TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Verteilte Verarbeitung von Joins in MapReduce

von

Nico Weisenauer

zur Erlangung des Grades
Bachelor of Science in Informatik

Betreuer:
M.Sc. Caetano Sauer
Prof. Dr.-Ing. Dr. h. c. Theo Härder

Eingereicht bei:
Fachbereich Informatik
TU Kaiserslautern

30. Oktober, 2013

Zusammenfassung

Diese Bachelorarbeit beschreibt, ausgehend von der Arbeit von Caetano Sauer [12], Verbesserungen an der Ausführung von Abfragen in BrackitMR. Der Schwerpunkt liegt dabei auf der Implementierung von verteilten Join-Verfahren. BrackitMR basiert auf der Query Engine Brackit, stützt sich bei der Ausführung von Abfragen aber auf MapReduce, ein von Google entwickeltes Programmiermodell, das Ausnutzung der größtmöglichen Parallelität erlaubt.

Das vorrangige Ziel dieser Arbeit ist es, die Ausführung von verteilten Joins effizienter zu machen. Joins stellen eine der Schwachstellen des MapReduce Modells dar, gleichzeitig spielen sie jedoch eine große Rolle in der Verarbeitung von Abfragen. In der heutigen Zeit wird häufig von „Big Data“ gesprochen, also großen Datenmengen, die natürlich verarbeitet und analysiert werden müssen. Die riesigen Mengen an Daten, die im modernen Leben im Web und in der Wissenschaft generiert werden, erfordern extreme Parallelität in der Abarbeitung. Häufig müssen diese riesigen Datenmengen auch mit einem Join in Relation gestellt werden, weswegen die Effizienz und der Verteiltheitsgrad ebendieser Joins eine wichtige Rolle spielen. Diese Bachelorarbeit soll einen Teil dazu beitragen, BrackitMR zu einer effizienten Query Engine für aktuellste Aufgabenbereiche zu machen.

Zuerst werden die Grundlagen für das Verständnis von BrackitMR geschaffen, indem wichtige Aspekte von Brackit, MapReduce und Hadoop vorgestellt werden. Dann wird im Hauptteil dieser Arbeit über verteilte Joins, BrackitMR und die Implementierung spezialisierter Join-Verfahren geschrieben.

Abstract

Based on the work of Caetano Sauer [12], this thesis introduces improvements on query execution in BrackitMR with focus on the implementation of distributed Joins. BrackitMR is based on the Brackit query engine, but makes use of the MapReduce model to achieve parallelism in the execution of tasks.

The primary goal of this thesis is to improve the execution time of distributed Joins. Joins resemble a weak spot of the MapReduce model, while at the same time they play a huge role in just about any query language. Today everyone talks of "Big Data", huge amounts of data, which are by definition too large to be analysed efficiently, generated by the web and modern science. To overcome the challenge of Big Data, extreme parallelism in the execution of queries is demanded. Often large tables need to be joined, which makes efficiency and parallelism of Joins a hot topic. This thesis should contribute to making BrackitMR an efficient query engine for modern day tasks.

First the foundation for understanding BrackitMR will be set by introducing the most important aspects of Brackit, MapReduce and Hadoop. The following parts about distributed Joins, BrackitMR and the implementation of specialized Joins form the main part of this thesis.

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit mit dem Thema "Verteilte Verarbeitung von Joins in MapReduce" selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 30.10.2013

Nico Weisenauer

Inhaltsverzeichnis

1 Einleitung 5
- 1.1 Motivation . . . 5
- 1.2 Aufgabenstellung . . . 5

2 Brackit Query Engine 7
- 2.1 XQuery . . . 7
- 2.2 Das Datenmodell von XQuery . . . 7
- 2.3 FLWOR-Ausdrücke . . . 8
- 2.4 Der Brackit Compiler . . . 10

3 MapReduce und Hadoop 13
- 3.1 Grundlagen von MapReduce . . . 13
- 3.2 Vor- und Nachteile . . . 14
- 3.3 Das generalisierte MapReduce Modell . . . 14
- 3.4 Hadoop Jobs . . . 16
- 3.5 Das Hadoop Dateisystem . . . 16
- 3.6 Der Distributed Cache . . . 17
- 3.7 Verteilte Join-Strategien . . . 18
  - 3.7.1 Reduce-Side Joins . . . 18
  - 3.7.2 Map-Side Joins . . . 19
  - 3.7.3 Komplexe Join-Verfahren . . . 20

4 BrackitMR und Erweiterungen 24
- 4.1 Operatoren . . . 24
- 4.2 Collections . . . 26
- 4.3 Skew Reduzierung . . . 27
- 4.4 Multi-Key Join . . . 29
- 4.5 Fragment Replicate Join . . . 30
- 4.6 Sort-Merge Join . . . 32

5 Leistungsanalyse 35

6 Fazit 39
- 6.1 Rekapitulation . . . 39
- 6.2 Zukünftige Entwicklungen . . . 40

Abbildungsverzeichnis

2.1 Beispiel XML Daten 9
2.2 Beispiel FLWOR-Ausdruck 9
2.3 Die vier Phasen des Brackit Compilers 11
2.4 Übersetzung eines FLWOR Ausdrucks in eine Pipeline 12

3.1 Verteilte Ausführung eines MapReduce Jobs in einem Cluster. 15
3.2 Verteilung der Tupel bei einem Reduce-Side Join 19
3.3 Verteilung der Tupel von L, M und R auf $k = m^2$ Reducer 21
3.4 Matrix/Reducer Mapping für einen Equi-Join 23

4.1 Umgeschriebener Query Plan (a) und Übersetzung in Task Functions (b) 25
4.2 Ein Ablaufdiagramm eines MapReduce Jobs. 'Exec' zeigt die eigentliche Rechendauer der Map und Reduce Tasks. 27
4.3 Multi-Key Equi-Join zwischen zwei Datensätzen L und R 29
4.4 PostJoin und FRJoin 31
4.5 Beispiel einer SMJoin Tabelle 32
4.6 Markierungen beschleunigen die Suche in der Tabelle 34

1 Einleitung

1.1 Motivation

Das weltweite Datenvolumen verdoppelt sich alle zwei Jahre, alleine im Jahre 2012 wurden geschätzte 2,8 Billionen Gigabyte neue Daten produziert [18]. Dafür ist einerseits das Internet verantwortlich, das sich immer stärker verbreitet und auf immer mehr Geräten verfügbar ist, andererseits generiert auch die moderne Wissenschaft (wie zum Beispiel die Kernphysik) immer größere Datenmengen. Teilchenbeschleuniger wie der LHC produzieren jährlich Petabyte an Daten, die analysiert und ausgewertet werden müssen, um logische Schlüsse daraus ziehen zu können. Sollen dann auch noch global sämtliche Internetaktivitäten überwacht werden, so bekommt man ein Verständnis dafür, was Big Data bedeutet.

Der Begriff „Big Data“ bezeichnet nämlich diese Datenmengen, die in der Größenordnung liegen, dass sie von aktuellen Datenbanksystemen nicht mehr effizient verarbeitet werden können. Dennoch hat die Verarbeitung von Big Data eine große wirtschaftliche Bedeutung erlangt, was durch neue Konzepte in der parallelen Datenverarbeitung ermöglicht wurde. Das wohl bekannteste und bedeutendste dieser Konzepte ist das von Google entwickelte MapReduce-Framework, das in seiner Open Source Implementierung Hadoop von zahlreichen großen Internetunternehmen wie Facebook, Amazon, Ebay und Yahoo genutzt wird.

MapReduce ermöglicht die Parallelverarbeitung von Big Data auf tausenden von günstigen Rechnern in einem Cluster und richtet sich vor allem an Programmierer mit Erfahrung in „General Purpose“ Programmiersprachen, im Gegensatz zu den traditionellen Systemen, die häufig Kenntnisse in einer domänenspezifischen Sprache (meist SQL) voraussetzen. Das macht MapReduce Systeme zu einer kostengünstigen Allroundlösung, perfekt für den Einsatz in modernen Web-Applikationen, deren Anforderungen sich von traditionellen relationalen System nicht immer so flexibel umsetzen lassen.

1.2 Aufgabenstellung

Diese Arbeit beschäftigt sich mit Brackit, einer Query Engine zur Verarbeitung von XQuery Abfragen, und deren Erweiterung BrackitMR, die Brackit so modifiziert, dass Abfragen in MapReduce Jobs übersetzt werden, die dann parallel verarbeitet werden können. Obwohl BrackitMR schon eine leistungsfähige Anwendung darstellt, wird sich der Kern dieser Arbeit damit beschäftigen, die Ausführung der Abfragen in MapReduce weiter zu optimieren. Unser Fokus liegt dabei auf der Implementierung spezialisierter Join-Verfahren.

Die Herausforderung liegt dabei darin, Join-Verfahren, die sich beim Einsatz in relationalen Datenbanksystemen bewährt haben, auf das MapReduce Modell zu übertragen. Da MapReduce nicht mit relationalen Tabellen, sondern mit Tupeln aus Key/Value-Paaren arbeitet und MapReduce-Berechnungen immer einem festgeschriebenen Ablauf folgen müssen, gilt es bei der Implementierung effizienter Join-Verfahren einige Hürden zu nehmen.

Von den auf MapReduce abgestimmten Join-Verfahren, die bisher in der Wissenschaft vorgestellt wurden, haben sich einige, wie der sogenannte Fragment Replicate Join, bereits bewährt. Dies macht sie zu optimalen Kandidaten für den Einsatz in BrackitMR. Im Laufe dieser Arbeit müssen also verschiedene verteilte Join-Strategien untersucht werden, damit solche, die sich für anwendungstypische Situationen eignen, implementiert werden können. Daraufhin sollen diese auf Performanz getestet werden.

Um die Grundlage hierfür zu schaffen, wird zuerst die Brackit Query Engine vorgestellt. Diese verarbeitet XQuery Abfragen, weswegen im gleichen Kapitel auch eine kurze Einführung in XML und XQuery gegeben wird. Das darauf folgende, dritte Kapitel handelt von MapReduce und dem freien MapReduce-Framework Hadoop. Es werden die wichtigsten Grundlagen von MapReduce und Hadoop erklärt, sodass die im nächsten Kapitel folgende Verknüpfung von Brackit und MapReduce nachvollzogen werden kann. Außerdem werden verteilte Join-Strategien präsentiert, die speziell auf den MapReduce Arbeitsablauf abgestimmt sind. Das vierte Kapitel handelt dann von BrackitMR, einer Erweiterung von Brackit, der das MapReduce-Framework zugrunde liegt, und erläutert kurz, wie sich XQuery Abfragen zur verteilten Ausführung in MapReduce Jobs übertragen lassen. An dieser Stelle werden die im Rahmen dieser Arbeit implementierten Erweiterungen von BrackitMR vorgestellt, wobei es sich hauptsächlich um situationsbedingte Verbesserungen in der Ausführung von verteilten Joins handelt. Im fünften Kapitel werden diese Erweiterungen einem Praxistest unterzogen, wobei der Fokus auf situationsbedingten Performancegewinnen liegt. Abschließend fasst das Fazit die Ergebnisse dieser Arbeit zusammen.

2 Brackit Query Engine

Brackit bildet die Basis für BrackitMR und somit auch für die im Rahmen dieser Arbeit durchgeführten Verbesserungen. Brackit ist eine Query Engine, die sich auf XQuery stützt und deren Stärke auf größtmöglicher Flexibilität beruht. Verschiedene Erweiterungen erlauben Brackit somit auch die Unterstützung von JSON *arrays* und CSV Dateien. Um die Grundlagen von Brackit verständlich zu machen, muss zuerst die Abfragesprache XQuery vorgestellt werden.

2.1 XQuery

XQuery ist weit mehr, als nur eine reine Abfragesprache, wie der Name vermuten lässt. Obwohl die Wurzeln von XQuery in XML und XPath liegen und es eigentlich entwickelt wurde, um ebendiese XML Dokumente abzufragen, hat es sich im Laufe der Zeit zu einer Turing-vollständigen und flexiblen Sprache entwickelt, wie auch das Vorbild SQL. Das liegt unter anderem am großen Erfolg von XML, das schemalose Daten akzeptiert und somit die Basis für eine flexible Abfragesprache bildet.

Wegen dieser häufig unterschätzten Mächtigkeit von XQuery, ist es eine Sprache, die man sowohl als Ersatz für SQL, in der Analyse von großen Datenmengen (Big Data), als auch in unzähligen anderen Bereichen der Informatik nutzen kann. Es fällt also nicht schwer den Schluss zu ziehen, dass XQuery auch geeignet ist, um parallele MapReduce Berechnungen auszuführen. Bevor wir allerdings dazu kommen, sollten wir uns kurz mit den Grundlagen von XQuery beschäftigen. Für eine vollständige Einführung in XQuery sei hier auf den W3C XQuery Standard verwiesen [15].

2.2 Das Datenmodell von XQuery

Im Folgenden wird nur grob zusammengefasst, wie das Datenmodell von XQuery aussieht. Für eine ausführliche Erklärung stellt das W3C eine Definition des XDM (XQuery and XPath Data Model) zur Verfügung [16].

Immer wenn ausgeführt wird, wie das XQuery Datenmodell aufgebaut ist, fällt die Aussage, dass in XQuery alle Datenstrukturen als Sequenz (geordnete Liste) betrachtet werden. Diese können völlig leer sein, komplette XML Dokumente enthalten oder auch Duplikate, allerdings nie eine weitere Sequenz. Verschachtelte Sequenzen werden also eigentlich nicht akzeptiert. Betrachtet man eine Sequenz, die einen einzelnen Wert enthält, so gibt es semantisch keinen Unterschied zwischen der Sequenz an sich und dem einzelnen Wert, obwohl Sequenzen geklammert werden. Da Sequenzen nicht geschachtelt

werden dürfen, ist eine Sequenz, die eine weitere geklammerte Sequenz enthält, also einfach eine große Sequenz, deren innere Klammern aufgelöst werden. Das ist auch wichtig für spätere Betrachtungen in Brackit.

Es wurde bereits erwähnt, dass eine Sequenz ganze XML Dokumente enthalten kann. Ein XML Dokument ist also ein Datentyp, der in einer Sequenz auftreten darf, spezifischer ist es ein Knoten (engl. "node"). Neben den Knoten dürfen auch atomare Werte, sogenannte `Atomics`, auftreten, diese umfassen die primitiven Datentypen wie `integer` und `string`. Die Knoten unterteilen sich nochmals in die Typen `Dokument`, `Element`, `Attribut`, `Text`, `Namensraum`, `Verarbeitungsanweisung` und `Kommentar`, von denen uns aber nur die ersten Vier näher interessieren.

