildung 3.33: Ausführungszeiten des GroupBy-Operators mit unterschiedlicher Anzahl an Threads bei 1000 Gruppen auf einer Dual Socket Intel Xeon CPU E5-2690 (16 Cores@2.90GHz), 64GB DRAM.

Diagramm 3.34 zeigt die gemessenen Ausführungszeiten für die SSB Abfragen mit Skalierungsfaktor 4 vor und nach den Optimierungen. Die SSB Anfragen, die den GroupBy-Operator mit einer größeren Tupel-Anzahl belasten, wurden stark beschleunigt. Abfragen mit kleinerer Tupel-Anzahl haben sich hingegen nur leicht verbessert. Eine Ausnahme bildet dabei die Abfrage 3_4 die sogar etwas langsamer läuft. Diese Abfrage enthält für den GroupBy-Operator wahrscheinlich nur sehr wenige Tupel, die aggregiert werden müssen. Die Laufzeit besteht also hauptsächlich aus dem Reservieren von Speicher für die HashMaps.

Die SSB Anfragen, die nicht sehr stark beschleunigt wurden, können noch über den Optimierer verbessert werden. Dazu könnte dem GroupBy-Operator eine geschätzte Anzahl an Gruppen übergeben werden. Auf diese Weise könnte die unnötige Reservierung von Speicher minimiert werden. Außerdem kann die Anzahl der Threads bei einer geringen Anzahl an Tupeln herunter gesetzt werden.

Anfrage 3_1 erhält als Eingabe die größte Anzahl an Tupel und hat sich am stärksten verbessert. So ist der GroupBy-Operator in bestimmten Fällen um fast 80 Mal schneller geworden.

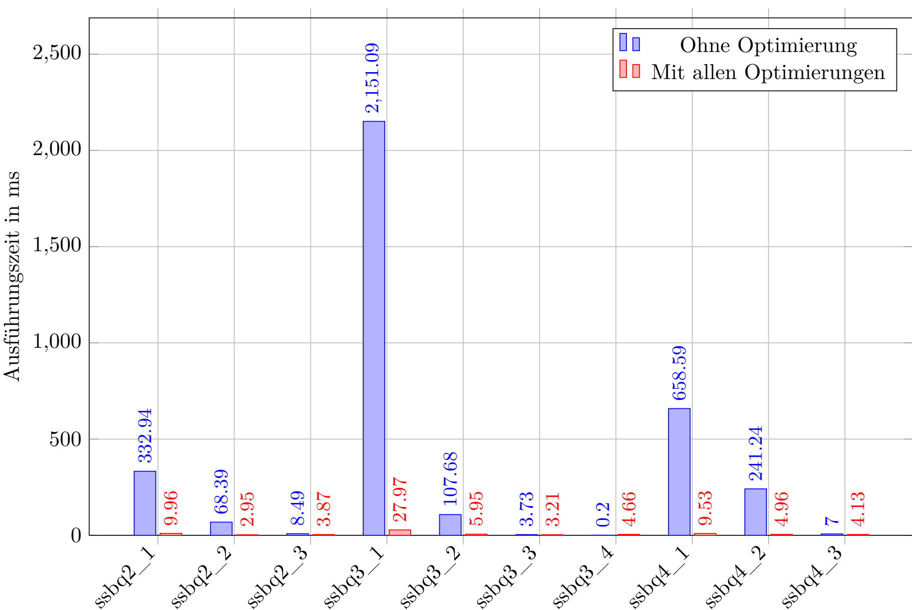


Abbildung 3.34: Ausführungszeiten der Anfragen des Star Schema Benchmarks für den GroupBy-Operator mit Skalierungsfaktor 4, zum einen ohne Optimierungen und zum anderen mit Parallelisierung und allen weiteren Optimierungen. (Hardware: Dual Socket Intel Xeon CPU E5-2690 (16 Cores@2.90GHz), 64GB DRAM.)

Ausblick

Es können immernoch einige Verbesserungen vorgenommen werden.

Slice als Eingabe Der GroupBy-Operator könnte so erweitert werden, dass er die Eingabe von mehreren Stücken unterstützt. So kann der Operator bereits auf Teilergebnissen des vorherigen Operators arbeiten und damit zeitlich früher die Arbeit beginnen. Gerade für den auf Hashing basierenden Operator sollten die Änderungen einen geringen Umfang haben. Um die Änderungen durchzuführen sollten ebenfalls Absprachen mit dem Planinterface Team getroffen werden.

GroupBy mit Sortierung Auch wenn der auf Sortierung basierende GroupBy-Operator nicht direkt von vorsortierten Daten profitieren kann, könnte er trotzdem schnell sein. Außerdem ergibt sich bei dessen Nutzung der Vorteil, dass Ergebnisse schon vorher direkt an den nächsten Operator weitergegeben werden können (siehe Kapitel 12).

Hashing Es können noch weitere Hashing Methoden implementiert werden. Für den jetzigen Ansatz eignet sich vermutlich ein auf *open addressing* basierendes Verfahren am besten, da es das schnellste Verfahren zu sein scheint. Soll die Performanz für viele Gruppen erhöht werden, sollte die jetzige `Merge` Phase wieder entfernt werden. Stattdessen kann dann eine `HashMap` implementiert werden, die Parallelität unterstützt. Zum Beispiel könnte ein parallel nutzbares `Cuckoo-Hashing` implementiert werden. [40]

Optimierer Anfragen mit kleinerer Tupel Anzahl könnten über den Optimierer verbessert werden. Dieser sorgt dafür, dass zu Beginn weniger Speicher in der HashMap reserviert wird. Außerdem könnte die Anzahl der Threads verringert werden.

OpenMP Der GroupBy-Operator könnte ebenfalls mit OpenMP anstatt der Boost Threads umgesetzt werden. Dafür könnten alle Gruppen zunächst in einer Build Phase in eine HashMap eingefügt werden. Danach können die Gruppen in einer for Schleife und mit OpenMP aggregiert werden. Dazu müssen jedoch wieder Locks beim Aggregieren verwendet werden.

3.3.8 Invisible Join *(Stefan Noll)*

Während des ersten Semesters der Projektgruppe wurde ein Großteil der Zeit darauf verwendet das Datenbanksystem zunächst funktionell vollständig zu implementieren. Der Fokus lag insbesondere auf der korrekten Ausführung der Anfragen des *Star Schema Benchmark* (SSB). Aus diesem Grund wurden nur selten effiziente Algorithmen für die unterschiedlichen Operationen implementiert. Im Rahmen des zweiten Semesters der Projektgruppe haben wir uns daher mit möglichen Optimierungen für das selbst entwickelte Datenbankmanagementsystem *SliceDB* beschäftigt. Ziel dieser Optimierungen ist eine deutliche Reduzierung der Ausführungszeiten der Datenbankanfragen des SSB.
Einen möglichen Ansatz zur Laufzeitoptimierung bietet das Schema bzw. Datenlayout des Star Schema Benchmark selbst. Dieses setzt sich im Kern aus einer Faktentabelle und mehreren Dimensionstabellen zusammen. Alle Anfragen führen dabei klassischerweise einen oder mehrere Joins jeweils immer mit der Faktentabelle und einer der Dimensionstabellen aus. Andere Formen von Joins werden hingegen nicht verwendet. Als weitere Besonderheit sind (fast) alle der Primärschlüssel der Dimensionstabellen aus dem Wertebereich 1 bis N sowie aufsteigend sortiert. Die einzige Ausnahme bildet die Dimensionstabelle `date`. Hier ist der Primärschlüssel eine Zahl, die ein Datum repräsentiert und sich aus der Verknüpfung von Jahr, Monat und Tag zusammen setzt.
Die oben beschriebenen Eigenschaften der verwendeten Daten können mit speziellen Joinalgorithmen wie dem *Invisible Join* ausgenutzt werden. Dieser wurde zur möglichen Optimierung der Ausführungszeit der Anfragen in SliceDB implementiert und anschließend evaluiert. Im Folgenden wird zunächst kurz der Algorithmus vorgestellt, danach auf die konkrete Implementierung eingegangen und anschließend eine Evaluation des implementierten Algorithmus vorgestellt.

Invisible Join

Der sogenannte *Invisible Join* Algorithmus von [3] wurde speziell für Anfragen auf Daten, die im Star-Schema vorliegen, entwickelt. Während mit herkömmlichen Joinalgorithmen, die zwei Spalten bzw. Tabellen miteinander joinen, oftmals eine Kette von Joins notwendig ist, werden beim Invisible Join alle Join-Operationen in einem Algorithmus bzw. Operator gebündelt. Der Algorithmus selbst lässt sich in drei Phasen unterteilen und geht dabei grob wie folgt vor:

1. Zunächst werden alle am Join beteiligten Dimensionstabellen getrennt betrachtet. So werden zunächst für jede der Dimensionstabellen die zugehörigen Filterprädikate ausgewertet. Dabei wird für jedes Tupel einer Dimensionstabelle, welches das Filterprädikat erfüllt, der Primärschlüssel in eine Hashtabelle eingefügt.

2. Anschließend werden die Fremdschlüssel der Faktentabelle betrachtet. Hierbei wird für jeden Fremdschlüssel einer Fremdschlüsselspalte geprüft, ob dieser auf einen Primärschlüssel der Dimensionstabelle, der das Filterprädikat erfüllt, verweist. Aus diesem Grund wird bei jedem Fremdschlüssel nachgeschaut, ob sich dieser in der zuvor erstellten Hashtabelle befindet. Diese Information wird wiederum zur Konstruktion einer Positionsliste genutzt, welche die Positionen der Fremdschlüssel in der Faktentabelle

auflistet, die in der Hashtabelle gefunden wurden. Als Ergebnis entsteht also für jede Fremdschlüsselspalte der Faktentabelle eine Positionsliste.

3. Zum Schluss wird von allen Positionslisten, die in der vorherigen Phase des Algorithmus entstanden sind, die Schnittmenge gebildet. Das Resultat ist eine Liste von Positionen der Faktentabelle, die die Position jener Fremdschlüssel angeben, welche über alle Dimensionstabellen betrachtet die Filterprädikate erfüllen. Mit Hilfe dieser Positionsliste werden zu guter Letzt auch die Positionen der Tupel der Dimensionstabellen ermittelt, die ebenfalls einen Teil des Ergebnisses des Joins darstellen.

Implementierung

Der vorgestellte Algorithmus lässt sich im Detail auf unterschiedliche Art und Weise implementieren. So können beispielsweise unterschiedliche Datenstrukturen verwendet werden, die je nach Anfrage die Laufzeit des Algorithmus stark beeinflussen. Im Folgenden soll daher die konkrete Implementierung in *SliceDB* vorgestellt werden und mögliche Entscheidungen im Bezug auf die Implementierung erläutert werden.
Die im Algorithmus benutzten Positionslisten können auf zwei verschiedene Arten implementiert werden. Zum einen können sie, wie der Name andeutet, in Form einer verketten Liste oder eines Arrays von Integern umgesetzt werden. Zum anderen können die Positionsinformationen auch durch eine Bitmap abgespeichert werden.
Bei einer *Bitmap* handelt es sich um eine kompakte Datenstruktur, die eine bestimmte Anzahl von Bits verwaltet. So können in einer Bitmap der Größe N beispielsweise die Positionen von 0 bis $N - 1$ einfach dadurch gespeichert werden, dass das i-te Bit der Bitmap auf 1 gesetzt wird, falls die i-te Position gespeichert werden soll. Ansonsten wird das i-te Bit auf 0 gesetzt. Der Vorteil bei der Verwendung von Bitmaps liegt im Wesentlichen in der Möglichkeit die Schnittmenge mehrerer Bitmaps effizient zu berechnen. Dies kann einfach durch bitweises verunden der Datenstruktur implementiert werden und mit modernen Prozessoren für eine feste Breite von beispielsweise 256 Bit durch eine SIMD Maschineninstruktion direkt ausgeführt werden. Ein weiterer Vorteil von Bitmaps ergiebt sich aus der kompakten Darstellung im Speicher. So können selbst Bitmaps mit einer Größe von 10^7 ($\approx$ 1.25MB) noch problemlos im L3 Cache des Prozessors gehalten werden.
Allerdings birgt die Verwendung von Bitmaps auch einen möglichen Nachteil bei Anfragen mit Filterprädikaten sehr hoher Selektivität. In solch einem Fall würden nur wenige Tupel der Tabelle das Prädikat erfüllen, wodurch nur wenige Position abgespeichert werden müssten. Eine Bitmap würde hier sehr viele Nullen enthalten und so letztendlich eher Speicherplatz verschwenden. Eine verkette Liste von Integern würde hingegen nur die tatsächlich benötigten Positionen als Zahlen enthalten. Dafür ist die Berechnung der Schnittmenge von mehreren Positionslisten verhältnismäßig rechenaufwändig – in jedem Fall deutlich aufwändiger als bei Bitmaps.
Prinzipell wäre es durchaus denkbar beide Datenstrukturen zu unterstützen und je nach Selektivität der gesamten Anfrage entweder Bitmaps oder verkettete Listen zu benutzen. Der Einfachheit halber wurden jedoch nur Bitmaps für die Implementierung des *Invisible Join* verwendet. Bei genauerer Betrachtung der

