Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Masterarbeit

Ein Verfahren zur Erzeugung großer Test-Datasets für die Duplikaterkennung unter Verwendung von Apache Spark

Kai Hildebrandt

6hildebr@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 5876145

Erstgutachter: Prof. Dr. Norbert Ritter
Zweitgutachter: Dr. Dirk Bade
Betreuer: Dr. Fabian Panse, Wolfram Wingerath, Steffen Friedrich

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 1
1.2. Problemstellung . . . 2
1.3. Zielsetzung . . . 3
1.4. Aufbau der Arbeit . . . 3

2. Grundlagen **5**
2.1. Der Dataset-Begriff . . . 5
2.2. Skalierbarkeit . . . 5
2.3. Duplikaterkennung . . . 6
2.3.1. Formales Modell . . . 6
2.3.2. Rolle der Datenqualität . . . 7
2.3.3. Der Prozess der Duplikaterkennung . . . 11
2.3.4. Test-Daten für die Duplikaterkennung . . . 14
2.4. Apache Spark . . . 16
2.4.1. Cluster Computing . . . 17
2.4.2. Allgemeine Informationen . . . 17
2.4.3. Sofware-Architektur . . . 18
2.4.4. Cluster-Architektur & Deployment . . . 19
2.4.5. Resilient Distributed Datasets . . . 20
2.4.6. Operationen . . . 22
2.4.7. Narrow-Dependencies vs. Wide-Dependencies . . . 23

3. Verwandte Arbeiten **25**
3.1. DBGen . . . 26
3.2. Febrl data set generator . . . 28
3.3. GeCo . . . 30
3.4. TDGen . . . 32
3.5. ProbGee . . . 34
3.6. Vergleich & Fazit . . . 36

4. Verfahren **39**
4.1. Motivation und Einordnung . . . 39
4.2. Anforderungen an das Verfahren . . . 39

4.3. Instrumente zur Umsetzung der Anforderungen 40
4.3.1. Parallelisierung & Verteilung 41
4.3.2. Data Profiling 41
4.3.3. Angereichertes Tabellenschema 41
4.3.4. Automatisierte Vorkonfiguration 41
4.3.5. Fehlermodell 42
4.4. Workflow 43
4.4.1. Analyse des Datasets 44
4.4.2. Semiautomatische Konfiguration 45
4.4.3. Ausführung des Verarbeitungsprozesses 47

5. Implementation: DaPo – Data Polluter 49
5.1. Apache Spark als Plattform 49
5.2. Scala als Programmiersprache 50
5.3. Softwarearchitektur 50
5.4. Basis-Datentypen & Abstrakte Datentypen 51
5.5. Prolog: Vorbereitung und Ausführung von DaPo 51
5.5.1. Basis-Konfiguration 51
5.5.2. Run-Modi 52
5.5.3. Ausführung 52
5.6. Der DaPo-Workflow 53
5.6.1. Analyse des Datasets 53
5.6.2. Konfiguration des Tabellenschemas 55
5.6.3. Konfiguration des Fehlerschemas 56
5.6.4. Ausführung des Verarbeitungsprozesses 56
5.7. Fehlermodell 59
5.7.1. Typ-Hierarchie der Fehler-Komponenten 59
5.7.2. Die verschiedenen Fehler-Komponenten 60
5.7.3. Fehlervererbung 63
5.7.4. Beispiel: Anwendung eines Fehlerschemas 63
5.8. Weitere Implementationsdetails 65
5.8.1. Die Klasse AttributeReasonerSimple 65
5.8.2. Die Klasse ErrorSchemaReasonerSimple 66
5.8.3. Partitionierung nach Duplikatcluster-ID 68
5.8.4. Explizites Caching 68

6. Experimentelle Evaluation 71
6.1. Datasets 71
6.2. Setup des Computerclusters 72
6.2.1. Konfigurationen der Cluster-Architektur 73

6.3. Experimente 73
6.3.1. Variieren der Tupel-Menge 74
6.3.2. Variieren der Ziel-Ähnlichkeit 76
6.3.3. Fazit 77
6.4. Vergleich mit anderen Tools 78

7. Abschluss **79**
7.1. Zusammenfassung & Fazit 79
7.2. Ausblick 80
7.2.1. Ansätze zur Verbesserung des Verfahrens 80
7.2.2. Ansätze zur Verbesserungen von DaPo 80

Literaturverzeichnis **83**

A. Anhang **89**
A.1. Schema der MusicBrainz-Datenbank 89
A.2. Beispiel einer Basis-Konfiguration 90
A.3. Ableiten eines abstrakten Datentyps 91
A.4. Liste der implementierten Fehler-Komponenten 92
A.4.1. BlockErrorComponents 92
A.4.2. ColErrorComponents 92
A.4.3. RowErrorComponents 92
A.4.4. FieldErrorComponents 93

Eidesstattliche Erklärung **97**

1. Einleitung

Im Kontext von Big Data und Cluster Computing wurde im Rahmen dieser Masterarbeit ein neuartiges Verfahren zum Ableiten großer Test-Datasets für die Duplikaterkennung aus vorhandenen Daten konzipiert und implementiert. Der Fokus lag dabei auf Effizienz, Skalierbarkeit und flexibler Konfigurierbarkeit. Für die Umsetzung wurde das Cluster-Computing-Framework Apache Spark [ZCF+10, ZCD+12] eingesetzt.

In diesem ersten Kapitel wird zunächst das Thema motiviert und die Problemstellung aufgezeigt. Anschließend wird die Zielsetzung abgesteckt und der Aufbau der Arbeit vorgestellt.

1.1. Motivation

Es kann im allgemeinen vorkommen, dass in Datenbeständen sogenannte Duplikate existieren [HS98]. Dies bedeutet, dass für dasselbe Objekt der realen Welt mehrere Repräsentationen in der Datenbasis vorliegen. Das ist i. d. R. unerwünscht, da es die Datenqualität senkt [Nau07, NH10]. Die Ursachen für die Existenz von Duplikaten können vielfältig sein. Bezogen auf eine einzelne Datenbasis können z. B. fehlerhafte Eingaben (Tippfehler, mehrfache Eingaben), unterschiedliche Eingabe-Konventionen oder signifikante Veränderungen des betroffenen Realwelt-Objektes zwischen zwei Eingaben, ursächlich sein [Chr12]. Eine andere häufige Quelle für Duplikate ist die Durchführung einer Integration mehrerer Datenbasen, bei der z. B. unterschiedliche Datenschemata, Konventionen oder Eingabe-Voraussetzungen zu voneinander abweichenden Repräsentationen eines Realwelt-Objektes führen können [DHI12]. Die effektive und effiziente Identifizierung solcher Duplikate ist eine komplexe Aufgabe und ist Gegenstand des Forschungsgebietes der *Duplikaterkennung* (siehe Abschnitt 2.3).

Zur Entwicklung, Erprobung und Evaluation von Methoden der Duplikaterkennung werden Test-Daten benötigt [Chr12, S. 163ff]. Hierfür haben sich Test-Datasets etabliert, welche sich dadurch auszeichnen, dass sie eine tabellarische Struktur aufweisen (z. B. CSV-Dateien oder relationale Datenbanktabellen) und dass der zugehörige Goldstandard vorhanden ist. Letzteres bedeutet, dass bekannt ist, welche Tupel der Datenmenge Duplikate zueinander sind. Darüber hinaus sollte die Charakteristik der Test-Daten den tatsächlich im Einsatz zu verarbeitenden Daten so ähnlich wie möglich sein. Idealerweise sollten als Test-Daten echte Daten aus dem (geplanten) Einsatzkontext verwendet werden. Das ist in der Realität allerdings aus verschiedenen Gründen häufig nicht möglich (z. B. keine Daten vorhanden, kein Zugriff oder keine Verwendungserlaubnis). Sollten

hingegen echte Daten vorhanden sein und benutzt werden dürfen, steht der tatsächlichen Verwendung oft noch ein fehlender Goldstandard im Wege (siehe Abschnitt 2.3.4).

1.2. Problemstellung

Um die Duplikaterkennung in das Zeitalter von *Big Data* und *Cluster Computing* zu heben, wurden in der jüngeren Forschung Methoden der Duplikaterkennung für verteilte Programmiermodelle [KKH+10] und Cluster-Computing-Frameworks wie Apache Hadoop (MapReduce) [VCL10, KTR12] oder Apache Spark [Wil15a] entwickelt. Solche Frameworks erlauben effiziente, skalierbare und verteilte Berechnungen auf sehr großen Datenmengen.

Zur Beschaffung von Test-Datasets für die Duplikaterkennung haben sich in der Forschung verschiedene Ansätze etabliert: Das (semi-)manuelle Klassifizieren von Duplikaten [Chr12, S. 174], das Verwenden existierender öffentlicher Test-Datasets [Bil03], Tools zum Ableiten von Test-Datasets aus vorhandenen Daten [BR12, FW12] und Tools zum Synthetisieren von Test-Daten [HS98, Chr09, CV13] (siehe Abschnitt 2.3.4). Um die Eignung und Performanz der oben genannten neuartigen Methoden der Duplikaterkennung sinnvoll evaluieren zu können, sind auch hinreichend große Test-Datasets notwendig. Diese neue Aufgabe stellt die bisherigen Ansätze zur Beschaffung von Test-Datasets allerdings vor Probleme:

- Das (semi-)manuelle Klassifizieren von Duplikaten ist für große Datasets ungeeignet, da es einen sehr arbeits- und zeitaufwendigen Begutachtungsprozess erfordert. (siehe Abschnitt 2.3.4).
- Die aktuell öffentlich verfügbaren Test-Datasets enthalten nur bis zu 64.000 Tupel (siehe Abschnitt 2.3.4).
- Zwar können einige Tools zur Synthetisierung prinzipiell große Test-Datasets erzeugen, allerdings sind diese entweder sehr unflexibel und führen zu unrealistischen Daten (DBGen), fehlerhaft (Febrl data set generator) oder ihre Verwendung bedarf eines großen Aufwandes (GeCo) (siehe Kapitel 3).
- Die existierenden Tools zum Ableiten von Test-Datasets aus vorhandenen Daten können auf handelsüblichen Desktop-Computern[1] bei annehmbarer Laufzeit nur mit Tupel-Mengen umgehen, deren Größen sich maximal im sechsstelligen Bereich bewegen (siehe Kapitel 3).

In dieser Arbeit wird ein Dataset als "groß" bezeichnet, wenn es mindestens zehn Million Tupel enthält und eine Größe von ca. 1 GB aufweist. Für die oben genannten neuartigen Methoden der Duplikaterkennung sind die vorhandenen Ansätze und Tools somit ungeeignet.

[1]Computer mit Intel Core i5-3570 Quad-Core-Prozessor 3.40 GHz und 8 GB RAM

Da es im Web durchaus sehr große öffentlich zugängliche Datenquellen gibt (z. B. *IMDb*[2], *MusicBrainz*[3], *DBLP*[4], *Wikipedia*[5], uvm.) ist es naheliegend, diese zur Lösung des Problems hinzuzuziehen. Das Ableiten von Test-Datasets aus vorhandenen Daten ist daher ein vielversprechender Ansatz, aber leider sind die öffentlich verfügbaren Tools dafür nicht leistungsfähig genug.

1.3. Zielsetzung

Im Rahmen dieser Masterarbeit soll ein Verfahren entwickelt werden, welches es ermöglicht unter Einsatz eines Computerclusters große Test-Datasets aus vorhandenen großen Datasets abzuleiten. Dabei sollen folgende Anforderungen erfüllt werden:

- **Effizienz & Skalierbarkeit:** Es soll möglich sein auf einem handelsüblichen Desktop-Computer Test-Datasets mit mindestens zehn Millionen Tupeln und einer Größe von ca. 1 GB in weniger als einer Stunde zu erzeugen. Zudem soll das Tool horizontal skalierbar sein, so dass die Hinzunahme weiterer Computer die Leistungsfähigkeit bezüglich Laufzeit und Datenmenge steigert.
- **Schemaunabhängigkeit:** Es soll unkompliziert möglich sein beliebige relationale Datasets zu verarbeiten.
- **realistische Fehler:** Um die erzeugten Test-Datasets in der Duplikaterkennung sinnvoll einsetzen zu können, sollten die injizierten Fehler denen realer Datasets der jeweiligen Domäne möglichst nahe kommen.
- **flexible Konfigurierbarkeit:** Auf der einen Seite soll das Tool viele Möglichkeiten der Konfiguration bieten. Auf der anderen Seite soll der Mindestaufwand bei der Konfiguration so gering wie möglich sein. Diese beiden Anforderungen gilt es in Einklang zu bringen.

1.4. Aufbau der Arbeit

Anschließend an diese Einleitung werden in Kapitel 2 zunächst die thematischen Hintergründe dargestellt, sowie die theoretischen und technischen Grundlagen für das entwickelte Verfahren gelegt. Dabei wird zunächst der Themenkomplex der Duplikaterkennung vorgestellt und die Bedeutung von Test-Daten in diesem erläutert. Anschließend wird das Cluster-Computing-Framework Apache Spark vorgestellt.

Kapitel 3 bietet eine Übersicht verschiedener Tools zum Generieren von Test-Datasets für die Duplikaterkennung. Diese werden auf Basis bestimmter Kriterien untersucht und anschließend miteinander verglichen.

[2] Internet Movie Database (IMDb): `http://www.imdb.com/`
[3] MusicBrainz: `http://musicbrainz.org/`
[4] Digital Bibliographic & Library Project (DBLP): `http://dblp.uni-trier.de/`
[5] Wikipedia: `https://www.wikipedia.org/`

In Kapitel 4 wird das konzipierte Verfahren zum Ableiten großer Test-Datasets für die Duplikaterkennung aus vorhandenen Daten vorgestellt. Nach einer kurzen Einordnung werden zunächst einige Anforderungen an das Verfahren formuliert. Es folgt die Vorstellung des eigentlichen Konzeptes, welches bestimmte Instrumente zur Umsetzung der Anforderungen beinhaltet, sowie den Vorschlag eines Workflows, in welchen diese Instrumente eingebettet sind.

In Kapitel 5 wird das Tool *Data Polluter (DaPo)* vorgestellt, wobei es sich um eine prototypische Implementierung des Verfahrens handelt, welches im Rahmen dieser Arbeit konzipiert wurde und in Kapitel 4 beschrieben ist. Zu Beginn wird die Wahl des verwendeten Frameworks begründet und die verwendete Programmiersprache Scala charakterisiert. Nach einer Beschreibung der Softwarearchitektur und einer Einführung in die Benutzung, wird die Implementierung im Detail vorgestellt. Dafür wird zunächst der DaPo-Workflow dargestellt und danach die Implementierung des Fehlermodells erläutert. Anschließend werden noch einige weitere interessante Aspekte dieser Implementation vertieft betrachtet.

In Kapitel 6 wird die experimentelle Evaluationen des Data Polluters beschrieben. Dabei werden insbesondere das Laufzeitverhalten und die Skalierbarkeit betrachtet.

Im abschließenden Kapitel 7 werden die Ergebnisse dieser Arbeit noch einmal zusammengefasst, ein Fazit gezogen und ein Ausblick auf mögliche zukünftige Erweiterungen gegeben.

2. Grundlagen

In diesem Kapitel werden Themen beleuchtet, welche Motivation sowie theoretische und technische Basis für das Verfahren sind, das in dieser Masterarbeit entwickelt wurde. Die ausgewählten Inhalte sollen also zum Verständnis des Hauptteils dieser Arbeit beitragen.

Hierfür wird zunächst in Abschnitt 2.1 der verwendete Dataset-Begriff definiert, anschließend in Abschnitt 2.3 der Themenkomplex der Duplikaterkennung vorgestellt und die Bedeutung von Test-Daten erläutert. Abschnitt 2.4 ist dem Cluster-Computing-Framework Apache Spark gewidmet.

2.1. Der Dataset-Begriff

In dieser Arbeit bezeichnet der Begriff *Dataset* eine Menge von Tupeln einer Relation. Ein Dataset kann prinzipiell in verschiedenen Formen vorliegen, wie z. B. als Datenbank-Tabelle oder CSV-Datei.

Die Größe eines Datasets kann hinsichtlich verschiedener Merkmale gemessen werden. Dazu gehören die Größe des benötigten Speichers (z.B. 10 MB), die Anzahl der Tupel oder auch die Anzahl der Datenfelder (d. h. die Anzahl der Attribute einer Relation multipliziert mit der Anzahl der Tupel).