Betrachten wir die Abbildung 2.1 als Beispiel. Wie der Name schon sagt, stehen `Dokument`-Knoten für komplette XML Dokumente, die durch genau einen `Element`-Knoten auf oberster Ebene definiert sind (die Wurzel des XML Baums). Im Beispiel wäre diese Wurzel das Element *adressbuch*. Ein `Element`-Knoten kann beliebig viele weitere `Element`-Knoten beinhalten, wie man im Beispiel anhand der zwei *adresse* Elemente sieht. Die `Attribut`-Knoten beschreiben die `Element`-Knoten genauer und enthalten deswegen Informationen als Key-Value Paare, in der Abbildung finden sich Attribute als Zusatz zu den *tel* Elementen, hier wird die Art des Anschlusses in privat und öffentlich unterschieden. `Text`-Knoten enthalten dagegen nur Benutzerdaten in reinem Fließtext, der als `Atomic` gehandelt wird, solange kein Schema einen anderen Typ bestimmt, im Beispiel finden wir Text zwischen den öffnenden und schließenden Tags der Elemente *name*, *vorname*, *strasse*, *plz*, *ort* und *tel*.

Mithilfe dieser vier verschiedenen Knotentypen lassen sich alle XML Daten modellieren, die für Brackit und BrackitMR relevant sind.

2.3 FLWOR-Ausdrücke

Eine Abfrage in XQuery besteht in der Regel aus einem sogenannten FLWOR-Ausdruck, die Anfangsbuchstaben stehen dabei für **for**, **let**, **where**, **order by** und **return**. Im Grunde ist so ein Ausdruck einem SELECT-Statement in SQL ähnlich.

Die Auswertung jedes der fünf Konstrukte in einem FLWOR-Ausdruck hängt vom sogenannten Kontext ab, einem Tupel, das man vom vorherigen Konstrukt erhält, sequentiell verarbeitet und an das nächste Konstrukt weiterleitet. Dieses Tupel enthält die Bindung freier Variablen, wobei jede Position des Tupels einer Variable zugeordnet ist. FLWOR-Ausdrücke können dadurch auch beliebig geschachtelt werden und generell überall dort als Argument gegeben werden, wo ein beliebiger Ausdruck erwartet wird. Das Ganze ist vergleichbar mit der Listenverarbeitung in funktionalen Programmiersprachen, wie zum Beispiel Haskell, wo die entsprechenden Funktionen eine ganze Liste von Tupeln als Eingabe erhalten. Da auch MapReduce seine Wurzeln in der funktionalen Listenverarbeitung hat [8], wird hier schon deutlich, dass BrackitMR sich diesen Zusammenhang zwischen XQuery und MapReduce zunutze macht. Im Folgenden werden wir uns die fünf Teile eines FLWOR-Ausdrucks genauer anschauen.

```
<adressbuch>
  <adresse>
    <name>Mueller</name>
    <vorname>Bob</vorname>
    <strasse>Friedenstr. 10</strasse>
    <plz>10000</plz>
    <ort>Musterdorf</ort>
    <tel art="priv">12345</tel>
    <tel art="off">54321</tel>
  </adresse>
  <adresse>
    <name>Meier</name>
    <vorn>Alice</vorn>
    <strasse>Bahnhofstr. 15</strasse>
    <plz>10000</plz>
    <ort>Musterdorf</ort>
    <tel art="priv">56789</tel>
    <email>ameier@provider.net</email>
  </adresse>
</adressbuch>
```

Abbildung 2.1: Beispiel XML Daten

```
for $t in doc('beispiel.xml')/adressbuch
let $y := $t/adresse/tel/@art
where $y = "priv"
order by $y
return
  <result>
    <privatnummer>$y</privatnummer>
  </result>
```

Abbildung 2.2: Beispiel FLWOR-Ausdruck

- Die wohl wichtigste Klausel ist die **for**-Klausel, da sie einen Ausdruck als Argument bekommt und diesen in einzelne Elemente herunterbricht. Für jedes dieser Elemente x nimmt sich die **for**-Klausel das Eingabe Tupel (was durchaus bereits gebundene Variablen enthalten kann, im Falle von geschachtelten Ausdrücken) und hängt eine Variable, die an das aktuelle Element x gebunden ist, an. Dieser neue, vergrößerte Kontext wird dann an die nachfolgenden Klauseln gesendet und ist mit der aktuellen Iteration assoziiert. So entsteht eine *Pipeline* an Tupeln innerhalb des FLWOR-Ausdrucks, ähnlich wie bei einem SCAN-Operator einer Datenbank.

- Die **let**-Klausel ist dazu da, weitere Ausdrücke an eine Variable zu binden, allerdings wird dabei der gesamte Ausdruck an die Variable gebunden und nicht die einzelnen Elemente nacheinander, wie bei der **for**-Klausel.

- Ähnlich dem WHERE in SQL filtert die **where**-Klausel eines FLWOR-Ausdrucks Tupel nach einem gegebenen Boolschen Ausdruck. Nur wenn dieser für das gegebene Eingabe Tupel zu *true* ausgewertet wird, wird das Tupel an die nächste Klausel weitergegeben.

- Auch die **order by**-Klausel funktioniert so wie in SQL, indem sie Tupel auf Basis eines gegebenen Ausdrucks sortiert. Hierzu sei gesagt, dass auch an dieser Stelle ein kompletter FLWOR-Ausdruck als Argument gegeben werden kann.

- Zuletzt muss immer eine **return**-Klausel folgen. Diese gibt als einzige Klausel eine Gesamtergebnis-Sequenz zurück, statt des Kontextes. Indem sie einen gegebenen Ausdruck für jedes einzelne Eingabetupel auswertet, wird diese Sequenz generiert. Häufig wird der **return**-Klausel ein Konstruktor Ausdruck gegeben, um aus den berechneten Werten direkt XML Elemente zu machen (ähnlich zu SQL, wo die Ergebnisse einer Abfrage in eine Tabelle gepackt werden).

Abbildung 2.2 zeigt einen Beispiel FLWOR-Ausdruck, der sich auf die XML Daten aus Abbildung 2.1 bezieht. Dieser FLWOR-Ausdruck liefert eine Ergebnissequenz zurück, die alle Telefonnummern von Adressen innerhalb des Adressbuchs enthält, deren *art* Attribut den Wert *priv* hat. Diese Telefonnummern, die während der Auswertung nacheinander an die Variable $\$y$ gebunden sind, werden in ein *privatnummer*-Element verpackt, das wiederum in ein *result*-Element verpackt wird. Wertet man die Abfrage korrekt aus, dann entstehen hier also insgesamt zwei *result*-Elemente.

2.4 Der Brackit Compiler

Brackit vollzieht die Kompilierung eines XQuery Ausdrucks in vier Phasen, welche in Abbildung 2.3 zu sehen sind.

In der ersten Phase kümmert sich ein Parser um die Übersetzung des Strings, der die XQuery Abfrage repräsentiert. Dieser wird in einen sogenannten Abstract Syntax Tree (kurz AST) umgewandelt, also in eine baumförmige Repräsentation der Abfrage,

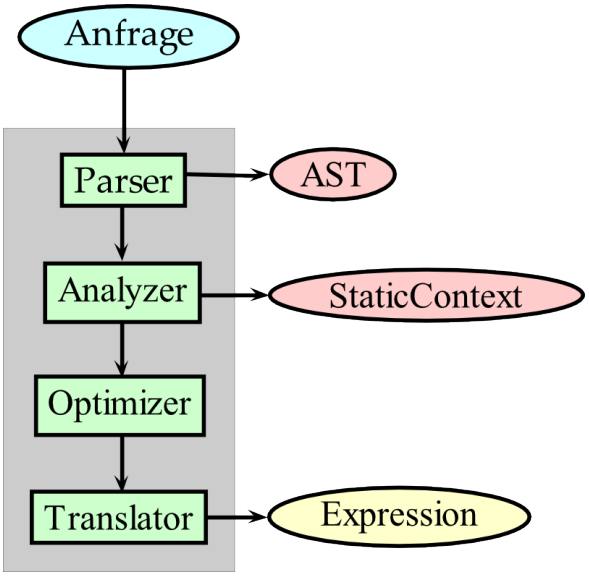


Abbildung 2.3: Die vier Phasen des Brackit Compilers

aus der die folgenden Phasen die Query-Syntax ablesen können. Syntaktisch inkorrekte Abfragen werfen an dieser Stelle eine entsprechende Fehlermeldung.

Die nächste Phase vollzieht die semantische Analyse. Ein Analyzer erstellt einen statischen Kontext, in dem zum Beispiel die Abhängigkeiten zu extern definierten Funktionen verzeichnet sind und prüft, ob die Semantik der Abfrage korrekt ist.

In der dritten Phase werden von einem Optimizer regelbasierte Optimierungen durchgeführt. Anhand einer gegebenen Liste an Optimierungsmöglichkeiten durchsucht der Optimizer den AST. Er durchläuft jeden Teilbaum und sucht nach bestimmten Mustern, beispielsweise einem Join. Wird ein solches Muster entdeckt, so wird der Baum nach vordefinierten Regeln umgeformt, wodurch sich bei der Ausführung der Abfrage ein Kostenvorteil ergeben kann.

Zuletzt übersetzt der Translator den AST in einen Ausdruck, der ein Java Objekt mit einer Methode zur Auswertung darstellt. Dieser Ausdruck gleicht semantisch einem XQuery-Ausdruck, durch Aufrufen der `evaluate()` Methode wird ein Iterator zurückgeliefert, mit dem man Tupel für Tupel der Ergebnissequenz anfordern kann.

FLWOR-Pipelines

Wie man in der Abbildung 2.4 sieht [12], bildet ein FLWOR-Ausdruck eine Pipeline, in der Kontexttupel gemäß dem Iterator-Modell [7] weitergereicht und verarbeitet werden. Beim Kompilieren wird also ein FLWOR-Ausdruck nach dem Parsen in einen Pipeline-Ausdruck umgeformt, eine sogenannte *PipeExpr*, die den gesamten FLWOR-Ausdruck repräsentiert. Dabei werden die verschiedenen Klauseln durch entsprechende Operatoren ersetzt und in ihrer Reihenfolge aneinandergehängt, um die Pipeline zu formen. Neben den Operatoren *ForBind*, *LetBind*, *Select* und *Sort*, welche die **for**-, **let**-, **where**- und

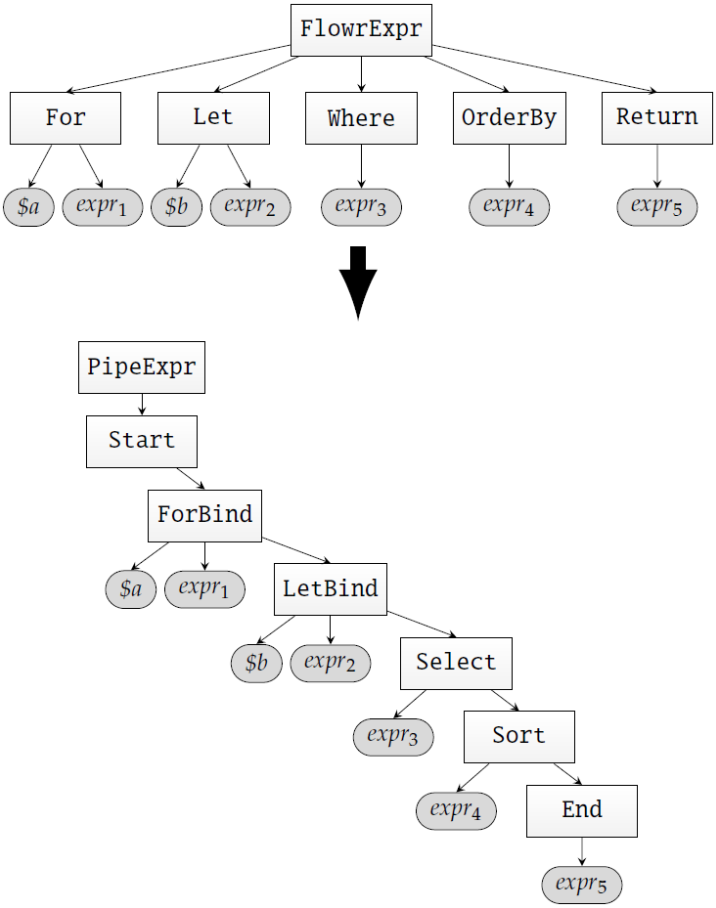


Abbildung 2.4: Übersetzung eines FLWOR Ausdrucks in eine Pipeline

order by-Klauseln repräsentieren, enthält jede Pipeline noch einen *Start*-Operator, der den leeren Kontext erzeugt, und einen *End*-Operator, der der **return**-Klausel entspricht und somit eine Sequenz als Ergebnis liefern muss. Der *End*-Operator beendet die Pipeline und liefert seine Ergebnisse an die *PipeExpr*.

3 MapReduce und Hadoop

MapReduce ist ein Programmiermodell, das die parallele Verarbeitung von großen Datenmengen, sprich Big Data, ermöglicht [5]. Dies geschieht durch parallele Ausführung der namensgebenden Funktionen *Map* und *Reduce* auf einem Rechencluster.

MapReduce wird als besonders einfach und kostengünstig beworben, da einerseits einfache Programmierkenntnisse ausreichen, um die *Map* und *Reduce* Funktionen zu spezifizieren, und andererseits der Cluster aus kostengünstiger Hardware bestehen kann. Außerdem ist MapReduce relativ fehlertolerant und kann mit Hardwareausfällen gut umgehen, solange es genügend funktionierende Rechner im Cluster gibt. Um dem Programmierer den Umgang mit dem System weiter zu vereinfachen, kümmert sich das MapReduce-Framework um jegliche Partitionierungs-, Scheduling-, Fehlerbehandlungs- und Kommunikationsaufgaben.

3.1 Grundlagen von MapReduce

Wie schon erwähnt, besteht das zentrale Konzept des Operator-Modells von MapReduce aus einer *Map* und einer *Reduce* Funktion. Diese sind inspiriert von den gleichnamigen Funktionen in funktionalen Programmiersprachen, unterscheiden sich aber in einigen Punkten von diesen. Dazu sei auf eine Publikation von R. Laemmel verwiesen, in der die Unterschiede im Detail erläutert werden [8].