Selektivitäten der Filterprädikate des Star Schema Benchmark, fällt außerdem auf, dass diese im Durchschnitt nicht so hoch sind, als dass sich ausschließlich die Verwendung von verketten Listen lohnen würde.
Angenommen ein Integer wäre 64 Bit breit, dann wäre eine Bitmap von N Positionen in etwa so groß wie ein Array von N Elementen, wenn die Selektivität $1/64 \approx 0.016$ beträgt. Bei einer verketten Liste, die zusätzlich noch einen Zeiger für jedes Element enthält, müsste die Selektivität erwartungsgemäß noch kleiner sein, damit sich eine Nutzung im Bezug auf den Speicherbedarf lohnt. Außerdem basiert die bisherige Implementierung von Filteroperationen in *SliceDB* schon auf Positionslisten, die als Arrays gespeichert werden. Insgesamt lohnt sich also eher die Betrachtung von Bitmaps für die Implementierung des *Invisible Join*.
Um den Algorithmus möglichst einfach zu parallelisieren wurde darüber hinaus die Programmierschnittstelle *OpenMP* [52] eingesetzt. Dadurch können die ersten beiden Phasen des Algorithmus für jede Dimensionstabelle mit wenig Aufwand parallel ausgeführt werden. So werden zunächst jeweils die Hashtabellen unabhängig voneinander konstruiert und anschließend die Positionslisten generiert. Die Berechnung der Schnittmenge geschieht hingegen nicht parallel. Dafür wäre eine Unterstützung für selbst definierte Reduktionen auf selbst definierten Datentypen in OpenMP notwendig, was derzeit (noch) nicht möglich ist.
Die einfache Parallelisierung mit OpenMP, wie oben beschrieben, besitzt allerdings auch einen wesentlichen Nachteil. Da die maximale Anzahl an parallel arbeitenden Threads direkt der Anzahl am Join beteiligten Dimensionstabellen entspricht, skaliert der parallelisierte Algorithmus schlecht bzw. gar nicht. Diese und weitere Eigenschaften sollen im Rahmen der Evaluation der Implementierung in Kapitel 3.3.8 näher untersucht werden.

Evaluation

Zur Evaluation der Implementierung des Invisible Join wurde neben dem Algorithmus auch ein entsprechender Knoten (Operator) für die Ausführung im Anfrageplan implementiert. Anschließend wurden die Anfragen Q2.1 bis Q4.3 des Star Schema Benchmarks einmal durch Nutzung des neuen Operator und einmal durch Nutzung einer Folge von Hash-Joins ausgeführt. Eine Evaluation der Anfragen Q1.□ wurde nicht durchgeführt. Da bei diesen Anfragen nur ein Join ausgewertet wird, erscheint eine Anwendung des Invisible Join hier weniger sinnvoll. Zusätzlich müsste die Implementierung aufwändig um einige Sonderfälle erweitert werden, weil bei den Anfragen Q1.□ auch Selektionen auf der Faktentabelle durchgeführt werden. Die Ergebnisse der Experimente sind in Abbildung 3.35 aufgeführt. Dabei wurden die Messungen für die Skalierungsfaktoren 1 (Abbildung 3.35a)), 4 (Abbildung 3.35b)), 10 (Abbildung 3.35c)) und 30 (Abbildung 3.35d)) durchgeführt.
Im Vergleich zur Beantwortung der Anfragen im klassischen Fall durch eine Folge von Hash-Joins, werden einige Anfragen bei der Nutzung des neuen Invisible Join Operators bis zu 3× so schnell ausgeführt. Bei anderen Anfragen kann es hingegen auch passieren, dass Anfragen maximal 2× langsamer ausgeführt werden. Die tatsächliche Verbesserung/Verschlechterung der Ausführungszeit variiert je nach Anfrage zum Teil sehr stark. So lässt sich beispielsweise die deutlichste Verbesserung der Laufzeit bei Anfrage Q2.1 erkennen. Die deutlichste Verschlechterung ergibt sich hingegen bei Anfrage Q3.3. Außerdem lässt sich beobachten, dass die Größe der Datenbank, hier durch die Größe des Skalie-

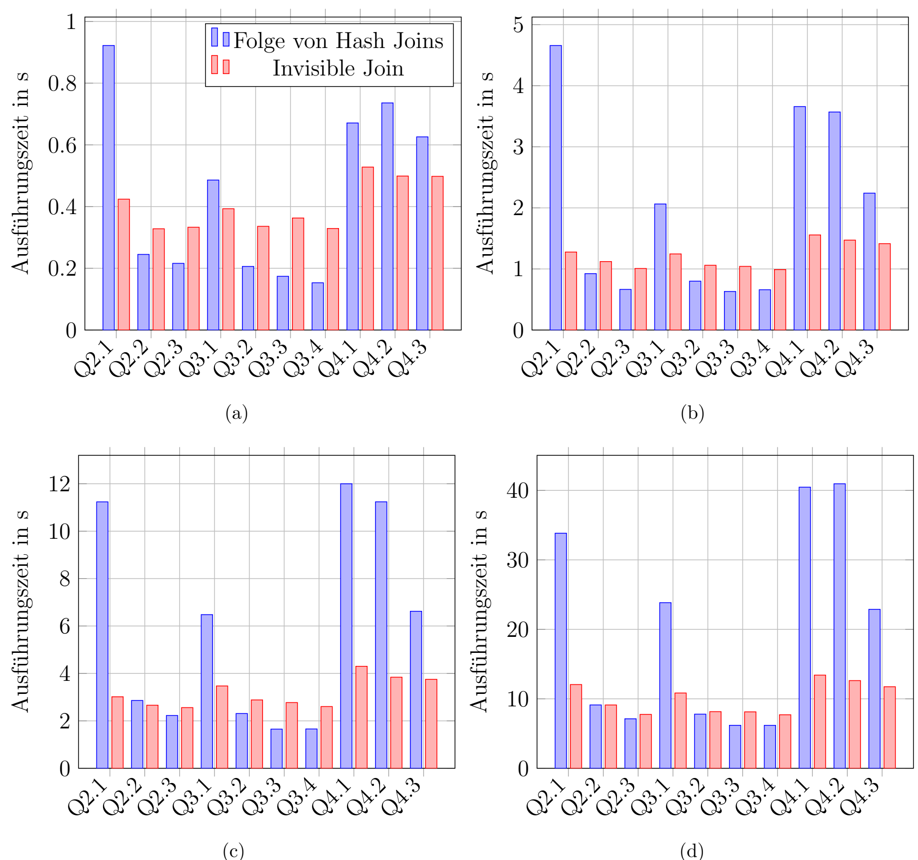


Abbildung 3.35: Ausführungszeiten der Anfragen Q2.1 bis Q4.3 des Star Schema Benchmark mit den Skalierungsfaktoren 1 (a), 4 (b), 10 (c) und 30 (d). Berechnung der Joins zum einen durch eine Folge von Hash-Joins auf jeweils zwei Spalten und zum anderen durch den Invisible Join. Dargestellt ist jeweils der Mittelwert von 50 Ausführungen. Hardware: Dual Socket Intel Xeon CPU E5-2690 (16 Cores@2.90GHz), 64GB DRAM.

rungsfaktors für die Datengenerierung ausgedrückt, einen großen Einfluss auf die relative Verbesserung der Ausführungszeit hat. Dies wird insbesondere bei den Anfragen Q4.1, Q4.2 und Q4.3 deutlich. Während die Verbesserung der Laufzeit bei einem Sklarierungsfaktor von 1 relativ marginal ausfällt, lässt sich ein erheblicher Unterschied bei einem Skalierungsfaktor von 30 erkennen. Bei größeren Skalierungsfaktoren schrumpft darüber hinaus auch der Laufzeitunterschied der Anfragen Q3.□. Dadurch ist der Invisible Join im Vergleich zu den herkömmlichen Hash-Joins bei diesen Anfragen zwar immer noch langsamer, aber im Durchschnitt über alle Anfrage betrachtet schneller.

Außerdem lässt sich ebenfalls beobachten, dass die Ausführungszeiten der Anfragen, die mit Hilfe des Invisible Join beantwortet wurden (rot), deutlich weniger variieren als die Ausführungszeit der Anfragen bei Nutzung der Hash-Joins (blau). Dies ist über alle aufgeführten Skalierungsfaktoren hinweg der Fall.

Insgesamt kann die Implemetierung des Invisible Join also die Laufzeit einiger Anfragen zum Teil verbessern. Doch wird nicht in allen Fällen eine Verbesserung der Gesamtlaufzeit erreicht. In wenigen Fällen wird die Ausführungszeit sogar leicht verschlechtert. Doch wodurch ist der Laufzeitunterschied zu erklären? Der Invisible Join ersetzt im Anfrageplan nicht nur die gesamten Join Operatoren, sondern auch die zuvor notwendigen Materialisierungs- sowie Filteroperationen. Alle Filteroperationen operieren dabei jeweils auf der Originaltabelle. Materialisierungsoperationen vor oder nach den Filteroperationen werden nicht durchgeführt. Außerdem erfolgt die Ausführung des Invisible Join parallel. Denkbar wäre also, dass vor allem die Parallelisierung des Algorithmus zur schnellen Ausführung der Anfragen führt.
Um diese Möglichkeit genauer zu untersuchen, wurden weitere Experimente durchgeführt. Die Ergebnisse sind in Abbildung 3.36 aufgeführt. Für die Messungen wurde jeweils vor der Ausführung von SliceDB die Umgebungsvariable `OMP_NUM_THREADS` unter Linux angepasst, um die maximal mögliche Anzahl an Threads für OpenMP zu konfigurieren. So ist es insbesondere möglich, die sequentielle Ausführung dadurch zu simulieren, dass die maximale Anzahl von Threads auf eins herunter gesetzt wird.

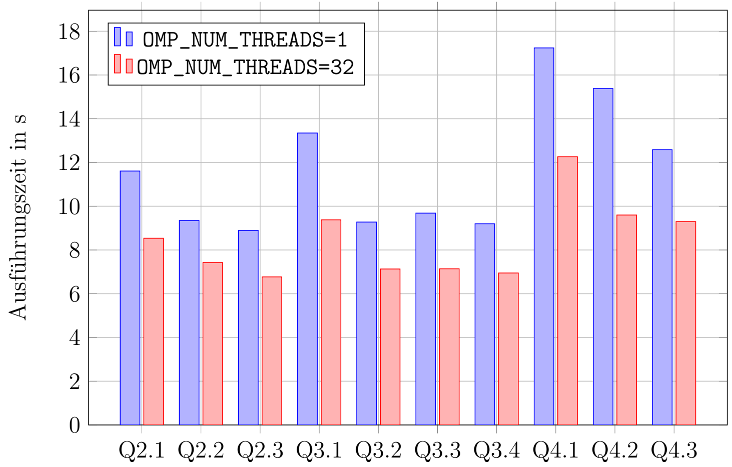


Abbildung 3.36: Ausführungszeiten der Anfragen Q2.1 bis Q4.3 des Star Schema Benchmark mit Skalierungsfaktor 30. Die Umgebungsvariable OMP_NUM_THREADS wurde einmal auf 1 und ein anderes mal auf 32 gesetzt. Dargestellt ist jeweils der Mittelwert von 50 Ausführungen. Hardware: Dual Socket Intel Xeon CPU E5-2690 (16 Cores@2.90GHz), 64GB DRAM.

Die Ergebnisse zeigen, dass die Parallelisierung des Invisible Join mit OpenMP im Durchschnitt zu einer geringen Verbesserung der Gesamtlaufzeit der Anfragen führt. Insbesondere bei Berücksichtigung der verfügbaren Hardwareressourcen, in diesem Fall 32 mögliche Threads, fällt der Unterschied doch verhältnismäßig gering aus. Zum einen werden, wie anfangs erwähnt, offensichtlich nicht alle 32 verfügbaren Threads genutzt, da für jede Dimensionsspalte nur ein Thread gestartet wird. Dies würde beispielsweise bei den Anfragen Q4.□ eine Ausführung mit vier Threads bedeuten. Zum anderen scheint die Parallelisierung bei den Anfragen Q4.□ mehr zu bewirken als bei den Anfragen Q2.□.