Wenn es nicht explizit anders angegeben ist, wird in dieser Arbeit mit der Größe eines Datasets die Anzahl seiner Tupel beschrieben. Dabei wird von einer bestimmten Größenordnung eines Tupels ausgegangen: So hat ein Tupel im Durchschnitt eine Größe von 80 – 135 Bytes und besteht aus 8 – 17 Attributen.

2.2. Skalierbarkeit

Als Skalierbarkeit wird nach Kaul [Kau11] die Eigenschaft eines Systems bezeichnet, sich wachsenden Ansprüchen an die Leistungsfähigkeit anzupassen. Systeme können sowohl auf Hardware- als auch auf Softwareebene skaliert werden. Die Skalierung auf Hardwareebene wird durch Bereitstellung zusätzlicher Ressourcen vorgenommen. Dabei wird zwischen vertikaler und horizontaler Skalierbarkeit unterschieden.

Definition 2.2.1 (Vertikale Skalierbarkeit)**.** *Ein System ist vertikal skalierbar, wenn eine Erweiterung der Ressourcen eines bestehenden Knotens des Systems zu einer Steigerung der Systemleistung führt.*

Beispiele für eine vertikale Skalierung sind eine Erweiterung des Arbeitsspeichers oder das Hinzufügen von zusätzlichen Prozessorkernen.

Definition 2.2.2 (Horizontale Skalierbarkeit)**.** *Ein System ist horizontal skalierbar, wenn die Hinzunahme eines neuen System-Knotens zu einer Steigerung der Systemleistung führt.*

Beispiele für eine horizontale Skalierung sind die Migration eines Software-Systems von einem Desktop-Computer auf ein Computercluster oder die Hinzunahme zusätzlicher Knoten in einem bestehenden Computercluster.

2.3. Duplikaterkennung

Im Folgenden werden zunächst die wichtigsten Begriffe der Duplikaterkennung formal definiert (Abschnitt 2.3.1), um danach die Rolle der Datenqualität zu erläutern (Abschnitt 2.3.2) und schließlich die Duplikaterkennung als Prozess vorzustellen (Abschnitt 2.3.3). Zum Abschluss werden die Rolle von Test-Daten aufgezeigt und einige typische Ansätze zu ihrer Erzeugung (Abschnitt 2.3.4) diskutiert.

2.3.1. Formales Modell

Als formale Grundlage für diese Arbeit, werden in diesem Abschnitt einige Begriffe aus dem Gebiet der Duplikaterkennung definiert. Diese Definitionen bilden eine (zum Teil) vereinfachte Version des formalen Modells aus [Pan15].

Zunächst werden als Basis die Begriffe Realwelt und Realwelt-Entität eingeführt.

Definition 2.3.1 (Realwelt-Entität)**.** *Eine Realwelt-Entität ist ein wohlunterscheidbares Ding (bzw. Gegenstand oder Objekt) aus dem betrachteten Diskursuniversums. Die Menge aller Realwelt-Entitäten wird als Realwelt* $\mathfrak{W}$ *bezeichnet.*

Hiervon ausgehend wird der Begriff *Datenbank-Entität* definiert. Da sich die Arbeit im Kontext von relationalen Daten bewegt und die Betrachtung stets auf eine einzelne relationale Tabelle beschränkt ist, wird später meist der für Entitäten in relationalen Datenbanken spezifischere Begriff *Tupel* benutzt.

Definition 2.3.2 (Datenbank-Entität)**.** *Eine Datenbank-Entität ist eine digitale Repräsentation einer Realwelt-Entität. Dementsprechend ist eine Datenbank-Entität ein wohlunterscheidbares Ding (bzw. Gegenstand oder Objekt) aus allen digitalen Abbildern der Realwelt.*

Es folgen die Definitionen der allgemeinen Begriffe *Cluster* und *Clustering*. Im weiteren Verlauf der Arbeit werden allerdings i. d. R. die Begriffe *Duplikatcluster* bzw. *Duplikatclustering* benutzt, um zu verdeutlichen mit welcher Semantik das entsprechende Cluster bzw. Clustering besetzt ist.

Definition 2.3.3 (Cluster & Clustering)**.** *Ein Cluster* $C = \{e_1, \ldots, e_k\}$ *ist eine Menge von Basis-Elementen (z. B. Datenbank-Entitäten). Ein Clustering* $\mathcal{C} = \{C_1, \ldots, C_l\}$ *ist eine Menge von nicht-leeren Clustern.*

Um Duplikatbeziehungen ausdrücken zu können, muss zunächst eine Funktion definiert werden, welche einer Datenbank-Entität eine entsprechende Realwelt-Entität zuordnet.

Definition 2.3.4 (Realwelt-Mapping). *Sei $\mathfrak{W}$ die Menge aller Realwelt-Entitäten und db eine Datenbank (und somit eine Menge von Datenbank-Entitäten). Dann bildet die Funktion $\omega\colon db \rightarrow \mathfrak{W}$ jede Entität aus db auf eine Entität aus $\mathfrak{W}$ ab.*

Die Definitionen der Begriffe *Duplikatbeziehung* und *Duplikate* werden nun entsprechend auf Basis der Funktion ω vorgenommen.

Definition 2.3.5 (Duplikatbeziehung & Duplikate). *Zwei Datenbank-Entitäten e_r, e_s stehen in einer Duplikatbeziehung, wenn $\omega(e_r) = \omega(e_s)$. In diesem Fall werden sie auch als Duplikate bezeichnet.*

Schließlich kann auf dieser Grundlage der Begriff der *Duplikaterkennung* formal definiert werden.[1]

Definition 2.3.6 (Duplikaterkennung). *Formal ist ein Duplikaterkennungsprozess eine Funktion δ, welche eine Menge von Datenbank-Entitäten $\mathfrak{E}$ auf ein Clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$ abbildet, so dass $\bigcup_{i=1}^{k} C_i = \mathfrak{E}$ (jede Entität ist einem Cluster zugeordnet) und $\forall C_p, C_q \in \mathcal{C}\colon C_p \cap C_q = \emptyset$ (alle Cluster sind disjunkt). Das (Duplikat-)Clustering $\mathcal{C}$ und damit der Duplikaterkennungsprozess δ wird als perfekt bezeichnet, gdw.:*

- *$\forall C \in \mathcal{C}\colon \forall e_r, e_s \in C\colon w(e_r) = w(e_s)$, d. h. alle Datenbank-Entitäten eines Clusters repräsentieren dieselbe Realwelt-Entität (alle erkannten Duplikate sind echte Duplikate und entsprechend ist die Anzahl der false positives[2] null).*
- *$\forall C_p, C_q \in \mathcal{C}\colon \forall e_r \in C_p, \forall e_s \in C_q\colon C_p \neq C_q \Rightarrow w(e_r) \neq w(e_s)$, d. h. alle Datenbank-Entitäten unterschiedlicher Cluster repräsentieren unterschiedliche Realwelt-Entitäten (alle echten Duplikate sind erkannt und entsprechend ist die Anzahl der false negatives[3] null).*

Zum Abschluss folgt an dieser Stelle noch die Definition des Begriffs *Goldstandard*.

Definition 2.3.7 (Goldstandard). *Das perfekte Duplikatclustering $\mathcal{C}_{Gold}$ einer Menge von Datenbank-Entitäten $\mathfrak{E}$ wird als ihr Goldstandard bezeichnet.*

2.3.2. Rolle der Datenqualität

Naumann definiert in [Nau07] den Begriff *Datenqualität* (auch als *Informationsqualität* bezeichnet) ganz allgemein als die Eignung von Daten für die jeweilige datenverarbeitende

[1] In [Pan15] wird zwischen deterministischer und indeterministischer Duplikaterkennung unterschieden. In dieser Arbeit wird allerdings nur die deterministische Duplikaterkennung thematisiert. Daher findet sich die verwendete Definition in der Quelle unter dem Bezeichnung *Deterministic Duplicate Detection*.

[2] Die fälschlicherweise als Duplikate klassifizierten Tupel-Paare werden als *false positives* bezeichnet.

[3] Die echten Duplikate, welche nicht erkannt wurden, werden als *false negatives* bezeichnet.

ID	*name*	*sex*	*street*	*zip*	*city*	*email*
5321	Kai Hildebrandt	m	Hafenstraße 8	20535	Hamburg	⊥
5322	Jule Fuchs	⊥	Domstraße 22	50668	Köln	jule.fuchs@...
5323	Sybille Sauber	f	Mauerstraße 1	10117	Berlin	s_sauber@...
...	...	...	...	...	...	...
5687	Kai Hildebrandt	m	⊥	⊥	Hamburg	6hildebr@...

Tabelle 2.1.: Dieser Datenbestand weist einige Probleme bzgl. der Datenqualität auf: (1.) Die Adresse des Kunden in der ersten Zeile kann nicht korrekt sein, da es in 20535 keine Hafenstraße gibt. (2.) In der zweiten Zeile ist das Geschlecht des Kunden unbekannt. (3.) Der Kunde in der letzten Zeile könnte möglicherweise ein Duplikat des Kunden in der ersten Zeile sein.

Anwendung. Diese Eignung wird auf Basis einer Menge von Qualitätsmerkmalen quantifiziert, welche durch Experten der jeweiligen Anwendungsdomäne definiert werden. Die Probleme die sich aus einer geringen Datenqualität ergeben können sind entsprechend vielfältig und domänenspezifisch.

Beispiel 1. *Für das Customer-Relationship-Management (CRM) eines bestimmten Versandhändlers ist die Korrektheit der Kundenadresse ein wichtiges Qualitätsmerkmal der Kundendaten. Nur durch eine korrekte Adresse ist gewährleistet, dass der Kunde Waren und Rechnungen erhält. Die Folgen solcher Probleme der Datenqualität liegen auf der Hand: Durch unzustellbare Sendungen und Rechnungen entstehen dem Unternehmen echte Kosten (Beispiel: siehe (1.) in Bildunterschrift von Tabelle 2.1).*

Das Vorhandensein einer Angabe über das Geschlecht des Kunden ist hingegen ein untergeordnetes Qualitätsmerkmal. Auch bei fehlender Angabe des Geschlechts können Waren und Rechnungen den Kunden erreichen. Gänzlich irrelevant ist sie allerdings nicht, da beispielsweise einer Sendung keine geschlechtsspezifische Werbung beigelegt werden kann (Beispiel: siehe (2.) in Bildunterschrift von Tabelle 2.1).

Für die Duplikaterkennung spielt Datenqualität eine besondere Rolle. Sie ist nämlich ihre größte Motivation und Herausforderung zugleich. In dem für dieses Forschungsgebiet wegweisenden Konferenzbeitrag *"Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem"* [HS98] stellen Hernandez und Stolfo schon im Titel klar, dass Datenbestände in der Regel "verschmutzt" sind. Im Sinne der Datenqualität kann das bedeuten, dass Daten inkonsistent, fehlerhaft oder lückenhaft sind [Chr12]. Diese Aspekte führen in der Regel auch zur Existenz von Duplikaten, welche wiederum selbst eine "Verschmutzung" darstellen (Beispiel: siehe (3.) in Bildunterschrift von Tabelle 2.1). In [NH10] unterscheiden Naumann und Herschel zwei Hauptursachen für Duplikate:

1. **intra-source duplicates:** Auf der einen Seite können fehlerhafte Eingaben innerhalb einer Datenbasis zu Duplikaten führen. Ursächlich können z. B. Tippfehler, mehrfache Eingaben, unterschiedliche Eingabe-Konventionen oder signifikante Verände-

rungen des betroffenen Realweltobjektes zwischen zwei Eingaben sein.

2. **inter-source duplicates:** Auf der anderen Seite ist die Durchführung einer Integration mehrerer Datenquellen eine häufige Ursache für Duplikate. Hierbei können z. B. unterschiedliche Daten-Schemata, Konventionen oder Eingabe-Voraussetzungen zu abweichenden Repräsentationen eines Realweltobjektes führen.

Data Cleansing (auch *Data Cleaning*) bezeichnet eine Reihe von Maßnahmen, die zur Verbesserung der Datenqualität von Datenbeständen durchgeführt werden. Die Duplikaterkennung ist ein integraler Bestandteil dessen [NH10].

Wie es zu Einbußen bei der Datenqualität durch Datenfehler kommen kann, kann beispielsweise durch ein Fehlermodell beschrieben werden.

Definition 2.3.8 (Fehlermodell)**.** *Ein Fehlermodell ist ein Modell für die Entstehung und die Art von Datenfehlern in einem mehr oder weniger spezifizierten Anwendungskontext. Eine Art von Fehlern kann auch als Fehlertyp bezeichnet werden.*

Zur Veranschaulichung wird im Folgenden ein Fehlermodell vorgestellt, welches versucht die Entstehung verschiedener Fehlertypen bei der Dateneingabe zu erklären und abzubilden.

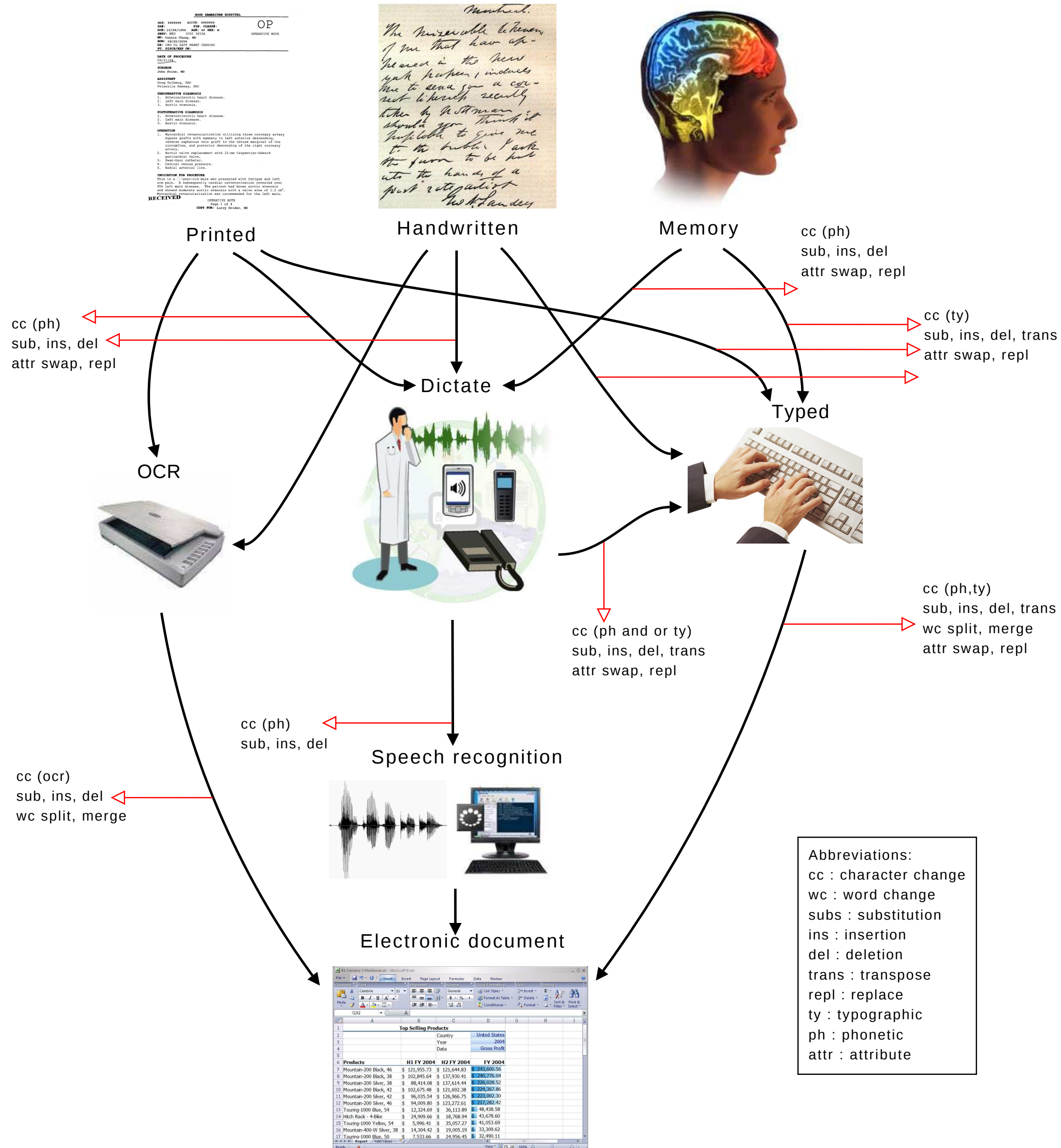


Abbildung 2.1.: Christens' Modell für die Entstehung von Fehlern während der Dateneingabe (entnommen aus [CP09])

Fehlermodell nach Christen

In [CP09] stellen Pudjinjono und Christen ein Modell vor, welches die Entstehung von möglichen Fehlern während der Dateneingabe abbildet (siehe Abbildung 2.1). Darin sind ausgehend von verschiedenen analogen Datenquellen, verschiedene Eingabe-Kanäle modelliert, welche in einer digitalen Repräsentation münden.

