technische universität
dortmund

Bachelorarbeit

Untersuchung der Programmiersprache Julia zur Analyse großer Datenbestände in der Logistik

Nils Neumann
149468
Dezember 2019

Gutachter:
Prof. Dr.-Ing. Markus Rabe
Prof. Dr. Jens Teubner
Betreuer:
M.Sc. Joachim Hunker

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Datenbanken und
Informationssysteme (VI)
http://dbis.cs.tu-dortmund.de

In Kooperation mit:
Fakultät Maschinenbau
Fachgebiet IT in Produktion und Logistik
http://www.itpl.mb.tu-dortmund.de

Abstract

In der Logistik fallen sowohl immer größere als auch schneller wachsende Datenbestände an. Diese Datenbestände sind oft semi- oder unstrukturiert, umfangreich und ständigen Veränderungen unterworfen. Dadurch stellt das Gewinnen von aussagekräftigen Informationen aus diesen Datenbeständen eine Herausforderung dar. Als Lösungsansatz für diese Herausforderung sind geeignete Programmiersprachen entwickelt worden. Eine solche Sprache ist Julia, die für die Verwendung in numerischem und wissenschaftlichem Rechnen entwickelt worden ist und gleichzeitig die Vorzüge einer Allzwecksprache besitzt. Diese Eigenschaften zeichnen Julia für die Anwendung im Kontext großer Datenbestände in der Logistik aus. Inwieweit Julia sich in der Verarbeitung großer Datenbestände gegen andere Programmiersprachen durchsetzen sowie auf große Datenbestände angewendet werden kann wird in dieser Arbeit untersucht.

Inhaltsverzeichnis

1 Einleitung

Logistik stellt ein Bindeglied zwischen Handel, Produktion sowie Information dar, wobei die Entwicklung von neuen, immer größeren und komplexeren Logistiksystemen grundlegend von rechnergestützten Informationssystemen abhängig ist (ten Hompel et al. 2014). Aus dem Zusammenspiel zwischen IT und Logistik entstehen unter anderem immer größere Datenmengen, die zur Analyse und für die Entscheidungsunterstützung in der Logistik herangezogen werden müssen (Lehmacher 2016). Die unterschiedlichen logistischen Bereiche, wie die Produktions-, die Lager- oder die Distributionslogistik, erzeugen dabei unter anderem eine Vielzahl von Daten mit unterschiedlichen Strukturen (Hausladen 2016).
Die aus großen Datenmengen resultierende Herausforderung ist das effiziente Gewinnen von Information und Wissen, für die eine, auf große Datenmengen optimierte, Programmiersprache nicht nur sinnvoll, sondern zwingende Notwendigkeit ist (Voulgaris 2016). Eine Sprache, die nach Balbaert (2015) speziell für das effiziente Gewinnen von Informationen aus großen Datenmengen entwickelt wurde, ist die Programmiersprache Julia. Denn Julias Fokus liegt auf der einen Seite auf einer optimalen Performanz bei der Verwendung im numerischem und wissentschaftlichem Rechnen. Auf der anderen Seite weist Julia ebenfalls die Vorzüge einer Allzwecksprache (GPL, engl. General Purpose Language) auf, wodurch sie sowohl den nötigen Abstraktionsgrad zur Beherrschung unterschiedlich geformter Daten, als auch die Fähigkeit zur Verarbeitung von großen Datenmengen mitbringt (Balbaert 2015).
Das Ziel dieser Arbeit ist die Untersuchung der Programmiersprache Julia in Hinblick auf die Arbeit mit großen Datenbeständen aus der Logistik. Für diese Untersuchung müssen zunächst große Datenbestände im Allgemeinen und Datenbestände im Kontext der Logistik im speziellen betrachtet werden. Desweiteren wird die Programmiersprache Julia von anderen Programmiersprachen im Bezug zur Arbeit mit großen Datenbeständen abgegrenzt. Dazu werden die Eigenschaften von Programmiersprachen betrachtet und unterschiedliche Programmiersprachen ausgewählt. Anschließend werden als weiteres Teilziel verschiedene Anforderungen aus den angestellten Betrachtungen abgeleitet und Kriterien zur Abgrenzung der verschiedenen Programmiersprachen voneinander erarbeitet. Abschließend wird die Anwendbarkeit von Julia auf große Datenbestände mittels eines exemplarischen Datenbestands evaluiert werden, wofür unterschiedliche Programmbeispiele entwickelt werden.
Zum Erreichen der zuvor definierten Ziele schafft Kapitel 2 eine Arbeitsgrundlage, indem eine Einführung in die Thematik der großen Datenbestände und in Datenbestände in der Logistik gegeben wird. Dazu werden die Begriffe der Information und des Wissens und die notwendigen Voraussetzungen zur Verwendung von Daten betrachtet. Außerdem werden die Eigenschaften großer Datenbestände und Datenbestände in der Logistik untersucht. Daraufhin rücken, in Kapitel 3, Programmiersprachen für die Datenverarbeitung in großen Datenbeständen in den Fokus der Betrachtung. Hier werden, nach einer Einführung in die allgemeinen Grundlagen von Programmiersprachen, unterschiedliche Statistiken betrachtet, um zu ermitteln welche Programmiersprachen in puncto Verbreitung und Nutzungsgrad für die Abgrenzung zur Programmierspache Julia herangezogen werden können. Diese Betrachtung wird um eine Untersuchung von exemplarischen Tools zur Arbeit mit großen Datenbeständen und die zu ihrer Entwicklung verwendeten Programmiersprachen erweitert.

Abschließend zu diesem Kapitel wird die Programmiersprache Julia selbst vorgestellt. In Kapitel 4 werden im ersten Schritt, auf Basis der vorhergegangenen Kapitel, Anforderungen an Programmiersprachen zur Verarbeitung großer Datenbestände im Kontext der Logistik abgeleitet. Im zweiten Schritt werden Kriterien an Programmiersprachen zur Datenverarbeitung im zuvor geannten Kontext erarbeitet, anhand derer anschließend die zuvor ausgewählten Programmiersprachen abgegrenzt werden. Kapitel 4 schließt mit der Bewertung von Julia anhand der durchgeführten Untersuchungen. Zum Abschluss dieser Arbeit wird in Kapitel 5 zusätzlich eine Evaluierung der Eignung von Julia auf große Datenbestände in der Logistik vorgenommen, indem die Anwendung von Julia mittels Programmbeispielen auf einem exemplarischen Datenbestand gezeigt wird.

2 Große Datenbestände

Dieses Kapitel beginnt, in Abschnitt 2.1, mit der Erläuterung der Begrifflichkeiten von Daten, Information und Wissen. In Abschnitt 2.2 wird betrachtet welche Anforderungen an zu untersuchende Daten gestellt werden und wie Daten in eine zur Untersuchung geeignetes Form transformiert werden können. Damit der Kontext der Untersuchung der Programmiersprache Julia aufzeigbar ist, erfolgt in Abschnitt 2.3 eine Herausstellung der Eigenschaften großer Datenbestände. Desweiteren wird, in Abschnitt 2.4, die Logistik als Anwendungsgebiet vorgestellt und ein Bezug zu Daten in der Logistik hergestellt.

2.1 Daten, Information und Wissen

Beim Gewinnen von Informationen und Wissen aus großen Datenmengen stellt sich zuerst die Frage, wie sich *Daten*, *Information* und *Wissen* überhaupt voneinander unterscheiden. Um eine Grundlage für die anstehende Untersuchung zu schaffen, werden in diesem Abschnitt deshalb Defintionen zu diesen Begrifflichkeiten erarbeitet.
Der Begriff der Daten bezeichnet, nach Cleve und Lämmel (2016), im Allgemeinen eine Menge von Zeichen mit einer definierten Syntax. Man unterscheidet zwischen unstrukturierten, semistrukturierten, quasi-strukturierten sowie strukturierten Daten. Gronwald (2017) stellt die These auf, dass in der Realität keine unstrukturierten Daten existieren, denn alle Daten besitzen zumindest eine versteckte Struktur, da sie letztendlich von Menschen, oder Maschinen erzeugt wurden. Im Gegensatz dazu bezeichnet Gronwald Daten, deren Struktur so komplex ist, dass sie nicht mittels einfacher Umformungen in eine strukturierte Form gebracht werden können als unstrukturierte Daten. Die benötigten Umformungen werden im Weiteren als Transformationen bezeichnet. Zur Wahrung einer immer gleichbleibenden Struktur werden Datentypen verwendet. Ein Datentyp legt fest, in welcher Art und Form Daten des jeweiligen speziellen Datentyps vorliegen und welche Operationen mit den Daten durchführbar sind (Dausmann et al. 2010).
Cleve und Lämmel (2016) und Gronwald (2017) klassifizieren Daten demnach wie folgt:

- Strukturierte Daten zeichnen sich durch eine feste Struktur, also einen festen Datentypen, aus. Zu den strukturierten Daten gehören beispielsweise relationale Datenbank-Tabellen oder CSV-Dateien, deren enthaltene Daten eine immer gleiche feste Struktur aufweisen.
- Unstrukturierte Daten besitzen keine feste Struktur. Klassische Beispiele für unstrukturierte Daten sind Bilder oder Texte. Um diese analysieren zu können, müssen diese Daten zunächst in eine strukturierte Form transformiert werden.
- Quasi-strukturierte Daten besitzen unregelmäßige Datentypen und können in strukturierte Daten transformiert werden. Hierzu gehören zum Beispiel Datenströme, also kontinuierliche Flüsse von Datensätzen, die Inkonsistenzen in ihren Daten oder Datentypen besitzen.
- Daten, deren Struktur Merkmale sowohl von unstrukturierten, als auch von strukturierten Daten aufweisen, nennt man semistrukturierte Daten. Beispiel hierfür sind Datein

mit einem erkennbaren Muster, wie HTML- Webseiten, die auf der einen Seite aus strukturierten Elementen, den sogenannten Tags, bestehen und auf der anderen Seite beliebigen unstrukturierten Inhalt besitzen können.

Von Bedeutung für die Analyse von Daten sind, nach Cleve und Lämmel (2016), vorrangig die strukturierten Daten auf deren Basis folgende Begrifflichkeiten definiert werden:

Definition 2.1 Datensatz: Zusammenhängende Daten in strukturierten Dateiformaten nennt man Datensätze. Die, in solchen Datensätzen enthaltenden, Daten befinden sich in einer fester Reihenfolge.

Definition 2.2 Attribut: Gleichartige Daten in einer festen Reihenfolge definiert man jeweils unter dem Begriff des Attributs. Ein Attribut enthält daher immer dieselbe Struktur von Daten und kann einem festen Datentyp zugeordnet werden.

Für das Finden von Lösungen wird ein Weg benötigt, mit dem es möglich ist aus Daten Wissen zu generieren und dieses zu verwenden. Dazu ist es jedoch nötig, den Zusammenhang zwischen Daten, Information und Wissen genauer zu betrachten.
Ackoff (1989) beschreibt diesen Zusammenhang zwischen Daten, Information und Wissen mittels einer Hierarchie, wie in Abbildung 2.1, nach Rowley (2007, S.164), dargestellt. Diese Hierarche nennt man DIKW-Hierarchie, wobei DIKW für die Anfangsbuchstaben von Daten (engl. Data), Information (engl. Information), Wissen (engl. Knowledge) und Weisheit (engl. Wisdom) steht.

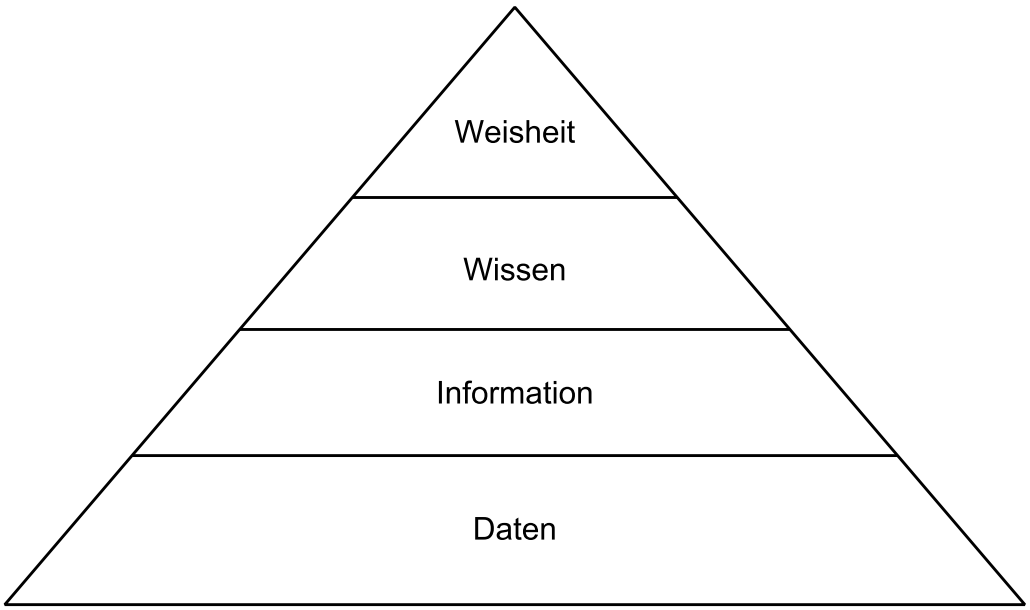

Abbildung 2.1: DIKW Hierarchie nach Rowley (2007, S.164)

- *Daten* definiert Ackoff (1989) dabei als reine Beobachtungsergebnisse, also Repräsentationen von Objekten, Ereignissen und ihrer Umwelt. Der Autor bezeichnet Daten alleine als unnütz, denn nur durch Bedeutung oder Funktionalität entstehe aus Daten Wissen.

- *Information* wird von Ackoff als Bestandteil von Beschreibungen und Antworten auf Fragen nach dem „Wer?“, „Was?“, „Wann?“, „Wie viel?“. Information wird dabei aus Daten gewonnen.
- *Wissen* beschreibt Ackoff als die Möglichkeit Informationen in Instruktionen umzuwandeln, wobei Wissen von anderen, die dieses Wissen besitzen, vermittelt, durch Instruktion erlernt oder Erfahrungen gemacht werden kann.
- *Weisheit* umschreibt, nach Ackoff, die Fähigkeit die Effektivität zu steigern. Weisheit verleiht Wissen einen Wert, womit man in der Lage ist Entscheidungen zu treffen.

Rowley (2007) entwickelte eine erweiterte Version der von Ackoff erstellten DIKW Hierarchie, die die Hierarchie um Indikatoren für den Wert, die Bedeutung sowie die Umsetzbarkeit in Rechnerprogrammen erweitert.

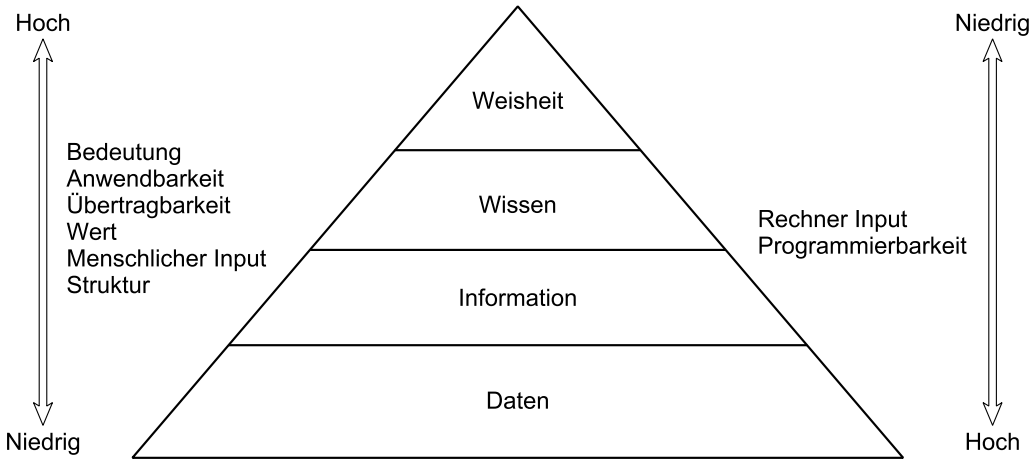

Abbildung 2.2: Wisdom Hierarchie nach Rowley (2007, S.176)

Wie in der erweiterten Hierarchie, in Abbildung 2.2, dargestellt, erhöhen sich die Eigenschaften *Bedeutung*, *Anwendbarkeit*, *Übertragbarkeit*, *Wert*, *Menschlicher Input* und *Struktur* von Daten zu Wissen. Das bedeutet demnach, dass zum Beispiel dem Wissen oder der Weisheit mehr Bedeutung beigemessen werden kann als den Daten und, dass den Daten weniger Struktur innewohnt als der Information. Im Gegensatz dazu stehen die Eigenschaften der *Programmierbarkeit* und des Rechner Inputs. Diese zeigen, dass die automatisierte Umsetzbarkeit von Programmen, umso komplizierter und komplexer wird, desto weiter man sich von der „Daten-Ebene“ entfernt. Rowley zeigt damit, dass *Daten*, *Information*, *Wissen* und *Weisheit* nicht nur aufeinander aufbauen, sondern, dass unter anderem ein Zugewinn an *Bedeutung*, *Anwendbarkeit*, *Übertragbarkeit* und *Wert* zu einer Reduktion von rechnergestützter Unterstützung führt, da die oberen Ebenen nur durch menschliche Erfahrung erreicht werden können.

Eine weitere Betrachung des Zusammenhangs zwischen Daten, Information und Wissen liefert Abbildung 2.3. In der nach North (2016) dargestellten Wissenstreppe wird nicht nur der Zusammenhang zwischen *Zeichen*, *Daten*, *Information*, *Wissen* und *Handeln* dargestellt, sondern es wird ebenfalls übersichtlich gezeigt, welche Eigenschaft jede Ebene mit der nächsten Ebene verbindet.

North (2016) bezeichnet *Daten* als geordnete *Zeichen* (Buchstaben, Ziffern oder Sonderzeichen). *Daten* sind *Zeichen* mit einer Syntax, „die noch nicht interpretiert sind“ und erst zu *Information* werden, sobald ihnen eine Bedeutung zugeordnet wird (North 2016, S.36).

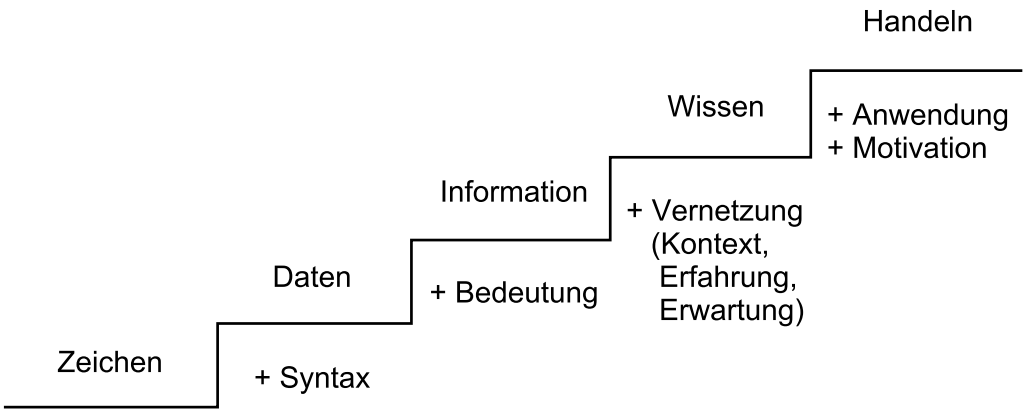

Abbildung 2.3: Wissenstreppe nach North (2016, S.37)

Allerdings sind diese Informationen für den Anwender noch ohne Wert, solange sie nicht mit anderen Informationen in Beziehung gesetzt werden. Erst durch die Vernetzung von *Information* mit anderen Informationen in Form von Kontext, Erfahrungen oder Erwartungen entsteht *Wissen*. In der Wissenstreppe nach North folgt auf das *Wissen* das *Handeln*, auch *Können* genannt, dass sich durch die Fähigkeit und die Motivation zur Anwendung des Wissens definiert. North definiert noch weitere Stufen in der Wissenstreppe, die jedoch für die hier angestellten Betrachtungen nicht weiter von Bedeutung sind und deshalb nicht beschrieben werden.

Aus den unterschiedlichen Betrachtungen zum Thema Daten, Information und Wissen ergeben sich einheitliche Definitionen für die Begriffe Daten, Information und Wissen:

Definition 2.3 Daten: Daten sind Mengen von Zeichen (Buchstaben, Ziffern oder Sonderzeichen) mit einer definierten Syntax (Cleve und Lämmel 2016).

Definition 2.4 Information: Information entsteht aus Daten, denen eine Bedeutung zugeordnet worden ist (Cleve und Lämmel 2016).

Definition 2.5 Wissen: Wissen bezeichnet vernetzte Information, also „Information in Verbindung mit der Fähigkeit, diese zu benutzen“ (Cleve und Lämmel 2016)

2.2 Verwendbarkeit von Daten

Die Menge der gesammelten Daten in allen Lebensbereichen steigt stetig an und dieser Zuwachs geschieht immer schneller (Han et al. 2011). Aufgrund der Menge der gesammelten Daten sind nicht immer alle Daten für die angestrebten Analysen relevant. Hinzu kommt, dass gesammelte Daten häufig nicht für die Analyse mittels algorithmischer Verfahren geeignet sind. Daher müssen diese Daten in der Regel zunächst Aufbereitungsschritte durchlaufen.

Einen dieser Schritte nennen Han et al. (2011) die *Datenvorverarbeitung* (engl. Data Preprocessing). Han et al. beschreiben, dass die gesammelten Daten bei der *Datenvorverarbeitung* mittels Schritten wie der *Datenbereinigung (engl. data cleaning)*, der *Datenintegration (engl. data integration)*, der *Datenreduktion (engl. data reduction)* oder der *Datentransformation (engl. data transformation)* auf die Analyse vorbereitet werden.

- *data cleaning*:
  Beim *data cleaning* werden die Daten „gesäubert“, indem beispielsweise fehlende Werte ergänzt, Ausreißer erkannt oder entfernt und Inkonsistenzen entfernt werden.
- *data integration*:
  Bei *data integration* werden Daten aus unterschiedlichen Quellen zusammengetragen. Dabei können jedoch Redundanzen und Inkonsistenzen entstehen, die ebenfalls in diesem Prozessschritt, beispielsweise mittels Methoden des *data cleanings*. behoben werden müssen.
- *data reduction*:
  Bei der *data reduction* werden die Daten, zum Beispiel durch Kodierung, komprimiert, um den Umfang der zu analysierenden Daten zu reduzieren.
- *data transformation*:
  Bei der *data transformation* werden die Daten zur Vorbereitung der Analyse beispielsweise normalisiert, also in eine Form frei von Redundanzen gebracht, oder diskretisiert, also in einzelne Elemente aufgeteilt.

Zur Umsetzung der einzelnen Schritte in der *Datenvorverarbeitung* gibt es viele unterschiedliche Ansätze, deren Betrachtung hier jedoch nicht im Fokus steht und auf die deshalb nicht näher eingegangen werden soll.

Eine weitere Möglichkeit der Datenaufbereitung wird von Cleve und Lämmel (2016) wie folgt erläutert: Zum Herausfiltern von benötigten Daten aus gesammelten Daten, werden alle Daten aus allen verfügbaren Quellen gesammelt. Dabei stellt bereits die Qualität der Daten ein häufig auftretendes Problem dar, denn selten liegen die gesammelten Daten in der zur Analyse erforderlichen Qualität vor. Demnach müssen die Daten nach dem Sammeln anhand von allgemeinen Kriterien, wie der Korrektheit oder der Aktualität, beispielsweise geprüft und transformiert oder gefiltert werden.
Farkisch (2011) und Schnider et al. (2016) beschreiben diesen Prozess des Sammelns und Aufbereitens von Daten genauer als ETL-Prozess (engl. Extract, Transform, Load - Process), wie in Abbildung 2.4 dargestellt.

Dieser ETL-Prozess dient dazu Daten aus unterschiedlichen Quellen mit unterschiedlichen Strukturen zusammenzuführen, zu bereinigen und für die weitere Verarbeitung bereit zu stellen. Farkisch (2011) und Schnider et al. (2016) beschreiben den ETL-Prozess wie folgt:

1. Der Extraktionsschritt (engl. Extraction) dient zum Auswählen und Kopieren von Daten aus unterschiedlichen Quellsystemen.
2. Der Transformationsschritt (engl. Transformation) bringt die extrahierten Daten in eine gültige und interpretierbare Form.
3. Der Ladeschritt (engl. Load) stellt die transformierten Daten für die weitere Verwendung bereit.

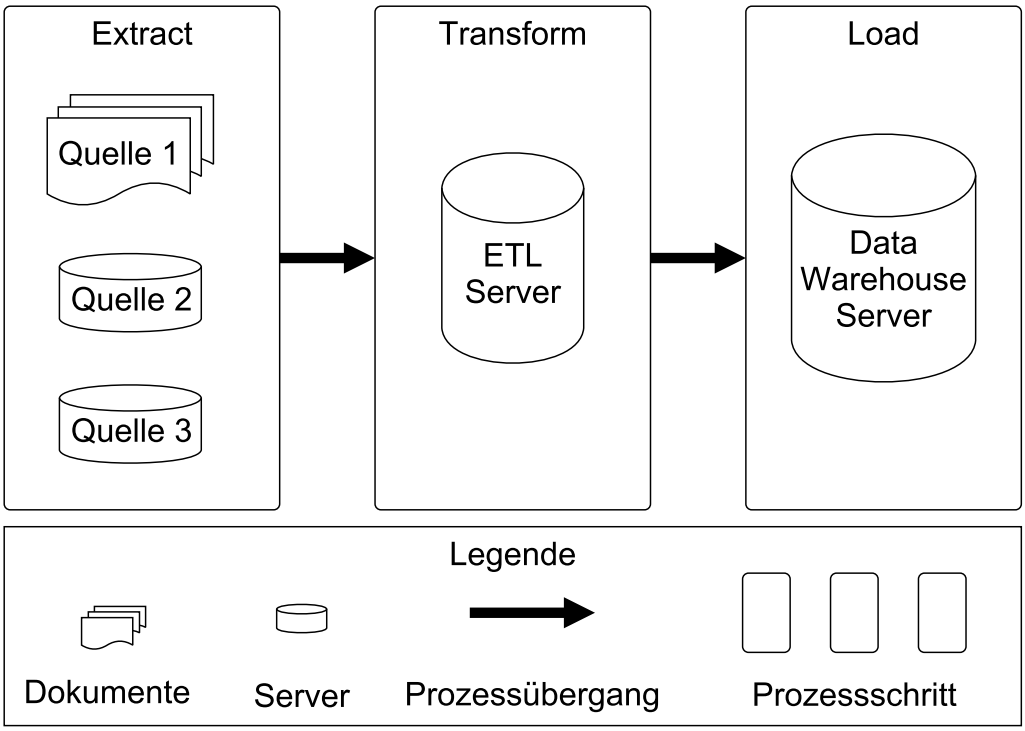

Abbildung 2.4: ETL-Prozess nach Ghosh (2019)

Bei der Transformation gibt es unterschiedliche Vorgehensweisen, wie das Abbilden auf Grundlage fester Regeln, das Korrigieren inkonsitenter oder fehlerhafter Daten oder das Filtern von Daten nach bestimmten Anforderungen. Die extrahierten Daten können dabei beliebig viele Transformationsschritte durchlaufen, wie zum Beispiel das:

- Zerlegen der Daten in einzelne Bestandteile (engl. Elementizing),
- Standardisieren (engl. Standardize),
- Verifizieren (engl. Verify),
- Vergleichen mit bereits vorhandenen Daten (engl. Matching) oder
- Gruppieren (engl. Grouping).

Durch beide, exemplarisch angeführten, Möglichkeiten zur Aufbereitung von Daten wird deutlich, dass es diverse Verfahren zur Verbesserung der Datenqualität gibt. Dabei steht immer die Vorbereitung der Daten auf die Analyse im Vordergrund. In Tabelle 2.1 werden dazu nach Cleve und Lämmel (2016) und Farkisch (2011) einige exemplarische Kriterien, die die Daten nach der Aufbereitung erfüllen sollten, aufgelistet.

