„Untersuchung zu den Begriffen Gültigkeit und Beweisbarkeit in einer mathematischen Theorie"

Reto Hablützel, Max Schrimpf

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

> Es soll augehend von einem gegebenen einfachen Axiomensystem (z.B "Gruppe") ein einfacher Satz einereits bewiesen werden und anderseits seine "Gültigkeit" illustriert werden. Ein Beweis soll entlang von gegebenen formalisierten Schlussregeln erfolgen, von der Gültigkeit soll man sich mittels "durchrechnen" überzeugen, bei unendlichen Strukturen mit einer zufallsgenerierten Auswahl von Fällen.

Ausgehend von dieser vorgegebenen Aufgabenstellung soll in der vorliegenden Projektarbeit eine Implementation mit der Fähigkeit die Gültigkeit eines Axiomsystemes in den Grundsätzen zu bewisen, erarbeitet werden. Durch das enorme Wachstum der Ausgangswerte sind dabei schon relativ kleine Systeme mit bedeutendem Rechenaufwand verbunden, weshalb ein sinnvoller Ansatz zum Umhang mit rechenintensiven Aufgaben gefunden werden muss.

Die genaue Schwerpunktsetzung bei der Umsetzung und das Vorgehen war dabei durch die offene Aufgabenstellung dem Projektteam überlassen. Vorgegeben war mur die Implementation eines automatisierten Gültigkeitsbeweises des Satzes zur Rechtsauslöschung in Gruppen.

1.2 Ausgangslage

Wir konnten uns im letzten Semester als Vertiefungsarbeit im Fach „Algorithmen und Datenstrukturen" mit dem Thema MapReduce beschäftigen. Dazu hatten wir, anhand des Entwrufsmuster (1.4 MapReduce), unser eigenes Framework für parallelisierte MapReduce Berechnungen implementiert.

Bei der Analyse der vorliegenden Problemstellung kam uns schnell die Idee, dass es möglich wäre die im letzten Semester gewonnenen Erkenntnisse sowie die bestehende Software zur Umsetzung einer Lösung zu Nutzen.

Aufgrund der beschränkt zur Verfügung stehenden Zeit, mussten wir es bei der damaligen Implementation auf einem einzelnen Rechner mit mehreren Threads belassen, anstatt ein ganzes Computernetz verwenden zu können. Ausserdem ergaben sich weitere technische Limitierungen. So konnte unser Framework zum Beispiel nicht mit dem Szenario eines abnormal terminierenden Threads, umgehen. Die Handhabung solcher Ereignisse ist, insbesondere im Ausblick auf eine echt verteilte Berechnung interessant, da in einem Netzwerk immer wieder eine einzelne Maschine aussteigt.

1.3 Abgeleitete Aufgabenstellung

Basierend auf unserem Wissen aus den Algebra-Vorlesungen, unserem Interesse für verteilte Berechnungen und dem bestehenden MapReduce Framework, einigten wir uns mit unserem Betreuenden Dozenten Herrn Heuberger dahingehend, dass wir in einem ersten Teil die Prüfung eines Satzes für ein bestimmtes Axiomsystem als MapReduce Problem formulieren und dieses dann mit unserem Framework ausführen sollten.

Anschliessend daran wurde uns die Option gewährt selbst eine vertiefende Weiterführung der Projektarbeit mit den in Teil 1 erworbenen Kenntnissen wählen. Hierzu mehr im Kapitel „Möglichkeiten der Vertiefung der Aufgabenstellung".

1.4 MapReduce

MapReduce ist ein Entwurfsmuster, welches die parallele Verarbeitung grosser Datenmengen vereinfachen soll und erstmals von Jeffrey Dean und Sanjay Ghemawat aus dem Google research Team beschrieben [DG04] sowie in einem Framework implementiert wurde. Google ist mit dem Problem grosser Datenmengen zum Beispiel beim erstellen der „Google Suche" konfrontiert. Hierbei muss quasi das gesammte Internet indexiert werden und durchsucht werden um es für die Suche in geeigneter Form aufzubereiten.

Um ein Programm so zu entwickeln, dass es auf einer verteilten Architektur gut skaliert, ist jedoch ein enormer Aufwand, welcher nichts mit dem eigentlich zu lösenden Problem zu tun hat, nötig. Daher wurde das Framework so konzipiert, dass es Benutzern erlaubt ihr Problem in zwei nur Phasen zu definieren ohne sich Gedanken um die anschliessende Verteilung und Berechnung machen zu müssen. So kümmert sich das Framework zum Beispiel darum, was passiert, wenn eine Maschine ausfällt oder wie die einzelnen Arbeittsschritte auf die verschiedenen Maschinen aufgeteilt werden.

Die zwei Phasen des Entwurfsmusters sind, wie der Name berieits suggeriert, Map und Reduce (mit einer eine optionalen, zwischenliegenden, für das Verständnis irrelevanten Combiner Phase). In diesen Phasen werden im Wesentlichen die beiden Funktionen Map und Reduce aufgerufen, welche hier kurz erklärt werden sollen.

$$map^1 : (a \to b) \to A \to B$$

Die Map Funktion bildet eine Menge von Elementen elementweise auf eine neue Menge ab. Dazu müssen ihr zwei Parameter übergeben werden – eine Funktion, welche eine Element aus der Menge A auf ein Element in der Menge B abbildet und eine Menge von Elementen A. Daraufhin wird die Funktion sukzessive auf die Elemente der Liste A angewendet und somit die Menge B konstruiert, welche das Resultat ist.

$$reduce : (b \to c \to c) \to c \to B \to c$$

Die Reduce Funktion bildet eine Menge B auf einen einzelnen Wert ab c. Dafür erhält sie als ersten Parameter eine Funktion, mit einen Wert aus der Menge B sowie einen Akkumulator, welche einen Wert b zurückgibt. Als zweiten Parameter erhält die Reduce Funktion einen initialen Wert für den Akkumulator und als dritten und letzten Parameter die Menge B. Dann wird die übergebene Funktion mit jedem Element aus der Menge B und dem Akkumulator aufgerufen. Dabei ist das Resulat der Funktion jeweils der neue Wert für den Akkumulator. Sobald die ganze Liste traversiert wurde, ist der Wert vom Akkumulator das Resultat.

Der Benutzer des MapReduce Frameworks muss also für eine auszuführende Berechnung nur noch eine Funktion schrieben, welche ein Element aus der Menge A auf ein Element in der Menge B abbildet (Map Phase), eine Funktion schreiben, welche für ein Element aus der Menge B, kombiniert mit einem

[1] Schreibweise: a ist ein Element aus einer Menge A

Akkumulator, auf ein Element aus der Menge C abbildet (Reduce Phase) und initial die Menge$A$ bereitstellen.

Für das MapReduce Framework ist es daraufhin möglich zum Beispiel die erste Funktion mit einem Element aus der Menge A an eine berechnende Maschine senden. Auch ordnet das Framework in mit einem sogenannten Shuffle nach der Map Phase die Elemente so an, dass sie von der Reduce Phase verarbeitet werden können. Die Anordnung der Elemente der Menge B muss dabei zu beginn der Reduce Phase nicht zwingend ihrer Anordnung nach der Map Phase entsprechen.

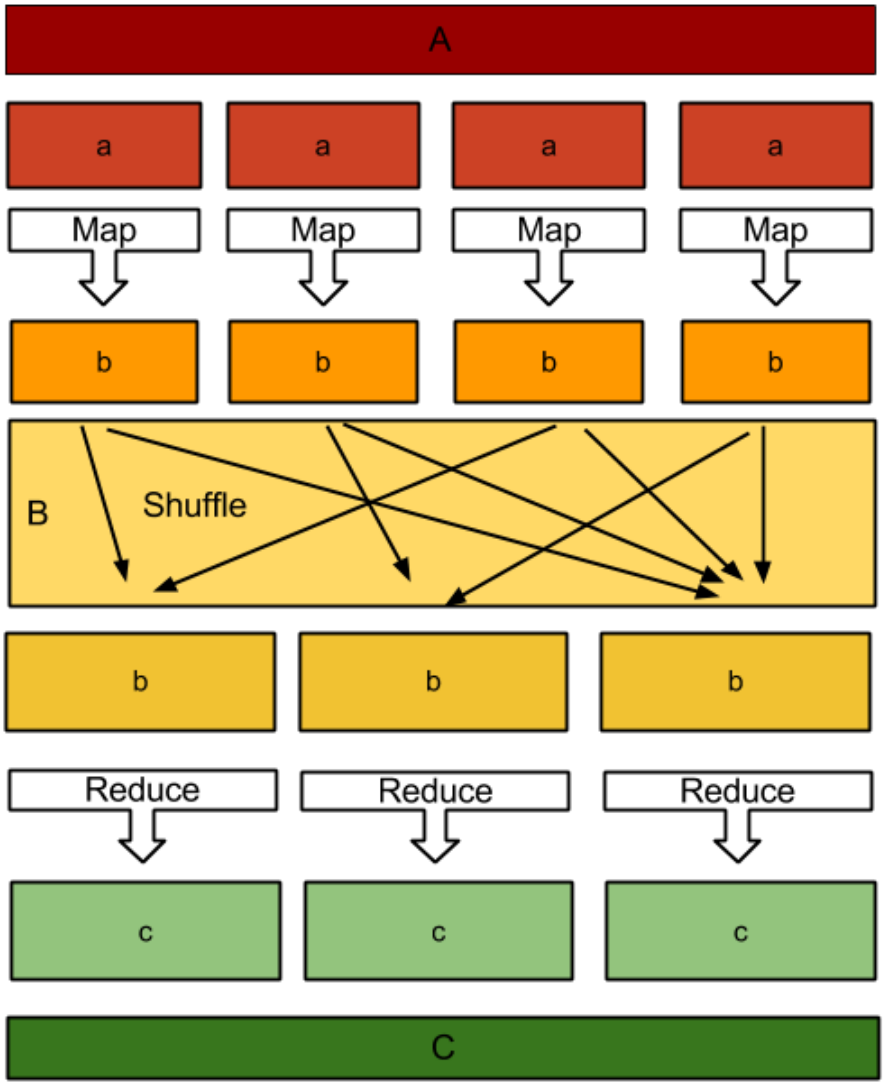


Abbildung 1: Übersicht über die Phasen des MapReduce

Wie die Grafik illustriert, gibt es in der Map Phase keine Datenabhängigkeiten der einzelnen Elemente (1:1 Mapping), weshalb jede Funktionsanwendung