Die *Map* Funktion von MapReduce iteriert über eine gegebene Eingabe von Key/Value-Paaren und berechnet ein Zwischenergebnis. Auf jedes einzelne Tupel der Eingabe wird also die Funktion angewandt. Danach wird das Zwischenergebnis von *Reduce* benutzt, vorher müssen die Tupel allerdings nach Schlüssel sortiert und gruppiert werden. Alle Tupel mit gleichem Schlüssel werden von der *Reduce* Funktion also als Liste verarbeitet. Was genau in der Verarbeitung geschieht wird, wie auch bei *Map*, vom Programmierer festgelegt.

$$\text{Map: } (K_1, V_1) \rightarrow [(K_2, V_2)]$$
$$\text{Reduce: } (K_2, [V_2]) \rightarrow [(K_3, V_3)]$$

Um große Datenmengen effizient verarbeiten zu können, muss die Eingabe eines MapReduce Jobs partitioniert werden. Außerdem muss es viele Rechner im Cluster geben, die das Ergebnis von *Map* und *Reduce* berechnen, diese nennt man entsprechend Mapper und Reducer.

Viele verschiedene Mapper erhalten also einen Teil der Eingabe und verarbeiten diese parallel. Nachdem der letzte Mapper mit seiner Arbeit fertig ist, werden die Ergebnisse aller Mapper gesammelt und sortiert, um danach den Reducern zugeteilt zu werden. Ein Reducer bekommt immer eine Liste mit allen Tupeln, die den Schlüssel enthalten, für den der Reducer verantwortlich ist (natürlich kann er auch für mehrere oder sogar alle Schlüssel verantwortlich sein). Sind die Reducer mit ihrer Verarbeitung fertig, schreiben sie ihre Ergebnisse in das Dateisystem und der MapReduce Job ist beendet. Wichtig hierbei ist, dass ein MapReduce Job immer aus den drei Phasen Map, Sort und Reduce besteht, was in einigen Szenarien einen deutlichen Performanceverlust bedeutet.

Neben den Mappern und Reducern gibt es noch den sogenannten Master, der sich um die fehlerfreie Ausführung und Koordination des Jobs kümmert. Erwähnenswert ist hierbei, dass der Master einen *Single Point of Failure* darstellt [5].

3.2 Vor- und Nachteile

Generell kann man sagen, dass MapReduce eine günstige Möglichkeit bietet, große Datenmengen parallel zu verarbeiten. Entwickler brauchen keine speziellen Vorkenntnisse, um die *Map* und *Reduce* Funktionen zu erstellen, solange sich die gewünschten Berechnungen auf dieses einfache Modell abbilden lassen. Sollen allerdings Aufgaben ausgeführt werden, die sich nicht eins zu eins auf den MapReduce Ablaufplan übertragen lassen, dann stößt MapReduce an seine Grenzen und es müssen zum Beispiel „leere“ Mapper ausgeführt werden, die Daten unnötig lesen und schreiben und damit die Effizienz einschränken.

Es gibt einige Ansätze, die versuchen, MapReduce für spezielle Aufgabenbereiche anzupassen, wie etwa Map-Reduce-Merge [19], das eine *Merge* Phase an den MapReduce Ablauf anhängt, um besser mit Joins umgehen zu können, aber generell erhöhen solche Erweiterungen natürlich immer die Gesamtkomplexität. Um die Berechnungen in BrackitMR darzustellen, berufen wir uns auf das sogenannte generalisierte MapReduce Modell, das im Folgenden vorgestellt wird.

3.3 Das generalisierte MapReduce Modell

Ein einfacher MapReduce Job sieht also so aus, dass die Eingabe in m Splits geteilt wird, die dann von m Mappern verarbeitet werden. Diese m verschiedenen Mapper können auf $n <= m$ Rechner verteilt werden. Ihre Ergebnisse werden anschließend von r Reducern benötigt, die auf den gleichen Rechnern ausgeführt werden, wie die r dazugehörigen Shuffles. Abbildung 3.1 zeigt die Ausführung eines solchen Jobs mit $m = 4$, $n = 3$ und $r = 3$. Im Folgenden soll jedoch ein spezielles MapReduce Modell [14] vorgestellt werden, das den Workflow eines MapReduce Jobs sowohl generalisiert als auch vereinfacht und damit die Basis für BrackitMR darstellt.

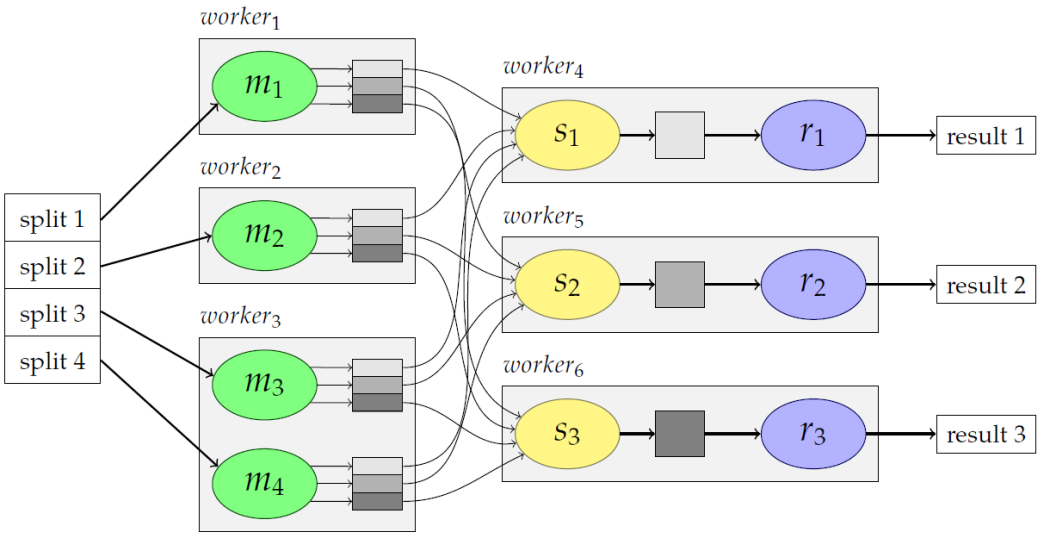


Abbildung 3.1: Verteilte Ausführung eines MapReduce Jobs in einem Cluster.

Generalisierungen

- Wenn man sich *map* und *reduce* in funktionalen Sprachen genauer anschaut, stellt man fest, dass Mapper und Reducer beide auf das funktionale *map* zurückzuführen sind, sich im Grunde also nur in ihrem Ein- und Ausgabeverhalten unterscheiden. Abstrahiert man davon, ergibt sich die Modellierung eines MapReduce Jobs als Map-Shuffle-Map, was die erste Generalisierung darstellt.

- Um bestimmte Operationen durchzuführen, ist es manchmal notwendig, mehrere Shuffle-Phasen innerhalb einer Berechnung zu haben. Dies würde bei dem Modell, was durch die erste Generalisierung entsteht, dazu führen, dass zwei Map-Phasen zwischen den Shuffles aufeinander folgen, wovon eine unnötig Daten in das verteilte Dateisystem schreibt. Deswegen erlaubt das generalisierte MapReduce Modell, die zweite Map-Phase wegzulassen, sodass eine Berechnung aus n Map-Phasen und $n-1$ Shuffles zusammengesetzt sein kann.

- Das generalisierte MapReduce Modell ermöglicht schon innerhalb eines Mappers Parallelverarbeitung in Form von Pipelining anzuwenden. Hierbei wird von der Verarbeitung eines Splits im Ganzen abstrahiert und statt von *Map* oder *Reduce* Funktionen wird von *Tasks* gesprochen. Diese ermöglichen innerhalb eines Mappers, durch Anwendung des Iterator Modells, Parallelität in der Abarbeitung zu erreichen.

- Die letzte Generalisierung erlaubt es Mappern mehrere Eingaben zu besitzen, denen jeweils eine Task-Funktion zugeordnet ist. Die Shuffle-Phase kombiniert dann

eine Teilmenge der verschiedenen Ausgaben. Binäre Operatoren, wie etwa Joins, die im normalen MapReduce Ablauf schwierig umzusetzen sind, sind dadurch viel einfacher zu modellieren.

Da eine Shuffle-Phase immer zwischen zwei Map-Phasen ausgeführt wird und ihr immer die gleiche Funktion zugrunde liegt, kann man sie in einem Job-Graph auch weglassen, was es uns erlaubt, Query Pläne sehr übersichtlich in MapReduce Jobs zu überführen. Abschließend sei erwähnt, dass das generalisierte MapReduce Modell vollständig in *Hadoop* simuliert werden kann, also nicht mächtiger, als das eigentliche MapReduce Modell ist. Dadurch, dass aber in gewissem Maße vom strikten MapReduce Ablaufplan abstrahiert wird und Konzepte wie das Iterator-Modell ausgenutzt werden, erleichtert das generalisierte Modell die Übertragung von SQL und XQuery Abfrageplänen auf MapReduce.

3.4 Hadoop Jobs

Hadoop ist ein weit verbreitetes, in Java geschriebenes Open Source MapReduce-Framework und findet auch bei BrackitMR Anwendung.

Um einen Hadoop Job in Java auszuführen, sind mehr Schritte nötig, als nur die *Map* und *Reduce* Funktionen zu implementieren. Das BrackitMR-Framework erledigt diese Aufgaben für uns und stellt den gesamten Kontext für die Berechnung bereit. Zu jeder Abfrage startet BrackitMR einen oder mehrere Jobs, deren Durchführung wir durch eine Konfigurationsdatei beeinflussen können.

Zu jedem Hadoop Job gehört ein entsprechendes Job Objekt, das Informationen darüber enthält, welche Klassen als Mapper und Reducer verwendet werden, wie die Eingabe und Ausgabe aussieht, wo man die Eingabe findet und wo man die Ausgabe hinschreibt. Weitere relevante Parameter können aus der Konfigurationsdatei geladen werden, etwa wie viele Mapper und Reducer ausgeführt werden sollen.

Um das generalisierte MapReduce Modell in Hadoop zu simulieren, müssen oft sogenannte *IdentityMapper* und *EmptyReducer* benutzt werden, die Daten unverändert weitergeben. Das verursacht zusätzliche Kosten, da die Daten häufig ins Dateisystem geschrieben werden. Hier könnten zukünftig weitere Optimierungen vorgenommen werden, die Hadoop dahingehend modifizieren, dass das generalisierte MapReduce Modell nativ unterstützt wird und somit unnötige Mapper und Reducer entfallen.

3.5 Das Hadoop Dateisystem

Das Hadoop Dateisystem (kurz HDFS) ist ein verteiltes Dateisystem, das darauf ausgelegt ist, große Dateien in einem Computercluster zu verwalten. Seine Stärken liegen in der Fehlertoleranz und in der Fähigkeit, Streaming auf großen Dateien zu ermöglichen. Von Interesse für uns ist hier lediglich das Java Interface, White allerdings bietet eine umfassende Einführung in das HDFS und generell Hadoop [17].

Kenntnisse über das Hadoop Dateisystem sind von großer Bedeutung für die später präsentierten Optimierungen in BrackitMR, da das verteilte Dateisystem einerseits zu einem großen Teil für die Ausführungsgeschwindigkeit eines Jobs verantwortlich ist und andererseits zur Verifikation der Änderungen an BrackitMR genutzt werden muss.

Dateien lesen

Im Gegensatz zur aus Java bekannten Repräsentation von Dateien als `File`, werden in Hadoop Dateien als `Path` deklariert, um sich vom lokalen Dateisystem zu distanzieren. Um die Daten aus solch einem `Path` zu lesen, muss man ihn mithilfe eines `FileSystem` Objekts öffnen. Dieses Objekt kann man sich mit verschiedenen statischen Methoden der dazugehörigen Klasse besorgen, dabei wird eine Konfiguration als Parameter übergeben, die beim Erstellen des dazugehörigen Jobs mitgeneriert wurde. Diese Konfiguration enthält auch die Informationen darüber, ob das Dateisystem lokal oder verteilt besteht. Öffnet man nun also den Pfad mit der entsprechenden Methode von `FileSystem`, so erhält man einen `FSDataInputStream` als Rückgabe. Dieser unterstützt Random Access und ermöglicht es anschließend die Dateien auszulesen.

Dateien schreiben

Um Dateien in das Dateisystem zu schreiben, kann man der `create()` Methode von `FileSystem` ein `Path` Objekt geben und erhält einen `FSDataOutputStream` zurück, also das Pendant zum `FSDataInputStream`, das es einem erlaubt, Dateien zu schreiben. Die Besonderheit des `FSDataOutputStream` liegt in der Möglichkeit, die aktuelle Position abzufragen. Allerdings müssen Dateien immer an der letzten Position weitergeschrieben werden. Hat man den `Path` einer Datei, kann man mit der `append()` Methode von `FileSystem` diese Datei auch mit dem zurückgegebenen `FSDataOuputStream` weiterschreiben.

Informationen abfragen

Um Informationen über das Dateisystem und den darin gespeicherten Dateien zu erhalten, stellt die `FileSystem` Klasse eine Methode `listStatus(Path)` bereit. Diese liefert ein Liste von `FileStatus` Objekten zurück, die mit dem übergebenen Pfad assoziiert ist. Die einzelnen `FileStatus` Objekte enthalten alle wissenswerten Informationen über die dazugehörigen Daten.

3.6 Der Distributed Cache

Wenn man zur Ausführung eines Jobs auf zusätzliche Daten angewiesen ist, die von allen Tasks gelesen werden sollen, so kann man den Distributed Cache [17] von Hadoop nutzen. Werden auf einem Rechner im Cluster mehrere Tasks durchgeführt, so können alle auf die Daten, die durch den Distributed Cache verteilt wurden, zugreifen. Ohne dieses Feature müssten die Dateien vom HDFS an die jeweiligen Tasks gesendet werden, was zusätzliche

Netzwerkbandbreite verschwenden würde. Es können auch Archive und Jar-Dateien in den Distributed Cache geladen werden, die dann erst auf den Ziel-Rechnern entpackt werden, was weitere Netzwerkkosten spart.

Der Distributed Cache eröffnet uns die Möglichkeit, einen sogenannten Map-Side Join durchzuführen, also eine Reduce-Phase beim Durchführen eines Joins auf zwei Datensätzen zu sparen. Genauere Erläuterungen folgen im Unterkapitel 3.7.

Funktionsweise

Um den Distributed Cache zu nutzen, muss man vor der Ausführung des Jobs festlegen, welche Daten in den Cache geladen werden sollen. Dies geschieht mithilfe der entsprechenden `DistributedCache` Klasse, welche die Methode `addCacheFile()` bereitstellt. Hadoop kopiert dann, kurz nachdem der Job gestartet wurde, alle spezifizierten Dateien auf die einzelnen Rechner. Diese Daten befinden sich dann im lokalen Dateisystem des jeweiligen Rechners, was die maximale Größe des Distributed Cache auf die Größe des kleinsten Speichers der einzelnen Cluster Rechner limitiert. Außerdem muss bedacht werden, dass auch noch andere Programmteile den Speicher dieser Rechner in Anspruch nehmen können, weswegen man den Distributed Cache nicht überbeanspruchen sollte.