Tabelle 2.1: Exemplarische Kriterien für die Datenqualität

Kriterium	Beschreibung
Aktualität	Die Daten sollen im Rahmen der gestellten Anforderungen aktuell sein.
Einheitlichkeit	Die Daten sollen in ihrer Repräsentation einheitlich sein.
Eindeutigkeit	Die Daten sollen eindeutig interpretierbar sein.
Exaktheit	Die Daten sollen präzise sein.
Korrektheit	Die Daten sollen den realen Sachverhalten entsprechen.
Nützlichkeit	Die Daten sollen für den vorgesehenen Einsatz von Mehrwert sein.
Validität	Die Daten sollen gültig sein.
Verständlichkeit	Die Daten sollen interpretierbar sein.
Volatilität	Die Daten sollen über eine gewisse Dauer repräsentativ sein und keinen extremen zeitlichen Schwankungen unterliegen.
Vollständigkeit	Die Daten sollen vollständig sein.
Zuverlässigkeit	Die Daten sollen nicht unsicher sein und Nachvollziehbar sein.

2.3 Eigenschaften großer Datenbestände

Bisher wurde von großen Datenmengen oder großen Datenbeständen gesprochen, doch was sind große Datenbestände und wie wird diese Größe gemessen? Ein Bericht von Forbes Media aus dem Jahr 2018 betitelt die Menge an Daten, die jeden Tag generiert werden, mit 2,5 Trillionen Bytes ($2{,}5 \cdot 10^{18}$ Bytes) (Marr 2019). Han et al. (2011) erklärten, dass täglich „riesige“ Datenmengen im Bereich von Terabytes($1 \cdot 10^{12}$ Bytes) bis Petabytes ($1 \cdot 10^{15}$ Bytes) erzeugt werden. Dabei nennen Han et al. zum Beispiel Verkaufstransaktionen, Börsenprotokolle, wissenschaftliche Experimente, Umweltüberwachung sowie weitere Sektoren wie beispielsweise die Telekommunikationsbranche als Produzenten von großen Datenmengen. Des Weiteren sprechen Fasel et al. (2016) bei großen Datenmengen in Bereichen von Terabyte ($1 \cdot 10^{12}$ Byte) bis Zettabyte ($1 \cdot 10^{21}$ Byte). Zwar werden nicht immer all diese Daten gespeichert und für Analysen verwendet, aber dennoch zeigen Fasel et al., Han et al., Cleve und Lämmel auf, dass die Menge der gesammelten Daten sehr groß ist. Doch die aufgezeigten Zahlen weisen Spannen von bis zu $1 \cdot 10^{9}$ Bytes auf und daher ist es nötig große Datenbestände nicht nur anhand ihres Volumens zu definieren, sondern weitere Eigenschaften für die Beschreibung hinzuzuziehen. Deshalb wird für die Beschreibung großer Datenbestände hier eine Charakterisierung aus der Literatur (vgl. Fasel et al. 2016, S.5f. Gao und Zhao 2016, S.25, Gobble 2013, S.64 sowie Meier und Kaufmann 2016, 12f.) aus dem Bereich des Big Data verwendet.
Dort werden große Datenbestände mit den drei V's beschrieben:

- *Volume*:
  Die Datenbestände sind umfangreich. Nicht nur in Anbetracht des benötigten Speicherplatzes, sondern ebenfalls in der Menge der gespeicherten Daten je Datensatz. Bei umfangreichen Datenbeständen spricht man dabei, nach Meier und Kaufmann (2016), von Tera- bis Zettabytebereichen (10^{12} Byte - 10^{21} Byte)

- *Variety*:
  Die Datenbestände enthalten vielfältige Datenformate (Strukturierte, Semistrukturierte und Unstrukturierte Daten).
- *Velocity*:
  Die Datenbestände sollen in Echtzeit verarbeitet werden und können eine hohe Fluktuation aufweisen.

NESSI (2012) ergänzt den Begriff der großen Datenmengen zusätzlich mit einem V:

- *Value*:
  Die Daten sollen einem Zweck dienen und einen Mehrwert generieren.

Und Meier und Kaufmann (2016) fügen den Eigenschaften großer Datenmegen ein weiteres V hinzu:

- *Veracity*:
  Die Daten liegen in unterschiedlichen Qualitäten vor, was in der Analyse berücksichtigt werden muss.

Große Datenbestände zeichnet also nicht alleine ihr immenser Umfang aus, sondern, durch die Vielzahl von Daten, rücken Eigenschaften, wie Vielfältigkeit und Unbeständigkeit in den Fokus der Betrachtung. Für die Untersuchung solcher Datenbestände ist demnach eine Berücksichtigung dieser Eigenschaften unerlässlich.
In dieser Arbeit wird die Eigenschaft des Value nicht weiter berücksichtigt, da das Werten der gewonnenen Informationen aus großen Datenbeständen mittels Wissen nicht Bestandteil der angestellten Untersuchung ist.

2.4 Datenbestände im Kontext der Logistik

Logistik beschreibt die räumliche und/oder zeitliche Transformation von Objekten, wozu unter anderem der Transport und die Lagerung von Informationen und Gütern gehören (Heiserich et al. 2011). Delfmann et al. (2011) beschreiben, in ihrem Positionspapier zum Grundverständnis der Logistik, die Logistik selbst hingegen allgemein als "das wissenschaftliche Prinzip [...], das wirtschaftliche Vorgänge und Prozesse als Flüsse von Gütern, Informationen, Werten oder Personen interpretiert". Damit definieren Delfmann et al. die Logistik nicht nur als Summe von logistischen Prozessen, die Flüsse von Gütern, Informationen, Werten und Personen, sondern auch als interdisziplinäre Wissenschaft. In der Praxis wird die Logistik, nach Heiserich et al. (2011), jedoch als unternehmensinterne, wie auch unternehmensübergreifende Komponente des Managements zur Durchführung effizienter, günstiger und anpassungsfähiger Material- und Informationsflüsse verstanden: „Die Logistik hat [...] die Aufgabe, die Systeme und Prozesse optimal zu gestalten und zu betreiben, um die gegebenen Leistungsanforderungen zu erfüllen“ (Heiserich et al. 2011, S.5).
Diese unterschiedlichen Definitionen zeigen, dass das Feld der Logistik umfangreich und vom Fokus der Betrachtung abhängig ist. Dennoch lassen sich die Herausforderungen, welche die Logistik an große Datenbestände stellt, entlang der, in Abschnitt 2.3, eingeführten Charakterisierung aufzeigen:
In der Logistik nimmt der Einsatz von automatisierten Systemen stetig zu, wodurch die Geschwindigkeit, mit der neue Daten erzeugt werden, immer weiter zunimmt (Hausladen 2016, vgl. ten Hompel et al. 2014). Diese Herausforderung bildet sich in den Eigenschaften großer Datenbestände in der *Velocity* (vgl. Abschnitt 2.3) ab. Die automatisierten Sys-

teme bilden Datenquellen und sind oft verteilte, dezentralisierte und teilweise autonome logistische Prozesse, wie beispielsweise unterschiedliche Transport- und Lagereinheiten oder Betriebe (Gao und Zhao 2016). Aus dieser Vielfältigkeit von Quellen entsteht eine Herausforderung, die sich in den Eigenschaften großer Datenbestände in einer hohen *Variety* (vgl. Abschnitt 2.3) widerspiegelt. Durch solche autonomen Teilnehmer an logistischen Prozessen können beispielsweise auch gleichartige Prozessschritte, wie das Verbuchen eines Einlagerungsvorgangs, in unterschiedlich geformten Datensätzen gespeichert werden (Gao und Zhao 2016). Die daraus resultierende Inkonsistenz lässt sich ebenfalls durch die Eigenschaft *Variety* (vgl. Abschnitt 2.3) abbilden. Da die Daten aus den verschiedensten Quellen stammen, sind sie meist wenig oder unstrukturiert und liegen zudem oft in unterschiedlichen Qualitäten vor (Meier und Kaufmann 2016). Die daraus folgende Herausforderung lässt sich den Eigenschaften der *Variety* und *Veracity* (vgl. Abschnitt 2.3) zuordnen. Weil die Daten durch diese Systeme immer häufiger automatisiert erzeugt werden, entstehen sowohl große Mengen an Daten als auch hohe Fluktuationen in den Datenbeständen (Meier und Kaufmann 2016). Die große Menge an Daten lässt sich mit der Eigenschaft des *Volume* (vgl. Abschnitt 2.3) und die ständige Veränderung der Daten mit der Eigenschaft der *Variety* beschreiben. Nicht zuletzt ist die Logistik von komplexen und dynamischen Beziehungen zwischen Zulieferern, Partnern und Kunden geprägt, woraus dynamische und komplexe Beziehungen innerhalb der Datenbestände entstehen können (Gao und Zhao 2016, vgl. Hausladen 2016). Diese dynamischen und komplexen Beziehungen lassen sich in den Eigenschaften des *Volume*, der *Variety* und der *Velocity* (vgl. Abschnitt 2.3) wiederfinden.

3 Programmiersprachen für die Datenverarbeitung in großen Datenbeständen

Die Betrachtung von Programmiersprachen und das Erstellen einer Abgrenzung von Programmiersprachen im Kontext großer Datenbestände, benötigt eine Einführung in die Programmiersprachen, wodurch eine Kategorisierung der Sprachen getroffen werden kann. Deshalb beginnt das Kapitel in Abschnitt 3.1 mit der Einführung von Programmiersprachen und ihrer Klassifizierung. Anschließend wird in Abschnitt 3.2 untersucht welche Programmiersprachen am häufigsten genutzt werden und welche von ihnen sich für die Arbeit mit großen Datenbeständen eignen. Anschließend erfolgt in Abschnitt 3.3 die Untersuchung der Programmiersprache Julia.

3.1 Programmiersprachen und ihre Kategorisierung

Für die Einführung von Programmiersprachen werden im Folgenden einige Begrifflichkeiten definiert. Zuerst wird erläutert, was eine Programmiersprache ist: Eigner et al. (2012) definieren eine Programmiersprache als „eine formale Sprache, die zur Erstellung von Verarbeitungsanweisungen für Rechnersysteme verwendet wird“. Dabei bezeichnet eine formale Sprache, wie in der theoretischen Informatik üblich, eine Menge von Wörtern, die aus einem Alphabet erzeugt werden können (Eigner et al. 2012). Das Ziel einer Programmiersprache ist, eine für Menschen einfache und verständliche Beschreibung von Rechnerbefehlen darzustellen (Eigner et al. 2012). Programmiersprachen werden dabei in sogenanntem Quelltext verfasst, der durch Editoren erstellt und zur Ausführung auf Rechnern in Maschinencode übersetzt wird (Eigner et al. 2012). Programmiersprachen werden nach Küveler (2006, S.29) in unterschiedliche Kategorien differenziert, wie in Abbildung 3.1 dargestellt.

Maschinensprachen und Assemblersprachen bilden die Kategorie der maschinen-orientierten Sprachen und dienen der direkten, vom Prozessortyp abhängigen, Programmierung von Rechnern mittels Binärcode (Folgen von „1“ und „0“) (Küveler 2006). Diese Ebene der Programmiersprachen findet in der hier angestellten Betrachtung jedoch keine weitere Beachtung, denn ihr Abstraktionsniveau ist nicht vergleichbar mit dem Abstraktionsniveau der Programmiersprache Julia, deren Untersuchung im Fokus dieser Arbeit steht.
Die Kategorie der problem-orientierten Sprachen ist unabhängig von bestimmten Rechnertypen und wird auch mit dem Begriff der „höheren Programmiersprache“ bezeichnet (Küveler 2006). Eine höhere Programmiersprache, im weiteren *Hochsprache* genannt, ist eine formalisierte, „aber der menschlichen Denk- und Ausdrucksweise“ angepassten Sprache, die sich durch einen hohen Abstraktionsgrad auszeichnet (Ernst et al. 2015). *Hochsprachen* ermöglichen dem Anwender demnach ein Programm zu schreiben, ohne sich mit den Einzelheiten der unterschiedlichen Rechnersysteme auseinandersetzen zu müssen. Die *Hochsprachen* unterteilen sich weiterhin in die Kategorie der Universalsprachen oder GPLs und die Spezialsprachen oder domänenspezifische Sprachen (DSLs, engl. Domain-Specific

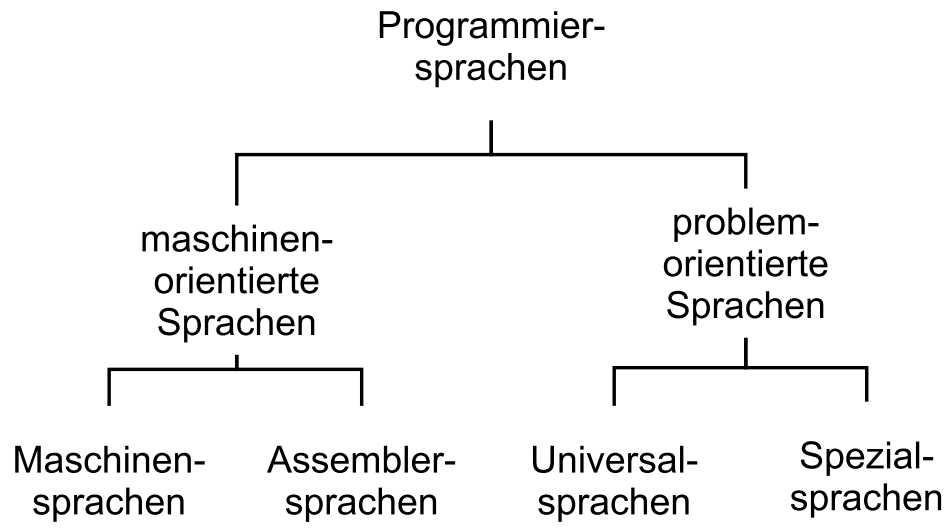

Abbildung 3.1: Programmiersprachen nach Küveler (2006, S.29)

Languages), wie aus Abbildung 3.1 hervorgeht. DSLss stellen dabei sowohl Notationen als auch Konstrukte zur Nutzung in ihrer Anwendungsdomäne zur Verfügung. Dadurch sollen Anwender aus der jeweiligen Domäne ausdrucksstarke und verständliche Werkzeuge an die Hand gegeben werden, um Programme zu entwickeln (Mernik et al. 2005). Die Begriffe der GPL und DSL werden Definition 3.1 und Definition 3.2 definiert.

> **Definition 3.1 GPL:** Eine GPL bezeichnet eine Programmiersprache, die für viele Anwendungsfälle verwendbar ist und ein hohes Maß an programmiertechnsicher Freiheit gibt (Kosar et al. 2016).

> **Definition 3.2 DSL:** Eine DSL zeichnet sich durch eine spezielle Anwendungsdomäne aus. Dabei stellen DSLs sowohl Notationen als auch Konstrukte zur Nutzung in ihrer Anwendungsdomäne zur Verfügung (Mernik et al. 2005).

Für die weitere Betrachtung von Programmiersprachen und ihre Kategorisierung werden in diesem Unterkapitel die Ausführungen von Ernst et al. (2015) verwendet, sofern nicht anders angegeben.
Sowohl Dausmann et al. (2010) als auch Ernst et al. (2015) erklären außerdem, dass bei den Programmiersprachen die *imperativen* und die *deklarativen* Sprachen voneinander abzugrenzen sind. Die Struktur der *imperativen* Sprachen ist an die Architektur von Rechnern angelehnt, denn Befehle in *imperativen* Sprachen bearbeiten Daten im gleichen Speicher, in dem die Daten gespeichert sind. Ein Programm, in einer *imperativen* Sprache, besteht aus Variablen und einer sequentiellen Folge von Befehlen zur Datenverarbeitung, deren Reihenfolge exakt festgelegt sein muss. Bei den *imperativen* Sprachen findet man unter anderem die Kategorien der:

- maschinenorientierten Sprachen
- prozeduralen Sprachen
- objektorientierten Sprachen

Prozedurale Sprachen werden von Ernst et al. (2015) als Sprachen beschrieben, die angelehnt an mathematische Formeln, sogenannte Prozeduren und Funktionen verwenden, die auch rekursiv, also sich selbst aufrufend, aufgerufen werden können. Prozeduren sind dabei Unterprogramme mit eigener Bezeichnung und eigenen Parameterlisten. Funktionen bilden eine spezielle Variante der Prozeduren, die einen Wert an das aufrufende Programm zurückgeben können. Sowohl Prozeduren als auch Funktionen fasst man unter dem Begriff der Methoden zusammen. Daraus ergeben sich die Definitionen Definition 3.3 und Definition 3.4.

Definition 3.3 Prozedur: Eine Prozedur ist ein Unterprogramm mit eigener Bezeichnung und eigener Parameterliste (Ernst et al. 2015).

Definition 3.4 Funktion: Eine Funktion ist ein Unterprogramm mit eigener Bezeichnung und eigener Parameterlisten, welches einen Wert an das aufrufende Programm zurückgeben kann (Ernst et al. 2015).

Ernst et al. (2015) beschreiben weiterhin objektorientierte Sprachen als Sprachen, in denen sowohl Daten als auch deren Eigenschaften und die darauf anwendbaren Operationen in sogenannten Klassen definiert werden. Bei der Initiierung dieser Klassen entstehen Objekte, die mittels Operationen als Nachrichten untereinander kommunizieren können, wodurch die Definitionen von Klassen und Objekten wie in Definition 3.5 und Definition 3.6 festgelegt, getroffen werden.

Definition 3.5 Klassen: Strukturen, in denen Daten, ihre Eigenschaften und die auf die Daten anwendbare Operationen zusammengefasst werden, heißen im Kontext der objektorientierten Sprachen Klassen (Ernst et al. 2015).

Definition 3.6 Objekte: Objekte werden aus Klassen abgeleitet und bilden einzelne Instanzen der jeweiligen Klasse (Ernst et al. 2015).

Die *deklarativen* Sprachen hingegen geben keine Befehle zur Datenverarbeitung an, sondern beschreiben das gewünschte Ergebnis der Verarbeitung, erklären Dausmann et al. (2010). Um aus einer Hochsprache für den Rechner ausführbaren Maschinencode zu generieren, also zur Ableitung von konkreten Befehlen an den Rechner, wird ein Übersetzer (Compiler) oder ein Interpretier-Programm (Interpreter) benötigt (Ernst et al. 2015). Dabei unterscheiden sich Interpreter und Compiler beispielsweise dadurch, dass Interpreter den Quelltext zeilenweise und Compiler als komplettes Programm in Maschinensprache übersetzen. Außerdem benötigt man für die meisten Sprachen eigens geeignete Compiler oder Interpreter. Weiterhin existieren verschiedenen Arten von Compilern, unter anderem die sogeannten Just-In-Time-Compiler (JIT-Compiler), die Quelltext immer erst dann zu Maschinencode generieren, sobald dieser benötigt wird. Doch JIT-Compiler und alle anderen möglichen Arten von Compilern sollen nicht Gegenstand der hier angestellten Betrachtung sein. Daher leben Definition 3.7 und Definition 3.8 die Begrifflichkeiten von Interpreter und Complier fest.

Definition 3.7 Interpreter: Ein Interpreter übersetzt ein Programm aus einer Hochsprache in Maschinensprache, indem der Quelltext des Programms zeilenweise übersetzt wird (Ernst et al. 2015).

Definition 3.8 Compiler: Ein Compiler übersetzt ein Programm aus einer Hochsprache in Maschinensprache, indem der vollständige Quelltext des Programm analysiert und anschließend übersetzt wird (Ernst et al. 2015).

Weiterhin beschreiben Ernst et al. (2015), dass Daten in *Variablen* gespeichert werden, die Datentypen besitzen. Damit legen *Variablen* Speicherplatz für Werte fest. Sobald einer Variable ein konkreter Wert zugewiesen wurde, ist dieser an der entsprechenden Stelle im Speicher abgelegt. *Variablen* können unter Umständen so initialisiert werden, dass sie später nicht mehr veränderbar sind. Solche *Variablen* nennt man *Konstanten*. Des Weiteren kann eine Variable aber auch nur einen Zeiger (Pointer) auf eine andere Variable enthalten und nicht in jeder Programmiersprache muss der Anwender einer Variable einen Datentyp zuweisen, da dies auch von der Programmiersprache intern getan werden kann (Ernst et al. 2015).

Definition 3.9 Variable: Eine Variable definiert einen festen Bereich im Speicher eines Rechners und kann einen Datentyp besitzen, der vorschreibt welche Art von Daten in der Variable gespeichert werden dürfen (Ernst et al. 2015).

Ernst et al. (2015) erklären, dass es möglich ist Datentypen in andere Typen umzuwandeln. Diese Operation nennt man Typenumwandlung (Casting).

Definition 3.10 Casting: Die Umwandlung eines Datentyps in einen anderen Datentyp wird als Casting bezeichnet (Ernst et al. 2015).

Desweiteren klassifiziert Brauer (2009) Typensysteme von Programmiersprachen mittels drei unterschiedlicher Kennwerte:

- Zum ersten existieren *starke* und *schwache* Typensysteme, wobei man von einem *starken* Typsystem (strong typing) spricht, wenn eine Programmiersprache sicherstellt, „dass auf einer Variable bzw. Objekt nur der typengemäße Satz von Operationen angewendet werden kann“, ein *schwaches* Typensystem (weak typing) hingegen kann solche Operationen zulassen (Brauer 2009).
- Als zweites existieren *statische* und *dynamische* Typensysteme, wobei in einem *statischen* Typensystem die Überprüfung zum Zeitpunkt der Übersetzung und in *dynamischen* Typensysteme zur Laufzeit erfolgt (Brauer 2009).
- Als drittes existieren *explizite* und *implizite* Typensysteme, wobei in *expliziten* Typensystemen das angeben des Datentyps für die Deklaration einer Variablen zwingend erforderlich ist und die Typen in einem *impliziten* Typensystem zur Laufzeit aus der Notation oder der Verwendung der Variablen der jeweilige Datentyp abgeleitet wird (Brauer 2009).

Güting und Erwig (2013) erklärt, dass implizite Datentypen beispielsweise mit der sogenannten *Typinferenz* oder Typenableitung abgeleitet werden können. Die Typinferenz bezeichnet ein Verfahren, bei dem in einer Programmiersprache mit einem *starken* Typensystem die Datentypen nicht vom Anwender explizit angegeben werden müssen, sondern von einem Algorithmus anhand des verwendeten Kontext abgeleitet werden.

Zur Erstellung von Programmen mit nicht linearen Programmabläufen, also mit Verzweigungen, Wiederholungen oder Möglichkeiten zum Überspringen von Code-Abschnitten, werden Konstrukte im Programm benötigt, die solche Operationen ermöglichen. Diese Konstrukte heißen *Kontrollstrukturen* (Ernst et al. 2015). Kontrollstrukturen werden in den Programmiersprachen intern definiert und zu ihnen zählen Ernst et al. (2015) unter anderem die Schleifen, Sprunganweisungen oder bedingte Anweisungen. Daher lautet die Definition für Kontrollstrukturen wie Definition 3.11 zeigt.

Definition 3.11 Kontrollstrukturen: Kontrollstrukturen sind festgelegte Konstrukte einer Programmiersprache, die eine nicht lineare Folge von Anweisungen in einem Programm ermöglichen (Ernst et al. 2015).

Das Volumen großer Datenmengen übersteigt, wie Abschnitt 2.3 aufzeigt, den überschaubaren Rahmen, womit die benötigte Rechenkapazität steigt. Um solch große Datenmengen verarbeiten zu können, wird die Rechenlast mittels *verteilter Systeme* oder der Parallelisierung von Prozessen verteilt. Pepper (2018) erklärt, dass diese *verteilten Systeme* „aus mehreren Objekten, die miteinander interagieren“ bestehen (Pepper 2018, S.262). In einem solchen *verteilten System* können durchaus beliebig viele Objekte gleichzeitig aktiv sein, woraus sich Anwendungen ergeben, welche die Parallelität von Prozessen verlangen. Aus der objektorientierten Sichtweise der Prozesse solcher Systeme folgert Pepper (2018), dass, durch die Äquivalenz von Objekten in *verteilten Systemen* und Objekten aus der objektorientieren Programmierung, bei den Objekten nicht unterschieden werden muss, ob es sich um ein durch Software realisiertes Objekt oder um eine „Hardware-Komponente“ handelt und, dass diese Objekte sogar beliebig gemischt werden können. So ergeben sich bei der Entwicklung paralleler Programme unterschiedliche Arten der Parallelität, behauptet Pepper. Zum einen die *Algorithmenparallelität*, bei der „mehrere Programmteile simultan und mehr oder weniger unabhängig voneinander an unterschiedlichen Aspekten eines Problems“ arbeiten (Pepper 2018, S.262). Zum anderen die *Datenparallelität*, bei der „dieselben Operationen simultan auf viele verschiedene Werte angewandt“ wird (Pepper 2018, S.262). Im Kontext der Datenverarbeitung scheint die *Datenparallelität* eine höhere Relevanz als die *Algorithmenparallelität* zu besitzen, denn eine Herausforderung bei der Verarbeitung großer Datenbestände ist eben die möglichst schnelle, beziehungsweise in diesem Fall gleichzeitige, Bearbeitung von Daten. Welche Art der Parallelität zur Verarbeitung großer Datenmengen vorliegt, hängt nicht von der Programmiersprache selbst ab, sondern von den Programmen die zur Verarbeitung großer Datenmengen eingesetzt werden. Daher wird zur folgenden Abgrenzung der Programmiersprachen Kontext lediglich die Fähigkeit zur Parallelisierung von Programmen betrachtet und Parallelität in Definition 3.12 definiert.

Definition 3.12 Parallelität: Parallelität beschreibt die Fähigkeit einer Programmiersprache unterschiedliche Objekte gleichzeitig auszuführen (Pepper 2018).

Für Programmiersprachen exisitieren häufig vorgefertigte Sammlungen von Methoden, die dem Anwender die Arbeit erleichtern sollen. Solche Sammlungen nennt man Programmbibliotheken oder kurz Bibliotheken (engl. library). Bibliotheken sind thematisch zuammenhängende Mengen von Prozeduren oder Funktionen, die im Allgemeinen keine eigenständigen Programme sondern Hilfsmittel darstellen (Ernst et al. 2015). Der Anwender kann Bibliotheken jedoch auch selber entwickeln und diese zu seinem Programm hinzufügen (Ernst et al. 2015). Beispiele für Programmbibliotheken sind unter anderem Bibliotheken für Datenbankanbindungen oder zu Datenanalyse, indem dem Anwender statistische Methoden

zur Verfügung gestellt werden. Daher wird der Begriff der Bibliotheken in Definition 3.13 definiert.

> **Definition 3.13 Bibliothek:** Eine Bibliothek oder „Library“ ist eine Sammlung von Methoden, die dem Anwender als Hilfsmittel dienen soll (Ernst et al. 2015).

Bei der Betrachtung der Zugänglichkeit von Programmiersprachen, für jedwede zu entwickelnden Programme, stellt die Verfügbarkeit der Programmiersprache selbst und die Verfügbarkeit von Dokumentationen, Anleitungen und ähnlichen zum Umgang mit der jeweiligen Programmiersprache eine nicht unbedeutende Rolle. Bei der Verfügbarkeit von Programmiersprachen ist zunächst die Art der Lizensierung der Programmiersprache für die Auswahl entscheidend. Dabei unterscheidet man prinzipiell in zwei Arten von Lizenzen (Grassmuck 2000). Auf der einen Seite die freien Lizenzen, die es dem Anwender erlauben Software kostenlos auszuführen, „zu analysieren, zu verändern anzupassen und weiter zu vertreiben“ und auf der anderen Seite proprietäre Lizenzen, welche die Verwendung der Programmiersprache einschränken (Lang 2003). Software unter solchen freien Lizensierungen werden auch als Open-Source-Software (OSS) bezeichnet (Hennig 2014). Prominentestes Beispiel für eine OSS ist dabei die GNU General Public License (GNU GPL), die eben die von Lang (2003) beschriebenen Eigeschaften und Nutzungsrechte an freier Software festlegt. Wird in der vorliegenden Arbeit also von Open-Source-Programmiersprachen oder anderen Open-Source-Projekten gesprochen, so bezieht sich diese Aussage auf Definition 3.14, die den Begriff der OSS festlegt.