sehr gut auf verschiedenen Maschinen ausgeführt werden kann. Die Map Phase gilt als abgeschlossen, wenn sämtliche Elemente aus der Menge A auf ein Element in der Menge B abgebildet wurden. Nun kann das MapReduce Framework auf einer Maschine die Reduce-Phase durchführen. Hierbei ist wichtig zu bemerken, dass die Reduce Funktion nicht äquivalent parallelisiert werden kann, da es Datenabhängigkeiten gibt. Eine parallele Verarbeitung ist nur möglich, wenn die Reduce Funktion assoziativ ist. In diesem Fall könnte die Reduce-Phase in einer Art Baum-Struktur ausgeführt werden – auf solche für das verständnis nicht förderlichen Details soll jedoch im folgenden nicht weiter eingegangen werden.

Hier ein Beispiel umd die Idee der zwei Phasen des MapReduce zu verdeutlichen. Wir definieren ein initiale Menge an Elementen

$$A = \{1, 2, 3, 4\}$$

eine Funktion für die Map Phase

$$f_m\, x = x^2$$

und eine Funktion für die Reduce Phase

$$f_r\, x\, a = x + a.$$

Nach der Map Phase entsteht die Menge

$$B = \{1, 4, 9, 16\}.$$

Welche in der der Reduce Phase aufaddiert[2] wird und als Endresultat die einelementige Menge

$$C = \{30\}$$

ergibt

[2] Der initiale Akkumulator ist in diesem Fall das erste Element aus der Menge B oder 0.

2 Teil 1 Gültigkeit des Satzes zur Rechtsauslöschung in Gruppen

2.1 Mathematische Grundlagen

2.1.1 Gruppe

Eine Gruppe ist eine algebraische Struktur bestehend aus einer Menge S und einer Verknüpfung $\diamond$, welche die folgenden Axiome erfüllt:

- Assoziativität: $\forall a, b, c.\ (a \diamond b) \diamond c\ = a \diamond (b \diamond c)$
- Neutrales Element: $\exists e.\, \forall a.\ (a \diamond e = e \diamond a = a)$
- Inverses Element[3]: $\forall a.\, \exists b.\ (a \diamond b = e)$

Es gibt verschiedene Strukturen, die diese Eigenschaften haben. Ein Beispiel davon ist die Restklasse 3 mit einer Operation definiert als: $x \diamond y = (x + y)\,\%\,3$, also die gewöhnliche Addition in dieser Restklasse. Für diese Struktur kann man nun beweisen, dass sie eine Gruppe ist, indem man sich überzeugt, dass die Addition assoziativ ist, dass ein neutrales Element existiert (nämlich 0) und dass jedes Element ein inverses Element hat, mit welchem es auf die 0 abbildet (z.B. $1 \diamond 2 = 0$).

Es gibt weit mehr Strukturen, die als Gruppe eingestuft werden können. Es ist zum Beispiel durch stupides ausprobieren, also mechanische Arbeit, die bestens für einen Computer geeignet ist, da sie strikt regelgesteuert abläuft, möglich diese zu finden. Genau dies soll der erste Schritt unserer Arbeit sein. Natürlich gibt es unendlich viele Möglichkeiten für Gruppen, daher ist es klar, dass wir mit unseren Berechnungen nur einen gewissen Bereich abdecken können. Jedoch ist dieser Bereich enorm klein, da bereits bei wenigen Elementen eine enorme Rechenleistung, welche sequenziell de facto unerreichbar wäre, erforderlich ist. Genau hier kommt unser MapReduce Framework zur paralellen Berechnung ins Spiel.

2.1.2 Satz

> Ein Satz oder Theorem ist in der Mathematik eine widerspruchsfreie logische Aussage, die mittels eines Beweises als wahr erkannt, das heißt, aus Axiomen und bereits bekannten Sätzen hergeleitet werden kann.[Sat]

Nachdem wir mittels des MapReduce Frameworks irgendwelche Strukturen gefunden haben, welche, eventuell per Zufall, die Axiome einer Gruppe erfüllen, können wir in einem zweiten Schritt noch prüfen, ob für diese auch bestimmte Sätze zutreffen. Im Gegensatz zu den Gruppen-Axiomen gibt es keine definierte Menge an Sätzen, welche für eine Gruppe gültig sind. Es werden ständig neue Sätze gefunden und ausserdem existieren Sätze, für die noch nicht allgemeingültig bewiesen werden konnte, dass sie für sämtliche Gruppen gelten, für die aber ebenso noch kein Gegenbespiel oder Gegebeweis gefunden werden konnte. Trotzdem wir mit den vorhandenen Mitteln keine allgemeingültigen Beweise

[3] Hierbei ist e das neutrale Element

führen können, ist es mit genügend Rechenleistung ein Ansatz nach Gegenbespielen, also bestimmte Konstellationen von Mengen und Verknüpfungen, welche zwar eine Gruppe sind, aber einen bestimmten Satz nicht erfüllen, zu suchen. Ein Beispiel eines Satzes, ist der Satz der Rechtsauslöschung:

$$\forall a, b, c.\, b \diamond a = c \diamond a \Rightarrow lb = c$$

Sämtliche Strukturen, welche die Axiome der Gruppe erfüllen, erfüllen auch diesen Satz. Den Beweis dazu findet man online[Can].

2.2 Generieren von Gruppen mittels Computer

Es gibt unendlich viele Mengen und darauf mögliche Verknüpfungen. In unserer Implementation kann bei Programmstart angegeben werden nach Mengen mit wie vielen Elementen gesucht werden soll[4]. Um Gruppen zu finden, haben wir einen Algorithmus implementiert, welcher alle möglichen zweistelligen Verknüpfungen auf einer endlichen Menge generiert. Grafisch können diese Verknüpfungen als Wertetabellen interpreiert werden. So wie bei diesen Beispielen für mögliche Verknüpfungen mit zwei Elementen:

$\diamond$	0	1
0	0	0
1	1	1

Tabelle 1: Intuitive Addition der Restklasse 2

$\diamond$	0	1
0	0	1
1	1	0

$\diamond$	0	1
0	1	1
1	1	1

Tabelle 2: Weitere mögliche Wertetabellen einer Verknüpfung

Insgesamt gibt es 16 mögliche Permutationen für Wertetabellen von zweistellige Verknüpfungen bei einer Menge aus 2 Elementen. Für eine Menge aus 3 Elementen sind es Bereits 19 683. Allgemein fomuliert besitzt eine Menge mit n-Elementen n^{n*n} mögliche Wertetabellen für zweistellige Verknüpfungen.

n	Mögliche Wertetabellen für zweistellige Verknüpfungen
1	1
2	16
3	19 683
4	4 294 967 296
5	298 023 223 876 953 125
6	10 314 424 798 490 535 546 171 949 056
7	256 923 577 521 058 878 088 611 477 224 235 621 321 607

Tabelle 3: Mögliche Permutationen einer zweistelligen Verknüpfung in einem n-Elementigen System

[4] Aus technischen Gründen war es jedoch nötig die mögliche Anzahl an Elementen auf 9 zu beschränken.

Weiterhin gibt es in jeder Gruppe ein für jedes Element ein inverses Element bezüglich der zweistelligen Verknüpfung. Um alle möglichen inversen Elemente zu finden, haben wir für diese einstellige Verknüpfung ebenfalls Wertetabellen generiert. Folgend zwei Beispiele für Wertetabellen mit zwei Elementen:

a	0	1
a^{-1}	1	0

a	0	1
a^{-1}	0	0

Tabelle 4: Beispiele für einstellige Verknüpfungen mit zwei Elementen

Für eine Gruppe aus zwei Elementen gibt es also 4 mögliche Wertetabellen und allgemein gibt es 2^n Möglichkeiten für einstellige Verknüpgungen aus Gruppen mit n-Elementen. Auch diese Tabellen können wir mit unserem Algorithmus simulieren.

Das neutrale Element ist anschliessend einfach zu finden, da es schlicht das Ergebnis der zweistelligen Verknüpfung eines Elementes mit seinem Inversen ist.

