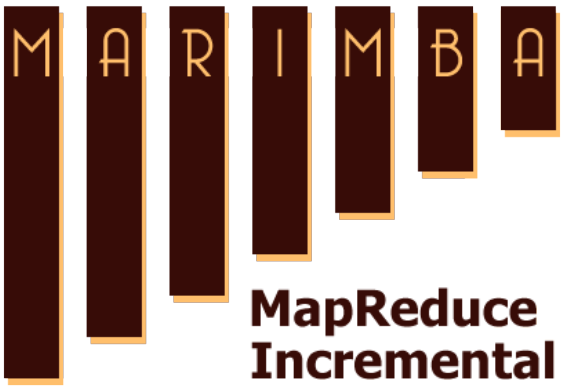
Fachbereich Informatik

Technische Universität Kaiserslautern

Masterarbeit

Ein MapReduce-basiertes Programmiermodell für selbstwartbare Aggregatsichten

Johannes Schildgen



Fachbereich Informatik

Technische Universität Kaiserslautern

Masterarbeit

Ein MapReduce-basiertes Programmiermodell für selbstwartbare Aggregatsichten

Autor:	Johannes Schildgen
Erstgutachter:	Prof. Dr.-Ing. Stefan Deßloch
Zweitgutachter:	Dr.-Ing. Thomas Jörg
Datum:	12. August 2012

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „Ein Map-Reduce-basiertes Programmiermodell für selbstwartbare Aggregatsichten“ 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 12. August 2012 Johannes Schildgen

Zusammenfassung

Hadoop ist ein beliebtes Framework für verteilte Berechnungen über große Datenmengen (Big Data) mittels MapReduce. Hadoop zu verwenden ist einfach: Der Entwickler definiert die Eingabedatenquelle und implementiert die beiden Methoden Map und Reduce. Über die verteilte Berechnung und Fehlerbehandlung muss er sich dabei keine Gedanken machen, das erledigt das Hadoop-Framework.

Allerdings kann die Analyse von Big Data sehr lange dauern und da sich die Eingabedaten jede Sekunde ändern, ist es vielleicht nicht immer die beste Idee, die vollständige Berechnung jedes Mal aufs Neue auf die kompletten Eingabedaten anzuwenden. Es wäre doch geschickter, sich das Ergebnis der vorherigen Berechnung zu betrachten und nur die Deltas zu analysieren, also Daten, die seit der letzten Berechnung hinzugefügt oder gelöscht würden. In dem Gebiet der selbstwartbaren materialisierten Sichten in relationalen Datenbanksystemen gibt es bereits mehrere Ansätze, die sich mit der Lösung dieses Problems beschäftigen. Eine Strategie liest nur die Deltas und inkrementiert oder dekrementiert die Ergebnisse der vorherigen Berechnung. Allerdings ist diese Inkrement-Operation sehr teuer, deshalb ist es manchmal besser, das komplette alte Ergebnis zu lesen und es mit den Deltas der Eingabedaten zu kombinieren.

In dieser Masterarbeit wird ein neues Framework namens Marimba vorgestellt, welches sich genau um diese Probleme der inkrementellen Berechnung kümmert. Einen MapReduce-Job in Marimba zu schreiben ist genau so einfach wie einen Hadoop-Job. Allerdings werden hier keine Mapper- und Reducer-Klasse implementiert, sondern eine Translator- und Serializer-Klasse. Der Translator ähnelt dem Mapper: Er bestimmt, wie die Eingabedaten gelesen und daraus Zwischenwerte berechnet werden. Der Serializer erzeugt die Ausgabe des Jobs. Wie diese Ausgabe berechnet wird, gibt der Benutzer durch Implementierung einiger Methoden an, um Werte zu aggregieren und invertieren.

Vier MapReduce-Jobs, darunter auch das Paradebeispiel für MapReduce WordCount, wurden im Marimba-Framework implementiert. Das Entwickeln von inkrementellen MapReduce-Jobs ist mit dem Framework extrem einfach geworden. Außerdem konnte mit Performanztests gezeigt werden, dass die inkrementelle Berechnung deutlich schneller ist als eine vollständige Neuberechnung.

Ein weiterer unter den vier implementierten Jobs berechnet Wortauftrittswahrscheinlichkeiten in geschriebenen Sätzen. Dies kann beispielsweise für Spracherkennung verwendet werden. Wenn ein Wort in einer gesprochenen SMS nicht richtig verstanden wurde, hilft der Algorithmus zu raten, welches Wort am wahrscheinlichsten an einer bestimmten Stelle stehen könnte, abhängig von den vorherigen Wörtern im Satz. Damit dieser Algorithmus auch brauchbare Ergebnisse liefert, ist die Menge und die Qualität der Eingabedaten wichtig. Durchaus brauchbare Ergebnisse wurden durch die Verarbeitung von Mil-

lionen von Twitter-Feeds, die deutsche Twitter-Nutzer in den letzten Monaten geschrieben haben, erreicht.

Abstract

The Hadoop framework for MapReduce jobs is a very popular tool for distributed calculations over big data. Working with Hadoop is simple: Define your input data source and implement the two methods Map and Reduce. You don't need to care about scheduling or fault tolerance. That's what the Hadoop framework does.

But analyzing big data can be very time-consuming and as data changes every second, it's not the best idea to do the calculation over the whole input data again and again. Why couldn't we just use the old result and simply analyze the delta, so the inserted and deleted data since the last computation? There are many approaches for solving this problem in the area of self-maintaining materialized views in relational databases. There's one strategy which just reads the delta and increments or decrements the values in the previous result. As such increments can be very expensive, it's sometimes better to use another strategy, which reads the complete old result and combines it with the delta.

In this thesis a new framework called Marimba is presented, which cares about all these problems and how to achieve self-maintenance. Writing a MapReduce job in Marimba is as simple as writing a Hadoop job. Instead of implementing a Mapper and Reducer class, a Marimba developer writes a Translator and Serializer class. The Translator is very similar to the Mapper, it defines how to read the input data and how to generate intermediate values. The Serializer produces the output of the job. And how this output is calculated, the user of the framework can tell in some methods to aggregate and invert values.

Four MapReduce-Jobs, including WordCount - the textbook example for MapReduce - were implemented in the Marimba framework. Incremental MapReduce jobs can be implemented easily now, but that's not all. Performance tests show that it's much faster to do an incremental instead of a full recomputation.

Another of the four implemented jobs calculate probabilities for words in a written sentence. This can be used in voice recognition. If you dictate the text of an SMS and a word could not be understood correctly, the algorithm helps guessing the word depending on the previous words in the sentence. For getting good results the amount and quality of input data is important, so we analyzed millions of Twitter feeds which people wrote in the last months.

Inhaltsverzeichnis

1 Einleitung

Suchmaschinen, Internetkaufhäuser und soziale Netzwerke. Dies sind drei Beispielanwendungen, bei denen enorme Mengen von Daten entstehen und analysiert werden. Die Suchmaschine durchforstet das Internet und erstellt eine Art Inhaltsverzeichnis, um Suchanfragen von Benutzern schnell und sinnvoll beantworten zu können. Im Internetkaufhaus gehen täglich Tausende von Bestellungen ein. Eine Analyse über die Bestellungen hilft, Trends herauszufinden und das Käuferverhalten zu erklären. Und auch im sozialen Netzwerk werden die Nutzer analysiert, um ihnen möglichst relevante Werbung zu zeigen oder ihnen Freunde und Seiten vorzuschlagen.

Die soeben genannten Analysen von Webseiten, Bestellungen und Personen sind sehr zeit- und rechenintensiv. Üblicherweise werden sie nur in bestimmten Intervallen ausgeführt, beispielsweise einmal nachts, wenn der Ansturm auf einer Webseite am geringsten ist. Dies sorgt natürlich dafür, dass die Berechnung eventuell ein paar Stunden alt ist und nicht mehr hundertprozentig korrekt ist. In den genannten und vielen weiteren Anwendungsfällen ist eine absolute Korrektheit der Daten jedoch gar nicht notwendig, es werden auch leicht veraltete Ergebnisse toleriert. Bei der Suchmaschine sorgt das dafür, dass eine brandneue Webseite erst Stunden oder Tage nach der Veröffentlichung gefunden werden kann. Benutzer von Suchmaschinen erwarten dies gar nicht anders. Das Internetkaufhaus liefert bei einer Analyse der Daten statistische Informationen über die aktuelle Situation, also etwa über Bestellungen bis zum Zeitpunkt, an dem die Analyse gestartet hat. Da sich das Käuferverhalten für gewöhnlich nicht stündlich ändert, ist auch hier keine absolute Aktualität erforderlich. Und im Falle des sozialen Netzwerks kann es bei leicht veralteten Statistiken passieren, dass die Anzahl der Freunde dritten Grades vielleicht um ein Prozent daneben liegt. Auch dies ist nicht tragisch.

Aufgrund der hohen Datenmengen dauern die Analysen oft sehr lange. Bei einer verteilten Berechnung auf mehreren leistungsstarken Rechnern lässt sich die Ausführungszeit jedoch immer weiter senken. Die Idee vom verteilten Rechnen ist fast so alt wie die Idee vom Computer selbst. Ansätze dafür gibt es viele, einige erfordern spezielle Hardware, andere erzwingen homogene Rechnertypen, wieder andere eine enorm aufwendige Programmierung der Software, welche sich um die Verteiltheit kümmert.

MapReduce ist ein von Google eingeführtes Framework für nebenläufige Berechnungen über große Datenmengen. Es existieren mehrere Implementierungen, eine davon das von der Apache Software Foundation entwickelte Hadoop, welches im Rahmen dieser Arbeit verwendet wurde.

MapReduce macht die Entwicklung von nebenläufigen Berechnungen für den Entwickler sehr einfach. Es wird vielfach eingesetzt und es gibt viele Beispielanwendungen. Jedoch fällt auf, dass viele der Anwendungen nicht für inkrementelle Berechnungen geeignet sind, sondern in jedem Anlauf alles komplett neu berechnen. Die Suchmaschine würde also bei ihrem täglichen Indizieren das alte Ergebnis verwerfen und alle Webseiten neu einlesen, das Internetkaufhaus kann nicht einfach nur die Bestellungen seit dem

letzten Analysevorgang betrachten, sondern muss alle Bestellungen neu analysieren. Und das soziale Netzwerk durchläuft alle Freundschaftsbeziehungen erneut, obwohl sich doch eventuell seit dem letzten Durchlauf gar nicht so viel geändert hat.

Damit eine Berechnung inkrementell ausgeführt werden kann, müssen viele Dinge beachtet werden. Die Entwicklung eines MapReduce-Jobs, welcher inkrementell arbeitet ist deutlich aufwendiger als die Entwicklung eines simplen Jobs, welcher eine komplette Neuberechnung durchführt. Dieses Problem wurde im Rahmen dieser Arbeit behandelt und ein Framework namens Marimba entwickelt, welches das einfache Erstellen von inkrementellen MapReduce-Jobs ermöglicht.

In dieser Arbeit wird zunächst in Kapitel 2 das Funktionsprinzip von MapReduce und Hadoop erklärt. Kapitel 3 stellt eine Idee vor, wie MapReduce-Jobs inkrementell arbeiten können. Die Ziele und Anforderungen an das Framework werden in Kapitel 4 genannt und auf die Implementierung wird in Kapitel 5 eingegangen. Einige typische Beispiele für MapReduce werden in Kapitel 6 vorgestellt und es wird erläutert, wie diese im Marimba-Framework implementiert werden. Nach einer Evaluation in Kapitel 7 folgt eine Zusammenfassung in Kapitel 8.

2 Grundlagen

Das im Rahmen dieser Arbeit entwickelte Framework basiert auf MapReduce und benutzt dazu das in Java geschriebene freie Framework Hadoop.

2.1 MapReduce

Der MapReduce-Algorithmus von Google für nebenläufige Berechnungen über große Datenmengen wurde inspiriert durch die in der funktionalen Programmierung oft verwendeten Funktionen map und reduce (siehe Abbildung 2.1).

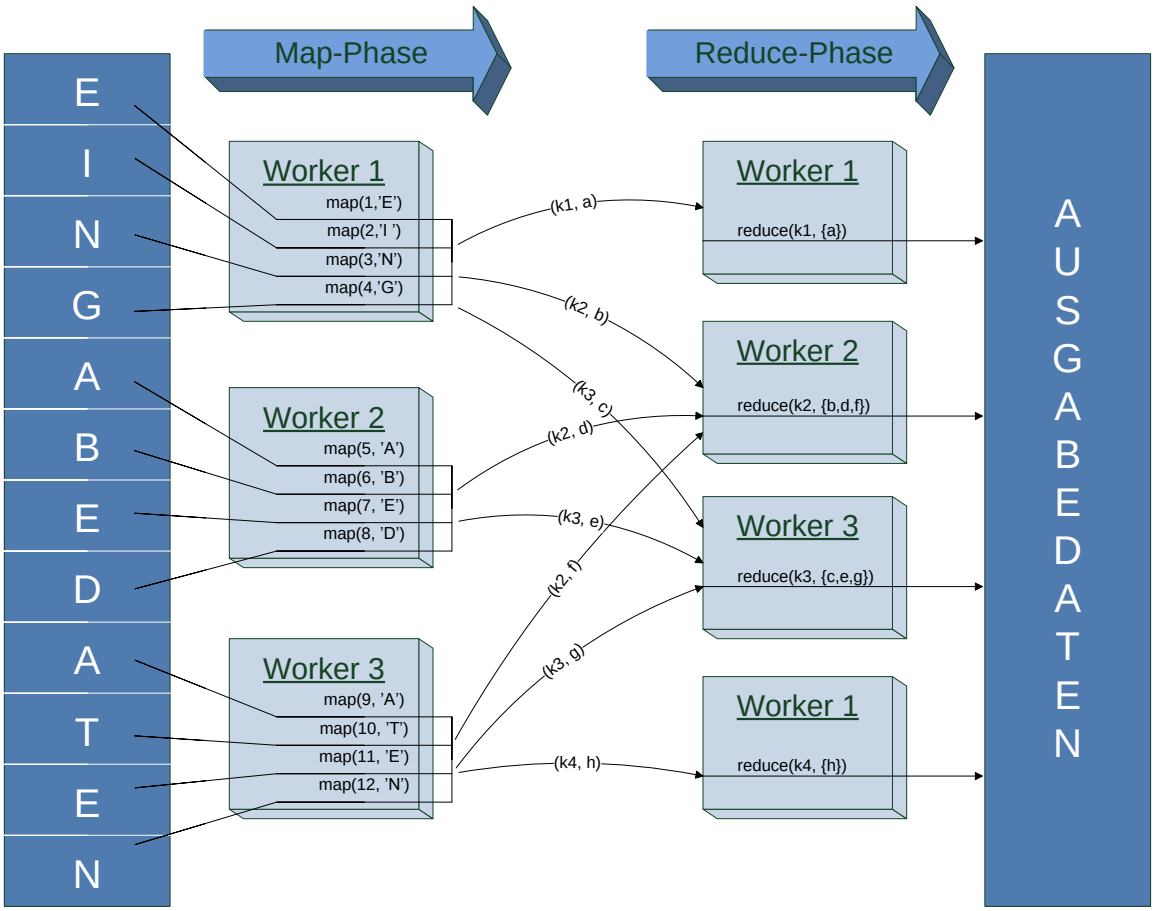


Abbildung 2.1: MapReduce

2.1.1 Map-Phase

Die Funktion *Map* kümmert sich um einen Teil der Eingabedaten (ein sogenannter Split). Die Eingabedaten sind Schlüssel-Wert-Paare und müssen unabhängig voneinander verarbeitbar sein. Die erste Phase des Map-Reduce-Algorithmus ist die Map-Phase, in welcher die Rechner im Cluster (sog. Worker) parallel die Map-Funktion auf die Schlüssel-Wert-Paare eines Splits der Eingabedaten anwenden. Auch die Ausgabe der Map-Funktion ist eine Menge von Schlüssel-Wert-Paaren, welche auch Intermediate-Key und Intermediate-Value genannt werden. Beendet der Worker die Durchführung der Map-Funktion, kann er sie erneut auf einen weiteren Split anwenden, bis schließlich alle verarbeitet wurden. Nun halten die Worker ihre berechneten Schlüssel-Wert-Paare bereit, welche in der zweiten Phase, der Reduce-Phase verwendet werden.

2.1.2 Reduce-Phase

Die *Reduce*-Funktion wird angewandt auf einen speziellen Schlüssel und die Menge der dazugehörigen Werte. Auch diese Funktion kann wieder parallel auf allen Workern ausgeführt werden. Dazu besorgt sich ein Worker, der sich um dem Schlüssel k_i kümmert, zunächst von allen anderen Workern die Intermediate-Values, die diese zu k_i bereit liegen haben. Auf diese Werte wird nun die Reduce-Funktion angewendet, welche überlichweise eine Ausgabe pro Schlüssel erzeugt. Diese Ausgabe kann beispielsweise ein Zeile sein, welche in eine Textdatei geschrieben wird oder eine Einfügung in eine Tabelle.

2.1.3 Beispiel: WordCount

Das Paradebeispiel für MapReduce ist der Algorithus WordCount, welcher die Anzahl der Vorkommnisse aller Wörter eines Textes zählt (siehe Algorithmus 1, aus [20]).

Algorithmus 1 WordCount

```
 1: function MAP(String id, String line)
 2:     for all word ∈ line do
 3:         emit(word, 1)
 4:     end for
 5: end function
 6:
 7: function REDUCE(String word, List⟨int⟩ counts)
 8:     sum ← 0
 9:     for all n ∈ counts do
10:         sum ← sum + n
11:     end for
12:     emit(word, sum)
13: end function
```

Die Eingabe ist hier ein beliebig langer Text, welcher Zeile für Zeile gelesen wird. Die Map-Funktion erzeugt für jedes Wort $word$ in einer Zeile das Schlüssel-Wert Paar $(word, 1)$. Die Reduce-Funktion addiert nun für ein spezielles Wort alle Werte (alles Einsen) auf und

liefert als Ausgabe die Gesamtanzahl der Vorkommnisse dieses Wortes im gesamten Dokument. Mehr zum WordCount-Algorithmus folgt in Kapitel 6.1.

2.1.4 Combiner

Um einen MapReduce-Job zu schreiben, müssen also nur die Funktionen Map und Reduce implementiert werden. Optional kann jedoch zusätzlich noch eine Combine-Funktion eingesetzt werden, welche der Reduce-Funktion ähnelt, also die Werte für einem bestimmten Schlüssel aufaggregiert. Allerdings führt diese der Mapper, also der Rechner, welcher die Map-Funktion ausführt, selbst aus. Die Menge der zu einem Schlüssel gehörigen Werte ist an dieser Stelle jedoch noch unvollständig, das endgültige Aufaggregieren erledigt wie gehabt der Reducer. Ziel des Combiners ist es, eine Art lokales Reduce auszuführen, um die Zwischenergebnisse, welche auf andere Rechner transportiert werden müssen, klein zu halten. Die Combine-Funktion für das WordCount-Beispiel sieht wie folgt aus:

Algorithmus 2 WordCount: Combiner

```
1: function COMBINE(String word, List⟨int⟩ counts)
2:     s ← 0
3:     for all n ∈ counts do
4:         s ← s + n
5:     end for
6:     emit(word, s)
7: end function
```

Während beim Fall ohne Combiner für die Reducer viele gleiche Worte bereitliegen, etwa $(\text{“}der\text{”}, 1), (\text{“}der\text{”}, 1), (\text{“}der\text{”}, 1)$, produziert der Combiner nun lokal einen zusammengefassten Eintrag, also $(\text{“}der\text{”}, 3)$. Weder der Mapper noch der Reducer müssen dafür verändert werden. Der Reducer erhält nun weniger Einsen, die er aufaddieren muss, stattdessen den zusammengefassten Eintrag.

Weitere Informationen über Combiner sind in [3], [14] und [27] zu finden. Dort wird empfohlen, immer einen Combiner zu verwenden, sofern es der Algorithmus erlaubt.