> **Definition 3.14 Open-Source-Software:** Der Begriff der OSS bezeichnet kostenlose Software, die „mit offenem Quellcode weitergegeben werden darf“ und bei der jeder das Recht besitzt diese Software zu lesen und zu verändern (Hennig 2014).

Des Weiteren spielt die Verfügbarkeit von Dokumentationen und Anleitungen eine zentrale Rolle für den Anwender, weil durch sie sowohl die Zugänglichkeit zur Funktionsweise einer Programmiersprache geschaffen als auch die Erlernbarkeit einer neuen Programmiersprache erleichtert wird. Mit der Dokumentation von Software wird in der Regel bezweckt die Prozesse und die Anwendung der Software möglichst umfassend und nachvollziehbar zu erklären, sodass ein Anwender diese korrekt verwenden kann (Forward und Lethbridge 2002). Anleitungen oder auch, den ans englische angelehnte, *Tutorials* beschreiben Hilfsmittel zum Lernen, die „auf Basis eines fundierten inhaltlichen und mediendidaktischen Konzepts Informationskompetenz [...] vermittelt und dabei hinsichtlich des Lerneffekts potenziell für sich allein bestehen kann“ (Pfeffer 2005). Daraus ergeben sich die Definitionen für Softwaredokumentation und Tutorials als Definition 3.15 und Definition 3.16.

> **Definition 3.15 Softwaredokumentation:** Softwaredokumentation beschreibt für den Anwender möglichst umfassend und nachvollziehbar das entsprechende Softwareprodukt (Forward und Lethbridge 2002).

> **Definition 3.16 Tutorial:** Ein Tutorial beschzeichnet ein Lern-Hilfsmittel mit einer fundierten inhaltlichen Basis, dass dem Lernenden Kompetenz vermittelt und für sich alleine als Lernutensil bestehen kann (Pfeffer 2005).

3.2 Verbreitung von Programmiersprachen

Für die Auswahl geeigneter Programmiersprachen für die Abgrenzung der Programmiersprache Julia gegenüber anderer Programmiersprachen werden in diesem Abschnitt verschiedene Statistiken zum Thema Verbreitung und Nutzungsgrad von Programmiersprachen betrachtet. Zum Vergleich finden sich die zugehörigen Tabellen aus der Literatur in Anhang A.
Zunächst wird der sogenannte TIOBE Index betrachtet, der ein von TIOBE Software BV (2019) zur Verfügung gestelltes Hilfsmittel zur Ermittlung von beliebten Programmiersprachen ist. Er wird gebildet, indem die Anzahl von Webseiten mit dem Thema von Programmiersprachen in verschiedenen Suchmaschinen zur Erstellung einer Statistik genutzt werden. Der TIOBE Index von Oktober 2019 wird in Tabelle A.1 dargestellt.
Die zweite betrachtete Statistik ist die von RedMonk (2019) bereitgestellte Rangliste der Programmiersprachen. RedMonk nutzt interne Rankings von GitHub und StackOverflow, um genutzten Code und geführte Diskussionen zu Programmiersprachen, zu einer Statistik zusammenzufassen. StackOverflow, auf der einen Seite, bildet dabei eine geeignete Basis für diese Untersuchung, da StackOverflow mit mehr als einer Millionen registrierten Nutzern (Stand August 2012, Quelle: Vasilescu et al. 2013, S.190) und monatlich 50 Millionen Aufrufen (Stand: 14.Februar 2019, Quelle: Stack Exchange Inc. 2019) zu einer der meistbesuchten Plattformen für Programmierer weltweit zählt (Vasilescu et al. 2013). GitHub, auf der anderen Seite, mit über zwei Millionen offenen Projekten (Stand Oktober 2019, Quelle: Zapponi 2019), ist eine der bekanntesten Plattformen für Software-Entwicklungsprojekte und damit ebenfalls eine geeignete Basis für RedMonk's Statistik. Das von RedMonk bereitgestellte Ranking von Juni 2019 wird in Tabelle A.2 dargestellt.
Abschließend wird die „PYPL PopularitY"-Statistik betrachtet, welche die Häufigkeit der in Google Inc.s (Googles)'s Suchmaschine eingegebenen Suchbegriffe, mit dem Thema „Programmiersprachen Tutorials", nutzt, um die Beliebtheit der einzelnen Programmiersprachen herauszufinden (Carbonnelle 2019). Die von „PYPL PopularitY" erstellte Statistik von Oktober 2019 wird in Tabelle A.3 dargestellt.
Für die folgende Untersuchung werden aus den zuvor aufgeführten Statistiken die Programmiersprachen ausgewählt, die zum Zeitpunkt der Untersuchung den höchsten Verwendungsgrad besitzen und in jeder der Untersuchungen mindestens Platz 20 belegen. Jede Programmiersprache, die nicht in allen drei Statistiken unter den ersten 20 Platzierten landet, wird für diese Untersuchung aus der Betrachtung genommen. Die Platzierung der Programmiersprachen innerhalb der einzelnen Statistiken wird dabei nicht berücksichtigt. Somit soll ein möglichst repräsentativer Querschnitt der beliebtesten Programmiersprachen unabhängig von der Art der durchgeführten Erhebung geschaffen werden. Daraus ergeben sich die, in Tabelle 3.1 aufgelisteten, Programmiersprachen.

Tabelle 3.1: Meistverwendete Programmiersprachen

Programmiersprachen	
C	Perl
C++	PHP
C#	Python
Go	R
Java	Ruby
JavaScript	Swift

Im folgenden werden die in Tabelle 3.1 dargestellten Programmiersprachen, für einen Überblick, kurz vorgestellt. Übersichtliche Darstellungen der Eigenschaften jeder der beschriebenen Sprachen befinden sich in tabellarischer Form in Anhang B.

C ist nach Dausmann et al. (2010) eine GPL, die es erlaubt hardwarenahe Programme zu schreiben, indem direkte Zugriffe auf den Speicher oder Operationen auf Bit-Ebene ermöglicht werden. Dausmann et al. erklären, dass C dabei mit einem Typenkonzept, welches Typenkonvertierungen zulässt, arbeitet und zu den imperativen Sprachen gehört. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.1 zusammengefasst (vgl. cppreference.com 2019; GitHub, Inc. 2019).
C++ ist eine Weiterentwicklung der Programmiersprache C und ist damit ebenfalls eine GPL (Dausmann et al. 2010, S.44). Dabei erweitert C++ die Möglichkeiten des Anwenders zum Beispiel um Funktionalitäten zur objektorientierten Programmierung, wie das Konzept der Klassen, oder ein strengeres Typenkonzept als C (vgl. Dausmann et al. 2010, S.44, Heiderich und Meyer 2016, S.26). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.2 zusammengefasst (cppreference.com 2019).
C# ist eine von Microsoft Corporation™ (Microsoft) entwickelte GPL, die zu Microsofts net-Framework gehört (Czarnecki 2015, S.1). Dabei nutzt C# eine strenge Typisierung und eine objektorientierte Programmierweise (Czarnecki 2015, S.1). Der Name C# verweist darauf, dass C# der Nachfolger der Programmiersprache C++ sein soll (Czarnecki 2015, S.1). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.3 zusammengefasst (Microsoft Corporation 2019a).
Go ist eine von Google entwickelte Programmiersprache, deren Fokus auf einer schnellen Kompilierbarkeit und einer guten Skalierbarkeit auf große Datenmengen liegt (Meyerson 2014). Dabei zeichnet sich Go unter anderem durch eine „knappe Ausdrucksweise“ sowie eine integrierte Speicherbereinigung aus (Maurer 2012). Maurer (2012) erklärt außerdem, dass Go ein „sehr ausdrucksstarkes statisches Typensystem“ besitzt, jedoch ohne eine Hierarchie in diesem Typensystem auskommt und diverse Konstrukte zur „Unterstützung von Parallel-, Multiprozessor- und Netzprogrammierung“ liefert. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.4 zusammengefasst (vgl. golang.org 2019; GitHub, Inc. 2019).
Java ist eine sehr weit verbreitete, plattformunabhängige, objektorientierte Programmiersprache (Fun et al. 2000). Krüger und Stark (2009) erklären, dass Java unter anderem Eigenschaften zur Parallelität, Ausnahmebehandlung sowie Speicherverwaltung mitbringt und durch ihre Nähe zu den Programmiersprachen aus der Familie der Sprachen von C und C++ als eine Art Nachfolger dieser Sprachen gesehen werden kann. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.5 zusammengefasst (vgl. Oracle 2019; Kull 2019).
JavaScript wird von Flanagan (2007) als eine Skriptsprache ohne feste Typisierung und mit Methoden zur Objektorientierung beschrieben. Dabei werden durch Objekte Eigenschaftsnamen auf beliebige Werte abgebildet. Es werden Datentypen wie Zahlen, Strings oder Boolsche Werte unterstützt und Funktionalitäten wie Arrays und reguläre Ausdrücke angeboten. JavaScript findet hauptsächlich Verwendung in Webbrowsern (Steyer 2014). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.6 zusammengefasst (vgl. Mozilla Foundation 2019; The jQuery Foundation 2019; GitHub, Inc. 2019).
Perl wird von Schröter und Seidel (2003) als eine frei verfügbare Programmiersprache mit Funktionalitäten für objektorientiertes Programmieren beschrieben. Dabei steht Perl für „Practical Extraction and Report Language“. Perl findet vorallem in Client-Server Strukturen und in Textprotokollen Anwendung. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.7 zusammengefasst (Perl.org 2019; Hietaniemi 2019).

PHP ist eine Open-Source Skriptsprache, die speziell für die serverseitige Programmierung von Webseiten entwickelt worden ist (Heller und Dittfurth 2013). Dabei eröffnet PHP dem Anwender Funktionalitäten wie die Steuerung von Datenbankzugriffen oder das Erstellen von dynamischen Webseiten (Heller und Dittfurth 2013). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.8 zusammengefasst (vgl. The PHP Group 2019; GitHub, Inc. 2019).
Python wird von Sanner et al. (1999) als eine interaktive, objektorientierte, höhere Programmiersprache mit dynamischer Typisierung beschrieben. Außerdem wird Python von Interpretern zu plattformunabhängigem Maschinencode übersetzt (Sanner et al. 1999). Kapil (2019) beschreibt, dass Python Anwendung in Feldern, wie der Künstlichen Ingelligenz, der Robotik oder der Datenanalyse, findet. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.9 zusammengefasst (vgl. Python Software Foundation 2019a; Python Software Foundation 2019b).
R wird von Wollschläger (2016) als „eine freie Umgebung zur statistischen Analyse und grafischen Darstellung von Datensätzen, die befehlsorientiert arbeitet“ beschrieben. R's Kernkompetenz liegt daher auf der computergestützen statistischen Datenverarbeitung und integriert viele unterschiedliche Möglichkeiten, um Daten zur Organisation, Analyse, Transformation und Darstellung von Daten (Wollschläger 2016). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.10 zusammengefasst (vgl. The R Foundation 2019; GitHub, Inc. 2019).
Ruby wurde ursprünglich für einen möglichst intuitiven Umgang mit der Programmiersprache gestaltet, beschreibt Bylina (2014). Daher ist Ruby eine vollständig objektorientierte Programmiersprache und besitzt keine primitiven Typen, wie beispielsweise Ganzzahlen, sondern jeder Typ ist gleichzeitig ein Objekt (Bylina 2014). Daraus folgt nach Bylina, dass Prozesse für Menschen in Ruby zwar sehr schnell umsetzbar sind, dies jedoch zulasten zusätzlicher Rechenzeit gehen kann. Weitere Eigenschaften der Programmiersprache werden in Tabelle B.11 zusammengefasst (vgl. Ruby-Community 2019; Insa 2019).
Swift ist eine von Apple Inc. (Apple) entwickelte Programmiersprache für Apples mobiles Betriebssystem „iOS“ (Rebouças et al. 2016). Feiler (2017) und Wilde et al. (2011) beschreiben jedoch, dass Swift zusätzlich in der Parallelisierung von Anwendungen eingesetzt wird, in denen der Fokus auf der Zusammensetzung vieler unabhängiger Prozesse und gleichzeitiger Ausführung liegt. Dabei ist Swfit sowohl parallel als auch ortsunabhängig (Wilde et al. 2011). Weitere Eigenschaften der Programmiersprache werden in Tabelle B.12 zusammengefasst (vgl. Apple Inc. 2019; Chistenko 2019; GitHub, Inc. 2019).

Bei der Betrachtung der einzelnen, in Tabelle 3.1 aufgelisteten, Programmiersprachen lassen sich, basierend auf der Reihenfolge, keine Rückschlüsse auf die Gewichtung der Programmiersprachen ziehen. Um die in Tabelle 3.1 dargestellten Programmiersprachen zu gewichten, werden im Folgenden einige exemplarische Tools betrachtet, die für die Arbeit mit großen Datenbeständen entwickelt worden sind:

- Apache Cassandra:
  Das von der Apache Software Foundation™ (Apache) entwickelte Apache Cassandra™ (Cassandra) ist ein Open-Source Datenbank Management System, zur Verwaltung großer Datenmengen, die über unterschiedliche Server verteilt sind. Dabei verwendet Cassandra einen NoSQL-Ansatz mit einer „key-value“-Struktur, in der ein *Key* auf mehrere Values verweisen kann. Cassandra ist in der Programmiersprache Java entwickelt worden (Cassandra, Apache 2014).

- Apache Spark:
  Zaharia et al. (2016) und Shoro und Soomro (2015) bezeichnen Apache Spark™ (Spark),

als das Werkzeug der Wahl, wenn es um die Arbeit mit großen Datenbeständen geht. Spark ist eine funktionale Programmier-API und basiert zwar auf einem ähnlichen Model wie Googles MapReduce, erweitert dieses jedoch um eine Funktionalität namens *Belastbare Verteilte Datensätze* (RDDs, engl. Resilient Distributed Datasets). Zaharia et al. (2012) beschreibt RDDs als eine verteilte Speicherabstraktion zur Arbeit auf großen Datenbeständen, die dem Anwender unter anderem erlauben, mittels eines Shared Memory Verfahrens, untersuchte Datensätze erneut zu verwenden, ohne diese neu generieren zu müssen. Durch diese Erweiterung kann bei der Anwendung in vielen *iterativen* Prozessen viel Zeit und Aufwand in der Datenreplikation, den I/O-Prozessen von Speichermedien und der Serialisierung eingespart werden. Spark besitzt zusätzlich Funktionen wie Stream-Processing, eine schnelle Störungsbehebung oder optimieres Scheduling, welche die API, die auf den Programmiersprachen Scala, Java, Python und R basiert, zu einer gängigen Alternative zu Googles MapReduce machen (vgl. Zaharia et al. 2012; Apache Software Foundation 2019c).

- Disco:
  Disco ist ein, von Nokia entwickeltes, leichtgewichtiges Open-Source-Framework, das ebenfalls auf MapReduce basiert. Dieses Tool ist in Python und der funktionalen Programmiersprache Erlang programmiert worden und dient zur Analyse und zum Indizieren von großen Datenbeständen (Nokia Corporation 2019).

- Dryad:
  Dryad ist eine, von Microsoft entwickelte, Infrastruktur zur Verarbeitung von großen Datenmengen durch Ressourcenverteilung und parallele Verarbeitung von Daten in C++. Dazu werden unterschiedliche Verfahren implementiert, wie Googles MapReduce oder relationale Algebra, die durch gerichtete aperiosche Graphen (DAG, engl. Directed Acyclic Graph) strukturiert werden (vgl. Isard et al. 2007; Microsoft Corporation 2009; Microsoft Corporation 2019b).

- DryadLINQ:
  Microsofts DryadLINQ offeriert eine Umgebung zur Entwicklung von Anwendungen zur Arbeit mit großen Datenbeständen. Mit den Programmiersprachen des .NET Frameworks, konkret Visual Basic™ (VB) und C# , lassen sich unter Zuhilfenahme von Language Integrated Query (LINQ), ebenfalls von Mircosoft, schnell und einfach Programme zur Untersuchung großer Datenbestände schreiben (vgl. Ekanayake et al. 2009 und Microsoft Corporation 2009).

- Hadoop:
  Apache Hadoop® ist eine Open-Source Software, die ein Java-basiertes Interface für MapReduce darstellt, das Funktionalitäten zur Darstellung, Speicherung und Analyse großer Datenmengen anbietet (vgl. Gronwald 2017; Apache Software Foundation 2019a).

- High Performance Computing Cluster (HPCC):
  HPCC ist ein von LexisNexis Risk Solutions entwickeltes Open-Source Tool zur schnellen Verarbeitung und Analyse von großen Datenbeständen (vgl. HPCC Systems, LexisNexis Risk Solutions 2019; Anthony und Arjuna. 2011). Mishra et al. (2019) erklären, dass HPCC auf zwei unabhängigen Clusterverarbeitungsumgebungen basiert, um die paralle Verarbeitung von großen Datenmengen zu verbessern. Diese Umgebungen sind:

  1. Data Refinery Engine:
     Die erste Umgebung, auch Thor Cluster genannt, dient dem klassichen ETL-Prozess, also dem Extrahieren, Transformieren und Laden von Daten. Sie ähnelt in Funktion und Struktur Googles MapReduce.

2. Rapid Data Delivery Engine:
   Die zweite Umgebung, auch Roxie Engine genannt, ist ein Hochleistungstool zur Analyse von Daten.

HPCC ist in den Programmiersprachen C++ und Enterprise Control Language (ECL), einer datenzentrierten Programmiersprache mit einer Datenfluss-Architektur (engl. dataflow architecture), die während des Komplierens in optimiertes C++ übersetzt wird, geschrieben (Anthony und Arjuna. 2011).

- Hive:
  Apache HIVE™ ist eine Java-basierte Erweiterung für Hadoop, die unter Zuhilfenahme von SQL „Data Warehouse-Funktionalitäten“ zu Hadoop hinzufügt und über Web-, Server- und Konsolen-Interfaces verfügt (vgl. Gronwald 2017; Apache Software Foundation 2019b).

- Hydra:
  Hydra ist ein verteiltes Datenverarbeitungssystem, entwickelt von AddThis™, das Datenströme verarbeiten und zu Bäumen aggregieren (engl. aggrigate), zusammenfassen (engl. sum) oder transformieren (engl. transform) kann. Es basiert auf der Programmiersprache Java und unterstützt den Anwender ebenfalls mit integrierter Ressourcenverteilung, Aufgabenmanagement, verteilten Backups und effizientem Transfer großer Datenmengen (AddThis 2019).

- MapReduce:
  Gronwald (2017) schreibt, dass zur Aufbereitung großer un- oder semistrukturierter Datenmengen oft, das von Google entwickelte, MapReduce verwendet wird. Dean und Ghemawat (2004) erläutern dieses Verfahren als einen Prozess zur Generierung analysefähiger Datensätze aus großen Datenbeständen. Dazu werden im ersten Schritt, dem *Map*, aus allen Daten Paare aus Schlüssel und Wert (engl. key/value) generiert. Anschließend werden im *Reduce* alle Paare, die den selben Key besitzen zusammengeführt. Dabei zeichnet dieses Verfahren vor allem aus, dass es sehr gut skalierbar, also parallelisier- und verteilbar, ist. Die MapReduce Bibliothek selbst ist in der Programmiersprache C++ verfasst und wird von unterschiedlichen anderen Tools implementiert.

- Qt Concurrent:
  Die Qt Group (Nasdaq Helsinki: QTCOM) implementiert mit ihrer C++-basierten API “Qt Concurrent” die Funktionsweise von Googles MapReduce und wirbt dabei mit seiner einfachen Möglichkeit des Multithreadings und seiner automatischen Skalierbarkeit auf größere oder kleinere Umgebungen (Qt Group (Nasdaq Helsinki: QTCOM) 2019).

- TEZ:
  Apache TEZ™ (TEZ) ist ein von der Apache Software Foundation™ (Apache) entwickeltes Open-Source Framework zur Verarbeitung von großen Datenmengen auf Basis von Hadoop und Microsofts Dryad, welches ein Teil des zuvor vorgestellten DryadLINQs ist (Kannan 2015). Es basiert auf MapReduce und verwendet dabei DAG zur Repräsentation des Datenverarbeitungsflusses (Saha et al. 2015). Singh und Kaur (2016) behaupten dazu, dass TEZ zusätzlich das interaktive Ausführen von Anfragen unterstützt. TEZ ist eine Java basierte Bibliothek (vgl. Saha et al. 2015; Apache Software Foundation 2019d).

Aus den zuvor aufgeführten Tools im Kontext großer Datenbestände ergeben sich die, in Tabelle 3.2 dargestellten, verwendeten Programmiersprachen. Diese in Tabelle 3.2 aufgeführten Tools werden nach den verwendeten Programmiersprachen zusammengefasst und anschließend nach der Häufigkeit des Auftretens der einzelnen Programmiersprachen sortiert. Daraus lässt sich die, in Tabelle 3.3 dargestellte, Zusammenfassung ableiten.

Tabelle 3.2: Verwendete Programmiersprachen in Tools zur Datenaufbereitung

Anwendung	Programmiersprache
Apache Cassandra	Java
Apache Spark	Java, Python, R, Scala
Disco	Erlang, Python
Dryad	C++
DryadLINQ	C#, LINQ, VB
Hadoop	Java
Hive	Java
HPCC	C++, ECL
Hydra	Java
MapReduce	C++
QTConcurrent	QT
TEZ	Java

Tabelle 3.3: Häufigkeit der verwendeten Programmiersprachen in Tools zur Datenaufbereitung

Häufigkeit	Programmiersprache
6	Java
3	C++
2	Python
1	C#, ECL, Erlang, LINQ, QT, R, Scala, VB

Die Betrachtung der aufgelisteten exemplarischen Tools zur Arbeit mit großen Datenbeständen und der Programmiersprachen, die in diesen zum Einsatz kommen, zeigt Überschneidungen mit den ausgewählten Programmiersprachen aus Tabelle 3.1. Aufgrund der großen Verbreitung sowie Verwendung in den angesprochenen exemplarischen Tools, kann die Programmiersprache Java im Kontext von großen Datenbeständen als äußerst relevant bezeichnet werden.
Des Weiteren wird zur Gewichtung der ausgewählten Programmiersprachen im Folgenden betrachtet, wie sich die einzelnen Programmiersprachen in ihrer Ausführungsgeschwindigkeit unterscheiden. Dazu wird zunächst eine nach Sherrington (2015) zitierte Tabelle von unterschiedlichen Benchmarks von Juli 2014 betrachtet. Die in in Tabelle 3.4 getroffenen Angaben bilden relative Angaben zu der Geschwindigkeit gleichartiger Benchmarks der Programmiersprache C.

Chen (2010) führten Benchmarks für die Programmiersprachen C, C++, C# und Java durch. Eine Zusammenfassung dieser Benchmarks findet sich in Tabelle 3.5. Die durchgeführten Benchmarks bedienen sich unterschiedlichen Berechnungen in den Bereichen der Ganzzahlen, der Fließkommazahlen, besonders großer Ganzzahlen, der Trigonometrie und der Geschwindigkeit beim Einlesen und Schreiben von Daten. Die in Tabelle 3.5 notierten Werte bilden dabei Zeiten in Millisekunden (ms) ab.

Tabelle 3.4: Programmiersprachenbenchmarks nach Sherrington (2015, S.4)

	Julia	**Python**	**R**	**JavaScript**	**Go**
fib	0,91	30,37	411,31	2,18	1,0
mandel	0,85	14,19	106,97	3,49	2,36
pi_sum	1,0	16,33	15,42	0,84	1,41
rand_mat_stat	1,66	13,52	10,84	3,28	8,12
rand_mat_mul	1,01	3,41	3,98	14,6	8,51

Tabelle 3.5: Programmiersprachenbenchmarks nach Chen (2010, S.35ff.)

	C	**C++**	**C#**	**Java**
Int arithmetic	14273.5	14275.3	12601.3	8916.9
Double arithmetic	18718.6	18659	17920.8	10322.7
Long arithmetic	33110.9	31781.9	37974.6	27716.4
trigonometrics	13626.8	13462.9	5308.4	67401.5
I/O Benchmark	6052.1	5563	4260	6098.1
Total elapsed time	85781	83742.1	78065.1	120455.6

Couto et al. (2017) führten weiterhin unterschiedliche Benchmarks zum Vergleich verschiedener Programmiersprachen in ihrer Energieeffizienz und Ausführungszeit aus. Als Ergebnis dieser Untersuchung wird Tabelle 3.6 angebeben, welche die Geschwindigkeiten der einzelnen Programmiersprachen in Verhältnis zur Programmiersprache C setzt.

Tabelle 3.6: Programmiersprachenbenchmarks nach Couto et al. (2017, S.4)

Programmiersprache	**Time**
C	1,0
Java	1,65
C#	2,44
Go	2,63
Ruby	44,68
Perl	68,45

Ergänzend zu den bisherigen Betrachtungen erklären Purer (2009), dass PHP und Ruby in ihrer Ausführungsgeschwindigkeit ähnliche Ergebnisse erzielen, weshalb für PHP im Ration zur Programmiersprache C der selbe Faktor angenommen wird, den auch Ruby aufweist und Wells (2015) erklärt, dass Swift in durchgeführten Benchmarks zur Geschwindigkeit beim Sortieren von 1.000.000 Objekten und beim Multiplizieren von $500x500$ Matrizen jeweils 3,9 mal beziehungsweise 1,4 mal schneller als Python war.

Diese Betrachtung unterschiedlicher Benchmarks liefert zunächst keine eindeutige Gewichtung der Programmiersprachen, wird jedoch im weiteren Verlauf für die Abgrenzung der verschiedenen ausgewählten Programmiersprachen verwendet werden.

3.3 Vorstellung der Programmiersprache Julia

Julia ist eine junge Programmiersprache, deren Entwicklung im Jahr 2009 am Massachusetts Institute of Technology begann, sich aber 2012 zu einem Open-Source-Projekt entwickelte. Im Oktober 2019 erhielt Alan Edelman, einer der Entwickler von Julia, den *Sidney Fernbach Award*, der von der *IEEE Computer Society* für herausragende Beiträge zum Einsatz von „high-performace"-Rechnern mittels innovativer Ansätze verliehen wird (IEEE Computer Society 2019). Ursprünglich war Julia als eine Programmiersprache für wissenschaftliche Anwendungszwecke gedacht (vgl. Balbaert 2015; Sherrington 2015; Voulgaris 2016). Dabei zeichnet Julia beispielsweise eine hohe Performanz aus, welche in unterschiedlichen Tests auf dem selben Niveau wie die Programmierspache C liegt (Voulgaris 2016). Weiterhin besitzt Julia eine umfangreiche Basisbibliothek, die viele verschiedene Operationen aus der Linearen Algebra beinhaltet, und bietet Funktionalitäten zur Verbindung mit anderen Programmiersprachen, wie R, Python oder C (Voulgaris 2016). Außerdem wurde Julia eigens für die Verwendung paralleler und verteilter Verarbeitungsmethoden entwickelt (Sherrington 2015). Nicht zuletzt sind Julia und ihre Dokumentation und Tutorials Teil eines Open-Source-Projektes (Voulgaris 2016).
Im Weiteren wird näher auf die Eigenschaften von Julia eingegangen.
Julia's Kern ist in C und C++ implementiert sowie für die Generierung von Maschinencode mit einem *LLVM JIT-Compiler* ausgestattet (Sherrington 2015). Balbaert (2015) beschreibt, dass durch die Generierung von Maschinencode mittels eines JIT-Compilers, Funktionen immer erst dann generiert werden, sobald diese benötigt werden. Zusätzlich werden zum Zeitpunkt der Generierung auch erst die Datentypen der verwendeten Variablen bestimmt. Dabei wird abhängig von den Typen der verwendeten Variablen eine einzigartige Instanz der Funktion generiert. Bei dieser Art der Generierung des Codes bleibt dieser nach seiner ersten Ausführung erhalten, so dass Julia bei jedem darauffolgenden Aufruf der selben Funktion mit Variablen desselben Typs deutlich weniger Zeit für die Ausführung benötigt. Gleichzeitig erhöht dieses Vorgehen die Arbeitsgeschwindigkeit von Julia, da für jeden Datentypen speziell instanziierte Funktionen vorliegen. Julia nutzt ein automatisches Speichermanagement und einen sogenannten Garbage Collector, der selbstständig Objekte entfernt, die nicht mehr verwendet werden, um dem Anwender diese Arbeit abzunehmen und den JIT-Compiler zu unterstützen (Balbaert 2015).
Julia ist keine objektorientierte Programmiersprache im herkömmlichen Sinn. Typen dürfen in Julia keine Methoden besitzen, deshalb ist es nicht möglich Methoden an andere Typen zu vererben (Sherrington 2015). Balbaert (2015) und Sherrington (2015) erklären, dass Julia dem Anwender die Möglichkeit bietet selbst zu entscheiden, ob Variablen im Quelltext Typen zugewiesen werden oder ob Julia diese Arbeit intern überlassen wird. Solange kein spezifischer Typ für eine Variable angegeben ist, kann diese beispielsweise zu einem Zeitpunkt einen Ganzzahl-Wert und zu einem anderen Zeitpunkt einen Text enthalten. Dieses integrierte dynamische Typensystem ist ein Schlüssel für die Geschwindigkeit von Julia (Balbaert 2015, vgl. Sherrington 2015).
In Tabelle 3.7 werden einige grundlegende verfügbaren Typen in Julia zur Übersicht dargestellt. Zu diesen Aufzählungstypen von Julia führt Balbaert (2015) beispielsweise Arrays, Matrizen, Tuples, Dictionarys oder Sets an.