Soll dann innerhalb eines Tasks auf Dateien des Distributed Cache zugegriffen werden, dann müssen zuerst die neuen Pfade der Dateien beschafft werden, schließlich wurden sie beim Starten des Jobs in das lokale Dateisystem der Rechner verschoben. Dazu besorgt man sich von dem `DistributedCache` Objekt eine Liste der aktuellen Pfade, was durch Aufruf der `getCacheFiles()` Methode geschieht.

3.7 Verteilte Join-Strategien

Es gibt zahlreiche Methoden, wie Joins in MapReduce ausgeführt werden können. Dabei wird grundsätzlich zwischen zwei verschiedene Klassen von verteilten Joins unterschieden, den sogenannten Reduce-Side Joins und den Map-Side Joins [17]. Abhängig von der Ausgangslage, also ob und wie viele Informationen über die Datensätze vorliegen und ob diese eventuell schon sortiert sind, muss man sich für eine Join-Strategie entscheiden, oder bestenfalls das Programm entscheiden lassen. Es gibt keine allgemeine Methode, die alle Joins effizient durchführt, deswegen sollten in jeder Situation die vorhandenen Möglichkeiten abgewägt werden.

3.7.1 Reduce-Side Joins

Wenn keine Informationen über die Datensätze vorliegen, zwischen denen ein Join durchgeführt werden soll, so stellt ein Reduce-Side Join die einzige Möglichkeit dar, da er allgemein anwendbar ist. Ein Reduce-Side Join besteht aus einem vollen MapReduce Job, also Map-Shuffle-Reduce. In der Map-Phase werden die zwei Datensätze mit einem *Tag* versehen, wodurch man sie nach dem Shuffle auseinanderhalten kann. Beim Zuteilen zu den Reducern wird das *Tag* ignoriert, damit die zueinander passenden Tupel

beider Datensätze beim gleichen Reducer landen. Beim Shuffle wiederum wird das *Tag* genutzt, um alle Tupel des einen (kleineren) Datensatzes vor den Tupeln des anderen zu platzieren, was wichtig für die effiziente Weiterverarbeitung ist.

In der Abbildung 3.2 ist zu sehen, wie zwei Eingaben eines Reduce-Side Joins auf vier Reducer verteilt werden. In der Darstellung wurde allen Tupeln der linken Eingabe das Join-*Tag* 0 und allen Tupeln der rechten Eingabe das Join-*Tag* 1 vorangestellt. Die Buchstaben symbolisieren den Wert des Join-Schlüssels der jeweiligen Tupel, Tupel mit gleichen Buchstaben und verschiedenen Ziffern sollen also miteinander gejoint werden.

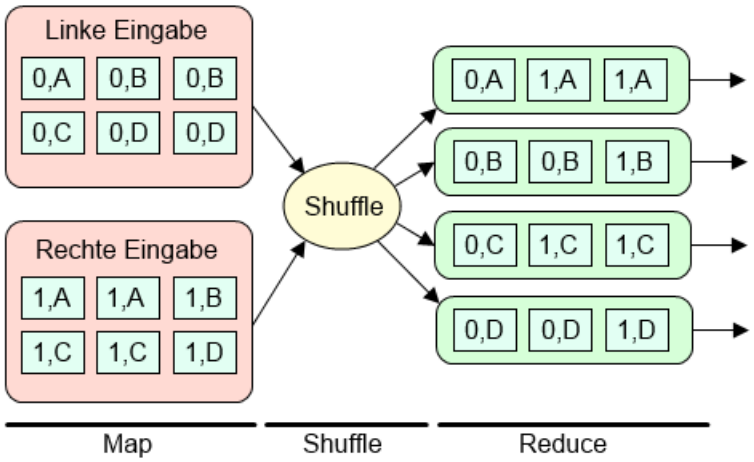


Abbildung 3.2: Verteilung der Tupel bei einem Reduce-Side Join

Im Reducer findet dann der eigentliche Join statt. Hier können verschiedene Strategien Anwendung finden, wobei allgemein ein Hash-Join sinnvoll ist. Man liest hierbei zuerst alle Tupel des ersten Datensatzes, was durch die Sortierung nach *Tag* vereinfacht wird, und baut eine entsprechende Hash Tabelle auf (den Schlüssel der Tabelle stellt das Join Attribut dar). Dann werden die restlichen Tupel durchlaufen und in der Tabelle nach eventuellen Join Partnern gesucht, um schließlich die Ausgabetupel zu erzeugen. Wenn sichergestellt werden kann, dass die Tupel korrekt sortiert und partitioniert sind, so genügt es, die zwei Datensätze zu separieren und ein Kreuzprodukt auszuführen. Man spricht dann von einem einfachen Repartition Join [3] oder Sort-Merge Join.

3.7.2 Map-Side Joins

Reduce-Side Joins haben den Nachteil, das beide Datensätze durch die Shuffle Phase müssen und somit viel Zeit durch das Sortieren und Senden der Daten über das Netzwerk verloren geht. In bestimmten Situationen kann man durch Map-Side Joins einen Performancegewinn erreichen, da hier die Daten schon in der Map-Phase gejoint werden und Shuffle sowie Reduce entfallen.
Eine solche Situation entsteht zum Beispiel, wenn einer der beiden Datensätze klein genug ist, um in den lokalen Speicher der einzelnen Rechner zu passen. Ist dies der

Fall, kann der häufig als Fragment Replicate Join oder Broadcast Join [3] bezeichnete Map-Side Join angewandt werden. Dabei wird der kleine Datensatz schon vor dem eigentlichen Map in das lokale Dateisystem der Rechner im Cluster geladen, die dann eine Hash Tabelle aufbauen. Die Verteilung der Daten kann auf zwei Wegen erfolgen, mittels der Job Configuration oder mittels des Hadoop Distributed Cache. Allgemein wird aber vom Benutzen der Job Configuration abgeraten, da deren Inhalt von vielen Prozessen gelesen wird, die nichts mit dem eigentlichen Join zu tun haben. In der Map-Phase verarbeitet dann jeder Mapper seinen Teil der großen Tabelle (Fragment) und führt den Join mit den Tupeln der lokalen kleinen Tabelle (Replicate) durch, daher der Name Fragment Replicate Join. Man kann das Verfahren weiter optimieren, indem man jeweils aus dem kleineren der beiden Datensätze, also dem Split der großen Tabelle oder der verteilten kleinen Tabelle, die Hash Tabelle aufbaut.

Eine andere Möglichkeit einen Map-Side Join durchzuführen ergibt sich, wenn die beiden Datensätze als Ausgabe eines vorherigen MapReduce Jobs entstanden sind [17]. Konkret bedeutet dies, dass die Datensätze nach ihrem Join Attribut sortiert sind und so partitioniert sind, dass sich alle Tupel mit gleichem Join Attribut in der gleichen Partition befinden und es gleich viele Partitionen in beiden Datensätzen gibt. Ist diese Voraussetzung erfüllt, kann man einen Map-Side Join unabhängig von der Größe der beiden Eingaben ausführen, wobei hier der eigentliche Join schon vor der Ausführung der Map Funktion angewandt wird.

3.7.3 Komplexe Join-Verfahren

Mit komplexen Join-Verfahren sind solche Verfahren gemeint, die nicht innerhalb eines einzigen MapReduce Jobs durchführbar sind oder sich aus anderen Gründen nicht in die obigen beiden Klassen einordnen lassen, aber in bestimmten Situationen einen Performancegewinn bedeuten. Diese Join-Strategien wurden nicht im Rahmen dieser Arbeit implementiert, dennoch ist eine kurze Vorstellung angebracht, da eine zukünftige Erweiterung des BrackitMR-Frameworks auf ebendiesen Strategien beruhen könnte.

Semi-Join

Im Kontext von Big Data kann es passieren, dass beim Join zweier Tabellen viele Tupel der einen Tabelle gar keine Rolle spielen, da sie in der anderen Tabelle kein einziges Mal referenziert werden. Diese Situation entsteht zum Beispiel, wenn eine sehr große Kundentabelle L vorhanden ist und diese mit einer kleineren Tabelle R gejoint werden soll, die die Aktivitäten (wie etwa Bestellungen) in einem bestimmten Zeitfenster dokumentiert. Alle Kunden, die in diesem Zeitfenster keine Aktion getätigt haben, spielen für den Join keine Rolle. In diesem Szenario stellt der sogenannte Semi-Join eine effiziente Alternative dar, der aus drei aufeinanderfolgenden MapReduce Jobs besteht [3].

Im ersten Job wird die Map-Phase genutzt, um alle verschiedenen Werte des Join Attributs aus der kleineren Tabelle R zu extrahieren. In der Reduce-Phase werden diese dann gesammelt und in eine Datei geschrieben, die klein genug sein sollte, um in den lo-

kalen Speicher der Rechner zu passen. Im nächsten Job wird eine Art Fragment Replicate Join durchgeführt, wobei die eben genannte Datei mit den verschiedenen Join Attribut Werten in den Cache geladen wird und auf den jeweiligen Rechnern eine Hash Tabelle daraus erstellt wird. In der Map-Phase wird nun für jeden Split der großen Tabelle L überprüft, welche Tupel im Join Attribut einen Wert enthalten, der in der Hash Tabelle aufgeführt ist. Generell besteht die Ausgabe dann für jeden Split aus einer Liste von Tupeln, die für den Join in Frage kommen. Der dritte Job führt nun einen Join zwischen dem kleineren Datensatz R und der Ausgabe des zweiten Jobs durch. Das Ergebnis sollte identisch mit dem Ergebnis eines Joins zwischen L und R sein. Durch dieses Verfahren erspart man sich das Versenden und Sortieren der nicht benötigten Tupel von L, allerdings zahlt sich das Ganze nur aus, wenn der Anteil der von R referenzierten Tupel in L sehr klein ist oder wenn man die beiden Jobs, die nur der Vorbereitung des eigentlichen Joins dienen, schon vor der Ausführung der Abfrage als Preprocessing durchführt.

Multiway Join

Möchte man mehr als zwei Tabellen mit einem Join verknüpfen, so wird dies standardmäßig durch eine Verkettung von binären Joins realisiert. Effizienter wäre es aber, alle Tabellen in einer einzigen Iteration des Joins, also in einem MapReduce Job, zu verarbeiten. Diese Strategie wird als Multiway Join bezeichnet, und auch hier kann man in bestimmten Situationen Performancegewinne erwarten [2]. Im Folgenden soll der Drei-Wege Join als Beispiel für die allgemeine Funktionsweise eines Multiway Joins herangezogen werden.

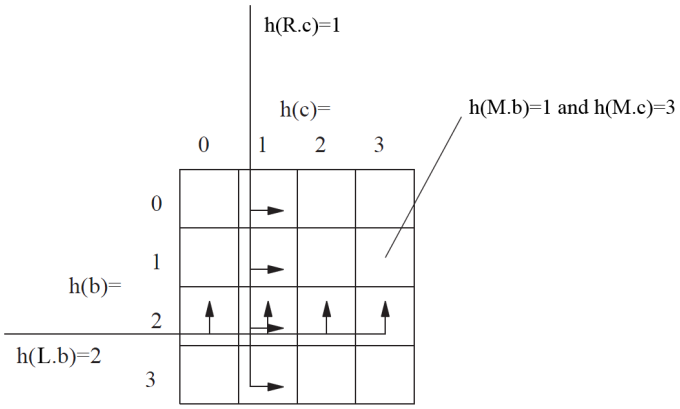


Abbildung 3.3: Verteilung der Tupel von L, M und R auf $k = m^2$ Reducer

Angenommen die drei Tabellen $L(A, B)$, $M(B, C)$ und $R(C, D)$ sollen miteinander verknüpft werden. Um dies in einem einzigen Job zu realisieren, wird so vorgegangen, dass ein Reduce-Side Join durchgeführt wird, bei dem die Tupel nach einem bestimmten Muster auf die Reducer verteilt werden, zu sehen in Abbildung 3.3 [2].

Der mittlere Datensatz M wird auf die gewohnte Art und Weise verteilt, nämlich so, dass jedes Tupel (b, c) einem einzigen Reducer zugewiesen wird, basierend auf $(h(b), h(c))$, also dem Tupel der Hash Werte der Join Attribute. Bei den Tabellen L und R ist allerdings nur jeweils ein Attribut für den Join von Interesse, deswegen werden alle Tupel (a, b) von L allen Reducern zugewiesen, die sich um die Hash Werte $(h(b), x)$ kümmern und alle Tupel (c, d) von R den Reducern zugewiesen, die sich um die Hash Werte $(y, h(c))$ kümmern (mit x,y beliebig). Es entsteht also die Situation, dass Tupel von L und R auf mehreren Reducern gleichzeitig vorhanden sind, dies ist aber notwendig um zu garantieren, dass der Join korrekt ausgeführt wird. Dadurch entstehen höhere Kommunikationskosten als bei einem binären Join, jedoch erspart man sich die Kommunikationskosten des kompletten zweiten Joins, den man ansonsten noch durchführen müsste. Diese Ersparnis kommt vor allem dann zum Tragen, wenn der Join Faktor der Tabellen hoch ist, also die Ausgabe des Joins groß ist.

Theta Join

Ein Theta Join ist ein Begriff für einen Join Algorithmus, der mit beliebigen Vergleichsoperatoren arbeiten kann. Während die meisten Join Algorithmen in MapReduce nur mit der Gleichheit effizient umgehen können, eröffnet der im Folgenden vorgestellte Algorithmus die Möglichkeit, beliebige Vergleichsoperatoren zu benutzen und trotzdem nur einen einzigen MapReduce Job dafür zu benötigen [10]. Diese Möglichkeit kommt mit einer Vorbedingung, nämlich dass die Kardinalität der beiden Eingabedatensätze bekannt ist.

Der Theta Join Algorithmus basiert auf einer speziellen Join Matrix. Die Zeilen der Matrix werden mit den Tupeln des einen Datensatzes und die Spalten mit den Tupeln des anderen Datensatzes gekennzeichnet. Wird die Join Bedingung durch zwei Tupel erfüllt, so wird die Zelle in der Matrix markiert, die der Schnittstelle zwischen Zeile und Spalte der beiden Tupel entspricht. Würde man ein Kreuzprodukt durchführen, dessen Ergebnis eine Obermenge aller möglichen Join Ergebnisse darstellt, wäre entsprechend die gesamte Matrix markiert. Das Ziel des Algorithmus ist es, die markierten Zellen so an die Reducer zu verteilen, dass jeder Reducer ungefähr gleich viele Eingabe- und Ausgabetupel erhält und jedes Ausgabetupel von exakt einem Reducer produziert wird, also den Join möglichst gleichmäßig und schnell ablaufen zu lassen. Dabei entstehen idealerweise zusammengesetzte Regionen innerhalb der Matrix, die einem Reducer zugeordnet sind.