Bei den analogen Datenquellen kann es sich um gedruckte Texte, handgeschriebene Texte oder Erinnerungen handeln. Als Kanäle sind manuelle Eingaben (über eine Tastatur), optische Eingaben (über Scannen und OCR[4]), sowie auditive Übertragungen (Dik-

[4]OCR (Optical Character Recognition) steht für Texterkennung

tieren von Informationen) und auditive Eingaben (Spracherkennung) vorgesehen. Dabei hat jeder Kanal seine eigene Fehler-Charakteristik.

Ausgehend von diesen Kanälen werden die drei folgenden Haupt-Eingabe-Fehlertypen für die Dateneingabe unterschieden, welche jeweils bestimmte Charakteristika aufweisen:

- **Typographische Fehler** entstehen beim Tippen und beinhalten auf Zeichenebene das Einfügen (*insertion*), Auslassen (*deletion*), Ersetzen (*substitution*) und Vertauschen (*transposition*) von Zeichen. Darüber hinaus können ganze Attributwerte vertauscht (*attr swap*) oder ausgelassen (*attr repl*) werden.
  Beispiel.

  eigentlich ⟶ eingentlich (insertion)
  eigentlich ⟶ eigenlich (deletion)
  eigentlich ⟶ eigemtlich (substitution)
  eigentlich ⟶ eigenltich (transposition)

- **OCR-Fehler** entstehen bei der optischen Texterkennung, welche auf der Ähnlichkeit der Formen von Zeichen basieren. Sie beinhalten das Einfügen (*insertion*), Auslassen (*deletion*) und Austauschen (*substitution*) von Zeichen. Durch das Einfügen oder Auslassen von Leerzeichen können sie auch zu einer Wort-Trennung (*word split*) oder Wort-Zusammenführung (*word merge*) führen.
  Beispiel.

  Hildebrandt ⟶ Hi1debrandt (substitution)
  dein ⟶ dem (substitution + deletion)
  Ich bin ⟶ Ichbin (word merge)

- **Phonetische Fehler** entstehen bei der Sprachübertragung und bei der automatisierten Spracherkennung durch akustische Ähnlichkeiten. Diese Fehler können zum Austauschen (*substitution*), Einfügen (*insertion*) oder Auslassen (*deletion*) einzelner Zeichen, Silben oder ganzer Wörter führen. Durch Sprachübertragung kann es auch vorkommen, dass ganze Attributwerte vertauscht (*attr swap*) oder verändert (*attr replace*) werden können.
  Beispiel.

  Schmitt ⟶ Schmidt (char substitution)
  Mainz ⟶ meins (word substitution)

Abhängig von den einbezogenen Kanälen, können diese Fehlertypen (und damit ihre Charakteristika) auch beliebig miteinander kombiniert auftreten.

2.3.3. Der Prozess der Duplikaterkennung

Die Aufgabe der Duplikaterkennung wird üblicherweise prozessorientiert gelöst. Ein *Duplikaterkennungsprozess* besteht aus mehreren Subprozessen und kann sehr unterschied-

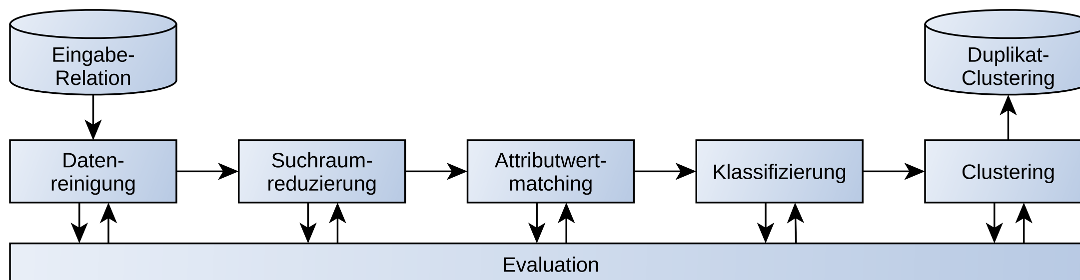


Abbildung 2.2.: Ein typischer Duplikaterkennungsprozess nach Christen

lich ausgestaltet sein. In der Forschung wurden in den vergangenen Jahren mehrere typische Subprozesse heraus gearbeitet [NH10, Chr12]. Abbildung 2.2 zeigt einen typischen Duplikaterkennungsprozess nach Christen, welcher aus den folgenden Phasen (bzw. Subprozessen) besteht:

1. **Datenreinigung** (auch *data pre-processing* oder *data preparation* genannt):
   Die Datenreinigung dient zur Aufbereitung und Vereinheitlichung der Datenbasis. Durch syntaktische und semantische Anpassungen kann die Datenqualität erhöht werden, was wiederum zur Verbesserung der Qualität einer Duplikaterkennung führen kann [HSW07]. Mögliche Maßnahmen beinhalten das Entfernen ungewollter Zeichen, das Vereinheitlichen von Abkürzungen oder auch das Hinzufügen abgeleiteter Attribute.

2. **Suchraumreduzierung** (auch *indexing* oder *blocking* genannt):
   Die meisten Verfahren der Duplikaterkennung basieren auf paarweisen Vergleichen der Tupel. Daher ist der Suchraum für Duplikate und damit die Komplexität einer Duplikaterkennung prinzipiell quadratisch zur Tupel-Menge. Insbesondere bei größeren Tupel-Mengen sind daher Methoden zur Suchraumreduzierung essentiell für eine hinreichende Effizienz eines Duplikaterkennungsprozesses. Bei solchen Methoden wird versucht, anhand bestimmter Kriterien Vergleiche zwischen Tupeln zu vermeiden, welche mit sehr hoher Wahrscheinlichkeit keine Duplikate sind. Bei der Auswahl der hierfür verwendeten Kriterien herrscht immer ein Trade-Off zwischen *Reduction Ratio*[5] und *Pairs Completeness*[6] [GH07, S. 143]. Die bekanntesten Verfahren zur Suchraumreduzierung sind das *Standard Blocking* [Jar89, BCC03] und die *Sorted Neighborhood Methode (SNM)* [HS95, HS98].

3. **Attributwertmatching** (auch *record pair comparison* genannt):
   Beim Attributwertmatching werden für jedes Tupel-Paar des (reduzierten) Suchraumes anhand geeigneter Maße Werte berechnet, welche den Grad der Ähnlich-

[5] Die Reduction Ratio misst die relative Reduktion des Suchraumes, also wie stark der Suchraum verkleinert wurde.
[6] Die Pairs Completeness beschreibt in diesem Fall den Anteil der tatsächlichen Duplikat-Paare, welche im reduzierten Suchraum noch vorhanden sind und durch die Reduzierung nicht aussortiert wurden.

keit der Attributwerte der beiden Tupel abbilden sollen. Dabei können verschiedene Ähnlichkeitsmaße zum Einsatz kommen, wie z. B. Edit-basierte (Levenshtein-Distanz [Lev66]), Sequenz-basierte (Jaro-Distanz [Jar89]) oder Token-basierte Stringmetriken (Jaccard-Koeffizient [ME+96], N-Gramme [CC04]).

4. **Klassifizierung** (auch *tuple matching* oder *decision model* genannt):
   Bei der Klassifizierung werden die verglichenen Tupel-Paare anhand ihrer Ähnlichkeitswerte in *Matches*, *Unmatches* und ggf. *Possible Matches* (d. h.„Duplikate“, „Nicht-Duplikate“ und ggf. „mögliche Duplikate“) unterteilt. Dafür können unter anderem probabilistische Verfahren, distanzbasierte Verfahren, regelbasierte Verfahren und überwachte sowie unüberwachte maschinelle Lernverfahren verwendet werden [Chr12]. In der Regel wird die Klassifizierungsentscheidung für ein Paar unabhängig von den anderen Tupel-Paaren getroffen. Dies führt dazu, dass die Ergebnisse widersprüchlich sein können. So kann es durchaus vorkommen, dass die Paare $\langle t_1, t_2 \rangle$ und $\langle t_1, t_3 \rangle$ als Match, aber das Paar $\langle t_2, t_3 \rangle$ als Unmatch klassifiziert sind, womit die Transitivität der Duplikatbeziehung verletzt ist.

5. **Clustering**:
   Nach Abschluss der Klassifizierung liegt eine (ggf. inkonsistente) Einteilung der Tupelpaare in Matches, Unmatches und Possible Matches vor. In Definition 2.3.6 ist die Duplikaterkennung allerdings als Abbildung der Eingabe-Relation auf ein Clustering definiert. Um ein solches global konsistentes Ergebnis zu erhalten, werden in der *Clustering*-Phase die Tupel auf Basis der (unabhängigen) paarweisen Klassifizierungsentscheidungen kollektiv klassifiziert. Das bedeutet, dass i.d.R. Informationen über die Duplikatbeziehungen der verschiedenen Tupel zueinander berücksichtigt werden, um eine Partitionierung der Eingabe-Relation in disjunkte Duplikatcluster zu erhalten. Beispiele für geeignete Clustering-Verfahren sind z. B. *Partitionierung anhand von Komponenten* [HS95] oder *Partitionierung anhand von Centern* [HM09].

6. **Evalutation** (auch *verifikation* genannt):
   Zur Evaluierung der Qualität eines Duplikaterkennungsprozesses gibt es verschiedene Messgrößen. Dabei ist zu beachten, dass ein Duplikatclustering immer auch als eine Menge paarweiser Klassifizierungsentscheidungen ausgedrückt werden kann. Ein verbreiteter Ansatz ist es, die Qualität des Prozesses durch die Güte dieser paarweisen Klassifizierungsentscheidungen auszudrücken. Hierbei werden die Zahlen der korrekt klassifizierten Duplikate (*True-Positives*), der fälschlicherweise als Duplikate klassifizierten Tupel-Paare (*False-Positives*) und der nicht gefundenen Duplikate (*False-Negatives*) in Beziehung gesetzt. Die gängigsten dieser Maße sind *Precision*, *Recall* und *F-Measure* [GH07, S. 140 ff.]. Ein anderer Ansatz misst die Ähnlichkeit des Ergebnis-Clusterings mit dem Goldstandard-Clustering. Zu diesen Messgrößen gehören z. B. der *Rand-Index* und der *Talburt-Wang-Index* [Tal11].

ID	artist	track	CID
1	AC/DC	Hells Bells	1
2	Doors	Touch Me	2
3	Beethoven	Symphony No. 4	3
4	Beethoven	Symphony No. 5	4
5	Beethoven	Schicksalssinfonie	4
6	ACDC	Hells Bells	1
7	Dark Age	Hells Bells	5
8	The Doors	Touch Me	2
9	van Beethoven	5th Symphony	4

(1 a)

ID	CID
1	1
2	2
3	3
4	4
5	4
6	1
7	5
8	2
9	4

(1 b)

ID 1	ID 2
1	6
2	8
4	5
4	9
5	9

(2)

Abbildung 2.3.: (1 a) und (1 b) zeigen clusterbasierte Repräsentationen des Goldstandards. In (2) ist eine paarbasierte Repräsentation zu sehen.

2.3.4. Test-Daten für die Duplikaterkennung

Zur Entwicklung, Erprobung und Evaluation von Methoden der Duplikaterkennung werden Test-Daten benötigt. Hierfür haben sich Test-Datasets etabliert, welche aus einer Datenbank-Relation (d. h. einer Menge von Datenbank-Entitäten) und ihrem Goldstandard bestehen. Auf Basis des formalen Modells aus Abschnitt 2.3.1 lässt sich dies wie folgt definieren:

Definition 2.3.9 (Test-Dataset (Version 1))**.** *Sei $\mathfrak{E}$ eine Menge von Datenbank-Entitäten und $\mathcal{C}_{Gold}$ ihr Goldstandard. Dann wird $\mathcal{T} = (\mathfrak{E}, \mathcal{C}_{Gold})$ als Test-Dataset bezeichnet.*

Üblicherweise liegen Test-Datasets in tabellarischer Struktur als CSV-Dateien oder relationale Datenbanktabellen vor. Für den Goldstandard haben sich hauptsächlich zwei Repräsentationsformen etabliert (siehe auch Abbildung 2.3):

1. **Clusterbasiert:** Die Duplikatbeziehungen sind über die Zugehörigkeit der Tupel zu bestimmten Duplikatclustern definiert. Dafür wird jedem Tupel eine Cluster-ID zugeordnet. Diese Zuordnung kann (a) über ein zusätzliches Attribut in derselben Relation oder (b) in einer separaten Relation vorliegen (sofern die Haupt-Relation über einen Primärschlüssel verfügt).

2. **Paarbasiert:** Die Duplikatbeziehungen sind explizit über alle Duplikatpaare definiert, welche in einer zusätzlichen Relation gehalten werden. Für diese Variante wird ein Primärschlüssel in der Haupt-Relation vorausgesetzt. Ein Nachteil dieser Variante ist der quadratische (und damit potentiell erhöhte) Speicherbedarf.

Darüber hinaus sollte die Charakteristik der Test-Daten der Charakteristik der tatsächlich im Einsatz zu verarbeitenden Daten so ähnlich wie möglich sein. Dazu gehören unter anderem Fehlertypen, Fehlermuster, Verteilungen und auch Abhängigkeiten

zwischen Attributen, die möglichst realistisch sein sollten. Der Einsatz unrealistischer Test-Datasets kann nämlich dazu führen, dass getätigte Evaluierungen ihre Aussagekraft verlieren (Garbage-In/Garbage-Out-Prinzip). Idealerweise sollten echte Daten aus dem (geplanten) Einsatzkontext als Test-Daten verwendet werden. Das ist in der Realität allerdings aus verschiedenen Gründen häufig nicht ohne weiteres möglich:

- Es existieren keine echten Daten aus dem Einsatz-Kontext.
- Es existiert keine Zugriffsmöglichkeit oder -berechtigung zu den vorhandenen Daten.
- Aus (Datenschutz-)rechtlichen Gründen dürfen die vorhandenen Daten nicht verwendet werden.
- Zu den vorliegenden Daten existiert kein Goldstandard.

Vor dem Hintergrund dieser Probleme haben sich in der Forschung verschiedene Ansätze zur Beschaffung von Test-Daten etabliert [Chr12, S. 163ff].

(Semi-)Manuelle Klassifizierung

Die manuelle Klassifizierung von Tupel-Paaren eines vorhandenen Datasets in Duplikate und Nicht-Duplikate ist der naive Ansatz zum Erzeugen von Test-Datasets. Dabei werden Menschen (mit Domänenwissen) Tupel-Paare vorgelegt und diese entscheiden, ob es sich dabei um Duplikate handelt oder nicht. Der Aufwand für jede einzelne Klassifizierung ist also sehr hoch und zudem ist die Anzahl der zu vergleichenden Tupel-Paare prinzipiell quadratisch zur Anzahl der Tupel. (Im Grunde handelt es sich bei diesem Vorgehen um eine manuelle Duplikaterkennung.)

Aus diesem Grund wird diese Methode in der Regel nur in Verbindung mit Techniken zur Reduzierung der Tupel-Vergleiche eingesetzt. Dazu gehören typische Methoden der Duplikaterkennung, Sampling von Tupel-Paaren oder Active-Learning-Ansätze [SB02, TKM02, AGK10]. Es wird also praktisch eine semi-automatische Duplikaterkennung durchgeführt, in der Unsicherheiten gekennzeichnet und in einem *Clerical Review* [Chr12, S. 174] durch einen Menschen aufgelöst werden. Der Vorteil dieses Ansatzes ist, dass ihn auf jedes beliebige Dataset anwenden kann. Das Hauptproblem ist der Trade-Off zwischen dem Aufwand und Ungenauigkeit (durch die Vorverarbeitung) und damit die Tatsache, dass der Ansatz für hinreichend große Datasets, mit einer zufriedenstellende Genauigkeit, nicht praktikabel ist.