Schlussendlich suchen wir Gruppen indem wir eine zweistellige Verknüpfung, eine einstellige Verknüpfung und ein neutrales Element, wie im Kapitel „Prüfen der Axiome als Map-Funktion", zusammen auf die Axiome überprüfen.

2.2.1 Datenstruktur der Wertetabellen

Bereits für Mengen mit drei Elementen gibt es, wie oben gezeitgt, bei der zweistelligen Verknüpfung tausende von Möglichkeiten. Da wir die Prüfung der Axiome, spätestens für die Berechnung von Gruppen aus einer vier-Elementigen Menge, verteilen müssen, gilt es einerseits einen Algorithmus finden, der sich auf verschiedene Rechner aufteilen lässt, sodass jeder Rechner eine Teilmenge der Wertetabellen überprüfen kann, und andererseits diese Teilmengen so abzubilden, dass sie nicht zu viel Speicher verbrauchen. Auf einem Rechner alle Wertetabellen zu generieren und diese dann an die verschiedenen Rechner über das Netz zu schicken ist also nicht praktikabel.

Nehmen wir als Beispiel die Menge mit zwei Elementen. Bei genauerer Betrachtung einer Wertetablle sieht man, dass diese anstatt zweidimensional auch eindimensional interpretiert werden könnte

$\diamond$	0	1
0	a	b
1	c	d

Tabelle 5: Logischer Aufbau einer zweistelligen Verknüpfung für zwei Elemente

eindimensional könnte man die abgebildete Wertetabelle folgendermassen interpretieren:

$$\diamond = \{a, b, c, d\}$$

Diese Interpretation wurde in unserem Projekt zur Abbildung der Wertetabellen gewählt.

In der zweidimensionalen Wertetabelle können wir die Funktionswerte mithilfe ihrer x und y Indizes ablesen. Diese x und y Werte können mit der folgenden

Funktion auf den Index in der Binärrepräsentation abgebildet werden:

$$f_{trans2}(x, y) = y * 2 + x$$

Wäre man als Beispiel in der sechsten Wertetabelle an der Abbildung $1 \diamond 0$ interessiert, würde das $x = 1$, $y = 0$ entsprechen. Transformiert auf die eindimensionale Repräsentation ergibt dies die Binärrepräsentation 0110 und für den Index $f_{trans2}(1, 0) = 1$. Also das zweite Element der Binärrepräsentation, was eine eins ist - wir in der Wertetabelle.

Für eine Menge mit zwei Elementen kann man sich leicht überzeugen, dass sämtliche Kombinationen die Binärwerte der Zahlen zwischen 0 und 15 sind: z.B. $0 = \{0, 0, 0, 0\}$(alles bildet auf 0 ab) oder $6 = \{0, 1, 1, 0\}$ (die intuitive Addition der Restklasse 2).

2.2.2 Generieren der Wertetabellen

Für eine Menge mit zwei Elementen kann man sich leicht überzeugen, dass sämtliche Kombinationen die Binärwerte der Zahlen zwischen 0 und 15 sind: z.B. $0 = \{0, 0, 0, 0\}$(alles bildet auf 0 ab) oder $6 = \{0, 1, 1, 0\}$ (die intuitive Addition der Restklasse 2). Diese Anschauungsweise birgt den Vorteil, dass alle möglichen Wertetabellen durch die simple Addition von 1 im Binärsystem generiert werden können.

Diese Betrachtung kann auch auf das Ternärsystem usw. übertragen werden, indem man alle Zahlen von 0 bis $3^{3*3} - 1$mit der oben beschriebenen Funktion f_{trans2}interpretiert. Allgemein können also alle Wertetabellen zweistelliger Verknüpfungen somit durch einfache Addition der 1 als zahl in ihrem jeweils korrospondierenden Stellenwertsystem interpretiert werden.

2.2.3 Implementation von Quantoren in Java

Wie im Kapitel „Mathematische Grundlagen" beschrieben, werden Axiome einer Gruppe und Sätze einer beliebigen Struktur mittels Quantoren definiert. Als Beispiel soll das Axiom 'Es gibt ein neutrales Element' der Struktur Gruppe näher erläutert werden: $\exists e \, \forall a \; (a \diamond e = e \diamond a = a)$. Dieses Axiom verwendet zwei Quantoren: Einen Existenzquantor ($\exists$) und einen Allquantor ($\forall$). Mit dem Existenzquantor wird ausgesagt, dass es in einer bestimmten Struktur ein e gibt, für das das folgende Prädikat wahr ist, während mit dem Allquantor gesagt wird, dass das Prädikat für alle Elemente dieser Struktur gelten muss.

In unserem Beispiel 'Es gibt ein neutrales Element' tauchen diese beiden Quantoren direkt nacheinander auf, wodurch dieses Axiom in Prosa folgendermassen formuliert werden kann:

> In einer Gruppe gibt es ein Element e, welches, wenn es mit irgendeinem Element a aus dieser Struktur verknüpft wird, immer auf das Element a abbildet. Dabei ist es gleichgültig, ob a mit e verknüpft wird oder e mit a - es muss immer auf a abbilden. Genau dann, wenn es ein solches Element e gibt, dann ist e das neutrale Element dieser Struktur und das Axiom ist erfüllt.

Da es in Java keine Quantoren gibt, mussten wir Schleifen nutzen, um die Axiome zu implementieren. Mittels der folgenden Methode wird für ein bestimmtes

Element e geprüft, ob es das neutrale Element zweistellligen Verknüpfung für eine Gruppe mit n-Elementen ist.

```
1 boolean isNeutral(int numEle, int aPerm, int e) {
2   for (int a = 0; a < numEle; a++) {
3     int b = map2d(a, e, aPerm, numEle);
4     if (a != b) { return false; }
5     int c = map2d(e, a, aPerm, numEle);
6     if (a != c) { return false; }
7   }
8   return true;
9 }
```

Abbildung 2: Beispielimplementation von Quantoren

Diese Methode besitzt drei Parameter:

1. Ein Wert vom Typ Integer, also Ganzzahl, welcher die Anzahl an Elementen der zu prüfenden Gruppe repräsentiert.
2. Der Identifikator für die zweistellige Wertetabelle, welcher auch als Ganzzahl modelliert ist.
3. Das vermeindlich neutrale Element, für welches diese Funktion prüft, ob es wirklich das neutrale Element ist.

In der zweiten Zeile wird durch alle Elemente der Menge iteriert, wodurch die Variable a den Wert jedes Elementes einmal annimmt (beginnend bei 0).

In der dritten Zeile wird die Methode map2d aufgerufen, welche das Element a (also jedes Element der Menge einmal) mit dem neutralen Element e verknüpft. Die Methode *map2d* implementiert den von uns entwickelten Algorithmus zur Generierung der Verknüpfungstabellen (vgl. Generieren der Wertetabellen). Der Rückgabewert dieser Funktion wird der Variable b zugewiesen um in der nächsten Zeile mit der Variable a verglichen zu werden. Wenn diese Bedingung zutrifft, bedeutet das, dass $a \diamond e = e$ gilt, also ist das e bereits linksneutral. Falls e es nicht linksneutral ist, wird die Funktion sofort abgebrochen, da das Prädikat einen Allquantor beinhaltet und es somit für alle Elemente gelten muss.

Auf den folgenden Zeilen fünf und sechs wird die Variable a nochmals mit dem vermeindlich neutralen Element verknüft um die Rechtsneutralität zu überprüfen (daher sind hier die Parameter a und$e$ für die Funktion map2d vertauscht).

Wenn sämtliche Elemente dieser Gruppe links- und rechtsneutral sind, wird die Schleife verlassen und die Funktion erreicht die Zeile Nr.8, in der der boolsche Wert 'wahr' zurückgegeben wird. Dies bedeutet, dass das Element e tatsächlich das neutrale Element in dieser Struktur ist.