Tabelle 3.7: Julias Datentypen

Typen	
Int8 - Int128	UInt8 - UIn128
Float16 - Float64	BigFloat
Char	String
Bool	Real
Complex Numbers	Date
Aufzählungstypen	

Gleich ob integrierter oder selbst definierter Datentyp, jeder Wert in Julia besitzt einen Datentypen. Auch Datentypen besitzen Datentypen, den sogenannten „DataType“, wodurch Datentypen als Objekte betrachtet werden. Zum Verständnis der Abhängigkeiten in der Typenhierarchie von Julia veranschaulicht Abbildung 3.2 nach Balbaert (2015, S.109) einen Teilbaum der Typenstruktur.

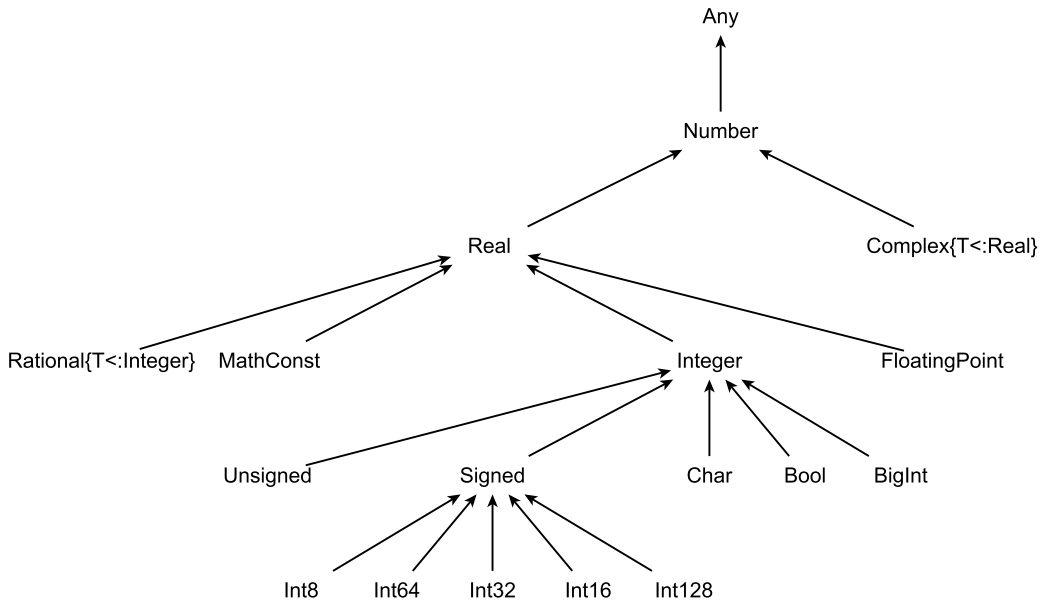

Abbildung 3.2: Baumansicht von Julias Datentypen nach Balbaert (2015, S.109)

Balbaert (2015) und Sherrington (2015) beschreiben Julia als eine funktionale Programmiersprache, da Programme in Julia mittels Funktionen strukturiert und Berechnungen sowie Datentransformationen mittels Funktionen durchgeführt werden und Funktionen auch miteinander verknüpft werden können. Dabei können Funktionen aufgrund der dynamischen Datentypen beliebig oft „überladen“ werden. Der Begriff des „überladens (engl. overloading)“ beschreibt dabei ein Vorgehen, bei dem ein Element der Programmiersprache, hier eine Funktion, unterschiedliche Bedeutungen annehmen kann, die durch den verwendeten Kontext des Elementes definiert wird (Cardelli und Wegner 1985). Es können ebenso dynamische wie optionale Übergabeparameter definiert werden und Funktionen können ebenso anonym definiert werden, um anonyme Funktionen, sogenannte Lambda-Ausdrücke, zu ermöglichen (vgl. Balbaert 2015; Sherrington 2015). Lambda-Funktionen kennzeichnet

beispielsweise, dass sie keinen Funktionsnamen besitzen.
Weiterhin erläutert Balbaert, dass Julia verschiedene Arten von Kontroll-Ausdrücken besitzt, darunter solche wie Bedingungen (*If-ElseIf-Else-End*), Schleifen (*For/While*), Ausnahmebehandlungen (*Exception-Handling*) oder Multitasking mittels sogenannter „Ko-Routinen (engl. coroutines)“ und *Channels*.
Zum Einlesen und Speichern von Daten mit Julia zeigen Balbaert (2015) und Sherrington (2015) unterschiedliche Möglichkeiten auf. Diese Möglichkeiten basieren grundlegend auf dem Prinzip von Datenströmen, also kontinuierlichen Flüssen von Datensätzen aus den Datenquellen, bei denen im Voraus unbekannt ist, wie umfangreich die eintreffenden Daten insgesamt sind. Die unterschiedlichen Möglichkeiten sind beispielsweise das Einlesen und Schreiben von Dateien, aus Netzwerkverbindungen, mittels *TCP/IP-Sockets*, verschiedene Datenbanktreiber (zum Beispiel *ODBC-*, *MySQL-*, *PyCall*-Treiber und weitere), zum Arbeiten mit relationalen wie auch *NoSQL-* und anderen Datenbankformaten (Sherrington 2015).
Desweiteren verfügt Julia über diverse integrierte Methoden zur statistischen Auswertung von Daten, von denen Sherrington (2015) nur einige aufzählt:

- Mittelwertfunktionen
- Zählfunktionen
- Skalarfunktionen
- Bilanzierungsfunktionen (Summary-Functions)
- Ranking-Funktionen
- Korrelationsfunktionen
- Samplingfunktionen
- Empirische Vorhersagemethoden

Weiterhin existieren für Julia unter anderem diverse externe Bibliotheken. Dazu gehören Bibliotheken zur Verarbeitung großer Datenmengen oder zur Arbeit mit unterschiedlichen Dateien und Dateiformaten, Bibliotheken mit statistischen und mathematischen Methoden sowie unterschiedliche Bibliotheken zur Erstellung von grafischen Benutzeroberflächen (GUI, engl. graphical user interface) und zur Verwendung von Julia auf *verteilten Systemen* (JuliaLang.org 2019).
All diese Eigenschaften zeigen auf, dass Julia dem Anwender viele Möglichkeiten zur Arbeit mit Datenbeständen zur Verfügung stellt und durch interne Funktionalitäten unterstützt.

4 Untersuchung von Julia in Hinblick auf die Arbeit mit großen Datenbeständen in der Logistik

Dieses Kapitel behandelt die Untersuchung von Julia und die Abgrenzung der zuvor ausgewählten Programmiergsprachen von Julia und voneinander. Um dieses Ziel zu erreichen werden in Abschnitt 4.1 aus den vorangegangenen Kapiteln Anforderungen an Programmiersprachen zur Verarbeitung großer Datenbestände in der Logistik abgeleitet. Diese Anforderungen dienen anschließend in Abschnitt 4.2 zur Entwicklung von Kriterien an Programmiersprachen. Die Kriterien werden dabei auf die Erfüllung der zuvor abgeleiteten Anforderungen ausgerichtet. Nachdem die Kriterien in Abschnitt 4.2 festgelegt worden sind, werden in Abschnitt 4.3 die unterschiedlichen ausgewählten Programmiersprachen voneinander abgegrenzt. Abschließend wird in Abschnitt 4.4 Julia anhand der durchgeführten Abgrenzung und der entwickelten Kriterien bewertet.

4.1 Ableitung von Anforderungen an Programmiersprachen zur Verarbeitung großer Datenbestände im Kontext der Logistik

Zur Entwicklung von Kriterien, die zur gegenseitigen Abgrenzung der zuvor vorgestellten Programmiersprachen dienen, werden im Folgenden Anforderungen an Programmiersprachen zur Arbeit mit großen Datenbeständen aus den vorangeganen Kapiteln abgeleitet. Diese Anforderungen stellen allgemeine Anforderungen für die Arbeit von Programmiersprachen mit großen Datenbeständen dar und bilden die Grundlage für die Entwicklung von Kriterien zur Abgrenzung der verschiedenen Programmiersprachen voneinander.
Aus der in Abschnitt 2.4 dargestellten Betrachtung ergibt sich, dass die Herausforderungen, welche die Logistik an große Datenbestände stellt, entlang der in Abschnitt 2.3 gezeigten Eigenschaften großer Datenbestände aufzeigbar sind. Anforderungen an Programmiersprachen, welche für die Verarbeitung großer Datenbestände in der Logistik eingesetzt werden sollen, lassen sich daher unter anderem entlang der in Abschnitt 2.3 erläuterten V's ableiten.
Das erste V, das *Volume*, beschreibt, dass große Datenbestände umfangreich sind. Dabei bezieht sich umfangreich nicht nur auf die Menge der Datensätze sondern ebenfalls auf die Menge der Daten je Datensatz. Außerdem lässt sich aus der in Abschnitt 2.1 angestellten Betrachtung der Zusammenhänge zwischen Daten, Information und Wissen ableiten, dass eine Programmiersprache für die Verarbeitung von Datenbeständen im Allgemeinen geeignet sein muss. Durch diese Nutzbarkeit soll für die Programmiersprache die Funktionalität gegeben sein, Daten (vgl. Definition 2.3) in Informationen (vgl. Definition 2.4) überführen zu können. Daher ergibt sich aus der Fähigkeit Daten zu verarbeiten zusammen mit der Eigenschaft des *Volumes* die Anforderung 1.

Anforderung 1 Verarbeitung umfangreicher Datenbestände: Die Programmiersprache soll umfangreiche Datenbestände verarbeiten können.

Das zweite V, die *Variety*, beschreibt, dass große Datenbestände vielfältige Datenformate, also Daten unterschiedlicher Struktur, enthalten. Wie ebenfalls in Kapitel 2 aufgezeigt, kann es sich bei diesen Daten demnach um unstrukturierte, quasi-strukturierte, semistrukturierte oder struktierte Daten handeln. Jedoch soll die Verarbeitung der Daten unabhängig von ihrer jeweiligen Struktur erfolgen können. Weiterhin beschreibt das dritte V, die *Veracity*, dass Daten in unterschiedlichen Qualitäten vorliegen können. Da die Qualität der vorliegenden Daten jedoch ebenfalls keinen Einfluss auf die Verarbeitung der Daten haben soll, lässt sich aus der Variety und der Veracity Anforderung 2 ableiten.

Anforderung 2 Unabhänigigkeit von der Art der Daten: Die Programmiersprache soll Daten unabhängig von der Struktur und der Qualität der Daten verarbeiten können.

Das vierte V, die *Velocity*, beschreibt als abschließend betrachtete Eigenschaft großer Datenbestände, dass große Datenbestände hohe Fluktuationen aufweisen können und Daten deshalb in Echtzeit verarbeitet werden sollen. Zusammen mit der Eigenschaft des *Volume* lässt sich daraus ableiten, dass Programmiersprachen bei der Verarbeitung großer Datenbestände ihre Aufgaben effizient abarbeiten sollen. Der Begriff der Effizienz bezieht sich dabei nicht nur auf möglichst geringe Rechenzeiten und Speicherbedarfe bei der Ausführung von Programmen sondern ebenfalls darauf Programme zuverlässig und mit möglichst wenig Aufwand erstellen zu können. Deshalb wird Anforderung 3 allgemein wie folgt definiert.

Anforderung 3 Effizienz: Die Programmiersprache soll bei der Verarbeitung von großen Datenbeständen effizient sein.

Für die Verwendung einer Programmiersprache zur Arbeit mit großen Datenbeständen spielt die Zugänglichkeit der Sprache eine nicht unbedeutende Rolle, wie in Abschnitt 3.1 erläutert wird. Dort wird erläutert, dass beispielsweise die Art der Lizensierung einer Programmiersprache ausschlaggebend für die Verwendung der Programmiersprache sein kann. Aus der Betrachtung der Verfügbarkeit der Programmiersprachen selbst und der Herleitung der Definition 3.14 lässt sich daher Anforderung 4 ableiten.

Anforderung 4 Verfügbarkeit: Die Programmiersprache soll frei verfügbar sein.

Weiterhin ist für die Verwendung und die Auswahl von Programmiersprachen nicht nur die Verfügbarkeit einer Programmiersprache selbst von Relevanz, sondern vielmehr spielt die Verfügbarkeit von Lernmaterial, also Anleitungen und Tutorials, die dem Anwender zugänglich sind, eine zentrale Rolle, wie Abschnitt 3.2 aufgezeigt hat. Außerdem ist die Verfügbarkeit von Anleitungen, Tutorials und Beispielen, die ein Anwender bezüglich einer Programmiersprache erhalten kann, ein ausschlaggebender Faktor beim Erlernen einer Programmiersprache (vgl. Abschnitt 3.1). Deshalb lässt sich aus der Verfügbarkeit von Lernmatieral Anforderung 5 ableiten.

Anforderung 5 Unterstützung: Für die Programmiersprache soll ein hoher Grad an Unterstützung sowohl in Form von Dokumentation als auch in Form von Anleitungen, Tutorials und Beispielen vorhanden sein.

Die hier abgeleiteten Anforderungen lassen sich in die Kategorien der *funktionalen* und *nichtfunktionalen Anfoderungen* gruppieren, wie Tabelle 4.1 zeigt. *funktionale Anforderungen* beschreiben dabei die Anforderungen, die Funktionalitäten der Programmiersprachen zum Gegenstand haben und *nichtfunktionalen Anfoderungen* beschreiben Anforderungen an die „Qualität“ der geforderten Funktionalitäten und die Programmiersprache selbst.

Tabelle 4.1: Anforderungen an Programmiersprachen für die Arbeit mit großen Datenbeständen

funktionale Anforderungen	nichtfunktionale Anforderungen
Anforderung 1	Anforderung 3
Anforderung 2	Anforderung 4
	Anforderung 5

Nachfolgend werden die abgeleiteten Anforderungen zur Entwicklung geeigneter Kriteren für die Abgrenzung der zuvor ausgewählten Programmiersprachen gegeneinander verwendet.

4.2 Erarbeitung von Kriterien an Programmiersprachen zur Datenverarbeitung anhand der erarbeiten Anforderungen

Zur Abgrenzung der in Abschnitt 3.1 vorgestellten Programmiersprachen gegeneinander werden im Folgenden aus den in Abschnitt 4.1 abgeleiteten Anforderungen Kriterien an Programmiersprachen zur Arbeit mit großen Datenbeständen entwickelt. Die Kriterien sollen dabei auf Grundlage der allgemeinen Anforderungen aus Abschnitt 4.1 entwickelt und für die Eignung der unterschiedlichen Programmiersprachen auf die Verarbeitung großer Datenbestände hin formuliert werden. Außerdem werden die unterschiedlichen Kriterien in drei Kategorien eingeteilt, die abschließend zu diesem Abschnitt in einer Übersicht in Tabelle 4.2 zusammengefasst werden.
Bei Betrachtung von Anforderung 1 ergeben sich, aus Kapitel 2, Kriterien, welche Programmiersprachen für die Arbeit mit großen Datenbeständen erfüllen sollten und welche hier diskutiert werden. Sowohl Abschnitt 2.2 als auch Abschnitt 2.4 zeigen auf, dass Daten zur Analyse oft aus unterschiedlichen Quellen zusammengetragen werden müssen. Diese Quellen können nicht nur logisch oder räumlich voneinander getrennt, sondern ebenfalls gänzlich unterschiedlicher Natur sein, wie Abschnitt 2.4 herausstellt. Daher ergibt sich für Anforderung 1, dass eine Programmiersprache in der Lage sein muss Daten aus verschiedenen Quellen zu extrahieren und zusammenzutragen. Umgekehrt zum Extrahieren und Zusammentragen zeigt Abschnitt 2.2 jedoch ebenfalls, dass eine Programmiersprache ebenfalls die Fähigkeit besitzen muss dem Anwender Daten bereitzustellen. Da große Datenbestände in der Regel in Datenbanken oder Dateien vorliegen (vgl. Abschnitt 2.3 und Abschnitt 2.4) und die verarbeiteten Daten dem Anwender auf der einen Seite in der Form von neuen Datensätzen für Datenbanken oder Dateien bereitgestellt werden können auf der anderen Seite aber auch grafisch aufbereitet werden können, ergeben sich Kriterium 1 und Kriterium 2.

Kriterium 1 Zugriff auf Daten: Die Programmiersprache soll Daten aus großen Datenbeständen extrahieren und bereitstellen können.

Kriterium 2 Grafische Aufbereitung: Die Programmiersprache soll Daten grafisch aufbereiten können.

Zur Erfüllung der Anforderung 1 muss Kriterium 1 erfüllt sein, um Daten aus großen Datenbeständen zur Verarbeitung zu erhalten und dem Anwender bereit zu stellen. Kriterium 2 bildet hingegen ein nicht notwendiges Kriterium ohne Bezug zum Kontext der großen Datenbestände, da die Daten nach der Verarbeitung dem Anwender zwar grafisch bereitgestellt werden können, dies aber keinen Einfluss auf die Fähigkeit der Programmiersprache zur Verarbeitung großer Datenmengen hat.

Aus Anforderung 2 wird deutlich, dass Programmiersprachen zur Verarbeitung von Daten die Fähigkeit besitzen sollen Daten zu transformieren, um unabhängig von der Datenqualität oder des Datentyps handlungsfähig zu sein. Zusätzlich ergibt sich aus Abschnitt 2.2, dass Daten oft nicht in der zur Analyse benötigten Qualität vorliegen und deshalb von der Programmiersprache aufbereitet werden müssen. Für diese Datenaufbereitung müssen die Daten auf unterschiedliche Weise transformiert werden können. Zu diesen Veränderungen zählen Operationen, wie das Zerlegen, das Gruppieren oder das Vergleichen von Daten, die unter dem Begriff der Datentransformation zusammengefasst werden. Daher wird Kriterium 3 wie folgt formuliert.

Kriterium 3 Transformierbarkeit: Die Programmiersprache soll Daten transformieren können.

Aus der in Abschnitt 3.1 angestellten Betrachtung der Programmiersprachen sowie der Definition von Daten und Datentypen in Definition 2.3 geht hervor, dass Programmiersprachen Daten in Variablen speichern, die einen Datentyp benötigen. Damit Programmiersprachen die Arbeit mit beliebigen Datentypen möglich ist, wurde das Konzept des *Castings* in Definition 3.10 festgelegt. Analog zur Transformation von Daten selbst (vgl. Kriterium 3) beinhaltet Kriterium 4 daher transitiv die Transformation von Datentypen.

Kriterium 4 Typenunabhängigkeit: Die Programmiersprache soll beliebige Datentypen verarbeiten können.

Sowohl Kriterium 3 als auch Kriterium 4 bilden notwendige Kriterien für die Erfüllung von Anforderung 2, da sie gemeinsam die gesamte Anforderung abbilden.

Weiterhin führt die Untersuchung von Anforderung 3 zur Betrachtung von Abschnitt 2.3, in dem gezeigt wird, dass das Volumen großer Datenmengen sehr groß ist und, wie in der Herleitung der Definition 3.12 erklärt, die damit verbundene benötigte Rechenkapazität für die Verarbeitung von großen Datenmengen ebenfalls sehr groß ist. Dabei folgt aus Anforderung 3 auf der einen Seite direkt, dass eine Programmiersprache bei die Verarbeitung von großen Datenmengen oder allgemein bei der Ausführung von Programmen effizient sein soll. Auf der anderen Seite folgt aus Anforderung 3 ebenso, dass eine Programmiersprache in ihrer Anwendung und Entwicklung effizient sein soll, also dem Anwender ermöglichen soll mit möglichst wenig Aufwand zuverlässig Programme zu erstellen. Die Effizienz in der Anwendung und Entwicklung lässt sich beispielsweise aus der Herleitung von Definition 3.7 und Definition 3.8 ableiten. Aus diesen beiden Sichtweisen auf Anforderung 3 folgen Kriterium 5 und Kriterium 6.

Kriterium 5 Effiziente Ausführung: Die Programmiersprache soll effizient in ihrer Ausführung sein.

Kriterium 6 Effiziente Anwendung: Die Programmiersprache soll effizient in ihrer Anwendung sein.

Ein weiteres Kriterium für eine Programmiersprache, welche für die Verarbeitung großer Datenmengen verwendet werden und dabei effizient sein soll, bezieht sich auf effiziente Möglichkeiten zur Verarbeitung großer Datenbestände. Dabei werden die Erläuterungen zur Definition 3.12 verwendet, um Kriterium 7 zu definieren.

Kriterium 7 Parallelisierbarkeit: Die Programmiersprache soll die Erstellung von parallelisierbaren Anwendungen ermöglichen.

Für das Erfüllen der Anforderung 3 bildet Kriterium 7 ein notwendiges Kriterium, denn ohne die Möglichkeit zu Parallelisierung von Programmen (vgl. Abschnitt 3.1) können die durch die Eigenschaft des Volumen bezeichneten Mengen an Daten, nicht effizient verarbeitet werden. Sowohl Kriterium 5 als auch Kriterium 6 sind keine notwendigen Kriterien zur Erfüllung von Anforderung 3, allerdings sollte eine Programmiersprache zur Verarbeitung von großen Datenmengen sehr wohl effizient in der Ausführung sein. Durch die Erfüllung von Kriterium 6 wird dem Anwender die Arbeit mit großen Datenbeständen erleichtert, jedoch hat dieses Kriterium keinen direkten Einfluss auf die Arbeit der Programmiersprache mit großen Datenbeständen, da die effiziente Anwendung von Programmiersprachen nur der Arbeit des Anwenders und nicht der Effizenz der Programmiersprache zugutekommt.

Bei der Betrachtung von Anforderung 4 ergibt sich entlang der Herleitung von Definition 3.14, der Defintion von OSS, dass eine Programmiersprache für die Verwendung in der Verarbeitung von großen Datenmengen eine wenig-restriktive Lizensierung besitzen soll. Durch eine freie Lizensierung wird es dem Anwender ermöglicht Programme zu entwickeln, die den jeweiligen spezifischen Anforderungen entsprechen und keinen Einschränkungen aus der Lizensierung der Programmiersprachen unterliegen. Daher wird Kriterium 8 analog zu Anforderung 4 definiert.

Kriterium 8 Verfügbarkeit: Die Programmiersprache soll unter einer OSS-Lizenz verfügbar sein.

Außerdem folgt aus der Untersuchung von Definition 3.13, dass für die Erfüllung von Anforderung 4 die unterschiedlichen Arten von Bibliotheken der Programmiersprachen betrachtet werden müssen. Programmiersprachen besitzen, wie in Abschnitt 3.1 diskutiert, oft integrierte Bibliotheken, die dem Anwender verschiedene Methoden zur Arbeit mit der Programmiersprache an die Hand geben sollen. Des Weiteren können solche Bibliotheken jedoch auch vom Anwender erstellt werden oder von externen Quellen zur Entwicklung von Programmen hinzugezogen werden. Zur Arbeit mit großen Datenbeständen soll eine Programmiersprache deshalb auf der einen Seite bereits integrierte Bibliotheken besitzen, welche die Entwicklung von Programmen zur Verarbeitung und Analyse großer Datenbestände unterstützen. Auf der anderen Seite soll die Programmiersprache dazu fähig sein eigene Bibliotheken zu entwickeln und von anderen Anwendern entwickelte Bibliotheken zur Arbeit mit großen Datenbeständen zu nutzen. Daraus ergeben sich Kriterium 9, Kriterium 10 und Kriterium 11.

Kriterium 9 Interne Bibliotheken: Die Programmiersprache soll integrierte Bibliotheken, zu Datenverarbeitung und -analyse besitzen.

Kriterium 10 Eigene Bibliotheken: Die Programmiersprache soll die Möglichkeit zur Erstellung eigener Bibliotheken für den Anwender besitzen.

Kriterium 11 Externe Bibliotheken: Für die Programmiersprache sollen externe Bibliotheken zur Arbeit mit großen Datenbeständen zur Verfügung stehen.

Da Anforderung 4 keine Notwendigkeit für die Arbeit einer Programmiersprache mit großen Datenbeständen darstellt, sind auch Kriterium 8, Kriterium 9, Kriterium 10 und Kriterium 11 keine notwendigen Kriterien. Weiterhin ist jedoch, wie in Abschnitt 2.4 erklärt worden ist, die Verarbeitung großer Datenbestände ein Thema mit hoher Relevanz in der Logistik, wodurch eine Programmiersprache sowohl Kriterium 9 als auch Kriterium 11 erfüllen sollte. Kriterium 8 und Kriterium 10 hingegen sind Kriterien, die sich nicht auf die direkte Arbeit von Programmiersprachen mit großen Datenbeständen auswirken, sondern dem Andwender lediglich die Arbeit mit der jeweiligen Programmiersprachen erleichtern.

Durch den von Anforderung 5 geforderten hohen Grad an Unterstützung ergeben sich aus der in Abschnitt 3.1 getroffenen Definition 3.15, dass das Vorhandensein von umfassender und nachvollziehbarer Softwaredokumentation eine Grundlage für die Zugänglichkeit zu der Programmiersprache für den Anwender bildet. Daher verlangt Kriterium 12, dass eine Programmiersprache eine solche Softwaredokumentation besitzt.

Kriterium 12 Dokumentation: Die Programmiersprache soll eine Dokumentation besitzen.

Weiterhin beschreibt Anforderung 5, dass eine Programmiersprache für den Anwender nachvollziehbar und einfach zu erlernen sein soll. Aus der Betrachtung der Zugänglichkeit und Erlernbarkeit von Programmiersprachen in Abschnitt 3.1 wird Kriterium 13 als Forderung nach geeigneten Lernutensilien, wie Tutorials oder Beispielen, formuliert.

Kriterium 13 Anleitung: Für die Programmiersprache sollen geeignete Anleitungen, Tutorials und Beispiele vorhanden sein.