2.1.5 Anzahl der Mapper und Reducer

Die Anzahl der Mapper und Reducer ist entscheidend für die Performanz eines MapReduce-Jobs. Wird die Anzahl ungeschickt gewählt, kann es beispielsweise passieren, dass ein Rechner auf einen anderen lange warten muss. Mapper gibt es immer so viele, wie Splits vorhanden sind. Zum Beispiel lässt sich beim Texteingabeformat konfigurieren, aus wie vielen Megabytes ein Split beim Lesen einer Textdatei bestehen soll. Dies ist bei HDFS (siehe nächster Abschnitt) standardmäßig 64 MB, sodass bei einer ein Gigabyte großen Eingabedatei sechzehn Mapper gestartet werden. Besteht nun der Rechencluster aus fünf Rechnern, welche jeweils zwei Map-Aufgaben gleichzeitig ausführen, werden zunächst zehn Aufgaben parallel abgearbeitet und sechs müssen warten. Bei Beendigung einer Aufgabe kann sich der betreffende Rechner um den nächsten noch offenen Split kümmern.

Die Anzahl der Reducer hängt von der Anzahl sogenannter Partitionen ab. Eine Partition ist eine Menge von Schlüsseln (Intermediate-Keys), um die sich ein Reducer kümmert. Bei Hadoop wird standardmäßig ein Hash-Partitioner verwendet, welcher die Schlüssel über eine Hash-Funktion in eine vom Benutzer festgelegte Anzahl von Partitionen verteilt. So ist die Anzahl der Reducer also frei wählbar, während die Anzahl der Mapper von der Größe der Eingabedaten abhängt. In [19] wird empfohlen, die Anzahl der Reducer auf (0,95 oder 1,75 * Anzahl der Knoten * Aufgaben pro Knoten) zu setzen. Im ersten Fall (0,95) starten alle Reducer gleichzeitig, im zweiten Fall nur knapp die Hälfte, sodass sich diejenigen Rechner, welche als erstes mit ihrer Aufgabe fertig werden, um eine weitere Aufgaben kümmern können. Dies verhindert, dass eine Reducer-Aufgabe, welche um einiges länger dauert als andere, die gesamte Berechnung aufhält. Der Rechner, welche sich um diese lang dauernde Aufgabe kümmert, erledigt nur diese, die anderen Rechner erledigen in der Zeit jeweils zwei Aufgaben.

Auch in [14] wird empfohlen, nur eine kleine Anzahl von Reducern zu verwenden. Der Vorschlag ist hier, die Zahl knapp unter die Anzahl von Reduce-Slots zu setzen, was der oben genannten Formel mit dem Faktor 0,95 entspricht. Dies sind Erfahrungen aus Benchmarks, mehr dazu auch in Kapitel 7.

2.2 Hadoop

Hadoop [1] [25] ist ein in Java geschriebenes freies Framework welches auf dem MapReduce-Algorithmus basiert. Es wird von Yahoo, Facebook und vielen anderen prominenten Webseiten eingesetzt. Hadoop ist der De-Facto-Standard für die Verarbeitung von Big Data [22] und soll laut einer Studie [18] ähnlich erfolgreich werden wie Linux.

Um Hadoop zu verwenden, muss der Benutzer die aktuelle Hadoop-Version als JAR-Datei herunterladen und in seinem Java-Projekt einbinden. Nun können MapReduce-Jobs durch Implementierung der Schnittstelle `Tool` und überschreiben der Methoden `map` und `reduce` in den Klassen `Mapper` und `Reducer` erstellt werden. Zusätzlich wird der Job konfiguriert, indem Eingabeformat, Ausgabeformat und Dateitypen für Intermediate-Key und -Value gesetzt werden. Als Ein- und Ausgabe eignen sich sowohl Textdateien, welche im HDFS, als auch Tabellen, welche in HBase vorliegen.

2.2.1 HDFS

Das *Hadoop Distributed File System* [16] ist ein verteiltes Dateisystem, welches Dateien in mehreren Datenblöcken zu üblicherweise je 64 MB ablegt. Durch die Kopie dieser Blöcke auf mehrere (standardmäßig drei) Rechner im Cluster werden Zuverlässigkeit und Geschwindigkeit gesteigert, ohne dass sich das Programm, welches auf Daten lesend oder schreibend zugreifen möchte, um die Verteilung kümmern muss. Ein Ausfall eines Rechnerknotens kann also durch diese Redundanz verlustfrei kompensiert werden. Des weiteren entfällt der Transport von Eingabedaten zum Mapper, da dieser bevorzugt Splits verarbeitet, welche er lokal gespeichert hat.

2.2.2 HBase

HBase [7] ist die zu Hadoop gehörige spaltenorientierte Datenbank, welche Googles Big Table implemeniert. Anders als bei relationalen Datenbanken besteht eine HBase-Tabelle nicht aus festgelegten Spalten, sondern viel mehr aus sogenannten Spalten-Familien (Column Families), welche beliebige Spalten beinhalten können. Beim Einfügen einer Zeile in eine HBase-Tabelle, wird ein eindeutiger Schlüssel (Row Key) angegeben sowie die Namen (Column Qualifier) und Werte beliebiger Spalten. Das folgende Beispiel zeigt, wie über die HBase-Konsole eine Tabelle $person$ mit einer Spalten-Familie $default$ angelegt, und wie eine Zeile eingefügt und abgerufen wird.

```
> create 'person', 'default'
> put 'person', 'p27', 'default:vorname', 'Anton'
> put 'person', 'p27', 'default:nachname', 'Schmidt'

> get 'person', 'p27'
COLUMN                CELL
 default:nachname     timestamp=1338991497408, value=Schmidt
 default:vorname      timestamp=1338991436688, value=Anton
2 row(s) in 0.0640 seconds
```

Die eingefügte Zeile mit dem Schlüssel $p27$ besitzt also zwei Spalten, $vorname$ und $nachname$. Andere Zeilen können durchaus andere Spalten besitzen. Bei HBase ist es üblich, dass eine Zeile aus meheren Hundert oder Tausend Spalten besteht. Eine Suche ist jedoch nur über den Row Key möglich. Sekundäre Zugriffspfade werden von HBase eben so wenig unterstüzt wie zeilenübergreifende Transaktionen.

Jede Spalte hat neben einem Wert auch einen Zeitstempel. Dies ermöglicht die Versionierung der Daten. Bei einem Überschreiben einer Spalte wird also der alte Wert nicht gelöscht, sondern ein neuer Wert mit dem aktuellen Zeitstempel eingefügt. Eine Leseoperation liefert standardmäßig den neusten Wert, es ist jedoch auch möglich die Werte zu einem beliebigen Zeitpunkt in der Vergangenheit abzurufen.

HBase baut auf HDFS auf und braucht sich nicht um Lastverteilung und Replikation zu kümmern, da HDFS dies erledigt. Dass also z.B. jeder Datenblock von HDFS dreimal repliziert wird, ist aus der Sicht von HBase unsichtbar.

Datenbanken wie HBase, die einen nicht-relationalen Ansatz verfolgen, zählen zu den sogenannten NoSQL-Datenbanken („Not Only SQL“). Sie sind optimiert für große Datenmengen sowie schnellen und simplen Datenzugriff, etwa durch die Vermeidung von Joins. Oft werden NoSQL-Datenbanken nicht als Alternative sondern als Ergänzung zu relationalen Datenbanken angesehen [26]. Beispielsweise kann bei einem Internetkaufhaus primär eine relationale Datenbank verwendet werden, zur Analyse werden die Daten jedoch in eine NoSQL-Datenbank übertragen und mittels MapReduce verarbeitet. Falls gewünscht, kann das Ergebnis wieder in die ursprüngliche Datenbank geschrieben werden.

2.3 Materialisierte Sichten

Eine Sicht bezeichnet in einer relationalen Datenbank eine virtuelle Relation, welche über eine gepeicherte Abfrage definiert wird. Bei jeder Abfrage auf einer Sicht, wird ihr Inhalt neu berechnet. Anders ist dies bei materialisierten Sichten, welche vor allem bei aufwändigen Berechnungen Sinn machen. Hier wird das Ergebnis der Sicht vorberechnet und kann somit schnell abgerufen werden. Jedoch muss bei einem Einfügen, Ändern und Löschen auf Sätzen der Basistabellen die materialisierte Sicht aktualisiert werden. Man unterscheidet dabei zwischen direktem Aktualisieren, sodass die Sicht stets aktuell ist, sowie verzögertem Aktualisieren, beispielsweise einmal nachts. Für gewöhnlich wird man sich bei lang andauernden Berechnungen für letzteres entscheiden (vergleiche mit Kapitel 1).

Eine materialisierte Sicht wird *selbstwartbar* genannt, wenn sie bei Änderungen in der Lage ist, diese einzubringen, ohne eine vollständige Neuberechnung durchzuführen. Die Sicht generiert ihren neuen Stand also nur durch Betrachtung der Änderungen und ihres alten Standes.

In [12] und [11] werden die Ähnlichkeiten von MapReduce-Jobs zu materialisierten Sichten genannt und überprüft, in wie fern sich bekannte Techniken zur Selbstwartbarkeit auf MapReduce übertragen lassen. Mehr dazu in Kapitel 3.

3 Vorangegangene Arbeiten

Im Rahmen des Virga Projekts [2] untersucht die AG Heterogene Informationssysteme die inkrementelle Neuberechnung von MapReduce-Ergebnissen. Dabei werden Verfahren zur Selbstwartkeit von materialisierten Sichten, welche bereits im Kontext relationaler Datenbanken erforscht wurden, in die MapReduce-Umgebung übertragen und entsprechend angepasst. Diese Arbeit baut auf den Ergebnissen der folgenden Bachelorarbeiten und wissenschaftlichen Publikationen auf.

3.1 Inkrementelle Neuberechnungen in MapReduce

In der Bachelorarbeit von Roya Parvizi [20] wurden einige typische Anwendungsfälle, die in Googles ursprünglichem MapReduce-Papier [6] beschrieben wurden, untersucht. Dazu zählen WordCount, Reverse Web-Link Graphs, URL-Zugriffshäufigeiten, Wort-Histogramme pro Host und invertierte Wortlisten. Einige dieser Anwendungen sind auch in dieser Arbeit in Kapitel 6 wiederzufinden. In der genannten Bachelorarbeit wurden für jeden Anwendungsfall MapReduce-Jobs zur inkrementellen Wartung entwickelt. Es konnte gezeigt werden, dass die inkrementelle Berechnung deutlich performanter ist als eine vollständige Neuberechnung. Je weniger Änderungen an den Basisdaten vorgenommen wurden, desto mehr machte sich dies in der Ausführungszeit positiv bemerkbar.

3.2 Inkrementelle Wartung von Data Cubes

Im Rahmen der Bachelorarbeit von Marc Schäfer [23] wurde ein weiterer Anwendungsfall untersucht: Die Berechnung von Data Cubes. Dies sind die Menge aller möglichen Gruppierungen in einer Tabelle, z.B. die Anzahl der Besucher einer Webseite pro unterschiedlichem Browser, pro Land, pro Browser und Betriebssystem, usw. In [17] wurde vorgeschlagen, Data Cubes mit MapReduce zu berechnen, allerdings wurde die inkrementelle Neuberechnung nicht betrachtet. Dieser Anwendungsfall ist komplexer als die bereits untersuchten, da die Aufgabe in mehrere Teilaufgaben zerlegt werden musste und Sequenzen von MapReduce-Jobs zur Berechnung verwendet wurden. Dennoch konnte auch hier im inkrementellen Fall eine Effizienzsteigerung erreicht werden.

3.3 Techniken zur Selbstwartbarkeit in MapReduce

In der Publikation „Incremental Recomputations in MapReduce“ [12] wurden die Techniken zur Selbstwartbarkeit von materialisierten Sichten auf MapReduce übertragen. Ein MapReduce-Job, der nur die Änderungen seit der letzten Ausführung sowie das alte Ergebnis betrachtet, kann in vielen Fällen schneller sein als eine vollständige Neuberechnung.

3.3.1 Overwrite-Strategie

Die inkrementelle Variante des in Kapitel 2 vorgestellten WordCount-Algorithmus sieht wie folgt aus (aus [12]):

Algorithmus 3 Inkrementeller WordCount - Overwrite Installation

```
1:  function MAP(String key, String value)
2:      if key is inserted then
3:          for all word ∈ value do
4:              emit(word, 1)
5:          end for
6:      else if key is deleted then
7:          for all word ∈ value do
8:              emit(word, -1)
9:          end for
10:     else
          emit(key, value)
11:     end if
12: end function
```

Die Eingabe dieser neuen Map-Funktion sind drei Arten von Schlüssel-Wert-Paaren: eingefügte und gelöschte Daten sowie Ergebnisse der vormaligen Berechnung. Es muss also gekennzeichnet werden, zu welchen der drei Arten ein Eingabetupel gehört, damit in der Map-Funktion unterschieden werden kann, wie mit der Eingabe umgegangen wird. Sind es eingefügte Daten, passiert das gleiche wie beim normalen WordCount-Algorithmus: Jedes Wort in der vorliegenden Zeile wird mit dem Wert 1 weitergegeben. Gelöschte Daten, also Zeilen, die in der vorherigen Berechnung noch enthalten waren und nun nicht mehr vorhanden sind, werden negativ gezählt, also durch die Ausgabe einer -1. Die dritte Art von Daten ist eine Zeile der vorherigen Berechnung. Der Schlüssel ist hier ein bestimmtes Wort und der Wert die Anzahl der Vorkommnisse dieses Wortes bei der vorherigen Berechnung.

Die alten Ergebnisse werden also einfach weitergeschleift, hinzugefügte Wörter dazugezählt und gelöschte Wörter abgezogen. Der Reducer bleibt bei diesem MapReduce-Job unverändert. Er addiert zur alten Zählung die Anzahl der neuen Wörter hinzu und zieht durch Addieren von -1-Werten die gelöschten Wörter ab. Die Ausgabe dieses MapReduce-Jobs ist eine vollständige Tabelle mit allen Anzahlen aller Wörter. Es ist also möglich, das Ergebnis in eine andere Tabelle zu schreiben als bei der vorherigen Berechnung. Üblicherweise wird jedoch die gleiche Tabelle, von der gelesen wird, wieder überschrieben. Aus diesem Grund nennt man diese Variante der inkrementellen Neuberechnung *Overwrite Installation*.

3.3.2 IncDec-Strategie

Die Alternative zur Overwrite Installation ist die *Increment Installation* (auch IncDec genannt), bei der das alte Ergebnis nicht gelesen werden muss. Stattdessen werden nur die eingefügten und gelöschten Daten verarbeitet und die Änderungen auf die bereits existie-

rende Tabelle angewendet, siehe Algorithmus 4.

Algorithmus 4 Inkrementeller WordCount - Increment Installation

```
1:  function MAP(String key, String value)
2:      if key is inserted then
3:          for all word ∈ value do
4:              emit(word, 1)
5:          end for
6:      else if key is deleted then
7:          for all word ∈ value do
8:              emit(word, -1)
9:          end for
10:     end if
11: end function
12:
13: function REDUCE(String word, List⟨int⟩ values)
14:     sum ← 0
15:     for all n ∈ values do
16:         sum ← sum + n
17:     end for
18:     inc(word, 'myColFamily', 'myColQualifier', sum)
19: end function
```

Da anders als bei der Overwrite Installation die Map-Funktion nicht die Ergebnisse der vorherigen Berechnung liest, wird hier nur zwischen eingefügten und gelöschten Daten unterschieden. Weiterhin fällt auf, dass die Ausgabe der Reduce-Funktion ein Inkrement ist. Das heißt, das die berechnete Summe nicht in eine Spalte geschrieben wird, sondern auf den bereits in einer Spalte stehenden Wert aufaddiert wird. Diese Operation ist deutlich teuerer als das schlichte Überschreiben. Ist ein Wort noch nicht in der Tabelle vorhanden, wird es neu in die Tabelle eingefügt. Vorteil der IncDec-Strategie ist, dass bei nur wenigen Änderungen ein Großteil der bereits berechneten Ergebnisse unberührt bleiben. Es werden lediglich die Zeilen gelesen, welche auch geschrieben werden. Eine Optimierung der Reduce-Funktion ist eine Überprüfung, ob sum ungleich 0 ist. Denn ein Wort, welches genau so oft hinzugefügt wie gelöscht wird, verändert das Ergebnis nicht.

3.3.3 Formalisierung

In diesem Abschnitt wird die Klasse der selbstwartbaren MapReduce-Jobs definiert. Was im Beispiel des WordCount-Algorithmus die Addition von natürlichen Zahlen war - also die Addition von bereits berechneten Wortanzahlen sowie positiven und negativen Einsen -, ist im allgemeinen Fall eine Aggregationsfunktion $\circ$ auf Werten eines beliebigen Wertetyps D_v.

Vorher müssen jedoch die Eingabedaten in Schlüssel-Wert-Paare übersetzt werden. Dies geschieht mit einer Funktion *translate*. Die Eingabedaten sind vom Typ $D_{sk} \times D_{sv}$, die Intermediate-Key-Values vom Typ $D_k \times D_v$. Die Intermediate-Values bilden zusammen mit der Aggregatsfunktion $\circ$ eine abelsche Gruppe $(D_v, \circ)$. Die für eine abelsche Grup-

pe notwendigen inversen und neutralen Elemente werden im folgenden über die beiden Funktionen $*$ und *e* definiert:

- Eine Funktion *translate* : $D_{sk} \times D_{sv} \to \mathcal{P}(D_k \times D_v)$
- Eine Funktion $\circ : D_v \times D_v \to D_v$
- Eine Funktion $* : D_v \to D_v$, welche jedes Element der abelschen Gruppe $(D_v, \circ)$ auf sein inverses Element abbildet.
- Eine Funktion $e : \varnothing \to D_v$, welche das neutrale Element der abelschen Gruppe $(D_v, \circ)$ ausgibt.

Für das Beispiel WordCount sind die Eingabedaten vom Typ $\mathbb{Z} \times$ Text, also Zeilennummer und der Text der Zeile. Die translate-Funktion zerlegt die Zeile in ihre Wörter und liefert eine Menge von (Text $\times \mathbb{Z}$)-Paaren, sprich das Wort und dessen Anzahl ($= 1$). Die Funktion $\circ$ ist die $+$-Operation auf den ganzen Zahlen $\mathbb{Z}$, das inverse Element $v*$ zu v ist $-1 \cdot v$, das neutrale Element e ist 0.

Das allgemeine MapReduce-Programm besteht nun aus einer Map-Funktion, die die Werte für eingefügte Daten mit der translate-Funktion in Schlüssel-Wert-Paare übersetzt. Gelöschte Daten werden ebenfalls übersetzt, jedoch danach invertiert. Alte Ergebnisse werden bei der Overwrite-Strategie einfach weitergeben. Die Reduce-Funktion nimmt nun das alte Ergebnis und die Werte von den eingefügten Daten sowie die invertierten Werte von gelöschten Daten und aggregiert mittels der $\circ$-Funktion das Ergebnis, welches daraufhin geschrieben werden kann.

Wird statt der Overwrite-Strategie IncDec verwendet, gibt es keine alten Daten als Eingabe. Der Reducer aggregiert die Werte (normale und invertierte) auf und schreibt sie als Inkrement in die Datenbank. Im WordCount-Beispiel kann dies eine positive oder auch negative Zahl sein. Mit den vorliegenden Funktionen lässt sich auch die Strategie *Vollständige Neuberechnung* formalisieren. Die Map-Funktion übersetzt dabei alle Eingaben mit der translate-Funktion, die Reduce-Funktion aggregiert die Werte eines Schlüssels auf und schreibt sie.

Ein Combiner arbeitet ähnlich wie der Reducer: Er aggregiert also die bereits vorliegenden Werte zu einem Schlüssel, siehe Algorithmus 5.