2.3 Map und Reduce Funktionen für die Aufgabenstellung

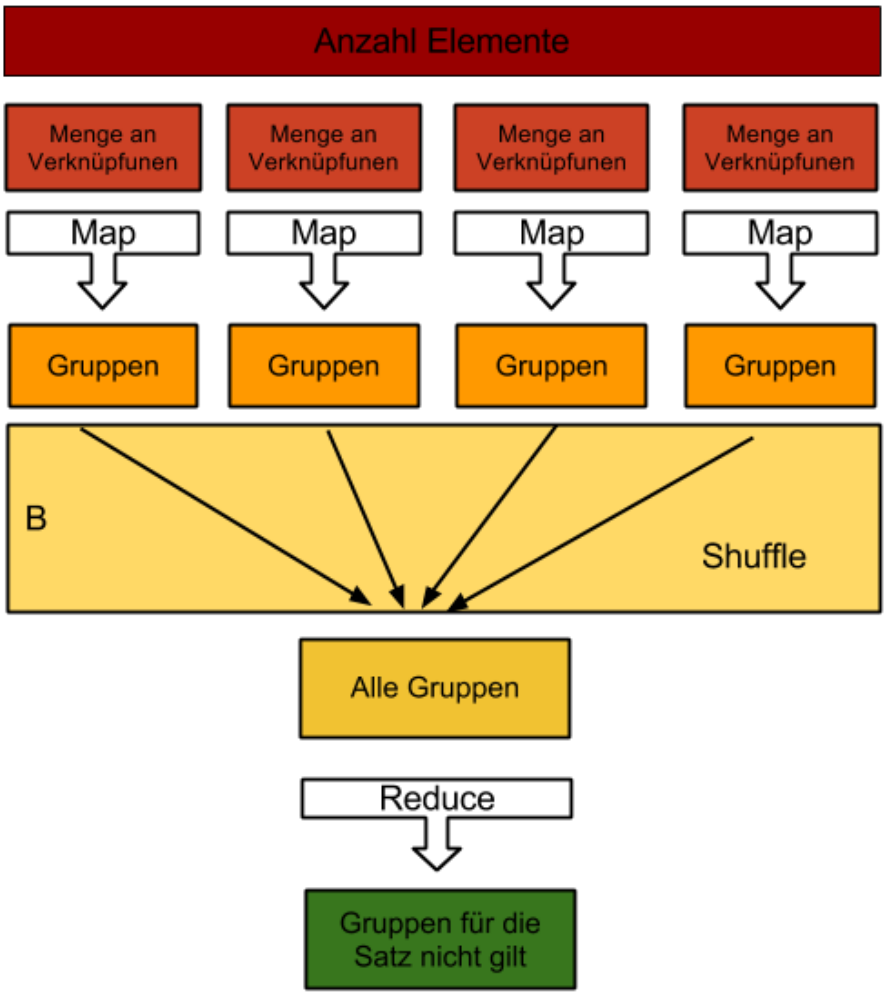


Abbildung 3: Überprüfen des Satzes als Map und Reduce Funktionen

2.3.1 Prüfen der Axiome als Map-Funktion

Das CPU-intensive überprüfen der Strukturen, welche möglicherweise die Axiome erfüllen, wurde als MAP-Funktion definiert. Dabei werden den einzelnen rechnenden Einheiten, im folgenden Worker genannt, jeweils eine Anzahl an Permutationen der Additionstabelle und ein n im Sinne der derzeit vorhandenen Anzahl an verschiedenen Elementen als Input übergeben. Die Worker führen dann die Überprüfung der Axiome durch, indem sie jeweils für alle Additionstabellen in ihrem Bereich alle möglichen Inversentabellen mit allen möglichen neutralen Elementen prüfen.

Hierbei gibt es Vereinfachungen, die inhaltlich korrekt sind jedoch die benötigte Rechenleistungen massiv reduzieren:

- Sowohl die Additions- wie auch die Permutationstabelle werden nicht immer vollständig vor der Prüfung der Axiome aufgestellt, sondern nur bei Bedarf bei Bedarf durch die oben beschriebene Algorithmus (2.2.2 Generieren der Wertetabellen) generiert. Das heisst der Rechenaufwand wird substantiell reduziert da bei einer Prüfung der Axiome im Normalfall nur ein quasi vernachlässigbarer Teil der Additionstabelle und, abhängig von Erfolg oder Misserfolg der Überprüfungen, auch nur ein minimer Teil der Inversentabelle benötigt wird.
- Für eine gegebene Additions- und Inversen-Tabellenkombination wird jeweils nur das neutrale Element, das für den ersten Eintrag in der Inversentabelle berechnet wurde, geprüft. Da per Definition nur ein neutrales Element für eine spezifische Kombination aus Tabellen existiert, muss es auch für alle weiteren Einträge an Inversentabellen gelten. Sobald ein Eintrag gefunden wurde für den ein anderes neutrales Element gelten würde, wird die Berechnung für die vorliegenden Tabellen verworfen Auch diese Annahme reduziert substantiell die durchgeführten, nicht zielführenden Rechnungen.

Die Ausgabe der Map-Funktion sind schlussendlich alle Gruppen auf die die gegebenen Axiome zutreffen. Eine Gruppe wird ausreichend durch 3 Zahlen beschrieben:

- Die eindimensionale Repräsentation der Additionstabelle als String.
- Die indimensionale Repräsentation der Inversentabelle als String.
- Das neutrale Element der Gruppe.

Diese effiziente Speicherung ermöglicht das Vorhalten von enorm vielen Gruppen ohne einen hohen Speicherbedarf.

2.3.2 Prüfen der Sätze als Reduce-Funktion

In der Reduce-Phase wird für die, relativ zur Menge der möglichen Gruppen, enorm kleine Menge an effektiven Gruppen der Satz jeweils geprüft. Als Output gibt sie die Gruppen für die der Satz nicht zutrifft. Da anzunehmen ist, dass somit der Output immer leer ist, kann er auch als booleanscher Wahrheitswert interpretiert werden:

- Wenn der Output leer ist, ist der Satz für die geprüften Gruppen wahr.
- Wenn der Output nicht leer ist, ist der Satz für die geprüften Gruppen falsch (was einer mathematischen Sensation gleich käme).

3 Weiteres Vorgehen

3.1 Möglichkeiten der Vertiefung der Aufgabenstellung

Wie bereits geschildert, wurde uns die genaue Vertiefung der Aufgabenstellung selbst überlassen. Für uns kamen drei mögliche Schwerpunkte in Frage auf die zuerst einmal näher erläutert werden sollen.

3.1.1 Erweiterung des MapReduce Frameworks

Bei diesem Schwerpunkt wäre die Zielsetzung eine möglichst grosse Kapazität Gültigkeiten zu beweisen mit dem MapReduce Framework erweitert um die Möglichkeit von auf mehreren Rechnern verteilten Berechnungen zu erzielen. Die enormen Datenmengen, die bereits bei kleinen Eingabewerten entstehen sind auf einem einzelnen Rechner spätestens ab Gruppen mit 4 Elementen quasi nicht zu bewältigen hierbei ist primär die CPU-Rechenzeit relevant. Es müssten daher Möglichkeiten gefunden werden, um eine Menge an Inputwerten zu anderen Computern zu senden, dort berechnen zu lassen und für das weitere Vorgehen relevante Daten zu returnieren.

Anforderungen bei diesem Schwerpunkt wären primär die Handhabung von verteilten Systemen, Design der Applikation und einer performanten Umsetzung.

3.1.2 Freie Eingabe von auf Gültigkeit zu prüfenden Annahmen

Hierbei läge das Hauptaugenmerk darauf einer Eingabesprache um ein Programm zum automatisierten Gültigkeitsbeweis zu erstellen. Es müsste eine Möglichkeit gefunden werden dem Programm mathematische Strukturen mit bestimmten Eigenschaften, darauf gültige Axiome und daraus resultierend zu prüfende Sätze zu übermitteln. Dazu müsste, gleich ob interaktiv oder durch die Auswertung von Eingabedateien, eine Eingabesyntax definiert werden und, was wohl die wirkliche Problemstellung des Schwerpunktes wäre, eine Auflösung der Syntax und eine darauf basierende dynamische Umsezuung in ausführbaren Code.

Anforderungen bei diesem Schwerpunkt lägen primär in der theoretischen Informatik beim Thema Compilerbau, der dynamischen Generierung einer Interpretation der Mathematischen Objekte und schlussendlich ihrem Einbinden sowie Ausführen in eine Applikation zur Gültigkeitsprüfung.

3.1.3 Automatisierte Beweisbarkeit

In diesem Fall wäre das Ziel ein Framework zu erstellen welches mit einem gegebenen Set aus mathematischen Operationen (zugelassenen Schlussregeln) automatisch Sätze aus Eingaben (Sätzen und Axiomen) herleitet. Die Operationen, die das System vornehmen könnten entsprächen den im Kalkül zugelassenen Schlussregeln. Ziel wäre es somit automatisiert mehr oder weniger sinnvolle Aussagen mathematisch formell korrekt herzuleiten. Dabei müssten Kriterien definiert werden mit denen die Unterscheidung von formal korrekten aber irrelevanten und formal korrekten, relevanten Aussagen möglich macht.

Anforderungen bei diesem Schwerpunkt wäre wieder die Interpretation der in einer gegebenen Syntax vorliegenden Eingaben jedoch des weiteren das Erkennen von neuen relevanten Schlussfolgerungen.

3.2 Entscheidung

Der auf Anhieb spannendste Schwerpunkt stellte nach der ersten Bertrachtung die „Freie Eingabe von auf Gültigkeit zu prüfenden Annahmen" dar. Die in dem Schwerpunkt vorkommenden Themen wurden weitestgehend im bisherigen Studium noch nicht behandelt und böten somit eine gute Möglichkeit neues kennenzulernen.

Die Automatisierte Beweisbarkeit zeigte sich bereits bei der ersten Betrachtung als eine eher unwahrscheinliche Vertiefung, da der schlussendliche Nutzen massiv in Frage gestellt werdern kann. Es wäre zu befürchten, dass eine enorme Menge an Arbeitszeit und Herzblut investiert würde um Schlussendlich eine Applikation zu erstellen welche Mathematisch völlig irrelevante Sätze herleiten würde.

Eine Erweiterung des MapReduce Framework erschien hingegen bereits nach der ersten Analyse als überschaubares Problem im Sinne der Beherrschbarkeit der Problemstellung. Zwar ist der mit der Vertiefung verbundene Aufand alles andere als als klein zu erachten, jedoch sollten keine vollkommen unvorhergesehenen Fragestellungen, zu denen ohne eine sehr grosse Menge an Vertiefung keine Lösung gefunden werden könnte, auftreten.