3.3.9 Bloom-Filter *(Jan Stallmann)*

Bei der Betrachtung der Ausführungszeiten der verschiedenen Operatoren, hat typischerweise die des Join-Operators einen großen Anteil an der Gesamtausführungszeit einer Anfrage. Die Ausführungszeit des Join-Operators lässt sich reduzieren, indem die Anzahl der Tupel, die dieser verarbeiten muss, reduziert wird. Eine mögliche Datenstruktur, mit der die Tupel effizient vorgefiltert werden können, sind Bloom-Filter.
Ein Bloom-Filter ist eine probabilistische Datenstruktur, mit der schnell bestimmt werden kann, ob ein Element in einer gegeben Menge enthalten ist. Dafür werden mehrere eindimensionale Hashtabellen in derselben Datenstruktur erstellt. Die Datenstruktur der Bloom-Filter wurde 1970 von Burton H. Bloom zur Analyse von Texten entwickelt [14]. Sie wird typischerweise beim Routing in Netzwerken, aber auch in der Datenverarbeitung eingesetzt. Der Vorteil von Bloom-Filtern liegt im geringen Speicherplatzbedarf und in den trotzdem effizienten Zugriffsoperationen.
Im Folgenden wird eine kurze Einführung in diese Datenstruktur gegeben, Implementierungsdetails erläutert und der Einsatz in SliceDB evaluiert.

Funktionsweise

Ein Bloom-Filter besteht aus einem Bit-Array mit m Positionen, die initial alle 0 sind. Für jedes Element der gegebenen Menge, werden mittels k Hashfunktionen, mit dem Wertebereich 0 bis $m-1$, k Positionen ermittelt, an denen im Bit-Array eine 1 gesetzt wird. Dabei ist wichtig, dass die Hashfunktionen unterschiedlich und die resultierenden Werte gleichverteilt sind. Um zu überprüfen, ob ein Element in der vorher eingefügten Menge enthalten ist, werden wieder k Positionen mit den gleichen k Hashfunktionen ermittelt. Befindet sich an einer diesen Positionen im Bit-Array eine 0, dann ist das Element sicher nicht in der eingefügten Menge enthalten, andernfalls ist das Element wahrscheinlich in der eingefügten Menge enthalten.

Analyse

Bloom-Filter lassen sich als ein statistisches Modell für das 2-Klasse Klassifikationsproblem verstehen. Das Modell lernt Elemente der Klasse „in der Menge enthalten“, indem diese wie zuvor beschrieben in den Bloom-Filter eingefügt werden. Mit dem Überprüfen, ob ein Element in der gelernten Menge enthalten ist, wird dann die Vorhersage der Klasse getroffen. Das spezielle an der Konstruktion der Bloom-Filter ist, dass falsch negative Elemente (Elemente die in der Menge enthalten sind und trotzdem als „nicht in der Menge enthalten“ klassifiziert werden) nicht vorkommen. Auf der anderen Seite ist es aber möglich, dass falsch positive Elemente (Elemente die nicht in der Menge enthalten sind und trotzdem als „in der Menge enthalten“ klassifiziert werden) vorkommen.
Das Verhältnis von falsch positiven Elementen zu allen Elementen, die nicht in der gegebenen Menge enthalten sind, lässt sich unter der Annahme, dass jede Position von einer Hashfunktion mit der gleichen Wahrscheinlichkeit gewählt wird, berechnen. Zusätzlich wird hier angenommen, dass die Wahrscheinlichkeit für eine Position unabhängig ist (Das Resultat lässt sich auch ohne diese Annahmen beweisen). Die Wahrscheinlichkeit, dass eine Position nicht von einer Hashfunktion ausgewählt wird, ist $1-\frac{1}{m}$. Über alle k Hashfunktionen ist dann die

Wahrscheinlichkeit, dass eine Position nicht ausgewählt wird, $(1 - \frac{1}{m})^k$. Betrachten man nun das Einfügen von n Elementen, so ist dann die Wahrscheinlichkeit, dass eine Position weiterhin nicht ausgewählt wird, $(1 - \frac{1}{m})^{kn}$. Damit ein Element als „in der Menge enthalten“ klassifiziert wird, müssen alle k Positionen der k Hashfunktionen gesetzt sein. Die Wahrscheinlichkeit für diesen Fall ist dann, mit den Vorüberlegungen, gleich $(1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{-kn/m})^k$. Damit sinkt die Wahrscheinlichkeit für falsch positive Elemente mit einer größeren Anzahl an Bits m oder Hashfunktionen k und steigt mit einer größeren Anzahl an Elementen, die in den Bloom-Filter eingefügt werden, n (vgl. [46, S. 107-112]).

Wahl der Parameter

Um den Speicherplatzbedarf und die Ausführungszeit der Operationen möglichst gering zu halten, sollten die Anzahl der Bits m und der Hashfunktionen k möglichst klein sein. Beides führt jedoch dazu, dass die Wahrscheinlichkeit für falsch positive Elemente steigt. Damit sinkt auch die Performance des Bloom-Filters, da weniger Elemente herausgefiltert werden. Um eine gegebene Performance zu erreichen, und damit auch eine beabsichtigte Wahrscheinlichkeit für falsch positive Elemente p, lassen sich die beiden Parameter mithilfe der Überlegungen im vorigen Abschnitt wie folgt wählen:

$$m = \left\lceil -\frac{n \cdot \ln(p)}{\ln(2)^2} \right\rceil$$
$$k = \left\lceil \frac{m}{n} \cdot \ln(2) \right\rceil$$

(vgl. [46, S. 107-112]).

Implementierungsdetails

Um die Operationen auf dem Bloom-Filter möglichst effizient ausführen zu können, wird die Implementierung des MurmurHash vom Autor Austin Appleby [9] verwendet. Diese Hashfunktion soll besonders schnell sein und die berechneten Werte möglichst gleichverteilt im Wertebereich streuen. Zu beachten ist dabei, dass dies keine kryptographische Hashfunktion ist, was aber für diesen Anwendungsfall auch keine Voraussetzung ist. Um die Laufzeit für die Berechnung von k Hashfunktionen zu reduzieren, werden zusätzlich die Resultate von Kirsch et al. [37] verwendet. Nach diesen lässt sich anstatt der Hashfunktionen $h_i(x)$ die Funktion $g_i(x)$ verwenden, wobei $g_i(x) = h_1(x) + i \cdot h_2(x)$ für $i > 0$ ist, ohne die asymptotische Wahrscheinlichkeit für falsch positive Elemente zu vergrößern. So reicht es aus, die zwei Hashfunktionen $h_1(x)$ und $h_2(x)$ mit dem MurmurHash zu berechnen.
Zur Speicherung des Bit-Arrays wird ein Array vom Datentyp `unsigned long long` verwendet. Dieser Datentyp ist 64 Bit groß, womit ein Element dieses Datentyp 64 Bit des Bit-Arrays repräsentiert. Neben den Parametern wird dieses Array in einem Objekt der Klasse `BloomFilter` gespeichert. Diese stellt für ein Objekt die beiden Methoden `learn` (3.37) und `predict` (3.38) bereit, die die beschriebenen Zugriffsoperatoren bereitstellen.
Die Klasse `BloomFilter` stellt zusätzlich noch die beiden statischen Funktionen `create` und `filter` bereit, die die Methoden `learn` und `predict` aufrufen. Die

Prozedur learn(Bloom-Filter Objekt *filter*, Element e)

1 $hash_1 \longleftarrow \text{MurmurHash}(e, seed = 0)$
2 $hash_2 \longleftarrow \text{MurmurHash}(e, seed = hash_1)$
3 **for** $i \in \{1, \ldots, k\}$ **do**
4 $\quad hash \longleftarrow |(hash_1 + i \cdot hash_2) \mod m|$
5 $\quad word \longleftarrow hash \ / \ 64$
6 $\quad mask \longleftarrow 2^{(hash \mod 64)}$
7 $\quad filter.array[word] \longleftarrow filter.array[word] \mid mask$

Algorithmus 3.37: Hinzufügen eines Elementes in einen Bloom-Filter.

Funktion predict(Bloom-Filter Objekt *filter*, Element e)

1 $hash_1 \longleftarrow \text{MurmurHash}(e, seed = 0)$
2 $hash_2 \longleftarrow \text{MurmurHash}(e, seed = hash_1)$
3 **for** $i \in \{1, \ldots, k\}$ **do**
4 $\quad hash \longleftarrow |(hash_1 + i \cdot hash_2) \mod m|$
5 $\quad word \longleftarrow hash \ / \ 64$
6 $\quad mask \longleftarrow 2^{(hash \mod 64)}$
7 $\quad$ **if** $(filter.array[word] \ \& \ mask) = 0$ **then**
8 $\quad\quad$ **return** $false$
9 **return** $true$

Algorithmus 3.38: Existenz eines Elementes in einen Bloom-Filter überprüfen.

Funktion `create` erstellt ein Objekt der Klasse `BloomFilter` und fügt ein Array von Elementen diesem hinzu. Die Funktion `predict` testet für ein Objekt der Klasse `BloomFilter` und einem Array von Elementen, ob die Elemente in dem Bloom-Filter vorhanden sind. Die Ausgabe ist eine Spalte von `RIDs`, die die `RIDs` der Elemente enthält, die der Bloom-Filter als „in der Menge enthalten“ klassifiziert.

Verwendung in SliceDB

Ein Bloom-Filter lässt sich in SliceDB auf einem Objekt der Klasse `BaseColumn` mit der Funktion `create_bloom_filter` erstellen. Mit diesem lässt sich ein Objekt der Klasse `BaseColumn` mit der Funktion `select_filter` filtern. Das Ergebnis dieser Funktion ist eine Spalte von `RIDs`. Diese Funktionen rufen die zuvor beschriebenen Funktionen `create` und `filter` auf. Die Parameter der Bloom-Filter werden so gewählt, dass diese eine beabsichtigte Wahrscheinlichkeit für falsch positive Elemente p von 1% aufweisen.
Bloom-Filter lassen sich in einem OLAP-Szenario einsetzen, um die Anzahl der Tupel der Faktentabelle zu reduzieren, die von einem Join-Operator verarbeitet werden müssen. Dafür werden rechts-tiefe Bäume benötigt, sodass die Tupel nacheinander mit den Dimensionstabellen gejoint werden. Bei dem Einsatz des Hash-Joins muss die Hashtabelle jeweils für die Dimensionstabellen erstellt werden. So kann durch ein frühes Filtern der Tupel der Faktentabelle, die Anzahl

der Nachschlage-Operationen in den Hashtabellen reduziert werden. Dafür wird für jede Dimensionstabelle vor dem Join-Operator ein Bloom-Filter erstellt und die Faktentabelle vor dem ersten Join-Operator mit diesen gefiltert.

Evaluation

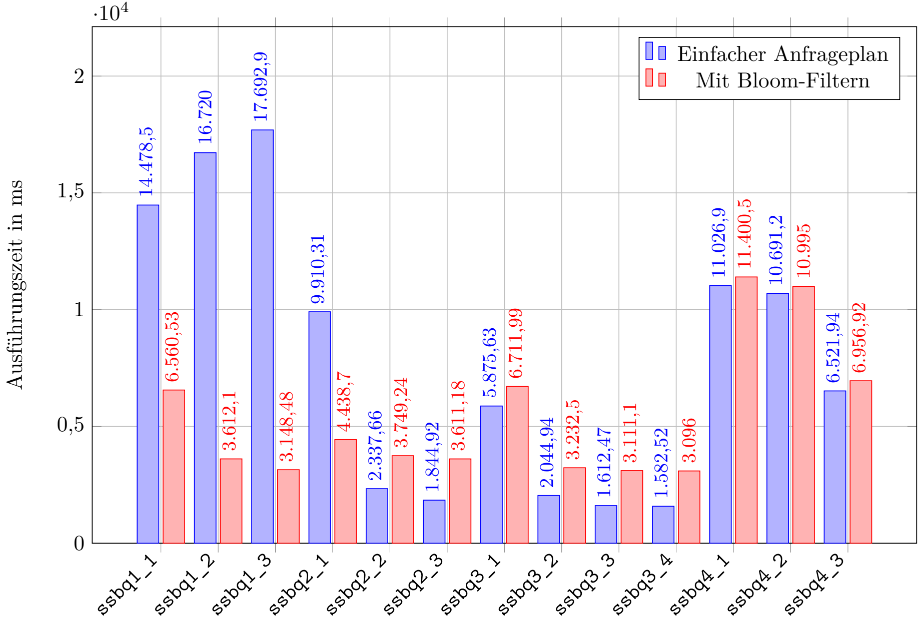


Abbildung 3.39: Durchschnittliche Ausführungszeiten über jeweils 20 Ausführungen der Anfragen des Star Schema Benchmarks [53] mit Skalierungsfaktor 10, zum einen mit einem einfachen Anfrageplan und zum anderen mit dem Einsatz von Bloom-Filtern (weitere Details zu den Benchmarks in Kapitel 5.1.1).