In Abbildung 3.4 sieht man zum Beispiel eine, in drei gleich große Bereiche unterteilte, Join Matrix, wobei jeder zusammengesetzte Bereich einem von drei Reducern zugeteilt ist. Die Spalten sind mit Tupeln des Datensatzes T gekennzeichnet, während die Zeilen zu den Tupeln des Datensatzes S gehören.

Wird in der Map-Phase nun ein Tupel verarbeitet, so wird in der vorher erstellten Matrix überprüft, welche dieser Regionen das gerade zu verarbeitende Tupel mit seiner Spalte/Zeile (je nach Eingabedatensatz) schneidet. Diese Information wird zusätzlich zu dem bereits bekannten Tag, das die Herkunft des Tupels anzeigt, an den

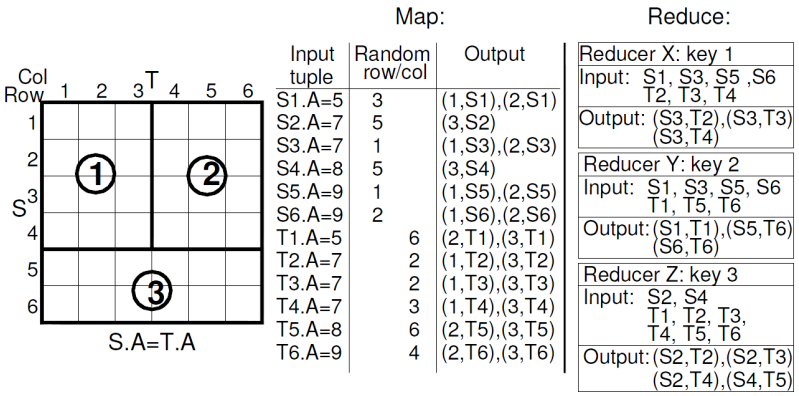


Abbildung 3.4: Matrix/Reducer Mapping für einen Equi-Join

Schlüssel des Tupels angehängt. Somit landet ein Tupel zur Weiterverarbeitung bei allen Reducern, deren Regionen es in der Join Matrix schneidet. Das einzige Problem hierbei ist, dass man nicht bestimmen kann, welche Zeile/Spalte das aktuelle Tupel identifiziert. Lediglich Informationen über die Herkunft des Tupels sind bekannt, also ob es in einer Zeile oder doch in einer Spalte zu finden ist. Um dieses Problem zu umgehen, wählt der Algorithmus zufällig aus der entsprechenden Menge aus, was bei großen Datenmengen zu einer Gleichverteilung der Tupel über die Reducer führt, solange alle Regionen in der Matrix etwa gleich groß sind.

Im mittleren Bereich der Abbildung 3.4 sieht man, wie *Map* den Tupeln des Datensatzes S zufällige Zeilen zuordnet. Daraufhin wird jeweils ein Ausgabetupel für jeden Reducer, dessen Bereich in dieser Zeile vorkommt, erzeugt und mit der Nummer des dazugehörigen Reducers versehen. In der gleichen Abbildung ist ersichtlich, dass jeder Reducer am Ende etwa gleich viele Tupel erhält und auch etwa gleich große Ausgaben produziert.

Insgesamt lässt sich also durch das Anfertigen dieser Join Matrix, die lediglich das Wissen über die Kardinalitäten der Eingabedatensätze voraussetzt, eine effektive Join-Strategie nutzen, die auch Daten, die sonst *Skew* verursachen würden (siehe Kapitel 4.3), gleichmäßig auf verschiedene Reducer verteilt und das Anwenden beliebiger Vergleichsoperatoren ermöglicht.

4 BrackitMR und Erweiterungen

BrackitMR ist eine Erweiterung von Brackit, die sich bei der Ausführung von XQuery Abfragen auf das MapReduce-Framework stützt [12]. In den ersten drei Kapiteln dieser Arbeit haben wir uns mit Brackit, MapReduce und Hadoop auseinandergesetzt, womit die nötigen Grundpfeiler für das Verständnis von BrackitMR gesetzt sind. Das Ziel von BrackitMR ist es, der Flexibilität von Brackit als Top-Layer Query Engine durch MapReduce ein mächtiges Framework zugrunde zu legen. Dabei sollen jedoch jegliche Berechnungen durch XQuery-Ausdrücke repräsentiert werden und das MapReduce Back-End komplett offen gelegt werden.

Während die erste Hälfte dieses Kapitels sich mit dem BrackitMR-Framework auseinandersetzt, beschäftigt sich die zweite Hälfte mit der Implementierung von speziellen Join-Verfahren und Erweiterungen, die in bestimmten Situationen Performancegewinne bedeuten.

4.1 Operatoren

Die XQuery Abfragen, die uns im Hinblick auf die Analyse von großen Datenmengen interessieren, bestehen im Kern aus FLWOR Ausdrücken, deren Pipelines zur Ausführung auf MapReduce auf entsprechende Jobs übertragen werden müssen.

Will man einen Abfrageplan in das generalisierte MapReduce Modell überführen, das BrackitMR zugrunde liegt, so muss man den Operatoren Task Funktionen zuweisen. Innerhalb eines Mappers wird nur auf einem Split gearbeitet, weswegen diesen Task Funktionen nur nicht-blockierende Operatoren zugewiesen werden können. Die einzige Möglichkeit einen blockierenden Operator darzustellen, besteht also darin, die Shuffle Funktion zu nutzen. Diese ist aber in unserem Modell unveränderbar. Basierend auf dem Ergebnis des Shuffles, müssen also die nachfolgenden Task Funktionen weitere Schritte durchführen, um den gewünschten Operator abzubilden. Da mehrere nicht-blockierende Operatoren hintereinander ausgeführt immer noch nicht-blockierend bleiben, kann man so viele von ihnen in eine Task Funktion packen, bis ein blockierender Operator die Kette bricht und somit ein Shuffle folgen muss. Abbildung 4.1 zeigt, wie eine Abfrage, die einen *Join* und ein *GroupBy* enthält, in Tasks aufgeteilt wird [14].

Nicht-blockierende Operatoren

Nicht-blockierende Operatoren verarbeiten einzelne Tupel nacheinander und können zusammen in einen Mapper oder Reducer gepackt werden. Sie bilden dann eine eigenständige Pipeline, die schon ein gewisses Maß an Parallelität bietet. Zu diesen Operatoren zählen in BrackitMR *ForBind*, *LetBind*, *End* und *Select*, die dem **for**, **let**, **return**

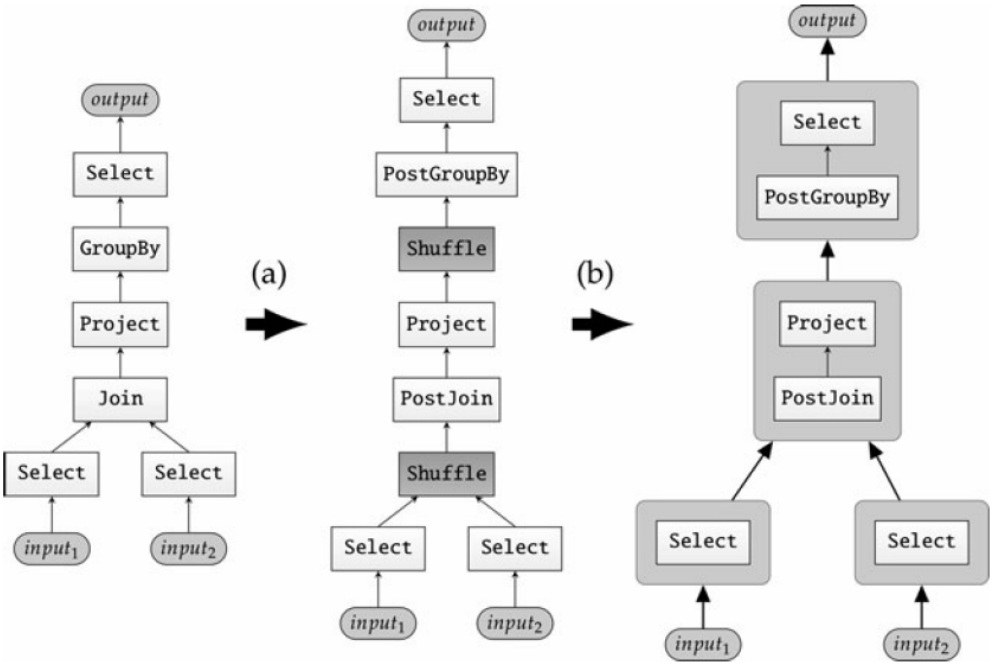


Abbildung 4.1: Umgeschriebener Query Plan (a) und Übersetzung in Task Functions (b)

und **where** in einem FLWOR Ausdruck entsprechen. Da der *End*-Operator seine Ausgabe in das Dateisystem schreibt, muss er theoretisch in einer *Reduce* Phase ausgeführt werden, was beim generalisierten Modell aber keine Rolle spielt. Die anderen oben genannten Operatoren weisen ebenfalls keine weiteren Einschränkungen auf.

Blockierende Operatoren

Zu den blockierenden Operatoren zählen *Group*, *Join* und *Sort*, wobei der Letztere dem **order by** eines FLWOR Ausdrucks entspricht. Mit zusätzlicher Nachbereitung können diese drei Operatoren mithilfe eines Shuffles simuliert werden, also sind sie geeignet, um als MapReduce Job ausgeführt zu werden. Die Nachbereitung geschieht durch sogenannte Post-Operatoren, wie zum Beispiel dem *PostJoin*. Außerdem benötigt man zur Duchführung eines Shuffles in BrackitMR auch noch die beiden Hilfsoperatoren *PhaseOut* und *PhaseIn*. Diese sind unter anderem notwendig, da die Eingabe des Shuffles in Form von MapReduce-typischen Key/Value-Paaren vorliegen muss, während Brackit Pipelines nur mit Tupeln arbeiten. Innerhalb eines MapReduce Tasks spielt das keine Rolle, da das gesamte Tupel als $Value$ deklariert werden kann, zur Durchführung eines Shuffles müssen aber Attribute als Key spezifiert werden, um die Sortierung entsprechend durchzuführen.

PhaseOut muss also die gegebenen Tupel so verändern, dass ein Shuffle die gewünschte Partitionierung vornimmt. Dies geschieht dadurch, dass der *PhaseOut*-Operator die Variablen extrahiert, die als Schlüssel für das Shuffle benutzt werden sollen. Er packt alle dafür vorgesehenen Variablen in einen zusammengesetzten Schlüssel und schreibt die restlichen Variablen in den *Value* Teil des entsprechenden Key/Value-Paars. Shuffle sortiert und gruppiert dann alle Tupel nach dem Wert des Schlüssels, bevor der *PhaseIn*-Operator die Variablen wieder in die ursprüngliche Position innerhalb des Tupels bringt, damit nachfolgende Operatoren diese wieder ordnungsgemäß lesen können.

Während Shuffle die *Values* aller Tupel mit dem gleichen Schlüssel in einer Liste aneinanderhängt, ist bei einem *Group* gemäß dem **group by** gewünscht, dass diese Liste in einer einzelnen Sequenz zusammengefasst wird. Auch hier wird also ein Post-Operator benötigt. Die Semantik eines *Sort* wird allerdings schon durch Shuffle und das nachfolgende *Reduce* garantiert.

Von besonderem Interesse für diese Arbeit ist die Ausführung von Joins. Diese haben immer mindestens zwei verschiedene Eingabe-Datensätze, die auf einem oder mehreren Attributen miteinander verglichen und dann zusammengefügt werden. In BrackitMR sind momentan nur binäre Joins erlaubt. Das generalisierte MapReduce Modell unterstützt hierfür verschiedene Mapper, die auf verschiedenen Datensätzen arbeiten. Diese verarbeiten beide Datensätze, die später Eingaben des Joins sind, gleichzeitig und liefern ihre Ergebnisse dann zusammen an den gleichen Shuffle. Zusätzlich zu *PhaseOut*, Shuffle und *PhaseIn* muss der oben genannte *PostJoin* durchgeführt werden, der standardmäßig einen Hash-Join implementiert.

4.2 Collections

Collections sind ein Konstrukt in Brackit und BrackitMR mithilfe dessen auf externe Daten zugegriffen werden kann. In BrackitMR wird grundsätzlich mit CSV Daten gearbeitet, die ähnlich einer Tabelle zu interpretieren sind. Dabei steht jede, durch einen Zeilenumbruch abgegrenzte Zeile des CSV-Dokuments für eine Zeile einer Tabelle, wobei innerhalb dieser Zeile die einzelnen Tabellenspalten durch ein spezielles Zeichen abgegrenzt werden. Diese CSV-Dateien können im Prolog einer BrackitMR Abfrage, also vor dem Code der eigentlichen XQuery Abfrage, als Collection deklariert werden. In diesem Fall als spezielle `CSVCollection`, einer Implementierung des Collection Interfaces, die den Itemtyp `Record` bereitstellt, wenn über die Daten iteriert wird.

Eine weitere Besonderheit der `CSVCollection` ist, dass das Zeichen, welches die Spalten der CSV-Datei trennt, spezifiziert werden kann. Um diese Collections in Hadoop zu nutzen, erweitert die Klasse `HadoopCSVCollection` die Klasse `CSVCollection` und stellt unter anderem eine Methode bereit, um die Collection als Eingabe des Hadoop Jobs einzutragen.

Generell funktionieren Collections also so, dass man ihrem Konstruktor einen Namen, Pfad, Itemtyp und eventuell zusätzliche Optionen übergibt. Daraus entsteht dann ein Objekt, das über die Daten im spezifizierten Pfad iterieren kann und die jeweiligen

Resultate als Items mit dem entsprechend definierten Typ zurückliefert.

4.3 Skew Reduzierung

Ein häufiges Problem, das im Zusammenhang mit verteilten Joins auftritt, ist der sogenannte *Skew* [20]. Dieser äußert sich in größeren Differenzen der Zeit, welche die verschiedenen Mapper und Reducer zur Berechnung ihrer Resultate benötigen. Da der Abschluss einer Map/Reduce-Phase erst dann erfolgt, wenn der letzte Rechner seine Arbeit abgeschlossen hat, resultiert Skew in einer Hinauszögerung des Gesamtergebnisses. Ist eine Berechnung frei von Skew, so benötigen also alle Mapper und alle Reducer untereinander gleich lang zur Berechnung ihres jeweiligen Ergebnisses. Skew hat verschiedene Ursachen, die im Folgenden kurz erläutert werden.