Algorithmus 5 Combiner

```
1: function COMBINE(key, value)
2:     temp ← e
3:     for all v ∈ values do
4:         temp ← temp ∘ v
5:     end for
6:     emit(key, temp)
7: end function
```

Weitere Details, Algorithmen und Funktionen sind in [12] beschrieben.

4 Entwurf eines Frameworks

4.1 Frameworks

Ein Framework oder auch Programmiermodell oder Programmgerüst wird in [21] wie folgt definiert: „Ein Programmgerüst ist ein erweiterbares und anpassbares System von Klassen, das für einen allgemeinen, übergeordneten Aufgabenbereich eine Kernfunktionalität mit entsprechenden Bausteinen bereitstellt. Die Kernfunktionalität und die Bausteine müssen eine geeignete Grundlage bieten, um die Bewältigung unterschiedlicher konkreter Ausprägungen der allgemeinen Aufgabenstellung erheblich zu vereinfachen. Programmgerüste sind also Systeme von Klassen, die speziell für die Wiederverwendung bzw. Mehrfachverwendung entwickelt werden."

4.2 Das Hadoop MapReduce Framework

Ein Beispiel für ein Programmgerüst ist das Hadoop-Framework. Die Kernfunktionalität ist die verteilte Ausführung von Jobs auf mehreren Rechnern. Hadoop ist ein in Java geschriebenes Programmgerüst. Der Benutzer verwendet es, indem er Klassen wie *Tool*, *Mapper* oder *Reducer* vererbt und Methoden wie *map* und *reduce* implementiert. Für den WordCount-Job sieht die Mapper-Klasse wie folgt aus:

```
public static class WordCountMapper extends
 Mapper<LongWritable, Text, ImmutableBytesWritable, LongWritable> {

    private Text word = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException {
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(new ImmutableBytesWritable(
                this.word.getBytes(), 0, this.word.getLength()),
                new LongWritable(1);
        }
    }
}
```

Durch Vererbung der Klasse Mapper unter Angabe der Parametertypen für die Ein- und Ausgabetypen sowie Implementierung der Methode `map` gibt der Benutzer das Verhalten der Mapper an. Auf gleiche Weise wird der Reducer definiert:

```
public static class WordCountReducer extends
 TableReducer<ImmutableBytesWritable, LongWritable, Writable> {
   @Override
   public void reduce(ImmutableBytesWritable key,
    Iterable<LongWritable> values, Context context)
    throws IOException, InterruptedException {
      long sum = 0;
      for (LongWritable value : values) {
         sum += value.get();
      }
      Put put = new Put(key.get());
      put.add(Bytes.toBytes("default"), Bytes.toBytes("details"),
      Bytes.toBytes(sum));
      context.write(key, put);
   }
}
```

In der Methode `reduce` liegen die einzelnen Werte als Iterable-Objekt vor, über die in einer Schleife iteriert werden kann. In dem Beispiel sind diese Werte vom Typ LongWritable und werden aufaddiert, um schließlich in ein Put-Objekt gepackt werden zu können. Dieses Put-Objekt sorgt für ein Einfügen in eine HBase-Tabelle.

Der eigentliche lauffähige Hadoop-Job ist eine Implementierung der Schnittstelle Tool. In dieser Klasse wird der Job konfiguriert. Er erhält einen Namen („wordcount“), die Anzahl der Reducer wird festgelegt, Formate für Ein- und Ausgabe werden gesetzt und auf die soeben erstellten Mapper- und Reducer-Implementierungen verwiesen. Obwohl bereits bei der Implementierung des Mappers und Reducers angegeben wurde, welche Typen die Intermediate-Key und -Value haben, ist dies bei der Konfiguration eines Hadoop-Jobs nochmals erforderlich. Die Ein- und Ausgabeformate müssen ebenfalls konfiguriert werden, also in diesem Fall der Pfad zu den Eingabedateien und der Name der Ausgabe-Tabelle:

```
public class WordCount implements Tool {
   @Override
   public int run(String[] args) throws Exception {
      Configuration conf = getConf();
      Job job = new Job(conf, "wordcount");
      job.setNumReduceTasks(18);

      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TableOutputFormat.class);

      job.setMapperClass(WordCountMapper.class);
      job.setReducerClass(WordCountReducer.class);

      job.setMapOutputKeyClass(ImmutableBytesWritable.class);
      job.setMapOutputValueClass(LongWritable.class);
```

```
        TableMapReduceUtil.initTableReducerJob("outputTable",
         WordCountReducer.class, job, HashPartitioner.class);

        job.setJarByClass(WordCount.class);
        TextInputFormat.setInputPaths(job, "/inputdata");

        return (job.waitForCompletion(true) == true) ? 0 : -1;
    }
}
```

Lediglich durch die Implementierung der Klassen `Mapper` und `Reducer` sowie die Konfiguration mit der Klasse `Tool` wird festgelegt, wie das MapReduce-Programm arbeitet. Der Benutzer gibt dabei an, wie die Eingabe interpretiert, die Berechnung durchgeführt und die Ausgabe geschrieben wird. Um alles weitere kümmert sich das Hadoop-Framework, also um die Koordination über die Rechner im Cluster, die Übertragung von Daten von einem Rechner zum anderen und die Fehlerbehandlung.

Das praktische am MapReduce-Framework - in diesem Fall Hadoop - ist also, dass der Benutzer nur die Methoden *map* und *reduce* für seinen Anwendungsfall implementiert und sich um den Rest keine Gedanken machen muss.

4.3 Das Marimba-Framework

4.3.1 Anforderungen

Soll der MapReduce-Job selbstwartbar sein und inkrementelle Neuberechnungen unterstützen, muss dies manuell entwickelt werden. Das heißt im Mapper muss unterschieden werden, um welche Art von Daten es sich handelt: Sind sie neu, gelöscht oder bereits in die vorherige Berechnung eingeflossen. Wird die Overwrite-Strategie benutzt, dient die Tabelle mit den alten Ergebnissen als zusätzliche Eingabe. Dazu muss der Benutzer zuerst ein neues Eingabeformat entwickeln, welches aus dieser Tabelle die alten Ergebnisse liest und aus einer anderen Tabelle oder Textdatei die neuen und gelöschten Daten. Bei der Strategie IncDec benötigt der Reducer einige Veränderungen, da Increment- statt Put-Objekte erzeugt werden.

Alle diese Punkte sind im wesentlichen nicht anwendungsspezifisch, also für jeden selbstwartbaren Job auf gleiche Weise zu entwickeln. Um dem Benutzer diese Arbeit abzunehmen, wurde basierend auf den in Kapitel 3 vorgestellten Ansätzen zur inkrementellen Neubrechnung im Rahmen dieser Masterarbeit das Framework *Marimba* entwickelt. Es baut auf Hadoop auf und stellt die Kernfunktionalität jedes selbstwartbaren MapReduce-Jobs zur Verfügung. Der Benutzer braucht sich nur noch um folgende anwendungsspezifische Dinge zu kümmern:

- Lesen der neu eingefügten Eingabedaten
- Aggregation und Invertierung
- Schreiben der Ausgabedaten

Durch die Angabe, wie neu eingefügte Eingabedaten interpretiert werden und wie sich Objekte invertieren lassen, ist das Framework in der Lage, auch gelöschte Eingabedaten zu behandeln. Des weiteren kümmert sich das Framework um folgende nicht anwendungsspezifische Punkte:

- Implementierung der IncDec- und Overwrite-Strategie
- Lesen der alten Ergebnisse bei Verwendung der Overwrite-Strategie
- Erstellung von Increment-Objekten bei Verwendung der IncDec-Strategie

4.3.2 Entwurf

Eine mit dem Marimba-Framework programmierte Software ist genau wie bei Hadoop ein MapReduce-Job. Dieser soll auf einem Hadoop-Rechencluster genau so ausführbar sein wie ein direkt in Hadoop entwickelter Job. Allerdings implementiert der Benutzer hier nicht die Methoden *map* und *reduce*, sondern die Methoden *translate*, *invert* usw. (siehe Kapitel 3.3.3). Ein Entwickler, der bereits mit dem Hadoop-Framework vertraut ist, sollte ohne eine lange Einarbeitungszeit in der Lage sein, einen Marimba-Job zu erstellen. Aus diesem Grund wurde bei der Entwicklung des Frameworks darauf geachtet, dass es auf ähnliche Weise zu benutzen ist wie Hadoop. Ein Paar Beispiele:

- `class MyTranslator extends Translator<LongWritable, Text>`

  Durch Vererbung der Klasse Translator mit den Parametertypen der Eingabeschlüssel und -werte erstellt der Benutzer den Translator, welcher die Eingabedaten übersetzt. Auf gleiche Weise wird auch in Hadoop der Mapper erstellt:

  ```
  class MyMapper extends Mapper<LongWritable, Text, Text, Text>
  ```

  Die beiden hinteren Parametertypen sind die Typen der Intermediate-Keys und Intermediate-Values. Da diese jedoch zusätzlich noch an einer anderen Stelle definiert werden, lässt man sie im Marimba-Framework einfach weg.

- `IncJob job = new IncJobOverwrite(conf);`

  Dies legt einen neuen Job an, und zwar in diesem Fall einen Job, welcher die die inkrementelle Neuberechnung mit der Strategie Overwrite durchführt. Genau wie bei Hadoop

  ```
  Job job = new Job(conf);
  ```

  nimmt der Konstruktor des Jobs ein Konfigurationsobjekt.

- `job.setTranslatorClass(MyTranslator.class);`

  Hier wird dem Job mitgeteilt, welche Translator-Klasse verwendet werden soll. Dadurch ist es möglich, einen Translator für mehrere Jobs zu verwenden. Hadoop-Programmierer sind mit der Konfiguration mittels Class-Objekten bereits vertraut:

```
job.setMapperClass(MyMapper.class);
job.setInputFormatClass(TextInputFormat.class);
```

Das zweite Statement setzt das Eingabeformat, hier Textdateien in HDFS. Dies funktioniert im Marimba-Framework auf gleiche Weise.

Das Erstellen eines Marimba-Jobs ist also so einfach wie das Erstellen eines Hadoop-Jobs: Einige Klassen müssen vererbt werden, Methoden werden implementiert und die Konfiguration des Jobs größtenteils über Class-Objekte vorgenommen. Neu im Marimba-Framework ist der Typ *Abelian*:

```
job.setAbelianClass(WordAbelian.class);
```

Dieser Typ beschreibt die Objekte, die der geschriebene Job erzeugen, invertieren, aggregieren und schreiben wird. Das Erzeugen eines Abelian-Objekts findet im Translator statt, welcher die Eingabedaten in abelsche Objekte übersetzt. Invertieren und Aggregieren sind Methoden in der Schnittstelle Abelian, die der Entwickler selbst implementiert:

```
public class WordAbelian implements Abelian<WordAbelian> {
 ...
 public WordAbelian invert() {
  return new WordAbelian(this.word, this.count*-1);
 }
 public WordAbelian aggregate(WordAbelian other) {
  return new WordAbelian(this.word, this.count+other.count);
 }
 public WordAbelian neutral() {
  return new WordAbelian(this.word, 0);
 }
 public boolean isNeutral() {
  return this.word.count == 0;
 }
 public ImmutableBytesWritable extractKey() {
   return new ImmutableBytesWritable(...);
 }
}
```

Die letzte Methode *extractKey* liefert den Teil des Objekts, welcher als Intermediate-Key verwendet wird. Das heißt, alle Objekte, welche dort den selben Schlüssel erhalten, werden in einem Reduce-Schritt verarbeitet, genauer: aggregiert.

Nun fehlt nur noch die Information, wie ein abelsches Objekt geschrieben werden kann. Da dies bei ein und der selben Abelian-Klasse auf unterschiedliche Weise passieren kann, legt der Benutzer eine oder mehrere Serializer-Klassen an:

```
public class WordSerializer implements Serializer<WordAbelian> {
 public Writable serialize(Writable key, Abelian<?> obj) { ... }
}
```

Die Methode *serialize* erzeugt ein schreibbares Objekt, also z.B. einen Text, welcher an eine HDFS-Textdatei angehangen werden kann oder ein Put-Objekt, welches eine Zeile in eine HBase-Tabelle einfügt.

Genau wie das Setzen der Abelian- und Translator-Klasse wird dies auch mit der Serializer-Klasse gemacht:

```
job.setSerializerClass(WordSerializer.class);
```

Möchte man beispielsweise den inkrementellen WordCount-Algorithmus mit dem Marimba-Framework implementieren, geht man dabei wie folgt vor:

- Implementierung einer Klasse `WordAbelian` mit Methoden zum Aggregieren und Invertieren
- Erstellen eines `WordTranslator`s, welcher aus den Eingabedaten `WordAbelian`-Objekte erzeugt
- Erstellen eines `WordSerializer`s, der ein `WordAbelian`-Objekt schreibbar macht, also z.B. in ein Put-Objekt wandelt, und in der Lage ist, Ergebnisse der vormaligen Berechnung als `WordAbelian`-Objekte einzulesen (für die Overwrite-Strategie)
- Implementierung eines Hadoop-Jobs, allerdings unter Verwendung der Marimba-Klasse `IncJob` mit Angabe der Strategie (Overwrite, ...) und durch Verweis auf die erstellten Klassen `WordAbelian`, `WordTranslator` und `WordSerializer`.

Der Marimba-Job für das Beispiel WordCount wird detailliert in Kapitel 6.1 vorgestellt und ist als Java-Code im Anhang zu finden.

Als Eingabeformat können im Marimba-Framework die klassischen Hadoop-Eingabeformate verwendet werden, wie TableInputFormat oder TextInputFormat, die aus HBase bzw. HDFS Daten lesen. Diese Formate geben jedoch keine Information darüber preis, ob ein Datum seit der letzten Berechnung gelöscht, eingefügt oder nicht verändert wurde, also geht das Framework davon aus, dass alle Daten neu sind. Für viele Jobs ist dies ausreichend, wenn die Berechnung auf Eingabedaten angewendet wird, die seit der letzten Berechnung dazugekommen sind. Sollen jedoch auch Löschungen unterstützt werden, muss der Benutzer ein Eingabeformat entwickeln, welches nicht einfache `Value`-Objekte ausgibt, sondern `InsertedValue`, `PreservedValue` und `DeletedValue`. Anhand dieser Typen kann das Framework entscheiden, wie mit den Eingabedaten umgegangen wird. Ein Beispiel für ein solches Eingabeformat ist das TextWindowInputFormat, welches in Kapitel 5.5 vorgestellt wird.

Abbildung 4.1 zeigt die wichtigsten Elemente des Marimba-Frameworks und wie sie miteinander zusammenhängen. Auf die Implementierungsdetails wird in Kapitel 5 eingegangen.

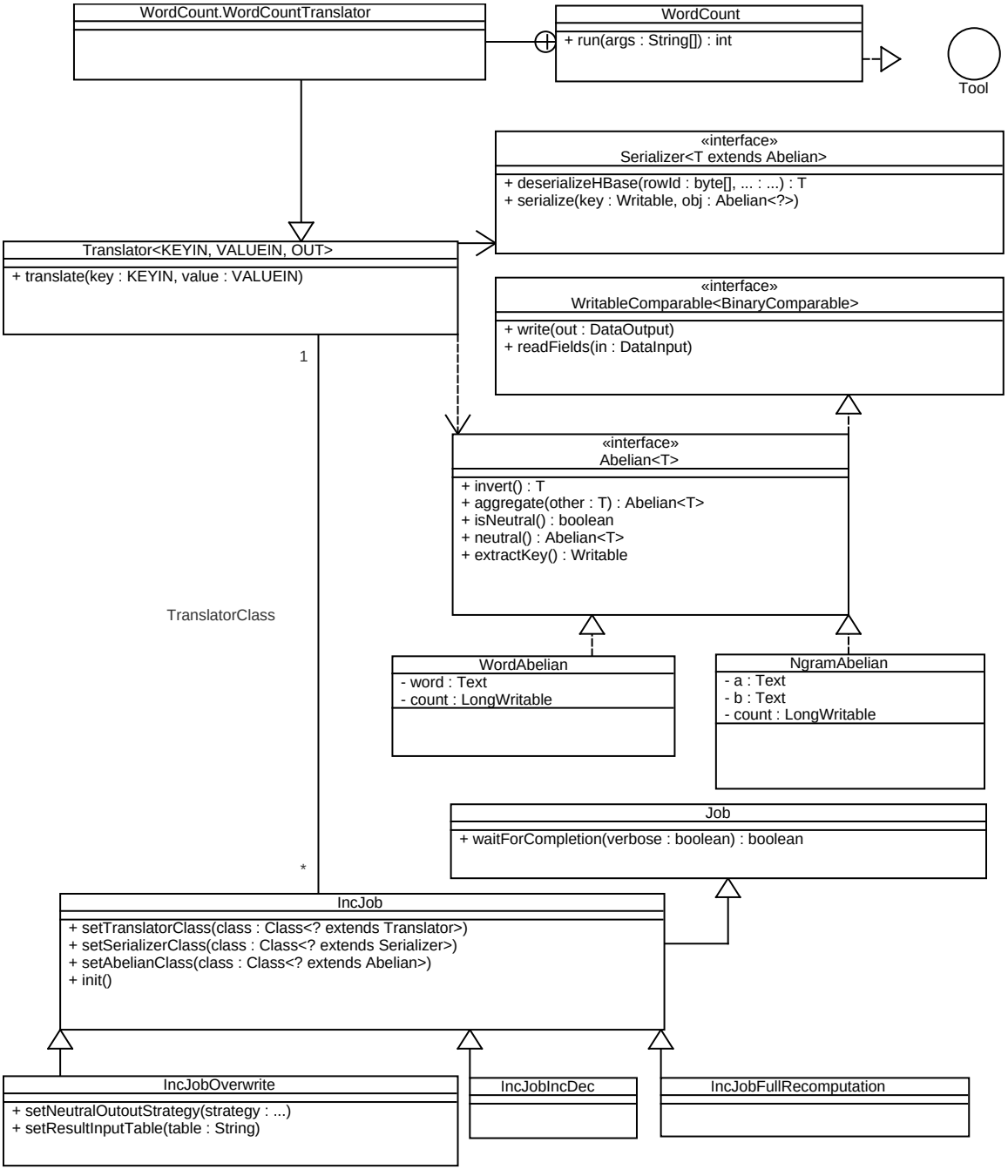


Abbildung 4.1: UML-Diagramm zum Marimba-Framework

5 Implementierung

Im vorherigen Kapitel wurde ein grober Überblick über das Marimba-Framework gegeben und es wurde erklärt, wie es zu benutzen ist. Weitere Details zur Benutzung sind in Kapitel 6 zu finden, in welchem einige Beispielanwendungen und deren Entwicklung vorgestellt werden. Dieses Kapitel beschäftigt sich mit den Details der Implementierung des Frameworks. Dazu gehört auf der einen Seite die Bereitstellung der Klassen und Methoden, welche der Benutzer erweitert und verwendet, um seinen Marimba-Job zu erstellen. Auf der anderen Seite beschäftigt sich dieses Kapitel mit der Abbildung des Marimba-Jobs auf einen Hadoop-Job, also beispielsweise die generelle Implementierung eines Mappers, welcher die vom Benutzer erstellte translate-Funktion aufrufen wird.

5.1 IncJob - Die abstrakte Klasse für inkrementelle MapReduce-Jobs

In Kapitel 4 wurde erklärt, wie ein inkrementeller Job initialisiert wird. Wie bei jedem Hadoop-Job ist auch die Hauptkomponente eines inkrementellen Jobs eine Klasse, welche die Schnittstelle *Tool* und damit auch *Configurable* implementiert. Dadurch wird der Job lauffähig gemacht und innerhalb der Methode *run* hat der Benutzer Zugriff auf das Konfigurationssobjekt (`this.getConf()`). Dies beinhaltet zu Beginn die Standardkonfiguration eines MapReduce-Jobs sowie über Kommandozeilenparameter gesetzte Eigenschaften. Dieses Konfigurationsobjekt wird bei der Erzeugung eines Jobs in Hadoop und auch eines IncJobs in Marimba übergeben.