Zur Erfüllung der Anforderung 5 bildet Kriterium 12 ein notwendiges Kriterium, da die Existenz von Dokumentation eine Grundlage für die Zugänglichkeit und Verständlichkeit einer Programmiersprache bildet. Kriterium 13 hingegen wird in die Kategorie der Kriterien eingeordnet, die eine Programmiersprache zwar nicht notwendigerweise erfüllen muss, aber erfüllen sollte, da die Existenz von geeigneten Tutorials und Beispielen die Zugänglichkeit und die Erlernbarkeit einer Programmiersprache zwar positiv beeinflussen, jedoch keine Grundlage für die Zugänglichkeit ist.
Die erarbeiteten Kriterien lassen sich wie einleitend ausgeführt in drei Kategorien einordnen. Die erste Kategorie bilden die notwendigen Kriterien, die eine Programmiersprache erfüllen muss. Die zweite Kategorie besteht aus den Kriterien, die zur Erfüllung der Anforderungen zwar nicht notwendig sind, aber dennoch von der jeweiligen Programmiersprache erfüllt werden sollten, um die Arbeit mit großen Datenbeständen zu vereinfachen. Die letzte Kategorie bilden die Kriterien, welche weder notwendig noch abhängig von der Thematik der großen Datenbestände sind. Diese Kriterien wurden aus den Anforderungen (vgl.

Abschnitt 4.1) an eine Programmiersprache im Kontext großer Datenbestände der Logistik herausgearbeitet.
Entlang der beschriebenen Kategorisierung werden die einzelnen Kategorien wie folgt benannt:

- Die Kategorie der notwendigen Kriterien wird als „must-have“ bezeichnet.
- Die Kategorie der nicht notwendigen, aber wünschenswerten Kriterien wird als „should-have“ bezeichnet.
- Die Kategorie der nicht nicht notwendigen und unabhängig von der Thematik großer Datenbestände existierenden Kriterien wird als „nice-to-have“ bezeichnet.

Eine Übersicht über alle Kategorien und die ihnen zugeordneten Kriterien wird in Tabelle 4.2 dargestellt.

Tabelle 4.2: Übersicht über Kriterien für Programmiersprachen für die Arbeit mit großen Datenbeständen

„must-have“	**„should-have“**	**„nice-to-have“**
Kriterium 1	Kriterium 5	Kriterium 2
Kriterium 3	Kriterium 9	Kriterium 6
Kriterium 4	Kriterium 11	Kriterium 8
Kriterium 7	Kriterium 13	Kriterium 10
Kriterium 12		

4.3 Abgrenzung der vorgestellten Programmiersprachen anhand der Kriterien

Die vorgestellten Programmiersprachen werden im Folgenden anhand der erarbeiteten Kriterien abgegrenzt. Dazu werden die einzelnen Kriterien der Reihe nach abgearbeitet. Inwiefern eine Programmiersprache ein einzelnens Kriterium erfüllt wird in Form von prozentualen Angaben festgehalten. Wird ein Kriterium beispielsweise durch eine spezifische Eigenschaft der Programmiersprachen erfüllt oder nicht erfüllt, so kann eine Programmiersprache dieses Kriterium zu 0% oder 100% erfüllen. Wird ein Kriterium durch mehrere Eigenschaften erfüllt, so ergeben sich die prozentualen Angaben aus der Anzahl der mögichen Eigenschaften. Zur übersichtlicheren Notation wird im Folgenden die Bezeichnung Erfüllungsgrad (DoF, engl. degree of fullfillment) verwendet. Damit wird am Ende dieses Abschnitts, in Tabelle 4.7, eine Übersicht über alle Kriterien und Programmiersprachen gegeben werden, welche anhand der in Tabelle 4.2 getroffenen Kategorisierung der Kriterien gewichtet wird.

Kriterium 1: Zugriff auf Daten

Zur Verarbeitung großer Datenbestände müssen Programmiersprachen Daten sowohl aus großen Datenbeständen extrahieren können als auch die verarbeiteten Daten oder Ergebnisse dem Anwender bereitstellen können (vgl. Abschnitt 2.2). Da solche Datenbestände

in der Regel in Datenbanken vorliegen und die verarbeiteten Daten in Form von neuen Datensätzen bereitgestellt werden können, wird auf der einen Seite zur Erfüllung von Kriterium 1 die Fähigkeit der unterschiedlichen Programmiersprachen zur Arbeit mit Datenbanken betrachtet. Auf der anderen Seite wird die Fähigkeit der Programmiersprachen zur Verwendung von Datenströmen, also kontinuierlichen Flüssen von Daten (vgl. Abschnitt 2.1), zur Erfüllung von Kriterium 1 betrachtet, da die Menge der zu verarbeitenden Daten in der Regel sehr groß ist (vgl. Abschnitt 2.3 und Abschnitt 2.4). Dabei werden nur Programmiersprachen mit integrierten Bibliotheken zur Arbeit mit Datenbanken oder Datenströmen in die Betrachtung genommen, da die Verwendung externer Bibliotheken zur Verwendung von Datenbanken oder Datenströmen in Kriterium 11 betrachtet werden. Konkret wird die Möglichkeit Bibliotheken mit den beschriebenen Funktionalitäten zu verwenden mit jeweils 50% des DoF festgelegt.
Die Programmiersprachen C und C++ besitzen auf der einen Seite zwar keine integrierten Bibliotheken zur Verbindung mit Datenbanken, auf der anderen Seite jedoch sehr wohl in die Basisbibliotheken integrierte Funktionalitäten für die Verwendung von Datenströmen (vgl. Tabelle B.1 und Tabelle B.2). Daraus ergeben sich für die Programmiersprachen C und C++ jeweils ein DoF von 50%.
Ebenso verfügt die Programmiersprache Ruby über keine Bibliotheken zur Verbindung mit Datenbanken, jedoch gehören Bibliotheken für die Verwendung von Datenströmen zum Standard (vgl. Tabelle B.11). Daraus ergibt sich für die Programmiersprache Ruby ein DoF von 50%.
Die weiteren Programmiersprachen, namentlich C#, Go, Java, Julia, Perl, PHP, Python, R und Swift, besitzen sowohl integrierte Bibliotheken zur Verbindung mit Datenbanken als auch für die Verwendung von Datenströmen (vgl. Anhang B). Daraus ergibt sich für diese Programmiersprachen ein DoF von 100%.

Kriterium 2: Grafische Aufbereitung

Die Analyseergebnisse, der von der Programmiersprache verarbeiteten Daten, sollen dem Anwender grafisch aufbereitet zur Verfügung gestellt werden. Doch damit eine Programmiersprache Elemente grafisch darstellen kann, benötigt sie Bibliotheken für die Erstellung einer GUI. Für die Erfüllung von Kriterium 2 wird betrachtet, ob eine Programmiersprache integrierte Bibliotheken, externe Bibliotheken oder keine Bibliotheken für die Erstellung von GUIs besitzen. Die Bewertung erfolgt dabei indem Programmiersprachen mit integrierten Bibliotheken zur Erstellung von GUIs einen DoF von 100%, Programmiersprachen mit externen Bibliotheken einen DoF von 50% und die übrigen Sprachen einen DoF von 0% erhalten. Die Verwendung von externen Bibliotheken wird hier in der Bewertung herabgestuft, da sich Kriterium 11 eigens mit der Existenz externer Bibliotheken beschäftigt.
Die Programmiersprachen C, C++, JavaScript, Julia und Perl besitzen keine integrierten Bibliotheken zur Erstellung von GUIs. Für jeder dieser Programmierspachen existieren jedoch mehr oder weniger umfangreiche externe Bibliotheken zur Entwicklung von GUIs (vgl. Anhang B). Daraus ergibt sich für die aufgezählten Programmiersprachen jeweils ein DoF von 50%.
Die Programmiersprachen C#, Go, Java, PHP, Python, R, Ruby und Swift besitzen hingegen allesamt integrierte Bibliotheken zur Erstellung von GUIs (vgl. Anhang B), woraus sich für diese Programmiersprachen jeweils ein DoF von 100% ergibt.

Kriterium 3: Transformierbarkeit

Kriterium 3 verlangt von den Programmiersprachen die Fähigkeit Daten transformieren zu können. Dabei beschreibt das Transformieren unter anderem Operationen wie das Zerlegen, das Gruppieren oder das Vergleichen von Daten (vgl. Abschnitt 4.2). Zur Ausführung solcher Operationen muss die entsprechende Programmiersprache in der Lage sein Daten zu interpretieren und die korrekten Basisoperationen auf die Daten anzuwenden, wie das Summieren oder Multiplizieren von Werten oder das logische Vergleichen zweier Ausdrücke. Zur Erfüllung dieses Kriteriums kann eine Programmiersprache dabei ebenso integrierte Standardbibliotheken für die Transformation von Daten besitzen wie auch dem Anwender lediglich die nötigen Kontrollstrukturen an die Hand geben, um die Transformation von Daten zu ermöglichen, um dieses Kriterium zu erfüllen. Die beschriebene grundlegende Verarbeitung von Daten gehört zu den fundamentalen Eigenschaften der Hochsprachen, die in Abschnitt 3.2 ausgewählt wurden, und wird daher von jeder ausgewählten Sprache erfüllt. Daher ergibt sich für Kriterium 3 für jede der Programmiersprachen ein DoF von 100%.

Kriterium 4: Typenunabhängigkeit

Zur Einordnung der ausgewählten Programmiersprachen in Kriterium 4 wird die in Abschnitt 3.1 eingeführte Klassifizierung von Typensystemen verwendet. Dabei werden Typensysteme von Programmiersprachen mittels drei verschiedener Kategorien klassifiziert:

- stark - schwach
- statisch - dynamisch
- explizit - implizit

Die erste Kategorie, „stark - schwach“ bezieht sich darauf, ob eine Programmiersprache sicherstellt, dass auf einer Variablen mit einem Datentyp nur solche Operationen durchgeführt werden, die ihrem Typ entsprechen oder nicht. Stellt eine Sprache dies sicher, so besitzt sie eine *starke Typisierung*, lässt sie hingegen typenfremde Operationen zu, so besitzt sie eine *schwache Typisierung*. Die zweite Kategorie, „statisch - dynamisch“, teilt die Programmierspachen in Gruppen von Sprachen, bei denen die Überprüfung der Datentypen zum Zeitpunkt des Übersetzens des Programms (*statisch*) oder zur Laufzeit (*dynamisch*) durchgeführt wird. Die dritte und letzte Kategorie, „explizit - implizit“, gruppiert die Programmiersprachen danach, ob Datentypen für Variablen bei der Entwicklung zwingend und explizit angegeben werden müssen oder ob diese implizit zur Laufzeit ermittelt werden. Um die unterschiedlichen in Abschnitt 3.2 ausgewählten Programmiersprachen anhand der Klassifizierung der Typensysteme einordnen zu können, wird eine Gewichtung der Ausprägungen *stark*, *schwach*, *statisch*, *dynamisch*, *explizit* und *implizit* vorgenommen. Aus Abschnitt 2.3 und Abschnitt 2.4, der Betrachtung der Eigenschaften großer Datenbestände und Datenbeständen im Kontext der Logistik, geht hervor, dass die Datenbestände, auf denen die hier ausgewählten Programmiersprachen arbeiten sollen auf der einen Seite sehr umfangreich sind, auf der anderen Seite jedoch ebenso vielen Veränderungen unterliegen. Außerdem enthalten die Datenbestände, welche die betrachteten Programmiersprachen verwenden, vielfältige Datenformate. Daraus ergibt sich, für die hier angestellte Untersuchung, dass eine Programmiersprache mit einer Ausprägung der Klassifizierung ihres Typensystems, die weniger Aufwand für die Verarbeitung solcher Datenmengen erzeugt, besser für den Einsatz im Kontex großer Datenbestände in der Logistik geeignet ist. Das schließt

jedoch ein, dass keine Ausprägung der Klassifizierung des Typensystems völlig ungeeignet für die Verwendung mit großen Datenmengen ist, da eine ungeeignetere Ausprägung nur den Aufwand beim Erstellen eines Programms erhöht, das auf großen Datenbeständen arbeiten soll.
Da die beschriebenen großen Datenbestände also vielfältige Datenformate enthalten und ständigen Änderungen unterliegen, sollte eine Programmiersprache mit einer *dynamischen* Typisierung bevorzugt werden, um dem Anwender das Entwickeln von Programmen auf Datensätzen mit beliebigen Datentypen zu erleichtern. Des Weiteren sollte eine Programmiersprache eine *starke* Typisierung bevorzugen, da dadurch die Geschwindigkeit von Programmen bei der Ausführung effizienter ist. Von Vorteil bei Sprachen mit *starker* Typisierung sind Verfahren wie die Typinferenzen (vgl. Abschnitt 3.1), wodurch der Anwender keine expliziten Datentypen bei der Programmierung angeben muss. Um dem Anwender die Programmierung zu erleichtern, sollten Programmiersprachen mit *impliziten* Typensystemen bevorzugt werden. Die Klassifizierung in Programmiersprachen mit *expliziten* und *impliziten* Typensystemen trägt, anders als die anderen beiden Kategorien, nicht direkt zur Arbeit mit großen Datenbeständen oder Datenbeständen in der Logistik bei. Daher wird diese Kategorie weniger stark gewichtet, als die anderen beiden Kategorien.
Es ergibt sich für den jeweiligen DoF folgende Einteilung der Ausprägungen der Klassifizierung von Typensystemen von Programmiersprachen. Die Bewertung der ersten und der zweiten Kategorie werden mit jeweils 40% und die Bewertung der dritten Kategorie mit 20% festgelegt. Programmierspachen mit einer *schwachen* Typisierung werden in der ersten Kategorie mit 20% bewertet. Solche Sprachen mit einer *starken* Typisierung hingegen mit 40%. In der zweiten Kategorie werden Programmiersprachen mit einem *dynamischen* Typenkonzept mit 40% und Sprachen mit einem *statischen* Typenkonzept mit 20% bewertet. In der dritten Kategorie erfolgt die Bewertung für Sprachen mit einem *impliziten* Typensystem mit 20% und für Sprachen mit einem *expliziten* Typensystem mit 10%. Abschließend werden die Bewertungen der einzelnen Kategorien addiert und ergeben so den DoF der jeweiligen Programmiersprache für Kriterium 4.

- Die Kategorisierung der Typisierung der Programmiersprachen C, C++, Go und Java sind stark, statisch und explizit, jedoch mit der Möglichkeit des impliziten Castings (vgl. Anhang B). Daraus ergibt sich für die Programmiersprachen jeweils ein DoF von 70%.
- Die Kategorisierung der Typisierung der Programmiersprache C# ist stark, statisch und explizit, unterstützt jedoch ebenfalls optional Typinferenz (vgl. Tabelle B.3). Daraus ergibt sich für die Programmiersprache C# ein DoF von 80%.
- Die Kategorisierung der Typisierung der Programmiersprachen Julia und Ruby sind stark, dynamisch und implizit durch Typinferenz (vgl. Abschnitt 3.3 und Tabelle B.11). Daraus ergiben sich für die Programmiersprachen jeweils DoFs von 100%.
- Die Kategorisierung der Typisierung der Programmiersprachen JavaScript, Perl, PHP und R sind schwach, dynamisch und implizit (vgl. Anhang B). Daraus ergibt sich für die Programmiersprachen jeweils ein DoF von 80%.
- Die Kategorisierung der Typisierung der Programmiersprache Python und Swift ist stark, statisch und implizit (vgl. Anhang B). Daraus ergeben sich für die Programmiersprachen DoFs von 80%.

Kriterium 5: Effiziente Ausführung

Kriterium 5 erfordert die Betrachtung der Ausführungsgeschwindkeiten der ausgewählten Programmiersprachen. Daher werden im Folgenden verschiedene Geschwindigkeitstests, sogenannte Benchmarks für die ausgewählten Programmiersprachen (vgl. Abschnitt 3.2) herangezogen. Die Benchmarks wurden von unterschiedlichen Autoren erläutert. Zur Abgrenzung der unterschiedlichen Programmiersprachen werden die Zeiten aus den Benchmarks in Relation zu der Programmierspache C gesetzt, um einen relativen Wert zur Vergleichbarkeit der Programmiersprachen zu erzeugen.
Zur Untersuchung der Geschwindigkeiten werden die in Abschnitt 3.2 dargestellten unterschiedlichen Benchmarks genauer betrachtet. Zunächst dient Tabelle 3.4 als erster Indikator für das Kriterium 5. Da sich die Angaben in Tabelle 3.4 mittels relativer Angaben auf die Geschwindigkeiten gleichartiger Benchmarks der Programmiersprache C beziehen, lässt sich diese Art der Betrachtung als eine Basis für den Vergleich der unterschiedlichen Geschwindigkeiten nutzen. Um einen Wert für die abschließende Abgrenzung zu erhalten, wird für jede Programmiersprache in Tabelle 3.4 das arithmetische Mittel über die unterschiedlichen Benchmarks gebildet und so ein Durchschnittswert errechnet. Die resultierenden Werte sind letztendlich Faktoren, die angeben wieviel schneller oder langsamer eine Programmiersprache als die Programmiersprache C ist. Daraus folgt, dass Sprachen, die schneller als die Sprache C sind, einen Faktor < 1 und Sprachen, die langsamer als die Sprache C sind, einen Faktor > 1 besitzen. Die gesamte Übersicht über die Benchmarks aus Tabelle 3.4 und die Berechnung der Mittelwerte wird in Tabelle 4.3 dargestellt.

Tabelle 4.3: Gemittelte Geschwindigkeiten aus Tabelle 3.4

	Julia	**Python**	**R**	**JavaScript**	**Go**
fib	0,91	30,37	411,31	2,18	1,0
mandel	0,85	14,19	106,97	3,49	2,36
pi_sum	1,0	16,33	15,42	0,84	1,41
rand_mat_stat	1,66	13,52	10,84	3,28	8,12
rand_mat_mul	1,01	3,41	3,98	14,6	8,51
Avg.	1,09	15,56	109,7	4,88	4,28

Desweiteren werden die Benchmarks aus Tabelle 3.5 zur Betrachtung herangezogen. Die in Tabelle 3.5 notierten Geschwindigkeiten werden zum Übertragen in das Bewertungsschema, wie bei der vorhergegangenen Tabelle 4.3, in Relation zur Geschwindigkeit der Programmiersprache C gesetzt. Dazu werden die aufsummierten Zeiten unter „Total elapsed time“ umgerechnet, sodass jede Sprache in Relation zu C betrachtet werden kann.

Außerdem werden die Ergebnisse aus Tabelle 3.6 und die unterschiedlichen Aussagen von Purer (2009) und Wells (2015) in Abschnitt 3.2 verwendet, um die weiteren Programmiersprachen in das Kriterium 5 einzuordnen. Daraus ergibt sich für die Programmiersprache PHP ein relativer Faktor von 44,68 für die Geschwindigkeit der Programmiersprache in ihrer Ausführung. Für die Programmiersprache Swift errechnet sich, dass Swift 2,65 mal langsamer als die Programmiersprache C ist. Da aus den durchgeführten Benchmarks zur Geschwindigkeit beim Sortieren von 1.000.000 Objekten und beim Multiplizieren von $500x500$ Matrizen, Swift jeweils 3,9 mal beziehungsweise 1,4 mal schneller als Python war.

Tabelle 4.4: Gemittelte Geschwindigkeiten aus Tabelle 3.5

	C	**C++**	**C#**	**Java**
Int arithmetic	14273.5	14275.3	12601.3	8916.9
Double arithmetic	18718.6	18659	17920.8	10322.7
Long arithmetic	33110.9	31781.9	37974.6	27716.4
trigonometrics	13626.8	13462.9	5308.4	67401.5
I/O Benchmark	6052.1	5563	4260	6098.1
Total elapsed time	85781	83742.1	78065.1	120455.6
Verhältnis zu C	1	0,98	0,91	1,4

Unter Berücksichtigung der relativen Geschwindikeit von Python zu C, errechnet sich im Mittel der Wert von 2,65.

Alle Geschwindigkeiten in Relation zur Geschwindigkeit der Programmiersprache C vereint werden in Tabelle 4.5 dargestellt. Bei den Programmiersprachen, die zuvor in unterschiedlichen Benchmarks mit verschiedenen Geschwindigkeiten aufgeführt worden sind, wird das artithmetische Mittel genutzt, um die relative Geschwindigkeit der Programmiersprache zu C zu bestimmen. Desweiteren werden die relativen Geschwindigkeiten mittels $x[i] = \frac{x[i]-min(x)}{max(x)-min(x)} * 100\%$ normiert, wobei x die Menge aller Geschwindigkeiten und $x[i]$ die jeweilige Geschwindigkeit der einzelnen Programmiersprachen darstellt. Dieser normierte Wert geht abschließend als DoF in die Abgrenzung ein. So haben die schnellsten Sprachen einen DoF von 100% und die langsamsten Sprachen einen DoF von 0%.

Tabelle 4.5: Geschwindigkeiten der Programmiersprachen in Relation zu C

Programmiersprache	**Geschwindigkeit**	**DoF**
C	1	100%
C++	0,98	100%
C#	1,68	99%
Go	3,46	98%
Java	1,53	99%
Java Script	4,88	96%
Julia	1,09	100%
Perl	68,45	38%
PHP	44,68	60%
Python	15,56	87%
R	109,7	0%
Ruby	44,68	60%
Swift	2,65	98%

Kriterium 6: Effiziente Anwendung

Kriterium 6 beinhaltet die Forderung nach Effizienz der Programmiersprachen in ihrer Anwendung. Zur Einordnung der verschiedenen ausgewählten Programmiersprachen werden in diesem Kontext die unterschiedlichen Arten der Übersetzung von Quelltext in Maschinencode betrachtet. Die betrachteten Sprachen teilen sich dabei in die Gruppe der Sprachen, die von *Interpretern*, und in die Gruppe der Sprachen, die von *Compilern* übersetzt werden (vgl. Abschnitt 3.1). Interpreter übersetzen den Quelltext von Programmen Zeile für Zeile in Maschinencode und führen diesen direkt aus, wohingegen Compiler das vollständige Programm in Maschienencode überführen und üblicherweise in separaten Dateien speichern. Diese Maschinencode-Dateien können direkt von Rechnern ausgeführt werden. Daraus folgt, dass von Interpretern übersetze Programme schneller in der Übersetzung und direkten Ausführung sind. Soll ein Programm jedoch wiederholt ausgeführt werden, so folgt aus der Art der Übersetzung durch Compiler, dass von Compilern übersetzte Programme effizienter in der Ausführung sind, da zu jedem Aufruf der Quelltext nicht erneut in Maschinencode übersetzt werden muss. Dies schlägt sich zwar ebenfalls im Speicherverbrauch der unterschiedlichen Varianten nieder, da Rechner zur Ausführung von Programmen, die durch Interpreter übersetzt werden, auf den Quelltext des Programmes angewiesen sind. Doch für die hier angestellte Betrachtung zur Einordnung der Programmiersprachen in Kriterium 6 ist nur die Geschwindkeit der Übersetzung des jeweiligen Programms ausschlaggebend. Die Unterschiede, welche die Compiler der verschiedenen Programmiersprachen oder die unterschiedlichen Compiler, die für einzelne Programmiersprachen existieren, werden in dieser Untersuchung nicht betrachtet.
Da die Übersetzung durch einen Compiler langsamer vonstattengeht als die Übersetzung durch einen Interpreter wird der DoF von durch Compilern übersetzten Programmiersprachen auf 50% festgelegt. Programme, die von Interpretern übersetzt werden erhalten jeweils einen DoF von 100%.
Daraus folgt, dass die Programmiersprachen C, C++, C#, Go, Java, Julia, Python und Swift einen DoF von 50% für Kriterium 6 besitzen, da sie von Compilern übersetzt werden. Die Programmiersprachen JavaScript, Perl, PHP, R und Ruby werden mittels Interpretern übersetzt, weshalb der jeweilige DoF 100% beträgt.

Kriterium 7: Parallelisierbarkeit

Wie in Abschnitt 3.1 diskutiert benötigt eine Programmiersprache zur effizienten Verarbeitung von großen Datenmengen die Fähigkeit Prozesse zu parallelisieren. Daher wird Kriterium 7 zur Abgrenzung der jeweiligen Programmiersprachen gegen die anderen ausgewählten Sprachen im Folgenden betrachtet. Die Programmiersprachen die Fähigkeit zur Parallelisierung von Prozessen unterstützen, besitzen einen DoF von 100%, jede andere Programmiersprache besitzt einen DoF von 0%.
Die unterschiedlichen Programmiersprachen verwenden verschiedene Begrifflichkeiten für die Strukturen zur Parallelisierung von Prozessen, beispielsweise „Tasks“ oder „Threads“ und wie auch in der Herleitung für die Parallelität (Definition 3.12) erläutert gibt es unterschiedliche Arten der Parallelität (vgl. Abschnitt 3.1). Doch die Erfüllung des Kriterium 7 erfolgt unabhängig davon, ob ein Programm mittels einer *Datenparallelität* oder einer *Algorithmenparallelität* umgesetzt worden ist, da diese Unterscheidung erst durch die vom Anwender durchgeführte Entwicklung des Programms entschieden werden kann. Zur Erfüllung dieses Kriterium wird dementsprechend lediglich betrachtet, ob die jeweiligen Programmiersprachen die Entwicklung parallelisierbarer Anwendungen ermöglicht. Weil je-

de der in Abschnitt 3.2 ausgewählten Programmiersprachen dem Anwender die Möglichkeit bietet parallelisierbare Anwendungen zu entwickeln, auch wenn die Benennung der parallelen Strukturen unterschiedlich lautet, erfüllt jede der ausgewählten Programmiersprachen das Kriterium 7. Daraus folgt, dass der DoF für jede der Sprachen 100% beträgt.

Kriterium 8: Verfügbarkeit

Zur Entwicklung von Anwendungen zur Verarbeitung großer Datenbestände fordert Kriterium 8, dass eine Programmiersprache unter einer OSS-Lizenz verfügbar ist. Um die einzelnen Programmiersprachen einzuordnen erfüllen Sprachen, die vollständig unter OSS-Lizenzen verfügbar sind dieses Kriterium vollständig (DoF: 100%). Solche Sprachen, die nur teilweise unter einer OSS-Lizenz verfügbar sind erfüllen das Kriterium zur Hälfte (DoF: 50%) und Programmiersprachen, die ausschließlich unter proprietären Lizenzen verfügbar sind erfüllen Kriterium 8 dementsprechend nicht (DoF: 0%).

- Die Programmiersprache C ist eine frei verfügbare Programmiersprache, die von der American National Standards Institute (ANSI) und der International Organization for Standardization (ISO) standadisiert worden ist (vgl. Tabelle B.1). Daraus ergibt sich für die Programmiersprache C ein DoF von 100%.
- Die Programmiersprache C++ ist genau wie C eine frei verfügbare Programmiersprache, die von der ISO standadisiert worden ist (vgl. Tabelle B.2). Daraus ergibt sich für die Programmiersprache C++ ein DoF von 100%.
- Die Programmiersprachen Go, JavaScript und Ruby sind alle unter der Berkeley Software Distribution Lizenz (BSD-Lizenz), einer OSS-Lizenz verfügbar (vgl. Anhang B). Daraus ergibt sich für die Programmiersprache Go ein DoF von 100%.
- Die Programmiersprachen Java, Perl und R sind unter einer OSS-Lizenz namens GNU GPL verfügbar (vgl. Anhang B). Daraus ergibt sich für die Programmiersprache Java ein DoF von 100%.
- Sowohl unter der GNU GPL als auch unter der BSD-Lizenz ist die Programmiersprache Julia verfügbar (vgl. Abschnitt 3.3). Daraus ergibt sich für die Programmiersprache Julia ebenfalls ein DoF von 100%.
- Swift ist unter der „Apache-Lizenz 2.0“ einer OSS von Apache verfügbar. (vgl. Tabelle B.12). Daraus ergibt sich für die Programmiersprache Swift ein DoF von 100%.
- Python nutzt für die Lizensierung ihre eigene OSS-Lizenz, die Python-Software-Foundation-Lizenz (PSF-Lizenz) (vgl. Tabelle B.9). Daher ergibt sich auch für die Programmiersprache Python ein DoF von 100%.
- Die Programmiersprache PHP hingegen ist zum Teil unter der PHP eigenen OSS-Lizenz verfügbar, steht zum Teil jedoch auch unter proprietärer Lizenz (vgl. Tabelle B.8). Daraus ergibt sich für die Programmiersprache PHP ein DoF von 50%.
- Anders als die anderen Sprachen wird C# letzendlich von Microsoft vertrieben und ist damit unter einer proprietären Lizenz verfügbar (vgl. Tabelle B.3). Daraus ergibt sich für die Programmiersprache C# ein DoF von 0%.