Das Kriterium der Beherrschbarkeit gab schlussendlich auch den Ausschlag anstatt der Vertiefung Nr. 2 eine Erweiterung des Frameworkes anzustreben. Die Gefahr durch die enorme Komplexität am Ende nur eine mehr prototypisch als sinnvolle Implementation eines Programmes zu erstellen schreckte uns ab. Desweiteren stellte die Idee mit der gegebenen Infrastruktur der Fachhochschule eventuell eine grosse Menge an PCs für einen Feldversuch zu besitzen einen grossen Ansporn dar.

4 Teil 2 MapReduce Framework zur verteilten Berechnung

4.1 Technologie

In Absprache mit unserem betreuenden Dozenten Herrn Albert Heuberger soll an dieser Stelle nur sehr Abstrakt auf die Implementation eingegangen werden. Um einen groben Überblick zu gewähren, sind die nun folgende Buzzwords leider unumgänglich.

Als Programmiersprache wählten wir, aufgrund der für den Aufbau eines verteilten Rechnernetzes sehr vorteilhaften Betriebssystemunabhängigkeit und unserer bereits vorhandenen Kenntnisse, Java aus. Git [GIT] wurde als Versionsverwaltungstool eingesetzt und Bitbucket [Bit] zum Austausch des Source verwendet. Für Build Management wurde Maven [MVN] eingesetzt, zum Tracking der Testcoverage Cobertura [Cob], für Dependency Injection Guice [GUI]und für das essentielle Logging die Standardmässige Java Logging API.

Auf ein GUI wurde mangels Notwendigkeit weitestgehend verzichtet. Die Applikation und das Framework lassen sich über Property-Files steuern. In einem solchen Property File können einzelne Key-Value Paare angegeben werden um bestimmte Aspekte der Berechnung oder Verhalten des Frameworks zu definieren. Mögliche Konfigurationen sind im Anhang im Kapitel „Property Files" einzusehen. Auf die genaue Konfigurationsparameter soll an dieser Stelle nicht eingegangen werden, da sie aus den Kommentaren ersichtlich sein sollten.

Als Ausgabe der Applikation werden, neben einem Log mit variablem Detailierungsgerad, HTML-Files mit den gefundenen Gruppen genutzt. Beispiele dafür sind ebenfalls im Anhang zu finden.

Das Framework besitzt 8800 Lines of Code und die Applikation zum Gültigkeitsbeweis 1672[5]. Es wurden insgesamt 38 Testklassen mit 149 Verschiedenen Testfällen erstellt und somit in den Kernkomponenten zur verteilen Berechnung eine Testcoverage von knapp 90% erreicht.

4.2 Struktur des Frameworks

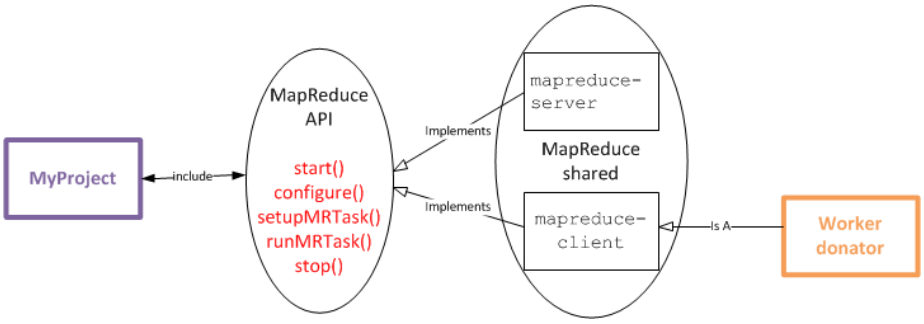


Abbildung 4: Struktur des Frameworks

Das Framework ist in vier Teile unterteilt:

[5]Code in Java Files, ohne Leerzeilen

1. Ein API das von Projekten, die das Framework einbinden wollen genutzt wird.
2. Gemeinsame Komponenten von Client und Server
3. Der MapReduce Client, der Rechenleistung für das Framework zur Verfügung stellen kann
4. Der zentrale MapReduce Server, der die Berechnungen verteilt etc.

Ein Projekt, das das Framework nutzen will ist hierbei völlig unabhängig von Client und Server, da es ausschliesslich die in der API befindlichen Interfaces und Klassen einbinden muss. Die Konkreten Umsetzungen des API befinden sich davon losgelösst in den Client und Server Projekten, von denen eines dem Build Path des Anwendungsprojektes hinzugefügt werden muss[6]. Es ist auch möglich den Client alleinstehend zu starten um einer anderen Berechnung Rechenkapazität zur Verfügung zu stellen.

4.3 Übersicht

Das Herz unserer MapReduce Implementation ist der Master, er Verwaltet die Aufgaben (Tasks) und überwacht ihre Ausführung. Ein Task ist eine abstrakte Einheit, welche von einem Worker erfüllt werden kann, hier also entweder ein Teil einer Map- oder einer Reduce-Berechnung. Im klassischen MapReduce, wie es von Google beschrieben wurde, ist ein Worker eine Maschine (PC). Somit verteilt eine Hauptmaschine (Master) die verschiedenen Aufgaben an verschiedene Rechner in einem Netzwerk.

Davon abweichend ist in unserer Implementation ein Worker nicht zwingend eine andere Maschine, sondern schlicht eine abstrakte Einheit, die Arbeit ausführen kann. In der ersten Version unseres Frameworks gab es, mit lokalen Threads auf Maschine des Master, nur eine Art von Workern.

Im Rahmen dieser Projektarbeit wurde die Möglichkeit geschaffen via Plugin irgendeine Art von Worker dem Master zu übergeben. Somit gibt es also zum Beispiel ein Plugin, welches Threads als Worker zur Verfügung stellt und ein Plugin, welches andere Rechner über eine Netzwerk-Verbindung zur Verfügung stellt. Dies wurde soweit abstrahiert, dass in Zukunft auch weitere Arten von Workern erstellt werden könnten. Denkbar wären dabei zum Beispiel CUDA (NVidia Grafikkarten) oder Android Smartphones.

Die konkrete Verwaltung der Worker wird vom sogenannten Pool übernommen, dieser kennt den Status seiner Worker (z.B. beschäftigt, frei, unauffindbar, etc) und startet die Berechnung einzelner Tasks auf ihnen.

[6] Leider ist es mit dem derzeitigen Stand noch nicht möglich, dass ein Projekt nur auf einem Client ausgeführt wird, welcher die Instruktionen etc an den Server sendet.

4.4 Ablauf

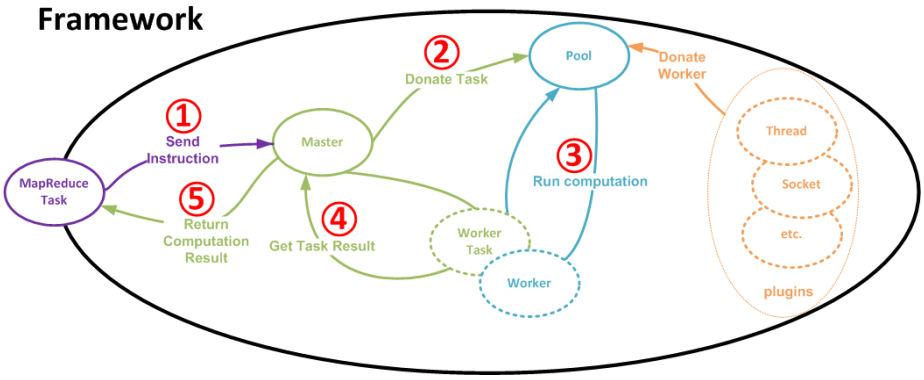


Abbildung 5: Ablauf einer Berechnung in MapReduce Framework

Wie im Kapitel „MapReduce" beschrieben muss der Benutzer nur eine Map- und eine Reduce Funktion zur Nutzung des Frameworks erstellen. In unserer Implementation geschieht dies durch von Nutzer erstellte Implementationen der beiden Interfaces *MapInstruction* und *ReduceInstruction*. Diese werden in einer MapReduceTask zusammengefasst und anschliessend mit einer Menge an Input dem Framework übergeben **(1)**.

Daraus kann dann der Master die Map- und Reduce Instruktionen extrahieren und erstellt zunächst aus je einem Teil vom Input und einer Instanz der Map-Instruction einen MapTask. Folglich entstehen für eine MapReduce Berechnung viele MapTasks, welche dem Pool zur Verarbeitung übergeben werden **(2)**. Hier wird vorausgesetzt, dass dem Pool durch ein oder mehrere Plugins bereits Worker zur Verfügung gestellt worden sind, da die Tasks sonst niemals ausgeführt werden könnten.

Nun kann der Pool einen auszuführenden MapTask einem verfügbaren Worker übergeben **(3)**. Wenn dieser Worker eine andere Maschine im Netz abstrahiert, würde das bedeuten, dass dieser MapTask auf einem anderen Rechner ausgeführt wird.

Sobald die Berechnung (z.B. auf einem anderen Rechner) ausgeführt wurde, meldet sich der Worker wieder beim Pool, sodass bereit ist weitere Aufgaben anzunehmen. Also könnte eine ganze Berechnung entweder von einem einzelnen Worker in mehreren Schritten oder auf mehrere Worker aufgeteilt ausgeführt werden.

Gleichzeitig prüft der Master kontinuierlich, ob alle MapTasks ausgeführt worden sind. Sobald alle MapTasks für eine Berechnung ausgeführt worden sind, beginnt der Master alle Resultate zu sammeln **(4)** und schliesst somit die Map-Phase ab.