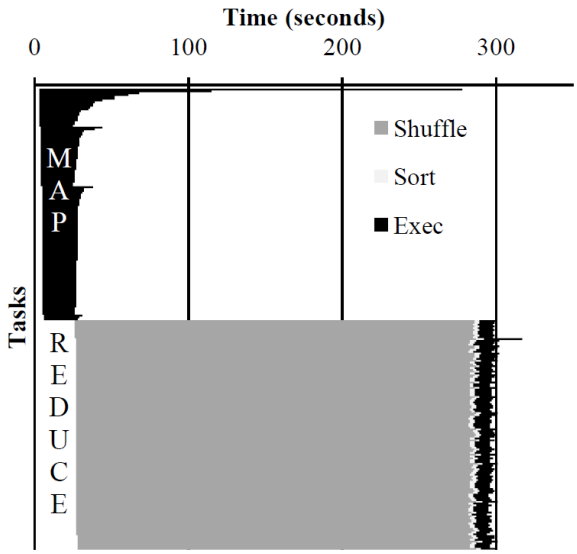


Abbildung 4.2: Ein Ablaufdiagramm eines MapReduce Jobs. 'Exec' zeigt die eigentliche Rechendauer der Map und Reduce Tasks.

Skew kann dadurch entstehen, das einzelne Eingaben einen größeren Rechenaufwand verursachen als gewöhnlich. Dies wiederum kann entweder in deren Größe oder in der Map/Reduce Funktion selbst begründet sein. Wird einem Mapper oder einem Reducer ein besonders rechenaufwändiger Datensatz zugeteilt, benötigt er länger als durchschnittlich, um seine Berechnung abzuschließen.

In der Reduce-Phase kann Skew außerdem durch schlechte Partitionierung der Daten verursacht werden. Wird die Map Funktion genutzt, um Aufgaben auszuführen, die normalerweise von Reducern erledigt werden, wie zum Beispiel bei einem Map-Side Join, so kann diese Art von Skew auch dort entstehen. Dieser Skew lässt sich relativ einfach beseitigen, indem bessere Algorithmen zur Partitionierung der Daten verwendet werden.

In MapReduce wird eine Hashfunktion genutzt, um die Daten auf die einzelnen Reducer zu verteilen. Wenn diese Hashfunktion die Daten schlecht verteilt, oder einfach zu viele gleichartige Datensätze vorkommen, dann passiert es, dass einigen Reducern viel mehr Daten zur Verarbeitung zugeordnet werden, als den anderen Reducern. Da mehr Daten meistens auch mehr Rechenaufwand bedeuten, resultiert dieser Umstand in einer größeren Gesamtrechendauer, also Skew. Abbildung 4.2 zeigt ein Ablaufdiagramm eines Jobs, der von Skew betroffen ist. Die oberen, langsamen *Map* Tasks verzögern die Ausführung des Sort und der *Reduce* Tasks, was zu einer insgesamt vielfach höheren Gesamtrechendauer führt [20].

Hash Codes in BrackitMR

In BrackitMR werden während der Durchführung eines Hash-Joins drei Hashfunktionen ausgeführt. Die Erste wird an der Stelle verwendet, wo die Ergebnisse der Map-Phase auf die Reducer verteilt werden. Diese Funktion wird auch sonst immer benutzt, wenn eine Reduce-Phase durchgeführt wird, im Gegensatz zu den nächsten beiden Hashfunktionen, die nur während eines Hash-Joins Anwendung finden.

Da BrackitMR zur Verarbeitung von Big Data entwickelt wird, gehen wir davon aus, dass jeder Reducer eine große Menge an Eingabetupeln bekommt, die sich im Join Attribut unterscheiden. Dies macht die zweite Partitionierung notwendig, sodass die folgende Konstruktion der Hash Tabelle zum Joinen der Datensätze die Grenzen des Hauptspeichers nicht übersteigt. Ein Reducer verarbeitet nun also mehrere interne Partitionen nacheinander, die bei großen Datenmengen aber immer noch eine Vielzahl an unterschiedlichen Join Attributen enthalten können.

Die dritte Hashfunktion wird benutzt, um die Hash Tabelle zu erzeugen, die für den eigentlichen Join der Datensätze verwendet wird. Die Tupel der kleineren Eingabe werden in Buckets innerhalb der Hash Tabelle gelegt und später mit Tupeln der größeren Eingabe verglichen und eventuell gejoint.

Damit es an diesen drei wichtigen Stellen zu keinem Skew aufgrund von schlechter Partitionierung kommt, müssen diese drei Hashfunktionen komplett unabhängig voneinander sein. Das ist vor allem wichtig bei großen Eingaben, wie sie bei der Analyse von Big Data vorkommen, da diese sonst den Hauptspeicher der Reducer zum Überlaufen bringen und den Job scheitern lassen.

Ein Problem hierbei ist, dass diese Hashfunktionen unterschiedliche Eingabedatentypen geliefert bekommen, nämlich `Byte Array` und `Atomic`. Die Eingabedatentypen lassen sich jedoch nicht verändern, da sie vom BrackitMR-Framework diktiert werden, weswegen ein Weg gefunden werden muss, die Eingaben zentral und vergleichbar zu behandeln, ohne die Effizienz durch kostspielige Berechnungen zu beeinträchtigen.

Die Klasse HashCodeGenerator

Die erste Erweiterung stellt die Implementierung der `HashCodeGenerator` Klasse dar, die sich um die Ausführung der Hashfunktionen kümmert. In ihr enthalten ist die polymorphe Funktion `hashCode()`, die je nach gegebenen Parametern unterschiedliche

Hash Werte berechnet, also die erste, zweite oder dritte der oben angesprochenen Hashfunktionen repräsentiert.

Wenn ein einzelner Wert vom Typ `Atomic` gehasht werden soll, so wird die interne Funktion von Java benutzt, die jedes Objekt standardmäßig bereitstellt. Wird jedoch ein `Atomic[]`, also ein Feld von Schlüsseln, oder ein `byte[]`, wie es bei der internen Partitionierung eines Reducers der Fall ist, als Parameter übergeben, dann sorgt eine Version des FNV-Hashs für eine effiziente Berechnung der Hash Werte. Der FNV-Hash ist ein bewährter Algorithmus, der auf dem Einsatz von Primzahlen beruht und sich durch Geschwindigkeit und Zuverlässigkeit auszeichnet [9].

Um letztendlich zu garantieren, dass wir unabhängige Ergebnisse erhalten, obwohl an drei Stellen die gleiche Funktion `hashCode()` aufgerufen wird, gibt es noch einen wichtigen Parameter, der übergeben werden kann. Dieser Parameter ist ein `Integer` Wert, der eine Maske identifiziert. Von diesen Masken, die aus einem `32bit Integer` bestehen, lassen sich beliebig viele über die Konfiguration des Jobs definieren, standardmäßig werden jedoch drei bereitgestellt, die den `32bit` Hashcode in drei Bereiche unterteilen.

Eine gute Hashfunktion, wie auch der FNV-Hash, garantiert, dass die einzelnen Bits des generierten Codes weitestgehend unabhängig voneinander sind. Um also einen Hash Code in drei unabhängige, getrennte Codes zu teilen, genügt es, den Hash mit den drei Masken zu multiplizieren und somit die gewünschten Bereiche des ursprünglichen Codes zu extrahieren.

4.4 Multi-Key Join

In BrackitMR sprachen wir bisher nur von Joins, die zwei Datensätze auf Basis eines einzelnen gemeinsamen Attributs verbinden, also Joins mit einem einzelnen Join Key. Um sogenannte Multi-Key Joins zu unterstützen, muss das Framework minimal angepasst werden, was vor allem den `HashPostJoin`-Operator betrifft. Die entsprechenden Hashfunktionen arbeiten, wie oben beschrieben, bereits mit Eingabefeldern, was keiner weiteren Umgestaltung bedarf. Im Operator müssen die bereits vorhandenen Funktionen so angepasst werden, dass auch Felder aus mehreren `Atomics` akzeptiert werden.

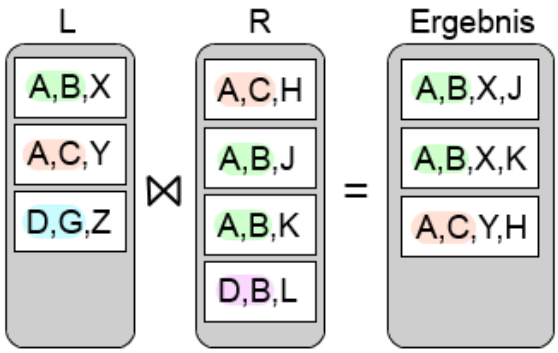


Abbildung 4.3: Multi-Key Equi-Join zwischen zwei Datensätzen L und R

In der Abbildung 4.3 wird die Durchführung eines Multi-Key Equi-Joins anhand eines Beispiels verdeutlicht. Die ersten beiden Attribute der Tupel von L und R werden als Schlüssel für den Join benutzt. Nur wenn die Kombination dieser beiden Attribute in einem Tupel der anderen Eingabe vorhanden ist, wird ein Join zwischen diesen beiden Tupeln durchgeführt.

4.5 Fragment Replicate Join

Der Fragment Replicate Join wurde bereits oben vorgestellt und gehört zur Klasse der Map-Side Joins. Das bedeutet, er ermöglicht den Verbund zweier Datensätze innerhalb einer einzelnen Map-Phase ohne den Ballast, den ein darauf folgendes Shuffle und *Reduce* mit sich bringen würde. Es werden also hauptsächlich Netzwerkkosten gespart. Damit ein FRJoin durchgeführt werden kann, gibt es allerdings die Bedingung, dass mindestens einer der beiden Datensätze, die die Eingabe des Joins bilden, klein genug ist, um in den lokalen Speicher der Rechner im Cluster zu passen. Der FRJoin ist bereits in Plattformen wie Pig und HIVE implementiert und bietet dort große Performancegewinne [1], weswegen eine Implementierung in BrackitMR eine logische Konsequenz darstellt, um BrackitMR als effiziente Alternative zu diesen Systemen aktuell zu halten.

DistrCacheCollection

Um den FRJoin zu implementieren, ist es notwendig auf eine Collection zuzugreifen, die zwar zunächst wie eine gewöhnliche `CSVCollection` konstruiert wird, ihre Daten aber in den Distributed Cache des Jobs überträgt. Dadurch bekommt jeder Rechner die für den Join erforderliche Daten in den lokalen Speicher übertragen. Wenn nun während des Jobs ein Mapper die Daten von dieser, im Folgenden `DistrCacheCollection` genannten, Collection lesen möchte, so sollte er die Daten aus dem Distributed Cache erhalten. Das spart Netzwerkkosten und beschleunigt die Ausführung des Jobs.

Abfragen und Jobs

Da der Inhalt des Distributed Cache als Ganzes jedem Knoten während des Jobs zur Verfügung stehen soll, ist es wichtig, dass man die `DistrCacheCollection` nicht wie eine reguläre Eingabe des Jobs behandelt. Ansonsten würde sie in mehrere Splits unterteilt bei verschiedenen Mappern landen, was nicht der Semantik des Distributed Cache entspricht. Der Datensatz, den wir aus dem Distributed Cache erhalten, stellt also lediglich eine Eingabe der Abfrage und nicht des Jobs dar.

Wenn man sich den rechten Teil der Abbildung 4.4 ansieht, so erkennt man an oberster Stelle den FRJoin-Operator. Dieser hat zwei Kindknoten, SubTask 1 und SubTask 2, welche die beiden Eingaben des Joins aus Sicht der Abfrage symbolisieren. Wenn der linke Kindknoten seine Daten aus einer `DistrCacheCollection` bezieht und der Rechte aus einer gewöhnlichen `HadoopCSVCollection`, dann wird davon nur diese `HadoopCSVCollection` als Eingabe des Jobs markiert. Das bedeutet

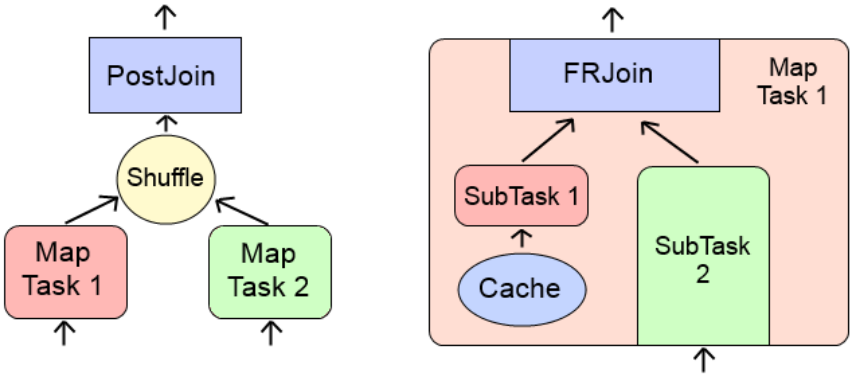


Abbildung 4.4: PostJoin und FRJoin

mehrere Mapper verarbeiten parallel diesen rechten Teilbaum, jeder von diesen Mappern muss aber auch eigenständig den kompletten linken Teilbaum für jedes Tupel der `DistrCacheCollection` durchlaufen, da er Teil des gleichen Tasks ist.

Hier wird deutlich, dass die Performance des FRJoins allgemein stark von der Komplexität dieses linken Teilbaums, sowie von der Größe der `DistrCacheCollection` abhängt.

Implementierung des FRJoin

Hat man die `DistrCacheCollection` und die damit einhergehende Unterscheidung der Eingaben implementiert, dann ist nur noch ein passender *Join*-Operator zu finden. Der Optimizer muss daraufhin so verändert werden, dass er FRJoin Situationen erkennt und den AST der Abfrage entsprechend modifiziert. Der FRJoin-Operator soll wie ein gewöhnlicher Hash-Join arbeiten und aus einer der beiden Eingaben, nämlich der, die aus dem Distributed Cache kommt, eine Hash Tabelle aufbauen. Anschließend soll für jedes Tupel der anderen Eingabe die Hash Tabelle auf Übereinstimmungen geprüft und ein Join der Tupel durchgeführt werden. Es passiert also semantisch genau das Gleiche, wie bei einem gewöhnlichen *Join*-Operator in Brackit, weswegen auch dieser Operator verwendet werden kann.