Synthetisierte Test-Datasets

Bei der Synthetisierung wird versucht künstlich möglichst realistische Test-Daten zu generieren. Zur Erzeugung der künstlichen Attributwerte werden in der Regel Lookup-Tables und/oder spezielle Regeln verwendet. Zusätzlich werden die erzeugten Werte

anhand bestimmter Fehlermodelle (siehe Definition 2.3.8) absichtlich verfälscht. Dieser Erzeugungsprozess sollte idealerweise unter Berücksichtigung realistischer Verteilungen und Abhängigkeiten für Attributwerte und Fehler geschehen. Dieser Ansatz erlaubt eine hohe Anpassbarkeit bei der Ausprägung der Daten, allerdings kann eine Implementierung für hinreichend realistische Daten extrem aufwendig sein. Einen solchen Ansatz verfolgen zum Beispiel die Tools DBGen [HS98], Febrl data set generator [CP09] und GeCo [CV13].

Abgeleitete Test-Datasets

Um die Vorteile der ersten beiden Ansätze zu vereinen, können die Test-Daten auch aus vorhandenen Daten abgeleitet werden. Hierfür wird ein echtes (möglichst fehlerfreies) Dataset als Basis genommen und gezielt mit Duplikaten sowie möglichst realistischen Fehlern verunreinigt. Die Datenqualität des Datasets wird also absichtlich verringert (vgl. Abschnitt 2.3.2). Dieser Ansatz ermöglicht die Verwendung geeigneter Test-Daten aus der gewünschten Domäne mit realistischen Verteilungen und Abhängigkeiten bei überschaubarem Implementations-Aufwand. Nach diesem Prinzip arbeiten die Tools TDGen [BR12] und ProbGee [FW12].

Öffentliche Test-Datasets

Über die Jahre hat die Forschungscommunity verschiedene Test-Datasets erarbeitet und veröffentlicht. Diese wurden in der Regel aufwendig manuell klassifiziert oder synthetisch erzeugt. Solche Datasets können zwar direkt und ohne großen Aufwand verwendet werden, allerdings führen die festen Schemata, Duplikate und Größen naturgemäß zu geringer Flexibilität und Anpassbarkeit. Hierdurch lassen sich Methoden der Duplikaterkennung nur in begrenztem Rahmen evaluieren. Beispiele für solche Datasets sind *Census* (450 Tupel), *Restaurant* (864 Tupel), *Cora* (ca. 1.300 Tupel), *DBLP-Scholar* (ca. 64.000 Tupel), welche z. B. im RIDDLE-Repository [Bil03] zu finden sind.

2.4. Apache Spark

In diesem Abschnitt wird auf Grundlage der ersten wissenschaftlichen Publikationen von Spark-Erfinder Matei Zaharia zum Thema [ZCF+10, ZCD+12] und zweier Fachbücher [HKZ+15, RLOW15] das Cluster-Computing-Framework Apache Spark vorgestellt. Im Rahmen dieser Masterarbeit wurde das Framework für die Implementation des in Kapitel 4 vorgestellten Verfahrens eingesetzt.

Im Folgenden wird zunächst Apache Spark im Kontext von Cluster Computing und Big Data eingeordnet (Abschnitt 2.4.1) und allgemein eingeführt (Abschnitt 2.4.2). Anschließend werden die Architektur (Abschnitt 2.4.3) und die wichtigsten Konzepte erläutert (Abschnitte 2.4.5 bis 2.4.7).

2.4.1. Cluster Computing

Seit einigen Jahren werden in Forschung und Wirtschaft immer größere und komplexere Datenmengen erzeugt, gespeichert, verändert und analysiert. Dieses Phänomen wird allgemein unter dem Begriff *Big Data* zusammengefasst.

Innovationen vom MPI-Standard[7] über NoSQL-Datenbanksysteme (z. B. HBase[8] oder Cassandra[9]) bis hin zum Programmiermodell MapReduce [DG08] machten es Entwicklern immer einfacher verteilte Computersysteme (z. B. Computercluster oder dedizierte Parallelrechner) – durch zunehmende Abstraktion von komplexen Details der Verteilung – für parallele Berechnungen zu nutzen. Erst dadurch wurde das Verarbeiten derart großer Datenmengen ermöglicht. Das Feld von Big Data wird daher einerseits durch die Fähigkeiten solcher Technologien definiert, andererseits wird ihre Entwicklung auch maßgeblich durch Big Data motiviert [RLOW15].

Cluster-Computing-Frameworks fassen verschiedene solcher Technologien und Methoden zu einer harmonischen Schnittstelle zusammen und erlauben dadurch auf bequeme Art und Weise, effiziente, skalierbare und verteilte Berechnungen auf sehr großen Datenmengen zu entwickeln und durchzuführen. Zu den bekannteren Vertretern gehören Apache Hadoop [10], Apache Flink [11] und Apache Spark [12]. Letzteres wird im Folgenden vorgestellt.

2.4.2. Allgemeine Informationen

Apache Spark ist ein relatives junges Cluster-Computing-Framework, welches seit 2009 an der University of Berkley entwickelt wird und im Mai 2014 in der Version 1.0 erschienen ist. Es wurde für die Verarbeitung großer Datenmengen (*large-scale data processing*) konzipiert und hat seinen Fokus auf Effizienz und Fehlertoleranz bei der Ausführung von *iterativen Algorithmen* und *interaktivem Data-Mining*.

Um die beiden Eigenschaften Effizienz und Fehlertoleranz zu ermöglichen, wurde als Kernkonzept die hochperformante in-memory Datenstruktur *Resilient Distributed Datasets (RDDs)* entwickelt. Ein Sortiment paralleler High-Level-Operationen auf dieser Datenstruktur sowie Konstrukte wie *Shared Variables* sind über APIs mit verschiedenen Programmiersprachen (Scala, Java, Python und R) benutzbar.

Diese Konzepte sollen dazu beitragen das Programmieren verteilter Anwendungen möglichst bequem zu gestalten, so dass es sich so ähnlich wie möglich wie das Schreiben herkömmlicher, nicht-verteilter Anwendungen anfühlt.

[7] Message Passing Interface (MPI): `http://www.mcs.anl.gov/research/projects/mpi/`
[8] HBase: `http://hbase.apache.org/`
[9] Cassandra: `http://cassandra.apache.org/`
[10] Apache Hadoop: `http://hadoop.apache.org/`
[11] Apache Flink: `http://flink.apache.org/`
[12] Apache Spark: `http://spark.apache.org/`

2.4.3. Sofware-Architektur

Softwaretechnisch basiert Spark auf einer Master-Slave-Architektur. Der Master wird als *Driver* bezeichnet und die n (mit $n \geq 1$) Slaves als *Executor*. Sowohl der Driver als auch jeder Executor stellen jeweils einen eigenen Java-Prozess dar. Zusammen bilden sie eine *Spark-Applikation* (siehe Abbildung 2.4). Um eine Anwendung für Spark zu programmieren schreibt der Benutzer ein sogenanntes *Driver-Programm*, welches die Programmlogik enthält, also hauptsächlich Definitionen von RDDs und Operationen, die auf diesen RDDs ausgeführt werden.

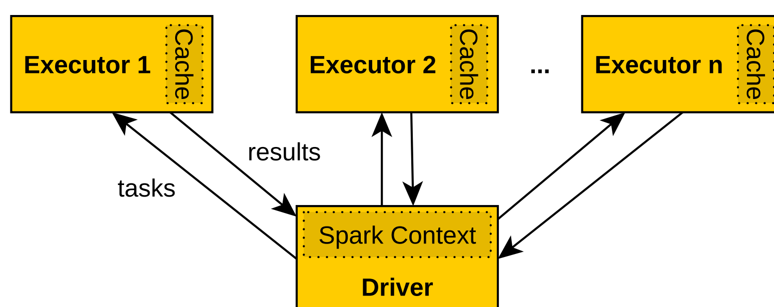


Abbildung 2.4.: Eine Spark-Applikation umfasst den Driver-Prozess und alle Executor-Prozesse

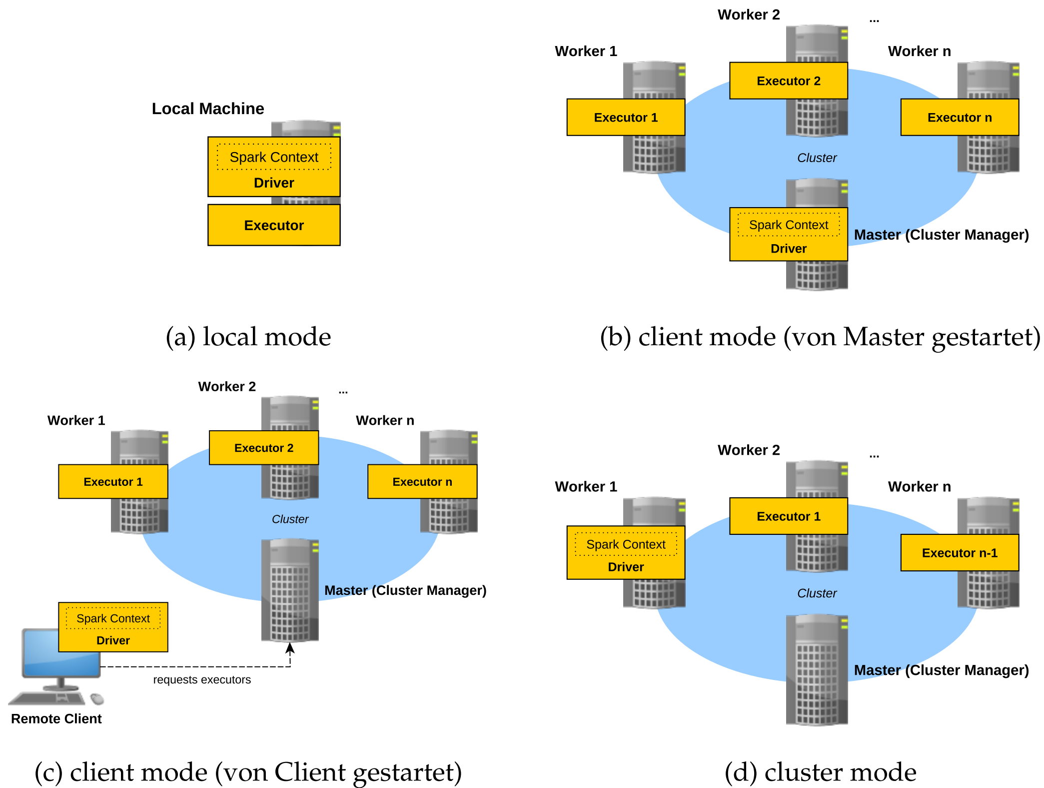
Driver

Der Driver als Haupt-Prozess führt den Code des Driver-Programms aus. Er erzeugt zu Beginn ein sogenanntes `SparkContext`-Objekt, welches eine Verbindung zum Computercluster repräsentiert und über das überhaupt erst auf Spark und seine Operationen zugegriffen werden kann.

Das Driver-Programm bildet implizit immer einen gerichteten azyklischen Graphen (DAG) aus Operationen. Dieser logische Graph wird bei der Verarbeitung durch den Driver in einen *Ausführungsplan* (engl. execution plan) überführt, welcher in mehrere *Stages* unterteilt ist, welche wiederum aus mehreren *Tasks* bestehen. Ein Task ist für Spark die kleinste ausführbare Einheit.[13] Beim Erzeugen des Ausführungsplans werden verschiedene Optimierungen durchgeführt, wie z. B. das *Pipelining* (siehe Abschnitt 2.4.7).

Schließlich hat der Driver die Rolle eines zentralen Koordinators. Er ist für das Scheduling und das Versenden der Tasks an die entsprechenden Executor zuständig. Grundprinzip beim Scheduling ist es, die Datenlokalität zu berücksichtigen. Dies bedeutet, dass versucht wird Berechnungen immer möglichst auf jenen Knoten durchzuführen, auf denen die entsprechenden Daten vorliegen. Darüber hinaus hält der Driver die zentralen Informationen über das Spark-Cluster und stellt diese, zusammen mit den laufenden und gelaufenen Spark-Applikationen, über ein Web-Interface bereit.

[13] Eine normale Spark-Applikation kann aus hunderten oder tausenden Tasks bestehen.



(a) local mode

(b) client mode (von Master gestartet)

(c) client mode (von Client gestartet)

(d) cluster mode

Abbildung 2.5.: Die verschiedenen Deployment-Modi ermöglichen Flexibilität beim Ausführen von Spark-Applikationen

Executor

Die Executor werden zusammen mit der Spark-Applikation gestartet und laufen bis diese terminiert. Dabei fallen ihnen zwei Aufgaben zu: Zum einen führen sie die Tasks aus, die ihnen vom Driver zugewiesen werden. Zum anderen stellen sie ihren Arbeitsspeicher als in-memory-Speicher für RDDs zur Verfügung. Beim Ausfall eines Executors kann die Applikation mit den verbliebenen Executorn weiterlaufen.

2.4.4. Cluster-Architektur & Deployment

Eine Spark-Applikation kann im *local mode* auf einem einzelnen Rechner ausgeführt werden oder im *distributed mode* auf einem Computercluster, welches aus einem Master- und mehreren Worker-Knoten besteht.

Im *local mode* werden der Driver und ein Executor nebeneinander auf ein und demselben Rechner gestartet (siehe Abbildung 2.5a). Im *distributed mode* wird eine Spark-Applikation über den Cluster-Manager (welcher sich i.d.R. auf dem Master-Knoten befindet) eines vorhandenen Computerclusters gestartet. Neben dem mitgelieferten Standalone-Cluster-Manager unterstützt Spark auch *Apache Hadoop YARN*[14] und *Apache Me-*

[14] Apache Hadoop YARN: `https://hadoop.apache.org/docs/current/hadoop-yarn/`

sos[15].

Darüber hinaus unterscheidet Spark bei der Ausführung auf einem Computercluster zwischen zwei Deployment-Modi: Im *client mode* wird der Driver auf dem Rechner gestartet, welcher den Job absendet. Das kann beispielsweise der Master-Knoten des Computerclusters sein (siehe Abbildung 2.5b) oder auch ein Rechner außerhalb des Clusters (siehe Abbildung 2.5c). Im *cluster mode* wird der Driver auf einem der Worker-Knoten gestartet, wodurch allerdings die Anzahl der für die Executor-Rolle zur Verfügung stehenden Worker-Knoten um einen reduziert wird (siehe Abbildung 2.5d).

2.4.5. Resilient Distributed Datasets

Ein Resilient Distributed Dataset (RDD) ist eine *partitionierte*, *parallele* Collection von Objekten, welche *verteilt* über Cluster-Knoten vorliegen kann. Im Grunde handelt es sich bei dieser Datenstruktur um eine Abstraktion von verteiltem Arbeitsspeicher für die effizientere Wiederverwendbarkeit von Zwischenergebnissen. Eine wesentliche Eigenschaft ist nämlich, dass RDDs so weit wie möglich im Arbeitsspeicher gehalten werden (*in-memory*).

Logisch betrachtet handelt es sich um eine *read-only* Datenstruktur, welche nur auf folgende Weisen mit bestimmten deterministische Operationen erzeugt werden kann:

- Durch Lesen von einem Sekundärspeicher (mit `textFile(path)`),
- durch Parallelisieren einer herkömmlichen Collection (mit `parallelize(data)`),
- oder durch Anwendung einer Transformation (siehe Abschnitt 2.4.6) auf ein bestehendes RDD.

Darüber hinaus ist die Berechnung von RDDs *lazy*. Das bedeutet, dass ihre Elemente nicht immer im physischen Speicher vorliegen, sondern erst dann materialisiert werden, wenn sie benötigt werden. Um das zu ermöglichen, enthält jedes RDD Informationen darüber, wie es abgeleitet wurde – seine sogenannte *Lineage*. Dabei handelt es sich um eine Art grob-granulares Transformations-Log, welches als DAG dargestellt wird (siehe Abbildung 2.6).

Durch das Transformations-Logging trägt die Lineage zusätzlich wesentlich zur *Fehlertoleranz* von RDDs bei: Geht eine Partition z. B. durch den Ausfall eines Worker-Knotens verloren, kann sie anhand der Lineage auf einem anderen Knoten rekonstruiert werden. Das funktioniert natürlich nur sofern die benötigten Quelldaten nicht ebenfalls vom Ausfall betroffen sind. Durch *Checkpointing* bei längeren Lineage-Graphen wird die Wahrscheinlichkeit für diesen Fall gering gehalten und der Aufwand für eine mögliche Recovery reduziert.