Nach der Map Phase folgt, wie im Kapitel „1.4" beschrieben, die Shuffle-Phase, die die Resultate der Map-Phase nach einem Kriterium gruppiert und somit die Reduce-Phase vorbereitet. Es ist möglich, dass der User eine von ihm definierte Shuffle Funktion angibt um die einfache Gruppierung nach Schlüsseln zu umgehen. Da vor der Shuffle-Phase alle MapTasks ausgeführt werden sein müssen, gilt dies als Bottleneck von MapReduce. Nach der Shuffle-Phase beginnt

der Master damit die einzelnen ReduceTasks, bestehend aus einer Instanz der ReduceInstruction und einem Teil der Shuffle Resultate, zu erstellen und diese wieder dem Pool zu übergeben.

Pool beginnt wiederum diese ReduceTasks den verfügbaren Worker zuzuteilen während der Master wieder auf deren Fertigstellung wartet. Sobald auch die ReduceTasks berechnet und ihre Ergebnisse abgeholt sind, ist die gesammte MapReduce Berechnung abgeschlossen und das Ergebnis wird zurückgegeben **(5)**.

4.5 Ablauf bei verteilter Berechnung

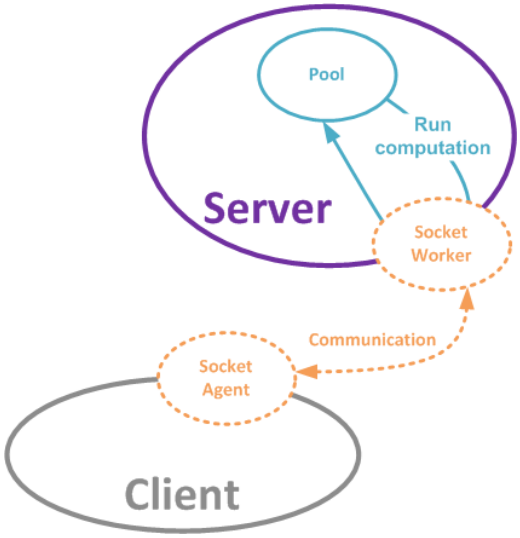


Abbildung 6: Interaktion von Client und Server

Für das Ausführen von verteilen Berechnungen verwendeten wir das Framework SIMON [SIM]. Mit Hilfe dieses Frameworks ist es möglch lokale Methoden Aufrufe auf einer entfernten Instanz an einem beliebigen Ort im Netzwerk auszuführen. Damit sich der Master nicht um die Komplexitäten im Zusammenhang mit Remote Procedure Call (RPC) [RPC] kümmern muss, wird ein einzelner Client, also eine entfernte Instanz, auf dem Server genauso wie ein gewöhnlicher Thread als Worker im Pool registriert. Die konkrete Implementation dieses Workers ruft aber, anstatt den Task direkt auszuführen, via RPC eine Methode auf dem Client auf und übermittelt ihm die Instruktion mit zugehörigen Eingabewerten.

Auf dem Client wartet ein sogenannter SocketAgent ständig darauf, dass er Tasks vom Server zugeordnet bekommt. Sobald ein Task angekommen ist, wird dieser ausgeführt und die Resultate werden wieder an den SocketWorker geschickt, welcher auf dem Server ist. Der SocketWorker kümmert sich dann um das korrekte Speichern der Resultate und meldet dem Pool, dass er wieder Arbeit aufnehmen kann.

4.6 Speichern von Daten

Die Speicherung der Zwischenergebnisse der einzelnen Berechnungen stellt, gerade unter dem Aspekt der verteilten Berechnung einen Spannenden Aspekt der Implementation dar. Die Rahmenbedingungen sind, dass für den Benutzer die Speicherung in den einzelnen Instructions immer gleich erscheinen soll, für den Master eine schnelle Rückgabe, wenn er die Zwischenergebnisse von einem Task erwartet, relevant ist und für Grosse Datenmengen ein Medium gewählt werden muss. Diese wiederstrebenden Anforderungen sind wieder mit einer Abstraktion gelöst worden: Dem sogenannten Kontext und der Persistenz.

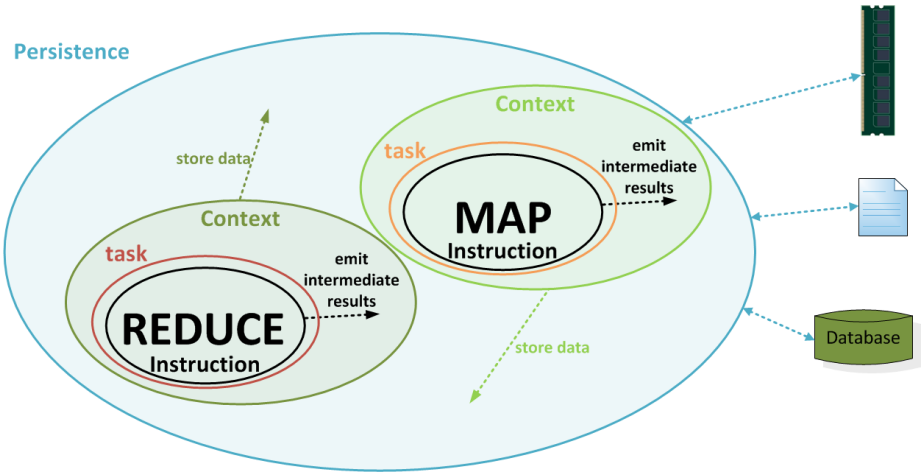


Abbildung 7: Speicherkonzept des Frameworks

Zur Speicherung von Zwischenergebnissen werden den einzelnen Instructions Kontexte durch die Tasks in denen sie gekapselt sind, übergeben. Somit ist es aus Sicht des Tasks und der Instruction nicht transparent und nicht relevant, wie gespeichert werden. Diese weitere Abstraktion hilft auch enorm bei der Abstraktion des Workers, da dieser mit dem Speicherkonzept ansich nicht tangiert wird.

Die Kontexte sind an die Existenz der einzelnen Tasks gekoppelt und können von Ihnen ausgelesen werden daher müssen sie dem Master für seine Abfrage der Ergebisse nicht bekannt sein. Ausserdem kann eine Task an einem beliebigen Ort, der ihm einen Kontext bietet, so auch auf einer anderen Maschine, ausgeführt werden. Intern speichern die Kontexte wiederum ihre Daten in eine wie auch immer geartete Persistierung der Daten.

Eine Persistenz kümmert sich um die tatsächliche Speicherung und das Auslesen der Daten um auch diese Aufgabe zu abstrahieren. Daraus folgt, dass zum Beispiel der Master und ein Client für verteilte Berechnungen ihre Daten auf unterschiedliche Arten, je nach beschaffenheit des Clients, speichern.