Die Klasse IncJob ist jedoch abstrakt. Die drei Subklassen *IncJobFullRecomputation*, *IncJobIncDec* und *IncJobOverwrite* können instanziiert werden, je nachdem welche Strategie der Benutzer wünscht. Der folgende Befehl erzeugt einen neuen Overwrite-Job:

```
IncJob job = new IncJobOverwrite(conf);
```

Wichtige Methoden, welche IncJob bereitstellt, sind Setter zum Setzen der Translator-, Abelian- und Serializer-Klassen. In jedem Marimba-Job müssen diese drei Klassen festgelegt werden. Zusätzliche werden die Namen der Eingabe- und Ausgabetabelle über die Methoden *setInputTable* und *setOutputTable* gesetzt, vorausgesetzt HBase ist Eingabe- bzw. Ausgabeformat. Andernfalls bleibt der Name der Tabelle `null`.

Die eigentliche Arbeit passiert in der Methode *waitForCompletion*, welche aufgerufen wird, sobald der Job starten soll. Diese Methode überprüft, ob alle notwendigen Klassen erfolgreich gesetzt wurden. Ist dies nicht der Fall, wird ein Fehler ausgegeben, ansonsten können die Klassen der Intermediate-Keys und -Values gesetzt werden. Die Klasse des Schlüssels ist dabei der Ausgabetyp der Methode *extractKey* in der vom Benutzer implementierten *Abelian*-Klasse. Dieser kann mittels Introspection ermittelt werden. Die Klasse des Wertes ist die Abelian-Klasse selbst.

Abhängig von der Wahl der Strategie, also volle Neuberechnung, IncDec oder Overwrite, müssen noch weitere Schritte ausgeführt werden. Dies passiert in der *init*-Methode der von IncJob erbenden Subklassen:

5.1.1 IncJobFullRecomputation

Ein inkrementeller Job von dieser Klasse berechnet das Ergebnis jedes mal komplett neu. Er liest weder das alte Ergebnis noch erzeugt er Inkremente. Vor allem zum Performanzvergleich wird diese Klasse häufig gebraucht, da es interessant ist zu wissen, ob ein nativ in Hadoop geschriebener MapReduce-Job, welcher keine inkrementelle Neuberechnung durchführt genau so lange dauert wie ein Job der Klasse IncJobFullRecomputation. Ist dies der Fall, kann verglichen werden, um wie viel schneller ein Job ausgeführt wird, wenn er die Strategie Overwrite oder IncDec verwendet. Ergebnisse dazu sind in Kapitel 7 zu finden.

Die Klasse IncJobFullRecomputation initialisiert lediglich das Ausgabeformat. Abhängig davon, ob ein Ausgabetabellenname gesetzt wurde oder nicht, ist das Ausgabeformat *TableOutputFormat* bzw. *TextOutputFormat*. Ersteres schreibt in HBase zweiteres in HDFS. Dies sind die gängisten Ausgabeformate. Andere als diese beiden werden von Marimba momentan nicht unterstützt.

5.1.2 IncJobIncDec

Ein Job mit der Strategie IncDec erzeugt Inkremente, welche in HBase geschrieben werden. Dies sind Werte, die auf bereits in der Ausgabetabelle existierende Werte aufaddiert werden. Möglich wird dies durch das Setzen des Ausgabeformats auf *TableOutputFormatWithIncrementSupport* aus [12]. Wichtig ist, dass ein Tabellenname gesetzt wurde, denn die Strategie IncDec funktioniert nicht bei Textdateien als Ausgabeformat.

5.1.3 IncJobOverwrite

Die komplexeste Strategie ist Overwrite. Denn hier wird noch eine zusätzliche Eingabetabelle benötigt: Die Tabelle mit den alten Ergebnissen. Dazu wurde das Eingabeformat *OverwriteInputFormat* entwickelt, welches zwei Eingabeformate kombiniert: Das vom Benutzer gesetzte Format sowie ein zusätzliches *TableInputFormat*, um die alten Ergebnisse zu lesen.

Zuerst legt der Benutzer die Tabelle fest, in welcher die alten Ergebnisse zu finden sind (`IncJobOverwrite.setResultInputTable`). Außerdem wählt er auf gewohnte Art und Weise das Eingabeformat (`Job.setInputFormatClass`). Dieses Format wird sich in der Konfiguration unter *OtherInputFormat* gemerkt und schließlich überschrieben durch das *OverwriteInputFormat*.

Die Funktionsweise des *OverwriteInputFormat* ist angelehnt an das in [12] implementierte Format MultipleTableInputFormat, das es erlaubt aus mehreren HBase-Tabellen Daten zu lesen, sowie TextTableInputFormat, welches Daten aus einer Textdatei und einer Tabelle liest. Das Prinzip dahinter ist, dass das neue Eingabeformat die anderen Eingabeformate beinhaltet und die Splits, die diese erzeugen, zusammenlegt und als gesamtes

ausgibt. Beim OverwriteInputFormat wird also eine Instanz eines TableInputFormats erzeugt sowie eine Instanz einer erst zur Laufzeit bekannten Klasse, das OtherInputFormat. Diese beiden Eingabeformate erzeugen jeweils eine Liste mit Splits, welche als Eingabe für einen Mapper bestimmt sind. Diese Listen werden konkateniert, sodass die Mapper nun Splits von zwei unterschiedlichen Eingabeformaten erhalten: TableSplits mit Ergebnissen der letzten Berechnung sowie irgendwelche andere Splits aus dem OtherInputFormat, also z.B. ebenfalls TableSplits oder Ausschnitte aus einer Textdatei.

Dass der Mapper Splits von unterschiedlichen Eingabeformaten erhält, stellt zunächst kein Problem dar. Dass es für ihn jedoch nicht möglich ist, herauszufinden, um welche Art von Split es sich handelt, ist in der Tat ein Problem. Vor allem wenn sowohl die Eingabe als auch die alten Ergebnisse aus einer HBase-Tabelle gelesen werden, erhält der Mapper in beiden Fällen *Result*-Objekte. Die Marimba-Klasse *OverwriteResult* löst dieses Problem. Da sie eine Subklasse von *Result* ist, kann sie überall dort verwendet werden, wo ein *Result*-Objekt erwartet wird. Mittels Delegation ist es im *OverwriteTableRecordReader* möglich, ein *Result*-Objekt in einem *OverwriteResult*-Objekt zu verpacken. Im Mapper kann dann mittels `instanceof`-Überprüfung ermittelt werden, ob es sich bei dem vorliegenden *Result* um eine Zeile in der Ergebnistabelle handelt oder um neue Daten (siehe Kapitel 5.4.1).

Das OverwriteInputFormat bietet Marimba-Benutzern einen großen Vorteil, da sie kein eigenes Eingabeformat entwickeln müssen, wenn die Strategie Overwrite benutzt werden soll. Unabhängig von der gewählten Strategie gibt der Benutzer lediglich das Eingabeformat der Datenquelle an, der Rest passiert - versteckt für den Benutzer - innerhalb des Frameworks.

Bei der Strategie Overwrite macht es Sinn, ein Ergebnis nicht zu schreiben, wenn es neutral ist. Steht etwa beim WordCount-Algorithmus in den alten Ergebnissen, dass ein Wort 100 mal gezählt wurde und nun wurde es (-100) mal dazuaddiert (durch Invertieren bei gelöschten Eingabetupeln), ist die neue Anzahl 0. Wird das Ergebnis in eine noch leere Tabelle geschrieben, ist es unnötig, diese Null zu schreiben, das Wort kann einfach weggelassen werden. Wird jedoch in die gleiche Tabelle geschrieben, in der die alten Ergebnisse bereits stehen, möchte man vielleicht doch die 0 hinschreiben, da andernfalls der alte Wert stehen bleibt. Dies ist jedoch nicht tragisch, da er einen alten Zeitstempel besitzt, während alle Werte der neuen Berechnung einen höheren Zeitstempel besitzen. Eine wieder andere Alternative ist, dass die Zeile aus der Tabelle sogar gelöscht wird. Zusammengefasst gibt es folgende drei Strategien, welche der Benutzer mit der Methode `IncJobOverwrite.setNeutralOutputStrategy` setzen kann:

- *PUT*: Obwohl die Ausgabe neutral ist, wird sie geschrieben. Im WordCount-Beispiel bedeutet dies, dass Wörter mit der Anzahl 0 in der Tabelle stehen werden.
- *DELETE*: Bei einer neutralen Ausgabe wird ein *Delete*-Objekte erzeugt, welches dafür sorgt, dass die Zeile aus der HBase-Tabelle gelöscht wird.
- *IGNORE*: Neutrale Ausgaben werden nicht geschrieben. Alte Werte bleiben mit einem alten Zeitstempel in der Tabelle erhalten.

Die Standardstrategie ist *PUT*, da dies auch das Verhalten bei IncDec ist: Steht in der Tabelle mit den alten Ergebnissen, dass ein Wort 100 mal gezählt wurde und es wird nun 100 mal gelöscht, schreibt der Reducer ein Increment-Objekt mit dem Wert -100. In der Tabelle steht nun der Wert 0. Bei der IncDec-Strategie, ist leider nur dieses Verhalten (PUT)

möglich. Sollen neutrale Zeilen gelöscht werden, muss der Benutzer entweder einen separaten MapReduce-Job schreiben, der dies tut oder einen Trigger in HBase einrichten, welcher Zeilen löscht, sobald sie neutral werden.

5.2 Die Klasse Abelian

In [12] wurde ein selbstwartbarer MapReduce-Job über die Funktionen *translate*, $\circ$, $*$ und *e* definiert (siehe Kapitel 3.3.3). Bis auf die Funktion *translate* sind diese Operationen im Marimba-Framework als Methoden im Interface *Abelian* implementiert:

```
public interface Abelian<T extends Abelian<?>>
extends WritableComparable<BinaryComparable>
```

Durch die Parametrisierung wird es möglich, dass die Methoden den implementierten Abelian-Typ selbst ausgeben. Somit erfolgt die Implementierung eines solchen Typs wie folgt:

```
public class WordAbelian implements Abelian<WordAbelian>
```

Das Erzeugen von Abelian-Objekten passiert im vom Benutzer erstellten Translator. Dafür kann er in seiner Abelian-Klasse beliebige Konstruktoren erstellen und diese im Translator verwenden. Durch die Schnittstelle Abelian müssen zudem folgende Methoden implementiert werden:

- `T invert()` - Ein Aufruf dieser Methode erzeugt das inverse Element $v*$ zum gegeben Objekt v. Dabei wird das Objekt selbst nicht modifiziert, lediglich eine invertierte Kopie ausgegeben.
- `Abelian<T> aggregate(final T other)` - Auch hier wird eine Kopie des Abelian-Objekts erzeugt. Das neue Objekt $v_1 \circ v_2$ ist die Aggregation der Objekte v_1 (`this`) und v_2 (`other`).
- `boolean isNeutral()` überprüft, ob ein Abelian-Objekt neutral ist.
- `Abelian<T> neutral()` erzeugt ein neutrales Abelian-Objekt e.
- `Writable extractKey()` gibt den Teil des Abelian-Objekts aus, welcher als Intermediate-Key benutzt wird. Dadurch wird festgelegt, wonach die Abelian-Objekte gruppiert werden. Elemente, die hier das Gleiche liefern, werden im Reduce-Schritt aggregiert.
- `void write(DataOutput out)` und `void readFields(DataInput in)` werden von der Oberschnittstelle `WritableComparable<BinaryComparable>` vererbt. Sie erzeugen und lesen einen Datenstrom, um das Abelian-Objekt von einem Rechner zu einem anderen zu übertragen. Der Benutzer kann dabei den Teil weglassen, der bei `extractKey` ausgegeben wurde, da dieser im Reducer schon als Schlüssel vorliegt. Der Wert ist das Abelian-Obejekt selbst.

Wird eine Abelian-Klasse innerhalb einer Kette von MapReduce-Jobs mehrfach verwendet, kann es vorkommen, dass im ersten Job ein anderer Schlüssel extrahiert werden soll als im zweiten Job. Aus diesem Grund hätte man die Methode `extractKey` in eine Klasse *KeyExtractor* auslagern können. Der Benutzer entwickelt also für seinen Abelian-Typ eine oder mehrere KeyExtractor-Klassen, ähnlich wie beim Serializer (siehe Abschnitt 5.3.2). Stattdessen wurde jedoch zugunsten des Bedienkomforts dafür entschieden, die `extractKey`-Methode in der Schnittstelle *Abelian* zu belassen und bei mehrfacher Verwendung einer Abelian-Klasse diese zu vererben und die Methode `extractKey` zu überschreiben.

5.3 Translator und Serializer

Die beiden Klassen *Translator* und *Serializer* spielen im Marimba-Framework eine ähnlich wichtige Rolle wie der *Mapper* und *Reducer* bei Hadoop. Dabei kann man den Translator mit dem Mapper vergleichen: Er verarbeitet die Eingabedaten und erzeugt Objekte, mit denen in den Folgeschritten weitergerechnet wird. Der Serializer entspricht dem letzten Schritt des Reducers, er erzeugt aus dem berechneten Endwert ein schreibbares Objekt, also z.B. eine Zeile, die an eine Textdatei angehängt wird.

5.3.1 Translator

Die abstrakte Klasse `Translator<KEYIN, VALUEIN>` nimmt Eingabedaten mit Schlüssel und Wert vom Typ `KEYIN` bzw. `VALUEIN` entgegen. Ähnlich wie bei der Hadoop-Klasse *Mapper* sieht die Signatur der Methode *translate* aus:

```
public abstract void translate(KEYIN key, VALUEIN value,
Context context) throws IOException, InterruptedException;
```

In dieser Methode steht das *Context*-Objekt zur Verfügung. Dies ist jedoch nicht der aus Hadoop bekannte *MapContext*, sondern eine modifizierte Variante, die alle Methodenaufrufe an ein MapContext-Objekt delegiert. Der einzige Unterschied besteht in der Methode *write*. Implementiert der Benutzer einen Translator, kann er ein Abelian-Objekt `v` mit dem Befehl `context.write(v)` weitergeben (vergleiche im Mapper: `context.write(key, value)`). Im Translator wird der Schlüssel weggelassen, da das Abelian-Objekt diesen beinhaltet. Die Delegation der *write*-Methode erfolgt wie folgt:

```
this.mapContext.write(abelianValue.extractKey(),
this.invertValue ? abelianValue.invert() : abelianValue);
```

Der Schlüssel wird also mit der *extractKey*-Methode ermittelt und der Wert vor der Weitergabe eventuell invertiert. Das Flag *invertValue* wird dem Translator vom Mapper mitgeteilt. Dieser entscheidet anhand des Typs (eingefügt oder gelöscht) der Eingabedaten, ob Werte normal oder invertiert geschrieben werden sollen.

5.3.2 Serializer

Die Hauptaufgabe des *Serializers* ist, ein abelsches Objekt in ein *Writable*-Objekt zu serialisieren, also z.B. in ein *Put* oder *Text*. Diese Methode wird vom Reducer aufgerufen, wenn die Aggregation abgeschlossen ist und das Resultat geschrieben werden soll. Ein Serializer wird wie folgt erstellt:

```
public class WordSerializer implements Serializer<WordAbelian>
```

Die Methode `public Writable serialize(Writable key, Abelian<?> obj)` der Schnittstelle `Serializer<T>` benötigt neben dem abelschen Wert zusätzlich noch den Schlüssel, da dieser, um unnötigen Datenverkehr zu vermeiden, nicht erneut mitübertragen werden sollte. Im Beispiel WordCount besteht das WordAbelian-Objekt dann nur noch aus dem Count-Wert und nicht mehr aus dem Wort selbst, da dieses als Schlüssel dient.

Eine zweite Methode des Serializers ist das Deserialisieren. Dies wird bei der Strategie Overwrite benötigt, bei der alte Ergebnisse gelesen und auf die neuen aggregiert werden. Folgende Methode muss dazu implementiert werden:

```
public T deserializeHBase(byte[] rowId, byte[] colFamily,
byte[] qualifier, byte[] value) throws NotSupportedException
```

Ein altes Ergebnis liegt in der HBase-Tabelle als Kombination aus Row-ID, Spaltenfamilie, Spalten-Qualifier und Spalten-Wert vor. Die `deserialize`-Methode erzeugt daraus ein abelsches Objekt. Wird Overwrite nicht unterstützt, kann eine NotSupportedException geworfen werden.

5.4 Der allgemeine Mapper, Reducer und Combiner

Die bis hierhin erläuterten Klassen und Methoden haben eine große Bedeutung für den Verwender des Marimba-Framework, da sie vererbt bzw. implementiert werden müssen. Die in diesem Abschnitt vorgestellten Klassen sind intern. Sie verwenden die vom Benutzer implementierten Translator-, Abelian- und Serializer-Klassen und übernehmen im Hadoop-Framework die Rolle des Mappers, Reducers und Combiners. Die folgenden Klassen bilden also eine Marimba-Implementierung auf eine Hadoop-Implementierung ab.

5.4.1 GenericMapper

Der *GenericMapper* ist der Mapper jedes Marimba-Jobs und funktioniert wie folgt:
Ist die Eingabe ein ...

- ... *OverwriteResult* (siehe Abschnitt 5.1.3), das aus Row-ID, Spalten-Familie, -Qualifier und -Wert besteht, wird dieses mittels des vom Benutzer konfigurierten Serializers deserialisiert. Das deserialisierte abelsche Objekt wird weitergeben.

- ... *PreservedValue*, also eine Eingabe, die bereits in die vormalige Berechnung eingegangen ist, kann sie ignoriert werden.

- ...*DeletedValue*, wird dies an den Translator weitergegeben und zusätzlich das Flag `invertValue` gesetzt, damit die Methode `Context.write` weiß, dass das erzeugte abelsche Objekt invertiert geschrieben werden muss.
- ...*InsertedValue*, wird es einfach nur an den Translator weitergegeben.

Der Aufruf der Methode `Translator.translate` ist die letzte Aktion des GenericMappers. In dieser Methode werden mit `context.write` abelsche Objekte geschrieben. Darum kümmert sich die Klasse *Context* im *Translator* (siehe Abschnitt 5.3.1).

5.4.2 GenericReducer

Der *GenericReducer* übernimmt im Hadoop-Framework die Rolle des Reducers für alle Marimba-Jobs. Er erhält als Eingabe einen bestimmten Schlüssel, also das Ergebnis der Methode `extractKey`, sowie eine Liste mit Abelian-Objekten zu diesem Schlüssel. Der GenericReducer arbeitet wie folgt:

1. Die Abelian-Werte werden aggregiert.
2. Der Endwert wird serialisiert.
3. Das serialisierte Objekt wird geschrieben.

Zu 1: In einer Schleife werden alle vorliegenden abelschen Objekte auf ein zu Beginn neutrales Element, welches mit `value.neutral()` erzeugt werden kann, aggregiert.

Zu 2: Das Serialisieren funktioniert über den Aufruf der `serialize`-Methode des Serializers. Dieser Methode wird nicht nur das Aggregat, sondern auch der Schlüssel übergeben. Als Ergebnis liegt ein schreibbares Objekt vor, z.B. ein *Text* oder *Put*.

Zu 3: Abhängig von der gewählten Strategie passieren beim Schreiben des serialisierten Objekts unterschiedliche Dinge. Bei der *Vollständigen Neuberechnung* wird das serialisierte Objekt einfach weitergegeben. Beim *IncDec*-Ansatz, muss ein *Increment*-Objekt ausgegeben werden. Da der Benutzer im Serializer jedoch ein Put erzeugt hat, muss dies mit der Funktion `putToIncrement` in ein Increment umgewandelt werden. Das Schreiben des Increment-Objektes sorgt dann in HBase dafür, dass die betreffenden Spaltenwerte nicht überschrieben sondern inkrementiert werden.