Kriterium 9: Interne Bibliotheken

Weiterhin verlangt Kriterium 9 von einer Programmiersprache, dass diese hilfreiche Bibliotheken zur Verarbeitung großer Datenmengen besitzt. Dazu werden die unterschiedlichen Programmiersprachen betrachtet und integrierte Bibliotheken zu den folgenden Bereichen fließen in Erfüllung dieses Kriteriums ein: Bibliotheken zur Arbeit mit Dateien, zur Arbeit mit Datenbanken, mit mathematischen Basisfunktionen oder Funktionen aus der Statistik. Außerdem werden Bibliotheken für Netzwerkkonnektivitäten betrachtet. Zur Bestimmung des DoF werden die unterschiedlichen genannten Arten von Bibliotheken zu gleichen Teilen gewichtet, woraus sich ein DoF von 20% für jedes der Themengebiete der Bibliotheken ergibt.
Die Programmiersprache C besitzt integrierte Bibliotheken für Mathematik, das Arbeiten mit Dateien und grundlegende Funktionen für Netzwerkverbindungen (vgl. Tabelle B.1). Daraus ergibt sich für die Programmiersprache C ein DoF von 60%.
C++ besitzt genau wie C integrierte Bibliotheken für Mathematik, das Arbeiten mit Dateien und Bibliotheken sowohl mit grundlegenden Funktionalitäten für das Suchen und Sortieren als auch grundlegende numerische Funktionen für Netzwerkverbindungen (vgl. Tabelle B.2). Daraus ergibt sich für die Programmiersprache C++ ebenfalls ein DoF von 60%.
C# reiht sich in seinen Funktionalitäten in die Reihe der Sprachen der C-Familie ein und ergänzt die Liste der integrierten Bibliotheken mit Funktionalitäten zur Datenbankanbindung(vgl. Tabelle B.3). Daher besitzt C# ein DoF von 100%.
Die Programmiersprachen Go und JavaScript unterstützen den Anwender sowohl mit Bibliotheken zur Arbeit mit Dateien, Datenbanken und Netzwerkkonnektivität als auch mit Bibliotheken mit mathematischen Funktionen (vgl. Anhang B). Daraus ergeben sich für diese Programmiersprachen DoFs von 80%.
Für die Programmiersprachen Java, Perl und Ruby existieren integrierte Bibliotheken zur Verwendung von Dateien, zur Anbindung von Datenbanken und für die Entwicklung von Netzwerkschnittstellen, wie auch eine Bibliothek mit grundlegenden mathematischen Funktionen (vgl. Anhang B), weshalb alle drei Programmiersprachen einen DoF von 80% besitzen.
Julia bietet dem Anwender zur Unterstützung bei der Entwicklung von Programmen zur Arbeit mit großen Datenbeständen interne Bibliotheken zur Verwendung von Dateien, zur Anbindung an Datenbanken und zur Verwendung von Netzwerken. Zudem werden dem Anwender Funktionen aus den Bereichen der Mathematik und der Statistik zur Verfügung gestellt (vgl. Abschnitt 3.3). Daraus ergibt sich für Julia ein DoF von 100%.
PHP und Python unterstützen den Anwender ebenfalls mit Funktionalitäten zur Arbeit mit Dateien und Datebanken und es gibt genau wie bei Julia umfangreiche Bibliotheken zur Verwendung von Netzwerken und mit Funktionen aus der Mathematik und der Statistik (vgl. Anhang B). Deshalb erhalten PHP und Python ebenfalls einen DoF von 100%.
Die Programmiersprache R nimmt eine exponierte Position unter den aufgeführten Sprachen ein, da sie dem Anwender sehr umfangreiche Bibliotheken zur statistischen Auswertung von Daten und diverse mathematische Funktionen bietet. Außerdem wird die Verwendung von Dateien und die Anbindung an Datenbanken über integrierte Bibliotheken unterstützt (vgl. Tabelle B.10). Daraus ergibt sich für die Programmiersprache R ein DoF von 80%.
Letztlich besitzt die Programmiersprache Swift Bibliotheken zur Anbindung an Netzwerke, zur Verwendung von Dateien und es werden grundlegende mathematische Funktionen bereitgestellt (vgl. Tabelle B.12), weshalb Swift einen DoF von 60% besitzt.

Kriterium 10: Eigene Bibliotheken

Kriterium 10 befasst sich mit der Möglichkeit, dass der Anwender selbst in der Lage sein soll Bibliotheken für die jeweilige Programmiersprache zu erstellen. Die Erstellung eigener Bibliotheken ermöglicht es dem Anwender dabei nicht nur die eigenen Programme zu strukturieren und abgeschlossene, thematisch zusammenhängende, Sammlungen von Methoden zu entwickeln, sondern vereinfacht so ebenfalls entwickelte Bibliotheken zu einem Thema oder einer Anwendungsdomäne anderen Anwendern zur Verfügung zu stellen (vgl. Abschnitt 3.1). Dabei werden Bibliotheken von den unterschiedlichen, in Abschnitt 3.2 ausgewählten, Programmierspachen auch Module (engl. modules) oder Pakete (engl. package) genannt. Die verschiedenen Formen in denen Bibliotheken gebildet oder interpretiert werden reichen von der Nutzung eigens definierter expliziter Dateiformate für Bibliotheken bis zur simplen Verwendung der konkreten Quelltext-Dateien mit den vordefinierten Methoden. Unabhängig von der Form und der Benennung der Bibliotheken bietet jedoch jede der ausgewählten Programmiersprachen dem Anwender die Möglichkeit eigene Bibliotheken zu erstellen. Somit erfüllt jede der in Abschnitt 3.2 ausgewählten Programmiersprachen Kriterium 10 und der DoF beträgt für jede Sprache 100%.

Kriterium 11: Externe Bibliotheken

Zur Erfüllung von Kriterium 11 werden verfügbare externe Bibliotheken zu den ausgewählten Programmiersprachen betrachtet. Dazu werden solche Bibliotheken in die Bewertung genommen, welche dem Anwender die Arbeit mit großen Datenbeständen und mit Datenbeständen aus der Logistik erleichtern (vgl. Abschnitt 4.2). Zur Arbeit mit großen Datenbeständen werden dabei Bibliotheken zur Datenverarbeitung (vgl. Abschnitt 2.2), wie auch Bibliotheken zur Analyse großer Datenbestände und Generierung von Wissen (vgl. Abschnitt 2.1) und Bibliotheken zur Arbeit mit *verteilten Systemen* (vgl. Abschnitt 2.4 und Abschnitt 3.1) oder Netzwerken gezählt. Zu Bibliotheken zur Analyse großer Datenbestände werden Bibliotheken mit Fokus auf statistischen und mathematischen Funktionen betrachtet. Aus der in Abschnitt 3.2 angestellten Betrachtung geht hervor, dass für jede der ausgewählten Programmiersprachen die zuvor zur Erfüllung dieses Kriteriums nötigen externen Bibliotheken verfügbar sind (vgl. Anhang B). Zwar werden die verfügbaren externen Bibliotheken der einzelnen Programmiersprachen nicht genauer betrachtet, jedoch geht aus der Unterschung hervor, dass für jede der Programmiersprachen diverse Bibliotheken zu den jeweiligen Themengebieten verfügbar sind, sogar dann, wenn die Programmiersprache selbst bereits integrierte Bibliotheken zu den selben Themengebieten besitzt. Daher ergibt sich für die hier angestellte Einordnung der Programmiersprachen anhand des Kriterium 11, dass jede der Sprachen dieses Kriterieum erfüllt, womit jeweils ein DoF von 100% festgelegt wird.

Kriterium 12: Dokumentation

Kriterium 12 verlangt von den betrachteten Programmiersprachen, dass für sie eine Dokumentation existieren soll. Diese Dokumentationen sollen umfassend und nachvollziehbar verfasst sein (vgl. Abschnitt 4.2). Bei der Untersuchung der unterschiedlichen Programmierspachen fällt zwar auf, dass manche Sprachen von den Herstellern geführte Dokumentationen besitzen, für jede der Programmiersprachen jedoch mindestens eine *Open-Source*- oder Community-generierte Dokumentation existiert. Die Vor- und Nachteile von proprietären und *Open-Source*-Dokumentationen werden jedoch an dieser Stelle nicht weiter evaluiert,

weshalb die Existenz diverser Arten von Dokumentationen für jede der in Abschnitt 3.2 ausgewählten Programmiersprachen dazu führen, dass Kriterium 12 von jeder der Sprachen erfüllt wird. Deshalb ergibt sich für jede der Programmiersprachen ein DoF von 100%.

Kriterium 13: Anleitung

Zur Einordnung der unterschiedlichen Programmiersprachen werden die Platzierungen der Programmiersprachen der, in Abschnitt 3.2 verwendeten, Statistik von RedMonk (2019) und der „PYPL PopularitY"-Statistik nach Carbonnelle (2019). Diese Betrachtung liefert einen verwendbaren Indikator für die Erfüllung von Kriterium 13, da RedMonk auf den internen Rankings von GitHub und StackOverflow basiert und die „PYPL PopularitY"-Statistik aus Suchanfragen bei Google' Suchmaschine nach Programmiersprachen-Tutorials gebildet wird. Zur Bestimmung des DoF werden die Platzierungen der Programmiersprachen im ersten Schritt jeweils normiert und anschließend wird im zweiten Schritt das arithmetische Mittel über die normierten Platzierungen gebildet. Dabei erfolgt die Normierung mittels $x[i] = \frac{x[i]-min(x)}{max(x)-min(x)} * 100\%$, wobei x die Menge der Platzierungen der Programmiersprachen in den jeweiligen Statistiken und $x[i]$ die jeweilige Platzierung der einzelnen Programmiersprachen in der Statistik ist. Die Platzierungen, die errechneten Normierungen und der jeweilige DoF werden in Tabelle 4.6 dargestellt.

Tabelle 4.6: Normierung für die Platzierungen der Programmiersprachen aus den Statistiken von Redmonk und PYPL für Kriterium 13

Sprache	RedMonk Index	RedMonk Norm.	PYPL Index	PYPL Norm.	DoF
C	9	0,76	6	0,76	76%
C++	6	0,85	6	0,76	81%
C#	5	0,88	4	0,86	87%
Go	15	0,58	15	0,33	45%
Java	2	0,97	2	0,95	96%
JavaScript	1	1,00	3	0,90	95%
Julia	34	0,00	22	0,00	0%
Perl	18	0,48	19	0,14	31%
PHP	4	0,91	5	0,81	86%
Python	3	0,94	1	1,00	97%
R	16	0,55	7	0,71	63%
Ruby	8	0,79	13	0,43	61%
Swift	11	0,70	9	0,62	66%

Übersicht und Zusammenfassung

Nachdem alle Kriterien betrachtet und jede der ausgewählten Programmiersprachen in die Kriterien eingeordnet worden sind werden abschließend zu dieser Einordnung sämtliche ermittelten Werte für die DoFs der Programmiersprachen für die einzelnen Kriterien in Tabelle 4.7 zusammengefasst.

Tabelle 4.7: Übersicht über die Erfüllung der Kriterien

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Kriterium 1	50	50	100	100	100	100	100	100	100	100	100	50	100
Kriterium 2	50	50	100	100	100	50	50	50	100	100	100	100	100
Kriterium 3	100	100	100	100	100	100	100	100	100	100	100	100	100
Kriterium 4	70	70	80	70	70	80	100	80	80	80	80	100	80
Kriterium 5	100	100	99	98	99	96	100	38	60	87	0	60	98
Kriterium 6	50	50	50	50	50	100	50	100	100	50	100	100	50
Kriterium 7	100	100	100	100	100	100	100	100	100	100	100	100	100
Kriterium 8	100	100	0	100	100	100	100	100	50	100	100	100	100
Kriterium 9	60	60	100	80	80	80	100	80	100	100	80	80	60
Kriterium 10	100	100	100	100	100	100	100	100	100	100	100	100	100
Kriterium 11	100	100	100	100	100	100	100	100	100	100	100	100	100
Kriterium 12	100	100	100	100	100	100	100	100	100	100	100	100	100
Kriterium 13	76	81	87	45	96	95	0	31	86	97	63	61	66

Legende:
1: C 2: C++ 3: C# 4: Go 5: Java 6: JavaScript 7: Julia 8: Perl 9: PHP
10: Python 11: R 12: Ruby 13: Swift

Aufgrund der in Abschnitt 4.2 getroffenen Einteilung der Kriterien in die Kategorien „must-have“, „should-have“ und „nice-to-have“ werden die in Tabelle 4.7 aufgelisteten Kriterien und die dazugehörigen DoFs gewichtet. Für diese Gewichtung werden die Kriterien der Kategorie „must-have“ mit dem Faktor 4x versehen, da diese Kategorie notwendige Kriterien zur Arbeit mit großen Datenbeständen enthält. Eine Sprache, welche diese notwendigen Kriterien nicht erfüllt wird dementsprechend in der hier angestellen Betrachung herabgesetzt. Des weiteren wird die Kategorie der „should-have“-Kriterien mit dem Faktor 2x versehen, um sie von den Kriterien der Kategorie „nice-to-have“ hervorzuheben. Die Kriterien der Kategorie „should-have“ erleichtern dem Anwender die Entwicklung von Programmen zur Arbeit mit großen Datenbeständen, wohingegen die Kategorie „nice-to-have“ nur solche Kriterien enthält, die weder notwendig noch abhängig von der Arbeit mit großen Datenbeständen sind. Daher ist der Faktor für Kriterien der Kategorie „nice-to-have“ 1x. Anschließend werden die gewichteten DoFs aufsummiert und bilden so den Kennwert für die entsprechende Programmiersprache. Tabelle 4.8 zeigt schließlich die Eignung der Programmiersprachen für die Arbeit mit großen Datenbeständen in Prozent.

Die Abgrenzung der verschiedenen ausgewählten Programmiersprachen zeigt auf der einen Seite, dass keine der Programmiersprachen ungeeignet für die Arbeit mit großen Datenbeständen und Datenbeständen in der Logistik ist. Auf der anderen Seite ist zu erkennen, dass

Tabelle 4.8: Eignung der vorgestellten Programmiersprachen

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Eignung	83%	83%	92%	90%	93%	94%	91%	87%	93%	95%	88%	88%	91%

Legende:
1: C 2: C++ 3: C# 4: Go 5: Java 6: JavaScript 7: Julia 8: Perl 9: PHP
10: Python 11: R 12: Ruby 13: Swift

sich alle betrachteten Programmiersprachen im oberen Bereich der errechneten Eignung bewegen. Daraus wird deutlich, dass entlang der entwickelten Kriterien aus Abschnitt 4.2 prinzipiell jede der Programmiersprachen für die Entwicklung von Programmen zur Verarbeitung und Analyse von großen Datenbeständen anwendbar sind. Als Programmiersprachen mit den höchsten errechneten Eignungsgraden ergeben sich nach Python unter anderem Java und JavaScript. Dies bestätigt die zuvor in Abschnitt 3.2 festgestellte Beobachtung, dass Java und Python zu den am häufigsten verwendeten Programmiersprachen in den betrachteten exemplarischen Tools zur Arbeit mit großen Datenbeständen zählen. Auffällig ist jedoch, dass die Programmiersprache C++ in Tabelle 3.3 zwar den zweite Platz nach Java belegt, in der hier anhand der erarbeiteten Kriterien angestellten Abgrenzung der Programmiersprachen voneinander, eine der schlechtesten Platzierungen erreicht. Dies ist durch die durchgeführten Untersuchungen unter anderem auf die Abwesenheit von, in den Standard der Programmiersprache integrierten, Bibliotheken zur Arbeit mit Datenbanken zurückzuführen. Durch die allgemeine Verbreitung von frei zugänglichen externen Bibliotheken für fast alle Programmiersprachen, auch zur Arbeit mit Datenbanken für C++ erklärt sich allerdings, weshalb C++ in der für die Herleitung von Tabelle 3.3 angestellten Untersuchung gut abschneidet. Weiterhin überrascht, dass PHP trotz der Anwendungsdomäne zur serverseitigen Programmierung von Webseiten, in der angestellten Betrachtung eine gute Platzierung erhält. Dies ist wohl auf die Anwenderfreundlichkeit bei der Entwicklung, da PHP den Anwender durch diverse integrierte Bibliotheken zu allen betrachteten, für die Arbeit mit großen Datenbeständen relevanten, Themengebieten unterstützt. Eine vollständige und übersichtliche Darstellung der Einordnung aller Programmiersprachen findet sich in Tabelle C.1.
Anschließend wird die Einordnung der Programmiersprache Julia ausführlich betrachtet.

4.4 Bewertung von Julia anhand der durchgeführten Abgrenzung

In der zuvor in Abschnitt 4.3 angestellten Abgrenzung der ausgewählten Programmiersprachen voneinander belegt Julia einen der mittleren Plätze mit einem Eignungsgrad von 91%. Dieser Grad errechnet sich aus den jeweiligen DoFs von Julia in den einzelnen betrachteten Kriterien.
In der Kategorie der notwendigen „must-have“ Kriterien erreicht Julia durchgehend 100%. Im Einzelnen stammen diese Werte besipielsweise für Kriterium 1, das Kriterium, in dem die verschiedenen Zugriffsmöglichkeiten auf Daten betrachtet werden, daher, dass Julia standardmäßig bereits mit unterschiedlichen Möglichkeiten zur Verwendung von Datenbanken und Dateien ausgestattet ist. Dazu gehören Bibliotheken wie „ODBC“- oder „MySQL“-Treiber sowie Funktionälitaten zur Verwendung von Datenströmen, die es ermöglichen kontinuierliche Flüsse von Datensätzen zu verarbeiten, ohne im Vorraus das Volumen der einzulesenden Datenmenge zu kennen. Damit bilden Datenströme eine grundlegende

Voraussetzung für die Arbeit mit großen Datenmengen, gleich ob diese von Datenbanksystemen bereitgestellt werden oder in Form von Dateien vorliegen. Kriterium 3 verlangt von Julia Daten transformieren zu können. Abschnitt 4.3 erläutert dabei, dass Julia dieses Kriterium zu 100% erfüllt, da sie als Hochsprache über alle nötigen grundlegenden Funktionen um Daten beispielsweise Zerlegen, Gruppieren oder Vergleichen zu können. Auch Kriterium 4 gehört zur Kategorie der notwendigen Kriterien und wird von Julia zu 100% erfüllt. Kriterium 4 verwendet die Klassifizierung von Typensystemen von Programmiersprachen zur Einordnung der Programmiersprachen und da Julia eine starke Typisierung, ein dynamisches Typensystem und eine implizite Typendeklaration besitzt wird dieses Kriterium erfüllt. Diese Kombination an Ausprägungen der Klassifizierung von Typensystemen von Programmiersprachen wird als optimal für die Arbeit von Programmiersprachen mit großen Datenbeständen angenommen, da große Datenbestände vielfältige Datenformate besitzen und ständigen Veränderungen unterliegen (vgl. Abschnitt 2.3). Daher soll ein Typensystem möglichst *dynamische* Typisierung verwenden. Weil große Datenbestände jedoch ebenfalls sehr groß sind (vgl. Abschnitt 2.3 und Abschnitt 2.4), wird eine *starke* Typisierung der Programmiersprachen bevorzugt, da dies die Geschwindigkeit der Programmiersprachen positiv beeinflusst. Letztendlich werden in der Betrachtung von Kriterium 4 noch *implizite* Typensysteme bevorzugt, da dies dem Anwender die Entwicklung von Programmen erleichtert, da er nicht im Voraus wissen muss welche Datentypen für die Variablen verwendet werden müssen. Außerdem verwendet Julia zur Generierung der impliziten Typen die Typinferenz, wodurch die Ausführungs- und die Anwendungsgeschwindigkeit von Julia ebenfalls positiv beeinflusst werden. Weiterhin gehört Kriterium 7, das Kriterium zur Parallelisierbarkeit von Programmen, zu den notwendigen Kriterien. Auch dieses Kriterium erfüllt Julia zu 100% mit ihren integrierten *coroutines* und *Channels*, die es dem Anwender ermöglichen parallele Anwendungen mit „Multi-Threading“ sowie Anwendungen für „Muli-Kern-Systeme“ oder *verteilte Systeme* zu entwickeln. Abschließendes notwendiges Kriterium ist Kriterium 12, das die existenz von Dokumentation zur Programmiersprache zum Gegenstand hat. Da zu Julia, neben der offiziellen Dokumentation, noch diverse andere Quellen, wie zum Beispiel die Werke von Balbaert (2015) und Sherrington (2015), exisistieren, erfüllt Julia auch dieses Kriterium zu 100%. So erklärt sich das gute Abschneiden von Julia in der Kategorie der „must-have“-Kriterien.

In der Kategorie der „should-have“ Kriterien erreicht Julia einen durchschnittlichen Grad der Erfüllung von 75%, wobei dieser Wert lediglich daraus resultiert, dass Julia den letzten Platz bei der Verfügbarkeit von Anleitungen und Beispielen inne hat. Im Kriterium 5, welches die effiziente Ausführung der Programmiersprache behandelt, erreicht Julia einen DoF von 100%, da Julia in den Benchmarks aus der Literatur zu den Sprachen mit den höchsten Geschwindigkeiten zählt. Ebenso erreicht Julia in Kriterium 9 und in Kriterium 11 die 100%. In beiden Kriterien wird die Existenz von Bibliotheken für die Programmiersprache betrachtet. Zur Erfüllung von Kriterium 9, in dem es um die internen Bibliotheken geht, trägt Julias umfangreiche Sammlung von unterschiedlichen Bibliotheken zur Arbeit mit großen Datenbeständen bei (vgl. Abschnitt 3.3), da Julia für die Verwendung bei eben solchen Herausforderungen, wie der Verarbeitung großer Datenmengen, entwickelt worden ist. Die Zahl der, für Julia existierenden, externen verfügbaren Bibliotheken ist, wie die Recherche ergeben hat, nicht so hoch wie die Anzahl verfügbarer Bibliotheken anderer Sprachen zum Thema große Datenbestände, Anaylse von Daten oder die Verwendung auf *verteilen Systemen*. Jedoch zeigt Abschnitt 3.3, dass Julia eine relativ junge Programmiersprache ist, die beispielsweise durch den erhaltenen *Sidney Fernbach Award* an Popularität gewonnen hat und seither an Verbreitung und Bekanntheitsgrad zunimmt. Diese erhöhte Aufmerksamkeit bezüglich Julia wird zwangsläufig zu einer vermehrten Entwicklung von externen Bibliotheken führen, umso mehr Anwender Julia für ihre Programme zur

Verarbeitung großer Datenmengen nutzen. Zudem besitzt Julia bereits standardmäßig einige Funktionalitäten zu diesen für die Arbeit mit großen Datenbeständen verwendbaren Themengebieten. Dennoch ist die Anzahl der verfügbaren externen Bibliotheken zu Julia ausreichend, um die für die Erfüllung von Kriterium 11 geforderten Arten von externen Bibliotheken zu enthalten. Letztlich fällt Kriterium 13 in die Kategorie der „should-have“-Kriterien. Dieses Kriterium erfüllt Julia nicht (0%). Dieser Wert resultiert allerdings nicht aus der Tatsache, dass es keine Anleitungen, Tutorials oder Beispiele für Julia vorhanden sind, sondern aus der Art des Vorgehens zur Berechnung der relativen DoFs der einzelnen Programmiersprachen für dieses Kriterium. Zur Berechnung dieses Wertes wurden die Statistik von RedMonk (2019) und die „PYPL PopularitY“-Statistik nach Carbonnelle (2019) verwendet, weil diese einen verwendbaren Indikator für die Erfüllung von Kriterium 13 liefern, da RedMonk auf den internen Rankings von GitHub und StackOverflow basiert und die „PYPL PopularitY“-Statistik aus Suchanfragen bei Google' Suchmaschine nach Programmiersprachen-Tutorials gebildet wird. Da Julia in beiden Rankings den letzten Platz belegt, errechnet sich Julias DoF für dieses Kriterium zu 0%. Jedoch lässt sich daraus lediglich folgern, dass es deutlich weniger Ergebnisse zu Julia auf den jeweiligen Plattformen gibt. Dies lässt sich jedoch aufgrund des geringen Alters der Programmiersprache Julia, des hohen Alters und der, relativ zu Julia hohen, Verbreitungsgrade der anderen ausgewählten Programmiersprachen relativieren.
In der letzten Kategorie, der Kategorie der „nice-to-have“ Kriterien, erreicht Julia ebenfalls einen durchschnittlichen Grad der Erfüllung von 75%. Dieser durchschnittliche Grad der Efüllung ergibt sich unter anderem daraus, dass Julia Kriterium 2, die Forderung nach Grafischer Aufbereitung von Daten nur zu 50% erfüllen kann. Für Julia existieren in den internen Bibliotheken zwar Funktionalitäten zur Anbindung an verbreitete externe GUI-Bibliotheken, jedoch, zum Stand der aktuellen Recherche, keine vollständigen Bibliotheken zu diesem Thema. Auch in der Erfüllung von Kriterium 6 erreicht Julia nur einen Grad von 50%, da die Art der Auswertung der Geschwindigkeit bei der Anwendung der unterschiedlichen Programmiersprachen auf dem Konzept der schnelleren Übersetzung von Programmen in der entsprechenden Sprache basiert. Wie in Abschnitt 4.3 zu diesem Kriterium erläutert sind Compiler in dieser spezeillen Betrachtung langsamer als Interpreter. Betrachtet man allerdings die Gruppe der Compiler im einzelnen, so wird Julia voraussichtlich im oberen Bereich der Programmiersprachen enden, die einen Compiler zur Übersetzung von Quelltext in Maschinencode verwenden, da Julia einen JIT-Compiler verwendet. Doch dieser Umstand wird in der angestellten Abgrenzung nicht betrachetet.
Abschließend besitzt Julia in Kriterium 8 und Kriterium 10 einen DoF von 100%, da Julia unter einer OSS-Lizenz verfügbar ist und die Erstellung von eigenen Bibliotheken für den Anwender ermöglicht.
Alles in allem führt die Betrachtung von Julia und Julias Abschneiden in den einzelnen erarbeiteten Kriterien zu dem Ergebnis, dass Julia für die Arbeit mit großen Datenbeständen und Datenbeständen in der Logistik verwendbar ist. Alle grundlegenden Eigenschaften, die eine Programmiersprache für diese Verwendung benötigt, besitzt Julia. Die angestellten Untersuchungen zeigen jedoch, dass Julias Potential noch nicht ausgeschöpft ist. In puncto Geschwindigkeit legt Julia gute Ergebnisse vor und sollte die Verbreitung von Julia weiter zunehmen, sollten auch Aspekte, wie die Verfügbarkeit von Anleitungen, Tutorials und Beispielen, besser bewertet werden.

5 Evaluierung der Eignung von Julia auf große Datenbestände in der Logistik

Kapitel 5 beschäftigt sich mit der Anwendung der Programmiersprache Julia auf einen exemplarischen Datenbestand. Dazu wird in Abschnitt 5.1 zunächst ein exemplarierscher Datenbestand vorgestellt. In Abschnitt 5.2 werden dann einige der zuvor verwendeten Definitionen und Eigenschaften von Programmiersprachen aufgegriffen und in Julia mit konkreten Programmbeispielen umgesetzt.

5.1 Vorstellung eines exemplarischen Datenbestands

Zur Darstellung von Julias Funktionalitäten wird im Folgenden ein exemplarischer Datenbestand vorgestellt, auf dem anschließend einige grundlegende Operationen mit Hilfe von Julia durchgeführt werden. Bei diesem Datenbestand handelt es sich um reale Unternehmensdaten, die in diesem Kontext anonymisiert betrachtet werden. Im Folgenden werden daher Eigennamen von Tabellen und die Einträge in der Datenbank verfremdet dargestellt. Der exemplarische Datenbestand liegt in Form einer MySQL-Datenbank vor und enthält 15 Tabellen. Insgesamt besteht der, in der Datenbank vorhandene, Datenbestand aus 12 623 262 Datensätzen mit einem Gesamtvolumen von 1,2 GiB ($1,2 * 2^{30}$ Byte = $1,29\text{GB} = 1,29 * 10^9$ Byte). Die Datensätze selbst bestehen größtenteils aus 1-9-stelligen Zahlen, kurzen Texten, in Form von Bezeichnungen und IDs, sowie Zeitstempeln. Eine Übersicht über die Verteilung der Daten über die verschiedenen Tabellen wird in Tabelle 5.1 dargestellt.