In Abbildung 3.39 werden die Ausführungszeiten der Anfragen mit einem einfachen Anfrageplan und die Ausführung mit dem Einsatz von Bloom-Filter verglichen. Die einfachen Anfragepläne verwenden dabei die optimierten Versionen der Sortierung, der arithmetischen Ausdrücke, der Gruppierung und der Aggregation. Auffällig bei dem Vergleich ist, dass der Einsatz von Bloom-Filtern nur bei den Abfragen `ssbq1_1` bis `ssbq2_1` die jeweiligen Ausführungszeiten reduziert. Dies deutet daraufhin, dass der zusätzliche Aufwand für das Erstellen und Auswerten der Bloom-Filter größer ist als die Einsparungen durch die reduzierte Anzahl an Nachschlage-Operationen in den Hashtabellen. Bei den Abfragen `ssbq1_1` bis `ssbq2_1` entfällt ein großer Teil der Ausführungszeit auf die Kombination von mehreren Filter-Operatoren. Da diese Kombination nicht optimiert wurde und sehr ineffizient arbeitet, können Bloom-Filter die Anzahl der Tupel reduzieren, für die diese Kombination berechnet werden muss. Für diese Abfragen reduziert der Einsatz von Bloom-Filtern die Ausführungszeiten um durchschnittlichen einen Faktor von 3,7 (Faktor $= \frac{\text{Laufzeit einfacher Anfrageplan}}{\text{Laufzeit mit Bloom-Filtern}}$), im Minimum um einen Faktor von 2,2 und im Maximum um einen Faktor von 5,7.

Um die Ausführungszeiten bei Verwendung von Bloom-Filter weiter zu reduzieren, soll im Weiteren erläutert werden, wie die Laufzeiten der Zugriffsoperationen mittels Parallelisierung verringert werden können.

Parallelisierung

Ein Ziel der Projektgruppe ist die Entwicklung eines Datenbankmanagementsystems speziell für moderne Hardware. Eine Eigenschaft moderner Hardware ist die Möglichkeit, mehrere Threads parallel ausführen zu können. Dies wird ermöglicht durch Systeme mit mehreren Prozessoren und Prozessoren mit mehreren Kernen. Um dies mit SliceDB auszunutzen, haben wir uns entschieden die einzelnen Operatoren mittels OpenMP zu parallelisieren. Um die Zugriffsoperationen auf Bloom-Filter zu parallelisieren wurden zwei Varianten implementiert.

1. Variante: Für die erste Variante wurden die Schleifen der Methoden `learn` (3.37) und `predict` (3.38) und der statischen Funktionen `create` und `filter` mittels OpenMP parallelisiert. Dabei ergaben sich die folgenden Probleme:

`learn`: Der schreibende Zugriff auf das Array ist kritisch und muss mit einem kritischen Abschnitt für jedes Element abgesichert werden.

`predict`: Ein Abbrechen der Schleife ist in OpenMP nicht möglich. Alternativ wird die Abbruchbedingung in einer booleschen Variable mittels OpenMP Reduktion gespeichert und deren Wert nach der Ausführung der gesamten Schleife ausgegeben.

`create`: Der schreibende Zugriff auf den Bloom-Filter ist kritisch. Dieser wurde mit den Änderungen in `learn` bereits abgesichert.

`filter`: Das Schreiben der `RIDs` der korrekt klassifizierten Elemente ist kritisch. Da andere Operatoren davon ausgehen, dass die `RIDs` in der gleichen Reihenfolge wie die Daten in der Eingabe gespeichert werden, wird davon abgesehen dies zu parallelisieren. So lassen sich lediglich die Aufrufe der `predict` Methode parallelisieren und die jeweiligen Rückgabewerte zwischenspeichern.

Zur Beurteilung der Parallelisierung werden in Abbildung 3.40 die Ausführungszeiten der Anfragen mit einem einfachen Anfrageplan und die Ausführung mit dem Einsatz von Bloom-Filter verglichen. Dabei sind die Zugriffsoperationen auf die Bloom-Filter wie in der 1. Version beschrieben parallelisiert. Bei der Betrachtung dieser Abbildung fällt im Vergleich zu Abbildung 3.39 auf, dass die Ausführungszeiten der Anfragepläne mit dem Einsatz von Bloom-Filtern durch die Parallelisierung größer geworden sind. Das bedeutet, dass durch die Parallelisierung der Aufwand für das Verteilen der Daten und das Synchronisieren der kritischen Abschnitte größer ist als der Laufzeitgewinn durch die parallele Ausführung. Bei der Betrachtung der Parameter fällt auf, dass die Anzahl der Hashfunktionen k bei einer beabsichtigten Wahrscheinlichkeit für falsch positive Elemente p von 1% nicht sehr groß ist. Bei den durchgeführten Experimenten ist k immer gleich 7. Das zeigt, dass bei der Parallelisierung der Methoden `learn` und `predict` nur 7 Schleifendurchläufe auf die Threads verteilt werden. Damit werden einem Thread nur sehr wenige Berechnungen zugeteilt und der

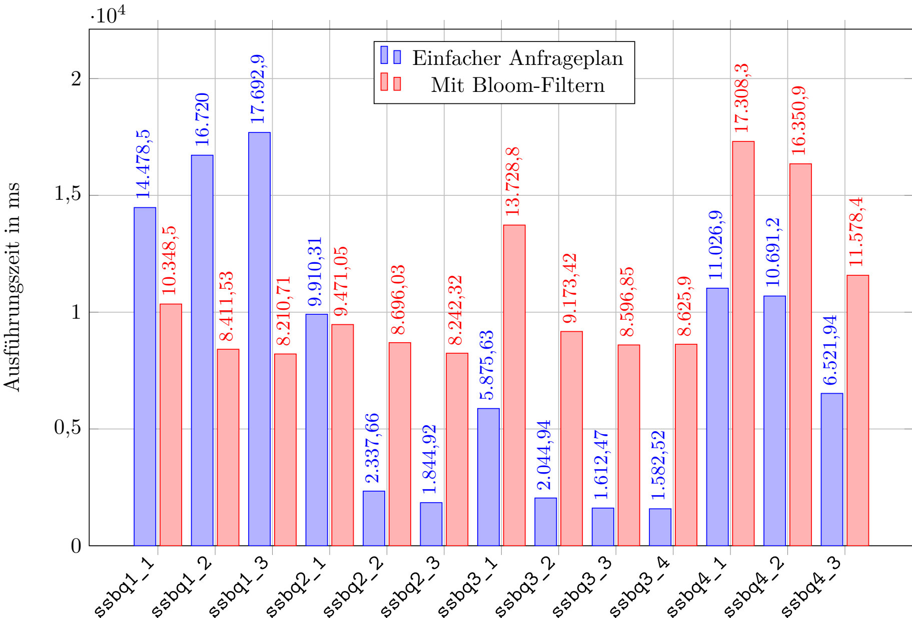


Abbildung 3.40: Durchschnittliche Ausführungszeiten über jeweils 20 Ausführungen der Anfragen des Star Schema Benchmarks [53] mit Skalierungsfaktor 10, zum einen mit einem einfachen Anfrageplan und zum anderen mit dem Einsatz von Bloom-Filtern. Die Zugriffsoperationen auf die Bloom-Filter wurden mit der 1. Version parallelisiert (weitere Details zu den Benchmarks in Kapitel 5.1.1).

Aufwand für das Verteilen der Daten und das Synchronisieren der kritischen Abschnitte ist größer als der Laufzeitgewinn durch die parallele Ausführung der Schleifendurchläufe.

2. Variante: Da sich gezeigt hat, dass die Anzahl der Schleifendurchläufe der `learn` (3.37) und `predict` (3.38) Methode relativ klein ist und sich die Laufzeit dieser durch die Parallelisierung vergrößert hat, wird auf die Parallelisierung dieser Methoden verzichtet. Da damit der Zugriff auf den Bloom-Filter nicht mehr abgesichert ist, wird auch auf die Parallelisierung der `create` Funktion verzichtet. Somit wird nur noch die `filter` Funktion parallelisiert und es ist nicht mehr nötig kritische schreibende Zugriffe auf den Bloom-Filter abzusichern. Um die Parallelisierung mit der 2. Version zu bewerten, werden in Abbildung 3.41 die Ausführungszeiten der Anfragen mit einem einfachen Anfrageplan und die Ausführung mit dem Einsatz von Bloom-Filter verglichen. Dabei sind die Zugriffsoperationen auf die Bloom-Filter wie in der 2. Version beschrieben parallelisiert. Mit der Parallelisierung der `filter` Funktion sieht man, dass die Ausführungszeiten der Anfragepläne, die mittels Bloom-Filter vorfiltern, kleiner sind als die Ausführungszeiten der einfachen Anfragepläne Abfragen. Am größten ist der Unterschied weiterhin bei den Abfragen `ssbq1_1` bis `ssbq2_1`. Für diese sind die Anfragepläne, die Bloom-Filtern einsetzen, im Durchschnitt um einen Faktor von 8,8 und für die restlichen Abfragen um einen Faktor von 1,4 schneller. Über alle Abfragen, sind die Anfragepläne, die Bloom-Filter einsetzen,

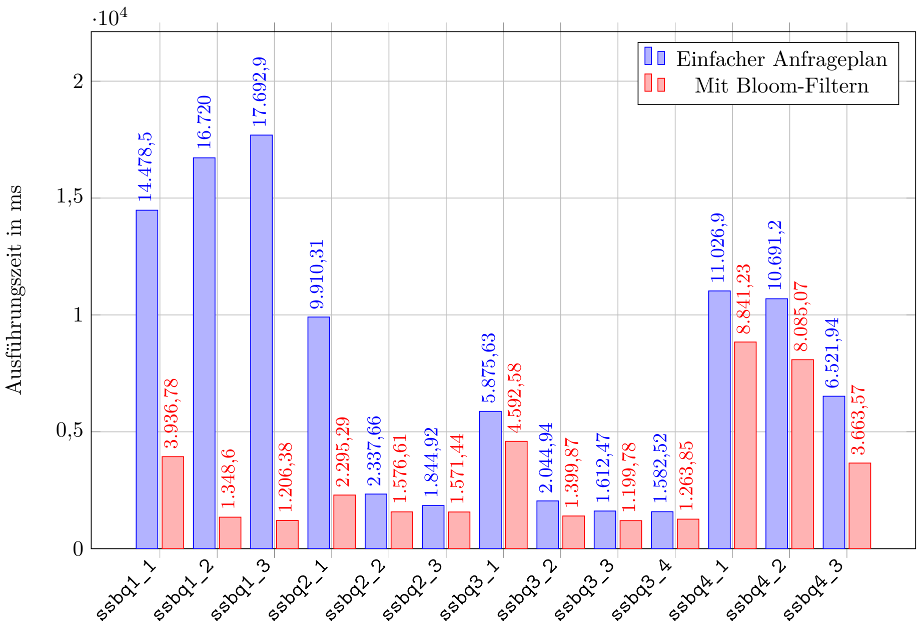


Abbildung 3.41: Durchschnittliche Ausführungszeiten über jeweils 20 Ausführungen der Anfragen des Star Schema Benchmarks [53] mit Skalierungsfaktor 10, zum einen mit einem einfachen Anfrageplan und zum anderen mit dem Einsatz von Bloom-Filtern. Die Zugriffsoperationen auf die Bloom-Filter wurden mit der 2. Version parallelisiert (weitere Details zu den Benchmarks in Kapitel 5.1.1).

im Durchschnitt um einen Faktor von 3,6, im Minimum um einen Faktor von 1,2 und im Maximum um einen Faktor von 14,7 schneller.

Zusammenfassung & Ausblick

In diesem Kapitel wurde gezeigt, wie sich mithilfe von Bloom-Filtern Tupel vorfiltern lassen, um die Anzahl der Tupel, die von einem Join-Operator weiterverarbeitet werden müssen, zu reduzieren. Um diese zusätzlichen Operationen im Anfrageplan effizient ausführen zu können, wurde ein Teil der Zugriffsoperationen mittels OpenMP parallelisiert. Mit dieser Optimierung und Anfrageplänen, die Bloom-Filter einsetzen, lassen sich die Abfragen des Star Schema Benchmarks [53] im Durchschnitt um einen Faktor von 3,6 schneller ausführen.