[15] Apache Mesos: `http://mesos.apache.org/`

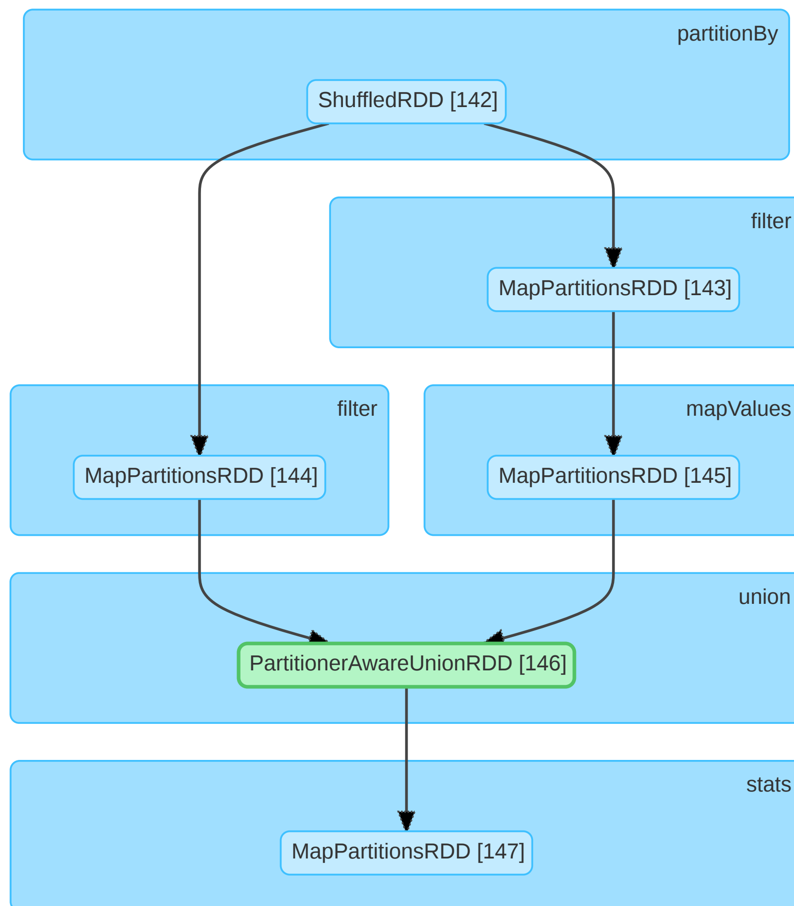


Abbildung 2.6.: Diese Visualisierung eines Teil-Graphen einer echten Lineage ist Sparks Web-Interface entnommen. Sie zeigt wie RDDs durch verschiedene Operationen (z. B. `partitionBy` oder `filter`) in weitere RDDs überführt werden. Grün hervorgehobene RDDs werden explizit persistiert. In den eckigen Klammern sind die IDs der RDDs festgehalten.

Persistenz

Wie oben erwähnt, versucht Spark RDDs so weit es geht im Arbeitsspeicher zu halten. Sollte der Arbeitsspeicher nicht für alle RDDs ausreichen, entscheidet Spark nach der Least-recently-used-Strategie (LRU) welche verdrängt werden. Was mit verdrängten RDDs passiert, wird durch ihre jeweilige Persistierungs-Strategie (*StorageLevel*) bestimmt. Standardmäßig müssen verdrängte RDDs bei einem erneuten Zugriff neu berechnet werden (*MEMORY_ONLY*). Eine andere Strategie besteht darin, verdrängte RDDs auf einen Sekundärspeicher auszulagern (*MEMORY_AND_DISK*).

Zum Zwecke von Laufzeit- und Speicheroptimierungen hat der Entwickler die Möglichkeit auf die Persistenz-Strategien Einfluss zu nehmen. Für jedes RDD kann ein eigenes explizites StorageLevel festgelegt werden. Für die Wahl der Strategie sollten Faktoren in Beziehung gesetzt werden wie z. B. der Aufwand für eine erneute Berechnung und der

Aufwand für Lese- und Schreiboperationen auf dem Sekundärspeicher.[16] Auch Hardware-Aspekte wie Größe und Geschwindigkeit des zur Verfügung stehenden Primär- und Sekundärspeicher können eine entscheidende Rolle spielen.

Für den programmiertechnischen Zugriff auf ein RDD ist sein StorageLevel transparent. Durch ein verändertes StorageLevel ändert sich bei einem erneuten Zugriff also lediglich die Performanz.

Partitionierung

Die Partitionierung von RDDs geschieht auf Basis von *Partitioner*-Klassen. Von Haus aus gibt es einen *Hash*- und einen *Range*-Partitioner, zudem können Entwickler eigene Partitioner implementieren. Um die Datenlokalität für Berechnungen zu optimieren, können einigen Operationen solche Partitioner als Parameter explizit mit übergeben werden, so dass die resultierenden RDDs entsprechend repartitioniert werden.

2.4.6. Operationen

Wie bereits erwähnt, enthält das Driver-Programm die Programmlogik, also vor allem die RDD-Definitionen und die Operationen, die auf den RDDs ausgeführt werden sollen. Bei den Operationen wird zwischen *Transformations* und *Actions* unterschieden.

Transformations

Transformations zeichnen sich dadurch aus, dass sie RDDs erzeugen. Dies kann durch Lesen von einem Sekundärspeicher geschehen, durch Parallelisieren einer herkömmlichen Collection oder durch Transformieren eines bestehenden RDDs in ein neues. Zu den wichtigsten Transformations gehören:

- `map(func)`: Erzeugt ein RDD, dessen Objekte jeweils durch Anwendung der Funktion `func` auf die einzelnen Objekte des Eingabe-RDDs entstanden sind.
- `flatMap(func)`: Ähnlich wie `map`, nur dass jedes Objekt des Eingabe-RDDs auf eine Menge von 0 bis *n* Objekten abgebildet wird, statt auf ein einzelnes.
- `filter(func)`: Erzeugt ein RDD, das nur die Objekte des Eingabe-RDDs enthält, welche die boolesche Funktion `func` erfüllen.
- `union(otherRDD)`: Erzeugt ein RDD, das die Vereinigung des Eingabe-RDDs und des übergebenen RDDs `otherRDD` darstellt.
- `join(otherRDD)`: Erzeugt aus zwei RDDs mit den Typen `(K,V)` und `(K,W)` ein RDD vom Typ `(K,(V,W))` mit allen Paaren `(V,W)` für jeden Schlüssel `K`.

[16] Werden auf einem RDD sehr aufwendige Berechnungen durchgeführt, kann es sich lohnen teure Disk-I/O in Kauf zu nehmen, um eine noch teurere Neuberechnung zu vermeiden.

- `groupByKey()`: Erzeugt aus einem RDD mit dem Typ `(K,V)` ein RDD vom Typ `(K,Iterable<V>)`.
- `repartition(numPartitions)`: Verteilt die Daten des RDDs zufällig neu, um die Verteilung der Daten zu balancieren oder die Anzahl der Partitionen zu verändern.

Actions

Actions hingegen verwenden RDDs, um auf ihrer Basis Werte zu berechnen, die zurückgegeben oder auf einen Sekundärspeicher geschrieben werden. Das bedeutet auch, dass erst die Ausführung einer Action auf einer RDD dazu führt, dass sie berechnet wird[17]. Zu den wichtigsten Actions gehören:

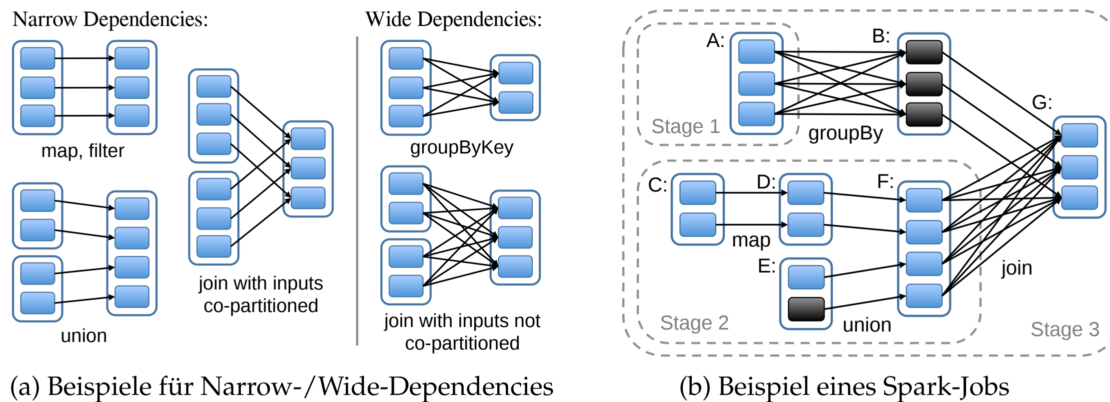
- `reduce(func)`: Aggregiert die Objekte eines RDDs, indem die Funktion `func` immer zwei Objekte in ein einzelnes Objekt desselben Typs überführt (z. B. Addition). Die Funktion muss kommutativ und assoziativ sein, damit die Operation auch nebenläufig korrekt ausgeführt werden kann.
- `collect()`: Überführt das (verteilte) RDD in ein klassisches Array von Objekten im Driver-Programm.
- `count()`: Zählt die Objekte eines RDDs.
- `countByValue()`: Gibt die Anzahl `count` jedes individuellen Wertes `value` in diesem RDD als Mapping `(value, count)` zurück.
- `stats()`: Berechnet die statistischen Werte Durchschnitt, Standardabweichung, Varianz, Minimum, Maximum, Summe und Anzahl und gibt diese in einem `StatCounter`-Objekt zurück. Kann nur auf RDDs mit Objekten vom Typ Double ausgeführt werden.
- `saveAsTextFile(path)`: Schreibt die Objekte des RDDs in eine Textdatei, dessen Pfad `path` auf ein unterstütztes Dateisystem (lokal, HDFS, etc.) zeigt.

2.4.7. Narrow-Dependencies vs. Wide-Dependencies

Eine Transformation überführt ein RDD in ein anderes. In [ZCD+12] bezeichnen die Autoren das Ausgangs-RDD als *Parent-RDD* und das aus der Operation resultierende RDD als *Child-RDD*. Durch die Transformation entstehen Abhängigkeiten (engl. *Dependencies*) der Partitionen des Child-RDDs von den Partitionen des Parent-RDDs. Bei diesen Dependencies wird zwischen zwei Typen unterschieden (siehe Abbildung 2.7a):

- **Narrow-Dependencies:** Von jeder Partition eines Parent-RDDs ist höchstens eine Partition des Child-RDDs abhängig.

[17] Zur Erinnerung: Die Berechnung (Evaluation) von RDDs ist “lazy”.



(a) Beispiele für Narrow-/Wide-Dependencies (b) Beispiel eines Spark-Jobs

Abbildung 2.7.: Die Kästen mit durchgehendem Rand stellen RDDs dar, welche aus einzelnen Partitionen (gefärbten Kästen) bestehen. Schwarz markierte Kästen liegen bereits im Cache vor. Die gestrichelten Kästen markieren Stages, welche jeweils Abfolgen von Narrow-Dependency-Operationen zusammenfassen (Pipelining). Quelle: [ZCD+12]

- **Wide-Dependencies:** Von jeder Partition eines Parent-RDDs können mehrere Partitionen des Child-RDDs abhängig sein.

Narrow-Dependencies erlauben das sogenannte *Pipelining*. Dabei werden auf einer Partition mehrere Operationen direkt hintereinander, unabhängig von den übrigen Partitionen, ausgeführt. Das ermöglicht einen hohen Grad an Parallelität mit minimalem Synchronisationsaufwand. Einige Transformations führen immer zu Narrow-Dependencies (z. B. `map`, `filter`, `union`), andere nur unter günstigen Umständen (z. B. `join` mit co-partitionierten[18] Parent-RDDs).

Wide-Dependencies führen hingegen zum sogenannten *Shuffling*. Das bedeutet, dass die Daten einer RDD neu über Partitionen hinweg verteilt werden müssen. Da dies meist auch bedeutet, dass Daten über Executor-Grenzen (und damit Worker-Grenzen) hinweg kopiert werden müssen, entsteht ein erheblicher Synchronisationsaufwand. Dazu können Disk-I/O, Serialisierung und Netzwerk-I/O gehören, was solche Operationen in der Regel sehr teuer macht. Viele Transformations resultieren für gewöhnlich in Wide-Dependencies und nur unter günstigen Umständen in Narrow-Dependencies. Dazu gehören z. B. `repartition`, `groupByKey` und `join` mit nicht-co-partitionierten Parent-RDDs.

Beispiel 2 (Berechnung von Stages). *Abbildung 2.7b zeigt beispielhaft wie Spark verschiedene Stages berechnet. Um eine Action an RDD G auszuführen, muss Stage 3 berechnet werden. Diese kombiniert die Ergebnisse von Stage 1 und Stage 2. Da in diesem Fall das (verarbeitete) Ergebnis von Stage 1 bereits im Cache vorliegt (RDD B), müssen nur die Stages 2 und 3 berechnet werden.*

[18]RDDs sind co-partitioniert, wenn sie mit demselben Partitioner partitioniert wurden.

3. Verwandte Arbeiten

Tools zur Erzeugung von Test-Daten für die Duplikaterkennung

In diesem Kapitel werden verschiedene Tools vorgestellt, welche die Forschungscommunity in den vergangenen Jahren zum Generieren von Test-Datasets für die Duplikaterkennung hervorgebracht hat. Dabei werden die verschiedenen Ansätze auf Basis bestimmter Kriterien (s.u.) untersucht, um sie anschließend zu vergleichen. Andere Aspekte (wie z. B. die Generierung synthetischer Daten) sind nur am Rande mit eingeflossen. Bei den ausgewählten Kriterien handelt es sich um:

Schema & Flexibilität Es wird überprüft, ob ein Tool schemaunabhängig ist, d. h. ob es nur mit einem festen Schema umgehen kann oder ob das Schema anpassbar ist. Außerdem wird untersucht wie das Schema aufgebaut ist bzw. wie und inwieweit es angepasst werden kann.

Duplikaterzeugung Hier wird festgehalten, ob und wie der Benutzer den relativen Anteil oder die absolute Anzahl, sowie die Verteilung der Duplikate beeinflussen kann. Zudem wird erläutert, wie die Duplikate erzeugt werden.

Fehlergenerierung Dieser Aspekt umfasst die Konfigurierbarkeit der Fehlerkonfiguration durch den Nutzer, den Ablauf der Fehlergenerierung, sowie die möglichen Fehlertypen.

Grenzen & Skalierbarkeit Hier wird die Skalierbarkeit untersucht, indem geprüft wird ob ein Tool Multithreading unterstützt und ob die Berechnung verteilt werden kann. Zudem wird erprobt, welche Größenordnungen von Datenmengen (bzw. Tupelmengen) ein Tool verarbeiten kann und wie sich dabei Speicherbedarf und Laufzeiten verhalten. Hieraus ergibt sich schließlich ein Bild zur Eignung eines Tools für die Erzeugung "großer" Test-Datasets.

Untersucht wurden die Tools auf Basis von Veröffentlichungen, Dokumentationen, der Quellcodes[1] und durch ihre Verwendung. Erprobt wurden alle Tools auf demselben Test-System, welches mit einem Intel Core i5-3570 Prozessor mit vier Kernen à 3.40 GHz und 8 GB Arbeitsspeicher ausgestattet war.