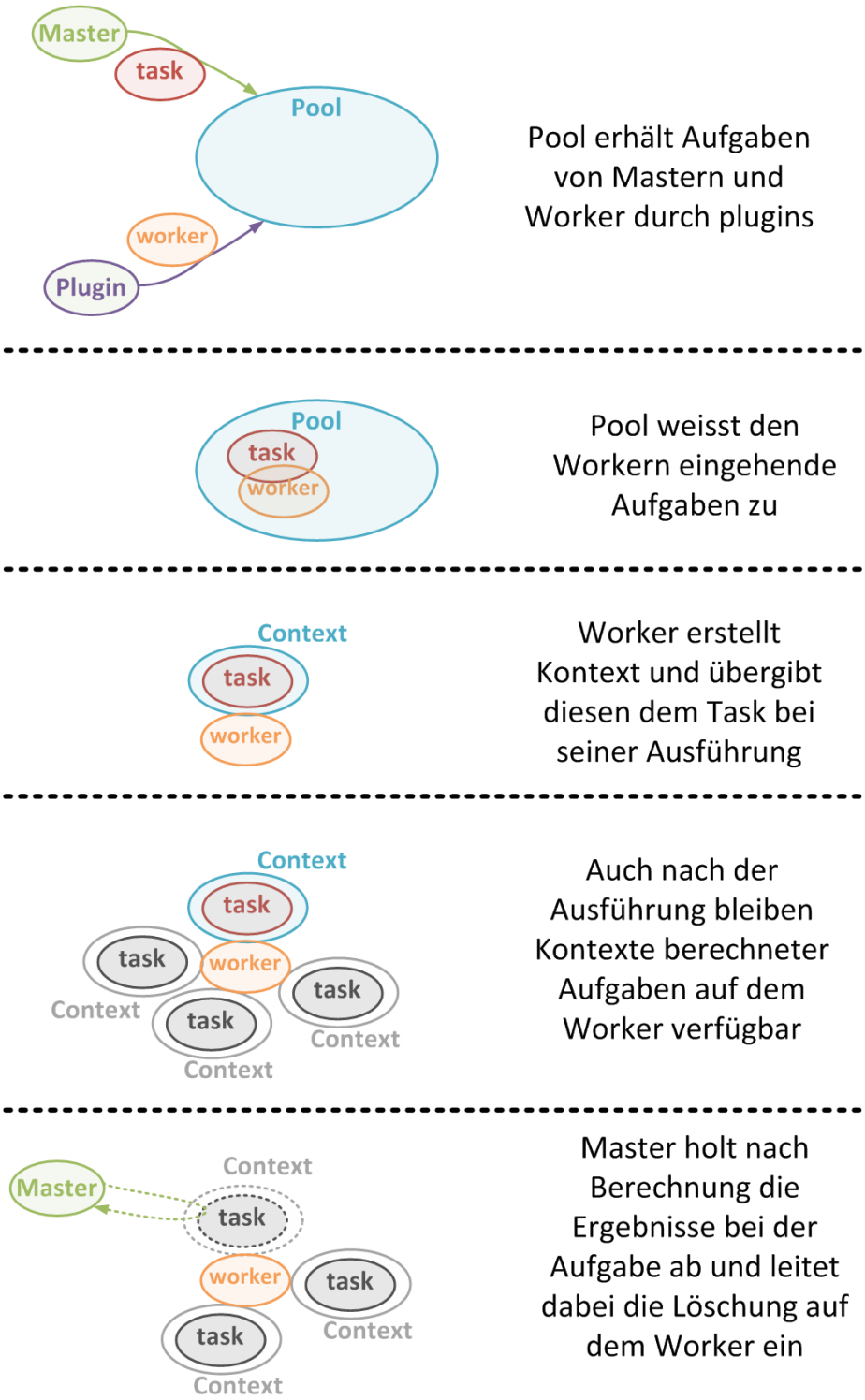


Abbildung 8: Interaktion von Task, Pool, Worker und Kontext

5 Ergebnisse

5.1 Anwendung der Applikation

Wie bereits im Kapitel „Technologie" erwähnt, wird die Applikation sowie das Framework primär durch Property Files gesteuert. In einem solchen Property File können einzelne Key-Value Paare angegeben wereden um bestimmte Aspekte der Berechnung oder Verhalten des Frameworks zu definieren. Mögliche Konfigurationen sind im Anhang im Kapitel „Property Files" einzusehen. Auf die genaue Konfigurationsparameter soll an dieser Stelle nicht eingegangen werden, da sie aus den Kommentaren ersichtlich sein sollten.

5.2 Durchführung

5.3 Auswertung der gefundenen Gruppen

5.4 Schlusswort

6 Apendix

6.1 Property Files

6.1.1 Applikation

```
# Einstellungen der Applikation
#---------------------------------------------------------------
# Pfad, an dem die Output Files gespeichert werden
path=/tmp/hsThesisValidator/

# Wert, bei dem die Berechnung startet
start=1

# Wert, bei dem die Berechnung stoppt. wenn der wert <= 0 ist, wird die Berechnung endlos ausgefuehrt
stop=-1

# Ein die Groesse der einzelnen zu berechnenden Teile
permoffset=5000
```

6.1.2 Framework

```
# Einstellungen des Frameworks
#---------------------------------------------------------------
# 10000L Milisekunden => 10 Sekunden
# 60'000 = 1 Min
# Intervall in Millisekunden in dem Statistiken geladen werden
statisticsPrinterTimeout=10000

# Ein Master hat eine Liste an Tasks, die gerade ausgefuehrt werden. Mit diesem Parameter wird die Groesse dieser Liste beschraenkt. Wenn diese Groesse erreicht worden ist, wartet der Master, bis einige Tasks beenedet wurden, bevor er neue erstellt. Wenn Tasks dieser Liste fehlschlagen, werden sie neu gestartet, bevor komplett neue Tasks erstellt werden.
MaxRunningTasks=10000

# Das Basisverzeichnis, mit dem die FilePersistence arbeitet. Muss beschreibbar sein!
filepersistence.directory=/tmp/filepers

# Durch Kommata getrennte Namen der Plugins, die durch den Loader geladen werden sollen. Moegliche Werte: Socket,Thread
plugins=Socket

#---------------------------------------------------------------
# Thread Plugin

# Anzahl an Threads die als Worker zur Verfuegung gestellt werden wenn dieses Attribut nicht gesetzt ist, werden Kerne +1 verwendet
nThreadWorkers=100

#---------------------------------------------------------------
# Socket Plugin

# Wie lange der Resultat-Status von einem verfuegbaren Resultat in der Liste vom SocketResultCollector gehalten werden soll. Dies passiert, wenn der Agent ein Resultat hat und den SocketWorker
```

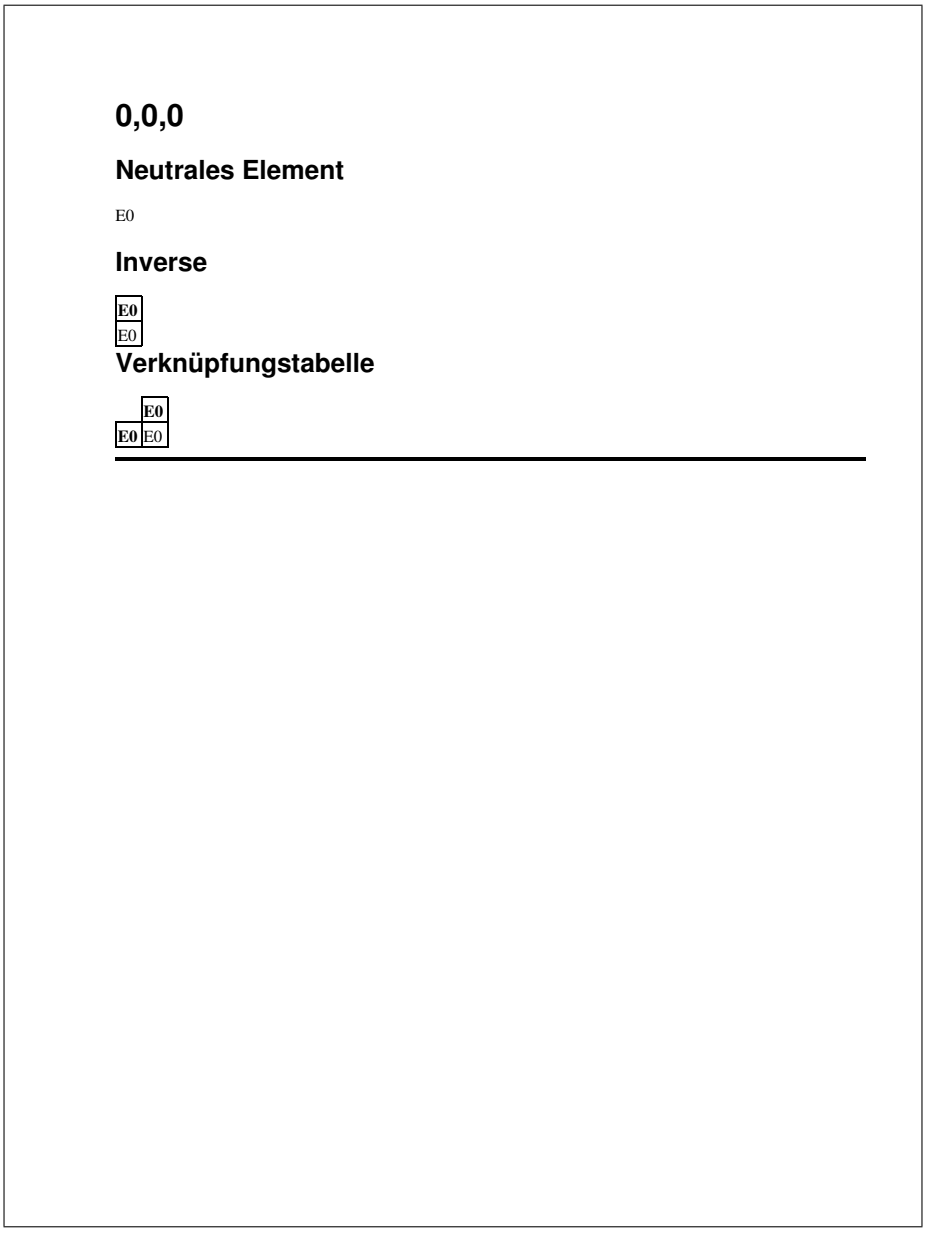
```
    darueber informiert. Typischerweise muesste dies der
    SocketWorker sofort akzeptieren, wordurch der Eintrag aus der
    Liste geloescht werden kann. (Der SocketWorker hat ja sonst
    nichts zu tun).
AvailableResultTimeToLive=10000

# Wie lange ein Eintrag von einem SocketWorker in der Liste der
    Resultat-Stati beim SocketResultCollector bleiben soll. Dieser
    Eintrag wird gemacht, sobald der SocketWorker einen Task dem
    Agent uebergibt. Dann registriert er sich naemlich beim
    SocketResultCollector, dass er an diesem Resultat interessiert
    ist. Typischerweise existiert also ein solche Eintrag ueber die
     ganze Dauer, die ein Task auf dem Agent ist - also etwas
    laenger als die Laufzeit eines Task.
RequestedResultTimeToLive=600000

# Der ResultCleanerTask geht periodisch ueber die Liste der Result-
    State und prueft, ob es veraltete Eintraege hat, fuer die
    entweder nie ein Resultat vom SocketAgent angekommen ist oder
    der SocketWorker ein angekommenes Resultat nie abgeholt hat.
SocketResultCleanupSchedulingDelay=60000

# Die Map-,Reduce- und Combiner Instruction werden serialisiert als
     Byte-Code zum Agent geschickt. Auf dem Server wird ein Cache
    verwendet, dass die Serialisierung nicht fuer jeden Task erneut
     durchgefuehrt werden muss. Dies ist ein LRU Cache und dieser
    Parameter bestimmt die Anzahl Eintraege im Cache, bevor der
    aelteste geloescht wird.
ObjectByteCacheSize=30

# Zeit, die gewartet wird, bevor der Task vom Agent als 'nicht-
    akzeptiert' klassifiziert wird. Dies koennte passieren, wenn
    der Agent nicht auf auf die Anfrage eines Task reagiert - ist
    aber sehr unwahrscheinlich, da die Verbindung SocketWorker-
    SocketAgent 1:1 ist. Wir brauchen das Timeout aber trotzdem,
    weil er sonst unter umstaenden ewig hangen koennte.
AgentTaskTriggeringTimeout=2000

# Jeder Agent wird periodisch gepingt, um zu schauen, ob er noch
    existiert. Falls er nicht mehr existiert, wird er vom Pool
    genommen und muesste sich ggf. selbststaendig neu anmelden.
    Dieser Parameter bestimmt die Zeitspanne in Millisekunden
    zwischen zwei Pings an den selben Agent.
AgentPingerDelay=10000

SocketSchedulerPoolSize=1

# Auf diesem Port wird die Registry auf Server/Masterseite
    gestartet. Dies ist der Standard IANA Simon-Port.
simonport=4753
```