5.2 Gundlegende Programmierung mit Julia

Zur Einführung in die Programmierung mit Julia werden im folgenden Konstrukte und Funktionalitäten von Programmiersprachen aufgegriffen. Dazu werden die Definitionen aus Abschnitt 3.1 und die Funktionsweise von Julia (vgl. Abschnitt 3.3) verwendet.

Variablen

Die Betrachtung von Julia wird mit Definition 3.9, der Definition von *Variablen* aus Abschnitt 3.1, begonnen. *Variablen* können in Julia sehr einfach und unabhängig vom Datentyp der in den Variablen gespeicherten Daten erzeugt werden, da Julia den Datentyp mittels Typinferenz zur Laufzeit des Programms ermittelt (vgl. Abschnitt 4.4). Daher kann in Julia eine Variable durch die direkte Zuweisung eines Wertes initialisiert werden und darauffolgend mit Werten beliebigen Typs belegt werden. Beispielsweise kann eine Variable, hier a, mit einem Wert vom Datentyp 64bit-Ganzzahl initialisiert werden und danach Werte mit Gleitkommazahl- oder Texttypen, sowie Datenstrukturen wie Arrays oder Matrizen

Tabelle 5.1: Struktur des exemplarischen Datenbestands

Tabelle	Volumen	Anzahl Datensätze	Anzahl Attribute
Tabelle 1	16 KiB	0	38
Tabelle 2	854,6 KiB	18 751	13
Tabelle 3	854,6 KiB	18 751	13
Tabelle 4	231,1 MiB	2 318 501	9
Tabelle 5	233,6 MiB	2 318 501	9
Tabelle 6	254,0 MiB	2 318 501	10
Tabelle 7	1,5 MiB	8 714	15
Tabelle 8	1,3 MiB	8 714	15
Tabelle 9	114,7 MiB	296 336	34
Tabelle 10	54,4 MiB	303 908	23
Tabelle 11	54,4 MiB	303 908	23
Tabelle 12	4,9 KiB	180	7
Tabelle 13	67,7 MiB	1 569 499	22
Tabelle 14	67,7 MiB	1 569 499	22
Tabelle 15	176,2 MiB	1 569 499	34

zugewiesen bekommen. Herauszuheben für die Strukturen der Arrays, im Kontext der Programmierung von Julia, ist, dass Arrays nicht auf einen Datentypen beschränkt sind, wie in vielen anderen Programmiersprachen. Wird ein Array mit dem Schlüsselwort *Any* initialisiert, so können die Datentypen in dem Array beliebig gemischt werden. Abschließend ist noch zu erwähnen, dass das Schlüsselwort *const* verwendet werden kann, um eine Variable für die Laufzeit des Programms unveränderbar zu machen. Dies ist immer dann sinnvoll, wenn es sich tatsächlich um Konstanten handeln soll. Eine Veranschaulichung für die Erzeugung von Variablen in Julia wird in Algorithmus 5.1 dargestellt.

Algorithmus 5.1: Variablen in Julia

```
1  a = 7                                  # Int64
2
3  a = 1.587                              # Float64
4
5  a = "hallo welt"                       # String
6
7  a = [20, 55, 13, 242, 450, 5, 33]      # Array{::Int64,1}
8
9  a = Any[20, 55, "Text"]                # Array{::Any,1}
10
11 a = [20, 55, 13; 450, 5, 33]           # 3x3 Matrix
12
13 const b = "3000"                       # const String
```

Funktionen

Weiterhin wird die Erstellung von Funktionen, wie in Definition 3.4 festgelegt, in Julia genauer betrachtet. Algorithmus 5.2 zeigt hierzu drei verschiedene grundlegende Arten der Defintion von Methoden in Julia. Zunächst wird in Zeile 1 eine eine sogenannte anonyme

Algorithmus 5.2: Methoden in Julia

```
1 f(x::Float64, y::Float64) = x + y
2
3 function fkt1(x, y=1)
4     return x * y
5 end
6
7 function fkt2(x, y)
8     return x + y, x * y
9 end
```

Funktion, auch Lambda-Funktion genannt (vgl. Abschnitt 3.3), definiert. Diese Art von Funktionen zeichnet aus, dass sie keinen Funktionsnamen besitzt (vgl. Abschnitt 3.3). Die Funktion in Zeile 1 besitzt zwei Übergabeparameter x und y, die beide einen festen Datentyp (*Float64*) zugewiesen bekommen haben und nichts anderes tut, als die beiden übergebenen Werte zu addieren und automatisch zurückzugeben. Damit können von dieser Funktion auch nur Daten mit dem entsprechenden Datentyp verarbeitet werden.
Funktion *fkt1* aus Zeile $3-5$ in Algorithmus 5.2 besitzt genau wie die Funktion aus Zeile 1 zwei Übergabeparameter, ist jedoch keine anonyme Funktion, da sie einen Namen (*fkt1*) besitzt. Die Definition der Funktion aus den Zeilen $3-5$ beginnt mit dem Schlüsselwort *function*, endet mit *end* und kann theoretisch beliebige Datentypen verarbeiten, da für die Übergabeparameter kein konkreter Datentyp festgelegt worden ist. Jedoch wird die Funktion einen Fehler werfen, sobald eine der Variablen einen Datentyp besitzt, auf dem der verwendete „$*$“-Operator nicht definiert ist, wie beispielsweise für Arrays. Eine weitere Besonderheit von *fkt1* ist der definierte Standardparameter der Funktion für $y = 1$. Dieser Standardparamter erlaubt es, die Funktion mit nur einem Paramter aufzurufen. Der zweite Parameter wird in diesem Fall standardmäßig auf 1 gesetzt. So ergäbe ein Aufruf $n = fkt1(7)$ beispielsweise $n = 7$ und $n = fkt(\text{"}hello\text{"})$ ergäbe $n = \text{"}hello1\text{"}$, da „$*$“ in Julia als Operator für die Konkatenation von Strings definiert ist.
Für Funktion *fkt2* aus Zeile $7-9$, gilt ebenso, dass Werte mit beliebigen Datentypen übergeben werden können. Die Besonderheit an dieser Funktion ist, dass sie mehrere Operationen auf den Übergabeparametern ausführt und diese anschließend beide in Form einer Datenstruktur ausgibt. So ergäbe zum Beispiel ein Funktionsaufruf mit $k = fkt2(7, 3)$ Werte von $k = (10, 21)$. Auch für diese Funktion gilt, dass ein Fehler geworfen würde, sobald einer der Operatoren auf den entsprechenden Datentyp der übergebenen Werte nicht anwendbar ist.

Casting

Definition 3.10, aus Abschnitt 3.1 erklärt den Begriff des Casting. Dies ist ein elementarer Bestandteil von Julias Arbeitsweise, wenn der Anwender keine Typen für Variablen definiert und auch in der zuvor genannten Typinferenz findet es Anwendung. Beim Casting wird der Datentyp eines Elementes in einen anderen Datentyp umgewandelt, sofern diese Umwandlung möglich ist. Julia verwendet das Casting dabei durchgehend. Algorithmus 5.3 zeigt

einige einfache Varianten des Castings auf, um die dem Anwender zur Verfügung stehenden Varianten des Castings zu erläutern. Zunächst wird in Zeile 1 eine Variable *a* mit einem

Algorithmus 5.3: Casting in Julia

```
1 a = 12                              # Int64
2 b = convert(Float64, a)             # b::Float64
3
4 c = promote(3, 1.5, 5.9, 7)         # Unterschiedliche Typen
5
6 # x::String
7 e1 = parse(Int64, x)                 # e1::Int64
8 e2 = tryparse(Int64, x)              # e2::Int64 oder nothing
```

Ganzzahl-Wert belegt. Diese Variable *a* besitzt dann also den bereits zuvor verwendeten Datentypen *Int64*. In Zeile 2 wird dann dieser Datentyp mittels der *convert*-Funktion in einen Gleitkomma-Datentypen (*Float64*) umgewandelt und in Variable *b* gespeichert. Die Variable *a* behält dabei weiter ihren ursprünglichen Datentypen und die Variable *b* besitzt den selben Wert wie die Variable *a*, jedoch mit dem neuen Datentypen.
Die zweite Variante der Typenumwandlung ist das *promote*. Mit Hilfe dieser Funktion aus Zeile 4 kann eine Menge von Werten unterschiedlicher Datentypen auf eine Menge mit einem gemeinsamen Datentypen umgewandelt werden. Dabei wird für alle Elemente ein Datentyp gesucht, der jedes der Elemente repräsentieren kann. Dieser Vorgang hat jedoch nichts mit Julias interner Hierarchie der Datentypen (vgl. Abbildung 3.2) zu tun. Die von der *promote*-Funktion vorgenommenen Umwandlungen müssen also nicht zwingend entlang der Typenhierarchie geschehen.
Als dritte Variante der Typenumwandlung wird die *parse*-Funktion vorgestellt, die genutzt werden kann um Daten im Textformat in andere Datentypen zu überführen. Bei der Verwendung der einfachen *parse*-Funktion aus Zeile 7 muss der Wert der an die Funktion übergebenen Variable auch zu einem Wert des entsprechenden Datentypens konvertierbar sein. Es muss also beispielsweise ein Text mit dem Inhalt *„5“* an die *parse*-Funktion übergeben werden, sodass diese den übergebenen Text in einen Zahlen-Datentypen konvertieren kann. Ansonsten wird diese Funktion einen Fehler. Für den Fall, dass vor dem Umwandeln des Datentyps nicht bekannt ist wie die umzuwandelnden Daten aussehen, steht dem Anwender die *tryparse*-Funktion, wie in Zeile 8 zur Verfügung. Die *tryparse*-Funktion wirft keinen Fehler, sobald ein Wert nicht in den gewünschten Datentyp überführbar ist, sondern gibt den in Julia definierten *nothing*-Wert zurück. Dieser *nothing*-Wert entspricht am ehesten dem aus anderen Programmiersprachen bekannten „NULL“-Wert.

Kontrollstrukturen

Um mit Julia vollständige Programme entwickeln zu können benötigt der Anwender die Kontrollstrukturen (vgl. Definition 3.11). Diese werden im Folgenden betrachtet, beginnend mit den bedingten Anweisungen.
Bedingte Anweisungen in Julia werden durch die Schlüsselwörter *if-elseif-else-end* dargestellt. Dabei bezeichnet *if* die erste zu überprüfende Bedingung. Der an *if* übergebende Parameter muss ein boolescher Ausdruck sein, also ausschließlich die Werte „Wahr“ und „Falsch“ annehmen können. Sollte der an *if* übergebene Parameter nach seiner Auswertung den Wert „Wahr“ annehmen, so wird der Quelltext unterhalb der *if*-Bedinung erfüllt. Die kürzest mögliche bedingte Anweisung besteht also aus einem *if* und einem dazugehörigen *end*. Sollte es weitere Bedingungen geben, die erfüllt werden können, so gibt

es die Möglichkeit das *if* und beliebig viele *elseif* zu erweitern. Dabei funktionieren die *elseif* analog zur *if*-Anweisung. Abschließend kann die *else*-Anweisung angefügt werden, die ausgeführt wird, sollte keine der zuvor angeführten Bedingungen erfüllt sein. Dabei schließen sich die einzelnen Fälle der bedingten Anweisung jeweils aus und werden der Reihe nach abgearbeitet. Es kann also jeweils immer nur eine der Bedingungen korrekt sein, beziehungsweise wird die bedingte Anweisung beendet, sobald eine Bedingung erfüllt und abgearbeitet worden ist. Algorithmus 5.4 zeigt in den Zeilen $1-7$ ein einfaches Beispiel für eine klassische bedingte Anweisung.
Zeile 9 zeigt, wie die bei den Funktionen eingeführte Möglichkeit zur Erstellung von anonymen Funktionen genutzt wurde, um einen Lamda-Ausdruck für die bedingte Anweisung zu schaffen. Dabei bezeichnet der Ausdruck vor dem ?, in diesem Beispiel das x, die Bedingung. Der Ausdruck hinter dem ? und vor dem :, also das a, den Teil der bedingten Anweisung, der ausgeführt wird, wenn die Bedingung „Wahr“ ist und der Teil hinter dem :, also b, letztlich den Teil der Anweisung, der ausgeführt wird, wenn die Bedingung x den Wert „Falsch“ annimmt.

Algorithmus 5.4: Bedingungen in Julia

```
1 if (x)
2     println("x condition is true")
3 elseif (y)
4     println("x condition is false, y condition is true")
5 else
6     println("x and y conditions are false")
7 end
8
9 x ? a : b
```

Weiterhin kann der Anwender auf unterschiedliche Arten von Schleifen zurückgreifen, also Kontrollstrukturen, welche Anweisungen wiederholen. Die erste betrachtete Schleife ist die *while*-Schleife. Die *while*-Schleife besteht aus den Schlüsselwörtern *while* und *end*, einer Bedingung, die angibt wie häufig der Inhalt der Schleife ausgeführt werden soll, und dem Quelltext, der für jede Iteration der Schleife ausgeführt werden soll. Dabei funktioniert die Überprüfung der Bedinungung ähnlich zur bedingten Anweisung. Solange die Bedingung erfüllt ist, wird der Inhalt der Schleife ausgeführt und am Ende der Ausführung wird die Bedingung erneut geprüft. Algorithmus 5.5 dient dabei als anschauliches Beispiel und zeigt eine *while*-Schleife, welche die Zahlen $0-10$ ausgibt.

Algorithmus 5.5: While-Schleife in Julia

```
1 i = 0
2 while i <= 10
3     println(i)
4     i += 1
5 end
```

Die zweite betrachtete Art von Schleifen ist die *for*-Schleife. Die *for*-Schleife besteht ähnlich wie die *while*-Schleife aus zwei Schlüsselwörtern *for* und *end*, einer Bedingung, bestehend aus einer Zählvariable und einer Abbruchbedinung sowie dem zu wiederholenden Teil innerhalb der Schleife. Dabei zeichnet die *for*-Schleife aus, dass sie, anders als die *while*-Schleife, auf

eine Zählvariable in der Bedingung angewiesen ist. Eine *for*-Schleife zählt also von einem Startwert bis zu einem Zielwert. Algorithmus 5.6 zeigt dazu drei Beispiele.

Algorithmus 5.6: For-Schleifen in Julia

```
1  for i = 2:6
2     println(i)
3  end
4  # 2 3 4 5 6
5
6  for i = 1:2:10
7      println(i)
8  end
9  # 1 3 5 7 9
10
11 for i in ["eins","zwei","drei"]
12     println(i)
13 end
14 # eins zwei drei
```

Im ersten Beispiel von Zeile $1-3$ zählt die *for*-Schleife vom Startwert 2, angegeben vor dem :, bis zum Zielwert 6, hinter dem :. Dabei speichert die Variable i zu jedem Zeitpunkt den aktuellen Iterationsfortschritt der Schleife. Über diese Variable kann der Iterationszähler zu jedem Zeitpunkt in der Schleife aufgerufen und verwendet werden. Die erste *for*-Schleife aus Algorithmus 5.6 zählt also von 2 bis 6 und gibt die jeweiligen Zahlen aus.
Das zweite Beispiel in Zeile $6-8$ ist analog zum ersten Beispiel, mit einer Ergänzung. In diesem Fall ist die Bedingung der *for*-Schleife um einen weiteren Wert ergänzt worden. Hier teilt sich die Bedinung nicht nur in Startwert und Zielwert, sondern in Startwert, Schrittweite und Zielwert. Durch die Möglichkeit die Schrittweite für *for*-Schleifen angeben zu können, bietet dem Anwender nicht nur neue Möglichkeiten, sonder zeigt ebenfalls, dass die *for*-Schleife standardmäßig einen, wie bei den Funktionen erläuterten, Standardparameter für die Schrittweite verwenden.
Abschließend zeigt die *for*-Schleife in Zeile $11-12$ von Algorithmus 5.6 eine Variante der *for*-Schleife, in der die Bedingung der Schleife dynamisch aus der übergebenen Datenstruktur generiert wird. Diese Form der *for*-Schleife beginnt beim ersten Element der Datenstruktur und wiederholt den Inhalt der Schleife solange bis jedes Element der Datenstruktur durchlaufen worden ist.

Datenverarbeitung mit Julia

Um die Funktionalitäten von Julia auf dem exemplarischen Datenbestand zu demonstrieren, wird zu diesem Zweck im ersten Schritt eine Verbindung zu der Datenbank mit den exemplarischen Daten hergestellt. Wie in Abschnitt 5.1 erläutert, liegt der hier betrachtete Datenbestand in Form einer „MySQL-Datenbank“ vor. Zur Verbindung mit dieser Datenbank wird Julias integrierte *MySQL*-Bibliothek verwendet. Anschließend wird mittels des *connect*-Befehls und den Verbindungsdaten eine Verbindung zur Datenbank aufgebaut und überprüpft, ob diese Verbindung etabliert werden konnte. Abschließend wird die existierende Verbindung zur Datenbank wieder geschlossen. Die Umsetzung dieses Verbindungsaufbaus wird in Algorithmus 5.7 dargestellt.

Nachdem eine Verbindung hergestellt worden ist, können mittels dieser in *conn* gespeicherten Verbindung Anfragen an das Datenbanksystem gesendet werden. Diese Anfragen sind hier

Algorithmus 5.7: Datenbankverbindung mit Julia

```
using MySQL

const host = <hostname>
const user = <username>
const pw = <password>
const dbn = <databasename>

conn = MySQL.connect(host, user, pw, db = dbn)

if(conn != 0)
    println("Verbindung erfolgreich.")
else
    throw(ErrorException("Verbindung nicht hergestellt."))
end

MySQL.disconnect(conn)
```

und im Folgenden in der Sprache SQL verfasst. Die Ergebnisse der Anfragen werden mit Hilfe des |> -Operators, der eine Funktion auf die vorhergehenden Argumente anwendet, in einem *DataFrame* in der Variable *res* gespeichert. Anschließend kann auf jedes Attribut, also jede Spalte des Dataframes, über die jeweilige Bezeichnung des Attributes zugegriffen werden. Im Folgenden werden dann einige beispielhafte statistische Standardfunktionen von Julia auf diesen Daten ausgeführt. Dazu gehören unter anderem Funktionen wie das Bilden des Mittelwertes oder die Bestimmung von Minimum, Maximum oder Median, aber auch das Berechnen der Standardabweichung oder verschiedener Quantile. Dargestellt wird die Programmierung dieses Vorgehens in Algorithmus 5.8.

Algorithmus 5.8: Statistische Datenauswertung mit Julia

```
using DataFrames
using Statistics

res =  MySQL.Query(conn, """SELECT * FROM (*<TableName>*);""") |> DataFrame
arr = foo.<AttributeName>

m = mean(arr, dims=1)
maxVal = maximum(arr)
medVal = median(arr)
minVal = minimum(arr)
stddev = std(arr)
quant = quantile(arr, [0.01,0.05,0.1,0.25,0.5])

print("mittelwert: "*m*", min: "*minVal*", median: "*medVal*", ")
print("max: "*maxVal*", standard abweichung: "*stddev*", ")
print("quantile: "*quant*"")
```

Auf dem selben Wege wie die Daten in diesem Fall aus der Datenbank extrahiert worden sind, können Daten mittels einfacher *SQL*-Befehle wieder in die Datenbank geladen werden. Ergänzend dazu existieren noch verschiedene weitere Bibliotheken zur Arbeit mit *SQL*- sowie anderen Datenbanken, auf die an dieser Stelle jedoch nicht weiter eingegangen wird. Eine exemplarische Anfrage auf die in Tabelle 5.1 angegebene Tabelle 6 liefert einen *DataFrame*, der alle 2 318 501 Einträge aus dieser Tabelle enthält. Auf dem dritten At-

tribut dieser Datenstruktur können dann beispielsweise die, in Zeile $7 - 12$ aufgeführten, statistischen Funktionen ausgeführt werden, woraufhin die Ausgabe in Zeile $14 - 16$ lautet: *mean: [350.5673553731484], min: 0, median: 47.0, max: 960000, standard deviation: 3642.1588810737217, quantile: [0.0, 6.0, 10.0, 20.0, 47.0]*. Ebenso bieten Julias Standardbibliotheken eine Vielzahl weiterer statistischer und mathematischer Funktionen auf die an dieser Stelle nicht näher eingegangen wird.
Die *DataFrame*-Datenstruktur erlaubt es dem Anwender bei der Arbeit mit Julia Daten in tabellarischer Form zu manipulieren, zu filtern sowie zu sortieren. So können unterschiedliche Tabellen mittels einfachen *SQL*-Befehlen aus der Datenbank extrahiert, in *DataFrames* gespeichtert und anschließend beispielsweise über den einen *join*-Befehl verknüpft werden. Dies ergibt für die Tabellen 10 und 13 einen resultierenden *DataFrame* mit insgesamt 44 Attributen und 1 569 431 Einträgen. Dieser neue *DataFrame* kann anschließend gefiltert oder mittels statistischer Funktionen analysiert werden. Die *DataFrames* liefern also umfangreiche Möglichkeiten zur Verarbeitung von extrahierten Daten. Doch die Untersuchung der *DataFrames* wird an dieser Stelle nicht weiter ausgeführt.

Fazit

Die Anwendung von Julia unterscheidet sich nicht viel von der Anwendung anderer Programmiersprachen bei der Erstellung von Programmmen. Die Verwendung von Variablen, Funktionen und Kontrollstrukturen lassen sich genau so in anderen Programmiersprachen wiederfinden. Das Casting wird hauptsächlich von Julia selbst, aufgrund der dynamischen Typisierung, durchgeführt und vereinfacht durch dieses Vorgehen das Entwickeln von Anwendungen, die mit Daten unterschiedlicher Datentypen arbeiten.
Bei den ausgeführten Beispielen legt Julia sowohl bei der Verarbeitung von großen Datenmengen mit den verwendeten *DataFrames* als auch bei den ausgeführten statistischen Methoden eine überzeugende Verarbeitungsgeschwindigkeit an den Tag. Außerdem lässt sich durch die diversen Bibliotheken, die Julia zur Arbeit mit großen Datenbeständen besitzt (vgl. Abschnitt 3.3), feststellen, dass Julia ein umfangreichens Potential zur Verarbeitung von großen Datenmengen besitzt.

6 Zusammenfassung und Ausblick

Große Datenbestände bilden eine der Herausforderungen in der Logistik und das effiziente Gewinnen von Information und Wissen stehen hierbei im Fokus. Zur Bewältigung dieser Herausforderung wurde die Programmiersprache Julia entwickelt, die mit optimaler Performance und Vorgzügen einer GPL aufwartet. Daher war es das Ziel dieser Arbeit, die Programmiersprache Julia genauer zu betrachten und sie auf ihre Eignung für die Arbeit mit großen Datenbeständen hin zu untersuchen.
Die Unterschung der Programmiersprache Julia verlangte zunächst eine Betrachtung des Kontext in den Julia anhand dieser Arbeit eingeordnet werden sollte. Deshalb beschäftigte Kapitel 2 sich mit einer Einführung in große Datenbestände. Dazu wurden die Begrifflichkeiten *Daten*, *Information* und *Wissen* definiert (vgl. Abschnitt 2.1), um diese im weiteren Vorgehen zu nutzen. Weiterhin wurde betrachtet, wie die Verwendbarkeit von Daten sichergestellt werden kann (vgl. Abschnitt 2.2) und welche Eigenschaften große Datenbestände aufweisen, um einzugrenzen welche Datenbestände als große Datenbestände zu klassifizieren sind und welche Besonderheiten solche großen Datenbestände in der Regel auszeichnen. Abschließend zu Kapitel 2 wurde der Kontext der Logistik beschrieben und parallelen zwischen den Herausforderungen der Logistik und der Definition der Eigenschaften großer Datenbestände aufgezeigt.
Als Bestandteil der Untersuchung von Julia sollte diese, als ein Teilziel dieser Arbeit, von anderen Programmiersprachen abgegrenzt werden. Aus diesem Grund betrachtet Kapitel 3 Programmiersprachen für die Datenverarbeitung sowie Verarbeitung großer Datenbestände. Dazu beginnt das Kapitel mit der Untersuchung von Programmiersprachen, ihren Eigenschaften und Möglichkeiten Programmiersprachen zu Kategorisieren (vgl. Abschnitt 3.1). Dieser Abschnitt lieferte eine Grundlage für die folgende Untersuchung der Programmiersprachen. Anschließend wurden verschiedene Statistiken zur Verbreitung von Programmiersprachen betrachtet und es wurden zwölf Programmiersprachen ausgewählt, die später zur Abgrenzung von Julia herangezogen worden sind (vgl. Abschnitt 3.2). Diese Programmierspachen wurden untersucht und anhand ihrer Eigenschaften zusammengefasst. Außerdem wurden verschiedene Tools zur Arbeit mit großen Datenbeständen betrachtet, um einen Eindruck von der Verbreitung der unterschiedlichen Programmiersprachen in exisiterenden Softwarelösungen zu erhalten. Abschließend zu diesem Kapitel wurde die Programmiersprache Julia selbst vorgestellt und ihre wichtigsten Eigenschaften herausgestellt.
Mit der eigentlichen Untersuchung und Abgrenzung der Programmiersprache Julia von den anderen ausgewählten Sprachen beschäftigt sich anschließend Kapitel 4. Hier wurden zunächst Anforderungen an Programmiersprachen für die Verarbeitung großer Datenbestände und für Datenbestände in der Logistik abgeleitet (vgl. Abschnitt 4.1). Diese Anforderungen leiten sich vorallem aus den Eigenschaften großer Datenbestände (vgl. Abschnitt 2.3), aus den Ausführungen zu Datenbeständen in der Logistik (vgl. Abschnitt 2.4) und den unterschiedlichen Definitionen aus dem Umfeld der Programmiersprachen (vgl. Abschnitt 3.1) ab. Anschließend wurden die abgeleiteten Anforderungen genutzt, um Kriterien an die Programmiersprachen zur Datenverarbeitung zu erarbeiten, welche nicht nur zur Erfüllung der Anforderungen sondern auch zur Abgrenzung der Programmiersprachen dienen sollen (vgl. Abschnitt 4.2). Eben diese Abgrenzung der Programmiersprachen, anhand der erarbeiteten

Kriterien, ist Gegenstand von Abschnitt 4.3. Dort wurden die einzelnen, zuvor erarbeiteten Kriterien der Reihe nach betrachtet und die unterschiedlichen Programmiersprachen in die Kriterien eingeordenet. So ergab sich die in Tabelle 4.8 zusammengefasst Eignung der verschiedenen Programmiersprachen für die Verarbeitung großer Datenmengen und die Arbeit mit Datenbeständen aus der Logistik. Kapitel 4 schließt mit der Bewertung von Julia anhand der zuvor durchgeführten Abgrenzung (vgl. Abschnitt 4.4).
Abschließend wurde für Kapitel 5 die Programmiersprache mit Hilfe eines exemplarischen Datenbestands evaluiert. Dazu wurde in Abschnitt 5.1 dieser Datenbestand kurz vorgestellt und in Abschnitt 5.2 aufgezeigt, wie eine praktische Anwendung der Programmiersprache Julia aussehen könnte.

