Masterarbeit

Sampling mit inkrementellem MapReduce

Marc Schäfer

AG Heterogene Informationssysteme
Prüfer: Prof. Dr.-Ing. Stefan Deßloch
Betreuer: M. Sc. Johannes Schildgen
15. Mai 2014

Abstract

The main goal of Mapreduce is the computation of waste amounts of data. But such computations are time-consuming. Since there is usually the need of a real-time result, additional resources are used. A resource-efficient alternative would be the computation of an approximation. In many cases real-time approximations are prefer over exact results, which need too much time to compute. Hadoop does not have an in-built function to estimate results. The user has to implement them by himself through special map or reduce functions. In this work, an automatic system to estimate arithmetic computations is introduced. It uses statistical sampling methods and provides a generic extrapolation. An additional incremental component, expanding the sample based on old approximations, is also part of this system. The new approximation is then based on a bigger sample, this incremental approach can be used iteratively to increase the precision. All introduced components can be integrated in the Hadoop-framework without the need to change it. In the following, this Hadoop expansion will be explained in more detail. Furthermore there will be tests showing that the intended speed-up is achieved.

Zusammenfassung

Die Berechnung von großen Datenmengen ist das Hauptziel von MapReduce. Die Berechnung solcher Daten ist aber zeitintensiv. Da aber in der Regel ein zeitnahes Ergebnis benötigt wird, werden zusätzliche Ressourcen benötigt. Eine ressourcenschonende Alternative ist die Berechnung einer Abschätzung des Ergebnisses, da in vielen Fällen ein abgeschätztes zeitnahes Ergebnis einem zwar genauen aber späten Ergebnis vorgezogen wird. Da Hadoop eine solche Abschätzung nicht von Hause aus unterstützt, ohne dass der Benutzer diese durch spezielle Map- und Reduce-Funktionen ermöglicht, wird in dieser Arbeit ein solches automatisches System zur Abschätzung arithmetischer Funktionen vorgestellt. Dabei werden unterschiedliche statistische Ansätze zur Ziehung der Stichprobe verwendet, sowie eine generische Hochrechnungsfunktion. Eine zusätzliche inkrementelle Funktionalität, die es ermöglicht, eine bereits berechnete Schätzung für eine neue Schätzung wiederzuverwenden, ist in dem System auch enthalten. Dadurch erhält man eine Abschätzung basierend auf einer größeren Stichprobe, bei der man nur die Erweiterung der Stichprobe neu berechnen muss. Diesen Vorgang kann man dabei iterativ wiederholen, bis man mit der Genauigkeit des Ergebnisses zufrieden ist. Dabei müssen keine Änderungen am Hadoop-Framework vorgenommen werden, um diese Funktionalitäten zu unterstützen. Im Folgenden wird diese Erweiterung für Hadoop genauer erläutert, sowie durch Test nachgewiesen, dass eine Laufzeitverringerung durch diese Komponenten erreicht wird.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der heutigen Informationsgesellschaft ist die Analyse von Daten für viele Unternehmen von großer Bedeutung. Solche Daten von Interesse fallen dabei zu großen Mengen an, und davon können Unternehmen und Wirtschaft profitieren[8]. Durch die immer stärkere Verflechtung von Alltag und Computern wächst diese Datenmengen auch stetig weiter. In 2013 nahm diese pro Tag um ungefähr 2,5 Exabyte zu[13]. Dabei waren für zwei Drittel der entstandenen Daten keine Unternehmen verantwortlich, sondern das Verhalten und Handeln von Nutzern[19]. Und dieses Datenwachstum nimmt auch immer weiter zu. Von der Gesamtmenge aller Daten in 2013 wurden 90% in einem Zeitraum von 2 Jahren generiert[13]. Und wenn man sich dann Prognosen[19, 5] vor Augen führt, dass sich die Gesamtmenge aller Daten ca. alle 1,5-2 Jahre verdoppelt, ist eine effektive Datenverarbeitung essentiell für eine erfolgreiche Unternehmensführung. Mittels der Analyse von unterschiedlichsten Daten lassen sich Grundlagen für Entscheidungsprozessen schaffen. Um sich den jeweils gegebenen Situationen anzupassen ist aber ein zeitnahes Analyseergebnis wichtig, um sich gegen die Konkurrenz behaupten zu können[17, 8]. Da die Daten in einer immer höheren Frequenz und Volumen generiert werden, müssen die Systeme zur Verarbeitung und Analyse Schritt halten können[5]. Dabei stoßen aber selbst verteilte und parallele System wie MapReduce[6] irgendwann an ihre Grenzen. Die Daten können zwar analysiert werden, aber ein zeitnahes Ergebnis zu erhalten wird immer schwieriger, da die Berechnungszeit mit der Datenmenge zunimmt. Der Einsatz zusätzlicher Ressourcen kann zwar verwendet werden um einer wachsenden Berechnungszeit Herr zu werden, aber auch hier sind Grenzen gesetzt, wie zum Beispiel physische und ökonomische. Auch lassen sich manche Berechnungen nur bis zu einem gewissen Grad parallelisieren. Eine sinnvolle Alternative kann auch auch sein ein abgeschätztes Ergebnis in kürzerer Zeit zu erhalten als ein genaues Ergebnis zu spät. Solch ein geschätztes Ergebnis kann man mittels einer Hochrechnung einer Analyse einer Stichprobe der Daten erhalten. Solche Schätzungen können dann Entscheidungsprozesse unterstützen oder zur Verbesserung der Analyse aller Daten herangezogen werden. Somit ist es sinnvoll, je nach Situation zwischen Genauigkeit, Berechnungszeit und Ressourceneinsatz abzuwägen, da sich diese Faktoren gegenseitig beeinflussen. Sollte man eine höhere Genauigkeit anstreben, muss die Stichprobe größer gewählt werden. Eine größere Stichprobe führt zu einem höheren Aufwand, welcher sich in einer höheren Berechnungszeit oder in einem zusätzlichen Bedarf an Ressour-

cen niederschlägt. Will man die Berechnungszeit verbessern, muss man mehr Ressourcen einsetzen oder Abstriche bei der Genauigkeit machen. Bei Einsparungen von Ressourcen erhöht sich die Berechnungszeit oder man gleicht es durch eine geringere Genauigkeit aus. Eine optimale Lösung mit minimalen Ressourcen Daten in schnellst möglicher Zeit mit maximaler Genauigkeit zu berechnen ist nicht möglich. Es müssen immer Abstriche bei mindestens einem Faktor gemacht werden.

Das Ziel dieser Arbeit ist es Hadoop[1][1] eine Funktionalität zur Verfügung zu stellen, mit der sich die Berechnungszeit verkürzen lässt, indem man das Ergebnis abschätzt. Ergänzen soll dies eine zusätzliche inkrementelle Erweiterung zur Erhöhung des Stichprobenumfangs. Diese generischen Komponenten soll sich dabei in das Framework einfügen, ohne dass daran Änderungen vorgenommen werden müssen. Dadurch können die Funktionalitäten auch bei zukünftigen Hadoop-Versionen verwendet werden, falls keine fundamentalen Änderungen am Framework vorgenommen werden. Des weiteren soll eine Benutzung sich möglichst nicht von der einer normalen kompletten Berechnung unterscheiden und damit benutzerfreundlich sein.

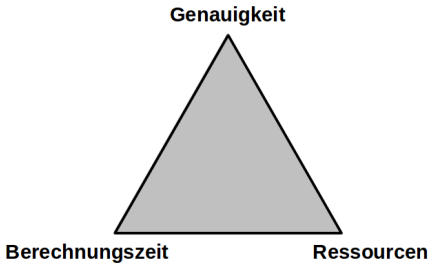


Abbildung 1.1: Die einzelnen Faktoren beeinflussen sich gegenseitig.

Dafür wird zuerst eine Komponente zur Stichprobenziehung entwickelt, welche verschiedene statistische Auswahlverfahren bereitstellt. Anschließend wird eine Komponente implementiert, die eine automatische Hochrechnung des Teilergebnisses der Stichprobe umsetzt. Dabei werden diese so gestaltet, dass ein Benutzer nur Map- und Reduce-Funktionen bereitstellen muss, welche für eine komplette Berechnung des jeweiligen Ergebnisses nötig wären. Aufbauend auf dieser einmaligen Ausführung einer Schätzung wird eine inkrementelle Erweiterung entwickelt, welche eine bereits berechnete Schätzung mit einer zusätzlichen Schätzung kombiniert, um so den Stichprobenumfang zu erhöhen, ohne dass eine komplette Neuberechnung nötig ist. Bei dieser Erweiterung ist der Nutzer für Map- und Reduce-Funktionen zur Handhabung der Repräsentation der Daten zuständig. Die Umsetzung der inkrementellen Erweiterung basiert dabei auf den Ideen eines inkrementellen Verfahrens zur Aktualisierung von Ergebnissen[14].

Die Arbeit gliedert sich dabei in folgende Kapitel: In Kapitel 2 werden zuerst die Grundlagen, wie Speichersystem, MapReduce, Sampling und Samplingansätze, besprochen. In Kapitel 3 wird Related Work wie inkrementelles MapReduce[14] und EARL[15][2] aufgeführt. Im Kapitel 4 werden die Überlegungen und Ideen zur Umsetzung der neuen Komponeten erläutert und in Kapitel 5

[1] Eine Java Implementierung von MapReduce.
[2] Early Accurate Result Library

eine inkrementelle Erweiterung vorgestellt. Anschließend folgt dann in Kapitel 6 die Implementierung der Komponenten. Kapitel 7 enthält die Evaluierung und Kapitel 8 schließt die Arbeit mit einer Zusammenfassung.

Kapitel 2

Grundlagen

Im Folgenden werden die Grundlagen wie HDFS[3], MapReduce[6][1] und Sampling[9] näher erläutert.

2.1 HDFS

Das Hadoop Distributed File System (HDFS)[3] ist das verteilte Dateisystem des Apache Hadoop Projekts[1]. Es basiert dabei auf dem Google File System (GFS)[10]. Das Hauptaugenmerk liegt auf der Speicherung großer Datenmengen und einer guten Skalierbarkeit. Außerdem verfolgt es den Ansatz die Berechnungen zu den Daten zu bringen, da dies effizienter ist als die Daten zu den Berechnungen zu transportieren, was gerade bei großen Datenmengen das Netzwerk stark belasten kann. Durch redundantes Speichern wird eine erhöhte Fehlertoleranz, sowie eine bessere Verfügbarkeit erreicht.

2.1.1 Realisierung

HDFS ist eine in Java implementierte Master/Slave-Architektur. Der NameNode repräsentiert hierbei den Master-Knoten und die DataNodes bilden die Menge der Slave-Knoten.

NameNode: Der NameNode kümmert sich um den Zugriff auf die Daten und die Metadatenverwaltung.

DataNode: Die eigentlichen Lese- und Schreibanfragen werden mittels der DataNodes bearbeitet. Des weiteren sind sie auch für die Blockverwaltung zuständig.

Beim Speichern in HDFS werden die Daten zuerst in Blöcke aufgeteilt, standardmäßig 64MB, für ein besseres sequentielles Lesen und um die benötigten Metadaten klein zu halten. Diese Blöcke werden dann zufällig auf die DataNodes verteilt.
Bei einer großen Menge von Knoten besteht immer die Gefahr, dass einzelne Knoten ausfallen und es zu einem Datenverlust kommt. Mittels Blockreplikation wird ein solches Risiko minimiert, da die Daten üblicherweise mehrfach auf

[1] Im speziellen Hadoop[1].

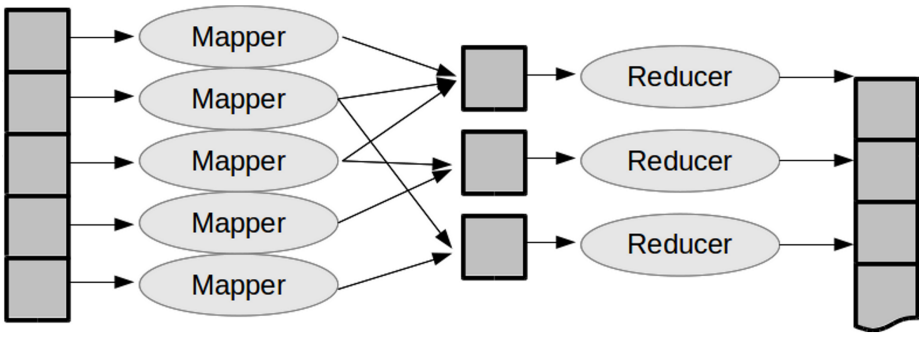


Abbildung 2.1: Das MapReduce-Programmiermodell

unterschiedlichen Knoten vorliegen. Da einmal gespeicherte Daten nicht mehr geändert werden, ist somit eine potentielle Inkonsistenz aufgrund der Redundanz unwahrscheinlich. Soll nun auf die Daten zugegriffen werden, wird der NameNode kontaktiert, dieser leitet dann die Adressen der angeforderten Blöcke an den Client weiter. Der eigentliche Ladeprozess erfolgt dann, indem der Client pro Block den entsprechenden DataNode kontaktiert und die Daten von diesem erhält.

2.2 MapReduce

Das von Google entwickelte Programmiermodell MapReduce[6] beschäftigt sich mit der Bearbeitung von großen Datenmengen. Eine lange Berechnungszeit bei solch umfangreichen Ausgangsdaten wird begegnet, indem eine verteilte Berechnung auf einem Cluster durchgeführt wird.
Ein dazugehöriges Framework erleichtert die Nutzung. Die bei der parallelen Datenverarbeitung anfallende Lasten- und Datenverteilung, sowie die Fehlerbehandlung fallen in die Zuständigkeit des Frameworks. Das Laden der Eingabedaten, das Verteilen der Zwischenergebnisse und das Speichern des Endergebnisses in dem verteilten Dateisystem ist somit nicht durch den Benutzer zu regeln. Dieser muss sich nur auf die zwei Schritte der Datenverarbeitung konzentrieren. Angelehnt an die funktionale Programmierung werden sie als Map- und Reduce-Schritt bezeichnet.

2.2.1 Programmiermodell

Zu Beginn werden die Eingabedaten aufgeteilt und dann Stück für Stück der benutzerdefinierten Map-Funktion übergeben. Die Map-Funktion gibt Schlüssel/-Wert-Paare aus, diese Zwischenergebnisse werden anhand ihrer Schlüssel aggregiert. Anschließend wird pro Schlüssel und der dazugehörigen Wertemenge eine benutzerdefinierte Reduce-Funktion aufgerufen. Die einzelnen Ergebnisse der Reduce-Funktion bilden das Ergebnis. Siehe Abbildung 2.1.

Map-Phase

In der Map-Phase wird die Map-Funktion auf die Eingabedaten angewendet, diese liegen als vom Eingabeformat abhängige Schlüssel/Wert-Paare vor. Die Map-

Funktion muss aufgrund der Parallelität der Datenverarbeitung unabhängig berechenbar sein.
Das Ergebnis der Map-Phase sind wieder Schlüssel/Wert-Paare, diese bilden das Zwischenergebnis.
Bei der Verwendung einer Stichproben, werden nicht alle Daten von der Map-Funktion berechnet, sondern nur eine Teilmenge.

Reduce-Phase

Die Reduce-Funktion erhält alle Werte, die einem bestimmten Schlüssel zugeordnet sind, aus den Zwischenergebnissen. Da auch hier eine parallele Verarbeitung durchgeführt wird, muss die Reduce-Funktion auch unabhängig berechenbar sein.
Die Ergebnisse der einzelnen Reduce-Aufrufe bilden konkateniert das Gesamtergebnis, wobei die Reihenfolge der Ausgaben nicht festgelegt ist und somit von Durchführung zu Durchführung variieren kann.
Sollte auf die Reduce-Phase verzichtet werden, bilden die Zwischenergebnisse das Gesamtergebnis.
Bei einer Ergebnisabschätzung, basierend auf einer Stichprobe, müsste das Endergebnis noch zusätzlich hochgerechnet werden, um der Tatsache gerecht zu werden, dass nur ein Teil der Eingabedaten in das Ergebnis eingeflossen ist.

2.2.2 Framework

Durch das Framework wird die Nutzung erleichtert, da es Rahmenfunktionalitäten bereitstellt. Der Anwender muss sich nicht explizit um Lastenverteilung, Knotenausfälle und die Datenübertragung kümmern. Hier sollen die generischen Komponenten zur normalen und inkrementellen Abschätzung hinzugefügt werden.
Im Folgenden wird auf Apache Hadoop[1], einer Java Implementierung der MapReduce-Frameworks, genauer eingegangen. Hadoop wurde auch im Zusammenhang der Arbeit genutzt.

Ausführung

Beim Laden der Eingabedaten werden diese in sogenannte *Splits* aufgeteilt. Diese *Splits* sind in der Regel 64 MB groß. Auf den jeweiligen Knoten des Clusters werden dann Kopien des Programms ausgeführt. Basierend auf dem Master/Slave-Ansatz ist eines dieser Programme der *Master*, die anderen sind sogenannten *Workern*. Der *Master* teilt den *Workern* Map- bzw. Reduce-Aufträge zu. Da der *Master* über die Verteilung und den Fortschritt dieser *Tasks* informiert ist, behält er einen Überblick über den Gesamtfortschritt.
Worker mit einem Map-Task laden den ihnen zugeteilten *Split*. Anhand des Eingabeformats wird dieser *Split* dann in Schlüssel/Wert-Paare aufgeteilt und der Map-Funktion übergeben. Die Ergebnisse der Map-Funktion werden im Arbeitsspeicher gespeichert und anschließend periodisch auf der lokalen Festplatte geschrieben. Nachdem die Daten auf der Festplatte gesichert wurden, wird der *Master* über deren Speicherort informiert.
Nach der erfolgreichen Ausführung aller Map-Tasks, werden die Reduce-Tasks an die *Worker* verteilt. Diese erhalten zusätzlich die Adressen der von ihnen

benötigten Zwischenergebnissen. Anschließend wird pro Schlüssel die Reduce-Funktion mit den zugehörigen Werten ausgeführt. Das Ergebnis der Reduce-Funktion bildet dann zusammen mit den Ergebnissen der restlichen Reducer das Gesamtergebnis.