Bei den Experimenten mit Bloom-Filtern ist aufgefallen, dass die Kombination von mehreren Filter-Operatoren sehr ineffizeint umgesetzt wurde. Dies bietet Potenzial für Optimierungen, vor allem für Abfragen, die mehrere Filter-Operatoren benötigen. Eine weitere Optimierungsmöglichkeit ist der Einsatz von SIMD-Instruktionen, die moderner Hardware bereitstellt, um die Zugriffsoperationen auf Bloom-Filter zu beschleunigen. Des Weiteren sollte SliceDB um einen Optimierer erweitert werden, der bei entsprechenden Anfragen automatisch Bloom-Filter in den Anfrageplan einbaut, sodass keine speziellen Anfragepläne manuell erstellt werden müssen.

3.3.10 Insert Operation *(Maximilian Berens)*

Ein weitere, besondere Operation ist die `Insert` Operation. Im Gegensatz zu den anderen Operatoren ist sie kein Teil des Ausführungsplans, sondern kann direkt von dem Benutzer über die Kommandozeile aufgerufen werden:

`insert [pfad]`
: Dieser Befehl fügt mit Hilfe eines XML Statements (in einer Datei) Daten in eine bereits bestehende Datenbank ein. Die spezifizierte Tabelle muss exisiteren. Nur vollständige Tupel angeben!

Zu beachten ist, dass die Tupel vollständig sein müssen. Es ist erforderlich, dass alle Spalten und Tabellen bereits existieren. Die Insertoperation erlaubt es zugleich mehrere Tupel an zu geben. *Der Benutzer ist jedoch angehalten, bei der Angabe mehrerer Tupel auch diese immer vollständig zu definieren, da es sonst dazu kommen kann, dass die Spalten einer Tabelle unterschiedlich lang werden.* Die Insert Operation macht Gebrauch von der Append-Funktionalität der `Column` Klasse, die Daten werden demnach immer an die Tabelle angehängt. Einzelne Attribute werden automatisch in den passenden Typ gecastet, der anhand der Spalte ermittelt wird. Falls nötig wird neuer Speicherplatz reserviert. Es gilt zu beachten, dass es in SliceDB keinerlei Konsistenzgarantien, ob zum Beispiel eine Spalte sortiert ist, exisiteren. Weiter ist es nicht möglich, Daten mittels *Delete* zu entfernen bzw. zu *Update*n. Ein Beispiel für ein einzufügendes Tupel in die Tabelle Lineorder ist nachfolgend zu finden:

```
1: <insert>
2:     <table name="lineorder">
3:         <column name="lo_orderkey">
4:             6000001
5:         </column>
6:         <column name="lo_linenumber">
7:             1
8:         </column>
9:         <column name="lo_custkey">
10:            15894
11:        </column>
12:        <column name="lo_partkey">
13:            2123
14:        </column>
15:        <column name="lo_suppkey">
16:            422
17:        </column>
18:        <column name="lo_orderdate">
19:            19940903
20:        </column>
21:        <column name="lo_orderpriority">
22:            3-MEDIUM
23:        </column>
24:        <column name="lo_shippriority">
25:            9
26:        </column>
27:        <column name="lo_quantity">
28:            42
```

```
29:         </column>
30:         <column name="lo_extendedprice">
31:             3142736
32:         </column>
33:         <column name="lo_ordtotalprice">
34:             3742529
35:         </column>
36:         <column name="lo_discount">
37:             2
38:         </column>
39:         <column name="lo_revenue">
40:             3113288
41:         </column>
42:         <column name="lo_supplycost">
43:             67387
44:         </column>
45:         <column name="lo_tax">
46:             8
47:         </column>
48:         <column name="lo_commitdate">
49:             18951204
50:         </column>
51:         <column name="lo_shipmode">
52:             WURST
53:         </column>
54:     </table>
55: </insert>
```

Kapitel 4

Projektablauf

In diesem Kapitel beschreiben die einzelnen Untergruppen Interface, Operatoren und Storage wie das Projekt, aus Gruppenperspektive betrachtet, gelaufen ist. Dabei teilt sich das Kapitel in drei Abschnitte auf. Zunächst werden im ersten Abschnitt die positiven und negativen Aspekte des ersten Semesters aufgezählt, damit anschließend im nächsten Abschnitt daraus Konsequenzen für das zweite Semester gezogen werden können.

Author: Majuran Rajakanthan

4.1 Erstes Semester *(Majuran Rajakanthan)*

In diesem Abschnitt legt jede Untergruppe ihre Meinung zum vergangenen ersten Semester dar.

Interface *(Maximilian Berens)*

Zusammenfassend lässt sich über das erste Semester sagen, dass es deutliche Defizite im Ablauf des Projektes gab. Dies zeigte sich insbesondere gruppenübergreifend. Anfängliche Planungen wurden nur teilweise umgesetzt und wesentliche Änderungen/Abweichungen wurden nicht dokumentiert bzw. mitgeteilt. Die zur Dokumentation und Kommunikation nutzbaren Mittel waren gegeben und mehr als hinreichend, wurden aber nur von einer Teilmenge aller Projektmitglieder überhaupt benutzt. Letztlich führte dies dazu, dass die Aufgabenverteilung nicht der zu Anfang Festgelegten entsprach. Viele Features, die zu Anfang angedacht waren, wurden so nicht umgesetzt. Grundsätzlich mangelt es an hinreichend dokumentierten Quellcodes, insbesondere der Schnittstellen, welche explizit für die Nutzung durch Andere (Gruppen) gedacht waren und überwiegend sehr wenige bis gar keine Kommentare oder Benutzungshinweise enthielten. Des Weiteren ließ die Gruppenarbeit auch ein notwendiges Maß an Eigeninitiative vermissen. Positiv zu erwähnen war die allgemeine Bereitschaft auf Nachfrage für kurzfristig benötigte Aufgaben ein zu springen und die wöchentlichen Treffen, die einen Austausch des aktuellen Stands ermöglichten, wobei bei den zwei zusätzlichen Terminen pro Woche nur selten die Mehrheit der Leute anwesend war.

Operatoren *(Tristan Schäfer)*

Die Gruppe Operatoren hat zu Beginn des Semesters zunächst ermittelt, welche Operatoren benötigt werden und welche Herangehensweise für die Implementierung gewählt wird. In diesem Zuge wurden im Dialog mit der Projektleitung erste Überlegungen zur Systemarchitektur getroffen und Konzepte erarbeitet. Dabei wurde eine Vielzahl an wissenschaftlichen Texten genutzt, um sich insbesondere mit dem Aspekt der Zwischenergebnisse zu befassen.
Die endgültige Systemarchitektur weichte von den bisherigen Planungen der Gruppe Operatoren ab. Diese Änderungen führten dazu, dass viele zuvor getätigte Überlegungen und auch erste Implementierungen nunmehr hinfällig waren. Diesbezüglich hätten sich weitere Treffen der aller Arbeitsgruppen angeboten, in denen Vorschläge zur Systemarchitektur gemeinsam erörtert werden. Rückblickend wäre es zudem sinnvoll gewesen, Entscheidungen zur Systemarchitektur bereits möglichst früh zu verschriftlichen. Somit wäre eine Aufgabenteilung klar und deutlich definiert. Ebenso könnte ein öffentlich einsehbarer Arbeitsplan aller Gruppen für weitere Transparenz sorgen. Auf diesem Wege wäre eine Priorisierung der Arbeitspakete möglich und eine Planungssicherheit gegeben. Insbesondere könnte somit vermieden werden, dass neue Anforderungen erst in der kritischen Projektphase bekannt werden.
Die Schnittstellen zu den übrigen Arbeitsgebiete wurden nicht formal definiert, sondern auf Zuruf implementiert. Wegen der starken Kapselung der verschiedenen Tätigkeitsbereiche traten hier jedoch keine Probleme auf. Insofern ist aus Sicht der Gruppe Operatoren eine formelle Definition der Schnittstellen weiteren Projektverlauf nicht zwingend erforderlich.

Storage *(Philipp Krause)*

Als gemeinsames Gruppentreffen wurden anfangs zwei Termine pro Woche veranschlagt, um den aktuellen Stand des Projekts sowie das weitere Vorgehen zu besprechen. Nachdem ein *grobes* Konzept gemeinsam ausgarbeitet worden war, wurde die Gruppe zur Bearbeitung einzelner Themenfelder unterteilt. Dies hat in der Anfangsphase gut funktioniert und alle Projektmitglieder konnten ihre Ideen und Vorschläge mit einbringen. Gut war ebenfalls, dass alle Projektmitglieder durch das regelmäßgige Treffen auf dem gleichen *Stand* waren und daher jeder das Gesamtprojekt im Blick hatte. Im Verlaufe des Semesters wurden die gemeinsamen Treffen allerdings deutlich schwieriger zu organisieren, da es in den meisten Fällen nicht möglich war, einen gemeinsamen Termin zu vereinbaren, an dem alle Gruppenmitglieder erscheinen können. Dadurch entstanden - bezüglich des Gesamtprojekts - Informationslücken, die die weiteren Planungsvorgänge sehr erschwerten. Ein großes Problem bestand zusätzlich - wie sich im späteren Projektverlauf zeigte - darin, dass Schnittstellen nicht eindeutig definiert worden waren. Dies hatte zur Folge, dass Konzepte verändert wurden und Methoden der einzelnen Gruppen mehrfach angepasst werden mussten. Hierbei ist allerdings zu erwähnen, dass die Festlegung von festen Schnittstellen schwierig gewesen wäre, da das für die einzelnen Themenfelder notwendige Wissen fehlte, welches sich erst im weiteren Semesterverlauf schrittweise ergab. Zudem wurden viele Methoden nicht mit Kommentaren versehen, obwohl sich darauf anfangs einheitlich verständigt wurde, sodass es teilweise schwierig und zeitaufwändig war, sich in das neue Konzept einzuarbeiten. Neue Konzepte, die beispielsweise mit anderen Gruppen

besprochen und umgesetzt wurden, wurden nicht immer schriftlich festgehalten und allen Projektmitgliedern, die von der Änderung betroffen waren, mitgeteilt. Die Änderungen, die im Redmine-System protokolliert worden waren, wurden nicht regelmäßig verfolgt, sodass sich auch aufgrund dessen ein unterschiedlicher Wissensstand entwickelte. Jede Gruppe hatte sich primär auf ihr Thema fokussiert, während sich das *Grundgerüst* des Projekts immer weiter verändert hatte. Verschiedene *Branches*, in denen entwickelt wurde, machten den aktuellen Projektstand unübersichtlich, da die Branches teilweise zu selten zusammengeführt worden sind. Hier wurde teilweise unabhängig voneinander auf der Basis eines alten Konzepts entwickelt, sodass im Nachhinein viele Anpassungen notwendig waren.

4.1.1 Verbesserungsvorschläge für das zweite Semester *(Majuran Rajakanthan)*

Im Folgenden geben die einzelnen Untergruppen Vorschläge, was in der zweiten Phase der Projektgruppe nächstes Semester verbessert werden kann, um die oben beschriebenen Probleme beseitigen zu können.

Interface *(Maximilian Berens)*

Grundsätzlich sollte die Kommunikation (Treffen, Nutzung des Redmines, Quellcodedokumentation, etc.) und die Eigeninitiative verbessert werden. Der Bedarf an ordentlich dokumentierten Schnittstellen wird in der Zukunft weiter zunehmen, sodass dies in keinem Fall weiter vernachlässigt werden darf. Auch wenn die wöchentlichen Treffen einen Austausch der Gruppen ermöglichte, so wäre es hilfreich, wenn die im letzten Semester vereinbarten, wöchentlichen Zusatztreffen im nächsten Semester weiter geführt und vor allem zahlreicher wahrgenommen werden. Eine administrative Leitung (Gruppenleiter) für die PG, welche sich um die direkte Verteilung von Aufgaben kümmert und auf die bisher verzichtet wurde, könnte das gemeinsame Arbeiten und die Absprache bezüglich der Schnittstellen verbessern, sollte das Redmine-Ticketsystem und das Forum nicht ausreichen. Außerdem könnte eine Neubewertung der Arbeitsverteilung für eine gleichmäßigere Aufteilung der Aufgaben sorgen.

Operatoren *(Tristan Schäfer)*

Zu Beginn des zweiten Semesters sollte in Rücksprache mit der Projektleitung festgelegt werden, welche Optimierungsmaßnahmen umgesetzt werden. Im Vorfeld sollte jedes Gruppenmitglied die in seinem bisherigen Arbeitsbereichen bestehenden Möglichkeiten grob zusammenfassen. Das Ergebnis dieser Planungsphase sind entsprechende Arbeitspakete, Verantwortlichkeiten und Deadlines. Die Zusammenarbeit innerhalb der Gruppe lief vergleichsweise problemlos, so dass hier keine zusätzlichen Maßnahmen erforderlich sind.