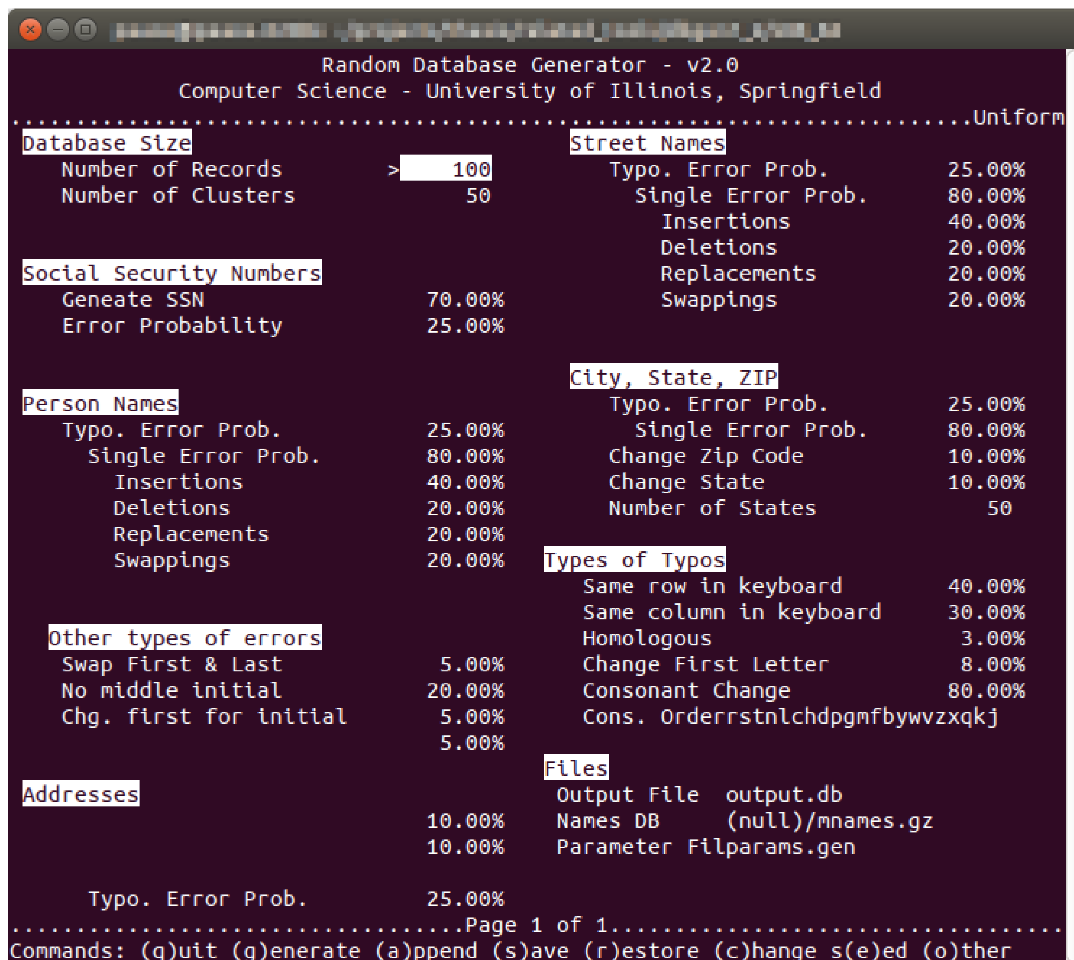


Abbildung 3.1.: Über die simple GUI von DBGen (1997) lassen sich ein paar Aspekte der Daten- und Fehlergenerierung konfigurieren

3.1. DBGen

Der *UIS database generator (DBGen)* [HS95, HS98] wurde 1997 von Mauricio Hernández veröffentlicht und ist damit das erste öffentliche Tool aus der Wissenschaft zum Generieren von Test-Daten für die Duplikaterkennung. Es erzeugt synthetische Personendaten auf Basis von Lookup-Tables und Regeln, welche mit Duplikaten und Fehlern angereichert werden. 1999 hat das Tool ein Update erfahren und ist in Version 2.0 erschienen. Es verfügt über eine rudimentäre graphische Benutzeroberfläche (GUI), kann aber auch über die Kommandozeile benutzt werden. Abbildung 3.1 zeigt die GUI von DBGen.

Schema & Flexibilität

Das Schema der erzeugten Relation ist hartkodiert und enthält zwölf Attribute (inklusive Cluster-ID für den Goldstandard) zur Repräsentation von Personen mit ihren Adressen. Die Software ist quelloffen, in C geschrieben und ein Anpassen des Schemas erfordert

[1] Alle untersuchten Tools sind Open-Source-Software.

daher entsprechende Kenntnisse und kann zu sehr hohem Aufwand führen.

Duplikaterzeugung

In Version 1.0 wurde die Anzahl der Duplikate D prozentual an der Menge der synthetisch erzeugten Original-Records (Cluster) C bestimmt, so dass sich die Gesamt-Tupelanzahl aus $N = C + D$ ergab. In Version 2.0 gibt der Nutzer hingegen die ungefähre Gesamt-Tupelanzahl N und die Anzahl der Original-Records (Cluster) C an, so dass sich die Anzahl der Duplikate aus $D \approx N - C$ ergibt. Die Ungenauigkeit von N folgt aus dem Mechanismus zum Erzeugen der Duplikatcluster-Größen, welcher sich aus einer benutzerdefinierten Verteilung ergibt. Neben der Auswahl einer Verteilung (Gleichverteilung, Poisson-Verteilung oder Pareto-Verteilung) kann der Benutzer auch noch einen entsprechenden Verteilungs-Parameter setzen.

Fehlergenerierung

Die Fehlergenerierung ist stark auf das vorgegebene Schema zugeschnitten. So ist vordefiniert, welche Fehlertypen auf welche Attribute angewendet werden. Es gibt folgende Fehlertypen:

- Typographischer Fehler (für fast alle Attribute)
  - basierend auf Tippfehlern auf einem englischen Tastatur-Layout
  - durch Ändern des ersten Buchstabens
  - durch Ändern eines Konsonanten
- Vertauschen von Vorname und Nachname
- Ersetzen des Vornamens durch sein Initial
- Löschen des Mittelnamen-Initials
- Ändern des Nachnamens (auf Basis einer Lookup-Table)
- Generieren einer neuen Adresse
- Ändern der Hausnummer
- Austauschen der Straßen-Endung (z. B. *street* → *avenue* oder *street* → *st*)

Für jeden dieser Fehlertypen kann der Benutzer einen Wahrscheinlichkeitswert angeben, der für jedes Duplikat unabhängig angewendet wird.

Grenzen & Skalierbarkeit

DBGen ist sehr schnell, benötigt wenig Speicher, seine Laufzeit skaliert linear zur Anzahl der erzeugten Tupel und es kann sogar sehr große Datasets effizient generieren. Allerdings sind die generierten großen Datasets weit entfernt von realistischen Personendaten, was mehrere Gründe hat: Die mitgelieferten Lookup-Tables sind sehr klein (ca. 2.500 Namen und ca. 2.100 *ZIP-City-State*-Tripel) und es wird für Vornamen, Nachnamen und Straßennamen dieselbe Lookup-Table verwendet. Das führt dazu, dass die Daten relativ wenig Diversität aufweisen. Darüber hinaus werden beim Generieren der Daten keine Abhängigkeiten zwischen Attributen (außer zwischen *ZIP*, *City* und *State*) und keine realistischen Verteilungen berücksichtigt. Lediglich beim Punkt der Diversität kann der Benutzer durch Verwendung größerer Lookup-Tables nachbessern.

DBGen ist weder multithreaded noch verteilbar, läuft somit immer nur auf einem Prozessor-Kern und skaliert somit nur vertikal. Immerhin hat sich in Experimenten herausgestellt, dass selbst bei Verwendung deutlich größerer Lookup-Tables (Faktor 500 gegenüber den mitgelieferten Lookup-Tables) nur der konstante Aufwand für das Laden in den Arbeitsspeicher leicht ansteigt, so dass DBGen auch in einem solchen Fall noch sehr effizient arbeitet. Sogar mit diesen deutlich größeren Lookup-Tables konnten Test-Datasets mit einer Million Tupel in unter fünf Sekunden erzeugt werden. Ein Test-Dataset mit einer Milliarde Tupel konnte mit den mitgelieferten Lookup-Tables in etwa 30,5 Minuten erzeugt werden.

3.2. Febrl data set generator

Der *Febrl data set generator* [Chr09] wurde zwischen 2005 (Version 1) und 2008 (Version 2) von Peter Christen und Agus Pudjijono entwickelt und ist Teil des Duplikaterkennungs-Frameworks *Febrl (Freely extensible biomedical record linkage)* der Australian National University. Das Tool folgt einem sehr ähnlichen Ansatz wie DBGen und erzeugt synthetische Personendaten auf Basis von Lookup-Tables sowie Regeln und reichert diese Daten mit Duplikaten und Fehlern an. Die erzeugten Daten sind allerdings deutlich realistischer als die von DBGen, da die Erzeugung der Original-Records deutlich ausgefeilter ist. So werden mehr und größere Lookup-Tables verwendet, relative Häufigkeiten für die möglichen Attributwerte und Abhängigkeiten zwischen bestimmten Attributen berücksichtigt. Das Tool besteht aus einem (quelloffenen) Python-Skript und kann über die Kommandozeile benutzt werden.

Schema & Flexibilität

Das Schema der erzeugten Relation enthält 18 Attribute (inklusive Informationen zum Goldstandard) zur Repräsentation von Personen mit den zugehörigen Adressen und ist hartkodiert in einem Python-Skript. Da es sich bei Python um eine Skript-Sprache han-

delt und der Code daher nicht kompiliert werden muss, ist ein Anpassen des Schemas einfacher als bei DBGen, erfordert aber ebenso entsprechende Kenntnisse und kann je nach Änderungsbedarf zu hohem Aufwand führen.

Duplikaterzeugung

Für die Duplikaterzeugung gibt der Benutzer neben der Anzahl der Original-Records C (entspricht der Anzahl der Duplikatcluster) auch die absolute Anzahl der Duplikat-Tupel D an. Die Gesamt-Tupelanzahl ergibt sich somit aus $N = C + D$. Zur Bestimmung der Clustergrößen kann der Benutzer (wie bei DBGen) zwischen Gleichverteilung, Poisson-Verteilung oder Pareto-Verteilung wählen. Außerdem kann noch ein Parameter für eine maximale Clustergröße gesetzt werden.

Fehlergenerierung

Für die Fehlergenerierung gibt es eine Reihe von Fehlertypen. Im Skript ist für jedes Attribut ein Wahrscheinlichkeitswert für jeden einzelnen dieser Fehlertypen angegeben, welcher entsprechend manuell angepasst werden kann. Aufbauend auf dem Fehlermodell von Christen (siehe Abschnitt 2.3.2) gibt es folgende Fehlertypen:

- Typographischer Fehler auf Basis eines englischen Tastatur-Layouts
- Misspelling-Fehler (auf Basis einer Lookup-Table)
- Einfügen eines Leerzeichens an zufälliger Position
- Entfernen eines Leerzeichens an zufälliger Position
- Löschen eines Attributwertes
- Einfügen eines neuen Wertes (Lookup-Table oder Regel)
- Phonetischer Fehler (auf Basis einer Lookup-Table)
- OCR-Fehler (auf Basis einer Lookup-Table)
- Vertauschen zweier String-Tokens (Wörter) innerhalb eines Attributwertes
- Vertauschen von Werten zwischen Tupeln innerhalb desselben Attributes

Grenzen & Skalierbarkeit

Grundsätzlich lassen sich mit dem *Febrl data set generator* bei annehmbarer Laufzeit einigermaßen große Test-Datasets erzeugen. Durch die ausgefeilten, großen Lookup-Tables sind diese auch deutlich realistischer als diejenigen, die sich durch DBGen erzeugen lassen.

Leider ist auch dieses Tool weder multithreaded noch verteilbar, läuft also immer nur auf einem Prozessor-Kern und skaliert somit nur vertikal. Die maximale Größe des erzeugten Test-Datasets ist dabei durch den Arbeitsspeicher beschränkt. Die Laufzeit steigt etwa linear zur Anzahl der erzeugten Tupel.

In der Praxis steht einer sinnvollen Benutzbarkeit zudem noch ein Bug im Wege, der zu einer Endlosschleife führen kann, welche das Programm nicht terminieren lässt. Die Wahrscheinlichkeit für sein Auftreten steigt mit der Anzahl der generierten Duplikate. So war es mit aktivierten phonetischen Fehlern nicht möglich ein Test-Dataset mit einer Million Tupel (oder mehr) zu erzeugen. Da der Bug nicht auftritt, wenn die phonetischen Fehler deaktiviert sind, wurden die Experimente ohne phonetische Fehler durchgeführt.

Auf dem verwendeten Test-System konnte so ein Test-Dataset mit einer Million Tupel in 4 Minuten und 13 Sekunden erzeugt werden. Das größte Test-Dataset was mit 8 GB RAM erzeugt werden konnte, umfasst drei Millionen Tupel[2].

3.3. GeCo

Abbildung 3.2.: GeCo kann online über ein Web-Interface (links) benutzt werden, welches allerdings nur bis zu 9.999 Tupel erlaubt. Offline gibt es keine GUI und GeCo muss über seine Python-API benutzt werden (rechts).

Als Nachfolger des Febrl data set generator wurde 2012 an der Australian National University das Tool GeCo (Data **Ge**nerator and **Co**rruptor) [CV13] entwickelt. Wie sein Vorgänger, kann es synthetische Daten erzeugen und diese mit Duplikaten und Fehlern

[2]bei Schritten von einer Million Tupel

anreichern. Es gibt aber einige Aspekte, welche GeCo deutlich von seinem Vorgänger abheben.

Statt aus einem einzigen großen Python-Skript, besteht GeCo aus mehreren Python-Modulen, welche in ein eigenes Projekt integriert werden können. Zudem existiert eine Weboberfläche[3] als GUI [TVC13], welche für kleine Datasets benutzt werden kann. Über die Kommandozeile kann GeCo hingegen nicht gesteuert werden.

Darüber hinaus ist die Erzeugung eines Test-Datasets in zwei Schritte unterteilt: das Generieren der Original-Records einerseits und das Erzeugen von Duplikaten und Fehlern andererseits. Indem nur der zweite Schritt angewendet wird, ist es somit auch möglich Test-Datasets aus vorhandenen Daten abzuleiten (vgl. Abschnitt 2.3.4).

Zu guter Letzt kann GeCo noch explizit mit verschiedenen Unicode-Zeichensätzen umgehen, so dass es z. B. in der Lage ist japanische Schriftzeichen zu verarbeiten.

Schema & Flexibilität

Während bei DBGen und dem Febrl data set generator standardmäßig nur ein festgelegtes Personendaten-Schema benutzt werden kann, sieht GeCo vor, dass der Benutzer ein beliebiges Schema definiert (wobei es sich für Personendaten besonders eignet). Wenn man nur ein kleines Test-Dataset (bis 9.999 Tupel) benötigt, kann man dafür die benutzerfreundliche Weboberfläche benutzen (siehe Abbildung 3.2 links). Benötigt man allerdings größere Datenmengen, muss der Benutzer ein Python-Programm schreiben, welches die zur Verfügung gestellten Module benutzt und anhand ihrer APIs einen *Generator*- und/oder *Corruptor*-Prozess definiert (siehe Abbildung 3.2 rechts).

Duplikaterzeugung

Die Duplikaterzeugung entspricht der des *Febrl data set generator* (siehe Abschnitt 3.2).

Fehlergenerierung

Der Prozess der Fehlergenerierung kann in GeCo sehr detailliert konfiguriert werden. So wird über Parameter die Anzahl der Fehler pro Tupel, die Anzahl der Fehler pro Attributwert und die Wahrscheinlichkeiten für die Anwendung eines Fehlers auf bestimmte Attribute gesteuert. Zudem wird für jedes Attribut eine Liste von Fehlertypen mit zugehörigen Wahrscheinlichkeitswerten angegeben. Die Konfiguration ist aufwendig[4] und geschieht entweder über die Weboberfläche oder über Python-*Dictionaries* und -*Lists*.

Die vorhanden Fehlertypen entsprechen im Wesentlichen denen des *Febrl data set generator* (siehe Abschnitt 3.2).

[3] ANU Online Personal Data Generator and Corruptor: `https://dmm.anu.edu.au/geco/`

[4] Beispielsweise muss die Summe der Wahrscheinlichkeitswerte innerhalb der jeweiligen Listen immer 1 ergeben.

Grenzen & Skalierbarkeit

GeCo kann Test-Daten aus vorhandenen Datasets erzeugen, hat viele Features, ist gut dokumentiert und vielfältig konfigurierbar. Wie sein Vorgänger ist es weder multithreaded noch verteilbar, läuft somit immer nur auf einem Prozessor-Kern und skaliert daher lediglich vertikal. Die maximale Größe des erzeugten Test-Datasets ist durch den Arbeitsspeicher beschränkt und die Laufzeit steigt in etwa linear zur Anzahl der erzeugten Tupel. Allerdings arbeitet GeCo dabei extrem langsam und ist daher für große Datasets nur dann geeignet, wenn man Laufzeiten von zig Stunden bis mehreren Tagen in Kauf nehmen kann.

Auf dem verwendeten Test-System benötigte GeCo etwa 13,5 Stunden um ein Test-Dataset mit einer Million Tupel aus einem vorhandenen Dataset abzuleiten. Aufgrund der langen Laufzeit wurde darauf verzichtet einen Versuch zu starten größere Test-Datasets zu erzeugen.