Fehlerbehandlung

Aufgrund der Anzahl der Knoten besteht ein hohes Risiko, dass einzelne Knoten ausfallen. Um diesem Umstand gerecht zu werden, überprüft der *Master* regelmäßig die verschiedenen Knoten. Sollte einer ausgefallen sein, wird dessen Aufgabe neu verteilt. Bereits abgeschlossene Aufgaben eines Knoten müssen nur dann wiederholt werden, wenn es ein Map-Task war, da dessen Ergebnisse lokal auf dem ausgefallenen Knoten im Hauptspeicher gespeichert waren.

Lokalität und Backup-Tasks

Um die Performanz zu verbessern, werden den Mappern *Splits* zugeteilt, die auf dem gleichen Knoten gespeichert sind oder auf Knoten in geringer Entfernung. So sollen Übertragungen auf das Nötigste reduziert werden.
Ein anderes Instrument zur Performanzoptimierung sind die sogenannten *Backup-Tasks*. Sollte ein *Worker* für einen Task ungewöhnlich lange brauchen, wird die Aufgabe zusätzlich einem anderen Knoten übertragen. Zur erfolgreichen Durchführung des Tasks reicht es, wenn ein Knoten die Aufgabe abschließt.

Eingabeformate

Es gibt bereits verschiedene Eingabeformate für unterschiedliche Daten, welche in der Regel für den normalen Gebrauch ausreichen. Bei den Eingabeformaten für HDFS[3] werden normalerweise die zu lesenden Blöcke der einzelnen Dateien in *Splits* festgelegter Größe aufgeteilt. Für jeden *Split* wird dann ein *RecordReader* aufgerufen, der die *Splits* nach einem gegebenen Muster in Schlüssel/Wert-Paare für die Map-Funktion umwandelt.
Bei dem *TextInputFormat* bilden zum Beispiel die einzelnen Zeilen den Wert und der jeweilige Zeilenoffset den Schlüssel.

Configuration und Job-Objekt

Für das Setzen der benötigten Klassen in einer Hadoop-Ausführung ist das Job-Objekt verantwortlich. Dieses erhält unter anderem die benutzerdefinierten Klassen, welche dann vom Job-Objekt in der *Configuration* gespeichert werden. Anhand der in der *Configuration* enthaltenen Parameter wird dann das Programm ausgeführt.

Mapper und Reducer

Der Mapper und der Reducer sind Klassen, die für den Map- bzw. den Reduce-Schritt verantwortlich sind. Die jeweilige Klassen erthält ein *Context*-Objekt, dass der Klasse alle zugeteilten Ausgabe-Schlüssel und Werte mitteilt. Die benutzerdefinierte Map- bzw. Reduce-Funktion wird in einer Methode gleichen Namens implementiert, standardmäßig wird eine Identitätsfunktion verwendet.

Die Ergebnisse der Funktion werden wiederum mit dem *Context*-Objekt, welches Zugriff auf den Ausgabe-Stream hat, gespeichert.

Writable

Hadoop verwendet *Writables* als Typen für die Werte bzw. WritableComparables für die Schlüssel[2]. Die *Writables* dienen der Serialisierung in Hadoop, sie müssen deshalb Methoden zum Schreiben und zum Lesen eines binären Streams bereitstellen.
Für die verschiedenen primitiven Java-Typen gibt es *Writables* die als Wrapper-Klassen fungieren, wie z.B. *IntWritable* für den Java-Typ *int*.

2.3 Sampling

Beim Sampling[9] geht es darum, aus einer Grundgesamtheit eine Stichprobe zu ziehen und anhand dieser, die Eigenschaften der Grundgesamtheit abzuschätzen. Dabei werden die Elemente basierend auf einer bestimmten Wahrscheinlichkeit ausgewählt. Diese kann von Element zu Element variieren. Dadurch versucht man eine repräsentative Teilmenge zu erhalten um effektiv auf die Grundgesamtheit schließen zu können.

2.3.1 Verfahren

Es gibt unterschiedliche Verfahren wie man eine zufällige Stichprobe auswählt. Im Nachfolgenden werden drei dieser Verfahren genauer erläutert.

Einfache Zufallsstichprobe

Das Verfahren der *Einfachen Zufallsstichprobe* ist das naheliegendste Auswahlverfahren. Bei diesem Verfahren gibt es eine festgelegte Wahrscheinlichkeit p, auf welcher basierend alle Elemente der Stichprobe ausgewählt werden. Somit hat jedes Element der Grundgesamtheit die gleiche Wahrscheinlichkeit in die Stichprobe zu kommen. Basierend auf dem *Gesetz der großen Zahlen*[9] erhält man für eine n-elementige Menge einer Stichprobe mit einem ungefähren Umfang von $n * p$.

Geschichtete Zufallsstichprobe

Bei der *Geschichteten Zufallsstichprobe* wird die Grundgesamtheit in Teilmengen zerlegt, sogenannte *Schichten*, aus denen dann anschließend *einfache Zufallsstichproben* gezogen werden. Dabei können aus unterschiedlichen Schichten unterschiedlich große Stichproben gezogen werden, um eventuelle Gewichtungen vorzunehmen. Dieses Verfahren eignet sich um genauere Schätzungen zu erlangen. Um die Schichten aufzuteilen sollte dabei ein Merkmal ausgewählt werden, das mit einem gesuchten Merkmal stark korreliert. Bei der Aufteilung der Schichten ist das Ziel die Schichten homogen bezüglich der beinhalteten Elemente zu gestalten, dabei aber zwischen den Schichten eine Heterogenität zu erreichen.

[2]WritableComparable implementiert zusätzlich das Comparable-Interface da es nötig ist Schlüssel mit einander vergleichen und sortieren zu können

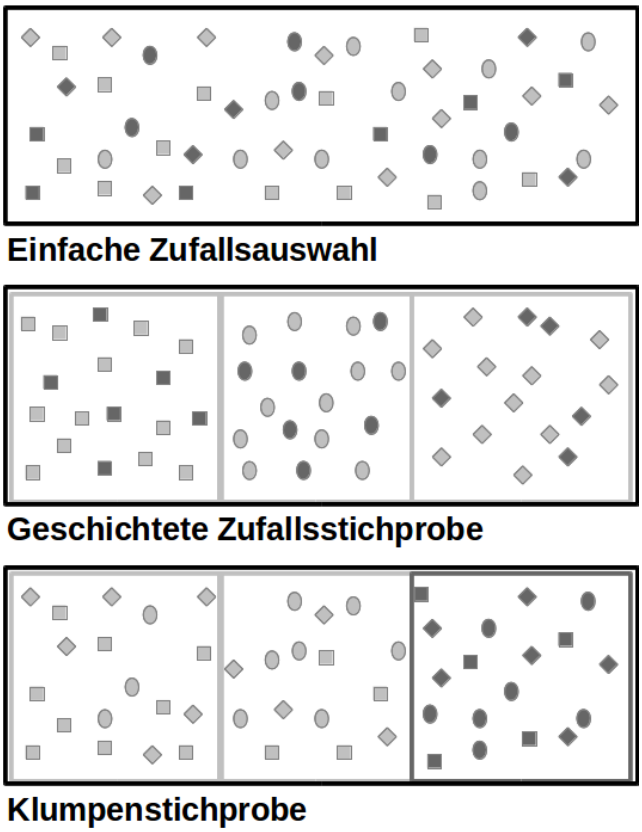


Abbildung 2.2: Beispiele für die drei vorgestellten Auswahlverfahren.

Klumpenstichprobe

Ähnlich zur *Geschichteten Zufallsstichprobe* wird die Grundgesamtheit auch hier in Teilmengen aufgeteilt. Diese Teilmengen, hier *Klumpen*, werden dabei *natürlich* abgegrenzt, also anhand von Faktoren wie zum Beispiel Lokalität oder anderen gegebenen Faktoren. Dies kann unter Umständen effizienter sein als die Teilmengen im Vorfeld *manuell* abzugrenzen und die einzelnen Elemente zusammenzuführen. Aus dieser Menge an Klumpen werden anschließend einzelne Klumpen zufällig ausgewählt und vollständig für die Stichprobe verwendet.
Hierbei besteht das Risiko, dass die Klumpen zu homogen sind und somit nicht repräsentativ für die Grundgesamtheit sind. Dies kann zu einer Verzerrung des geschätzten Ergebnisses führen.
Es gibt zusätzlich noch mehrstufige Auswahlverfahren der Klumpenauswahl. Bei einer zweistufigen Klumpenauswahl werden zum Beispiel aus den ausgewählten Klumpen *einfache Zufallsstichproben* gezogen, um das Risiko der Verfälschung zu minimieren.

Kapitel 3

Related Work

In diesem Kapitel werden zwei Ansätze vorgestellt, welche auch das Ziel der Zeitersparnis haben und im Rahmen der Arbeit kombiniert werden. Angefangen wird mit der EARL-Komponente[15], welche auch eine Laufzeitoptimierung basierend auf einer Ergebnisabschätzung durchführt. EARL hat dabei einen ähnlichen Ansatz wie die im Verlauf der Arbeit vorgestellte Komponente, weißt jedoch auch Unterschiede auf. Anschließend wird der Ansatz des inkrementellen MapReduce[14] betrachtet. Dabei geht es speziell um die Berechnung veränderter Eingabedaten. Die dabei verwendete Vorgehensweise ist die Grundlage für die inkrementelle Aktualisierung von Schätzungen, welche in Kapitel 5 behandelt wird.

3.1 EARL

Die *Early Accurate Result Library (EARL)* ist eine Erweiterung für Hadoop, welche in [15] vorgestellt wird. Dabei wird das Ziel verfolgt, einen Näherungswert mit einer bestimmten Genauigkeit zu berechnen.
Dabei wird das Verfahren *Bootstrapping* verwendet, ein Resampling-Verfahren. Hierbei wird der Fehler anhand des Variationskoeffizienten zwischen der Standardabweichung und dem arithmetischen Mittel herangezogen. Ein iterativer Sampling-Prozess wird dabei solange durchgeführt, bis die benutzerdefinierte Genauigkeit erreicht ist.

3.1.1 Architektur

EARL gliedert sich in drei Teile. Eine Stichprobe wird im ersten Teil, der *Sampling Stage*, gesammelt. Aus dieser Stichprobe werden dann die als *Resamples* bezeichneten Unterstichproben gewonnen. Diese dienen als Eingabedaten für mehrere MapReduce-Berechnungen[1], welche den zweiten Teil darstellen. In der *Accuracy Estimation*, dem dritten Teil, wird dann aus den erhaltenen Ergebnissen deren Verteilung hergeleitet, die dann zur Bestimmung der Genauigkeit verwendet wird.
Um dies umzusetzen wurden drei Änderungen am Hadoop-Framework vorgenommen. Mapper werden nicht mehr automatisch beendet, wenn sie die ihnen

[1] Pro Resample wird eine MapReduce-Berechnung durchgeführt.

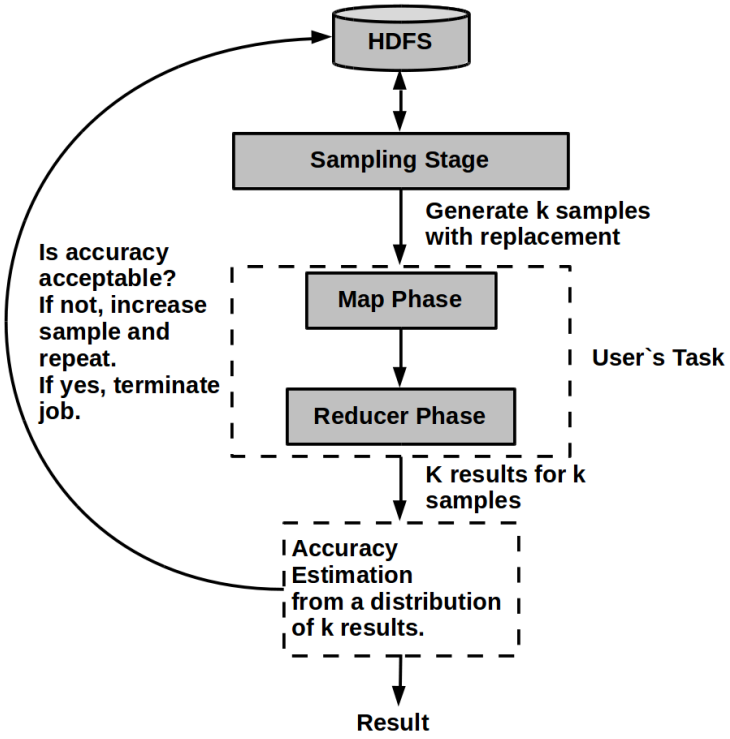


Abbildung 3.1: Vereinfachte EARL-Architektur aus [15]

zugeteilten Aufgaben erledigt haben, sondern werden erst beendet, wenn die angestrebte Genauigkeit erreicht ist. Zusätzlich müssen Reducer nicht mehr auf das Beenden der Mapper warten, bevor sie starten können[2]. Als dritte Änderung wurde eine Kommunikationsschicht zwischen Mappern und Reducern hinzugefügt. Diese dient der Kontrolle des Terminierungskriteriums. Mittels dieser Kommunikationsschicht ist es den Mapper möglich, die Genauigkeit zu berechnen. Sollte diese noch nicht der verlangten Genauigkeit entsprechen, so wird die initiale Stichprobe vergrößert. Da die Mapper noch aktiv sind, müssen keine neuen Map-Tasks gestartet werden. Sollte die verlangte Genauigkeit erreicht sein, beenden sich die Mapper.
Der Reduce-Task wird auch erweitert und in vier Schritte aufgeteilt:

1. *initialize()*
2. *update()*
3. *finalize()*
4. *correct()*

Im *initialize()*-Schritt wird aus den Werten der Zwischenergebnisse ein *state* für jedes *Resample* gebildet, dieser *state* wird dabei mittels der benutzerdefinier-

[2]Unter anderem ist dies nötig, da sich wegen der ersten Änderung ansonsten ein Deadlock bilden würde.

ten Reduce-Funktion berechnet. *State*s werden verwendet, um Speicherplatz zu sparen, da jedes *Resample* ein eigenes Ergebnis besitzt. Die *update()*-Funktion wird für die Aktualisierung eines *state* genutzt. Dabei kann als Eingabe ein weiterer *state* dienen, aber auch ein Schlüssel/Wert-Paar. Diese Funktion basiert dabei auch auf der Reduce-Funktion und ermöglicht das verwendete iterative Vorgehen, da Eingabedaten gelesen und berechnet werden müssen, bis das Terminierungskriterium erreicht ist. Im *finalize()*-Schritt wird der aktuelle Fehler anhand der der einzelnen *Resample*-Ergebnisse berechnet. Dieser Fehler setzt sich dabei aus den Einzelfehlern zusammen. Darüber hinaus wird in diesem Schritt auch das Gesamtergebnis berechnet. Im letzten Schritt, dem *correct()*-Schritt, wird das Ergebnis noch berichtigt. Dies ist nötig da nur eine Teilmenge berechnet wurde und man somit nur eine Teilergebnis erhalten hat. Um ein abgeschätztes Gesamtergebnis zu erhalten, ist eine Korrektur nötig, diese ist abhängig von der verwendeten Reduce-Funktion. Da das System nichts über die Reduce-Funktion weiß, ist eine benutzerdefinierte Korrekturfunktion nötig. Für das Beispiel einer Berechnung der Summe aller Elemente, für eine Stichprobe mit der Wahrscheinlichkeit p, müsste man das erhaltene Ergebnis mit dem Faktor $\frac{1}{p}$ multiplizieren.

3.1.2 Sampling-Ansätze

EARL verwendet zwei Ansätze zur Bestimmung der Stichprobe, welche im Folgenden genauer erklärt werden.

Pre-Map-Sampling

Beim *Pre-Map-Sampling* wird ein festgelegter Prozentsatz der Eingabedaten geladen. Dies geschieht bevor die Daten den Mappern übergeben werden. Somit muss nur ein Bruchteil der Daten berechnet werden, was zu einer Zeitersparnis führt. Der Nachteil dieses Ansatzes ist aber, dass er ungenau sein kann, da in der Regel die Daten zeilenweise gelesen werden und aus den Zeilen potentiell unterschiedlich viele Schlüssel/Wert-Paare der Map-Ausgabe generiert werden. Somit kann auch eine gewisse Ungenauigkeit im Bezug auf die Korrekturfunktion auftreten, da nur der Prozentsatz der Datenmenge verwendet wird und nicht der Prozentsatz der Anzahl der betrachteten Wertpaare.

Post-Map-Sampling

Beim *Post-Map-Sampling* wird die Stichprobe von der Menge der Schlüssel/Wert-Paare des Zwischenergebnisses der Mapper entnommen. Dabei wird anhand einer Hash-Funktion, die auf dem benötigten Umfang der Stichprobe basiert, das Sampling durchgeführt. Die erhaltene Stichprobe ist im Bezug auf die Anzahl der Datensätze genauer als beim *Pre-Map-sampling*. Da aber alle Daten erst mit der Map-Funktion verarbeitet werden müssen, ist dieser Ansatz langsamer.