Storage *(Philipp Krause)*

Im Folgenden werden einige Verbesserungsvorschläge vorgestellt, die die Gruppenarbeit für das zweite Semester des Projekts verbessern und erleichtern sollen. Im zweiten Semester soll die Funktionsweise von SliceDB optimiert werden.

Dazu sollten nach einer Aufgabenteilung genaue Schnittstellen definiert werden und geklärt werden, welche Funktionen diese Aufgaben erfüllen sollen. Wichtig ist ebenfalls, dass diese Vereinbarungen schriftlich festgehalten werden, sodass die einzelnen Gruppen auch unabhängig voneinander arbeiten können und sich immer auf das schriftlich festgehaltene Konzept beziehen können. Während der Planungsphase sollten alle Gruppenmitglieder anwesend sein, um ihre Ideen einbringen zu können und Fragen zu klären. Fragen wurden in der Vergangenheit meist über Email oder Whatsapp kommuniziert. Dies hatte zur Folge, dass oft gleiche Fragen von unterschiedlichen Personen gestellt wurden. Besser wäre es daher im kommenden Semester das eingerichtete Redmine-System der Projektgruppe regelmäßiger zu nutzen. Außerdem sollte jede Methode zumindest mit einem kurzen, aussagekräftigen Kommentar versehen werden. Wie von Prof. Teubner beschrieben, sollte hierbei der Quellcode dem Kommentar folgen.

4.2 Zweites Semester *(Maximilian Berens)*

Nachfolgend wird kurz der Ablauf des zweiten Semesters wieder gegeben. Dabei wird im Wesentlichen auf die Aufgabenverteilung eingegangen, da die Kleingruppenarbeit aus dem ersten Semester im Zweiten nicht fortgesetzt wurde. Abschließend folgt eine zusammenfassende Bewertung der Gruppenarbeit.

4.2.1 Aufgabenverteilung *(Maximilian Berens)*

Im Wesentlichen wurden nach der Implementierung der grundlegenden Features im ersten Semester auf die Optimierung beziehungsweise auf die Implementierung spezialisierter Operatoren zur Verbesserung der Performance eingegangen. Die Auswahl zwischen alten und neuen Operatoren geschieht im Wesentlichen über die Datei *optimizations.h* im *include/util* Verzeichnis des Projektes.
Von Stefan Noll wurde im zweiten Semester ein Invisible Join implementiert. Von Eric Fiege wurde ein verbessertes, paralleles Group-By mit neuem Hashing Verfahren und die Korrektheitstests gegen MonetDB umgesetzt. Order-By bzw. Sort wurde von Ersin Özdemir parallelisiert und durch ein partielles Gather verbessert, wobei Instruktionen innerhalb des Operators eingespart werden. Tristan Schäfer optimierte den Hash Join weiter, indem er Probe- und Buildphase trennte. Christian Schnieder verbesserte den Parser. Andreas Blume setzte die Möglichkeit um, den Operator für arithmetische Ausdrücke parallel auszuführen und implementierte dazu noch Loop Unrolling. Außerdem parallelisierte er die Aggregation. Jan Stallmann implementierte Bloom-Filter und erweiterte das Interface und den *Storage-Layer* um die Datenbankverwaltung. Maximilian Berens fügte den Insert Operator hinzu, erlaubte den Import von Datenbankschemata und stellte Logging- und Konfigurationsmöglichkeiten zur Verfügung. Majuran Rajakanthan stellte Benchmark- und Zeitmessungsmöglichkeiten für SliceDB zur Verfügung. Um Seitenstatistiken bzw. Histogramme kümmerte sich Harun Kara. Phillip Krause und Milad Nayebi waren für die Dictionarycompression zuständig.

4.2.2 Bewertung der Gruppenarbeit *(Harun Kara)*

Terminfindung Ein großes Problem bei der Gruppenarbeit war es, einen wöchentlichen Termin für das gemeinsame Meeting und die Präsenzzeit im Poolraum zu finden, an dem auch wirklich alle zwölf Mitglieder kommen konnten. Da das Meeting immer nur von kurzer Dauer war, weil nur die wichtigsten Sachen dort besprochen wurden, waren wir meistens auch vollzählig. Während der Präsenzzeit im Poolraum hingegen waren immer nur Wenige anwesend. Die schlechte Ausstattung des Poolraums mit Wlan, senkte vermutlich die Motivation noch weiter, regelmäßig an diesem Treffen teilzunehmen.

Gruppendynamik und Rollenverteilung An dem Projekt waren zwölf Personen beteiligt. Es entstanden drei Teams von 3, 4 und 5 Mitgliedern, je nach Schwierigkeitsgrad der Aufgabenstellung. Später im zweiten Semester haben sich diese Teams mehr oder weniger aufgelöst, weil sich neue Aufgabenstellungen und -Verteilungen ergeben haben. Am Anfang des Projekts haben wir zwar verschiedenen Projektmitgliedern Rollen verteilt, aber im Laufe des Projekts haben wir kaum auf diese Rollen Wert gelegt. Ein harmonisches Miteinander war uns viel wichtiger und nützlicher als eine Hierarchie. Der Professor hat die Rolle des Auftraggebers/Kunden eingenommen, der Betreuer die Rolle des Projektleiters, dann erhielt jeweils einer aus jeder Team die Rolle des Teamleiters und die Anderen waren normale Projektmitarbeiter. Der Professor hat uns nie vorgeschrieben, wie wir das System bauen sollen, sondern immer nur mögliche Wege aufgezeigt und uns die Entscheidung überlassen. Auch der Betreuer wollte uns nichts vorschreiben und hat nur entschieden, wenn wir im unklaren waren, was besser für uns ist.

Kommunikation In der Anfangsphase des Projekts waren für die Verständigung zwischen Teams, die jeweiligen Teamleiter vorgesehen. Mit dem Fortschreiten des Projekts haben sich solche klaren Strukturen aber nicht als nützlich erwiesen. Bei schwierigen Themen hat der Betreuer die Teams beraten und für das Gelingen des Projekts beigetragen. Das Redmine Projektmanagementsystem wurde von allen Mitgliedern optimal für das Projektmanagement genutzt. Durch den Einsatz von Versionsverwaltung konnten die Mitglieder alle Änderungen mitvervolgen und Keiner wurde abgehängt. Insbesondere bei hartnäckigen Fehlern wurden von Anderen Hilfestellungen geleistet.

Planung und Aufgabenverteilung Der Fahrplan für das jeweilige Semester wurde im Voraus geplant, Abgabefristen für Features und Berichte wurden vorab festgelegt, sodass sich Jeder daran richten konnte. Eine kürzere Auszeit während der Klausurphase wurde erlaubt. Die Aufgabenverteilung erfolgte auf freiwilliger Basis. Zu Beginn wurden die offenstehenden Aufgaben aufgelistet und dann konnte sich Jeder aussuchen, welche Aufgabe er gerne erledigen würde. Es arbeiteten immer Einer alleine oder zwei Personen zusammen an einem Feature. Nur selten hat sich für eine Aufgabe keiner freiwillig gemeldet und der Betreuer musste dann einen aussuchen.

Fazit Im Endeffekt hat sich jedes Mitglied in der Gruppe seinen (Lieblings-)Partner herausgesucht und die meiste Zeit mit ihm zusammengearbeitet. Wer

nicht so oft anwesend sein konnte, konnte sich trotzdem gut auf dem Laufenden halten, dank des immer gut gepflegten Projektmanagementsystems (das Wiki war sehr hilfreich) und der Rundmails bei wichtigen Entscheidungen im wöchentlichen Meeting. Unter Anderem können die Projektmitglieder von diesem einjährigen Projekt wichtige Erfahrungen bezüglich des Teamworks für zukünftige Projekte mitnehmen.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung *(Majuran Rajakanthan)*

Im ersten Teil dieses Endberichts wurden im Rahmen einer Seminarphase verschiedene Hauptspeicher-Datenbanksysteme vorgestellt. Dadurch wurden Grundlagen, wie die Architektur eines DBMS, Speicherverwaltung, Anfrageverarbeitung und die verschiedenen eingesetzten Technologien vermittelt.
Der zweite Teil beschäftigte sich mit unserer Datenbank SliceDB. Dabei wurde zunächst der Aufbau, die grobe Struktur und die Abarbeitung von Anfrageplänen beschrieben, welches das *Interface* des DBMS darstellt. Daraufhin wurde auf die zweite Schicht des DBMS eingegangen, der *Storage Layer*. Diese ist für die Verwaltung der Daten und für das Delegieren von Operatorenaufrufen zuständig. Danach wurden die Implementierungen und anschließende Optimierungen der einzelnen Operatoren vorgestellt.
In diesem Kapitel folgt nun die Evaluation von SliceDB mit Hilfe des *Star Schema Benchmarks* (SSB). Daraufhin wird die Leistung von MonetDB (2.2) mit SliceDB verglichen. Anschließend folgt ein Fazit und Ausblick.

5.1.1 Benchmarks *(Majuran Rajakanthan)*

Das Interface des SliceDB hat zwei Befehle für das Benchmarking.

`benchmark [function] [runs]`
: Mit diesem Befehl kann die Ausführungszeit einer beliebigen Funktion des Interfaces [runs]-mal gemessen werden. Dabei werden die einzelnen Zeiten pro Durchlauf und die Durchschnittszeit aller Durchläufe der Funktion in eine CSV-Datei exportiert. Beispielsweise führt *benchmark ssbq1_1 20* die Query 1.1 des Star Schema Benchmarks (SSB) 20-mal aus.

`benchmark_all [runs]`
: Mit diesem Befehl werden alle 13 Querys des SSB nacheinander [runs]-mal ausgeführt, wobei die einzelnen, Gesamt- und Durchschnittsausführungszeiten als CSV-Datei gespeichert werden.

Damit die Laufzeiten der einzelnen Operatoren ebenfalls ausgewertet werden können, besteht die Möglichkeit diese durch den folgenden Code messen zu lassen.

```
if (OPTION(OPT_BENCHMARK,OPT_BENCHMARK_TYPE) | Util::bm_force) {
   Util::Timestamp start = Util::getTimestamp();
   Zu_messender_Operator;
   Util::Timestamp stop = Util::getTimestamp();
   Util::result("Name_des_Operators", Util::diff_ns(start, stop));
}
```

Dadurch werden neben den Queryzeiten, die Zeiten der Operatoren zusätzlich in die CSV-Datei eingetragen. Damit der Operator nicht jedes Mal bei Ausführung automatisch gemessen wird, kann durch eine Konfiguration das Benchmarking aus- und eingeschaltet werden (siehe auch 3.1.2).

Star Schema Benchmark (SSB)

SliceDB wird in diesem Abschnitt mit Hilfe des SSB getestet. Dafür wird folgende Testumgebung verwendet:

DBMS:	SliceDB
OS:	Ubuntu 12.04.5 LTS 64-bit (Linux Kernel 3.13.0)
Hardware:	Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz (64GB RAM)

Für die Datenerzeugung werden die *Scale Faktoren (SF)* 1, 4, und 10 verwendet. Nach dem Laden der Tabellen in die Datenbank, werden die 13 Queries des SSB ausgeführt (20 Durchläufe). Dabei werden die Messungen ohne Optimierungen, mit Invisbile-Join (3.3.8) und mit Bloom-Filtern (3.3.9) durchgeführt. Es wurden folgende Optimierungen verwendet:

```
#define _PARALLEL_BLOOMFILTER_
#define _SORT_STABLE_
#define _SORT_PARTIAL_GATHER_
#define _PARALLEL_ARITHMETIC_OPERATION_
#define _GROUPY_CUCKOO_HASH_
#define _PARALLEL_AGGREGATION_
```

Die einzelnen Optimierungen werden in den jeweiligen Abschnitten in dem Kapitel Operatoren (3.3) erläutert. Im Folgenden sind die Ergebnisse des Benchmarks tabellarisch und als Diagramm dargestellt.