Bei der Strategie *Overwrite* funktioniert das Schreiben im wesentlichen wie bei der vollständigen Neuberechnung. Einziger Sonderfall ist die Behandlung von neutralen Elementen. Abhängig von der in Abschnitt 5.1.3 vorgestellten NeutralOutputStrategy wird das Ausgabeobjekt, falls es neutral ist, entweder gar nicht geschrieben (*IGNORE*), trotzdem geschrieben (*PUT*) oder in ein Delete-Objekt umgewandelt, um die Zeile aus der HBase-Tabelle zu löschen (*DELETE*). Letzteres passiert in der Methode `putToDelete` auf ähnliche Weise wie bei der Umwandlung in ein Increment-Objekt.

5.4.3 GenericCombiner

In den „7 Tips for Improving MapReduce Performance"[14] lautet Tipp Nummer vier: „Write a Combiner". Der Combiner verringert die Datenmengen, die von einem Rechner zum anderen übertragen werden und verringert die Laufzeit der Reduce-Jobs. Auch in [27] wird empfohlen, einen Combiner zu verwenden, wenn es der Algorithmus erlaubt.

Bei einem Marimba-Job ist es immer möglich einen Combiner zu verwenden. Er beschleunigt die Ausführung des Jobs enorm (siehe Kapitel 7) und funktioniert nach einem ganz simplen Prinzip: Die Eingabe der *combine*-Funktion ist ein Schlüssel und eine Liste von Abelian-Werten. Der Combiner aggregiert nun diese Werte mit der Funktion `aggregate` und gibt den Schlüssel sowie das berechnete Aggregat aus. Siehe auch Algorithmus 5 in Kapitel 3.3.3.

5.5 TextWindowInputFormat

In Hadoop häufig verwendete Eingabeformate sind das TableInputFormat und das TextInputFormat. Ersteres liest eine HBase-Tabelle und gibt Zeile für Zeile an den Mapper weiter. Das TextInputFormat liest alle Textdateien, die sich in einem Ordner auf dem HDFS-Dateisystem befinden und unterteilt sie in mehrere Splits (Standard-Split-Größe 64 MB). Ein Split ist die Eingabemenge, um die sich ein Mapper kümmert.

Ein Eingeformat besteht aus einer `InputFormat<K, V>`-Klasse, wobei `K` für den Typ des Eingabeschlüssels und `V` für den Typ des Wertes steht, sowie einem `RecordReader`. Über den `RecordReader` kann mit der Methode `nextKeyValue()` über die einzelnen Eingabetupel iteriert werden.

Beim TextInputFormat lierfert diese Methode die einzelnen Zeilen einer Textdatei. Dabei ist der Schlüssel die Zeilennummer (innerhalb der Textdatei) vom Typ *LongWritable* und der Wert der Text der Zeile, Typ *Text*.

Im Marimba-Framework, genauer im GenericMapper (siehe Abschnitt 5.4.1), reicht jedoch ein Text-Objekt nicht als Eingabe aus. Es muss unterschieden werden, ob es ein eingefügter, gelöschter oder beibehaltener Text ist, also etwas was seit der letzten Berechnung dazugekommen oder entfernt wurde oder bereits in die letzte Berechnung eingeflossen ist. Dazu wurden die Schnittstellen *InsertedValue*, *DeletedValue* und *PreservedValue* erstellt. Diese Schnittstellen stellen keine Methoden bereit, sondern dienen nur zur Kennzeichnung von Klassen. Verwendet wird dies wie folgt:

```
public class InsertedText extends Text implements InsertedValue
```

Durch die Vererbung der Klasse `Text` besitzt ein `InsertedText` also die gleiche Funktionalität und durch die Implementierung der Schnittstelle `InsertedValue` wird gekennzeichnet, das diese Klasse eingefügte Texte behandelt. Auf gleiche Weise werden auch die Klassen `DeletedText` und `PreservedText` definiert.

Bei einem Eingabeformat, welches zwischen eingefügten, gelöschten und beibehaltenen Eingaben unterscheidet, stellt sich die Frage, nach welchen Kriterien er dies tut. Im Produktiveinsatz sind die folgenden zwei Szenarios sinnvoll: Erste Möglichkeit: Der Benutzer, der den MapReduce-Job ausführt, gibt zu Beginn an, welche Daten hinzugefügt, gelöscht und beibehalten werden, z.B. durch Angabe von drei HDFS-Verzeichnissen. Zweite Möglichkeit: Das System erkennt an den Eingabedaten, welche Daten seit der letzten Berechnung dazugekommen bzw. weggefallen sind. Dieses Verfahren wird *Change Data Capture* genannt und ist vor allem bei HBase gebräuchlich. Dabei kann entweder auf den Zeitstempel der Spaltenwerte geschaut werden, um zu sehen, welche Daten hinzugefügt oder gelöscht wurden. Alternativ kann auch die Log des Datenbanksystems betrachtet werden. Weitere Details dazu sind in [10] zu finden.

Um die in Kapitel 6 vorgestellten Marimba-Jobs auf Performanz zu testen, wurde das Eingabeformat *TextWindowInputFormat* entwickelt. Das Format basiert auf dem TextInputFormat und liest Textdateien, welche in HDFS gespeichert sind. Der Benutzer gibt vor der Ausführung des Marimba-Jobs an, welche Teile der Textdatei als gelöschte und welche Teile als eingefügte Daten behandelt werden sollen. Dies erfolgt über die Angabe zweier Fenster (siehe Abbildung 5.1).

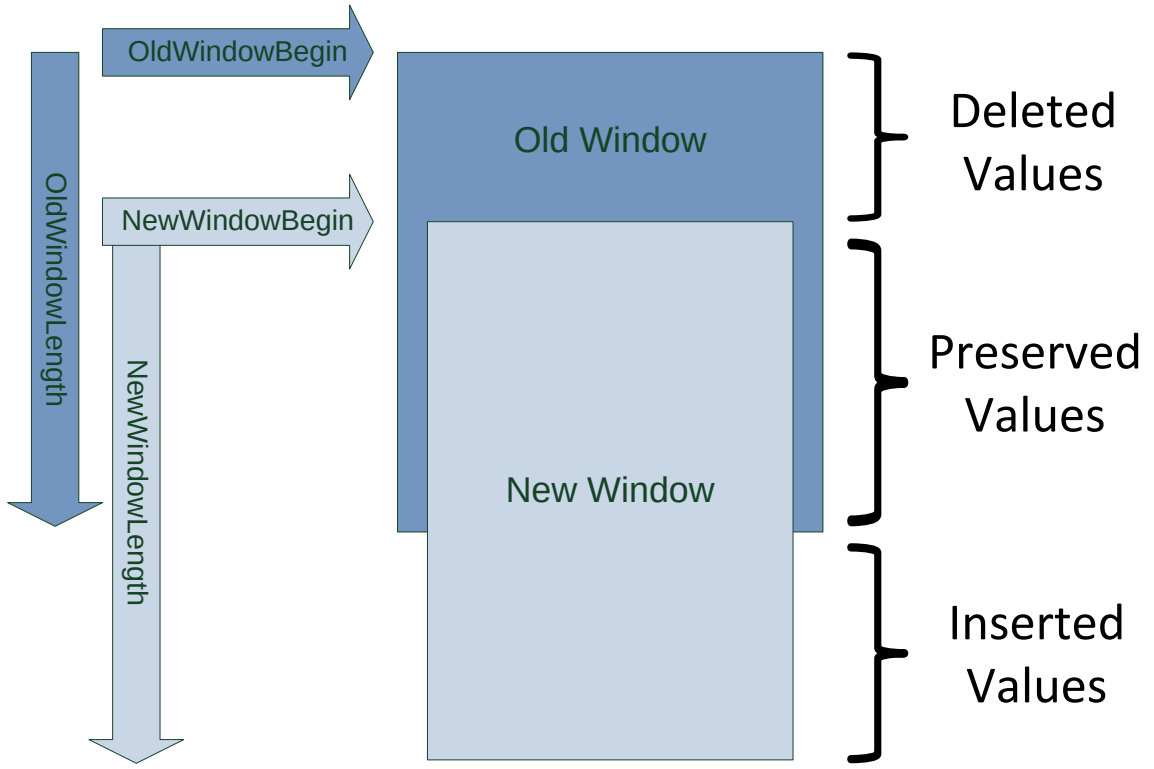


Abbildung 5.1: TextWindowInputFormat

Die Methode `nextKeyValue()` liest genau wie beim TextInputFormat eine Zeile. Anhand der Zeilennummer kann berechnet werden, in welchem der beiden Fenster sich diese Zeile befindet. Ist die Zeile im alten Fenster, aber nicht im neuen, wird ein *InsertedText*-Objekt ausgegeben, im umgekehrten Fall ein *DeletedText*-Objekt. Zeilen in der Schnittmenge sowie außerhalb beider Fenster sind irrelevant und werden übersprungen. Splits, die ausschließlich aus solchen uninteressanten Zeilen bestehen, werden in der Methode `getSplits` im `TextWindowInputFormat` direkt entfernt.

Ein Beispiel zur Verwendung des TextWindowInputFormats: Gegeben sei eine ein Gigabyte große Textdatei, welche in sechzehn Splits zu je 64 MB aufgeteilt wird. Der Benutzer startet einen Marimba-Job zum Zählen von Wörtern mit den folgenden Parametern (angegebene Größen in Megabyte):

```
-D oldwin.begin=100 -D oldwin.length=400
-D newwin.begin=300 -D newwin.length=500
```

Der erste Split (0-63 MB) sowie die letzten drei Splits (768-831 MB) liegen in keinem der beiden angegeben Fenster, müssen also nicht gelesen werden. Das Gleiche gilt für die beiden Splits, die vollständig in der Schnittmenge liegen (320-448 MB). Die übrig gebliebenen Splits werden gelesen und für die Zeilen zwischen 100 und 300 MB werden DeletedText-Objekte erzeugt sowie InsertedText-Objekte für die Zeilen zwischen 500 und 800 MB. Erstere liegen im alten, aber nicht im neuen Fenster, letztere genau umgekehrt. Die Änderungsrate beträgt in diesem Beispiel 60%:

$$\text{Änderungsrate} = \frac{\text{geändert}}{\text{neue Fenstergröße}} = \frac{\text{neue Fenstergröße} - \text{beibehalten}}{\text{neue Fenstergröße}} = \frac{500\,\text{MB} - 200\,\text{MB}}{500\,\text{MB}} = 60\%$$

Ein ähnliches Eingabeformat *ChangeLogInputFormat* wurde in [23] vorgestellt. Dieses liest die Datei jedoch immer von vorne. Der Benutzer gibt lediglich an, wie viel Prozent der Datei als geändert behandelt werden soll.

6 Beispielanwendungen

Um das Marimba-Framework auf Korrektheit und Performanz zu testen, wurden vier Beispielanwendungen mit dem Framework erstellt. Zusätzlich wurde jede dieser Anwendungen auch im reinen Hadoop ohne die Verwendung des Frameworks entwickelt, damit ein Vergleich möglich ist. Es empfiehlt sich, den Java-Code der Beispielanwendungen zu betrachten, um einen Einblick zu bekommen, wie eigene Marimba-Jobs entwickelt werden. Die wichtigsten Elemente, welche für jeden Job entwickelt werden müssen, sind ein *Translator*, ein *Serializer* und eine *Abelian*-Klasse.

6.1 WordCount

Das MapReduce-Paradebeispiel *WordCount* wurde in den vergangenen Kapiteln bereits mehrfach vorgestellt. Der klassische - nicht inkrementelle - Job teilt im Mapper den Eingabetext in seine einzelnen Wörter auf und liefert Schlüssel-Wert-Paare `(wort, 1)` aus. Der Reducer summiert die Einsen zu einem Wort auf und liefert somit die Gesamtanzahl des Wortes im gesamten Eingabetext.

Die inkrementelle Variante von WordCount (vergl. Kapitel 3.3) erzeugt Einsen für eingefügte Wörter, negative Einsen für gelöschte Wörter und addiert diese auf den bereits berechneten Wert. Dieser wird entweder als zusätzliche Eingabequelle (Overwrite) gelesen oder die Addition wird als Inkrement direkt in HBase ausgeführt (IncDec).

Um WordCount in Marimba zu implementieren (kompletter Java-Code im Anhang), muss ein Translator (Algorithmus 6), ein Serializer und eine Abelian-Klasse implementiert werden:

Algorithmus 6 WordCount: Translator

```
1: function TRANSLATE(String key, String value, Context context)
2:     for all word ∈ value do
3:         context.write(new WordAbelian(word, 1))
4:     end for
5: end function
```

In der Klasse *WordAbelian* arbeiten die Methoden wie folgt:

- Der Konstruktor `WordAbelian(String s, long l)` schreibt ein Wort und seine Anzahl in die Attribute `word` (Text) und `count` (LongWritable)
- `invert()` erzeugt ein neues WordAbelian-Objekt `new` mit der invertierten Anzahl: `new.count = this.count*-1`
- `aggregate(WordAbelian other)` erzeugt ein neues WordAbelian-Objekt `new` mit der Summe der Anzahlen: `new.count = this.count + other.count`

- `isNeutral()` ist wahr, wenn `this.count == 0`
- `neutral()` erzeugt ein WordAbelian-Objekt `new` mit der Anzahl `new.count = 0`
- `extractKey()` liefert das Wort (`word`)
- `write` / `readFields` schreibt bzw. liest nur den `count`-Wert.

Der *WordSerializer* wurde als abstrakte Klasse implemeniert, um zwei Ausgabeformate zu unterstützen. Seine Subklassen *WordHBaseSerializer* und *WordHDFSSerializer* erzeugen Put-Objekte (mit dem Wort als Row-ID, konstanten Werten für Spaltenfamilie und Spaltenname, sowie der Anzahl als Wert) bzw. LongWritable-Objekte (welche in eine Textdatei geschrieben werden können). Somit werden die Ausgabeformate *TableOutputFormat* und *TextOutputFormat* unterstützt. Das Eingabeformat für diesem Job ist das *TextWindowInputFormat* (siehe Kapitel 5.5). Die Methode `deserializeHBase` im WordSerializer wird für die Overwrite-Strategie benötigt und erzeugt aus einer HBase-Spalte, in welcher die Anzahl eines Wortes zu finden ist, ein neues WordAbelian-Objekt.

Zur Ausführung des Jobs sind nur die Parameter `input.dir` (der HDFS-Ordner mit den Eingabedaten) und `output.table` bzw. `output.dir` (Name der HBase-Tabelle / des HDFS-Ordners für die Ausgabedaten) anzugeben.

6.2 Friend-Of-Friends

Der Friend-Of-Friends-Algorithmus wird im Bereich Social-Network-Analysis verwendet, um zu einer Person die Freunde zweiten Grades zu ermitteln, also Personen, die man über eine weitere Person kennt. Die Anzahl dieser Personen verhält sich in etwa quadratisch zur Anzahl der eigenen Freunde. Im Xing-Profil des Autors stehen bei einer Anzahl von 70 direkten Kontakten knapp 12.000 Kontakte von Kontakten und fast eine Million Kontakte dritten Grades.

Vor allem Personen mit einer hohen Anzahl an gemeinsamen Freunden können in einem sozialen Netzwerk wie Facebook in einer Rubrik „Kennst du vielleicht...?“ vorgeschlagen werden.

Eine weitere Verwendung der Friends-Of-Friends ist bei einigen sozialen Netzwerken die Funktion, Beiträge nur direkten Freunden sowie Freunden zweiten Grades sichtbar zu machen. Dies bietet einen Kompromiss zwischen öffentlichen Beiträgen und nur mit Freunden geteilten Beiträgen. Kommentiert bei einem Beitrag, der nur für direkte Freunde sichtbar ist, ein solcher, erscheint auf dessen Profil die Nachricht „B hat As Beitrag kommentiert“, welche jedoch nur für Personen sichtbar ist, die die Personen A und B kennen. Soll dies für alle Freunde von B sichtbar sein, kann A seine Beiträge für die Freunde der Freunde sichtbar machen.

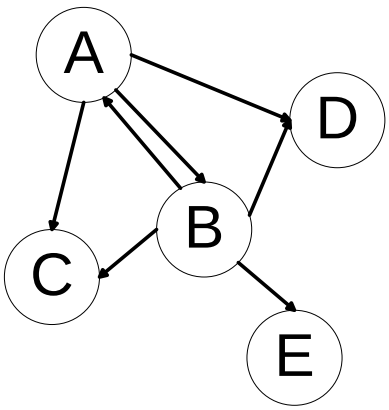
Zusätzlich kann die Liste der Friends-Of-Friends zur Sortierung beim Suchen nach Personen genutzt werden. Dabei erscheinen gefundene Personen, mit welchen man befreundet ist, ganz oben in der Ergebnisliste, direkt darunter diejenigen, die mit den eigenen Freunden befreundet ist (sortiert nach Anzahl der gemeinsamen Freunde) und danach erst die übrigen Ergebnisse.

Die n:m-Beziehung „ist Freund von“ wird in relationalen Datenbanken üblicherweise über eine Tabelle abgebildet, welche zwei Spalten hat. Die erste Spalte beinhaltet die ID

der Person, welche die Freundschaftsanfrage versendet hat, die zweite Spalte die ID des Freundes. Sollen Freundschaftsanfragen erst vom gegenüber bestätigt werden, ist dafür eine dritte boolsche Spalte notwendig. Eine Person mit fünfzig Freunden würde in dieser Tabelle in fünzig Zeilen vorkommen.

In Spaltenorientierten Datenbanken, speziell in HBase, ist es üblich, die „ist Freund von"-Beziehung etwas anders abzubilden. Als Row-ID wird die ID einer Person verwendet. Da HBase jedoch in der Anzahl und Benennung der Spalten flexibel ist, enthält eine Zeile so viele Spalten wie es Freunde gibt. Die HBase-Tabelle `friends` hat in der Spaltenfamilie `default` für jeden Freund eine Spalte mit dessen ID (siehe Abbildung 6.1b). Der Wert dieser Spalte ist egal, denn die Freunde sind in den Spaltennamen angegeben. Es kann also einfach ein Platzhalter-Wert wie `1` verwendet werden oder ein Datum, seit wann die Personen Freunde sind. Sollen Freundschaftsbeziehungen symmetrisch sein, ist darauf zu achten, dass eine Person immer auch Freund aller seiner Freunde ist. Im folgenden wird jedoch der Einfachheit halber auf die Symmetrie verzichtet. Bei dem weiter unten beschriebenen Algorithmus geht jedoch die Asymmetrie verloren, was das Ergebnis, also die Menge der Friends-Of-Friends, wenig aussagekräftig macht. Bei symmetrischen Eingabedaten würde der Algorithmus jedoch zum korrekten Ergebnis führen.

Die gerade definierte Tabelle `friends` dient als Eingabe des Friend-Of-Friend-Algorithmus. Die Ausgabe ist eine ähnlich aussehende Tabelle, welche zu einer Person, dessen ID auch hier als Row-ID verwendet wird, für jeden Freund zweiten Grades eine Spalte hat (siehe Abbildung 6.1c). Eine sinnvolle Verwendung des Spaltenwertes ist hier die Anzahl der gemeinsamen Freunde.



a)

FRIENDS

ROW-ID	A	B	C	D	E
A		1	1	1	
B	1		1	1	1

b)

FRIENDS OF FRIENDS

ROW-ID	A	B	C	D	E
A			1	1	1
B			1	1	
C	1	1		2	1
D	1	1	2		1
E	1		1	1	

c)

Abbildung 6.1: Friends Of Friends (asymmetrisch)

In [9] und [13] werden zwei Möglichkeiten vorgestellt, den Friends-Of-Friends-Algorithmus mit MapReduce zu implemenieren. Der in diesem Kapitel vorgestellte Algorithmus orientiert sich am ehesten an [9].