3.1.3 Optimierung

EARL verwendet zusätzlich zwei Optimierungen. Zum einen eine *Inter-Iteration*-Optimierung. Hierbei wird der Prozess der Erhöhung des Stichprobenumfangs verbessert. Dabei wird die neue Teilmenge, aus der die *Resamples* gezogen werden, aus der ursprünglichen Teilmenge und einer neuen Menge gebildet. Um

Zeit zu sparen, werden bereits berechnete *Resamples* aus einer vorherigen Iteration wiederverwendet. Die Anzahl der *Resamples* aus der alten Teilmenge kann dabei zur Anzahl der aus der vorherigen Iteration variieren. Das bedeutet, dass potentiell neue *Resamples* aus der alten Teilmenge gezogen werden müssen oder *Resamples* verworfen werden. Um Speicherplatz zu sparen, werden die für jede Iteration zusätzlichen Mengen und *Resamples* als Veränderungen(Deltas) zur vorherigen Iteration gespeichert. Von diesen *Delta*-Mengen wird nun nur ein Teil im Arbeitsspeicher gehalten. Falls zusätzliche *Resamples* eines Iterationsschritts benötigt werden kann aus der *Delta*-Stichprobenmenge[3] ein neues *Resample* gezogen werden. Falls die Anzahl der *Resamples* eines Iterationsschrittes verringert werden soll, werden *Resamples* aus der *Delta*-Menge der *Resamples*[4] gelöscht.
Die zweite Optimierung ist die *Intra-Iteration*-Optimierung, welche für das Ziehen der *Resamples* verbessert. Beim *Resampling* kann es zu Überschneidungen der einzelnen *Resamples* kommen. Abhängig vom Umfang und der daraus resultierender Wahrscheinlichkeit, wird bestimmt, wie groß diese Überschneidung voraussichtlich ist. In diesem Umfang werden dann *Resamples* wiederverwendet und man spart die erneute Berechnung.

3.1.4 Fazit

Der EARL-Ansatz ermöglicht zwar eine Abschätzung von Ergebnissen vorzunehmen und somit die Berechnungszeit zu optimieren, dies geschieht aber durch Änderungen des Hadoop-Frameworks und macht den Ansatz somit abhängig von der verwendeten Hadoop-Version. Dies steht im Gegensatz zum Entwurf dieser Arbeit, der eine Abhängigkeit zu einer speziellen Version vermeidet indem die Komponenten in das Framework integriert werden ohne dass Änderungen nötig sind. Des weiteren stellt EARL nur eine automatische Stichprobenziehung bereit, die Hochrechnungsfunktionalität, hier `correct()`, muss der Nutzer selbst unter Berücksichtigung seiner Reduce-Funktion implementieren. Dies wird im vorgestellten Entwurf vermieden, hier muss der Nutzer nur entscheiden, ob er die Funktionalität aktiviert oder eine eigene Hochrechnung im Rahmen der Reduce-Funktion verwenden möchte. Durch die Verwendung von Bootstrap[7] in EARL wird auch nicht eine Berechnung der Stichprobe durchgeführt, sondern mehrere Berechnungen auf Unterstichproben. Somit ist eine Vergrößerung der Stichprobe zwar möglich, aber dies geschieht nur im Rahmen einer einzelnen MapReduce-Berechnung. Eine Iteration basierend auf bereits berechneten Ergebnissen ist hierbei aber nicht möglich.

3.2 Inkrementelle Neuberechnung in MapReduce

Beim inkrementellen MapReduce[14] wird die Methodik der inkrementellen Sichtwartung[11] auf MapReduce-Ergebnisse angewendet. So kann man eine komplette Neuberechnung umgehen, wenn sich nur ein Teil der Ausgangsdaten geändert hat. Dies wird ermöglicht, indem man die Änderungen der Ausgangs-

[3] Die Menge, die die einzelnen Datenelemente enthält.
[4] Die Menge, die die *Resamples* vergangener Iterationen enthält.

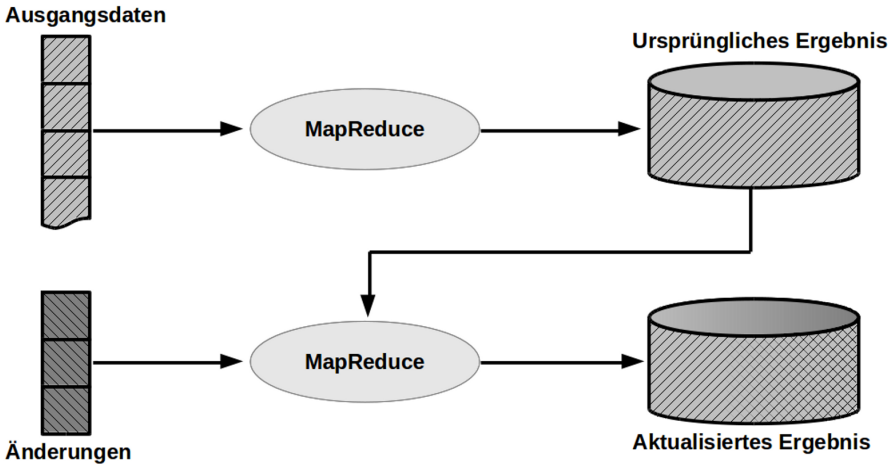


Abbildung 3.2: Ablauf einer normalen MapReduce-Berechnung mit anschließender Overwrite-Berechnung.

daten und das veraltete Ergebnis für die Berechnung des aktualisierten Ergebnisses verwendet. Änderungen der Ausgangsdaten können dabei auf zwei Arten reduziert werden, zum einen das Hinzufügen neuer Daten und zum anderen das Löschen von Daten[5]. Diese Änderungen werden dann auf das veraltete Ergebnis, welches einer materialisierten Sicht entspricht, angewendet. Dies kann mittels zweier verschiedener Ansätze durchgeführt werden.
Basierend auf den Ideen und Ansätzen der inkrementellen Neuberechnung wird, im Rahmen der Arbeit, die inkrementelle Erweiterung der Schätzungen entwickelt.

3.2.1 Neuberechnungsansätze

Bei beiden Ansätzen werden die Effekte der Änderungen aggregiert und anschließend wird das Ergebnis basierend auf diesen Effekten aktualisiert. Diese Vorgehensweise geht auf den *Summary Delta Algorithmus*[16] zurück.

Overwrite-Ansatz

Beim Overwrite-Ansatz gibt es zwei Arten von Eingabedaten, zum einen das veraltete Ergebnis und zum anderen die Änderungen der Eingabedaten. Diese Änderungen, *Deltas*, werden dann in Inkrementierungen bzw. Dekrementierungen umgewandelt. Diese werden dann mit dem alten Ergebnis verrechnet. Das daraus entstandene aktualisierte Ergebnis wird dann gespeichert.

IncDec-Ansatz

Bei diesem Ansatz bilden, wie beim Overwrite-Ansatz, die *Deltas* die Eingabedaten, aber das veraltete Ergebnis wird nicht gelesen. Für die Durchführung

[5]Das Ändern von Daten kann man durch das Löschen des alten Wertes und das Hinzufügen eines neuen Wertes darstellen.

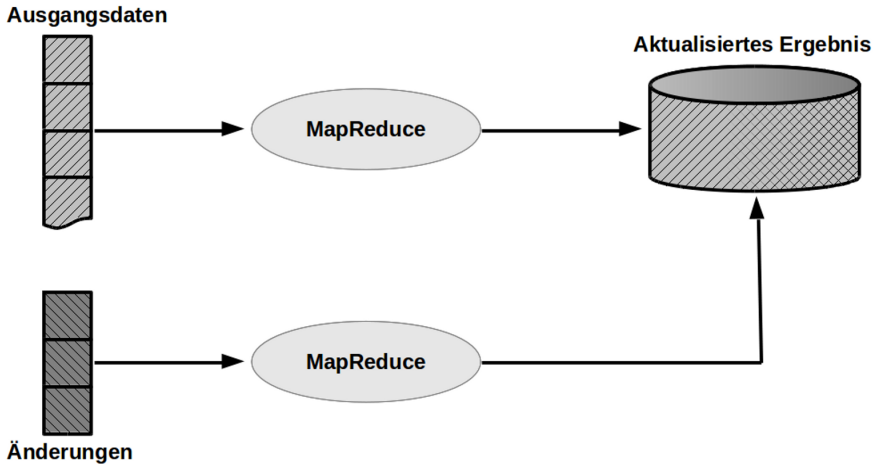


Abbildung 3.3: Ablauf einer normalen MapReduce-Berechnung mit anschließender IncDec-Berechnung.

dieses Ansatzes ist ein Ausgabeformat nötig, das Änderungen an gespeicherten Daten zulässt. Eine Möglichkeit dafür ist das HBase[2]-Ausgabeformat. Die Änderungen werden auch wie beim Overwrite-Ansatz in Inkremente bzw. Dekremente umgewandelt. Diese berechneten Änderungen werden dann direkt beim Speichern auf das veraltete Ergebnis angewendet. Das Ergebnis wird somit direkt aktualisiert. Aufgrund dieser Vorgehensweise kann der IncDec-Ansatz nicht für Ergebnisse in HDFS angewendet werden.

3.2.2 Ergebnis

Bei einer Evaluierung[14] anhand von MapReduce *use cases*[6] wurde eine Performanzsteigerung bei Änderungen bis zu 50% festgestellt. Bei größeren Änderungen war eine Neuberechnung effizienter. Außerdem hatte die Verteilung der Eingabedaten einen Einfluss auf die Performanz abhängig vom jeweiligen Berechnungsansatz.

3.2.3 Fazit

Das Verfahren zur inkrementellen Neuberechnung beschäftigt sich mit der Aktualisierung von Ergebnissen, basierend auf Änderungen der Eingabedaten. Dieses iterative Vorgehen bildet die Grundlage für die inkrementelle Erweiterung dieser Arbeit. Dafür wird ein angepasster Neuberechnungsansatz verwendet um eine Aktualisierung, basierend auf einer Vergrößerung der Stichprobe durch eine zusätzliche Stichprobenberechnung, durchzuführen.

Kapitel 4

Sampling für Hadoop

Um den Herausforderungen von großen und schnell wachsenden Datenmengen gerecht zu werden, ist eine Optimierung der Laufzeit von MapReduce wünschenswert. Im Folgenden wird deshalb ein Konzept zur Durchführung einer Stichprobenauswahl mit anschließender Hochrechnung vorgestellt. Durch die Berechnung einer Teilmenge und einer anschließenden Extrapolation des Ergebnisses kann man einen Zeitgewinn erreichen. Man muss aber beachten, dass man sich diesen Zeitvorteil aber auf Kosten der Genauigkeit erkauft.
Des weiteren soll dieses Konzept für Nutzer von Hadoop intuitiv benutzbar sein, indem die Stichprobenziehung und die Hochrechnung automatisiert und von Map- und Reduce-Funktion größtmöglich unabhängig gemacht werden. Dadurch sollen Berechnungen nicht an die Tatsache angepasst werden müssen, ob sie für eine komplette Berechnung oder nur für eine Stichprobe angewendet werden.

4.1 Ausgangssituation

Sampling ist nicht für alle Eingabedaten und Berechnungen geeignet oder sinnvoll. Bei Berechnungen der Graphentheorie kann man zum Beispiel mit einer Teilmenge der Graphen in der Regel nicht auf das Gesamtergebnis schließen. Auch bei einer einfache Suche nach einem bestimmten Wort ist eine Ergebnis einer Stichprobe nur bedingt aussagekräftig[1]. Bei arithmetischen Funktionen funktioniert dieser Ansatz besser, da man hier in der Regel abhängig von den Operanden eine Extrapolation durchführen kann. Spezialfälle wie die Berechnung des Mittelwertes, welche die Anzahl der verarbeiteten Elemente in die Berechnung einbeziehen, benötigen keine zusätzliche Abschätzung, hier kann man bereits das Ergebnis der Stichprobe als geschätztes Gesamtergebnis verwenden.
Trotz dieser Einschränkungen ist Sampling ein sinnvoller Ansatz um die Laufzeit zu optimieren, da in manchen Situationen ein schnelles ungefähres Ergebnis hilfreicher ist als ein genaues Ergebnis, auf das man eine längere Zeit warten muss.

[1] Findet man das Wort, weiß man, dass es vorkommt, findet man es jedoch nicht, weiß man nur, dass es in der Stichprobe nicht vorkommt, man kann keine Aussage über die Gesamtmenge treffen.

Um Sampling in Hadoop einzusetzen, ist aber bisher ein gewisser Mehraufwand für den Benutzer die Folge, da es nur begrenzte Möglichkeiten gibt, um Sampling in eine Hadoop-Berechnung zu integrieren. So kann ein Benutzer zum Beispiel im Vorfeld eine zufällige Stichprobeziehung durchführen und diese dann als Eingabedaten seiner Berechnung verwenden. Dies kann jedoch jedoch abhängig von der Art der Eingabedaten äußerst umständlich oder zeitraubend sein. Am Beispiel des Wordcount-Algorithmus' müsste man - je nach Granularität der Auswahl - im schlimmsten Fall[2] den kompletten Text durchgehen, einzelne zufällig ausgewählte Wörter extrahieren und diese dann zu einer neuen Eingabe zusammenführen, was für große Texte einen erheblichen Aufwand darstellt. Eine weiter Möglichkeit ist es, die jeweilige Map-Funktion dementsprechend anzupassen, dass diese die Stichprobenziehung zusätzlich durchführt. Dabei kann man die Stichprobenziehung vor der Berechnung der eigentlichen Map-Funktion, siehe Algorithmus 1, oder nach der der Berechnung durchführen, siehe Algorithmus 2. Wie in Abschnitt 3.1.2 zu sehen, wird dies auch bei EARL[15] verwendet. Bei diesen beiden Ansätzen werden aber zuerst alle Daten geladen und beim zweiten Ansatz zusätzlich auf alle Daten die Map-Berechnung angewendet. Dies hat einen negativen Effekt bezüglich dem Optimierung der Laufzeit.

Algorithm 1 premap

```
procedure MAP(key, value)
    if inZufallsStichprobe(p) then                ▷ Wahrscheinlichkeit p
        result ← realMap(key, value);             ▷ Eigentliche Berechnung
        write(result);
    else
        skip(key, value);
    end if
end procedure
```

Algorithm 2 postmap

```
procedure MAP(key, value)
    result ← realMap(key, value);                 ▷ Eigentliche Berechnung
    if inZufallsStichprobe(p) then                ▷ Wahrscheinlichkeit p
        write(result);
    else
        skip(result);
    end if
end procedure
```

Am Beispiel einer Wordcount-Berechnung mit einer Stichprobe von 10% müsste man die Ergebnisse[3] jeweils um den Faktor zehn[4] erhöhen, um eine Abschätzung des Gesamtergebnisses zu erhalten. Diese Korrektur kann man entweder auf die Ausgabe der Reduce-Funktion anwenden, siehe Algorithmus

[2] Eine Stichprobenziehung aus der Gesamtmenge der Worte eines Textes.
[3] Pro Wort wird eine Summe gebildet und ausgegeben.
[4] $\frac{UmfangEingabedaten}{\frac{1}{10}UmfangEingabedaten}$

3, oder auf die Ausgabe der Map-Funktion, letzteres würde eine Anpassung der Reduce-Funktion überflüssig machen. Da es aber in der Regel mehr Zwischen-

Algorithm 3 Korrektur in Reduce-Funktion

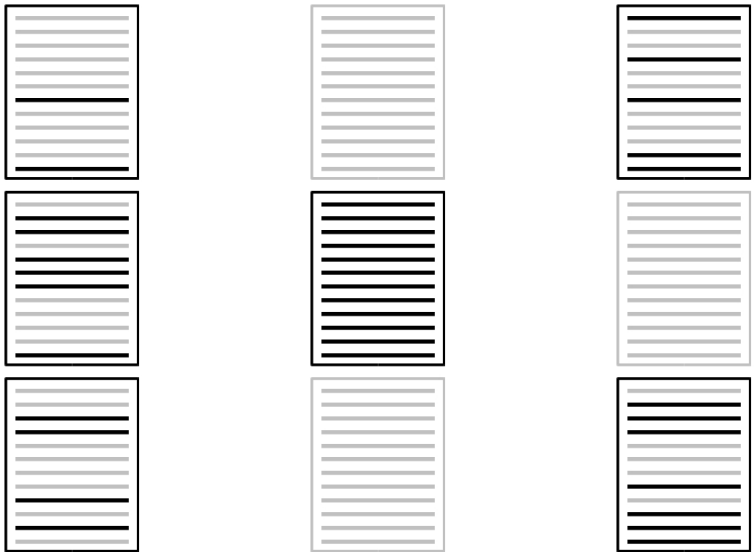
```
procedure REDUCE(key, values)
    result ← realReduce(key, values);        ▷ Eigentliche Berechnung
    correctedResult ← result * factor;       ▷ Korrektur
    write(correctedResult);
end procedure
```

ergebnisse bzw. Map-Ausgaben gibt als Endergebnisse bzw. Reduce-Ausgaben müsste diese Berechnung öfter ausgeführt werden, was bei komplexeren Korrekturen sich in der Laufzeit niederschlagen kann.
Komplexere Korrekturen können zustande kommen, da sie abhängig von der jeweiligen Reduce-Funktion sind, bei Wordcount ist diese Reduce-Funktion eine einfache Summe und die Korrektur nur eine Multiplikation eines Faktors[5]. Benutzer mit komplexeren Reduce-Funktionen müssen dann auch selbst die jeweilige Korrektur herleiten und ihre jeweiligen Funktionen dahingehend anpassen. Deshalb wird im Folgenden vorgestellt wie man diesen Mehraufwand dem Benutzer abnehmen kann.