Ausführungszeit der Queries							
Optimierung	**Q1.1**	**Q1.2**	**Q1.3**	**Q2.1**	**Q2.2**	**Q2.3**	**Q3.1**
Ohne	1407	1694	1861	962	232	174	481
Invisible-Join	-	-	-	407	387	387	433
Bloom-Filter	404	135	116	180	158	135	391

Optimierung	**Q3.2**	**Q3.3**	**Q3.4**	**Q4.1**	**Q4.2**	**Q4.3**	**Total**
Ohne	183	169	164	688	705	608	9336
Invisible-Join	332	355	355	528	489	510	-
Bloom-Filter	149	123	132	458	474	269	3130

Tabelle 5.1: SSB Query Ausführungszeit für SF 1 in Millisekunden

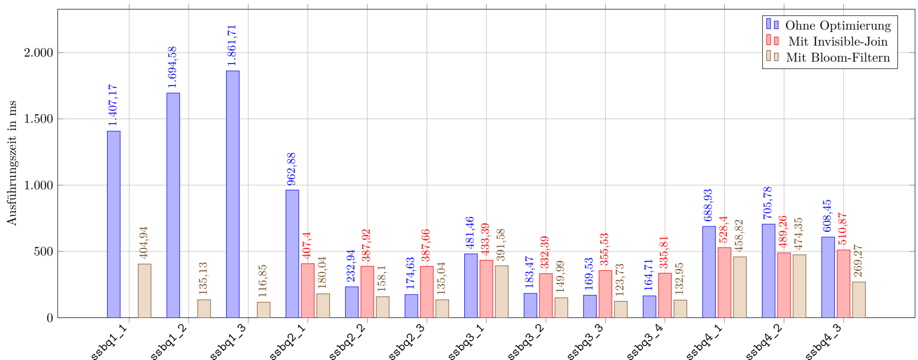


Abbildung 5.1: SSB Query Ausführungszeit für SF 1 in Millisekunden

Ausführungszeit der Queries							
Optimierung	**Q1.1**	**Q1.2**	**Q1.3**	**Q2.1**	**Q2.2**	**Q2.3**	**Q3.1**
Ohne	5599	6852	7128	4035	827	661	2244
Invisible-Join	-	-	-	1169	1219	1208	1368
Bloom-Filter	1651	617	508	801	600	499	1791

Optimierung	**Q3.2**	**Q3.3**	**Q3.4**	**Q4.1**	**Q4.2**	**Q4.3**	**Total**
Ohne	809	675	747	3555	3319	2018	38469
Invisible-Join	1125	1122	920	1532	1431	1494	-
Bloom-Filter	639	571	618	2808	2537	1203	14843

Tabelle 5.2: SSB Query Ausführungszeit für SF 4 in Millisekunden

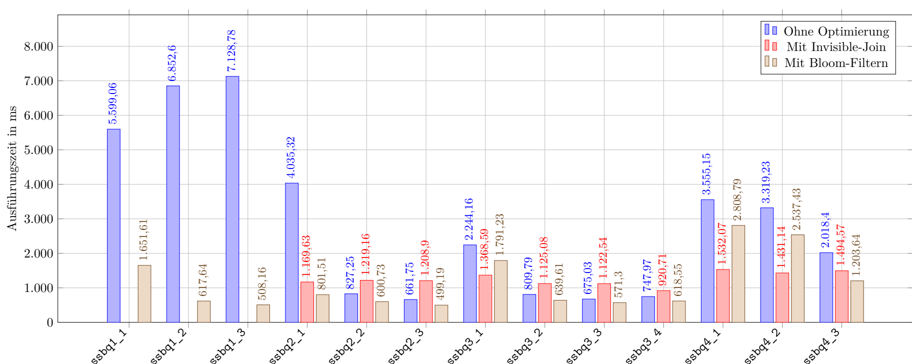


Abbildung 5.2: SSB Query Ausführungszeit für SF 4 in Millisekunden

Ausführungszeit der Queries							
Optimierung	**Q1.1**	**Q1.2**	**Q1.3**	**Q2.1**	**Q2.2**	**Q2.3**	**Q3.1**
Ohne	13382	16281	17647	9695	2393	1853	5997
Invisible-Join	-	-	-	2780	2515	2500	3238
Bloom-Filter	3936	1348	1206	2295	1576	1571	4592

Optimierung	**Q3.2**	**Q3.3**	**Q3.4**	**Q4.1**	**Q4.2**	**Q4.3**	**Total**
Ohne	2072	1493	1595	10699	10046	5807	98960
Invisible-Join	2948	2243	2082	3614	3263	2916	-
Bloom-Filter	1399	1199	1263	8841	8085	3663	40974

Tabelle 5.3: SSB Query Ausführungszeit für SF 10 in Millisekunden

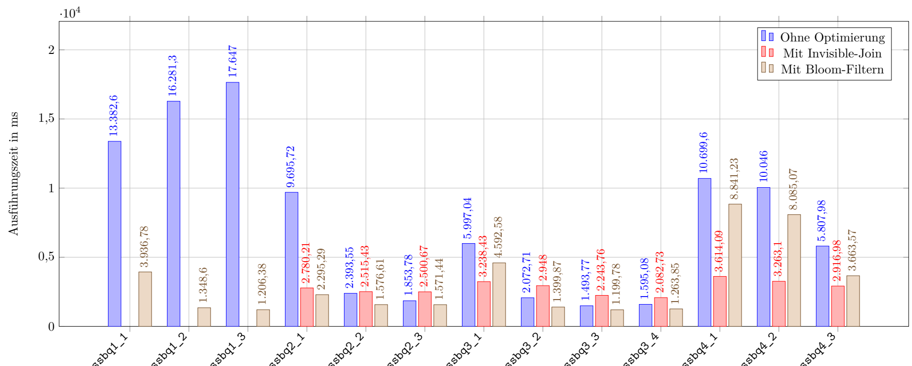


Abbildung 5.3: SSB Query Ausführungszeit für SF 10 in Millisekunden

Ausführungszeit der Queries		
SF	**SliceDB**	**MonetDB**
1	3130	1396
4	12279	2766
10	28827	4855

Tabelle 5.4: Gesamtausführungszeit der SSB Querys in Millisekunden

5.1.2 Korrektheit *Eric Fiege*

Um zu gewährleisten, dass SliceDB auch die korrekten Ergebisse für alle SSB Anfragen zurückliefert, wurde ein Test erstellt. Dieser überprüft ob die Ergebnisse von SliceDB mit den korrekten Ergebnissen aus MonetDB übereinstimmen. Dazu wurde ein Skript erstellt, das alle Anfragen des Star-Schema-Benchmarks nacheinander ausführt und die Ergebnisse, jeweils einer Anfrage, in einer Text-Datei ablegt.

Als nächstes wurde ein neuer Google-Test angelegt. Dieser lädt im ersten Schritt die SSB-Daten in SliceDB. Danach wird für jede SSB-Anfrage ein Test ausgeführt. Dazu werden jeweils die Ergebnis-Dateien von MonetDB, die durch das Skript erstellt wurden, in SliceDB geladen und im Anschluss die SSB Anfragen in SliceDB ausgeführt. Die Anfragen der ersten Kategorie enthalten lediglich eine Zahl als Ergebnis. So kann das Ergebnis von SliceDB direkt mit dem Ergebnis von MonetDB verglichen werden. Alle weiteren Anfragen enthalten mehrere Zeilen. Deshalb müssen die Ergebnisse von MonetDB in die Tabellen-Datenstruktur von SliceDB geladen werden. Danach können die Tabellen miteinander verglichen werden. Alle Anfragen des Star Schema Benchmarks enden mit einer Sortierung der Daten. Daher werden getauschte Positionen in den Datensätzen nicht vorkommen und alle Zeilen können direkt miteinander verglichen werden.

Der Test wird als ausführbare Datei im `bin` Ordner abgelegt. Außerdem gibt es einen weiteren Test, der die Korrektheit des BloomFilters überprüft und ebenfalls im `bin` Ordner zu finden ist.

Vergleichsbenchmark zwischen SliceDB und MonetDB

Im folgenden Abschnitt werden die gleichen Daten des SSB auch auf MonetDB (2.2) ausgeführt, um ein Vergleichsbenchmark zwischen beiden Datenbanken aufstellen zu können. Hierbei wurden aus den vorherigen Messungen jeweils die beste Kombination aus Invisible-Join und Bloom-Filter gewählt, um die beste Performance von SliceDB mit MonetDB zu vergleichen.

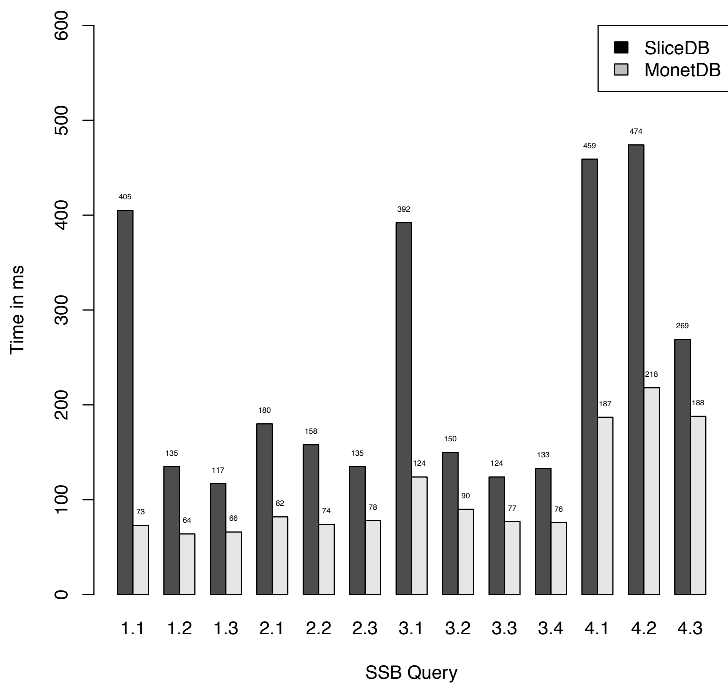


Abbildung 5.4: Vergleich der SSB Query Ausführungszeit zwischen SliceDB und MonetDB für SF1

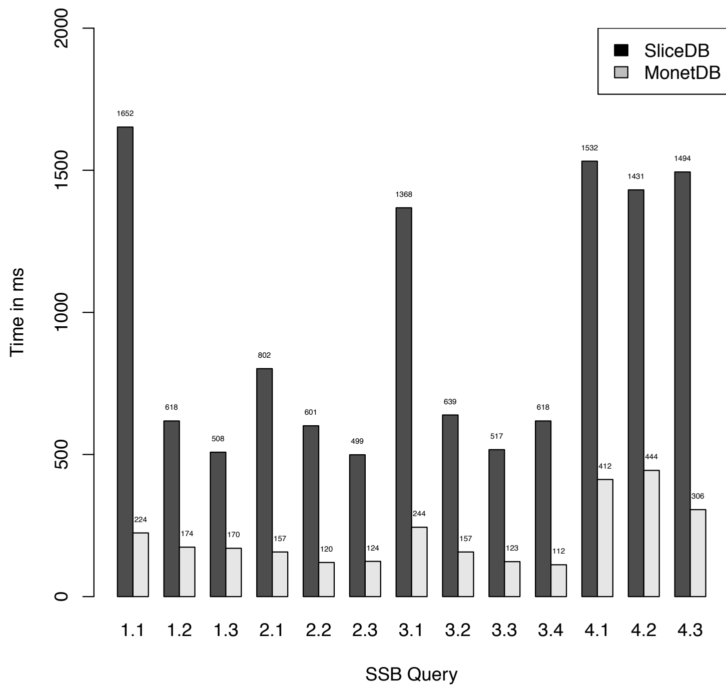


Abbildung 5.5: Vergleich der SSB Query Ausführungszeit zwischen SliceDB und MonetDB für SF4

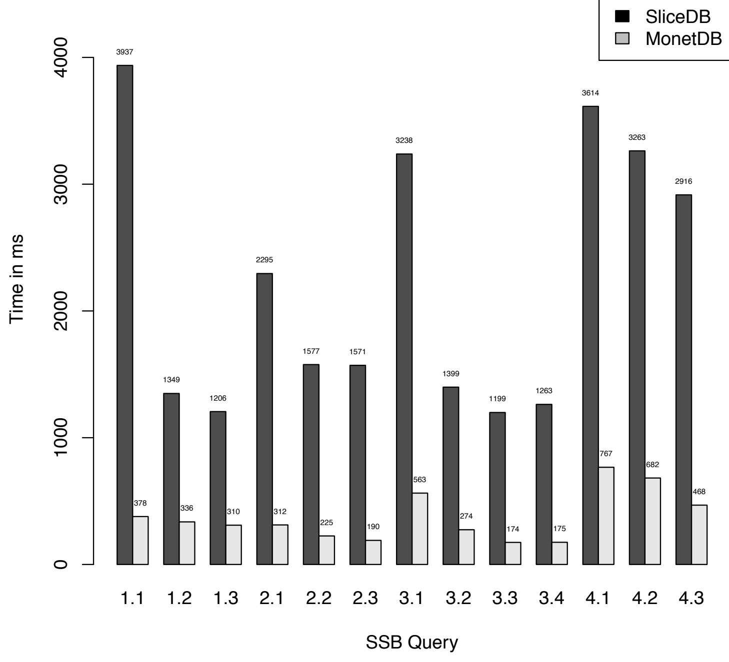


Abbildung 5.6: Vergleich der SSB Query Ausführungszeit zwischen SliceDB und MonetDB für SF10

5.2 Fazit *(Milad Nayebi / Christian Schnieder)*

In diesem Abschnitt wird ein zusammenfassender Überblick über die Arbeit der Projektgruppe und die erreichten Ziele gegeben.