Der MapReduce-Job ohne Verwendung des Marimba-Frameworks (siehe Abbildung 6.2) berechnet für jede Zeile in der HBase-Tabelle, also für jede Person, das Kreuzprodukt der Freunde mit sich selbst. Hat die Person A also die Freunde B, C und D, ist die Ausgabe folgende:

$\{B, C, D\}^2 = \{(B, B), (B, C), (B, D), (C, B), (C, C), (C, D), (D, B), (D, C), (D, D)\}$
Entfernt man die reflexiven Beziehungen (B, B), (C, C) und (D, D), bleiben die Beziehungen übrig, wer wen über A kennt. Diese Beziehnungen werden im Mapper für jede Person erzeugt. Der Intermediate-Key und der Intermediate-Value sind also zwei unterschiedliche Freunde von der betreffenden Person. Der Reducer gruppiert alle zu einem Freund gehörenden indirekten Freunde zusammen und schreibt das Ergebnis in HBase.

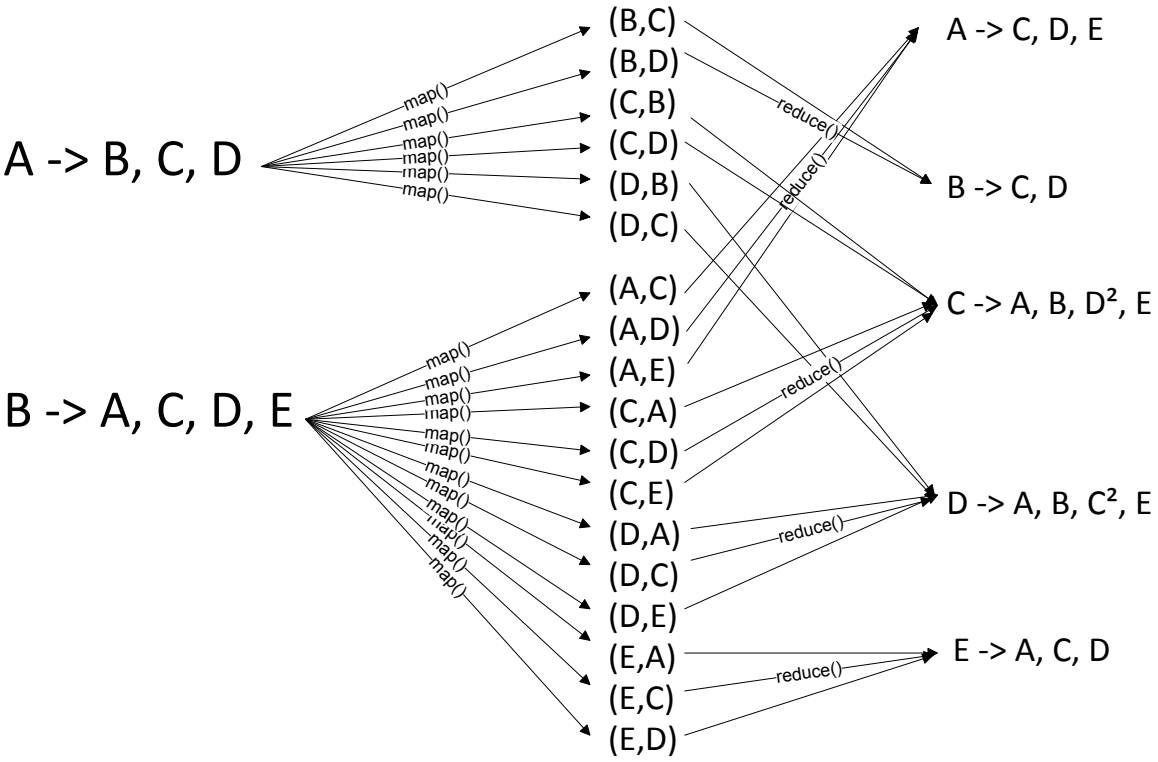


Abbildung 6.2: Friends Of Friends als MapReduce-Job[1]

Im Marimba-Framework wurden für den Friends-Of-Friends-Algorithmus die Klassen *FoFTranslator* (siehe Algorithmus 7), *FoFSerializer* und *FoFAbelian* entwickelt. Das Eingabeformat ist das *TableInputFormat*, um Freundesbeziehungen wie oben beschrieben aus einer HBase-Tabelle zu lesen. Das *TableOutputFormat* dient als Ausgabeformat, um eine ähnliche Struktur der Freundesfreunde zu generieren. Die Parameter für diesen Job sind lediglich die Namen der Eingabe- und Ausgabetabelle.

Die Methoden der Klasse *FriendAbelian*:

- Der Konstruktor `FriendAbelian(Text person)` initialisiert die Menge der Freunde zweiten Grades einer Person
- `addFriend(Text friend, long count)` fügt einen Freund zweiten Grades zu der Person hinzu, der `count`-Wert ist die Anzahl der gemeinsamen Freunde

[1] Die Schreibweise D^2 gibt an, dass die Person mit D zwei gemeinsame Freunde hat.

Algorithmus 7 Friends Of Friends: Translator

```
1: function TRANSLATE(String key, Result friends, Context context)
2:     for all friend1 ∈ friends do
3:         friendAbelian ← new FriendAbelian(friend1)
4:         for all friend2 != friend1 ∈ friends do
5:             friendAbelian.addFriend(friend2, 1)
6:         end for
7:         context.write(friendAbelian)
8:     end for
9: end function
```

- `invert()` erzeugt ein neues FriendAbelian-Objekt mit invertierten `count`-Werten bei jedem Element der Friends-Of-Friends-Menge
- `aggregate(FriendAbelian other)` verbindet die Mengen zweier FriendAbelian-Objekte. Bei Übereinstimmungen werden die `count`-Werte addiert
- `isNeutral()` ist wahr, wenn $\forall_{friend}$ `friend.count == 0`
- `neutral()` erzeugt ein FriendAbelian-Objekt mit einer leeren Menge von Freundesfreunden
- `extractKey()` liefert die ID der Person
- `write` / `readFields` schreibt bzw. liest die Menge der Freundesfreunde

Die Menge der Freunde zweiten Grades wurde realisiert über eine Map mit dem Schlüsseltyp Text (Freundes-ID) und dem Wertetyp LongWritable (`count`-Wert). Da diese Datenstruktur des Öfteren benötigt wird (u.a. beim Reverse Web-Link Graph, siehe Kapitel 6.3), wurde die Klasse *TextLongMapWritable* implementiert. Diese beinhaltet die Methoden `aggregate`, `isNeutral` sowie `write` und `readFields`, damit sie in den betreffenden Abelian-Klassen nicht jedesmal erneut entwickelt werden müssen.

Der Serializer *FoFSerializer* erzeugt aus einem FriendAbelian-Objekt ein Put-Objekt. Dabei wird die ID der Person als Row-ID verwendet und in einer Spaltenfamilie mit dem festen Wert „default“ für jeden Freundesfreund eine Spalte erstellt. Der Name der Spalte ist die ID des Freundesfreundes, der Wert ist die Anzahl der gemeinsamen Freunde (aufaddierte `count`-Werte). Um bei der Overwrite-Strategie alte Ergebnisse einlesen zu können, erzeugt die Methode `deserializeHBase` aus diesen neue FriendAbelian-Objekte.

6.3 Reverse Web-Link Graph

Der Algorithmus „Reverse Web-Link Graph”wird dazu verwendet, einen invertierten Web-Index zu erstellen, der zu einer Webseite alle URLs auflistet, die auf sie verweisen. Mit MapReduce arbeitet der Algorithmus [12] so, dass der Mapper den Quellcode einer Seite nach Links durchsucht und für jeden Link ein Ausgabetupel mit dem Ziel als Schlüssel ausgibt. Der Wert ist die URL des Dokuments, in dem der Link auftaucht, sodass die

Sammlung aller URLs von Seiten, die auf ein gemeinsames Zieldokument verlinken, im Reducer geschrieben werden können (siehe Algorithmus 8).

Algorithmus 8 Reverse Web-Link Graph

```
1:  function MAP(url, document)
2:      for all link in document do
3:          emit(link.target, url)
4:      end for
5:  end function
6:
7:  function REDUCE(target, list of sources)
8:      for all distinct s in sources do
9:          put(target, 'linkFamily', s, '')
10:     end for
11: end function
```

Eine Zeile in der Ausgabetabelle steht also für ein Dokument (siehe Abbildung 6.3b). Die Spalten haben als Namen die URLs der auf dieses Dokument verweisenden Seiten. Der Wert kann hier weggelassen werden. Bei der inkrementellen Variante ist jedoch der Wert von Bedeutung. Dieser gibt dort an, wie oft diese Verlinkung vorkommt. Würde man sich diese Zahl nicht merken, ist beim Löschen eines Links nicht klar, ob noch weitere Links auf das Dokument zeigen oder nicht.

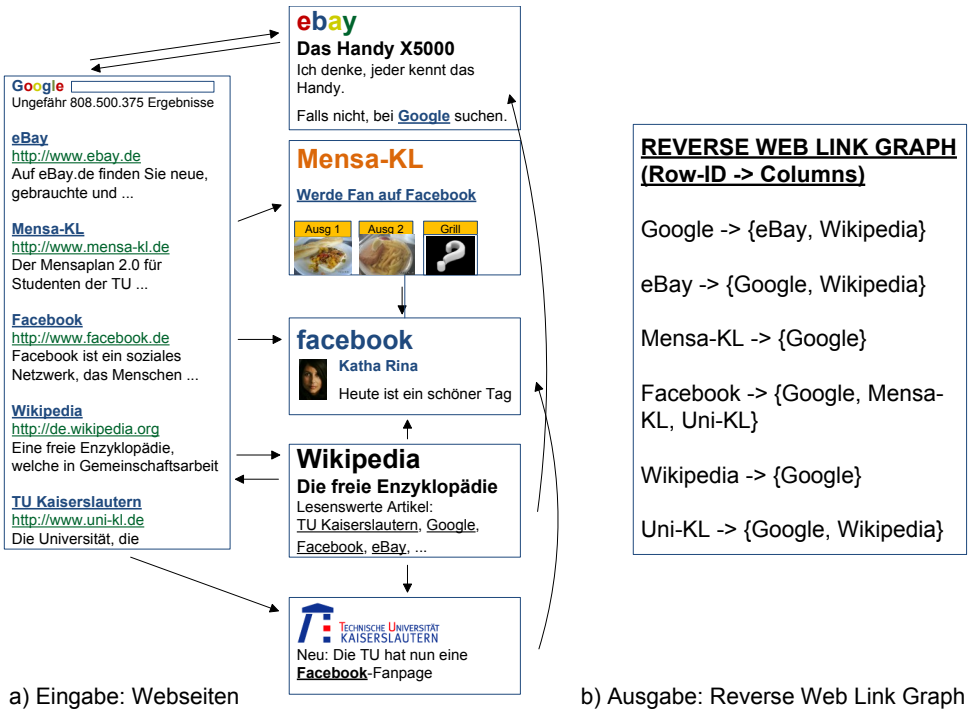


Abbildung 6.3: Reverse Web-Link Graph

Erstellt man den Reverse Web-Link Graph mit dem Marimba-Framework fällt eine enor-

me Ähnlichkeit zum Firends-Of-Friends-Algorithmus auf. Auch dort muss sich die Anzahl der Pfade zwischen zwei Personen (hier: Dokumenten) gemerkt werden. Erst wenn diese Anzahl null ist, kann die Verbindung gelöscht werden. Am *LinkTranslator* (siehe Algorithmus 9), *LinkSerializer* und *LinkAbelian* sind die Ähnlichkeiten zum Friend-Of-Friends-Algorithmus deutlich erkennbar:

Algorithmus 9 Reverse Web-Link Graph: Translator

```
1: function TRANSLATE(String url, String document, Context context)
2:     for all link in document do
3:         linkAbelian ← new LinkAbelian(link)
4:         linkAbelian.addLink(url, 1)
5:         context.write(linkAbelian)
6:     end for
7: end function
```

Bis auf den Fakt, dass kein Kreuzprodukt zwischen den Links erstellt wurde, funktioniert der Translator genau wie der *FoFTranslator*. Die Klasse *LinkAbelian* wurde wie folgt implementiert:

- Der Konstruktor `LinkAbelian(Text url)` initialisiert die Menge der eingehenden Links zur angegebenen URL
- `addLink(Text link, long count)` fügt einen Link zum Dokument hinzu, der `count`-Wert ist die Anzahl der Links vom Dokument `link` zum Dokument `this.url`
- `invert()` erzeugt ein neues LinkAbelian-Objekt mit invertierten `count`-Werten bei jedem Link
- `aggregate(LinkAbelian other)` verbindet die Mengen zweier LinkAbelian-Objekte. Bei Übereinstimmungen werden die `count`-Werte addiert
- `isNeutral()` ist wahr, wenn $\forall_{link}$ `link.count == 0`
- `neutral` erzeugt ein LinkAbelian-Objekt mit einer leeren Menge von Links
- `extractKey` liefert die URL
- `write` / `readFields` schreibt bzw. liest die Menge der Links

Als interne Datenstruktur für die Menge der Links wurde wieder die *TextLongMapWritable* (siehe Kapitel 6.2) verwendet. Diese erspart die Implementierung der Methoden `aggregate`, `isNeutral`, `write` / und `readFields`. Auch der *LinkSerializer* arbeitet ähnlich zum *FoFSerializer*. Er erzeugt ein Put-Objekt mit der Ziel-URL als Row-ID und für jeden eingehenden Link eine Spalte, die die Quell-URL als Namen und die Anzahl der Verlinkungen als Wert hat. Das Augabeformat ist also auch hier das *TableOutputFormat*. Da die Eingabe in Form von Textdateien in HDFS vorliegt, arbeitet der Job mit dem *TextInputFormat*.

6.4 Bigrams

Die vierte Beispielanwendung ist ein Schwerpunkt dieser Masterarbeit. Ein Vertreter eines japanischen Telekommunikationsunternehmens berichtete über diesen Anwendungsfall. Das Unternehmen nutzt MapReduce, um aus Texten Wahrscheinlichkeiten dafür abzuleiten, dass nach einem Wort ein bestimmtes anderes Wort folgt. Dies wird verwendet, um die Qualität von Spracheingaben zu verbessern. Wurde ein Wort in einem gesprochenen Satz nicht verstanden, kann abhängig von den vorherigen Wörtern das Wort ermittelt werden, welches am wahrscheinlichsten an der Stelle steht. Beginnt beispielsweise eine gesprochene SMS mit „Alles Gute zum“ und das letzte Wort muss aufgrund eines Störgeräuschs geraten werden, schlägt die Software das Wort „Geburtstag“ vor. Nach einem ähnlichen Prinzip arbeitet die alternative Smartphone-Tastatur-Software Swiftkey [15], die bereits beim Tippen das nächste Wort vorschlägt.

Zwei Fragen sind noch offen: Wie funktioniert der Algorithmus und woher bekommt man sinnvolle Testdaten. Die letzte Frage könnte man spontan mit „Wikipedia“ beantworten. Man könnte die komplette deutsche Wikipedia, welche frei verfügbar und herunterladbar ist, analysieren und so die Worthäufigkeiten ermitteln. Allerdings wird bereits bei Betrachten des oben beschriebenen Beispielsatzes klar, dass Wikipedia-Artikel anders geschrieben sind als typische SMS-Texte. Deshalb verwendet das japanische Telekommunikationsunternehmen als Eingabedaten Twitter-Feeds („Tweets“). Solche Tweets sind der gesprochenen Sprache sehr ähnlich und addressieren häufig auch andere Personen [5]. Sie werden sogar dazu verwendet, um herauszufinden, wo es gerade schneit, abhängig davon, wo Leute wohnen, die „Schnee“ oder „schneit“ twittern. In [4] werden aus Twitter-Feeds Aussagen zum Gesundheitszustand der Bevölkerung abgeleitet.

Nicht-inkrementeller MapReduce-Job

Der Bigrams-Algorithmus, um Twitter-Feeds zu analysieren, ist in zwei Schritte unterteilt, die als einzelne MapReduce-Jobs implementiert werden:

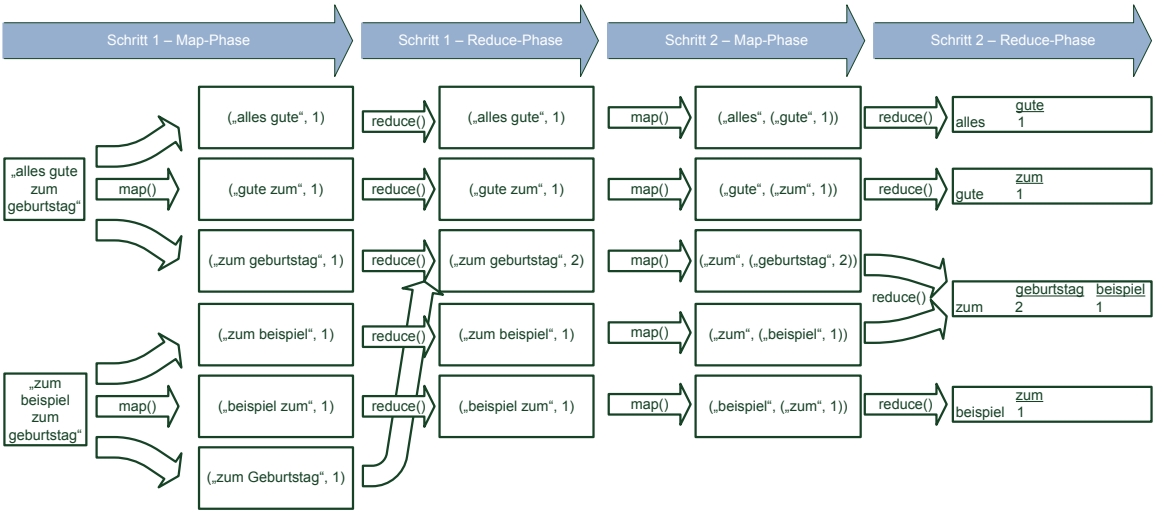


Abbildung 6.4: Bigrams

Der erste Schritt ähnelt dem WordCount-Algorithmus (siehe Kapitel 6.1). Allerdings werden hier nicht einzelne Wörter gezählt, sondern Wortpaare:

Algorithmus 10 Bigrams: Schritt 1

```
1:  function MAP(linenumber, line)
2:      for all word_i ∈ line do
3:          if word_i is not the last word in the line then
4:              emit(word_i+" "+word_{i+1}, 1)
5:          end if
6:      end for
7:  end function
8:
9:  function REDUCE(wordpair, counts)
10:     sum ← 0
11:     for all n ∈ counts do
12:         sum ← sum + n
13:     end for
14:     emit(wordpair, sum)
15: end function
```

Das Ergebnis sind die Häufigkeiten der Vorkommnisse bestimmter Wordpaare. Es kann entweder in HBase oder HDFS geschrieben werden. Da sowohl das Schreiben dieser Zwischenergebnisse als auch das Lesen im zweiten Schritt sequentiell erfolgt, ist HDFS hier die sinnvollere Wahl. Im zweiten Schritt werden die Wortpaare wieder aufgeteilt und nach dem ersten Wort gruppiert:

Algorithmus 11 Bigrams: Schritt 2

```
1:  function MAP(wordpair, count)
2:      split wordpair into (word1, word2)
3:      emit(word1, (word2, count))
4:  end function
5:
6:  function REDUCE(word1, pairs)
7:      sum ← 0
8:      for all (word2, count) ∈ pairs do
9:          put(word1, 'columnFamily', word2, count)
10:     end for
11: end function
```

Die Reduce-Funktion liefert HBase-Zeilen, die als Row-ID ein bestimmtes Wort haben. Die Spalten dieser Zeile tragen als Namen ein Wort, welches auf das erste Wort folgt und als Wert die Anzahl der Vorkommnisse. Beispiel: Die Row-ID lautet „zum“, die beiden Spalten „Geburtstag“ und „Beispiel“ haben die Werte „2“ und „1“. Am wahrscheinlichsten ist also, dass auf das Wort „zum“ das Wort „Geburtstag“ folgt.