4.2 Stichprobenziehung

Das Ziel ist es, die Ziehung der Stichprobe zu vereinfachen bzw. zu automatisieren. Am naheliegendsten ist hierbei, ein spezielles HDFS-Eingabeformat zu entwickeln. Als Grundlage für die Granularität der Stichprobenziehung dient dabei die Art, wie die Daten in HDFS vorliegen. Die Daten liegen zeilenweise vor, wobei diese zusätzlich zu Gruppen, *Splits*, zusammengefasst werden. Aufgrund dieser Repräsentation bietet es sich an die Ziehung der Stichprobe auf der Menge der Zeilen oder der Splits durchzuführen.
Eine zeilenbasierter Ansatz würde einer einfachen Zufallsauswahl, siehe Abschnitt 2.3.1, entsprechen. Bei diesem Auswahlverfahren ist eine potentiell gleichmäßige Abdeckung ein Vorteil, da man die Auswahl auf der kleinsten Einheiten[6] ausführt. Da aber über jede Zeile iteriert werden muss, sowie pro Zeile eine Entscheidung getroffen wird, ist dieses Auswahlverfahren zeitintensiver als das nächste Verfahren, eine splitbasierte Auswahl. Eine Stichprobenziehung auf Splitebene entspricht einer Klumpenstichprobe (Abschnitt 2.3.1), bei der die Klumpen durch die Splits repräsentiert werden. Da hier die Auswahl pro Split getroffen wird ist die Laufzeit geringer als bei der Zeilenauswahl, aber die Abdeckung kann, abhängig von der Verteilung der Daten, ungenauer sein, siehe Beispiel im folgenden Abschnitt, was für die Abschätzung des Gesamtergebnisses von Nachteil ist, da die Stichprobe möglichst repräsentativ sein sollte.
Angelehnt an mehrstufige Auswahlverfahren, im speziellen der zweistufigen Klumpenstichprobe, kann man die beiden vorgestellten Auswahlverfahren auch kombinieren. Dabei kann der Benutzer durch Ändern der Wahrscheinlichkeiten

[5] $\frac{1}{p}$, wobei p der Wahrscheinlichkeit der Auswahl der Elemente für die Stichprobe entspricht.
[6] Für eine Auswahl auf der Wortebene müsste man zusätzliche Berechnungen durchführen.



(a) Zeilenbasiert (b) Splitbasiert (c) Kombination

Abbildung 4.1: Die drei Auswahlansätze im Vergleich.

für die Zeilenauswahl bzw. die Splitauswahl die jeweiligen Vor- bzw. Nachteile an die Eingabedaten und Berechnung anpassen.

Beispiel für eine potentielle Verfälschung

Als Beispiel wird eine Wordcount-Berechnung über eine Anzahl von Texten über die Flora und Fauna verschiedener Regionen verwendet. Zur Vereinfachung wird davon ausgegangen, dass jeder Split genau einen Text enthält.
In der Menge der Eingabedaten gibt es nun einen Text/Split, der sich mit der Region des Kongo befasst, sowie einen Text/Split, der sich mit der Region Indonesien befasst. Wird nun eine Stichprobe anhand der Splits durch geführt kann es vorkommen, dass der Text über den Kongo ausgewählt wird und der Text über Indonesien nicht. Bei der Abschätzung des Gesamtergebnisses ist jetzt der komplette Text über den Kongo enthalten, wodurch alle Begriffe, die nur mit dieser Region im Zusammenhang stehen in der Stichprobe enthalten sind. Diese Begriffe kommen in dem geschätzten Ergebnis nun häufiger vor als in dem wirklichen Ergebnis. Wohingegen Begriffe die typisch für Indonesien sind, wie zum Beispiel Komodowaran, überhaupt nicht im geschätzten Ergebnis vorkommen. Dadurch, dass in diesem Beispiel Begriffe, die miteinander im Zusammenhang stehen lokal gruppiert sind, entsteht eine Verfälschung. Mit einer Stichprobenziehung basierend auf den Zeilen kann man eine solche Verfälschung reduzieren, falls die Eingabedaten solche lokale Abhängigkeiten aufweisen.

4.3 Korrekturfunktion

Die Abschätzung des Gesamtergebnisses basierend auf dem berechneten Stichprobenergebnis ist der zweite Teil, der Anpassungen der Funktionen seitens der Nutzer erfordert. Da die Korrekturfunktion abhängig von der benutzerdefinierten Reduce-Funktion ist, wäre eine Möglichkeit eine dritte benutzerdefinierte Phase zu integrieren. In einer solchen Korrektur-Phase könnte der Benutzer ein entsprechende Korrekturfunktion implementieren, die auf die Reduce-Ausgaben angewendet wird. Ein generischer Ansatz ohne Einbindung des Benutzers ist aber erstrebenswerter. Die Abhängigkeit von Korrekturfunktion und Reduce-Funktion erschwert eine solche generische Lösung. Aus diesem Grund wurden mehrere Aggregatfunktionen[7] betrachtet, um eine mögliche Klassierung zu erhalten, die einheitliche Korrekturen erlaubt. Die betrachteten Aggregatfunktionen waren algebraische, distributive und holistische Funtkionen. Bei näherer Betrachtung konnte zwischen zwei Gruppen von Funktionen unterschieden werden. Zum einen Funktionen, bei denen eine Korrektur durchgeführt werden muss, wie zum Beispiel bei der Summe oder dem Produkt. Sowie Funktionen bei denen eine Korrektur unnötig oder nicht möglich ist, wie zum Beispiel bei der Berechnung des Mittelwerts oder des Minimums. Bei Funktionen wie Min, Max, Top-K, etc. ist das Ergebnis ein einzelnes bzw. mehrere Elemente der Stichprobe, da aber keine Informationen über die Elemente der restlichen Menge vorliegen besteht auch keine Optimierungsmöglichkeit. Dabei fiel auch auf das alle holistischen Funktionen zu dieser Gruppen von Funktionen gehörten, dies ist darauf zurückzuführen, dass bei holistischen Funktionen zur Berechnung in der Regel alle Daten benötigt werden. So kann man zwar eine solche Funktion auf einer Stichprobe berechnen, aber man kann keine Aussage über die Gesamtmenge treffen. Im Gegensatz dazu, handelt es sich bei der AVG-Funktion, welche den Mittelwert berechnet, um eine Funktion die sich aus zwei Summenfunktionen(SUM und COUNT[8]) zusammensetzt: $AVG = \frac{SUM}{COUNT}$. Da SUM und COUNT die gleiche Korrekturfunktion nach sich ziehen, kürzen sie sich in der Berechnung der Korrektur für AVG gegenseitig. Dies kann auch für andere Funktionen wie etwa die Berechnung der Varianz gelten. Zum besseren Verständnis der Korrektur wird nun die erste Gruppe genauer betrachtet anhand der Beispiele SUM und der Multiplikationsfunktion PRODUCT.

4.3.1 Beispiele: SUM und PRODUCT

Im Folgenden steht M für die Gesamtmenge der Eingabedaten und X für die ausgewählte Stichprobe.

$SUM(M) = \sum_{m_i \in M} m_i$ ist somit die Berechnung des Gesamtergebnisses und

$SUM(X) = \sum_{x_i \in X \subset M} x_i$ die Berechnung des Teilergebnisses.

[7] Summe, Anzahl, Produkt, Mittelwert, Minimum, Maximum, Top-K, Standardabweichung, Varianz, Median, Rang

[8] COUNT ist eine spezielle Version der SUM-Funktion mit immer dem gleichen Summanden: 1

Bei einer Abschätzung würde man das Teilergebnis mit dem Faktor $\frac{|M|}{|X|}$ multiplizieren, unter der Annahme, dass X repräsentativ für M ist. Der Faktor basiert auf dem Größenunterschied der beiden Mengen, sollte zum Beispiel der Stichprobenumfang 25% betragen, so wäre der Faktor 4, da $\frac{|M|}{\frac{1}{4}|M|}$ gilt. Somit wäre die Abschätzung viermal die Summe der Stichprobe.

$$\left(\sum_{x_i \in X} x_i\right) * 4 = \left(\sum_{x_i \in X} x_i\right) + \cdots + \left(\sum_{x_i \in X} x_i\right)$$

Analog würde eine Korrektur der PRODUCT-Funktion funktionieren, mit der Ausnahme, dass das Stichprobenergebnis nicht viermal addiert wird sondern multipliziert, was zu folgender Korrektur führt:

$$\left(\prod_{x_i \in X} x_i\right) * \cdots * \left(\prod_{x_i \in X} x_i\right) = \left(\prod_{x_i \in X} x_i\right)^4$$

Hierbei sieht man wieso unterschiedliche Korrekturfunktionen für die unterschiedlichen Funktionen verwendet werden müssen.

4.3.2 Problem

Wie bereits bei der AVG-Funktion, können sich potentielle Funktionen aus anderen Funktionen zusammensetzen. Dabei kann man die Korrekturfunktion aus den jeweiligen Teilkorrekturfunktionen bilden. Eine generische Korrektur basierend auf den Stichprobenergebnissen benötigt dabei das Wissen aus welchen Funktionen die Reduce-Funktion wie gebildet wird. Das würde eine Analyse der Reduce-Funktion oder eine manuelle Eingabe erfordern. Zusätzlich müsste für jede potentielle Basisfunktion eine Korrekturfunktion vorliegen. Eine Analyse komplexer Reduce-Funktion würde sich negativ auf die Laufzeit auswirken und bei einer manuellen Eingabe müsste der Benutzer über die verschiedenen Funktionstypen informiert sein und diese auch korrekt zuordnen können. Da das Ziel aber eine Optimierung der Laufzeit, mit nur geringer oder keiner Mehrbelastung für den Benutzer, ist, wird ein anderer Korrekturansatz verwendet.

4.3.3 Lösung

Bei der Betrachtung der Korrekturfunktionen kann man Gemeinsamkeiten erkennen:

$$\left(\sum_{x_i \in X} x_i\right) * n = \left(\sum_{x_i \in X} x_i\right) + \cdots + \left(\sum_{x_i \in X} x_i\right) = (x_1 * n) + ... + \left(x_{|X|} * n\right)$$

$$\left(\prod_{x_i \in X} x_i\right)^n = \left(\prod_{x_i \in X} x_i\right) * \cdots * \left(\prod_{x_i \in X} x_i\right) = (x_1 * n) * ... * \left(x_{|X|} * n\right)$$

Die Korrekturfunktion bildet die Differenzmenge M\X durch X ab. Abhängig vom Größenunterschied wird die Stichprobenmenge X dabei $\frac{|M|}{|X|}$ herangezogen. Diese Gemeinsamkeit kann für eine generische Korrektur verwendet werden. Dabei ist der Ansatzpunkt der Korrektur nicht das Endergebnis, sondern die

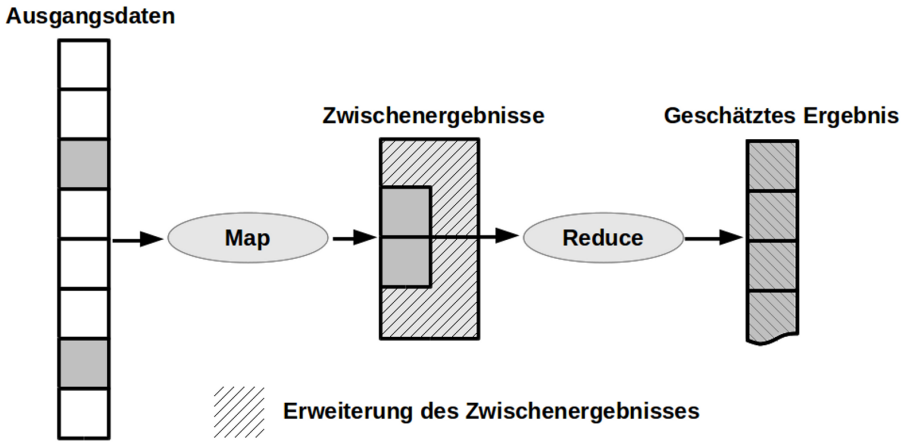


Abbildung 4.2: Samplingidee basierend auf der Vervielfachung der Zwischenergebnisse.

Ergebnisse der Map-Funktion, diese Zwischenergebnisse dienen der Reduce-Funktion als Eingabe. Indem man nun diese abhängig von dem Faktor $\frac{|M|}{|X|}$ vervielfacht, erhält man eine Korrektur unabhängig von der Reduce-Funktion. Dies ist möglich, da die Korrektur angewendet wird bevor die Operanden der Reduce-Funktion auf die Zwischenergebnisse angewendet werden. Bei der Vervielfältigung der Zwischenergebnisse kann man nur ganzzahlige Faktoren verwenden, da man kein Element 1,5-mal vervielfältigen kann. Dies ist ein Problem, da es je nach Prozentsatz der Stichprobe ein Abweichung von bis zu 32% zur Folge haben kann. Dies errechnet sich, wenn man den Faktor rundet. Bei einem Prozentsatz von 66% wäre der Faktor ungefähr 1,52, das wäre gerundet 2. Aber bei einem Faktor von 2 wäre das geschätzte Ergebnis 132% des eigentlichen Ergebnisses[9]. Eine Abweichung von bis zu einem Drittel ist nicht tolerabel. Um diesen Umstand auszugleichen, werden einzelne Elemente zufällig ein zusätzliches Mal ausgegeben. So würde bei einem Faktor von 2,5 jedes Element mindestens zweimal ausgegeben und mit einer Wahrscheinlichkeit von 0,5 ein drittes Mal. Dieser Ansatz ist nicht perfekt, da einzelne Elemente stärker gewichtet werden als andere, aber bei großen Datenmengen sind diese potentiellen Diskrepanzen vernachlässigbar, insbesondere im Vergleich mit einem sicheren Fehler von bis zu einem Drittel ohne diese Anpassung.
Da diese Korrektur auf alle Zwischenergebnisse angewendet werden muss, ist sie zeitintensiver als eine Korrektur am Ende der Berechnung, da es in der Regel mehr Zwischenergebnisse als Endergebnisse gibt. Der generische Aspekt wiegt aber diesen geringen Zeitverlust auf.
Für Funktionen wie die Berechnung des Minimums, des Mittelwerts, etc. ist eine solche bzw. keine Korrektur nötig, in diesen Fällen kann man auf die Korrektur zugunsten der Laufzeit verzichten. Sollte eine Korrektur trotzdem durchgeführt werden würde sich das Ergebnis der Stichprobe nicht ändern, es würde als auch kein Fehler entstehen. Bei Funktionen wie Top-K zum Beispiel wäre eine Verfälschung möglich, falls die Funktion nicht verlangt, dass die Werte

[9]Das Ergebnis der 66%igen Stichprobe würde verdoppelt, was 132% ergibt.

voneinander verschieden sein müssen[10]. Für solche Funktionen dürfte man die Korrektur nicht anwenden, da sie das Ergebnis verfälschen könnte.

[10] Wäre $k = \frac{|M|}{|X|}$ würde das Maximum k-mal aufgelistet werden.

Kapitel 5

Inkrementelle Erweiterung

In Kapitel 3.2 wurde bereits eine inkrementelle Variante[14] für MapReduce vorgestellt. Solch ein Ansatz ist auch geeignet, um die Genauigkeit bereits berechneter Schätzungen zu erhöhen. Anstatt eine neue Berechnung mit einem höheren Stichprobenumfang durchzuführen, kann man eine zusätzliche kleine Stichprobe berechnen und deren Ausgabe mit der bereits vorhandenen Schätzung vereinen. So erhält man eine Schätzung basierend auf einer größeren Stichprobe. Hierbei ist zu beachten, dass diese Stichprobe auf einer Ziehung mit Zurücklegen beruht. Die Anzahl inkrementeller Iterationen bildet dabei die Obergrenze wie oft ein Element mehrmals gezogen werden kann, da die einzelnen Unterstichproben jeweils eine Ziehung ohne Zurücklegen verwenden. Da die einzelnen Stichproben unabhängig voneinander sind, können sie auf einer unterschiedlich großen Teilmenge der Eingabedaten basieren. Damit keine Verfälschung der aktualisierten Schätzung entsteht, muss eine Gewichtung vorgenommen werden. Aktualisierung und Gewichtung sollen dabei automatisch geschehen, ohne dass der Benutzer dies manuell in seinen Funktionen durchführen muss.

5.1 Inkrementeller Sampling-Job

Bei einer inkrementellen Berechnung einer Schätzung benötigt man zum einen die zu verfeinernde Schätzung und zum anderen eine zusätzliche Stichprobe. Um die bereits vorhandene Schätzung verwenden zu können, wird eine benutzerdefiniert Map-Funktion benötigt. Deren Aufgabe ist es, die Ergebnisse in Schlüssel/Wert-Paare umzuwandeln und die darauf durchgeführte Korrektur rückgängig zu machen. Dies ist nötig, da zum einen die Ausgabe abhängig von der benutzerdefinierten Reduce-Funktion ist und somit unterschiedliche Darstellungsformen der Daten möglich sind. Zum anderen wurde eine Korrektur aufgrund der Größe der Teilmenge der Eingabedaten berechnet, welche sich durch den inkrementellen Ansatz ändert. Der Nachteil hierbei ist, dass der Benutzer dazu in der Lage sein muss, die Korrektur zurück zu rechnen, was je nach Komplexität der Reduce-Funktion nicht trivial ist. Zusätzlich muss der Nutzer auch wissen, wie die Korrektur durchgeführt wird. Da die Zwischenergebnisse, die auf der bereits berechneten Schätzung basieren, jeweils mehrere Zwischenergebnisse einer normalen Berechnung repräsentieren, kann ein Gewichtungsfehler bei nicht ganzzahligen Faktoren durch die Korrektur entstehen.