Bei der Untersuchung der Programmiersprache Julia wird deutlich, dass Julia viele Vorteile für die Verarbeitung von großen Datenbeständen mitbringt. Aus der Abgrenzung zu den anderen Programmiersprachen geht hervor, dass Julia alle relevanten Kriterien erfüllt und somit für die Arbeit mit großen Datenbeständen geeignet ist. Trotzdem lässt Julias Platzierung im Mittelfeld der betrachteten Programmiersprachen erkennen, dass das Potential noch nicht ausgeschöpft ist. Dort wo Julia beispielsweise in Sachen Geschwindkeit vorlegt, sind andere Aspekte ausbaufähig. Die stetig wachsende Community, die zunehmende Verbreitung sowie der wachsende Bekanntheitsgrad der Programmiersprache lassen darauf schließen, dass Julias Defizite in der angestellten Betrachtung schnell beseitigt werden könnten. Letzten Endes ist abzuwarten, ob Julia unter den anderen weit verbreiteten und etablierten Programmiersprachen, die sich mit der Verarbeitung großer Datenmengen befassen, wie Python und Java dauerhaft bestehen kann.
Für weiterreichende Untersuchungen zu diesem Thema könnten beispielsweise einige der betrachteten Kriterien, wie das Kriterium 6 zur effizienten Anwendung der Programmiersprachen, erneut aufgegriffen und tiefergehend untersucht werden. Im angeführten Fall könnten zum Beispiel die Funktionsweisen von Compilern genauer untersucht werden, um eine feinere Einordnung anhand der Übersetzungsgeschwindigkeiten zu erzeugen. Außedem könnte theoretisch untersucht werden, welche Tools für die Entwicklung mit den unterschiedlichen Programmiersprachen zur Verfügung stehen, da diese oft bei der Auswahl einer Programmiersprache zur Bewältigung einer bestimmten Aufgabe herangezogen werden.

Literaturverzeichnis

Ackoff, R. L.: From data to wisdom. Journal of applied systems analysis 16 (1989) 1, S. 3–9.

AddThis: Hydra. 2019. URL: https://github.com/addthis/hydra (zuletzt geprüft am 19. 09. 2019).

Anthony, M.; Arjuna., C.: Introduction to HPCC (High-PerformanceComputing Cluster). White Paper. HPCC Systems®, 2011.

Apache Software Foundation: Apache™Hadoop®. 2019a. URL: https://hadoop.apache.org (zuletzt geprüft am 19. 09. 2019).

Apache Software Foundation: Apache Hive™. 2019b. URL: https://hive.apache.org (zuletzt geprüft am 19. 09. 2019).

Apache Software Foundation: Apache Spark®. 2019c. URL: https://spark.apache.org (zuletzt geprüft am 19. 09. 2019).

Apache Software Foundation: Apache TEZ®. 2019d. URL: https://tez.apache.org (zuletzt geprüft am 24. 09. 2019).

Apple Inc.: Swift. 2019. URL: https://swift.org (zuletzt geprüft am 15. 11. 2019).

Balbaert, I.: Getting Started with Julia -. Birmingham: Packt Publishing Ltd, 2015.

Brauer, J.: Typen, Objekte, Klassen - Teil 1: Grundlagen -. Elmshorn: Nordakademie, Hochschule der Wirtschaft, 2009.

Bylina, H.: Ruby programming language. Ruby on rails framework. In: Moderne Techniken und Technologien: Proceedings of the XX International Scientific and Practical Conference of Students, Postgraduates and Young Scientists, Tomsk, 14. bis 18. April 2014 T. 2.-Tomsk, 2014. Bd. 2. Ezdo TPU. 2014, S. 287–288.

Carbonnelle, P.: PYPL PopularitY. 2019. URL: http://pypl.github.io/PYPL.html (zuletzt geprüft am 22. 10. 2019).

Cardelli, L.; Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Comput. Surv. 17 (1985) 4, S. 471–523.

Cassandra, Apache: Apache Cassandra. 2014. URL: http://planetcassandra.org/what-is-apache-cassandra (zuletzt geprüft am 06. 11. 2019).

Chen, H.: Comparative Study of C, C++, C# and Java Programming Languages. (2010).

Chistenko, S.: Awesome Swift. 2019. URL: https://github.com/Wolg/awesome-swift (zuletzt geprüft am 22. 11. 2019).

Cleve, J.; Lämmel, U.: Data Mining. 2. Auflage. Berlin: De Gruyter Oldenourg, 2016.

Couto, M. L.; Pereira, R. A.; Ribeiro, F. J.; Rua, R.; Saraiva, J. A.: Towards a green ranking for programming languages. (2017).

cppreference.com: C++ Reference. 2019. URL: http://www.cplusplus.com/reference/ (zuletzt geprüft am 03. 11. 2019).

Czarnecki, L.: C# für Ingenieure - Mit Beispielen zur Analyse elektrischer Schaltungen. Berlin: Walter de Gruyter GmbH & Co KG, 2015.

Dausmann, M.; Bröckl, U.; Goll, J.; Schoop, D.: C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen. 7. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2010.

Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, 2004, S. 137–150.

Delfmann; Wimmer, T.; (Hrsg.), T. G.: Positionspapier zum Grundverständnis der Logistik als wissenschaftliche Disziplin. In: Flexibel – sicher – nachhaltig. 28. Hamburg: DVV Media Group GmbH, 2011, S. 262–274.

Eigner, M.; Gerhardt, F.; Gilz, T.; Mogo Nem, F.: Programmiersprachen und Techniken. In: Informationstechnologie für Ingenieure. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 183–237.

Ekanayake, J.; Gunarathne, T.; Fox, G.; Balkir, A. S.; Poulain, C.; Araujo, N.; Barga, R.: DryadLINQ for Scientific Analyses. In: 2009 Fifth IEEE International Conference on e-Science. 2009, S. 329–336.

Ernst, H.; Schmidt, J.; Beneken, G.; Schmidt, J.; Beneken, G.: Grundkurs Informatik - Grundlagen und Konzepte für die erfolgreiche IT-Praxis - Eine umfassende, praxisorientierte Einführung. 5. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2015.

Farkisch, K.: Data-Warehouse-Systeme kompakt - Aufbau, Architektur, Grundfunktionen. 1. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2011.

Fasel, D.; Meier, A.; Fasel, D.; Meier, A.: Big Data - Grundlagen, Systeme und Nutzungspotenziale. 1. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2016.

Feiler, J.: Learn Computer Science with Swift - Computation Concepts, Programming Paradigms, Data Management, and Modern Component Architectures with Swift and Playgrounds. 1. Aufl. New York: Apress, 2017.

Flanagan, D.: JavaScript - das umfassende Referenzwerk ; [behandelt Ajax und DOM-Scripting]. 3. Aufl. Köln: O'Reilly Germany, 2007.

Forward, A.; Lethbridge, T. C.: The Relevance of Software Documentation, Tools and Technologies: A Survey. In: Proceedings of the 2002 ACM Symposium on Document Engineering. DocEng '02. McLean, Virginia, USA: ACM, 2002, S. 26–33.

Fun, S.; Hardin, D.; Turnbull, M.: The real-time specification for Java. Addison-Wesley, 2000.

Gao, F.; Zhao, Q.: Big Data Based Logistics Data Mining Platform: Architecture and Implementation. International Journal of Interdisciplinary Telecommunications and Networking (IJITN) 6(4) (2016). doi:10.4018/IJITN.2014100103, S. 24–34.

Ghosh, A.: ETL-Process. 2019. URL: https://www.techapeek.com/2019/08/15/what-is-etl-extract-transform-and-load/ (zuletzt geprüft am 25. 10. 2019).

GitHub, Inc.: GitHub. 2019. URL: https://github.com (zuletzt geprüft am 22. 11. 2019).

Gobble, M. M.: Big Data: The Next Big Thing in Innovation. Research-Technology Management 56 (2013) 1, S. 64–67.

golang.org: The Go Programming Language Specification. 2019. URL: https://golang.org/ref/spec (zuletzt geprüft am 03. 11. 2019).

Grassmuck, V.: Freie Software. Geschichte, Dynamiken und gesellschaftliche Bezüge (2000).

Gronwald, K.-D.: Integrated Business Information Systems - A Holistic View of the Linked Business Process Chain ERP-SCM-CRM-BI-Big Data. 1st ed. 2017. Berlin, Heidelberg: Springer, 2017.

Güting, R. H.; Erwig, M.: Übersetzerbau: Techniken, Werkzeuge, Anwendungen. Springer-Verlag, 2013.

Han, J.; Pei, J.; Kamber, M.: Data Mining: Concepts and Techniques -. 3. Aufl. Amsterdam: Elsevier, 2011.

Hausladen, I.: IT-gestützte Logistik - Systeme - Prozesse - Anwendungen. 3. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2016.

Heiderich, N.; Meyer, W.: Technische Probleme lösen mit C/C++ - Von der Analyse bis zur Dokumentation. M: Carl Hanser Verlag GmbH Co KG, 2016.

Heiserich, O.-E.; Helbig, K.; Ullmann, W.: Logistik - Eine praxisorientierte Einführung. 4. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2011.

Heller, S.; Dittfurth, A.: PHP 5.4 - Grundlagen der Erstellung dynamischer Webseiten ; GPHP54. Bodenheim: Herdt, 2013.

Hennig, S.: Open Source Software: Wirtschaftlichkeitsanalysen -. Hamburg: Igel Verlag, 2014.

Hietaniemi, J.: Comprehensive Perl Archive Network (CPAN). 2019. URL: https://www.cpan.org (zuletzt geprüft am 22. 11. 2019).

HPCC Systems, LexisNexis Risk Solutions: High-Performance Computing Cluster. 2019. URL: https://hpccsystems.com (zuletzt geprüft am 19. 09. 2019).

IEEE Computer Society: Alan Edelman of MIT Recognized with Prestigious 2019 IEEE Computer Society Sidney Fernbach Award. 2019. URL: https://www.computer.org/press-room/2019-news/2019-ieee-fernbach-award-edelman (zuletzt geprüft am 31. 10. 2019).

Insa, M. A.: Awesome Ruby. 2019. URL: https://github.com/markets/awesome-ruby (zuletzt geprüft am 22. 11. 2019).

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: ACM SIGOPS operating systems review. Bd. 41. 3. ACM. 2007, S. 59–72.

JuliaLang.org: Julia Observer - julia language package browser. 2019. URL: https://juliaobserver.com (zuletzt geprüft am 21. 11. 2019).

Kannan, P.: Beyond Hadoop MapReduce Apache Tez and Apache Spark. San Jose State University. (2015).

Kapil, S.: Clean Python - Elegant Coding in Python. 1st ed. New York: Apress, 2019.

Kosar, T.; Bohra, S.; Mernik, M.: Domain-Specific Languages: A Systematic Mapping Study. Information and Software Technology 71 (2016), S. 77–91.

Krüger, G.; Stark, T.: Handbuch der Java-Programmierung. Pearson Deutschland GmbH, 2009.

Kull, A.: Awesome Java. 2019. URL: https://github.com/akullpp/awesome-java (zuletzt geprüft am 22. 11. 2019).

Küveler, G.: Informatik für Ingenieure und Naturwissenschaftler - Grundlagen : Programmieren mit C/C++ ; großes C/C++-Praktikum. 5, vollst. überarb. u. akt. Aufl. 2006. Wiesbaden: Vieweg, 2006.

Lang, B.: Der Kampf um die freie Software. Schweizerisches Jahrbuch für Entwicklungspolitik (2003) 22-2, S. 201–205.

Lehmacher, W.: Globale Supply Chain - Technischer Fortschritt, Transformation und Circular Economy. Berlin Heidelberg New York: Springer-Verlag, 2016.

Marr, B.: How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. 2019. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#314e517460ba (zuletzt geprüft am 26. 10. 2019).

Maurer, C.: Nichtsequentielle Programmierung mit Go 1 kompakt - Einführung in die Konzepte der grundlegenden Programmiertechniken für Betriebssysteme, Parallele Algorithmen, Verteilte Systeme und Datenbanktransaktionen. 2. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2012.

Meier, A.; Kaufmann, M.: SQL- & NoSQL-Datenbanken -. 8. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2016.

Mernik, M.; Heering, J.; Sloane, A. M.: When and How to Develop Domain-specific Languages. ACM Comput. Surv. 37 (2005) 4, S. 316–344.

Meyerson, J.: The go programming language. IEEE software 31 (2014) 5, S. 104–104.

Microsoft Corporation: Dryad and DryadLINQ - Unlocking the Power of Parallelism and Data Centers. White Paper. Microsoft Corporation®, 2009.

Microsoft Corporation: C# Reference. 2019a. URL: https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/ (zuletzt geprüft am 03. 11. 2019).

Microsoft Corporation: Dryad Project. 2019b. URL: https://www.microsoft.com/en-us/research/project/dryad/ (zuletzt geprüft am 08. 10. 2019).

Mishra, H.; S, J.; Chala, A.; Camper, D.; G, S.; Shetty, J.: Data Skew Profiling Using HPCC Systems. In: Proceedings of the 2019 International Conference on Big Data and Education. ICBDE'19. London, United Kingdom: ACM, 2019, S. 66–69.

Mozilla Foundation: JavaScript. 2019. URL: https://developer.mozilla.org/de/docs/Web/JavaScript (zuletzt geprüft am 03. 11. 2019).

NESSI: Big Data -. A New World of Opportunities. White Paper. NESSI (Networked European Software and Services Initiative), 2012.

Nokia Corporation: Disco-Project. 2019. URL: http://discoproject.org (zuletzt geprüft am 19. 09. 2019).

North, K.: Wissensorientierte Unternehmensführung - Wissensmanagement gestalten. 6. akt. und erw. Aufl. 2016, Korr. Nachdruck 2016. Wiesbaden: Springer Fachmedien Wiesbaden, 2016.

Oracle: Java^{TM}Platform, Standard Edition 7 API Specification. 2019. URL: https://docs.oracle.com/javase/7/docs/api/overview-summary.html (zuletzt geprüft am 03. 11. 2019).

Pepper, P.: Grundlagen der Informatik. 2., verbesserte Auflage. Reprint 2018. Berlin: Walter de Gruyter GmbH & Co KG, 2018.

Perl.org: Perl 5 version 30.0 documentation. 2019. URL: https://perldoc.perl.org/5.30.0/index-language.html (zuletzt geprüft am 03. 11. 2019).

Pfeffer, J.: Online-Tutorials an deutschen Universitäts- und Hochschulbibliotheken - Verbreitung, Typologie und Analyse am Beispiel von LOTSE, DISCUS und BibTutor. Freuburg i.B., 2005.

Purer, K.: PHP vs. Python vs. Ruby–The web scripting language shootout. Vienna University of Technology (2009).

Python Software Foundation: The Python Language Reference. 2019a. URL: https://docs.python.org/3/reference/ (zuletzt geprüft am 03. 11. 2019).

Python Software Foundation: The Python Package Index (PyPI). 2019b. URL: https://pypi.org (zuletzt geprüft am 22. 11. 2019).

Qt Group (Nasdaq Helsinki: QTCOM): Qt Concurrent. 2019. URL: https://doc.qt.io/qt-5/qtconcurrent-index.html (zuletzt geprüft am 19. 09. 2019).

Rebouças, M.; Pinto, G.; Ebert, F.; Torres, W.; Serebrenik, A.; Castor, F.: An Empirical Study on the Usage of the Swift Programming Language. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Bd. 1. 2016, S. 634–638.

RedMonk: The RedMonk Programming Language Rankings: June 2019. 2019. URL: https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/?utm_source=rss&utm_medium=rss&utm_campaign=language-rankings-1-19 (zuletzt geprüft am 22. 10. 2019).

Rowley, J.: The wisdom hierarchy: representations of the DIKW hierarchy. Journal of Information Science 33 (2007) 2, S. 163–180. eprint: https://doi.org/10.1177/0165551506070706.

Ruby-Community: Ruby Dokumentation. 2019. URL: https://www.ruby-lang.org/de/documentation/ (zuletzt geprüft am 05. 11. 2019).

Saha, B.; Shah, H.; Seth, S.; Vijayaraghavan, G.; Murthy, A.; Curino, C.: Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In:

Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, S. 1357–1369.

Sanner, M. F. et al.: Python: a programming language for software integration and development. J Mol Graph Model 17 (1999) 1, S. 57–61.

Schnider, D.; Jordan, C.; Welker, P.; Wehner, J.: Data Warehouse Blueprints - Business Intelligence in der Praxis. M: Carl Hanser Verlag GmbH Co KG, 2016.

Schröter, J.; Seidel, H.: Perl - Grundlagen und effektive Strategien. 1. Aufl. Deutschland: Oldenbourg, 2003.

Sherrington, M.: Mastering Julia. Birmingham: Packt Publishing Ltd, 2015.

Shoro, A. G.; Soomro, T. R.: Big Data Analysis: Apache Spark Perspective. Global Journal of Computer Science and Technology (2015).

Singh, R.; Kaur, P. J.: Analyzing performance of Apache Tez and MapReduce with hadoop multinode cluster on Amazon cloud. Journal of Big Data 3 (2016) 1, S. 19.

Stack Exchange Inc.: Developer Survey Results 2019. 2019. URL: https://insights.stackoverflow.com/survey/2019#community (zuletzt geprüft am 26. 10. 2019).

Steyer, R.: JavaScript - Die universelle Sprache zur Web-Programmierung. M: Carl Hanser Verlag GmbH Co KG, 2014.

ten Hompel, M.; Heistermann, F.; Rehof, J.: Logistik und IT als Innovationstreiber für den Wirtschaftsstandort Deutschland - Die neue Führungsrolle der Logistik in der Informationstechnologie. These 1. Bremen: Bundesvereinigung Logistik (BVL) e.V., 2014.

The jQuery Foundation: jQuery UI. 2019. URL: https://jqueryui.com (zuletzt geprüft am 22. 11. 2019).

The PHP Group: PHP Documentation. 2019. URL: https://www.php.net/docs.php (zuletzt geprüft am 03. 11. 2019).

The R Foundation: The R Project for Statistical Computing. 2019. URL: https://www.r-project.org (zuletzt geprüft am 22. 11. 2019).

TIOBE Software BV: TIOBE Index. 2019. URL: https://www.tiobe.com/tiobe-index/ (zuletzt geprüft am 22. 10. 2019).

Vasilescu, B.; Filkov, V.; Serebrenik, A.: Stackoverflow and github: Associations between software development and crowdsourced knowledge. In: 2013 International Conference on Social Computing. IEEE. 2013, S. 188–195.

Voulgaris, Z.: Julia for Data Science. Bradley Beach: Technics Publications, 2016.

Wells, G.: The Future of iOS Development: Evaluating the Swift Programming Language. CMC Senior Theses. 1179., Bachelor, 2015.

Wilde, M.; Hategan, M.; Wozniak, J. M.; Clifford, B.; Katz, D. S.; Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing 37 (2011) 9. Emerging Programming Paradigms for Large-Scale Scientific Computing, S. 633–652.

Wollschläger, D.: R kompakt - Der schnelle Einstieg in die Datenanalyse. 2. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2016.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S.; Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12. San Jose, CA: USENIX Association, 2012, S. 2–2.

Zaharia, M.; Xin, R. S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M. J.; Ghodsi, A.; Gonzalez, J.; Shenker, S.; Stoica, I.: Apache Spark: A Unified Engine for Big Data Processing. Commun. ACM 59 (2016) 11, S. 56–65.

Zapponi, C.: GITHUT. 2019. URL: https://githut.info (zuletzt geprüft am 28. 10. 2019).

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Abkürzungsverzeichnis

ANSI	American National Standards Institute
Apache	Apache Software Foundation™
Apple	Apple Inc.
BSD-Lizenz	Berkeley Software Distribution Lizenz
Cassandra	Apache Cassandra™
DAG	gerichtete aperiosche Graphen (engl. Directed Acyclic Graph)
DoF	Erfüllungsgrad (engl. degree of fullfillment)
DSL	domänenspezifische Sprache (engl. Domain-Specific Language)
ECL	Enterprise Control Language
ETL-Prozess	Prozess zur Vereinigung von Daten (engl. Extract, Transform, Load - Process)
GNU GPL	GNU General Public License
Google	Google Inc.
GPL	Allzwecksprache (engl. General Purpose Language)
GUI	grafische Benutzeroberfläche (engl. graphical user interface)
Hadoop	The Apache™Hadoop®
HPCC	High Performance Computing Cluster
ISO	International Organization for Standardization
JIT-Compiler	Just-In-Time-Compiler
LINQ	Language Integrated Query
Microsoft	Microsoft Corporation™
OSS	Open-Source-Software
PSF-Lizenz	Python-Software-Foundation-Lizenz
RDDs	*Belastbare Verteilte Datensätze* (engl. Resilient Distributed Datasets)
Spark	Apache Spark™
TEZ	Apache TEZ™
VB	Visual Basic™

Anhang A: Statistiken zur Verbreitung von Programmiersprachen

Tabelle A.1: TIOBE Index von Oktober 2019 nach TIOBE Software BV (2019)

Platz	Programmiersprache	Anteil	Platz	Programmiersprache	Anteil
1	Java	16,88%	11	Groovy	1,39%
2	C	16,18%	12	Swift	1,36%
3	Python	9,09%	13	Ruby	1,32%
4	C++	6,23%	14	AssemblyLanguage	1,31%
5	C#	3,86%	15	R	1,26%
6	Visual Basic .NET	3,75%	16	Visual Basic	1,23%
7	JavaScript	2,08%	17	Go	1,10%
8	SQL	1,94%	18	Delphi/Object Pascal	1,05%
9	PHP	1,91%	19	Perl	1,02%
10	Objective-C	1,50%	40	Julia	0,02%

Tabelle A.2: RedMonk Ranking von Juni 2019 nach RedMonk (2019)

Platz	Programmiersprache	Platz	Programmiersprache
1	JavaScript	11	Swift
2	Java	12	TypeScript
3	Python	13	Scala
4	PHP	14	Shell
5	C#	15	Go
6	C++	16	R
7	CSS	17	PowerShell
8	Ruby	18	Perl
9	C	19	Haskell
10	Objective-C	34	Julia

Tabelle A.3: PYPL PopularitY von Oktober 2019 nach Carbonnelle (2019)

Platz	**Programmiersprache**	**Platz**	**Programmiersprache**
1	Python	13	Ruby
2	Java	14	VBA
3	JavaScript	15	Go
4	C#	16	Scala
5	PHP	17	Visual Basic
6	C/C++	18	Rust
7	R	19	Perl
8	Objective-C	20	Lua
9	Swift	21	Haskell
10	Matlab	22	Julia
11	TypeScript	23	Delphi
12	Kotlin		

Anhang B: Eigenschaften der ausgewählten Programmiersprachen

Tabelle B.1: Eigenschaften der Programmiersprache C

Eigenschaften	
Kategorie	imperativ
Objektorietiert	nein
Datenbanken	externe Bibliotheken
Datenströme	integriert
GUI	externe Bibliotheken
Übersetzung	Compiler (Visual C++ und weitere GNU Compiler)
Typisierung	statisch, implizit, stark
Parallelität	möglich
Standard-Bibliotheken	File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Standadisiert von ANSI und ISO

Tabelle B.2: Eigenschaften der Programmiersprache C++

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	externe Bibliotheken
Datenströme	integriert
GUI	externe Bibliotheken
Übersetzung	Compiler (Visual C++, GCC und weitere)
Typisierung	statisch, implizit, stark
Parallelität	möglich
Standard-Bibliotheken	File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Standadisiert von ISO

Tabelle B.3: Eigenschaften der Programmiersprache C#

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	integriert
Datenströme	integriert
GUI	integriert
Übersetzung	Compiler (Roslyn, Mono, DotGNU, CoreRT)
Typisierung	statisch, explizit, stark
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke, Statistik
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	proprietär (Microsoft)

Tabelle B.4: Eigenschaften der Programmiersprache Go

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	integriert
Datenströme	integriert
GUI	integriert
Übersetzung	Compiler (Gc, Gccgo)
Typisierung	statisch, implizit, stark
Parallelität	ja
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (BSD-Lizenz)

Tabelle B.5: Eigenschaften der Programmiersprache Java

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	externe Bibliotheken
Datenströme	integriert
GUI	integriert
Übersetzung	Compiler
Typisierung	statisch, stark
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (GNU GPL)

Tabelle B.6: Eigenschaften der Programmiersprache JavaScript

Eigenschaften	
Kategorie	Skriptsprache
Objektorietiert	möglich
Datenbanken	integriert
Datenströme	integriert
GUI	mittels HTML
Übersetzung	Interpreter
Typisierung	dynamisch, schwach
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (BSD-Lizenz)

Tabelle B.7: Eigenschaften der Programmiersprache Perl

Eigenschaften	
Kategorie	prozedural
Objektorietiert	teilweise
Datenbanken	integriert
Datenströme	integriert
GUI	externe Bibliotheken
Übersetzung	Interpreter
Typisierung	dynamisch, implizit, schwach
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (GNU GPL und Artistic License)

Tabelle B.8: Eigenschaften der Programmiersprache PHP

Eigenschaften	
Kategorie	prozedural
Objektorietiert	ja
Datenbanken	integriert
Datenströme	integriert
GUI	integriert und mittels HTML
Übersetzung	Interpreter
Typisierung	dynamisch, schwach
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke, Statistik
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	teilweise Open-Source (PHP-Lizenz), teilweise proprietär

Tabelle B.9: Eigenschaften der Programmiersprache Python

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	integriert (SQLite)
Datenströme	externe Bibliotheken
GUI	integriert
Übersetzung	diverse Compiler (u.a. JIT-Compiler))
Typisierung	statisch, stark
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke, Statistik
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (PSF-Lizenz)

Tabelle B.10: Eigenschaften der Programmiersprache R

Eigenschaften	
Kategorie	funktional
Objektorietiert	ja
Datenbanken	integriert
Datenströme	integriert
GUI	integriert
Übersetzung	Interpreter (diverse)
Typisierung	dynamisch, implizit, schwach
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (GNU GPL)

Tabelle B.11: Eigenschaften der Programmiersprache Ruby

Eigenschaften	
Kategorie	GPL
Objektorietiert	vollständig
Datenbanken	integriert
Datenströme	integriert
GUI	integriert (diverse)
Übersetzung	Interpreter (CRuby und weitere)
Typisierung	dynamisch, stark
Parallelität	möglich
Standard-Bibliotheken	Datenbanken, File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (BSD-Lizenz)

Tabelle B.12: Eigenschaften der Programmiersprache Swift

Eigenschaften	
Kategorie	GPL
Objektorietiert	ja
Datenbanken	integriert
Datenströme	integriert
GUI	integriert
Übersetzung	Compiler (LLVM)
Typisierung	statisch, stark
Parallelität	möglich
Standard-Bibliotheken	File-IO, Mathematik, Netwerke
Externe Bibliotheken	Datenbanken, Netzwerke, Mathematik, GUI, Statistik
Lizensierung	Open-Source (Apache-Lizenz 2.0)

Anhang C: Abgrenzung der ausgewählten Programmiersprachen

Tabelle C.1: Vollständige Abgrenzung anhand der ermittelten Kriterien für die vorgestellten Programmiersprachen

Kriterium	**C**	**C++**	**C#**	**Go**	**Java**	**JS**	**Julia**	**Perl**	**PHP**	**Python**	**R**	**Ruby**	**Swift**
Zugriff auf Daten	50	50	100	100	100	100	100	100	100	100	100	50	100
Grafische Aufbereitung	50	50	100	100	100	50	50	50	100	100	100	100	100
Transformierbarkeit	100	100	100	100	100	100	100	100	100	100	100	100	100
Typenunabhängigkeit	70	70	80	70	70	80	100	80	80	80	80	100	80
Effiziente Ausführung	100	100	99	98	99	96	100	38	60	87	0	60	98
Effiziente Anwendung	50	50	50	50	50	100	50	100	100	50	100	100	50
Parallelisierbarkeit	100	100	100	100	100	100	100	100	100	100	100	100	100
Verfügbarkeit	100	100	0	100	100	100	100	100	50	100	100	100	100
Interne Bibliotheken	60	60	100	80	80	80	100	80	100	100	80	80	60
Eigene Bibliotheken	100	100	100	100	100	100	100	100	100	100	100	100	100
Externe Bibliotheken	100	100	100	100	100	100	100	100	100	100	100	100	100
Dokumentation	100	100	100	100	100	100	100	100	100	100	100	100	100
Anleitung	76	81	87	45	96	95	0	31	86	97	63	61	66
Eignung der Sprache	83	83	92	90	93	94	91	87	93	95	88	88	91

Eidesstattliche Versicherung
(Affidavit)

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.