6.2 Beispiel Output

```
02:57:02:709 HSThesisValidator - Software Projekt 2 - ZHAW FS 2013
    - Reto Habluetzel & Max Schrimpf
02:57:05:824 ----------------------------------------------------
02:57:05:825 Anzahl Elemente: 1
02:57:05:826 Untersuchte Permutationen: 1
02:57:05:826 Benoetigte Zeit ~ 00:00.03
02:57:05:826 Satz war in allen Faellen gueltig
02:57:06:736 ----------------------------------------------------
02:57:06:737 Anzahl Elemente: 2
02:57:06:737 Untersuchte Permutationen: 16
02:57:06:738 Benoetigte Zeit ~ 00:00.03
02:57:06:738 Satz war in allen Faellen gueltig
02:57:07:705 ----------------------------------------------------
02:57:07:705 Anzahl Elemente: 3
02:57:07:706 Untersuchte Permutationen: 19683
02:57:07:706 Benoetigte Zeit ~ 00:00.05
02:57:07:706 Satz war in allen Faellen gueltig
04:06:17:637 ----------------------------------------------------
04:06:17:638 Anzahl Elemente: 4
04:06:17:638 Untersuchte Permutationen: 4294967296
04:06:17:638 Benoetigte Zeit ~ 01:09.09
04:06:17:639 Satz war in allen Faellen gueltig
```

6.3 Gefundene Gruppen

6.3.1 Ein Element



6.3.2 Zwei Elemente

6,1,0

Neutrales Element

E0

Inverse

E0	E1
E0	E1

Verknüpfungstabelle

	E0	E1
E0	E0	E1
E1	E1	E0

9,1,1

Neutrales Element

E1

Inverse

E0	E1
E0	E1

Verknüpfungstabelle

	E0	E1
E0	E1	E0
E1	E0	E1

6.3.3 Drei Elemente

11453,11,2

Neutrales Element

E2

Inverse

E0	E1	E2
E1	E0	E2

Verknüpfungstabelle

	E0	E1	E2
E0	E1	E2	E0
E1	E2	E0	E1
E2	E0	E1	E2

14001,21,1

Neutrales Element

E1

Inverse

E0	E1	E2
E2	E1	E0

Verknüpfungstabelle

	E0	E1	E2
E0	E2	E0	E1
E1	E0	E1	E2
E2	E1	E2	E0

4069,7,0

Neutrales Element

E0

Inverse

E0	E1	E2
E0	E2	E1

Verknüpfungstabelle

	E0	E1	E2
E0	E0	E1	E2
E1	E1	E2	E0
E2	E2	E0	E1

6.3.4 Vier Elemente

3836825115,27,3

Neutrales Element

E3

Inverse

E0	E1	E2	E3
E0	E1	E2	E3

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E3	E2	E1	E0
E1	E2	E3	E0	E1
E2	E1	E0	E3	E2
E3	E0	E1	E2	E3

3834475035,39,3

Neutrales Element

E3

Inverse

E0	**E1**	**E2**	**E3**
E0	E2	E1	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E3	E2	E1	E0
E1	E2	E0	E3	E1
E2	E1	E3	E0	E2
E3	E0	E1	E2	E3

3786677070,75,2

Neutrales Element

E2

Inverse

E0	**E1**	**E2**	**E3**
E1	E0	E2	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E3	E2	E0	E1
E1	E2	E3	E1	E0
E2	E0	E1	E2	E3
E3	E1	E0	E3	E2

3323686065,147,1

Neutrales Element

E1

Inverse

E0	E1	E2	E3
E2	E1	E0	E3

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E3	E0	E1	E2
E1	E0	E1	E2	E3
E2	E1	E2	E3	E0
E3	E2	E3	E0	E1

3034664475,75,3

Neutrales Element

E3

Inverse

E0	**E1**	**E2**	**E3**
E1	E0	E2	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E2	E3	E1	E0
E1	E3	E2	E0	E1
E2	E1	E0	E3	E2
E3	E0	E1	E2	E3

2984516430,27,2

Neutrales Element

E2

Inverse

E0	**E1**	**E2**	**E3**
E0	E1	E2	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E2	E3	E0	E1
E1	E3	E2	E1	E0
E2	E0	E1	E2	E3
E3	E1	E0	E3	E2

2982550380,57,2

Neutrales Element

E2

Inverse

E0	**E1**	**E2**	**E3**
E0	E3	E2	E1

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E2	E3	E0	E1
E1	E3	E0	E1	E2
E2	E0	E1	E2	E3
E3	E1	E2	E3	E0

2367415410,216,1

Neutrales Element

E1

Inverse

E0	E1	E2	E3
E3	E1	E2	E0

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E2	E0	E3	E1
E1	E0	E1	E2	E3
E2	E3	E2	E1	E0
E3	E1	E3	E0	E2

1927551885,216,2

Neutrales Element

E2

Inverse

E0	E1	E2	E3
E3	E1	E2	E0

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E1	E3	E0	E2
E1	E3	E2	E1	E0
E2	E0	E1	E2	E3
E3	E2	E0	E3	E1

1823589915,147,3

Neutrales Element

E3

Inverse

E0	E1	E2	E3
E2	E1	E0	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E1	E2	E3	E0
E1	E2	E3	E0	E1
E2	E3	E0	E1	E2
E3	E0	E1	E2	E3

1310450100,30,1

Neutrales Element

E1

Inverse

E0	E1	E2	E3
E0	E1	E3	E2

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E1	E0	E3	E2
E1	E0	E1	E2	E3
E2	E3	E2	E0	E1
E3	E2	E3	E1	E0

1310450865,27,1

Neutrales Element

E1

Inverse

E0	**E1**	**E2**	**E3**
E0	E1	E2	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E1	E0	E3	E2
E1	E0	E1	E2	E3
E2	E3	E2	E1	E0
E3	E2	E3	E0	E1

460492260,39,0

Neutrales Element

E0

Inverse

E0	**E1**	**E2**	**E3**
E0	E2	E1	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E0	E1	E2	E3
E1	E1	E3	E0	E2
E2	E2	E0	E3	E1
E3	E3	E2	E1	E0

460108230,57,0

Neutrales Element

E0

Inverse

E0	**E1**	**E2**	**E3**
E0	E3	E2	E1

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E0	E1	E2	E3
E1	E1	E2	E3	E0
E2	E2	E3	E0	E1
E3	E3	E0	E1	E2

458142180,27,0

Neutrales Element

E0

Inverse

E0	**E1**	**E2**	**E3**
E0	E1	E2	E3

Verknüpfungstabelle

	E0	**E1**	**E2**	**E3**
E0	E0	E1	E2	E3
E1	E1	E0	E3	E2
E2	E2	E3	E0	E1
E3	E3	E2	E1	E0

458142945,30,0

Neutrales Element

E0

Inverse

E0	E1	E2	E3
E0	E1	E3	E2

Verknüpfungstabelle

	E0	E1	E2	E3
E0	E0	E1	E2	E3
E1	E1	E0	E3	E2
E2	E2	E3	E1	E0
E3	E3	E2	E0	E1

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Bit] *Bitbucket Homepage*. `https://bitbucket.org`

[Can] *Rechtsausloeschung im Proofwiki*. `http://www.proofwiki.org/wiki/Cancellation_Laws`. – Stand: 30.05.2013

[Cob] *Cobertura Hompage*. `http://cobertura.sourceforge.net/`

[DG04] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI* (2004)

[GIT] *Git Homepage*. `http://git-scm.com/`

[GUI] *google-guice Hompage*. `http://code.google.com/p/google-guice/`

[MVN] *Maven Hompage*. `http://maven.apache.org/`

[RPC] *Remote Procedure Call auf Wikipedia*. `http://de.wikipedia.org/wiki/Remote_Procedure_Call`. – Stand: 30.05.2013

[Sat] *Satz auf Wikipedia*. `http://de.wikipedia.org/wiki/Satz_(Mathematik)`. – Stand 30.05.2013

[SIM] *Simon Hompage*. `http://dev.root1.de/projects/simon`