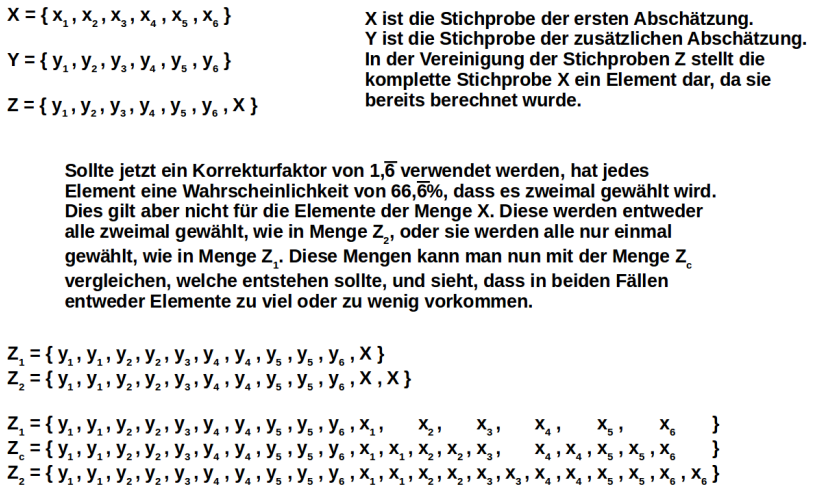


Abbildung 5.1: Durch die Vervielfältigung der Zwischenergebnisse entsteht ein Fehler.

Dieses Ungleichgewicht einzelner Element fällt bei einer großen Anzahl an Elementen nicht so sehr ins Gewicht, aber da die Zwischenergebnisse selbst viele Elemente repräsentieren, kann dies zu einer Verzerrung der Schätzung führen. Dies ist möglich, da das Zwischenergebnis einer bereits berechneten Schätzung als ein Element gewertet wird und nicht die Subelemente einzeln. Dies kann man am Beispiel in Abbildung 5.1 sehen. Bei der wiederholten Verwendung der inkrementellen Berechnung kann sich ein solcher Fehler auch verschleppen und aufsummieren.
Um ein solches Fehlerwachstum nicht zu begünstigen und dem Benutzer die Verwendung zu erleichtern wird ein inkrementelles Sampling-System bestehend aus zwei MapReduce-Berechnungen verwendet. Dazu werden zuerst die potentiellen Neuberechnungsansätze im Kontext einer inkrementellen Schätzung betrachtet.

5.2 Neuberechnungsansätze

Hier werden die beiden Neuberechnungsansätze IncDec und Overwrite aus Kapitel 3.2[14] im Bezug auf eine Aktualisierung von Schätzungen betrachtet.

5.2.1 IncDec-Ansatz

Beim diesem inkrementellen Neuberechnungsansatz werden die Änderungen, die durch die Berechnung neuer Daten entstehen, auf ein bereits berechnetes Ergebnis angewandt. Dieser Ansatz ist vor allem bei bei einer kleinen Menge an Änderungen vorteilhaft, da im Vergleich zur Overwrite-Technik, wirklich nur die Datensätze geändert bzw. geschrieben werden müssen die betroffen sind. Man benötigt zwar für jede solche Änderung ein Random-Read, aber man spart einen Schreibvorgang für jeden nicht geänderten Datensatz. Wenn man diesen Ansatz auf die Aktualisierung von Sampling-Ergebnissen überträgt, würde das Ergebnis einer zusätzlichen Schätzung verwendet werden, um bei einer bereits

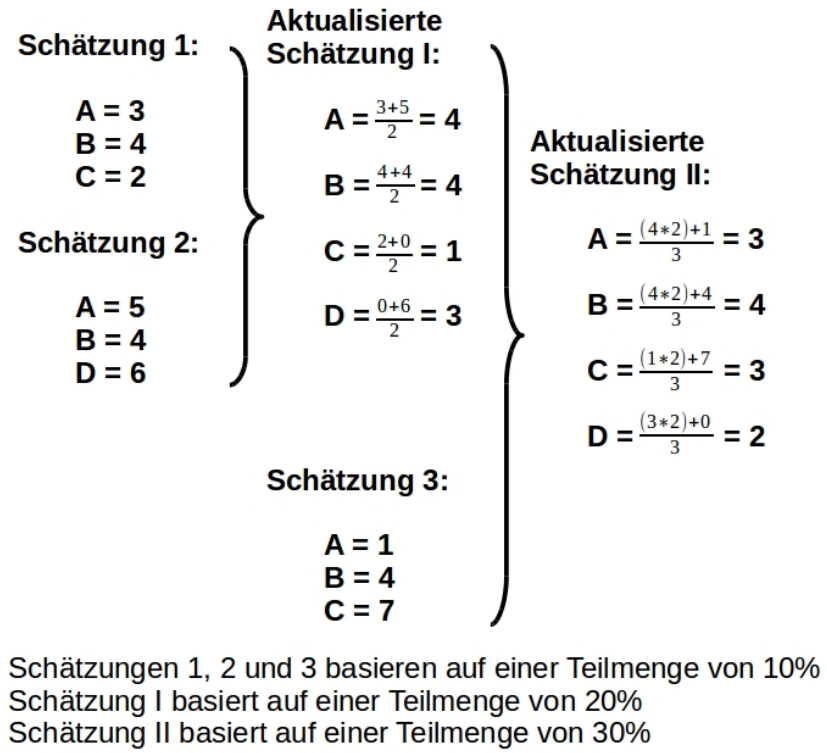


Abbildung 5.2: Beispiel einer gewichteten Aktualisierung.

vorhandenen Schätzung die Genauigkeit zu erhöhen.
Das Problem dieses Ansatzes bei dieser Verwendung ist aber, dass sich das aktualisierte Ergebnis aus den beiden Schätzungen berechnet. Um die Änderungen für die zu aktualisierende Schätzung zu erhalten, wird diese als zusätzliche Eingabe benötigt. Dies ist bei der Verwendung im inkrementellen MapReduce[14] nicht nötig, da die Eingabedaten der inkrementellen Berechnung bereits Änderungen darstellen.

Am Beispiel in Abb. 5.2 sieht man, dass ein Ergebnis nur beibehalten wird, wenn der Wert in beiden Schätzungen gleich ist. Man kann auch sehen, dass eine Gewichtung der Werte verwendet wird. Dies ist nötig, da sich das Ergebnis des ersten Iterationsschritts aus einer doppelt so großen Teilmenge der Ausgangsdaten zusammensetzt als die dritte Schätzung. Im Generellen ist immer eine Gewichtung der Schätzungen vorzunehmen, wenn Ergebnisse vereint werden die auf unterschiedlich großen Teilmengen basieren.
Aufgrund der hier aufgeführten Eigenschaften des IncDec-Ansatzes im Bezug auf das Sampling und die Tatsache, dass der hier vorgestellte Sampling-Ansatz ausschließlich HDFS als Speichersystem nutzt, wird der IncDec-Ansatz nicht verwendet.

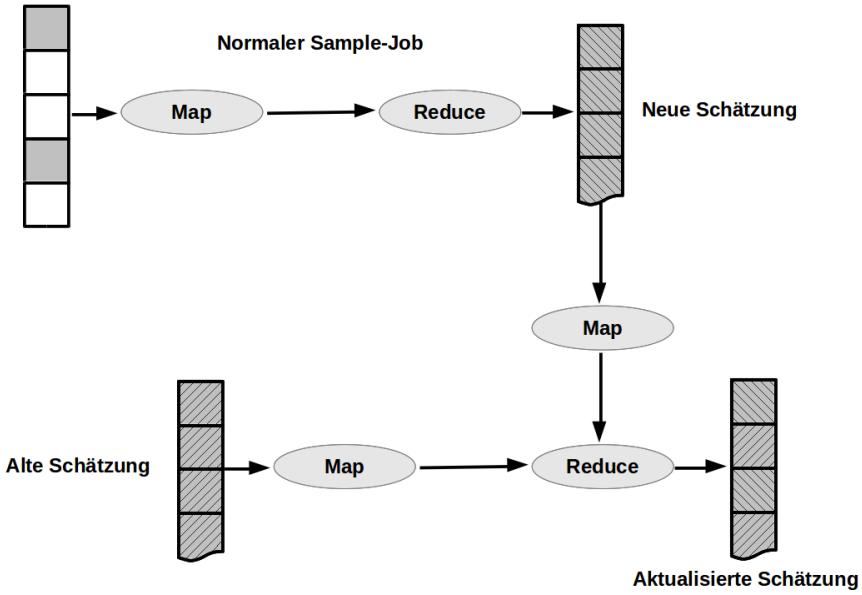


Abbildung 5.3: Inkrementelles Sampling bestehend aus zwei Berechnungen.

5.2.2 Overwrite-Ansatz

Beim Overwrite-Ansatz dienen die beiden Schätzungen als Eingabedaten, diese werden dann verrechnet und bilden das neue Ergebnis. Dieses wird dann komplett neu geschrieben. Hier liegt der Vorteil bei großen Änderungsmengen. Die Tatsache, dass beide Schätzungen die Eingabe bilden, ist weiterhin von Vorteil für die Berechnung der Aktualisierung. Und da nur Daten gelesen und geschrieben werden und die Änderungen intern im Verlauf der Berechnung der neuen Ausgabe angewendet werden, ist dieser Ansatz auch geeignet für die Berechnung in HDFS. Diese Vorteile führen dazu, dass Overwrite als Neuberechnungsansatz für die inkrementelle Erweiterung verwendet wird.

5.3 Umsetzung einer zweistufigen Berechnung

Die inkrementelle Berechnung wird hierbei in zwei Schritte aufgeteilt. Im ersten Schritt wird eine zusätzliche Schätzung berechnet und im zweiten Schritt wird diese mit einer bereits berechneten Schätzung vereint, um ein genaueres Ergebnis zu erhalten. Der zweite Job ist somit der inkrementelle Teil. Der Benutzer ist nur für das Lesen der Eingabe, deren Umwandlung in Schlüssel/Wert-Paare und das Schreiben der Ausgabe verantwortlich. Die ersten beiden Aufgaben werden mittels spezieller Map-Funktionen durchgeführt, die jeweils auf den beiden Schätzungen ausgeführt werden. Für die Ausgabe wird dann eine Reduce-Funktion verwendet, in der nur noch entschieden werden muss, wie die Daten dargestellt werden sollen.
Die Gewichtung der Zwischenergebnisse übernimmt die Map-Funktion. Diese Gewichtung kann man einfach durch eine mehrfache Ausgabe der einzelnen Schlüssel/Wert-Paare erreichen. So würden zum Beispiel die Paare einer auf 20%

basierenden Schätzung zweimal ausgegeben, wenn sie mit einer auf 10% basierenden Schätzung vereint werden würde. Dies würde aber abhängig vom Umfang der Zwischenergebnisse und dem Replizierungsfaktor zu einem enormen Kommunikationsoverhead führen. Mit dem Einführen spezieller Zwischenergebnisse kann man dies jedoch umgehen. Dafür müssen diese eine interne Gewichtung enthalten. So kann man ohne die Werte mehrmals auszugeben eine Gewichtung bei der Berechnung erreichen.
Diese Werte erhält nun ein spezieller Reducer. In dessen Reduce-Funktion wird nun basierend auf den Werten und ihrer jeweiligen Gewichtung ein Durchschnitt gebildet, das neue Ergebnis. Dieses Ergebnis wird mit dem zugehörigen Schlüssel an die benutzerdefinierte Reduce-Funktion übergeben. In diesem Aufruf muss das bereits berechnete Ergebnis nur noch in die verlangte Repräsentation überführt und anschließend ausgegeben werden.
Diese automatische Aktualisierung funktioniert nur bei berechneten Ergebnissen. Bei solchen, in denen Elemente oder ähnliches das Ergebnis bilden, ist allerdings der Durchschnitt nicht das verlangte Resultat. Dies betrifft Funktionen wie die Berechnung des Maximums o.ä.. $Max(12_3, 7_2) = 12$ (12 dreifach gewichtet und 7 zweifach) und nicht 10 wie es automatisch berechnet werden würde[1]. Aus diesem Grund ist die Aktualisierung optional, wenn sie nicht verwendet wird, erhält die benutzerdefinierte Reduce-Funktion alle gewichteten Werte und es kann eine manuelle Aktualisierung vorgenommen werden.

[1] $\frac{12+12+12+7+7}{5} = 10$

Kapitel 6

Implementierung

Die Komponenten wurden für Hadoop[1] in Java implementiert. Zuerst werden die einzelnen Komponenten aufgeführt, die für das Durchführen einer Stichprobenziehung und einer Hochrechnung benötigt werden. Diese bilden zusammen die Grundlage für die `SamplingJob`-Klasse. Danach werden die Klassen für die inkrementelle Berechnung aufgeführt. Diese werden dann von der `EstimateJob`-Klasse verwendet, welche zusammen mit dem `SamplingJob` die Grundlagen der `SamplingEstimationComputation` sind.
Die implementierten Klassen sind als generische Klassen konzipiert, dadurch hat der Benutzer nur einen minimalen Mehraufwand bei der Verwendung im Vergleich zur Entwicklung einer kompletten Berechnung.

6.1 Eingabeformate

Zur Durchführung der Stichprobenauswahl wurden spezielle Eingabeformate implementiert. Diesen führen für einen per Parameter gegebenen Prozentsatz eine zufällige Stichprobenauswahl durch. Dabei kann zwischen drei Arten gewählt werden, eine Auswahl auf Zeilenebene, eine auf Splitebene und eine Kombination dieser beiden. Diese Eingabeformate ermöglichen es einem Benutzer eine Stichprobenziehung durchzuführen, nur durch die Verwendung eines der Eingabeformate und eines Parameters, zur Angabe des gewünschten Prozentsatzes.

6.1.1 RandomTextInputFormat

Beim `RandomTextInputFormat` wird die Zufallsauswahl auf den einzelnen Zeilen durchgeführt. Dafür wurde der `LineRecordReader` erweitert. Dieser neue `RandomRecordReader` führt beim Aufruf der `nextKeyValue()`-Methode eine Zufallsevaluierung durch, welche entscheidet, ob das jeweilige Schlüssel/Wert-Paar[1] gelesen oder übersprungen wird.

6.1.2 RandomSplitInputFormat

Dieses Eingabeformat führt die Auswahl anhand der gegebenen Splits durch. In der normalen `getSplits`-Methode des TextInputFormats werden die Ein-

[1] Zeilenoffset/Zeilentext

gabedaten in eine Liste von Splits überführt. In der Sampling-Variante wird die zusätzlich Auswahl getroffen ob die Splits in diese Liste aufgenommen oder verworfen werden sollen. Abhängig von Splitgröße kann dies eine signifikante Verringerung der Anzahl an Auswahlberechnungen im Vergleich zur Auswahl auf Zeilenebene sein.

6.1.3 RandomInputFormat

Diese Eingabeformat ist eine Kombination der vorherigen Eingabeformate. Zuerst führt es die Auswahl auf Splitebene durch wie im `RandomSplitInputFormat`. Anschließend wird für die einzelnen Splits der `RandomRecordReader` verwendet, welcher im `RandomTextInputFormat` für die Auswahl der Zeilen verantwortlich ist. Diese Kombination aus Zeilen- und Splitauswahl kann man mittels Parameter flexibel an Eingabedaten und Berechnungen anpassen. Dadurch, dass man beide Parameter unterschiedlich wählen kann, ist es möglich das `RandomTextInputFormat` und das `RandomSplitInputFormat` mittels des `RandomInputFormats` zu ersetzen, da man einen der Parameter auf 100% setzen kann und somit dann nur auf Split- bzw. Zeilenebene eine Auswahl durchführt. Zusätzlich ist eine Funktionalität integriert, um die Benutzung einer Kombination von Prozentsätzen zu vereinfachen. Es ist möglich einen Gesamtprozentsatz anzugeben, dieser wird dann gleichmäßig für die beiden Ebenen berechnet. Dabei wird zuerst überprüft, welche Parameter angegeben wurden, der Standardwert bei *keiner Eingabe* ist 100%. Die Eingabe bezüglich der Zeilenebene bzw. der Splitebene haben immer den Vorzug vor der Eingabe des gesamten Prozentsatz. Dieser Vorzug ist damit man eine mögliche Feineinstellung nicht aus Versehen durch die Hilfsfunktionalität stört.
In Algorithmus 4 sieht man wie die Berechnung des Prozentsatzes auf einer Ebene[2] erfolgt. Als Eingabe dienen hierbei die Eingabe von Gesamtprozentsatz, dem lokalen Prozentsatz[3] und dem externen Prozentsatz[4]. Das Ergebnis dieses Algorithmus' gibt an wie groß die Stichprobe in Prozent für die jeweilige Ebene sein muss. In den Zeilen 2 und 3 sieht man die bereits erwähnte *Bevorzugung* der spezifischen Eingabe der jeweiligen Ebene. Für den Fall, dass gar keine Eingabe gemacht wurde, wird standardmäßig eine Stichprobe von 1% durchgeführt indem auf Zeilen- sowie auf Splitebene jeweils eine 10% Auswahl getroffen wird, zu sehen in Zeile 7. In Zeile 13 tritt der Fall auf, dass nur ein totaler Prozentsatz angegeben wurde, ohne näher zu spezifizieren, wie dieser aufgeteilt werden soll. In diesem Fall wird der Prozentsatz gleichmäßig aufgeteilt. Die Verwendung der Quadratwurzel und einer kaufmännischen Rundung kann dazu führen, dass eine Abweichung zwischen angegebenem und berechnetem Prozentsatz entsteht. Solch eine Diskrepanz beläuft sich aber auf maximal 0,99% und die größten Werte treten hauptsächlich im oberen Prozentbereich auf. Die durchschnittliche Diskrepanz beträgt ca. 0,066%. In Zeile 16 tritt der Fall auf, dass ein spezifischer Prozentsatz für die andere Ebene angegeben wurde und ein Gesamtprozentsatz, hier wir nun der daraus resultierende Prozentsatz für die lokale Ebene berechnet, der benötigt wird, um auf den Gesamtprozentsatz zu kommen. In den Zeilen 9 und 18 tritt der Fall ein, dass keine Auswahl auf der aktuellen Ebene getroffen

[2] Split- oder Zeilenebene
[3] Wenn der Algorithmus auf Zeilenebene ausgeführt wird ist dies der angegebene Prozentsatz für die Zeilenauswahl.
[4] Der Prozentsatz für die andere Ebene der Auswahl

wird und somit einer Vollerhebung stattfindet.