Erkennt der BrackitMR Optimizer, dass mindestens eine Eingabe eines Joins aus einer `DistrCacheCollection` stammt, dann wird der AST der Abfrage automatisch so umgeformt, dass ein FRJoin durchgeführt werden kann. Konkret bedeutet das, dass *PhaseOut*, Shuffle, *PhaseIn* und *PostJoin*-Operator entfallen und stattdessen ein einfacher *Join*-Operator eingesetzt wird. Da dieser *Join*-Operator vollständig innerhalb des *Map* Tasks ausgeführt wird, entfällt so häufig ein kompletter *Reduce* Task des Jobs.

In der Konfigurationsdatei des Jobs lässt sich definieren, ob Eingaben in den Distributed Cache geladen werden und welche Größe sie dafür maximal haben dürfen.

4.6 Sort-Merge Join

Da der Fragment Replicate Join nur ausgenutzt werden kann, wenn mindestens ein Datensatz entsprechend klein ist, ist es wichtig, dass man auch im Fall zweier großer Datensätze eine effiziente Join-Strategie zur Hand hat. Um eine Alternative zum bereits implementierten Hash-Join zu bieten, wurde eine angepasste Version des Sort-Merge Joins implementiert, der allerdings eine spezielle Sortierung der Tupel voraussetzt.

Funktionsweise

Ein gewöhnlicher Sort-Merge Join funktioniert so, dass die Tupel in der Shuffle Phase nach Join Attribut sortiert und *gruppiert* werden, sodass jeder Aufruf von *Reduce* als Eingabe nur Tupel mit einem bestimmten Join Schlüssel erhält. Dann kann der Reducer die Herkunft der Tupel anhand eines *Tags* unterscheiden und zwei Tabellen daraus aufbauen. Zwischen der Tabelle, die alle Tupel der linken Eingabe enthält und der Tabelle, die alle Tupel der rechten Eingabe enthält, wird dann ein kartesisches Produkt durchgeführt, um den Join abzuschließen.

In BrackitMR lässt sich der Sort-Merge Join allerdings nicht direkt auf diese Weise durchführen, da meistens eine andere Ausgangssituation vorliegt. Reduce Tasks in BrackitMR verarbeiten im Allgemeinen keine gruppierten Eingaben und die Zahl der voneinander verschiedenen Werte des Join Schlüsselattributs übersteigt meistens auch die Anzahl der bereitgestellten Reducer um ein Vielfaches, sodass es trotz doppelter Partitionierung dazu kommt, dass in der Eingabe eines *Reduce* Tasks Tupel mit verschiedenen Werten des Join Schlüssels vorhanden sind. Ein kartesisches Produkt direkt zwischen der linken und rechten Eingabe eines solchen Reducers würde also zu einem falschen Ergebnis führen.

Die SMJoin Tabelle

Damit zwischen den unterschiedlichen Werten des Join Attributs differenziert werden kann, ohne eine Gruppierung der Tupel zu benötigen, müssen zwei Tabellen angelegt werden.

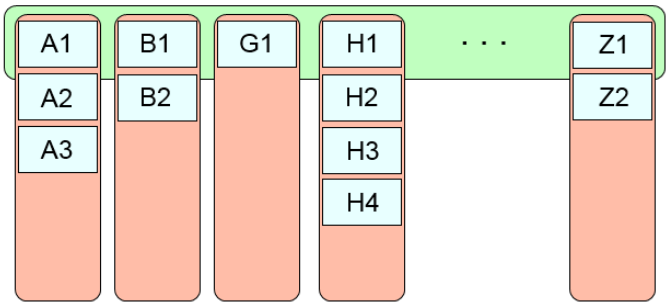


Abbildung 4.5: Beispiel einer SMJoin Tabelle

Abbildung 4.5 zeigt die Ausprägung einer solchen Tabelle. Alle Tupel, die im Join Attribut den Wert A haben, befinden sich in der ersten Spalte. In der Darstellung sind die Tupel nummeriert, um sie später besser unterscheiden zu können, diese Nummerierung hat jedoch nichts mit der Herkunft der Tupel, also dem Join-*Tag*, gemeinsam. Folglich gibt es drei verschiedene Tupel, deren Join-Schlüssel den Wert A besitzt. Das besondere an dieser Tabelle ist, dass die Spalten die ursprüngliche Sortierung der Tupel beibehalten, im Gegensatz zu der Tabelle, die während eines Hash-Joins erstellt wird. Nachdem jeweils eine Tabelle für die linke und rechte Eingabe erstellt worden ist, gilt es zwischen dem Inhalt der Spalten, die im Wert des Join Attributes übereinstimmen, ein kartesisches Produkt zu bilden.

Allerdings gibt es noch eine Verbesserung, die das Anlegen der zweiten Tabelle überflüssig macht und die erwähnt werden sollte, bevor der weitere Ablauf des Joins geschildert wird. Sortieren wir die Eingabe des Sort-Merge Join-Operators während des Shuffles so, dass sie nicht nur nach Join Attribut sondern primär auch nach Herkunft, also *Tag*, geordnet ist, dann befinden sich alle Tupel der rechten Join-Eingabe vor denen der linken Eingabe. Das ermöglicht es uns, die Tabelle für die rechte Eingabe an einem Stück aufzubauen, ohne die Tupel der linken Eingabe zwischenspeichern zu müssen (siehe Algorithmus 1). Ist die Tabelle aus der rechten Eingabe aufgebaut, können wir für jedes darauffolgende linke Tupel den Join 'on-the-fly' durchführen. Hierbei kommt uns die Sortierung der Tabellenspalten zugute.

```
schlüsselIndex = 0;
tupelIndex = 0;
aktuellesTupel = erstesTupel;
while aktuellesTupel gehört zu rechter Eingabe do
    schlüsselWert = schlüsselWert(aktuellesTupel);
    if schlüsselWert identisch mit dem des vorherigen Tupels then
        Tabelle[schlüsselIndex][tupelIndex] = aktuellesTupel;
        tupelIndex = tupelIndex + 1;
    else
        schlüsselIndex = schlüsselIndex + 1;
        tupelIndex = 0;
        Tabelle[schlüsselIndex][tupelIndex] = aktuellesTupel;
    end
    aktuellesTupel = nächstesTupel();
end
```

Algorithm 1: Aufbau der SMJoin Tabelle

Abbildung 4.6 zeigt, wie die Sortierung der Tabellenspalten ausgenutzt werden kann, um Markierungen zu setzen, die die Suche in der Tabelle beschleunigen. Das linke, blau hinterlegte Rechteck stellt die linke Eingabe des Joins dar. Sie wird von oben nach unten durchlaufen, das erste Tupel ist also $A4$. Für jedes Tupel der linken Eingabe durchlaufen wir die Join Tabelle und suchen nach Spalten, die Tupel enthalten, die im Wert des

Join Attributs mit dem aktuellen Tupel der linken Eingabe übereinstimmen. Wurde eine solche Spalte gefunden, markieren wir sie und joinen das Tupel der linken Eingabe mit allen Tupeln, die sich in dieser Spalte befinden. Die Markierung sagt uns, dass wir in Zukunft erst ab dieser Stelle der Tabelle anfangen müssen zu suchen. In der Abbildung weisen die beschrifteten Pfeile darauf hin, wo die genannten Tupel mit der Suche in der Tabelle anfangen müssen. Die Markierungen basieren auf der Annahme, dass die Eingaben sortiert sind und ersparen es uns, jedesmal die komplette Tabelle durchsuchen zu müssen.

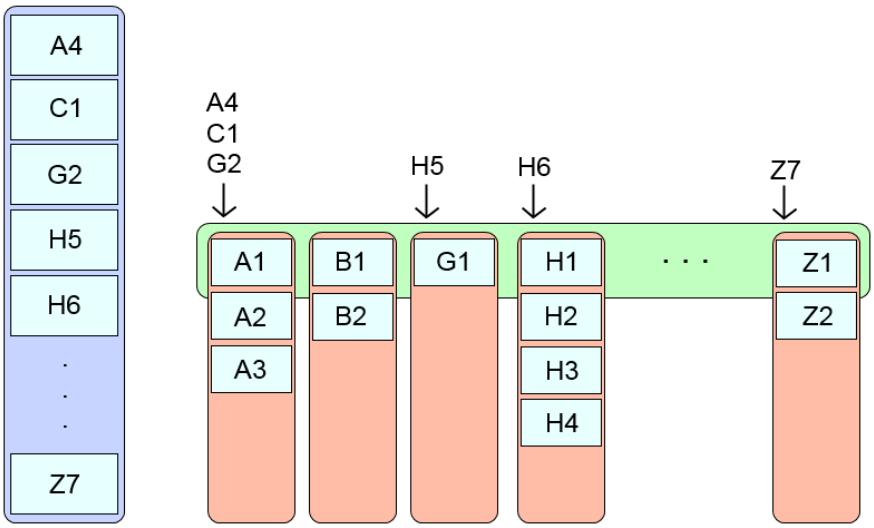


Abbildung 4.6: Markierungen beschleunigen die Suche in der Tabelle

Implementierung des SMJoins

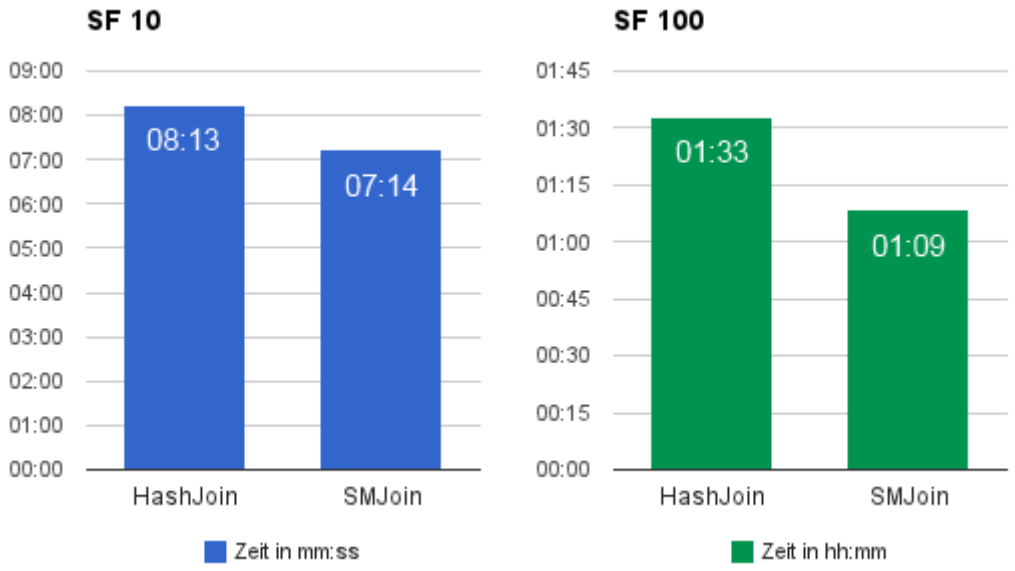
Die oben genannte Funktionalität wird im `SortMergePostJoin`-Operator bereitgestellt. Er verfügt über die gleichen Schnittstellen wie der `ArrayHashPostJoin`-Operator, der einen Hash-Join implementiert, und kann deswegen, bis auf eine kleine Einschränkung, genauso eingesetzt werden. Diese Einschränkung besteht in der Sortierung der Tupel. Der `ArrayHashPostJoin` setzt lediglich eine Sortierung nach *Tag* voraus, während der `SortMergePostJoin` eine Sortierung nach *Tag* und Join-Schlüssel voraussetzt. In der Konfigurationsdatei des Jobs lässt sich festlegen, welcher Join Typ verwendet werden soll, wodurch auch die Sortierung der Tupel entsprechend angepasst wird.

5 Leistungsanalyse

In diesem Abschnitt werden die Ergebnisse praktischer Testläufe vorgestellt. Es werden verschiedene Abfragen auf unterschiedlich großen Datensätzen präsentiert, wobei ein Vergleich zwischen Hash-Join, Fragment Replicate Join und Sort-Merge Join erfolgt. Die entsprechenden Datensätze wurden mittels des TPC-H Benchmarks erstellt [4], der sich als Standard im Bereich der relationalen Datenanalyse etabliert hat. Die Größe der Datensätze wird durch den sogenannten *Scale Factor SF* bestimmt, ein *SF* von 1 orientert sich dabei an einer Gesamtgröße von 1GB. Zur Durchführung der Experimente wurde ein Cluster mit fünf Rechenknoten und Hadoop 1.0.4 verwendet.