Inkrementeller MapReduce-Job: IncDec

Die inkrementelle Variante des Bigrams-Algorithmus ist für die Strategie IncDec sehr ähnlich zu der gerade vorgestellten nicht-inkrementellen Variante. Der Mapper des ersten Schrittes muss lediglich unterscheiden, ob ein Text hinzugefügt oder entfernt wurde und dann das Wortpaar plus die 1 bzw. (-1) weitergeben. Der Reducer bleibt unverändert. Die Zwischenergebnisse sind nun also die positive oder negative Differenz, wie oft ein Wortpaar dazugekommen oder weggefallen ist. Auch der Mapper des zweiten Schrittes bleibt unverändert. Nur der Reducer muss modifiert werden, damit er statt Put-Objekten Inkremente ausgibt.

Inkrementeller MapReduce-Job: Overwrite

Komplizierter ist der Bigrams-Algorithmus mit Verwendung der Overwrite-Strategie. Man muss sich überlegen in welchem der beiden Schritte die Overwrite-Strategie verwendet wird. Würde man den ersten Schritt auf gleiche Weise wie beim IncDec durchführen und die Overwrite-Strategie erst auf den zweiten Schritt anwenden, tritt das folgende Problem auf: Die Map-Funktion erzeugt $(word1, (word2, count))$-Paare für die Zwischenergebnisse aus dem ersten Schritt und aus Ergebnissen der letzten Berechnung. Der Reducer erhält zu einem $wort1$ eine Liste mit $(wort2, count)$-Paaren. Folgewörter, die sowohl in den alten Ergebnissen vorhanden waren als auch hinzugekommen oder gelöscht wurden, erscheinen in dieser Liste doppelt. Für den Reducer ist es nicht möglich, dies zu erkennen und die Werte aufzuaddieren. Als Lösung könnte man alle bereits verarbeiteten Wörter in einer Map merken, aber es ist anzumerken, das dies enorme Datenmengen sind. Ein Beispiel: Der Mapper liest ein Zwischenergebnisse aus dem ersten Schritt $(\text{"der Hund"}, 5)$ und erzeugt daraus $(\text{"der"}, (\text{Hund"}, 5))$. Außerdem liest er aus der Tabelle mit den alten Ergebnissen die Zeile mit der Row-ID „der“. Für jede Spalte, also auch die Spalte „Hund“ wird ein Schlüssel-Wert-Paar erzeugt: $(\text{"der"}, (\text{Hund"}, 100))$. Der Reducer, welcher sich um den Schlüssel „der“ kümmert, erhält eine Liste mit Millionen Wörtern, u.a. zwei mal das Wort „Hund“. Er müsste sich alle bereits verarbeiteten Wörter merken, um zu erkennen, dass das Wort bereits einmal vorkam. Aus diesem Grund, wurde entschieden, die Overwrite-Strategie bei der Bigrams-Anwendung im ersten Schritt durchzuführen. Der Mapper liest neue und gelöschte Texte sowie alte Ergebnisse und erzeugt die $(word1+\texttt{" "}+word2, count)$-Paare. Im Reducer werden die Anzahlen aufaddiert. Der zweite Schritt kann nun unverändert durchgeführt werden. Einziger Nachteil an dieser Lösung ist, dass das Zwischenergebnis sehr groß ist, da es alle Daten beinhaltet, sowohl die Änderungen als auch die alten Ergebnisse.

Marimba-Job

Da die Bigrams-Anwendung aus zwei Schritten besteht, müssen zwei Marimab-Jobs geschrieben werden. Die Abelian-Klasse ist dabei in beiden Fällen das *NGramAbelian*, die wie folgt implementiert wurde:

- Konstruktor `NGramAbelian(Text a, Text b, long count)` schreibt zwei Wörter und die Anzahl in die Attribute `a`, `b` und `count`

- `invert()` erzeugt ein neues NGramAbelian-Objekt `new` mit der invertierten Anzahl: `new.count = this.count*-1`
- `aggregate(NGramAbelian other)` erzeugt ein neues NGramAbelian-Objekt `new` mit der Summe der Anzahlen: `new.count = this.count + other.count`
- `isNeutral()` ist wahr, wenn `this.count == 0`
- `neutral` erzeugt ein NGramAbelian-Objekt `new` mit der Anzahl `new.count = 0`
- `extractKey` liefert beide Wörter (`a+" "+b`)
- `write` / `readFields` schreibt bzw. liest nur den `count`-Wert.

Die Translator-Klasse für den ersten Schritt arbeitet wie der Mapper beim nicht-inkrementellen Bigrams-Job (siehe Algorithmus 10). Er liest eine Textzeile und erzeugt NGramAbelian-Objekte mit einem Wortpaar und der Anzahl 1 (bzw. (-1) bei gelöschten Zeilen). Der *BigramsHDFSSerializer* schreibt lediglich den Count-Wert. Zusammen mit dem Schlüssel, also dem Wortpaar, ergibt sich dann die Ausgabe, die in die HDFS-Textdatei geschrieben wird: `wordpair<TAB>count`.

Um die Klasse *NGramAbelian* auch für den zweiten Schritt verwenden zu können, wurde eine Subklasse von dieser namens *NGramStep2Abelian* erstellt, welche die Methoden `extractKey`, `write` und `readFields` überschreibt, da im zweiten Schritt der Intermediate-Key nur aus dem ersten Wort besteht. Der Translator für den zweiten Schritt erzeugt aus einer Zeile lediglich ein NGramStep2Abelian-Objekt, der *BigramsStep2Serializer* wandelt ein solches Objekt in ein Put-Objekt, um es in eine HBase-Tabelle zu schreiben. Dabei ist das erste Wort des Wortpaares die Row-ID. Das zweite Wort ist der Name der Spalte, die als Wert die Anzahl hat.

Die Klasse *NGramAbelian* benutzt anders als *FriendAbelian* und *LinkAbelian* keine Map, um die Menge der Folgewörter zu speichern. Das resultiert darin, dass die Map-Funktion statt einem Objekt sehr viele ausgibt und die Aggregation der Objekte lediglich eine Addition zweier `count`-Werte ist. Auch die Reduce-Funktion arbeitet mit einer viel größeren Zahl von Objekten und gibt für jedes ein eigenes Put-Obejekt mit nur einer Spalte aus, während bei *FriendAbelian* und *LinkAbelian* die Put-Objekte tausende Spalten besitzen können. Das Resultat sieht in den Tabellen gleich aus. Man kann für jeden Marimba-Job selbst entscheiden, ob man als interne Datenstruktur eine Map benutzen möchte oder lieber viele einzelne Abelian-Objekte ausgibt.

Bemerkungen

Die Anwendung wurde „Bigrams“ genannt, da mit Wortpaaren (Bigrammen) gearbeitet wurde. In der Ergebnistabelle kann also nachgeschlagen werden, welches Wort am wahrscheinlichsten auf ein gegebenes Wort folgt. Um die Qualität zu verbessern, ist es sinnvoll, nicht Bigramme, sondern Trigramme oder allgemein N-Gramme beliebiger Größe zu betrachten. Beinhaltet eine gesprochene SMS den abgebrochenen Satz „Viele liebe“, liefert die Bigrams-Ergebnistabelle den Vorschlag „dich“, da dies auf das Wort „liebe“ am wahrscheinlichsten folgt. Betrachtet der Algorithmus jedoch mehrere Wörter in Folge, würde

dies zu einem deutlich sinnvollerem Ergebnis („Grüße“) führen. Aus diesem Grund wurde der Bigrams-Algrorithmus über einen Parameter *numberOfParts* erweitert. In diesem kann angegeben werden, wie viele Wörter maximal in ein NGram gepackt werden. Die Eingabe „alles gute zum geburtstag“ liefert dann also bei $numberOfParts = 4$ nicht nur Bigramme wie „zum geburtstag“, sondern auch Trigramme wie „alles gute zum“ und ein 4-Gramm (der komplette Satz). Um die Qualität noch mehr zu verbessern, könnte man zusätzlich zu den vorherigen auch die folgenden Wörter betrachten. Dann würde „ich habe <?> Geld“ zu einem anderen Ergebnis führen als „ich habe <?> Lust“: Im ersten Fall „kein“, im zweiten Fall „keine“.

Bei der Implementierung des Bigrams-Algorithmus wurden Zahlen in den Eingabedaten durch einen Platzhalter <num> ersetzt, da es in den meisten Fällen keinen Unterschied macht, um welche Zahl es sich genau handelt. Der Satz „kannst du mich in etwa 10 <?> abholen“ wird dann zuerst umgewandelt in „kannst du mich in etwa <num> <?> abholen“. Die Ergebnistabelle schlägt bei „in etwa <num>“ das Wort „Minuten“ vor. Dieses 4-Gramm kam 24 mal in den eingelesenen Twitter-Feeds vor. Selbst „etwa <num>“ führt zum richtigen Ergebnis („Minuten“). Das Trigramm kam 382 mal vor. Nur nach <num> zu suchen macht jedoch keinen Sinn, da nach Zahlen oft weitere Zahlen folgen, hier 109.825 mal. Es ist also geschickt, möglichst viele vorangehende Wörter zu betrachten. Erst wenn eine Wortfolge gar nicht gefunden wurde, kann ein Wort weggelassen werden und mit der nächst kürzeren Folge fortgefahren werden. Um die geraden vorgestellten Ergebnisse zu ermitteln, wurde das Programm *HBaseMax* erstellt. Es nimmt als Konsolenparameter der Name einer Tabelle, eine Row-ID (eine oder mehrere Wörter) und der Name der Spaltenfamilie. Als Ergebnis liefert sie den Spaltennamen und Spaltenwert der Spalte mit dem größten Wert:

```
> java -jar HBaseMax.jar bigrams "viel" default
Qualifier: spaß
Value: 17894
```

7 Evaluation

Die im vorherigen Kapitel vorgestellten Beispielanwendungen wurden auf einem Cluster, welcher aus sechs Knoten besteht, ausgeführt. Jeder der Knoten besitzt einen Quadcore-Prozessor mit 2,53 GHz, 4 GB RAM, eine 1 TB SATA-II-Festplatte sowie Gigabit-Ethernet. Die Jobs werden mit Hadoop ausgeführt und lesen und schreiben in HDFS bzw. HBase.

7.1 Nicht-inkrementelle Jobs

Für jede der Beispielanwendungen wurde sowohl eine nicht-inkrementelle, eine inkrementelle und eine Variante im Marimba-Framework erstellt. Zunächst soll überprüft werden, ob die nicht-inkrementelle Variante, welche direkt in Hadoop geschrieben wurde, gleich schnell ist zu dem im Marimba-Framework implementierten Job, welcher die Strategie „Full Recomputation“ verwendet. In beiden Fällen werden also keine inkrementellen Berechnungen durchgeführt.

Für diese Tests wurden die Laufzeiten der Jobs „WordCount“ und „Friends Of Friends”(FoF) sowohl mit als auch ohne dem Marimba-Framework gemessen. Die Eingabedaten vom WordCount-Job sind eine 1,4 GB große Textdatei mit Twitter-Feeds, die Eingabe für den Friend-Of-Friend-Algorithmus ein zufällig generiertes Netzwerk aus 5000 Personen. Dazu wurde ein Programm *SocialNetGenerator* geschrieben, welches zufällige Freundesbeziehungen in eine HBase-Tabelle einfügt.

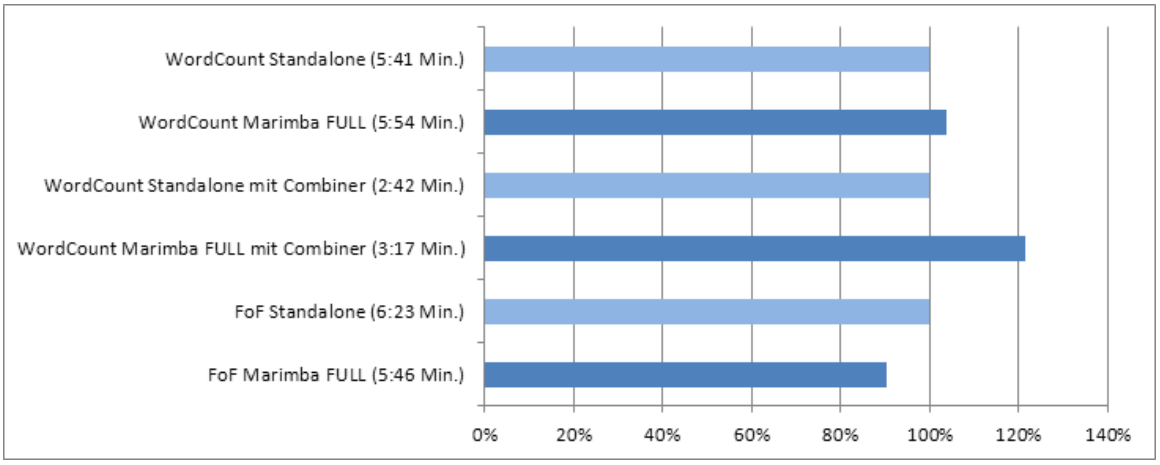


Abbildung 7.1: Vergleich: Vollständige Neuberechnung, Standalone vs. Marimba

Abbildung 7.1 zeigt, dass ein Marimba-Job in den meisten Fällen etwa fünf bis zwanzig Prozent langsamer ist, als ein nativ in Hadoop geschriebener Job. Die Marimba-Variante des Friend-Of-Friend-Algorithmus' läuft sogar etwas schneller als die native Version. Wei-

tere Tests zeigten, dass die Unterschiede minimal sind und vor allem davon abhängen, ob ein Mapper oder Reducer wegen eines Fehlers neustarten muss. Die Daten, welche in dem Standalone und im Marimba-Job transportiert werden, sind im wesentlichen die gleichen, auch die Berechnungen sind ähnlich. Lediglich die Verwendung eines Abelian-Typs statt der direkten Verwendung von primitiven Typen wie Text oder LongWritable können zu einem Mehraufwand führen. Da die Unterschiede jedoch sehr gering sind, wird im folgenden davon ausgegangen, dass Marimba-Jobs, welche die Strategie *FULL* (vollständige Neuberechnung) verwenden, eine ähnliche Laufzeit haben, wie nativ erstellte Hadoop-Jobs, welche ebenfalls keine inkrementelle Neuberechnung unterstützen.

7.2 Inkrementelle Marimba-Jobs

Die Marimba-Jobs WordCount, Reverse Web-Link Graph und „Bigrams“ wurden auf dem Cluster mit einer großen Eingabedatenmenge ausgeführt. Diese sind bei WordCount und dem Reverse Web-Link Graph eine Sammlung von mit [8] generierten HTML-Dokumenten mit einer Gesamtgröße von 30 GB. Der Bigrams-Algorithmus wurde auf 900 MB deutschen Twitter-Feeds ausgeführt. Dabei erfolgte zuerst eine vollständige Neuberechnung und danach eine inkrementelle Ausführung unter Verwendung des *TextWindowInputFormat* (siehe Kapitel 5.5). Dieses nutzt ein gleitendes Fenster, welches angibt, welche Daten als gelöscht, beibehalten und hinzugefügt angesehen werden. Abhängig von der Änderungsrate wurden die Zeiten gemessen, die die inkrementellen Marimba-Jobs unter Verwendung der Strategie IncDec bzw. Overwrite gebraucht haben:

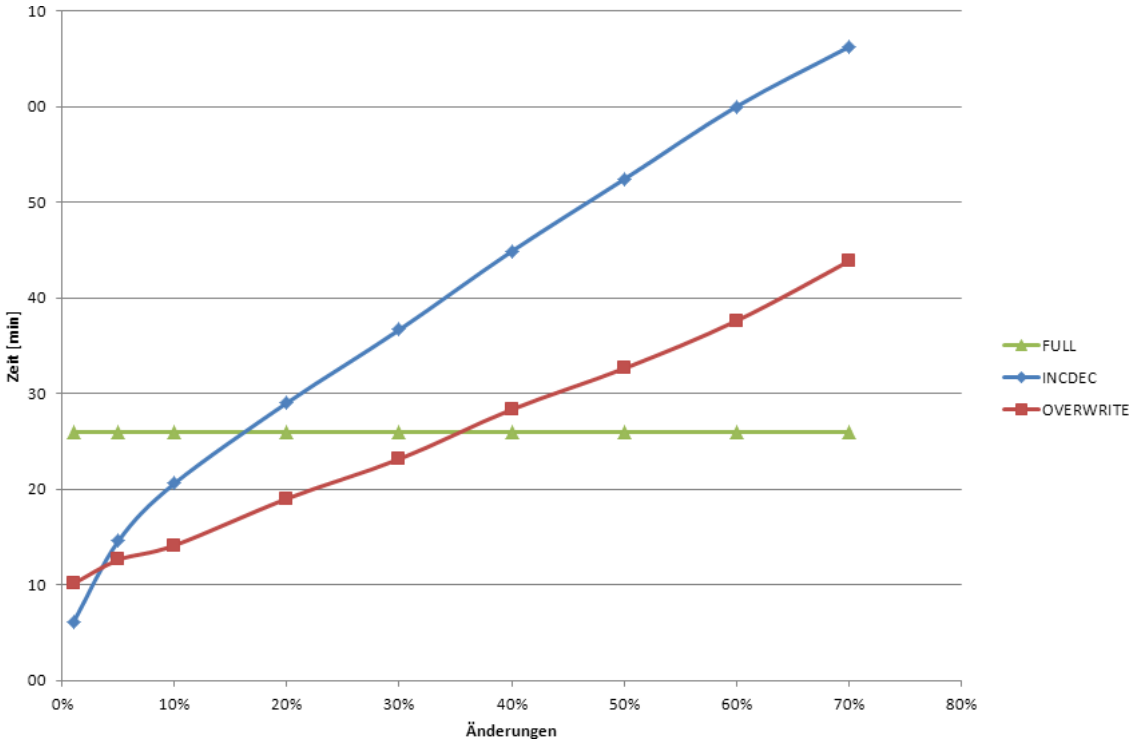


Abbildung 7.2: WordCount - Laufzeitvergleich

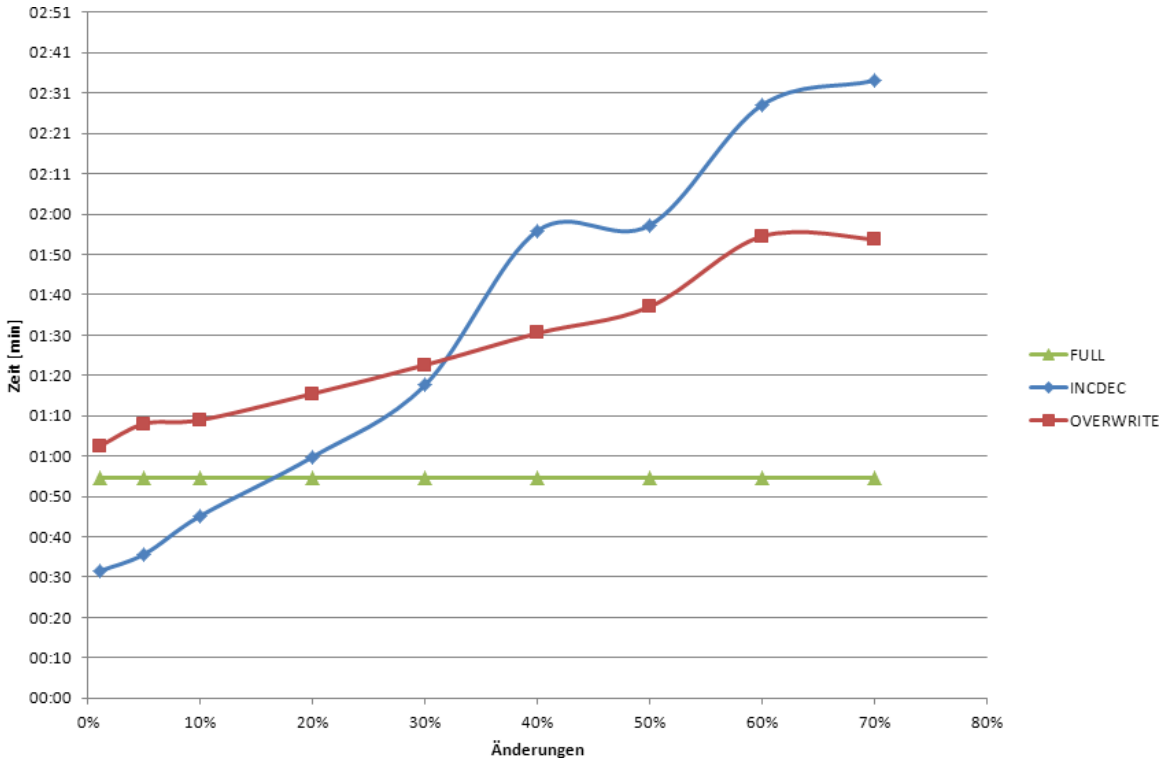


Abbildung 7.3: Reverse Web-Link Graph - Laufzeitvergleich

Die Abbildungen 7.2 und 7.3 zeigen, dass bei einer geringen Änderungsrate der IncDec-Ansatz eine bessere Wahl ist als die Overwrite-Strategie, da die alten Ergebnisse nicht erst komplett eingelesen werden müssen. Da die Inkrement-Operation in HBase jedoch sehr teuer ist, lohnt sich die IncDec-Strategie ab einer gewissen Anzahl von Änderungen nicht mehr, ein Einlesen der alten Ergebnisse macht sich dann also bezahlt. Bei großen Änderungsraten (WordCount: 40%, Reverse Web-Link Graph: 20%) lohnen sich beide Ansätze nicht mehr. Hier ist eine vollständige Neuberechnung günstiger.