Algorithm 4 Berechnung des Prozentsatzes

```
1: procedure PERCENT(totalPercent, localPercent, externalPercent)
2:     if localPercent ≠ 100 then
3:         return localPercent;
4:     else
5:         if totalPercent = 100 then
6:             if externalPercent = 100 then
7:                 return 10;                          ▷ Standardstichprobe 1%
8:             else
9:                 return 100;
10:            end if
11:        else
12:            if externalPercent = 100 then
13:                return round((sqrt(totalPercent/100)) ∗ 100);
14:            else
15:                if totalPercent < externalPercent then
16:                    return round((totalPercent/externalPercent) ∗ 100);
17:                else
18:                    return 100;
19:                end if
20:            end if
21:        end if
22:    end if
23: end procedure
```

Sollte dieser Algorithmus zum Beispiel im `RandomRecordReader`[5] mit den Werten 10(%), 100(%) und 50(%) aufgerufen werden, so würde sich der Prozentsatz der Zeilenauswahl in Zeile 16 berechnen. Das Ergebnis wäre dann $round(\frac{10}{50} * 100) = 20$. Für eine Eingabe von 81(%), 100(%) und 100(%) würde Zeile 13 das Ergebnis $round(\sqrt{\frac{81}{100}} * 100) = 90$. Und bei einer Eingabe von 34(%),12(%) und 7(%) würde Zeile 3 mit dem Ergebnis 12 aufgerufen.

Das `RandomInputFormat` hat auch eine zusätzliche Test-Funktionalität für kleine Eingabedaten. Durch das Setzen eines speziellen Parameter wird ein anderer `RecordReader` verwendet. Dieser `RandomExactRecordReader` iteriert in seiner Initialisierung einmal über alle Zeilen eines Splits und zählt diese, und anhand des gegebenen Prozentsatzes wird dann berechnet, wieviele Zeilen genau benötigt werden[6]. Bei dem Aufruf der `nextKeyValue()`-Methode wird dann solange über den Split iteriert, bis die erforderliche Anzahl an Zeilen erreicht ist. Dabei wird die Auswahl der Zeilen zufällig getroffen. Sollte das Ende des Splits erreicht werden und noch nicht genügend Zeilen ausgewählt worden sein, so wird der Split wieder von vorne angefangen. Die Zufallsauswahl, die so zustande kommt, ist von der Anzahl auch bei kleinen Eingabedaten genau, es handelt sich dabei dann aber um eine Ziehung mit Zurücklegen.

[5] Der für die Zeilenauswahl zuständig ist.

[6] Bei kleinen Eingabedaten kann man nicht auf das Gesetz der großen Zahlen[9] zurückgreifen und eine Zufallsauswahl wird ungenau.

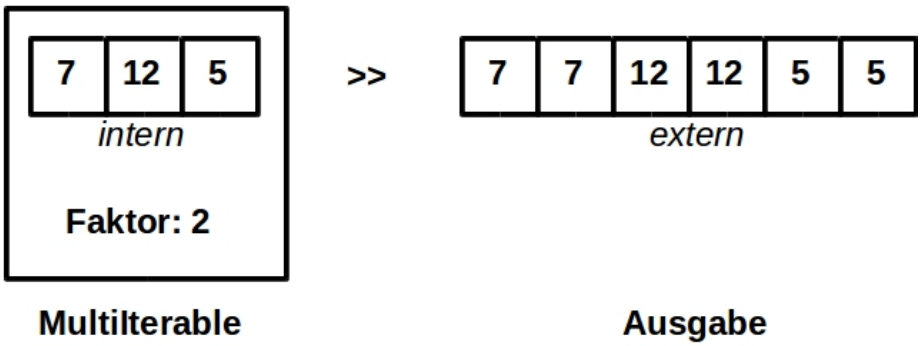


Abbildung 6.1: Intern gespeicherte Elemente werden basierend auf dem Faktor ausgegeben.

6.2 SampleReducer

Dieser `Reducer` ist für die Hochrechnung der Schätzung verantwortlich. In seiner `reduce()`-Methode wird, anhand des in der `Configuration` enthaltenen Prozentsatzes, der Faktor berechnet. Dieser gibt an, wie oft die Elemente vervielfacht werden müssen, um eine Schätzung für 100% zu erhalten.
Anschließend werden die Werte in ein spezielles `Iterable`-Objekt gespeichert, dieses erhält zusätzlich den berechneten Faktor. Dieses `MultiIterable` wird dann der benutzerdefinierten Reduce-Funktion zusammen mit dem korrespondierenden Schlüssel und `Context` übergeben. Falls die optionale Hochrechnung nicht gewählt wurde, wird die benutzerdefinierte Reduce-Funktion direkt ausgeführt.

6.3 MultiIterable

Die generische `MultiIterable`-Klasse übernimmt die Aufgabe, die Elemente zu vervielfachen. Dies geschieht, um Speicherplatz zu sparen, virtuell. Beim Zugriff auf ein Element, wird geprüft wie oft dieses Element auszugeben ist und wie oft es bereits ausgegeben wurde. Falls es noch nicht so oft ausgegeben wurde wie verlangt, wird es wieder ausgegeben. Sollte die maximale Anzahl erreicht sein, wird zum nächsten Element voran geschritten.
Sollte der Faktor keine ganzzahlige Zahl sein, werden Elemente zusätzlich ausgegeben. Welche Elemente für diese Zusatzausgaben gewählt werden, ist zufällig. Für diese Auswahl wurden zwei Ansätze implementiert. Zum einen basierend auf der Wahrscheinlichkeit, die sich aus dem Rest herleitet, bei einem Faktor von 1,5 wäre dies ein Rest von 0,5 und eine Wahrscheinlichkeit von 50%. Dabei wird für jedes Element, nachdem es die maximale ganzzahlige Höchstgrenze an Ausgaben erreicht hat, zufällig entschieden, ob eine zusätzliche Ausgabe erfolgt. Beim zweiten Ansatz wird eine Art Überlauf verwendet. Hierbei wird pro Element der Rest des Faktors aufaddiert. Sollte diese Summe größer oder gleich 1 sein wird das aktuelle Element zusätzlich ausgegeben und die Summe um 1 verringert. Da die Anordnung der Elemente standardmäßig zufällig ist, entsteht auch hier eine zufällige Auswahl. Dies geschieht in der `next()`-Methode des

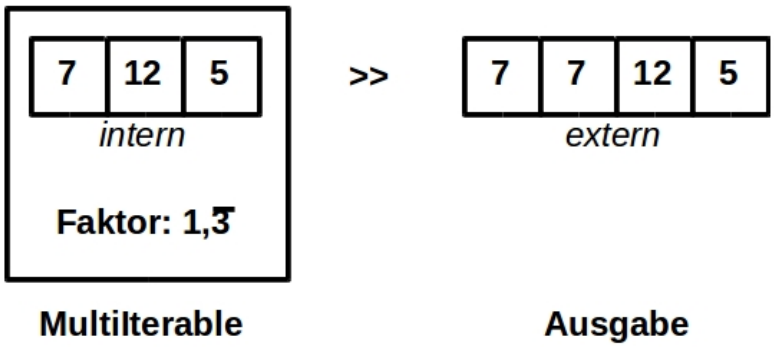


Abbildung 6.2: Der nicht ganzzahlige Rest $\frac{1}{3}$ bedingt, dass ein (zufälliges) Element doppelt ausgegeben wird.

enthaltenen `Iterator`s.

```
public T next() {
     if (count == threshold) {
          if (!spilled) {
               spill += rest;
          }
          if ((spill >= 1.0) && (!spilled)) {
               spill = spill - 1.0;
               spilled = true;
          } else {
               spilled = false;
               count = 1;
               index++;
          }
     } else {
          count++;
     }
     return (T) data.get(index);
}
```

`threshold` ist die Obergrenze, `count` die Anzahl an Ausgaben, `spill` der Überlauf, `index` der Index des aktuellen Elements und `spilled` gibt an, ob für dieses Element bereits eine zusätzliche Ausgabe durchgeführt wurde.

6.4 SampleJob

Ein `SampleJob` ist eine Erweiterung der Hadoop `Job`-Klasse, welche für die Ausführung einer MapReduce-Berechnung verantwortlich ist. Diese Klasse sorgt dafür, dass der `SampleReducer` anstatt des benutzerdefinierten `Reducer`s aufgerufen wird. Dieser greift dann auf den ursprünglichen `Reducer`, der in der `Configuration` gespeichert wird, zu. So kann in den meisten Fällen die gleiche Reduce-Funktion verwendet werden, welche auch für eine komplette Berechnung verwendet wird.

6.5 EstimateResultsMapper

Der `EstimateOldResultsMapper` und der `EstimateTmpResultsMapper` sind Bestandteil der inkrementellen Komponente. Sie sind generische Mapper-Klassen, welche als Superklasse für die benutzerdefinierten `Mapper` zum Lesen der Schätzungen verwendet werden müssen. Bei der Ausgabe wird ein spezielles `Context`-Objekt verwendet. Dieses prüft, ob der Ausgabetyp das `WeightedWritable`-Interface implementiert. Sollte dies der Fall sein, so wird die Gewichtung berechnet und mittels der `setFactor()`-Methode des Typs gesetzt. Anschließend wird das gewichtete Zwischenergebnis ausgegeben.

6.6 WeightedWritable-Interface

`WeightedWritable<T>` ist eine Erweiterung des `Writable`-Interfaces. Es bildet ein 2-Tupel aus einem generischen Typ T, der den Wert repräsentiert und einem `Integer`, welcher die Gewichtung ermöglicht. Der Benutzer muss in seiner Map-Funktion einen Ausgabewert verwenden, der das `WeightedWritable`-Interface implementiert, damit eine Gewichtung durchgeführt werden kann.
Zusätzlich zu den Standardmethoden `get()` und `set()` für den Wert, bietet es noch Methoden[7] um den Gewichtungsfaktor zu beeinflussen.

6.7 EstimateReducer

Der `EstimateReducer` bildet das Gegenstück zu den `Mapper`, da hier die gewichteten Werte wieder vereint werden. Diese Berechnung erfolgt vor der benutzerdefinierten Reduce-Funktion. Wenn die neue Schätzung berechnet ist, wird diese an die Funktion des Nutzers weitergeleitet. Diese ist dann nur noch für die Datenrepräsentation zuständig.
Sollte das Zwischenergebnis nicht das `WeightedWritable`-Interface implementieren oder keinem numerischen Type angehören, werden die Werte direkt der benutzerdefinierten Funktion übergeben. Diese ist dann für die Aktualisierung verantwortlich.

6.8 EstimateJob

Der `EstimateJob` fasst die inkrementellen Komponenten zusammen. Er basiert wie der `SampleJob` auf der `Job`-Klassen. Dabei übernimmt auch hier die `waitForCompletion()`-Methode die Aufgabe, den `Reducer` des Nutzers mit dem `EstimateReducer` auszutauschen. Diesem wird der Originalreducer mittels der `Configuration` zugänglich gemacht, da dieser auf die Reduce-Funktion nach seiner Berechnung zugreift.
Da die automatische Aktualisierung optional ist, wird, falls deaktiviert, der benutzerdefinierte Reducer direkt aufgerufen, welcher dann die gewichteten Ergebnisse direkt erhält.

[7] `setFactor()`,`getFactor()`,`amplifyFactor()`

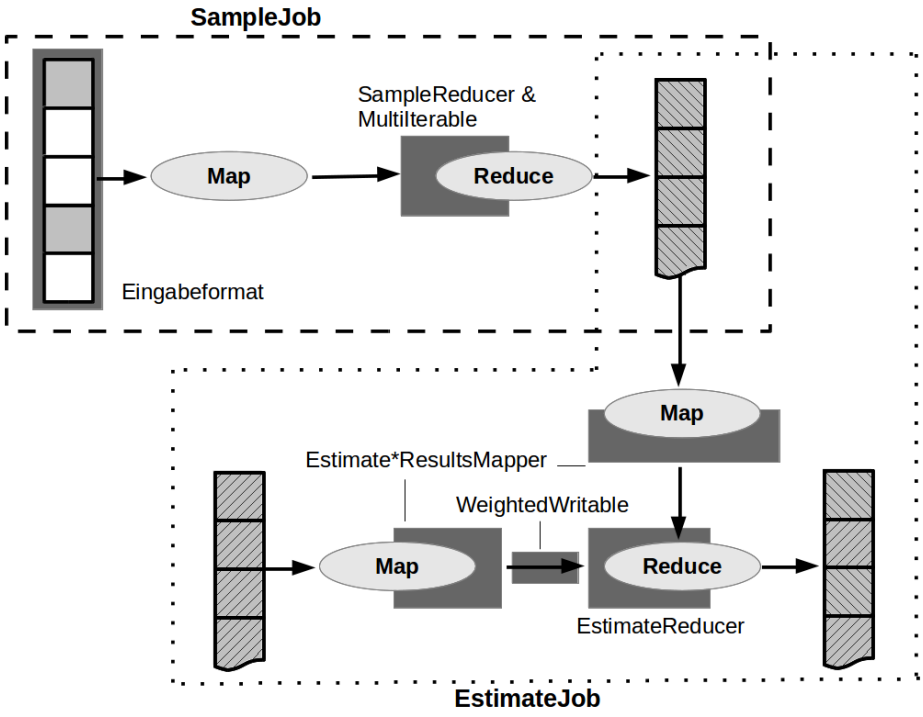


Abbildung 6.3: Gesamtübersicht einer inkrementellen Berechnung. Die ovalen Komponenten sind die benutzerdefinierten Funktionen und die dunkelgrauen Kästen symbolisieren die zusätzlichen Komponenten zur Automatisierung.

6.9 SamplingEstimationComputation

Diese Komponente vereint `SampleJob` und `EstimateJob`. Sie wird ähnlich wie die `Job`-Klasse verwendet. Ihr Aufbau ist in Abbildung 6.3 zu sehen. Dabei besitzt sie, neben diverser `setter`-Methoden, drei Hauptmethoden:

`waitForCompletion()` :
führt eine komplette inkrementelle Berechnung aus. Dabei wird ein `EstimateJob` an einen `SampleJob` gekettet. Dafür werden die beiden folgenden Methoden verwendet.

`runSampleJob()` :
führt einen `SampleJob` basierend auf den gesetzten Klassen und Werten aus. Wenn nicht anders spezifiziert, verwendet es das `RandomInputFormat` und das `TextOutputFormat`. Sollte diese Methode im Rahmen einer kompletten inkrementellen Berechnung aufgerufen werden, so wird die Ausgabe im temporären Ergebnis-Ordner mittels des `TextOutputFormats` gespeichert.

`runEstimateJob()` :
führt eine Aktualisierung basierend auf zwei Schätzungen aus. Die `Mapper` müssen `EstimateOldResultsMapper` und `EstimateTmpResultsMapper`

implementieren, da ansonsten keine automatische Gewichtung durchgeführt wird. Diese Methode kann einzeln oder im Rahmen der `waitForCompletion()`-Methode aufgerufen werden.

In Abbildung 6.4 sieht man ein Beispiel für eine Wordcount-Berechnung mittels der `SamplingEstimationComputation`-Komponente.

6.10 Parameter

mapreduce.sample.percent	Gesamtprozentsatz der Stichprobe
mapreduce.sample.percent.lines	Prozentsatz der Stichprobenauswahl auf Zeilenebene
mapreduce.sample.percent.splits	Prozentsatz der Stichprobenauswahl auf Splitebene
mapreduce.sample.correctingResult	Hochrechnung aktivieren default true
mapreduce.sample.reduce.class	Hier wird der benutzerdefinierte Reducer zwischengespeichert, erfolgt automatisch
mapreduce.sample.sampled	Bereits berechneter Prozentsatz für inkrementelle Berechnung
mapreduce.estimation.update	Zusammenführen der Schätzungen aktivieren, default true
mapreduce.sample.percent.lines.exact	Aktivieren des genauen `RandomExactRecordReader` für Tests auf kleinen Eingabedaten

```
SamplingEstimationComputation sec = new SamplingEstimationComputation(conf, name);
sec.setJarByClass(Wordcount.class);
sec.setIntermediatePath(new Path(tmpPath));
sec.setOldResultInputPath(new Path(oldResult));
sec.setOldResultsEstimationMapperClass(OldResultMapper.class);
sec.setTmpResultsEstimationMapperClass(TmpResultMapper.class);
sec.setOutputPath(new Path(out));
sec.setSamplingInputPaths(in);
sec.setSamplingMapOutputKeyClass(Text.class);
sec.setSamplingMapOutputValueClass(LongWritable.class);
sec.setSamplingInputFormatClass(RandomInputFormat.class);
sec.setSamplingMapperClass(CountMapper.class);
sec.setSamplingReducerClass(CountReducer.class);
sec.setEstimationReducerClass(EstimationReducer.class);
sec.setEstimationMapOutputKeyClass(Text.class);
sec.setEstimationMapOutputValueClass(WeightedLongWritable.class);
return (sec.waitForCompletion(true) == true) ? 0 : -1;
```

Abbildung 6.4: Verwendung der `SamplingEstimationComputation` in einer inkrementellen Wordcount-Berechnung.