Standard Join

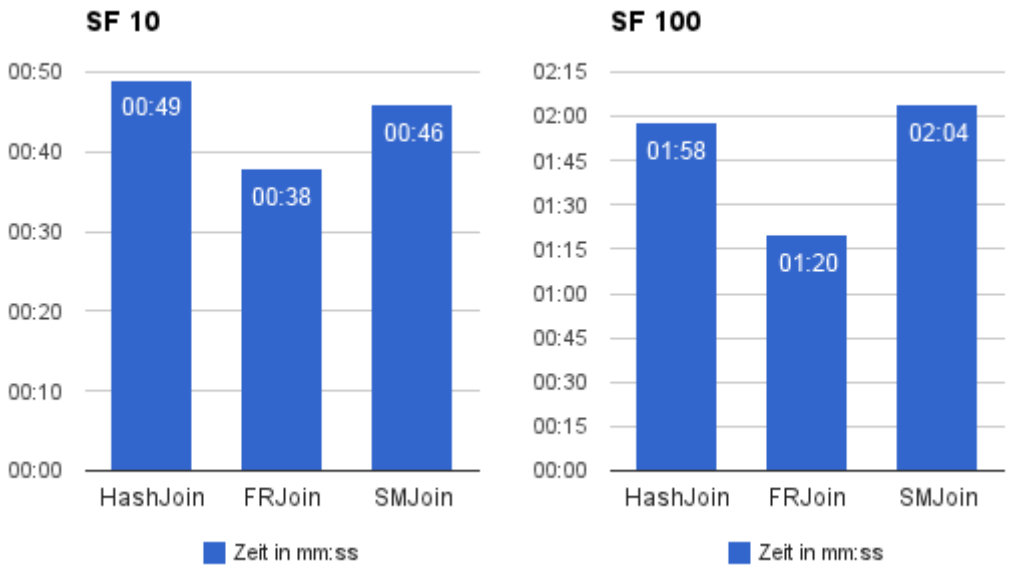
Die erste Abfrage ist ein Join zwischen den beiden großen Tabellen *lineitem* und *orders*. Der Vergleich erfolgt deswegen nur zwischen Hash-Join und Sort-Merge Join.




```
for $l in collection('lineitem')
for $o in collection('orders')
where $l=>orderkey eq $o=>orderkey
return { order: $o=>orderkey, lineitem: $l=>linenumber }
```

Join mit kleiner Tabelle

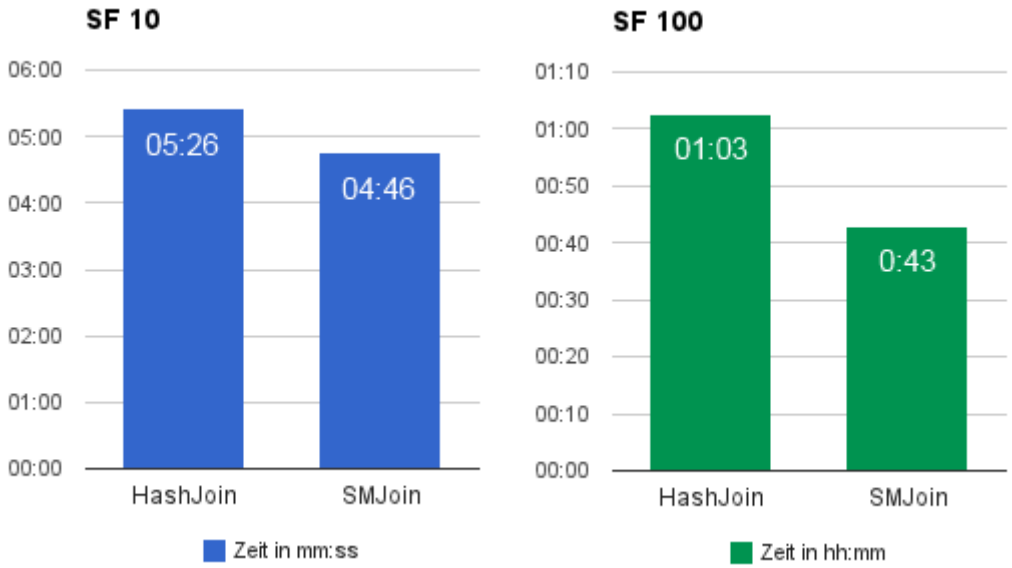
Diese Abfrage führt einen Join zwischen der relativ großen *customer*-Tabelle und der kleinen *nation*-Tabelle durch, die, unabhängig vom *Scale Factor*, nur 25 verschiedene Tupel beinhaltet. Dadurch bietet sich ein FRJoin an, der in diesem Szenario einen deutlichen Leistungsvorsprung erzielt.




```
for $c in collection('customer')
for $n in collection('nation')
where $c=>nationkey eq $n=>nationkey
return { customer: $c=>custkey, nation: $n=>nationkey }
```

Join zwischen gleich großen Tabellen

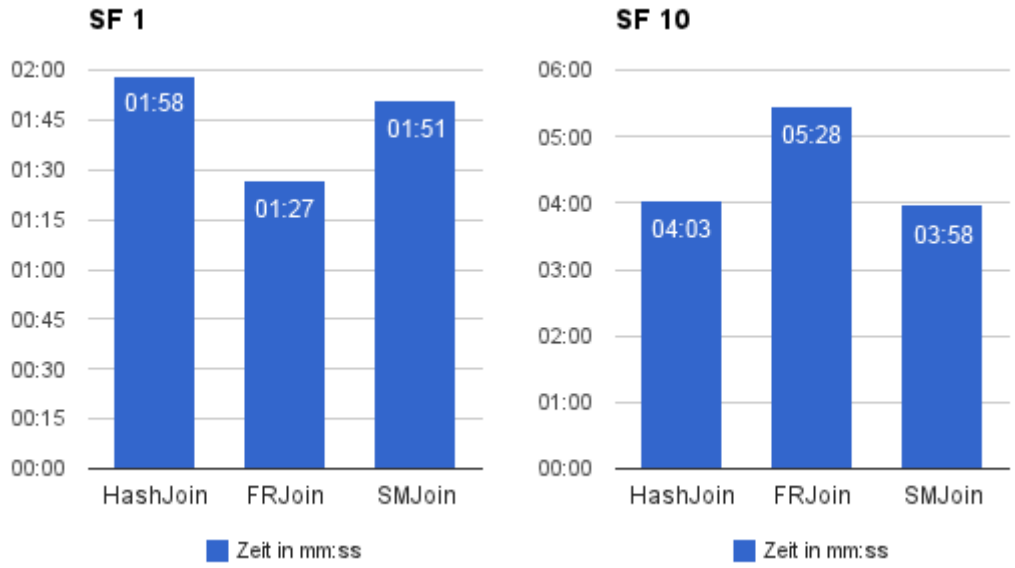
Die zweite Abfrage filtert zuerst alle Tupel der *lineitem*-Tabelle heraus, deren Spalte *linenumber* ungleich 1 ist. Das Ergebnis dieser Selektion ist eine Tabelle, die genauso groß ist, wie die *orders*-Tabelle, mit der daraufhin ein Join durchgeführt wird.




```
for $l in collection('lineitem')
for $o in collection('orders')
where $l=>linenumber eq 1
   and $l=>orderkey eq $o=>orderkey
return { order: $o=>orderkey, lineitem: $l=>linenumber }
```

Komplexe Abfrage

Die letzte Abfrage beinhaltet mehrere blockierende Operatoren und wird deswegen auch auf mehrere Hadoop Jobs verteilt. Das Laden der kleineren *customer*-Tabelle in den Distributed Cache zahlt sich diesmal nur für den kleineren Datensatz mit $SF = 1$ aus. Beim Verarbeiten des größeren Datensatzes stößt der FRJoin an seine Grenzen, wie man an der deutlich gestiegenen Ausführungszeit erkennen kann.




```
for $o in collection('orders')
for $c in collection('customer')
where $o=>custkey eq $c=>custkey
let $custkey := $c=>custkey
group by $custkey
let $avg := avg($o=>totalprice)
order by $avg
return { customer: $custkey, avg_price: $avg }
```

6 Fazit

6.1 Rekapitulation

Im Laufe dieser Arbeit wurden verschiedene Systeme, Konzepte und Strategien vorgestellt, um letztendlich an die in Kapitel 4 vorgestellten Erweiterungen des BrackitMR-Frameworks heranzuführen.

Begonnen haben wir in Kapitel 2 mit der Einführung der Brackit XQuery-Engine. Im gleichen Kapitel wurde auch XQuery und die dazugehörige Auszeichnungssprache XML präsentiert, sodass der Aufbau von Brackit in seinen Grundzügen nachvollzogen werden kann. Dadurch wurde auch bereits die erste Grundlage für BrackitMR gesetzt.

In Kapitel 3 wurde das von Google entwickelte MapReduce Programmiermodell vorgestellt. Zentral für dieses Modell sind die beiden namensgebenden Funktionen *Map* und *Reduce*, weswegen auch diese, zusammen mit dem allgemeinen Ablauf einer MapReduce-Berechnung, erläutert wurden.

Die frei zugängliche Implementierung des MapReduce Programmiermodells Hadoop, sowie das generalisierte MapReduce Modell, waren in diesem Kapitel ebenfalls von großer Bedeutung, da sie uns ermöglichen, Abfragepläne auf MapReduce Berechnungen abzubilden. Diese Übertragung der seriellen Ausführung einer XQuery-Abfrage auf eine effiziente Parallelverarbeitung in einem Rechencluster bildet die zweite wichtige Grundlage für BrackitMR.

Zusätzlich dazu, wurden in diesem Kapitel verteilte Join-Strategien vorgestellt, die speziell auf das MapReduce Programmiermodell abgestimmt sind und von denen insbesondere der Fragment Replicate Join und der Sort-Merge Join die Basis für die später vorgestellten Erweiterungen des BrackitMR-Frameworks bilden.

Im zentralen vierten Kapitel der Arbeit haben wir uns mit BrackitMR befasst und spezieller damit, wie die Übertragung von Abfrageplänen auf MapReduce Berechnung im Detail funktioniert. Daraufhin wurden die im Rahmen dieser Arbeit implementierten Erweiterungen des BrackitMR-Frameworks vorgestellt. Dazu gehören verbesserte Hashfunktionen und die damit verbundene Reduzierung von Skew aufgrund schlechter Partitionierung, die Erweiterung des Hash-Joins, damit Joins mit mehreren Join-Schlüsseln durchgeführt werden können, die Implementierung eines Fragment Replicate Joins, der es uns unter bestimmten Umständen ermöglicht, auf eine Reduce-Phase zu verzichten, sowie die Implementierung eines Sort-Merge Join-Operators als Alternative zum bereits vorhandenen Hash-Join Operator.

Schließlich wurden in Kapitel 5 die Ergebnisse einiger Experimente präsentiert, die deutlich zeigen, dass die vorher vorgestellten, spezialisierten Join-Strategien in einigen Situationen einen deutlichen Zeitgewinn bei der parallelen Auswertung von Abfragen bedeuten.

Obwohl die Durchführung von Joins als eine der Schwachstellen des MapReduce Konzeptes gilt, ist es möglich, durch angepasste und spezialisierte Join-Strategien diese Schwäche wieder auszugleichen, was letztendlich das primäre Ziel dieser Arbeit darstellte. Diese Anpassungsfähigkeit macht MapReduce und BrackitMR zu einer kostengünstigen und effizienten Alternative im Bereich der „Big Data Analysis“, deren volles Potential noch lange nicht ausgeschöpft ist. Ein weiteres Ziel der vorliegenden Arbeit war es, Einsteigern die nötigen Grundkenntnisse zu vermitteln, die für die nächsten Entwicklungen innerhalb des BrackitMR Projektes nötig sind. Abschließend wird vorgestellt, in welche Richtung diese zukünftige Arbeit gehen könnte.

6.2 Zukünftige Entwicklungen

Die oben vorgestellten Erweiterungen des BrackitMR-Frameworks unterliegen noch einigen Einschränkungen, die im Folgenden kurz aufgeführt werden sollen.

Die erste Einschränkung besteht in der Auswahl der passenden Join-Strategie. In Zukunft sollte der Compiler Join-Strategien gezielt auswählen können, ohne dass, wie es bisher der Fall ist, die Auswahl durch den Wert verschiedener Parameter in einer Konfigurationsdatei bestimmt wird. Dazu muss unter anderem die Größe von Zwischenergebnissen abgeschätzt werden, beispielsweise wenn sich eine Berechnung aus mehreren Jobs zusammensetzt. Um einen Fragment Replicate Join einsetzen zu können, muss die Größe der Eingabedatensätze bekannt sein, das beschränkt momentan die Nutzung des Fragment Replicate Joins auf den ersten MapReduce Job einer Berechnung. Auch für den Sort-Merge Join spielen die Größen der Eingabedatensätze eine wichtige Rolle, weswegen diese bei der Auswahl der Join-Strategie abgeschätzt werden sollten.

Zudem sollte der Sort-Merge Join in Zukunft mit mehreren Join-Schlüsseln umgehen können; eine Erweiterung, die weiter oben für den Hash-Join bereits vorgestellt wurde. Momentan beschränkt der Aufbau der Sort-Merge Join-Tabelle die Nutzung auf Joins mit nur einem einzigen Join-Attribut.

Zusätzlich zu der Verfeinerung der bereits implementierten Join-Verfahren, können auch die im Kapitel 3.7.3 vorgestellten komplexen Join-Verfahren noch in Betracht gezogen werden. Diese umfassen unter anderem den Multiway-Join, der bei Joins über mehrere Tabellen einen deutlichen Leistungsgewinn erwarten lässt, sowie den Theta-Join, der den Einsatz beliebiger Vergleichsoperatoren möglich macht. Der Theta-Join nutzt außerdem eine spezielle Matrix, um Daten, die sonst Skew verursachen würden, gleichmäßig auf die Reducer zu verteilen, was ebenfalls einen Leistungsgewinn mit sich bringen kann.

Auch abseits von Join-Verfahren lassen sich noch Leistungssteigerungen in BrackitMR erzielen, beispielsweise indem Hadoop modifiziert wird, sodass das generalisierte MapReduce Modell nativ unterstützt wird und nicht simuliert werden muss.

Wie bereits erwähnt, stellen die Anforderungen, die „Big Data“ an die heutigen Systeme stellt, in jeglicher Hinsicht eine große Herausforderung dar. Ob MapReduce und die darauf aufbauenden Systeme weiterhin so erfolgreich sind, oder ob andere, neuartige Konzepte die Zukunft für sich entscheiden werden, wird unter anderem von denen be-

stimmt, die das Potential in dem einen oder dem anderen Konzept sehen. Diese Arbeit hat letztendlich einen Teil des Potentials von BrackitMR offengelegt, was die Zukunft dieses Projektes umso spannender macht.

Literaturverzeichnis

[1] Pig FRJoin. http://wiki.apache.org/pig/PigFRJoin, Oktober 2013.

[2] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[4] Transaction Processing Council. The TPC-H Benchmark. http://www.tpc.org/tpch/, Oktober 2013.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[7] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[8] Ralf Lämmel. Google's MapReduce programming model - Revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[9] Landon Curt Noll. FNV Hash. http://www.isthe.com/chongo/tech/comp/fnv/, Oktober 2013.

[10] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.

[11] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[12] Caetano Sauer. XQuery Processing in the MapReduce Framework. Master's thesis, University of Kaiserslautern, Germany, 2012.

[13] Caetano Sauer, Sebastian Bächle, and Theo Härder. Versatile XQuery Processing in MapReduce. In Barbara Catania, Giovanna Guerrini, and Jaroslav Pokorný, editors, *Advances in Databases and Information Systems*, volume 8133 of *Lecture Notes in Computer Science*, pages 204–217. Springer Berlin Heidelberg, 2013.

[14] Caetano Sauer and Theo Härder. Compilation of Query Languages into MapReduce. *Datenbank-Spektrum*, 13(1):5–15, 2013.

[15] W3C. XQuery 3.0: An XML Query Language. http://www.w3.org/TR/xquery-30/, 2013.

[16] W3C. XQuery and XPath Data Model 3.0. http://www.w3.org/TR/xpath-datamodel-30/, 2013.

[17] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[18] Internet World. Internet-Datenvolumen 2012 - 2,8 Zettabyte in nur einem Jahr. http://www.internetworld.de/Nachrichten/Technik/Zahlen-Studien/Internet-Datenvolumen-2012-2-8-Zettabyte-in-nur-einem-Jahr-72246.html, Oktober 2013.

[19] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[20] Bill Howe YongChul Kwon, Magdalena Balazinska. A Study of Skew in MapReduce Applications. Technical report, University of Washington, USA, 2011.