Der Bigrams-Algorithmus ist aufgeteilt in zwei Schritte, welche jeweils einen eigenen MapReduce-Job bilden. In der folgenden Abbildung 7.4 sind für die Strategien FULL, INCDEC und OVERWRITE drei Bereiche eingezeichnet. Die obere Kante eines solchen Bereichs gibt die Gesamtlaufzeit beider Schritte an, die Unterkante die Laufzeit des zweiten Schritts und die Bereichsbreite die Laufzeit des ersten Schritts. Es fällt auf, dass bei der IncDec-Strategie die Laufzeit des ersten Schrittes sehr gering ist und der Job lediglich lange benötigt um im zweiten Schritt die Inkremente in HBase zu schreiben. Ab einer Änderungsrate von 10% ist eine vollständige Neuberechnung günstiger. Das vollständige Einlesen der alten Ergebnisse dauert bei der Overwrite-Strategie so lange, dass sich diese Strategie bei der Bigrams-Anwendung in keinem Fall lohnt.

Bei der Ausführung des Bigrams-Algorithmus fiel auf, dass die Ausführungszeit der einzelnen Reduce-Aufgaben sehr unterschiedlich ist. Das liegt daran, dass Reducer zwar in etwa die gleiche Anzahl von Schlüsseln verarbeiten, jedoch die Menge der Werte zu einem Schlüssel unterschiedlich groß sein kann. So muss der Reducer, welcher sich um das Wort „der“ kümmert eine riesige Menge von Werten aggregieren und daraus viele

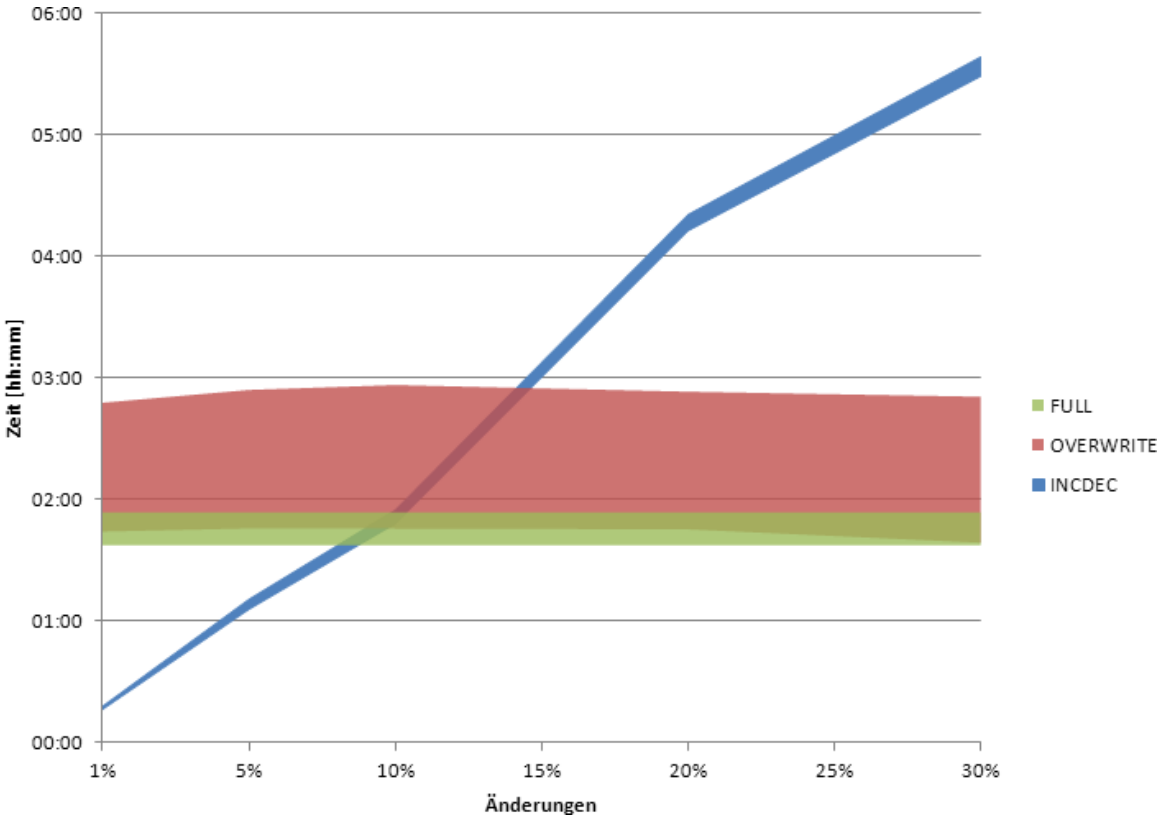


Abbildung 7.4: Bigrams - Laufzeitvergleich

Put-Objekte erzeugen und schreiben.

Zusammengefasst konnte mit den durchgeführten Tests gezeigt werden, dass inkrementelle Marimba-Jobs bei geringen Änderungsraten deutlich schneller laufen als Jobs, welche eine vollständige Neuberechnung durchführen. Der Vorteil an einem Marimba-Job ist zudem, dass er leicht zu entwickeln ist und bei der Entwicklung nicht darauf geachtet werden muss, ob als Strategie IncDec oder Overwrite verwendet wird. Die Tests zeigten außerdem, dass es keinen klaren Gewinner unter diesen beiden Strategien gibt. Durch einfache Konfiguration kann nach der Entwicklung eines Jobs die Strategie festgelegt werden und so ausprobiert werden, welche die sinnvollste für die eigene Anwendung ist.

Marimba-Jobs werden als normale Hadoop-Jobs ausgeführt. Das heißt, dass sie auf gleiche Weise über die Konsole gestartet und über das Webinterface von Hadoop überwacht werden können. Zudem kann die Konfiguration des Jobs nach belieben gesetzt werden. Eine entscheidende Konfiguration ist die Anzahl der Reducer. In Kapitel 2.1.5 wurde bereits erklärt, dass man die Zahl auf (0,95 oder 1,75 * Anzahl der Knoten * Aufgaben pro Knoten) setzen soll [19].Tests im Cluster mit sechs Knoten (davon ein Head-Node und fünf Worker) ergaben, dass 18 Reducer die optimale Anzahl ist (1,75 * 5 Knoten * 2 gleichzeige Aufgaben pro Knoten = 17,5). So konnten zehn Reduce-Aufgaben gleichzeitig auf den fünf Rechnern abgearbeitet werden. Sind die ersten Aufgaben abgeschlossen, kümmern sich die Rechner um die restlichen acht.

Weitere Empfehlungen über die Konfiguration von Hadoop-Jobs werden in [24] gege-

ben. Sie führten in den getesteten Marimba-Jobs zu einer deutlichen Performanzsteigerung. Beispielsweise bewirkt der Parameter `mapred.compress.map.output` eine Kompression der Intermediate-Keys und -Values, also der Abelian-Objekte. Vor allem bei den Beispielen FriendAbelian und LinkAbelian, bei denen eine TextLongMapWritable verwendet wurde, sorgt eine Kompression für eine enorme Verringerung des Datenvolumens. Die Parameter `mapred.tasttracker.map/reduce.tasks.maximum` wurden auf dem Standardwert 2 belassen. Es wird empfohlen, diesen Wert auf mindestens (Anzahl der Kerne / 2) zu setzen (was in unserem Fall $4/2 = 2$ ist) und maximal auf (Anzahl der Kerne * 2). Allerdings empfiehlt sich diese Konfiguration erst bei einem sehr großen Cluster.

8 Zusammenfassung

Die Ergebnisse im vorherigen Kapitel bestätigten die Beobachtungen aus den vorangegangenen Arbeiten. Es konnte gezeigt werden, dass inkrementelle MapReduce-Jobs schneller ausgeführt werden als Jobs, die eine vollständige Neuberechnung durchführen. Diese Performanzsteigerung ist umso deutlicher, je geriger die Änderungsrate an den Basisdaten ist.

Da inkrementelle MapReduce-Jobs vieles gemeinsam haben und es diverse Ansätze gibt, die vom Benutzer selbst entwickelt werden müssen, wurden diese Gemeinsamkeiten extrahiert und ein Framework namens Marimba erstellt. Vor allem Entwickler, welche mit dem Hadoop-Framework vertraut sind, können mit dem Marimba-Framework schnell eigene Jobs schreiben. Dabei muss der Anwender nur einige anwendungsspezifische Methoden implementieren, die sich um das Lesen, Aggregieren, Invertieren und Schreiben von Daten kümmern. Um die Selbstwartbarkeit kümmert sich das Framework. Da Marimba auf Hadoop aufbaut, hat der Entwickler auch gleichzeitig alle Vorteile von Hadoop. Dazu zählen die Fehlerbehandlung und das Scheduling.

Die beiden Strategien IncDec und Overwrite stellen zwei Alternativen dar, wie die inkrementelle Neuberechnung durchgeführt wird. Das Marimba-Framework unterstützt beide Strategien, welche der Benutzer einfach über eine Konfiguration einstellen kann. Nutzt ein Marimba-Job die IncDec-Strategie, liest er nur die Änderungen und erzeugt daraus positive oder negative Inkremente, welche direkt auf die HBase-Tabelle, in der das Ergebnis der letzten Berechnung zu finden ist, angewendet werden können. Bei der Overwrite-Strategie werden die Ergebnisse der vorherigen Berechnung komplett eingelesen. Das bedeutet, dass der MapReduce-Job aus zwei Quellen Eingaben lesen muss: Die Tabelle mit den alten Ergebnissen sowie die eigentlichen geänderten Daten. Auch darum kümmert sich das Framework, sodass der Benutzer kein eigenes Eingabeformat dafür entwickeln muss.

Mit dem Marimba-Framework wurden vier Beispielanwendungen entwickelt: WordCount, Reverse Web-Link Graph, Friends-Of-Friends sowie Bigrams. Die Entwicklung aller vier Jobs ist in Marimba auf eine sehr einfache Art möglich. Statt der aus Hadoop verwendeten Klassen Mapper und Reducer implementieren Marimba-Entwickler Translator, Serializer und eine Abelian-Klasse. Ein vollständiges Code-Beispiel für den Marimba-Job WordCount befindet sich im Anhang.

Anhang

8.1 WordCount (Marimba-Job)

WordCount und WordCountTranslator

```
public class WordCount extends MarimbaJobBase implements Tool {

  private static final String INPUT_DIR = "input.dir";
  private static final String OUTPUT_TABLE = "output.table";
  private static final String USE_WINDOW = "use.window";

  public static class WordCountTranslator
  extends Translator<LongWritable, Text> {
    @Override
    public void translate(LongWritable key, Text value,
    Context context) throws IOException, InterruptedException {
      String line = value.toString().toLowerCase();
      StringTokenizer tokenizer = new StringTokenizer(line);

      while (tokenizer.hasMoreTokens()) {
        String word = tokenizer.nextToken();

        context.write(new WordAbelian(word, 1L));
      }
    }

  }

  @Override
  public int run(String[] args) throws Exception {
    IncJob job = createIncJob();

    if(getConf().getBoolean(USE_WINDOW, true)) {
      job.setInputFormatClass(TextWindowInputFormat.class);
    } else {
      job.setInputFormatClass(TextInputFormat.class);
    }

    job.setOutputTable(this.getConf().get(OUTPUT_TABLE));

    job.setAbelianClass(WordAbelian.class);
    job.setTranslatorClass(WordCountTranslator.class);
    job.setSerializerClass(WordHBaseSerializer.class);
```

```
    String in = this.getConf().get(INPUT_DIR);
    TextInputFormat.setInputPaths(job, in);

    job.setJarByClass(WordCount.class);

    return job.waitForCompletion(true) ? 0 : -1;

  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(HBaseConfiguration.create(),
        new WordCount(), args);

    System.exit(res);
  }

}
```

WordHBaseSerializer

```
public class WordHBaseSerializer extends WordSerializer {

  @Override
  public Put serialize(Writable key, Abelian<?> obj)
  throws WrongClassException {

    if(!(obj instanceof WordAbelian)) {
      throw new WrongClassException();
    }
    WordAbelian word = (WordAbelian) obj;

    byte[] rowId;

    if(key instanceof ImmutableBytesWritable) {
      rowId = ((ImmutableBytesWritable) key).get();
    } else if(key instanceof Text) {
      rowId = Bytes.head(((Text) key).getBytes(),
              ((Text) key).getLength());
    } else {
      throw new WrongClassException();
    }

    byte[] colFamily = Bytes.toBytes("default");
    byte[] qualifier = Bytes.toBytes("details");
    byte[] value = Bytes.toBytes(word.getCount().get());
```

```
    Put put = new Put(rowId);
    put.add(colFamily, qualifier, value);

    return put;
  }
}
```

WordAbelian

```
public class WordAbelian implements Abelian<WordAbelian> {

  private Text word;
  private LongWritable count;

  public WordAbelian() {
    this.word = new Text();
    this.count = new LongWritable(0L);
  }

  public WordAbelian(String s) {
    this.word = new Text(s);
    this.count = new LongWritable(0L);
  }

  public WordAbelian(Text t) {
    this.word = t;
    this.count = new LongWritable(0L);
  }

  public WordAbelian(String s, long l) {
    this.word = new Text(s);
    this.count = new LongWritable(l);
  }

  public WordAbelian(Text t, LongWritable l) {
    this.word = t;
    this.count = l;
  }

  public void setWord(String s) {
    this.word.set(s);
  }

  public void setWord(Text t) {
    this.word = t;
  }
```

```
  public Text getWord() {
    return this.word;
  }

  public void setCount(LongWritable count) {
    this.count = count;
  }

  public LongWritable getCount() {
    return this.count;
  }

  @Override
  public void write(DataOutput out) throws IOException {
//     this.word.write(out);
    this.count.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException {
//     this.word.readFields(in);
    this.count.readFields(in);
  }

  @Override
  public int compareTo(BinaryComparable o) {
    int compare = this.word.compareTo(o);
    if (compare == 0) {
      return this.count.compareTo(o);
    }
    return compare;
  }

  @Override
  public WordAbelian invert() {
    return new WordAbelian(this.word,
      new LongWritable(-1 * this.count.get()));
  }

  @Override
  public WordAbelian aggregate(WordAbelian other) {
    return new WordAbelian(this.word,
      new LongWritable(this.count.get() + other.getCount().get()));
  }
```

```
  @Override
  public boolean isNeutral() {
    return (this.count.get() == 0L);
  }

  @Override
  public String toString() {
    return "("+this.word.toString()+", "+this.count.get()+")";
  }

  @Override
  public Abelian<WordAbelian> neutral() {
    return new WordAbelian(this.word, new LongWritable(0L));
  }

  @Override
  public ImmutableBytesWritable extractKey() {
    return new ImmutableBytesWritable(this.getWord().getBytes(),
      0, this.getWord().getLength());
  }

}
```

Abbildungsverzeichnis

Literaturverzeichnis

[1] Apache Hadoop project. http://hadoop.apache.org/.

[2] Virga: Incremental Recomputations in MapReduce. http://wwwlgis.informatik.uni-kl.de/cms/?id=526.

[3] Philippe Adjiman. Hadoop Tutorial Series, Issue #4: To Use Or Not To Use A Combiner, 2010. http://www.philippeadjiman.com/blog/2010/01/14/hadoop-tutorial-series-issue-4-to-use-or-not-to-use-a-combiner/.

[4] Kai Biermann. Big Data: Twitter wird zum Fieberthermometer der Gesellschaft, April 2012. http://www.zeit.de/digital/internet/2012-04/twitter-krankheiten-nowcast.

[5] Julie Bort. 8 Crazy Things IBM Scientists Have Learned Studying Twitter, January 2012. http://www.businessinsider.com/8-crazy-things-ibm-scientists-have-learned-studying-twitter-2012-1.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clus-ters. *OSDI*, pages 137–150, 2004.

[7] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011.

[8] Brown University Data Management Group. A Comparison of Approaches to Large-Scale Data Analysis. http://database.cs.brown.edu/projects/mapreduce-vs-dbms/.

[9] Ricky Ho. Map/Reduce to recommend people connection, August 2010. http://horicky.blogspot.de/2010/08/mapreduce-to-recommend-people.html.

[10] Yong Hu. Efficiently Extracting Change Data from HBase. April 2012.

[11] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Can mapreduce learn form materialized views? In *LADIS 2011*, pages 1 – 5, 9 2011.

[12] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *CloudDB 2011*, 10 2011.

[13] Steve Krenzel. MapReduce: Finding Friends, 2010. http://stevekrenzel.com/finding-friends-with-mapreduce.

[14] Todd Lipcon. 7 Tips for Improving MapReduce Performance, 2009. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

[15] TouchType Ltd. SwiftKey X - Android Apps auf Google Play, February 2012. http://play.google.com/store/apps/details?id=com.touchtype.swiftkey.

[16] Karl H. Marbaise. Hadoop - Think Large!, 2011. http://www.soebes.de/files/RuhrJUGEssenHadoop-20110217.pdf.

[17] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Distributed Cube Materia-lization on Holistic Measures. *ICDE*, pages 183–194, 2011.

[18] Alexander Neumann. Studie: Hadoop wird ähnlich erfolgreich wie Linux, Mai 2012. http://heise.de/-1569837.

[19] Owen O'Malley, Jack Hebert, Lohit Vijayarenu, and Amar Kamat. Partitioning your job into maps and reduces, September 2009. http://wiki.apache.org/hadoop/HowManyMapsAndReduces?action=recall&rev=7.

[20] Roya Parvizi. *Inkrementelle Neuberechnungen mit MapReduce.* Bachelorarbeit, TU Kaiserslautern, Juni 2011.

[21] Arnd Poetzsch-Heffter. *Konzepte objektorientierter Programmierung*. eXamen.press. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[22] Dave Rosenberg. Hadoop, the elephant in the enterprise, June 2012. http://news.cnet.com/8301-1001_3-57452061-92/hadoop-the-elephant-in-the-enterprise/.

[23] Marc Schäfer. *Inkrementelle Wartung von Data Cubes.* Bachelorarbeit, TU Kaiserslautern, Januar 2012.

[24] Sanjay Sharma. Advanced Hadoop Tuning and Optimizations, 2009. http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation.

[25] Jason Venner. *Pro Hadoop*. Apress, Berkeley, CA, 2009.

[26] Dick Weisinger. Big Data: Think of NoSQL As Complementary to Traditional RDBMS, Juni 2012. http://www.formtek.com/blog/?p=3032.

[27] Tom White. 10 MapReduce Tips, May 2009. http://www.cloudera.com/blog/2009/05/10-mapreduce-tips/.