3.4. TDGen

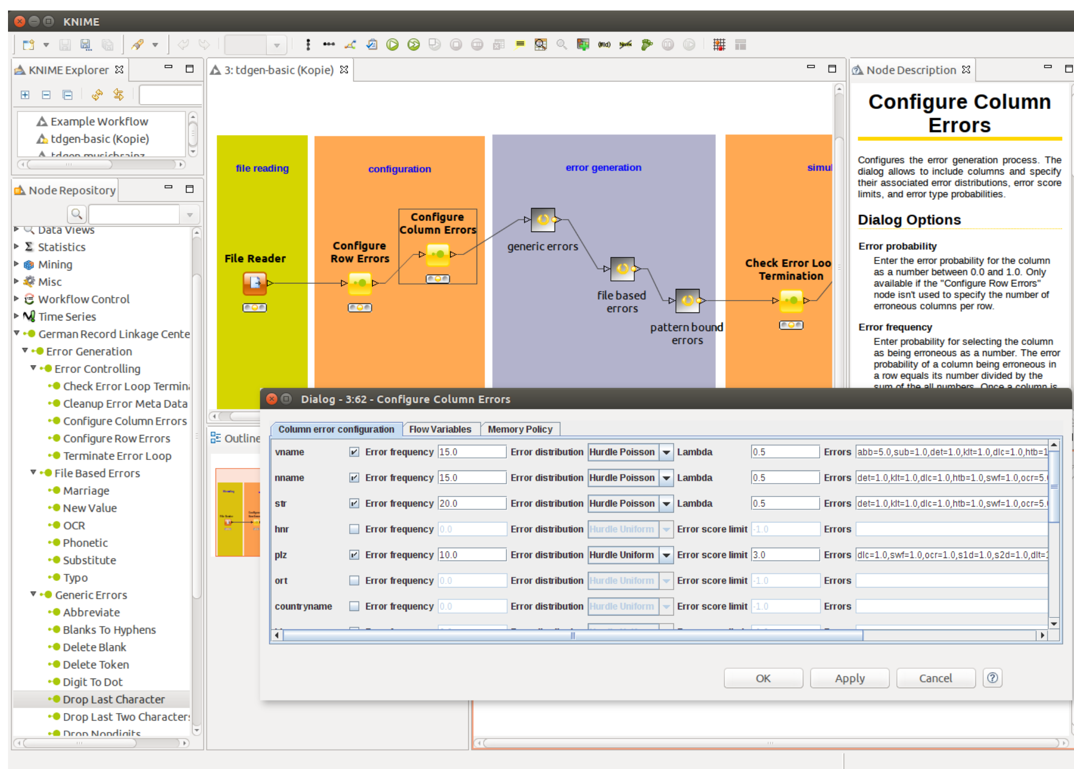


Abbildung 3.3.: TDGen ist als Workflow für die Analytics Platform KNIME implementiert und wird daher über die mächtige KNIME-GUI bedient

TDGen [BR12] wurde 2012 am German Record Linkage Center der Universität Duisburg-Essen entwickelt. Es baut nach eigenen Angaben inhaltlich auf dem Febrl data set generator auf, ist aber fokussiert und beschränkt auf das Anreichern vorhandener Da-

tasets mit simulierten Fehlern. Es soll also als Tool zum Ableiten von Test-Datasets aus vorhandenen Daten dienen (vgl. 2.3.4), wobei es allerdings von Haus aus keine Duplikate erzeugen kann.

TDGen ist als Extension für die Analytics Platform *KNIME*[5] (in Java) implementiert und stellt einen vorkonfigurierten Workflow zur Fehlergenerierung bereit. Standardmäßig wird es über die GUI von KNIME benutzt (siehe Abbildung 3.3), allerdings kann es (wie jeder KNIME-Workflow) prinzipiell auch über die Kommandozeile ausgeführt werden[6].

Schema & Flexibilität

TDGen funktioniert prinzipiell schemaunabhängig und kann beliebige Datasets (als CSV-Datei) laden und verarbeiten. In der Praxis hat sich allerdings herausgestellt, dass für unterschiedliche Datasets mit unterschiedlichen Datentypen (TDGen unterscheidet die Typen String, Integer und Double) und möglichen NULL-Werten deutliche Anpassungen am TDGen-Workflow notwendig sein können. Andernfalls können Fehler produziert werden, welche zu einem Abbruch des Prozesses führen.

Duplikaterzeugung

Eine explizite Duplikaterzeugung ist in TDGen nicht vorgesehen. Um ein Test-Dataset für die Duplikaterkennung mit Duplikaten und Goldstandard zu erzeugen, ist daher eine Vorverarbeitung der Daten durch ein anderes Tool nötig, welches exakte Duplikate einfügt deren Attributwerte anschließend durch TDGen verunreinigt werden können.

Fehlergenerierung

Der Hauptgrund für die Entwicklung von TDGen war es eine flexiblere Kontrolle über Grad und Art der Verschmutzungen zu haben als beim Febrl data generator. So kann der Benutzer den vorhandenen KNIME/TDGen-Workflow anpassen oder einen komplett neuen erstellen. Innerhalb des Workflows können diverse Parameter angepasst oder gesetzt werden.

Der Prozess der Fehlergenerierung kann für Zeilen, Spalten und Felder separat über maximale Fehlerzahlen, Wahrscheinlichkeitswerte und Scores gesteuert werden. Dabei müssen auch Scores für jeden Fehlertyp in jedem Attribut gesetzt werden. Zusätzlich können auch Parameter (falls vorhanden) für die einzelnen Fehlertypen angepasst werden. Insgesamt ist der Fehlergenerierungs-Prozess also sehr präzise steuerbar, allerdings – trotz GUI – nicht besonders benutzerfreundlich.

Insgesamt verfügt TDGen über sehr viele verschiedene Fehlertypen. Diese sind zum großen Teil vom Febrl data set generator übernommen. Andere sind spezieller (z. B. *Keep*

[5]KNIME: `https://www.knime.org/`

[6]Hinweis: Da dies von den TDGen-Entwicklern anscheinend nicht explizit vorgesehen wurde, zeigen sich in der Praxis dabei einige Fallstricke.

Longest Token) oder wurden für bestimmte Attribut-Datentypen (z. B. Datum, Postleitzahl) entwickelt. Eine präzise Dokumentation aller Fehlertypen findet sich im Anhang von [BR12].

Grenzen & Skalierbarkeit

TDGen nutzt zum Teil mehrere Prozessorkerne und hat prinzipiell eine befriedigende Laufzeit, ist aber nur für kleine Datasets einsetzbar. Da die mögliche zu verarbeitende Datenmenge durch den Arbeitsspeicher beschränkt und der Prozess extrem speicherintensiv ist, ist diese obere Schranke relativ schnell erreicht.

In der Standard-Konfiguration konnte TDGen auf dem verwendeten Test-System lediglich zehntausend Tupel verarbeiten. Durch verschiedene Maßnahmen zur Optimierung des TDGen-Workflows konnte die maximale verarbeitbare Tupel-Menge auf ca. hunderttausend hochgeschraubt werden. Dafür benötigte der Prozess etwa 4,5 Minuten.

Grundsätzlich ist TDGen nur vertikal skalierbar. Allerdings ist es denkbar das Problem der Verschmutzung eines großen Datasets in mehrere kleine aufzuteilen und in einem Computercluster zu verteilen. Ein Ansatz hierfür wäre möglicherweise das (kommerzielle) KNIME-Plugin *KNIME Cluster Execution* für den Einsatz auf einem Cluster mit Sun Grid Engine.

3.5. ProbGee

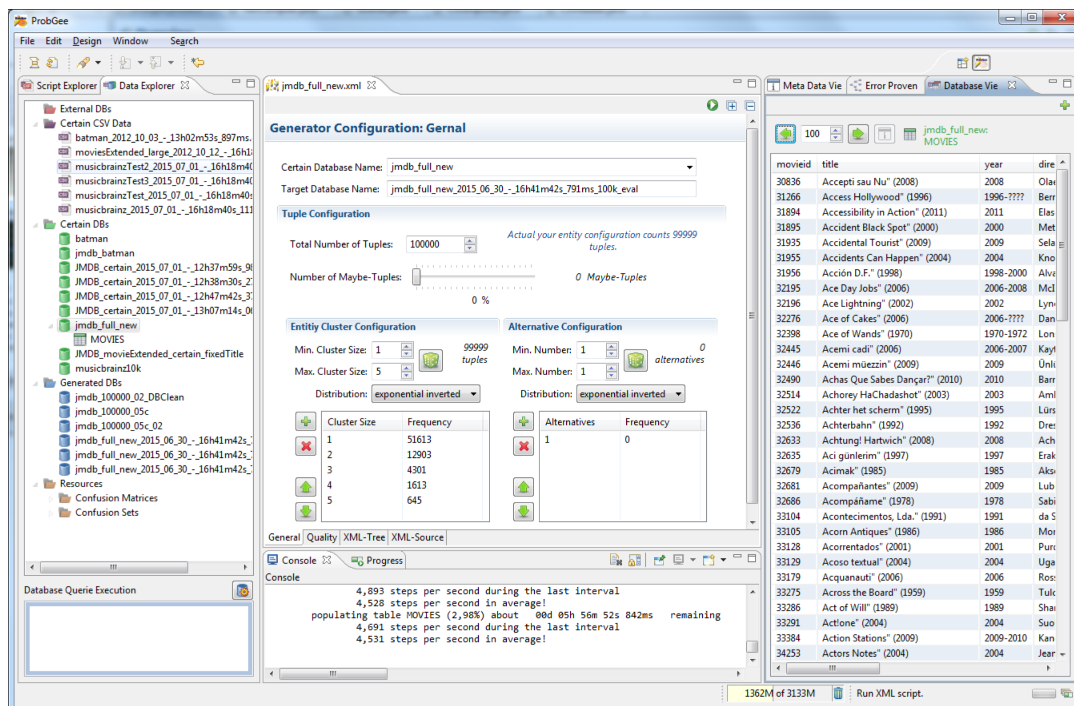


Abbildung 3.4.: Die GUI von ProbGee basiert auf der Rich Client Platform von Eclipse

ProbGee [FW12] ist ein Tool zum Ableiten von Test-Datasets aus vorhandenen Daten,

welches 2012 an der Universität Hamburg entstanden ist. Das besondere ist, dass es speziell für die Duplikaterkennung in probabilistischen Datenbanken[7] [Pan15] konzipiert wurde und somit probabilistische Daten erzeugt. Es kann allerdings auch so konfiguriert werden, dass herkömmliche Test-Daten erzeugt werden.

ProbGee ist auf Basis der Java-basierten *Rich Client Platform* von Eclipse implementiert und kommt daher auch mit einer Eclipse-ähnlichen GUI, welche zur Darstellung eine handvoll von Eclipse-Modulen verwendet (siehe Abbildung 3.4).

Schema & Flexibilität

ProbGee kommt vorkonfiguriert mit Tool-spezifischen Verarbeitungsskripten für eine JMDB[8]-Datenbank, welche aus Film-Daten der *Internet Movie Database (IMDb)* erzeugt wurde. Prinzipiell können allerdings relationale Daten mit beliebigen Schemata aus CSV-Dateien oder gängigen Datenbanken über die Datenbankschnittstelle *Java Database Connectivity (JDBC)* eingelesen werden. Darüber hinaus gibt es noch eine Reihe an Konfigurationsmöglichkeiten, welche sich allerdings auf die Erzeugung probabilistischer Daten beziehen und daher hier nicht näher betrachtet werden.

Duplikaterzeugung

Für die Duplikaterzeugung gibt der Benutzer exakt an, wie viele Duplikatcluster welcher Größe erzeugt werden sollen. Damit nicht jeder Wert einzeln in die Eingabemaske eingegeben werden muss, kann sie auf Basis verschiedener Parameter per Knopfdruck automatisch ausgefüllt und anschließend ggf. manuell angepasst werden. Die Parameter sind die Anzahl der Tupel im erzeugten Test-Dataset, die minimale und maximale Clustergröße, sowie die verwendete Verteilung (Gleichverteilung, Normalverteilung, Exponentialverteilung (auch invertiert), lineare Funktion (steigend oder fallend), zufällige Verteilung).

Fehlergenerierung

Die Fehlergenerierung ist in ProbGee besonders ausgefeilt und stark konfigurierbar. Zur Steuerung des Grades der Verschmutzung kann für jedes Attribut individuell eine Gewichtung und ein Ähnlichkeitsmaß angeben werden. Darüber hinaus legt der Benutzer die sogenannte *Duplicate Similarity* fest, welche einen ungefähren Zielwert für die durchschnittliche Ähnlichkeit innerhalb der Duplikatcluster darstellt.[9] Auch die Fehlertypen und ihre relativen Wahrscheinlichkeiten können attributbezogen festgelegt werden. Dafür muss allerdings das XML der automatisch generierten Generator-Skripte manipuliert

[7] In probabilistischen Datenbanken werden Unsicherheiten bezüglich der Daten anhand von Wahrscheinlichkeitswerten quantifiziert. Dadurch kann nicht nur eine, sondern mehrere "mögliche Welten" modelliert werden.

[8] Java Movie Database: `http://www.jmdb.de/`

[9] Es gibt noch einige weitere Parameter, auf die hier nicht weiter eingegangen wird.

werden.

Die Duplikaterzeugung wird auf Basis der vorher festgelegten Konfiguration durchgeführt. Dabei wird zunächst vom Original-Tupel ausgehend, durch randomisiertes Ausführen von Fehlern, ein Baum von Tupel-Variationen (*Error Provenance Tree*) aufgebaut. Ein bestimmter Algorithmus wählt nun aus diesem Baum eine Menge von Tupel-Variationen, welche gemeinsam ein Duplikatcluster bilden, dessen Duplicate Similarity dem benutzerdefinierten Zielwert möglichst nahe kommt.

In ProbGee sind eine handvoll Fehlertypen implementiert, welche sich zwar besonders für die JMDB-Daten eignen, aber auf beliebige String-Werte angewendet werden können. Prinzipiell erlaubt die Software-Architektur auch, dass weitere Fehlertypen (auch für andere Datentypen) für diese Open-Source-Software implementiert werden können.

Die enthaltenen Fehlertypen sind:

- Typographischer Fehler auf Basis von *Confusion Matrices*
- Semantischer Fehler auf Basis von mitgelieferten oder benutzerdefinierten *Confusion Sets* z. B. für Synonyme, unterschiedliche Schreibweisen oder Ähnliches
- Verfälschen von Jahren bzw. Intervallen von Jahren oder Listen von Jahren bzw. Jahres-Intervallen (z. B. *1976* → *1968* oder *2002-2005* → *2012-2015*)
- Vertauschen von Werten benutzerdefinierter Attribute innerhalb eines Tupels
- Ersetzen eines Attributwertes durch *NULL*
- Der Meta-Fehler *ErrorGeneratorGroup* erlaubt die zufällig gewichtete Auswahl und Ausführung eines Fehlers aus einer List von Fehlern

Grenzen & Skalierbarkeit

ProbGee kann Test-Daten aus vorhandenen Datasets erzeugen, hat viele Features und ist vielfältig konfigurierbar. Allerdings ist es weder multithreaded noch verteilbar. Die Laufzeit steigt zwar in etwa linear zur Anzahl der erzeugten Tupel, allerdings ist ProbGee dabei sehr langsam. Die maximale Größe des erzeugten Test-Datasets ist durch den Arbeitsspeicher beschränkt.

Auf dem verwendeten Test-System benötigte ProbGee knapp 6 Stunden um ein Test-Dataset mit hunderttausend Tupeln aus einem vorhandenen Dataset abzuleiten. Ein Test-Dataset mit einer Million Tupel konnte hingegen nicht erzeugt werden, weil der Arbeitsspeicher nicht ausreichte. Für große Datasets ist ProbGee daher grundsätzlich ungeeignet.

3.6. Vergleich & Fazit

In diesem Kapitel wurden verschiedene Tools vorgestellt, welche die Forschungscommunity in den vergangenen Jahren zum Generieren von Test-Datasets für die Duplikat-

Tool	Erscheinungsjahr	Interaktion	Konfigurierbarkeit	Konf'aufwand (mind.)	Schemaunabhängigkeit	Ziel-Ähnlichkeit	horizontale Skalierbarkeit	Laufzeit 100k	Laufzeit 1mio
DBGen	1997	GUI/CLI	gering	gering	✗	✗	✗	5 s	7 s
Febrl	2008	CLI	gering	gering	✗	✗	✗	2 min	(4,2 min)[1]
GeCo	2012	Lib/Web-UI	hoch	hoch	✓	✗	✗	80 min	13,4 h
TDGen	2012	GUI/(CLI)	hoch	mittel	✓	✗	✗	4,5 min[2]	✗
ProbGee	2012	GUI	hoch	mittel	✓	✓	✗	5,9 h	✗

[1] terminiert nicht mit phonetischen Fehlern, daher ohne phonetische Fehler gemessen
[2] Um das Dataset überhaupt verarbeiten zu können, waren einige manuelle Optimierungen notwendig.