In den zwei Semestern der Projektgruppenarbeit haben sich alle Teilnehmer mit der Entwicklung der eigenen Hauptspeicherdatenbank SliceDB befasst.

Zunächst wurde die Seminarphase genutzt, um sich mit unterschiedlichen existierenden Hauptspeicherdatenbanken auseinander zu setzen. Die vorgestellten Datenbanken und die darin verwendeten Techniken wurden verglichen und die einzelnen Vor- und Nachteile abgewägt. So konnten die Teilnehmer wertvolle Erfahrungen aus der Seminarphase mitnehmen, welche bei der Implementierung der eigenen Datenbank hilfreich war.

Das Minimalziel der Projektgruppe war der Entwurf und die Implementierung eines lauffähigen Hauptspeicherdatenbanksystems, welches die Anfragen des Star Schema Benchmark verarbeiten kann. Während der Planung und Implementierung der Datenbank konnten die Teilnehmer Erfahrungen in Projektarbeit sammeln. Durch die Rollen Auftraggeber und Projektleiter, sowie die anfängliche Aufteilung der Teilnehmer in Teams wurde eine Umgebung geschaffen, die einer Projektumgebung im Berufsleben nahe kommt. Hierbei haben die Teilnehmer gelernt, sich selbständig in Teams zu organisieren und mit den damit verbundenen Herausforderungen und Konflikten umzugehen. Die Projektmanagementsoftware Redmine war hierbei besonders hilfreich, um den Code in einem Git-Repository zu verwalten, die verschiedenen angefallenen Aufgaben effizient zu koordinieren und abzuarbeiten, sowie Ergebnisse im Wiki zu dokumentieren. Insgesamt haben

alle Teilnehmer im Laufe des Projektes wesentliche Erfahrungen für die Arbeit an weiteren Projekten gesammelt.
Bei der Programmierung der Hauptspeicherdatenbank SliceDB waren die technischen Erkenntnisse aus der Seminarphase hilfreich und das Minimalziel, eine lauffähigen Version der Datenbank zu implementieren, wurde bereits zum Ende des ersten Semesters erreicht. So wurde im zweiten Semester verschiedene Optimierungen, vor allem im Bereich der Operatoren, vorgenommen und ausgewertet. Die Datenbank gliedert sich im wesentlichen in drei Bereiche: Interface, Datenhaltung und Operatoren. Im Folgenden wird jeweils kurz auf die einzelnen Bereiche eingegangen:

Interface *(Maximilian Berens)* Die Interfacegruppe befasste sich mit der Implementation des Parsers, der Benutzerschnittstelle (Eingabeprompt), des Vulcano-Interfaces für sämtliche Operatoren und die damit verbundenen Aufrufe der konkreten Implementationen. Des Weiteren wurde eine Datenstruktur für den Anfrageplan sowie Beispielpläne für die Anfragen des Star Schema Benchmarks erstellt und für die Verknüpfung mit den Daten der Datenhaltungsschicht gesorgt. Darüber hinaus wurden Utility Funktionen für das Debugging und die Ausgabe sowie Frameworks für die Implementation von Unittests und die mehr oder weniger plattformunabhängige Erstellung des Projekts (CMake) bereit gestellt.

Datenhaltung *(Milad Nayebi)* Die Aufgabe von Storage-Layer ist die Verwaltung und Speicherung von Daten. Dazu gehören die Verwaltung des Hauptspeichers und das Laden und Speichern der Daten. Zusätzlich stellt der Storage-Layer verschiedene Datenstrukturen bereit. Die Datenhaltung-Gruppe ist insgesamt für Repräsentation der Daten, Speicherverwaltung, Operatoren auf Spalten, Schmaverwaltung und Zwischenergebnisse verantwortlich.

Operatoren *(Andreas Blume)* Der Aufgabenbereich des Operatoren-Teams lag in der Implementierung der einzelnen Operationen, die für den Star Schema Benchmark [51] benötigten werden, und der Integration der Operatoren im Executer sowie in den Column-Datenstrukturen der Datenhaltung. Die einzelnen Operationen wurden zunächst auf eine einfache Art und Weise umgesetzt, damit sie schnellst möglichst in das System integriert werden konnten und so das Zusammenspiel der einzelnen Teilbereiche erprobt werden konnte. Im zweiten Semester konnte im Bereich der Operatoren durch Optimierungen bestehender, sowie durch die Implementierung neuer Algorithmen hohe Performancesteigerungen erreicht werden. So wurden unter anderem der Invisible-Join (Abschnitt 3.3.8), sowie der Bloom-Filter (Abschnitt 3.3.9) implementiert. Dieses macht sich auch im Vergleich zu MonetDB bemerkbar, wo SliceDB es teilweise schafft nah an die Laufzeit von MonetDB heran zu kommen.

5.3 Ausblick *(Harun Kara / Christian Schnieder)*

Wie in diesem Bericht zu sehen ist, erfüllt die im Rahmen der Projektgruppe entwickelte Datenbank SliceDB alle vorgegebenen Anforderungen. Auch im Bereich der Optimierung wurden bereits weitere Algorithmen implementiert, wodurch schnelle Anfragen möglich sind.

Wenn man SliceDB im Vergleich zu den vorgestellten bekannten Hauptspeicherdatenbanken betrachtet, wird man feststellen, dass sie noch einige Möglichkeiten der Optimierung bietet. Diese Möglichkeiten ziehen sich durch alle Schichten von SliceDB.
Im Bereich des Interfaces können derzeit geparste Anfragen nicht von den Optimierungen bei den Operatoren profitieren, da bisher keine automatische Optimierung für die Auswahl von Operatoren zur Verfügung steht. Ebenfalls sinnvoll wäre auch eine automatische Optimierung für die Umstrukturierung des Anfageplans um so Datenflüsse zu optimieren. Der Storage-Layer kann um effiziente Caching- und Komprimierungsalgorithmen erweitert werden, um in diesem Bereich die Hardwaremöglichkeiten voll auszunutzen. Im Bereich der Operatoren können weitere Algorithmen implementiert werden. Außerdem sind auch bei bestehenden Algorithmen noch nicht alle Möglichkeiten durch effiziente Parallelisierung ausgeschöpft.
Dies ist nur eine kleine Auswahl an Verbesserungsmöglichkeiten für SliceDB. Der Herausforderung moderne Hardware immer effizienter zu nutzen, stellen sich auch die großen Datenbankentwickler und auch bei denen ist die Optimierung ein andauernder, spannender Prozess, welcher immer wieder interessante Lösungen zum Vorschein bringt.

Literaturverzeichnis

[1] MonetDB Assembly Language Syntax. https://www.monetdb.org/Documentation/Manuals/MonetDB/Appendices/MALsyntax, 2015.

[2] A. Colin Cameron. Excel 2007: Histogram. http://cameron.econ.ucdavis.edu/excel/ex11histogram.html, 2009.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[4] Actian vector - the revolutionary high performance analytics database. http://www.odbms.org/wp-content/uploads/2014/08/WP01-ActianVector-0424.pdf, 2014.

[5] SAP AG. Sap hana. hana.sap.com/. Stand Juni 2015.

[6] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[7] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[8] Amazon. Amazon web services: Sap hana one. https://aws.amazon.com/marketplace/pp/B009KA3CRY. Stand Juni 2015.

[9] Austin Appleby. MurmurHash. https://sites.google.com/site/murmurhash/, 2011. Zugegriffen am: 09.12.2015.

[10] Jürgen Auer. Operatoren für SQL-Abfragen. http://www.sql-und-xml.de/server-daten/sql-befehle/operatoren.html, Stand: 28.02.2015, zuletzt abgerufen: 06.08.2015.

[11] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[12] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373, 2013.

[13] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, et al. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[15] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, December 2008.

[16] Peter A Boncz, Stefan Manegold, and Martin L Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[17] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[18] Sebastian Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[19] Sebastian Breß, Norbert Siegmund, Max Heimel, Michael Saecker, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake. Load-aware inter-co-processor parallelism in database query processing. *Data & Knowledge Engineering*, 93:60 – 79, 2014. Selected Papers from the 17th East–European Conference on Advances in Databases and Information Systems.

[20] C. Brücher, F. Jüdes, and W. Kollmann. *SQL thinking: Vom Problem zum SQL-Statement*. Mitp bei Redline. mitp, 2011.

[21] Arian Bär. Mehrprozessorarchitekturen (smp, cluster, uma/numa).

[22] Clock-pro: An effective improvement of the clock replacement. `http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-05-3.pdf`, 2005.

[23] Betriebssysteme (bs) virtueller speicher. `https://ess.cs.tu-dortmund.de/Teaching/SS2011/BS/Downloads/09-Virtueller-Speicher.pdf`, 2011.

[24] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[25] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[26] Google. Google benchmark. `https://github.com/google/benchmark`. Stand Februar 2016.

[27] Google. Google test. `https://github.com/google/googletest`. Stand Februar 2016.

[28] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 102–111, 1990.

[29] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[30] Bingsheng He and Qiong Luo. Cache-oblivious nested-loop joins. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 718–727. ACM, 2006.

[31] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[32] Natalie Heller. Parsergeneratoren am beispiel von flex und bison. 2009.

[33] Algorithmen für sortierung und relationale operatoren. `https://homepages.thm.de/~hg11260/mat/algos-l.pdf`, 2014.

[34] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An architecture for recycling intermediates in a column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM.

[35] Jean-Philippe Lang. Redmine. `http://www.redmine.org/`, 2016.

[36] A. Kemper and T. Neumann. Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206, April 2011.

[37] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 456–467. Springer, 2006.

[38] H.W. Lang. Mergesort. Technical report, FH Flensburg, 2000. `http://hypepage.de/proseminar/mergesort_iti_fh_flensburg.pdf`.

[39] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[40] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[41] Lru - least recently used. `http://www.wi.fh-flensburg.de/lru_riggert.html`.

[42] S. Manegold. *Understanding, Modeling and Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam, 2002.

[43] S. Manegold, P.A. Boncz, and M.L Kersten. Generic database cost models for hierarchical memory systems. In *International Conference on Very Large Data Bases (VLDB)*.

[44] S. Manegold, M. L. Kersten, and P. A. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *Proceedings of the International Conference on Very Large Databases (VLDB, 2009)*. VLDB, August 2009. 10-year Best Paper Award for Database Architecture Optimized for the New Bottleneck: Memory Access. In Proceedings of the International Conference on Very Large Data Bases (VLDB), pp 54-65, Edinburgh, United Kingdom, September 1999.

[45] Stefan Manegold, Peter Boncz, Martin Kersten, and IEEE Computer Society. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, July/August 2002.

[46] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[47] A. Munshi, B. Gaster, and T.G. Mattson. *OpenCL Programming Guide*. Addison-Wesley, Amsterdam, 2011.

[48] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[49] Thomas Neumann and Viktor Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.

[50] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. Stand März 2015.

[51] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[52] OpenMP Architecture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013.

[53] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 2007.

[54] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 256–276, New York, NY, USA, 1984. ACM.

[55] Vasumathi Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donalcd Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 60–69. IEEE, 2008.

[56] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, August 2013.

[57] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer DE, Berlin, 3. aufl. edition, 2012.

[58] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. *Cache conscious algorithms for relational query processing*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[59] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742, 2012.

[60] SSB Data Generator. `https://github.com/electrum/ssb-dbgen`, 2010.

[61] Citus data cstore_fdw (postgresql column store) vs. monetdb tpc-h shootout. `https://www.monetdb.org/content/citusdb-postgresql-column-store-vs-monetdb-tpc-h-shootout`, 2014.

[62] MonetDB Developer Team. Monetdb. `monetdb.org/`. Stand Juni 2015.

[63] MonetDB Developper Team. Monetdb solutions. `monetdbsolutions.com/`. Stand Juni 2015.

[64] Matthias Teschner. Algorithmen und datenstrukturen - sortieren. Technical report, Universität Freiburg, 2012. `http://cg.informatik.uni-freiburg.de/course_notes/info2_09_sortieren.pdf`.

[65] Lars Tornow. Arithmetische Ausdrücke. `http://www.cfd.tu-berlin.de/Lehre/EDV1/skripte/f95_skript/node25.html`, Stand: 31.03.2003, zuletzt abgerufen: 06.08.2015.

[66] Tpc-h - top ten performance results (non-clustered). `http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster`, 2015.

[67] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing database column scans with complex predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013.*, pages 1–12, 2013.

[68] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[69] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[70] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *VLDB*, pages 186–197, 1990.

[71] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[72] Marcin Zukowski, Peter A Boncz, et al. Vectorwise: Beyond column stores. 2012.

[73] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - A DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.