Kapitel 7

Evaluierung

Um festzustellen, inwiefern sich die entwickelten Komponenten auf die Laufzeit auswirken, wurden unter Verwendung verschiedener Parameter Tests durchgeführt. Als Beispielanwendung dient dabei eine Wordcount-Berechnung. Dabei wurden die beiden Ansätze der Zufallsauswahl, zeilenbasiert und splitbasiert, miteinander und mit einer vollständigen Berechnung verglichen. Des weiteren wurden eine Kombination dieser beiden betrachtet. Anschließend wurden die Leistungen der inkrementellen Komponenten untersucht. Dabei wurden sie aufbauend auf einer normalen Stichprobenberechnung verwendet im Vergleich zu einer Stichprobenziehung mit einem höheren Prozentsatz. Bei diesen Tests wurde in erster Linie die Berechnungszeit betrachtet, es wurden aber auch die Genauigkeit der jeweiligen Schätzungen erfasst.

7.1 Testaufbau

Die Tests wurden auf einem Cluster mit fünf Knoten durchgeführt. Die Knoten waren Quadcore 2,53GHz Rechner mit 4GB RAM, 1TB SATA-II Platten und mit Gigabit Ethernet. Als Daten wurden über 3200 englischsprachige Texte des Project Gutenbergs[12] verwendet, welche zusammen einen Umfang ca. 1,2 GB hatten.

7.2 Ergebnisse

Nachfolgend werden die jeweiligen Ergebnisse vorgestellt, angefangen bei einer normalen Stichprobenberechnung bis hin zu einer inkrementellen Berechnung. Zuerst werden dabei die Laufzeiten der unterschiedlichen Ansätze betrachtet und anschließend die Genauigkeit.

7.2.1 Zeilenbasierte Schätzung

Bei einer zeilenbasierten Schätzung wird eine Zufallsauswahl auf der Menge aller Zeilen getroffen, dabei wird für jeden Zeile eine Zufallsentscheidung getroffen, ob die Zeile in die Stichprobe gelangt oder nicht.
In Abbildung 7.1 ist ein Vergleich zwischen der Berechnungsdauer einer kompletten Berechnung und der Berechnung einer Stichprobe mit anschließender

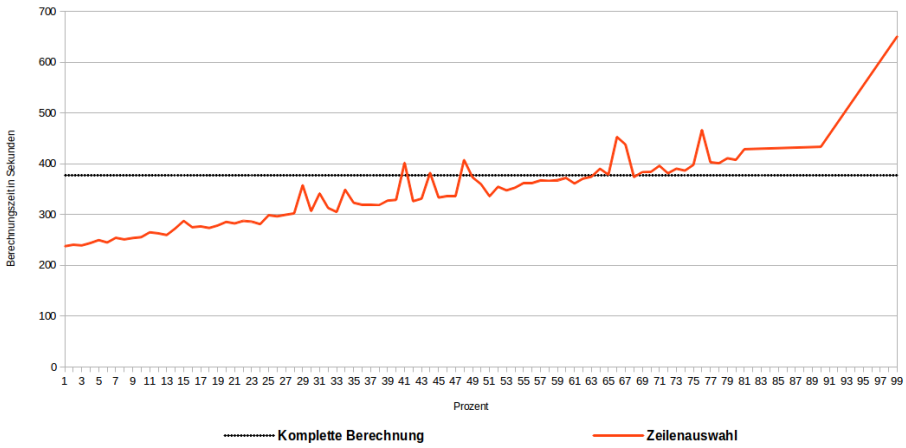


Abbildung 7.1: Zeilenbasierte Auswahl.

Hochrechnung zu sehen. Es ist dabei zu erkennen, dass die Berechnung der Schätzung bis zu einem gewissen Prozentsatz eine deutlich verringerte Laufzeit aufweist. Dabei ist zu Beginn für die 1%-Stichprobe eine Laufzeitminimierung von ca. 40% erreicht worden. Und erst ab einer Stichprobengröße von ca. 70% ist eine komplette Berechnung gleich schnell bzw. schneller. Ein wesentlicher Faktor bei der Laufzeit einer zeilenbasierten Zufallsauswahl ist die Tatsache, dass trotzdem über alle Zeilen iteriert wird. Somit könnte der Zeitgewinn für Funktionen mit rechenintensiven Map-Funktionen höher ausfallen, da der Zeitgewinn zum größten Teil auf dem Wegfallen der Daten in der Map-Funktion entfällt.

7.2.2 Splitbasierte Schätzung

Die Ergebnisse der splitbasierten Auswahl sind in Abbildung 7.2 zu sehen. Auch hier sieht man bereits, wie bei der zeilenbasierten Auswahl, dass die Berechnungszeit im Vergleich zur kompletten Berechnung verringert wurde. Bei Berechnungen für Stichproben von 1% der Gesamtdaten ist die Verbesserung der Berechnungszeit bei über 90%. Bei einem Stichprobenumfang von ca. 85% erreicht die Berechnungszeit der Schätzung auch erst die Berechnungszeit einer kompletten Neuberechnung. Der Vorteil an diesem Ansatz ist, dass bei einer Zufallsauswahl auf der Menge der Splits nicht mehr über jeden Datensatz iteriert werden muss. Da nach der Auswahl der jeweiligen Splits eine Vollerhebung dieser durchgeführt wird, wird in diesem Ansatz tatsächlich nur über die Daten iteriert, die auch letztendlich berechnet werden. Auch werden in diesem Ansatz die Zufallsentscheidungen zur Auswahl der Splits alle in einem Methodenaufruf durchgeführt, diese `getSplits`-Methode wird von dem `JobClient` aufgerufen. Dieser erhält daraufhin eine Liste der jeweilige Splits, die Information über die Position der Splits wird dann dem `jobTracker` übermittelt, der für die Planung der Map-Tasks zuständig ist. Somit wird der ganze Zufallsentscheidungprozess bereits vor dem Start der eigentlichen Berechnungen durchgeführt, wohingegen bei der zeilenbasierten Auswahl die Entscheidung in Echtzeit bei dem Aufruf der jeweilige Folgezeile bzw. Schlüssel/Wert-Paar geschieht. In Abbildung 7.2

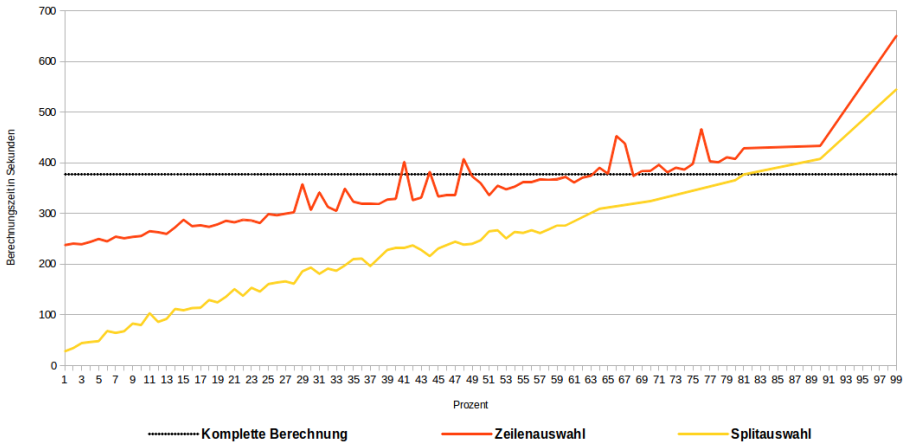


Abbildung 7.2: Vergleich zwischen den beiden primitiven Auswahlansätzen.

erkennt man auch den signifikanten Geschwindigkeitsvorteil der splitbasierten Auswahl im Vergleich zum zeilenbasierten Ansatz. Dabei beträgt die Berechnungszeit des Split-Ansatzes zu Beginn nur ca. 10% der Berechnungszeit des Zeilen-Ansatzes. Mit zunehmendem Prozentsatz sieht man aber, dass der Abstand kontinuierlich abnimmt, da die Dauer der Berechnung bei dem splitbasierten Ansatz stärker zunimmt als bei dem zeilenbasierten Ansatz. Dies lässt sich durch die Tatsache erklären, dass mit zunehmendem Prozentsatz die Anzahl der Splits immer weiter zunimmt und somit auch die Anzahl der Zeilen über die für die Auswahl iteriert wird. Im ersten Ansatz ist diese Anzahl jedoch fest und der Geschwindigkeitsvorteil beruht dabei hauptsächlich auf den gesparten Berechnungen. Trotz dieser Annäherung bleibt der zweite Ansatz dem ersten jedoch die ganze Zeit überlegen, da bei diesem der Zeitvorteil durch Einsparungen beim Lesen und beim Berechnen der Daten entsteht.

7.2.3 Kombination von Zeilen- und Splitauswahl

Diese Kombination wurde durch die in Kapitel 6.1.3 angesprochene gleichmäßige und automatische Aufteilung durchgeführt. Dabei wird vom Benutzer nur der gewählte Prozentsatz der Stichprobe angegeben, dieser wird dann auf das `RandomInputFormat`[1] und den `RandomRecordReader`[2] aufgeteilt. Auch dieser Ansatz zeigt eine Verbesserung der Laufzeit, bis zu einem Prozentwert von ca. 80%, siehe Abbildung 7.4. Dabei beträgt sein Zeitgewinn bei einem Prozent ungefährt 87% der Zeit, die für eine normale Berechnung benötigt werden würde. In Abbildung 7.4 werden nun alle drei Ansätze einander gegenübergestellt, hier sieht man auch, dass die Zunahme der Berechnungsdauer im Kombinationsansatz der Zunahme der Splitauswahl ähnelt. Diese Ähnlichkeit beruht auf der Tatsache, dass beim Kombinationsansatz auch durch hinzukommen von Splits über mehr Datensätze iteriert werden muss. Die Berechnung von 1% wird dabei auf eine 10% Zeilenauswahl und eine 10% Splitauswahl aufgebrochen. Die Zeit einer 10% Schätzung, basierend nur auf Splits, ist dabei nur etwas langsamer

[1] Splitbasierte Auswahl
[2] Zeilenbasierte Auswahl

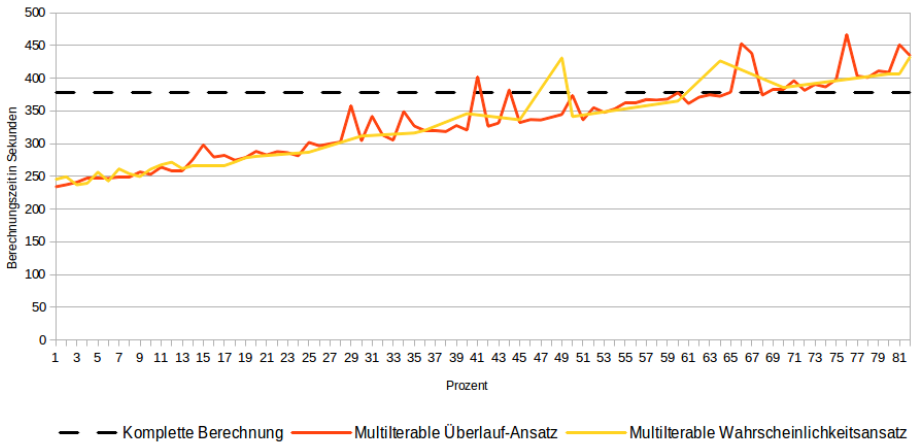


Abbildung 7.3: Die beiden `MultiIterable`-Ansätze im Vergleich.

als die Berechnung der 1% Schätzung der kombinierten Auswahl. In beiden Berechnungen wird dabei über 10% der Zeilen iteriert, bei der ersten Schätzung werden aber noch zusätzlich das zehnfache der Daten berechnet, dies könnte den Hauptbestandteil der Differenz ausmachen.

7.2.4 Vergleich der beiden MultiIterable-Implementierungen

In diesem Abschnitt wird das Abschneiden der beiden `MultiIterable`-Implementierungen verglichen. Diese unterscheiden sich bei der Zufallsauswahl der zusätzlich ausgegebenen Elemente, siehe Kapitel 6.3. Dabei verwendet eine Implementierung eine Auswahl basierend auf einer Zufallszahl und die andere Implementierung verwendet einen Überlauf. In Abbildung 7.3 werden die beiden Implementierungen für eine Zeilenauswahl angewendet. Da hier die Berechnungszeit mehr von der Berechnung der Daten als von der Eingabe abhängt, wurde dieser Ansatz zur Evaluierung gewählt. Wie zu sehen ist, gibt es aber keine erwähnenswerten Performanzunterschiede zwischen den beiden Ansätzen.

7.2.5 Inkrementeller Ansatz

Der inkrementelle Ansatz kann dazu genutzt werden, eine bereits vorhandene Schätzung zu erweitern. Dazu wurden Tests durchgeführt in denen Stichproben jeweils noch mal um den bereits berechneten Prozentsatz erweitert wurden, die Berechnungszeit dieser Verdopplung für die Auswahlverfahren wird in Abbildung 7.5 mit der Berechnungszeit einer nicht inkrementellen Berechnung dieses neuen Prozentsatzes verglichen. Dabei kann man erkennen, dass mit der Verwendung der Zeilenauswahl erst ein zeitlicher Vorteil gewonnen wird, wenn die inkrementelle Berechnung eine 20%-Stichprobe berechnet[3]. Dabei erhöht sich

[3]Dies entspricht einer Berechnung einer 10%-Stichprobe vereint mit einer bereits vorhandenen Schätzung von 10%, da in diesem Beispiel jeweils eine Verdopplung der initialen Stichprobe durchgeführt wird.

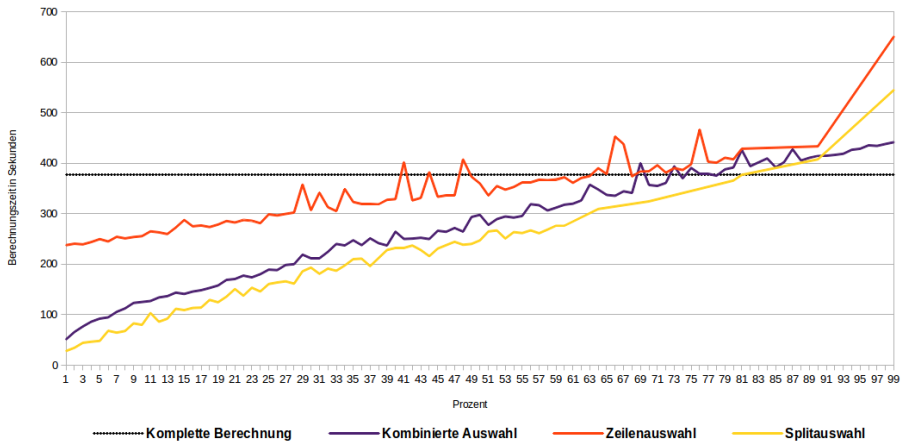


Abbildung 7.4: Die drei Auswahlverfahren im direkten Vergleich.

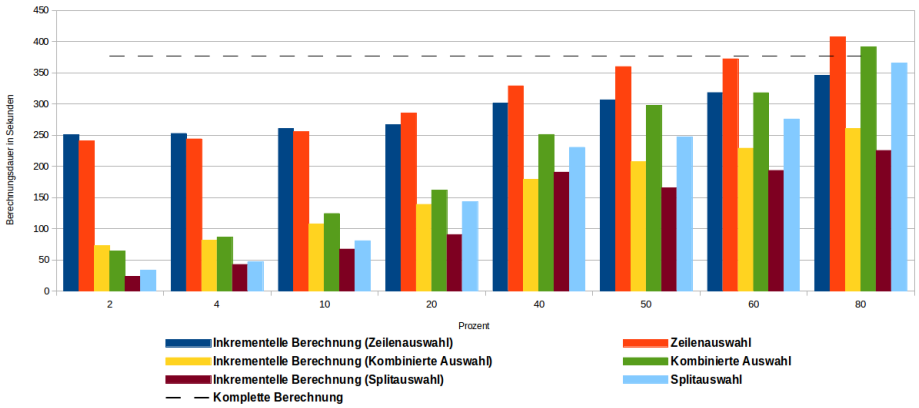


Abbildung 7.5: Inkrementelle Stichprobenerweiterung(Verdopplung) im Vergleich zu einfacher Schätzung.

dieser Zeitvorteil bei zunehmendem Stichprobenumfang. Für die kombinierte Auswahl ist eine Zeitvorteil der inkrementellen Variante schon ab einer Berechnung einer Stichprobe von insgesamt 4% gegeben. Auch hier ist ein Verbesserung des Zeitvorteil mit zunehmendem Stichprobenumfang zu beobachten. Man kann auch erkennen, dass diese Verbesserung schneller wächst als die Verbesserung wenn man eine einfache Zeilenauswahl verwendet. Für die Splitauswahl verhält es sich ähnlich wobei diese direkt bei der Erweiterung auf eine 2% Stichprobe einen zeitlichen Vorteil gegenüber einer normalen 2% Stichprobe aufweist. Auch hier ist die Vergrößerung des Zeitvorteils bei wachsender Stichprobe dem zeilenbasierten Ansatz überlegen. Aufgrund des Umfangs der Testdaten unterliegt die Splitauswahl gewissen Schwankungen, da durch die geringe Anzahl von Splits das *Gesetz der großen Zahlen* noch keinen so großen Einfluss hat. In Abbildung 7.6 sieht man die Zunahme der Berechnungsdauer für einen Stichprobenerweiterung eines bestimmten Prozentsatzes in der inkrementellen Berechnung. Dabei ist auch hier der Zeitvorteil des splitbasierten Ansatzes deutlich zu sehen.

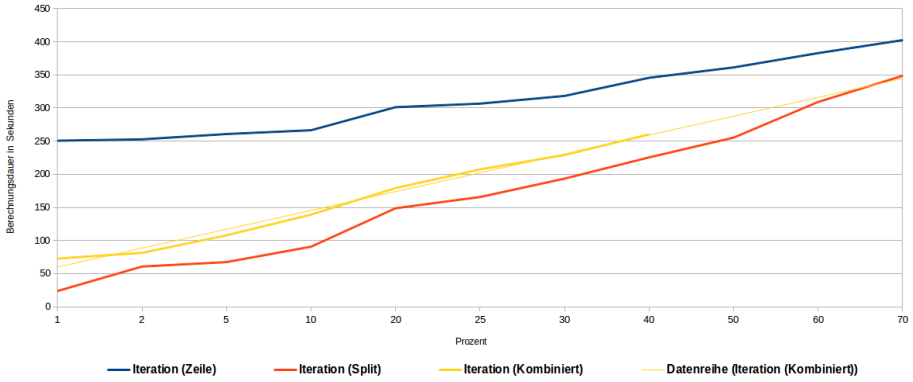


Abbildung 7.6: Zunahme der Berechnungsdauer für die inkrementelle Berechnung.

7.2.6 Genauigkeit

Zusätzlich zu der Berechnungszeit wurde die Genauigkeit der jeweiligen Tests erfasst, dabei wurde das Ergebnis der Hochrechnung mit dem Ergebnis einer kompletten Berechnung verglichen. Für die einzelnen Worte wurde die Differenz zwischen Schätzung und genauem Ergebnis berechnet, dies erfolgt auf der Menge aller Worte der kompletten Berechnung, damit auch die Worte erfasst wurden, die nicht in der Stichprobe vorkamen. Anschließend wurde der Prozentsatz dieser Differenz bestimmt indem diese durch die korrekte Anzahl[4] dividiert wurde. Anhand dieser Werte wurde der durchschnittliche Fehler berechnet[5]. In Abbildung 7.7 sieht man den durchschnittlichen Fehler für die drei Auswahlansätze im Vergleich. Dieser Fehler ist bei den drei Ansätzen für eine Berechnung einer Stichprobe von 1% zwischen 147% für den Splitansatz und 180% für den Zeilenansatz. Dieser Fehler nimmt immer weiter ab, desto größer die berechnete Stichprobe wird. In dem Bereich von ca. 50% - 58% weißen weisen alle Ansätze nur noch Fehlerwerte von ca. 50% auf. Unter die 10% Fehlergrenze fallen die Ansätze bei ca. 90% berechneter Daten. Man sieht auch, dass der Splitansatz und der kombinierte Ansatz Schwankungen unterworfen ist, welche beim Splitansatz stärker ausgeprägt sind, wohingegen der durchschnittliche Fehler bei der Zeilenauswahl nur geringe Schwankungen aufweist. Dies lässt sich durch die Größe der Testdaten erklären, da es noch verhältnismäßig wenige Splits bei einem Datenumfang von 1,2 GB gibt, greift das *Gesetz der großen Zahlen* hier noch nicht richtig. Im Gegensatz dazu gibt es schon eine große Anzahl an Zeilen, womit hier eine bessere Annäherung gegeben ist. Die Splitauswahl ist deshalb am stärksten von den Schwankungen betroffen und da die kombinierte Auswahl auch auf die Splitauswahl zurückgreift, wird auch diese beeinflusst. Hier sind die Schwankungen aber nicht so stark, da ein Teil der Auswahl auf Zeilenebene getroffen wird.

Ein Grund für den relativ großen durchschnittlichen Fehler zu Beginn ist die Tatsache, dass die Wörter einer ungefähren Zipfverteilung unterliegen, da sie aus englischen Texten stammen und die Verteilung der Worte einer natürlichen Spra-

[4] Dem kompletten Ergebnis.
[5] Die Summe alles Differenzprozentsätze durch die Anzahl.

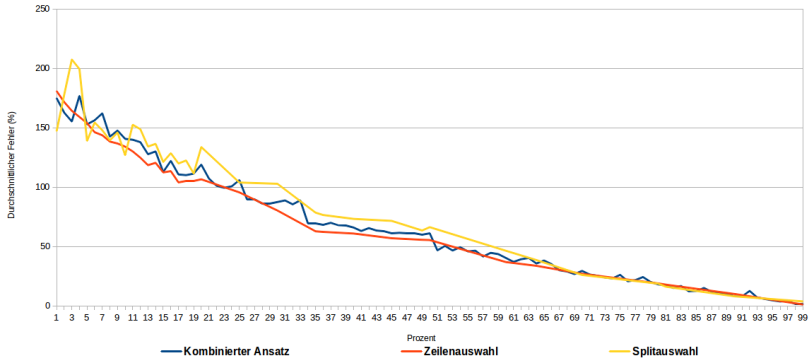


Abbildung 7.7: Durchschnittlicher Fehler für die Auswahlansätze.

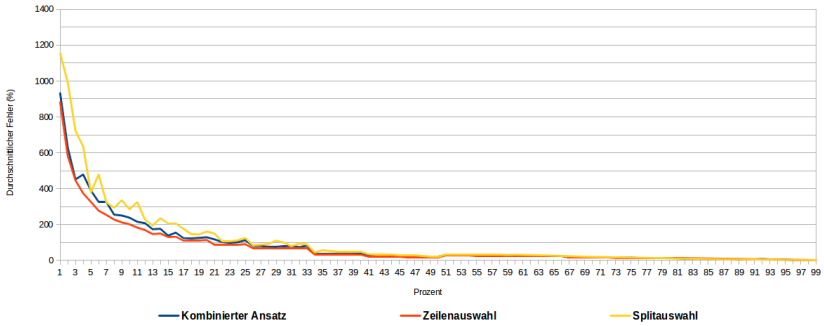


Abbildung 7.8: Durchschnittlicher Fehler für die Auswahlansätze ohne Betrachtung fehlender Wörter.

che annähernd einer Zipfverteilung entspricht[18]. Wenn man sich den durchschnittlichen Fehler ohne die Worte, die in der Stichprobe nicht vorkommen, erhält man einen etwas anderen Verlauf. Hier ist der durchschnittliche Fehler zu Beginn sehr hoch, dies lässt sich auch auf die Zipfverteilung zurückführen. Da es viele Wörter gibt, die selten vorkommen, aber wenn solch ein Wort in einer kleinen Stichprobe vorkommt, dann wird die Häufigkeit solcher Worte wesentlich höher eingeschätzt durch die Hochrechnung. Was man aber auch sieht ist, dass hier der durchschnittliche Fehler wesentlich schneller abnimmt. Dies beruht unter anderem darauf, dass, wenn ein Wort in der Stichprobe nicht vorkommt, dies keine negative Auswirkung auf den Fehler hat. Und da es basierend auf der Verteilung viele seltene Wörter gibt, sind viele Wörter nicht in den Stichproben, was für jedes Wort einen Fehler von 100% in Abbildung 7.7 nach sich zog. Auch fallen solche seltenen Worte in den jeweiligen Stichproben immer weniger ins Gewicht, da der Faktor der Hochrechnung mit steigender Stichprobengröße abnimmt und somit auch der Fehler der mit den Worten assoziiert wird. Deshalb hat man einen durchschnittlichen Fehler von nur noch 50% hier schon bei einem Drittel der gesamten Eingabedaten.

In Abbildung 7.9 sieht man auch noch einmal, wie groß der Einfluss der Verteilung auf die Genauigkeit ist. In diesem Diagramm werden nur die Wörter betrachtet, die mehr als 5000 mal vorkommen. Hier ist der durchschnittliche

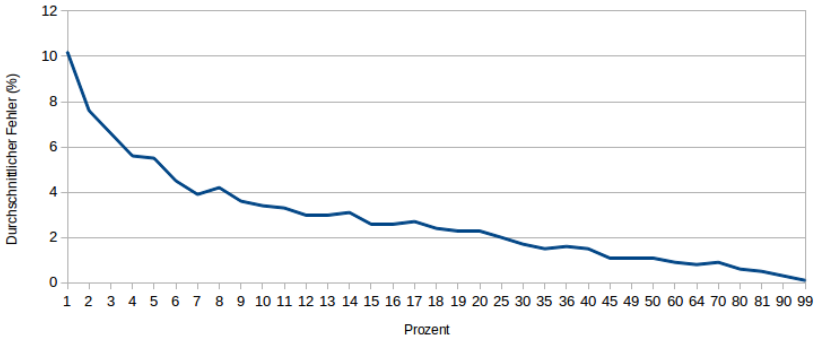


Abbildung 7.9: Durchschnittlicher Fehler für alle Wörter die mehr als 5000 mal in den Eingabedaten vorkommen.

Fehler bereits bei ca. 10% für eine Stichprobe von 1% und sinkt unter 1% bei einer Stichprobe von 60%.

7.3 Fazit

Anhand der beobachteten Ergebnisse kann man verfolgen, wie sich die unterschiedlichen Auswahlverfahren auf Genauigkeit und Laufzeit auswirken. Dabei sieht man, dass die splitbasierte Auswahl, die geringste Laufzeit aufweist, aber von den drei Ansätzen auch die geringste Genauigkeit liefert. Eine höhere Genauigkeit bietet das kombinierte Auswahlverfahren, dieses ist aber auch etwas langsamer als die splitbasierte Auswahl. Die höchste Genauigkeit, der drei Verfahren, zeigte die zeilenbasierte Auswahl, diese hatte aber auch die höchste Berechnungszeit. Vergleicht man diese Ergebnisse mit einer kompletten Berechnung, sieht man, dass alle Auswahlverfahren, bis zu einem gewissen Prozentsatz, eine geringere Laufzeit aufweisen.
Zusätzlich hat sich gezeigt, dass man, wenn man eine inkrementelle Berechnung zur Erweiterung einer bereits vorhandenen Stichprobe verwendet, einen Zeitvorteil erreicht im Vergleich zu der Berechnung einer neuen größeren Stichprobe. Dieser Zeitvorteil ist dabei abhängig von dem Umfang der Erweiterung.

Kapitel 8

Zusammenfassung

Das Ziel dieser Arbeit war es, eine Laufzeitoptimierung für Hadoop-Berechnungen zu erreichen, durch eine Abschätzung des Ergebnisses. Diese Abschätzung sollte eine zusätzliche inkrementelle Erweiterung umfassen, die für bereits berechnete Schätzungen den Stichprobenumfang vergrößert. Dabei sollten alle Komponenten in das Hadoop-Framework integrierbar sein, ohne dass daran Änderungen vorgenommen werden müssen. Des weiteren sollten die Funktionalitäten für einen mapreducevertrauten Nutzer intuitiv zu bedienen sein und durch eine Automatisierung von Stichprobenziehung und Hochrechnung auch eine gewisse Benutzerfreundlichkeit aufweisen.
Dies wurde durch die Implementierung verschiedener Klassen erreicht, der Nutzer ist bei einer einfachen Schätzung nur für Map- und Reduce-Funktion verantwortlich. Für die Stichprobenziehung wurden unterschiedliche Eingabeformate entwickelt, die basierend auf einem durch den Nutzer festgelegten Prozentsatz automatisch die Ziehung einer Stichprobe durchführen. Dabei kann er mittels der Wahl des Eingabeformates bzw. der Art der Auswahl auch bis zu einen gewissen Grad auf die Eingabedaten eingehen. Für die Hochrechnung wurde die `Job`-Klasse erweitert, sodass der Nutzer nur diese Klassen verwenden muss, um eine Hochrechnung zu erhalten. Die inkrementelle Berechnung wurde mittels einer zweiten zusätzlichen Berechnung realisiert, welche zwei Schätzungen gewichtet vereint, dabei ist der Nutzer nur für das Lesen und Schreiben der Daten verantwortlich. So kann, mittels eines iterativen Prozesses, die Genauigkeit der Abschätzung erhöht werden. Die gewichtete Vereinigung erfolgt auch hier durch eine speziell erweiterte `Job`-Klasse automatisch.
Basierend auf einer Implementierung dieser Komponenten wurden Laufzeittests durchgeführt. Dafür wurden drei unterschiedliche Ansätze zur Stichprobengewinnung untersucht. Für eine zeilenbasierte Zufallsauswahl, eine splitbasierte Klumpenauswahl und für eine mehrstufige Zufallsauswahl basierend auf Splits und Zeilen wurden die Daten erhoben. Dabei hat sich gezeigt, dass bei allen drei Ansätzen eine Verbesserung der Laufzeit erfolgte. Dabei lieferte die Auswahl auf Splitebene die besten Ergebnisse. Für eine Stichprobe von 1% wurde eine Verbesserung der Laufzeit von ca. 90% festgestellt, dicht gefolgt von der kombinierten Auswahl mit einer Verbesserung von ca. 87%. Bei der Zeilenauswahl wurde für den gleichen Stichprobenumfang nur eine Verbesserung von 40% erreicht, dies ist auf die Art der Auswahl zurückzuführen, da über alle Zeilen iteriert werden muss unabhängig von der Größe der Stichprobe, was bei

den anderen beiden Ansätzen nicht der Fall ist. Die Berechnung der Schätzung ist bis zu einem Prozentsatz von 70% für die Zeilenauswahl, bzw. ca. 80% und 85% für die kombinierte Auswahl und die Splitauswahl, schneller als eine komplette Berechnung. In Tests bezüglich der inkrementellen Berechnung wurden auch Zeitersparnisse festgestellt. So ist eine inkrementelle Berechnung zur Verdopplung des Stichprobenumfangs schneller als eine komplette Berechnung. Außerdem ist eine inkrementelle Berechnung auf einer vorhandenen Schätzung ab einem gewissen Prozentsatz schneller als die normale Berechnung einer Stichprobe. Für eine Zeilenauswahl ist dies eine Erweiterung von 10% und für die kombinierte Auswahl bereits für eine Erweiterung von 2%. Bei der splitbasierten Auswahl ist dies bereits bei 1% der Fall. Im Bezug auf die Genauigkeit ist auch auf die Verteilung der Daten zu achten. Da die Hochrechnungsfunktion von gleich verteilten Daten ausgeht, kann für andere Verteilungen eine Verfälschung entstehen. Die Ergebnisse hängen natürlich auch von der Art der jeweiligen Berechnung ab, so ist im Fall der Wordcount-Berechnung der Zeitgewinn hauptsächlich auf die Zeitersparnis beim Lesen der Eingabedaten zurückzuführen, da die Map- und Reduce-Funktion für diese Anwendung nicht sonderlich komplex und zeitaufwendig sind. So kann es sein, dass für Anwendungen mit komplexeren Map-Funktionen ein größerer Zeitvorteil erreichbar ist, da weniger Daten in der Map-Funktion verarbeitet werden müssen. Durch die Art wie die Hochrechnungsfunktion umgesetzt ist, ist kein zusätzlicher Zeitgewinn durch komplexere Reduce-Funktionen wahrscheinlich, da die Reduce-Funktion ähnlich viele Daten wie bei einer kompletten Berechnung verarbeiten muss Dieses Potential für Zeitersparnis kann man in dieser Implementierung nur durch Deaktivieren der Hochrechnungsfunktion und durch eine manuelle Korrektur des Ergebnisses in der Reduce-Funktion erreichen. Dem Problem, dass bei der Zeilenauswahl über jede Zeile iteriert wird, könnte man eventuell begegnen indem man ein Eingabeformat ähnlich dem EARL-*pre-map*-Eingabeformat gestaltet. In diesem wird nicht für jeden Zeile eine Zufallsentscheidung getroffen sondern es wird bei jedem Aufruf der Methode zufällig zu einer Zeile im Split gesprungen und diese dann verwendet. Für solch ein Eingabeformat müsste dann noch ein Terminierungskriterium gefunden werden. Auch wäre eine Art Kontrollfunktion der Genauigkeit ähnlich der in EARL in der inkrementellen Berechnung denkbar, vlt. basierend auf der Differenz der zu vereinenden Schätzungen. Außerdem wäre für die inkrementelle Berechnung noch eine Funktionalität sinnvoll, die für alle Berechnungen, wie MIN, MAX, etc, eine gewichtete Zusammenführung ermöglicht und nicht nur für berechnete Ergebnisse. Aufgrund der Tatsache, dass die Stichprobenziehung mittels eines Eingabeformats durchgeführt wird, ist hier noch Potential für zusätzliche Speichersysteme und Auswahlverfahren gegeben.

Literaturverzeichnis

[1] Apache Software Foundation. *Apache Hadoop project*. `http://hadoop.apache.org`.

[2] Apache Software Foundation. *Apache HBase*. `http://habase.apache.org`.

[3] Apache Software Foundation. *Hadoop Distributed File System*. `http://hadoop.apache.org/hdfs`.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Seventh Symposium on Operating System Design and Implementation*, pages 205–218, 2006.

[5] Laurence Goasduff Christy Pettey. Gartner says solving 'big data' challenge involves more than just managing volumes of data, 2011.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 01 1979.

[8] James Manyika et al. Big data: The next frontier for innovation, competition, and productivity, 2011.

[9] Ludwig Fahrmeir. *Statistik*. Springer-Lehrbuch. Springer, 6., überarb. aufl. edition, 2007.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system, 2003.

[11] Ashish Gupta and Inderpal Singh Mumick. Materialized views. chapter Maintenance of Materialized Views: Problems, Techniques, and Applications, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[12] Michael S. Hart. Project gutenberg, 1971. `http://www.gutenberg.org/wiki/Main_Page`.

[13] IBM. Ibm big data, 2013. `http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`.

[14] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *Proceedings of the third international workshop on Cloud data management*, CloudDB '11, pages 7–14, New York, NY, USA, 2011. ACM.

[15] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *CoRR*, abs/1207.0142, 2012.

[16] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *IN SIGMOD*, pages 100–111, 1997.

[17] Kelton Research. Global survey: The business impact of big data, 2010.

[18] Catriona Tullo and James R Hurford. Modelling zipfian distributions in language.

[19] David Reinsel Vernon Turner, John F. Gantz and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Kaiserslautern, 15. Mai 2014

Marc Schäfer