Tabelle 3.1.: Eine Übersicht der vorgestellten Tools bzgl. einiger ausgewählter Merkmale

erkennung hervorgebracht hat. In Tabelle 3.1 sind diese Tools in einer vergleichenden Übersicht einiger wichtiger Merkmale zusammengefasst.

Von den vorgestellten Tools sind nur DBGen und GeCo (mit Abstrichen auch der Febrl Dataset Generator) überhaupt in der Lage Test-Datasets mit mindestens einer Million Tupel zu erzeugen. Dabei ist DBGen allerdings sehr unflexibel (nicht schemaunabhängig, wenig konfigurierbar), so dass die erzeugten Daten dementsprechend unrealistisch und nicht wirklich an den geplanten Einsatzkontext anpassbar sind. Immerhin sind Performanz und Skalierbarkeit überragend, so dass sogar Datenmengen von 100 Millionen Tupeln problemlos in wenigen Minuten erzeugt werden konnten. Der Febr Dataset Generator ist ähnlich unflexibel wie DBGen (nicht schemaunabhängig, wenig konfigurierbar), dabei aber deutlich weniger performant. Zudem verhindert ein Bug, dass größere Datasets mit phonetischen Fehlern erzeugt werden können. GeCo hingegen hat viele Features, ist vielfältig konfigurierbar und dementsprechend sehr flexibel. Allerdings hat es eine überaus schwache Performanz und braucht schon für eine Million Tupel über 13,5 Stunden, was es für sehr große Datasets uninteressant macht. Im Ergebnis bleibt festzuhalten, dass keines der betrachteten Tools den Anforderungen aus Abschnitt 1.3 gerecht wird.

Auffällig ist, dass nur wenige dieser Tools mehrere Prozessorkerne nutzen, keines der Tools verteilbar ist und viele der Tools schnell an Speichergrenzen stoßen. Es scheint daher naheliegend an diesen Problempunkten anzusetzen, um Performanz und Skalierbarkeit zu verbessern und somit die Erzeugung größerer Test-Datasets zu ermöglichen.

4. Verfahren

In diesem Kapitel wird ein Verfahren zum Ableiten großer Test-Datasets für die Duplikaterkennung aus vorhandenen Daten vorgestellt. Dafür wird das Verfahren zunächst im Kontext von Duplikaterkennung und dem Generieren von Test-Datasets eingeordnet (Abschnitt 4.1), um anschließend einige Anforderungen an das Verfahren zu formulieren (Abschnitt 4.2). Es folgt die Vorstellung des eigentlichen Konzeptes, welches eine Beschreibung der Instrumente zur Umsetzung der Anforderungen beinhaltet (Abschnitt 4.3), sowie den Vorschlag eines Workflows, in welchen diese Instrumente eingebettet sind (Abschnitt 4.4). Eine konkrete Implementation dieses Verfahrens wird im hierauf folgenden Kapitel 5 vorgestellt.

4.1. Motivation und Einordnung

Wie in den Abschnitten 2.3.3 und 2.3.4 bereits ausführlich dargelegt, werden zur Entwicklung, Erprobung und Evaluation von Verfahren der Duplikaterkennung sogenannte Test-Datasets benötigt. Um an geeignete Test-Datasets zu gelangen, gibt es unterschiedliche Vorgehensweisen (siehe Abschnitt 2.3.4). Dazu gehört unter anderem die Verwendung von Tools zum Ableiten von Test-Datasets aus vorhandenen Daten. Bei dem vorgestellten Verfahren handelt es sich um ein Konzept für ein solches Tool, welches allerdings eine neuartigen Ansatz verfolgt.

Motiviert ist dieser neue Ansatz durch neuartige Verfahren der Duplikaterkennung, welche darauf abzielen auch mit sehr großen Datenmengen umgehen zu können (siehe Abschnitt 1.2). Um auch solche Verfahren sinnvoll entwickeln, erproben und evaluieren zu können, werden hinreichend große Test-Datasets benötigt, welche allerdings durch herkömmliche Tools zum Generieren von Test-Daten nicht (zufriedenstellend) erzeugbar sind. Mit dem vorgestellten Ansatz können solche hinreichend großen Test-Datasets in akzeptabler Laufzeit erzeugt werden, wodurch diese Lücke geschlossen werden soll.

Betrachtet man den typischen Duplikaterkennungsprozess in Abbildung 2.2, handelt es sich bei dem hier vorgestellten Verfahren also um ein Hilfsmittel, welches die Phase der *Evaluation* für große Datenmengen ermöglicht.

4.2. Anforderungen an das Verfahren

Um der Motivation und dem angedachten Zweck gerecht zu werden, muss das geplante Verfahren eine Reihe von Kriterien erfüllen. Die wichtigsten Aspekte sind zusammenge-

fasst in den folgenden Anforderungen formuliert:

Effizienz & Skalierbarkeit Die primäre Anforderung an das Verfahren ist, dass es sehr große Test-Datasets in akzeptabler Laufzeit[1] erzeugen kann. Kaum eines der in Kapitel 3 vorgestellten Tools ist dazu in der Lage. Ein Grund dafür ist, dass alle diese Tools nur vertikal skalieren. Dadurch ist die Datenmenge der zu generierenden Datasets von der Hardware eines monolithischen Systems begrenzt. Durch die geringe Effizienz der meisten dieser Tools, ist diese Grenze zudem meist sehr schnell erreicht. Um diese Hauptanforderung erfüllen zu können, muss das Verfahren also zumindest hinreichend effizient sein oder horizontal skalieren. An dieser Stelle werden beide Aspekte als Anforderungen formuliert: möglichst hohe Effizienz und Skalierbarkeit.

Schemaunabhängigkeit Das Verfahren soll Test-Datasets aus beliebigen vorhandenen relationalen Datasets ableiten können. Dazu muss es in der Lage sein mit verschiedenen Schemata und Datentypen umgehen zu können.

realistische Fehler Da als Eingabe von einem (möglichst) sauberen Dataset ausgegangen wird, ist das Verfahren unter anderem für das Injizieren von Fehlern zuständig. Nach dem Gargabe-In/Garbage-Out-Prinzip können nur Test-Datasets mit realistischen Fehlern zu aussagekräftigen Evaluierungen führen (vgl. Abschnitt 2.3.4). Um die erzeugten Test-Datasets in der Duplikaterkennung also sinnvoll einsetzen zu können, sollten die injizierten Fehler daher möglichst denen realer Datasets der jeweiligen Domäne entsprechen.

flexible Konfigurierbarkeit Auf der einen Seite soll das Verfahren die Möglichkeit bieten, aus einem Eingabe-Dataset – je nach Konfiguration – verschiedene Test-Datasets mit unterschiedlichen Eigenschaften zu erzeugen. Dazu gehören zum Beispiel Parameter wie die Tupel-Anzahl, der Anteil der Duplikate, der Verschmutzungsgrad oder die Art der enthaltenen Fehler. Auf der anderen Seite soll der notwendige Konfigurationsaufwand für den Benutzer so gering wie möglich gehalten werden, um die Nutzung dieses Tools zu erleichtern und auch unversierteren Benutzern zu ermöglichen. Diese scheinbaren Gegensätze gilt es in Einklang zu bringen.

4.3. Instrumente zur Umsetzung der Anforderungen

Zur Erfüllung der verschiedenen Anforderungen ist ein Konzept entstanden, welches eine Reihe von Instrumenten in einen Workflow einbettet. Die Instrumente dienen jeweils als Hilfsmittel zur Umsetzung bestimmter Anforderungen (fettgedruckt). Im Folgenden werden diese Instrumente erläutert, der gesamte Workflow wird anschließend in Abschnitt 4.4 beschrieben.

[1] Es wird hier absichtlich darauf verzichtet konkrete Anforderungen an die Laufzeit zu stellen, da diese von zu vielen Faktoren abhängig ist.

4.3.1. Parallelisierung & Verteilung

Parallelisierung & Verteilung ermöglichen sowohl vertikale als auch horizontale Skalierbarkeit: Zum einen können durch Parallelisierung auf einzelnen Rechnern alle Kerne moderner Mehrkern-Prozessoren ausgenutzt werden (**vertikale Skalierbarkeit**). Zum anderen kann durch die Verteilung von Arbeitslast über mehrere Rechner hinweg (z. B. in Computerclustern) **horizontale Skalierbarkeit** erreicht werden. Durch letzteres können Hardware-Ressourcen wie Rechenleistung, Arbeitsspeicher und Sekundärspeicher theoretisch beliebig erweitert werden.

4.3.2. Data Profiling

Data Profiling [RD00, Nau14] beschreibt die automatisierte Analyse von Daten durch unterschiedliche Analysetechniken. In dem hier vorgestellten Verfahren wird attributbezogenes Data Profiling eingesetzt, um verschiedene statistische Charakteristika der einzelnen Attribute zu ermitteln. Diese werden zusammen mit Schema-Informationen der Datenquelle in einem *Datenprofil* festgehalten. Dieses Instrument kann allein zwar keine der gestellten Anforderungen erfüllen, dafür dient es aber als vielseitiges Mehrzweck-Hilfsmittel, da es eine mächtige Basis für die folgenden Instrumente schafft.

4.3.3. Angereichertes Tabellenschema

Die Schemata von Datenbank-Relationen können, je nach Art der Datenquelle, unterschiedlich viele Informationen enthalten. Während eine CSV-Datei nur die Anzahl und ggf. die Namen der Attribute festlegt (siehe RFC 4180 [Sha05]), enthält z. B. das Schema einer MySQL-Tabelle auch Informationen zu den Datentypen der Attribute oder darüber ob ein Attribut NULL-Werte enthalten darf.

In dem hier vorgestellten Verfahren wird ein angereichertes Tabellenschema verwendet, welches die typischen strukturellen Schema-Informationen des Eingabe-Dataset umfasst und darüber hinaus noch weitere Informationen für die spätere Verarbeitung der Daten beinhaltet. Das ermöglicht zum einen die **Schemaunabhängigkeit** des Verfahrens und legt zum anderen eine Grundlage zur Erzeugung **realistischer Fehler** und zur Realisierung **flexibler Konfigurierbarkeit**.

4.3.4. Automatisierte Vorkonfiguration

Mit Hilfe von Reasoning-Verfahren wird aus den Charakteristika der Eingabe-Daten automatisiert eine Vorkonfiguration für den eigentlichen Prozess der Test-Daten-Generierung abgeleitet. Für das Reasoning können z. B. bestimmte Regeln oder auch Klassifizierer aus dem maschinellen Lernen eingesetzt werden. Die erzeugten Vorkonfigurationen sollen dazu führen, dass **trotz vieler Konfigurationsparameter**, welche durch den Nutzer manuell angepasst werden können, der notwendige **Konfigurationsaufwand so gering wie möglich** bleibt.

4.3.5. Fehlermodell

Das Fehlermodell (siehe Definition 2.3.8) des vorgestellten Verfahrens wurde in Hinblick darauf entwickelt, dass es möglichst gut zur Umsetzung der unterschiedlichen Anforderungen aus Abschnitt 4.2 beitragen kann.

Das Fehlermodell basiert daher auf sogenannten *Fehlerschemata*, welche jeweils einen definierten Ablauf von Anwendungen verschiedener Fehlertypen auf die Daten darstellen. Dieser Ablauf wird durch flexible Verknüpfung und Verschachtelung verschiedener Fehlertypen mit Hilfe von *Meta-Fehlern* definiert. Durch den Einsatz von Wahrscheinlichkeitswerten und Gewichtungen kann ein solcher Ablauf teilweise randomisiert gestaltet werden. Die Kombination dieser Konzepte ermöglicht eine **flexible Konfigurierbarkeit**.

Die Fehlertypen werden nach ihrem Skopus klassifiziert, also nach dem Bereich der Daten-Tabelle in dem sie wirken. *Zeilen-Fehler* beziehen sich auf einzelne Zeilen (d. h. Tupel), *Spalten-Fehler* auf einzelne Spalten (d. h. Attribute) und *Feld-Fehler* auf einzelne Tabellenfelder (d. h. Attributwerte). Die Unterscheidung nach Skopus ist hilfreich bei der nebenläufigen Ausführung und Verteilung der Fehleranwendung (auf ggf. partitionierten Daten) und unterstützt dadurch die Anforderung der **Effizienz & Skalierbarkeit** (vgl. Abschnitt 5.7).

Feld-Fehler können darüber hinaus noch in weitere Subklassen unterteilt werden, welche sich nach dem Datentyp des Feldes richten. So sind z. B. für numerische Werte andere Fehler geeignet als für Wörter oder Wortfolgen. Diese Unterscheidung ermöglicht den flexiblen Umgang mit verschiedenen Datentypen sowie die Erzeugung noch **realistischerer Fehler**.

Eine weitere Dimension bei der Konfiguration realistischer Fehlerschemata stellen sogenannte *Tupelgruppen* dar. Mit diesem Konzept lassen sich Gruppen von Tupeln festlegen, welche ein gemeinsames Gruppen-Fehlerschema haben. So kann strukturelle, syntaktische und semantische Heterogenität zwischen den Tupeln unterschiedlicher Tupelgruppen erzeugt werden, um unterschiedliche Datenquellen (z. B. Datenbank-Relation oder CSV-Datei), Eingabequellen (z. B. Formular oder API), Erhebungskontexte (z. B. Vertragsabschluss oder unverbindliches Gespräch) oder Zeitpunkte der Erhebung zu simulieren. Die Definition einer einzelnen Tupelgruppe geschieht über eine Bedingung im Fehlerschema, welche über das Tabellenschema definiert ist, wie z. B. *ID <= 1000* oder *CITY="Hamburg"*. Alle Tupel, welche diese Bedingung erfüllen, gehören zur Tupelgruppe dieses Fehlerschemas. Da sich die einzelnen Bedingungen nicht ausschließen müssen, ist es zudem möglich, dass ein Tupel zu mehreren Tupelgruppen gehört.

Die konkrete Auswahl möglicher Fehlertypen ist implementationsabhängig und wird daher im nächsten Kapitel behandelt (siehe Abschnitt 5.7).

4.4. Workflow

Voraussetzung für die Anwendung des vorgestellten Verfahrens ist das Vorhandensein eines möglichst duplikatfreien, sauberen Datasets mit tabellarischer Struktur, welches im Sinne von Abschnitt 1.2 als "groß" bezeichnet werden kann. Ausgehend von einem solchen Dataset als Eingabe, wird mit Hilfe des folgenden Workflows ein Test-Dataset für die Duplikaterkennung erzeugt. Der Workflow ist in Abbildung 4.1 dargestellt und besteht im Wesentlichen aus den folgenden drei Phasen:

1. **Analyse des Datasets:** Das „saubere" Eingabe-Dataset wird mit Hilfe von attributbezogenem Data Profiling analysiert. Dabei werden Charakteristika der Daten ermittelt und in attributspezifischen Datenprofilen zusammengefasst. Darin enthalten sind im Wesentlichen die Verteilung der Datentypen, die Individualität der Attributwerte und datentypspezifische Informationen. Ergänzt werden diese Datenprofile noch um Schema-Informationen der Datenquelle. Die gewonnenen Informationen werden im weiteren Verlauf des Workflows zur Bestimmung eines spezifischen Tabellenschemas genutzt.

2. **Semiautomatische Konfiguration:** Für den späteren Verarbeitungsprozess müssen vorab einige Aspekte konfiguriert werden. Dazu gehören zum einen schemaspezifische Aspekte wie die Konfiguration eines Tabellenschemas und eines Fehlerschemas, und zum anderen schemaunabhängige Aspekte wie die Verteilung der Duplikatcluster und der gewünschte Verschmutzungsgrad.

   Die schemaspezifischen Aspekte können automatisiert aus den Datenprofilen gefolgert werden, welche in der Analyse-Phase erstellt wurden. Dafür wird zunächst aus den Datenprofilen ein Tabellenschema abgeleitet und aus diesem wiederum ein Fehlerschema. Beide automatisiert erzeugten Schemata sind dabei als Vorkonfigurationen anzusehen, welche jeweils unmittelbar nach der Erzeugung durch den Benutzer begutachtet und ggf. manuell angepasst werden können.

   Die Konfiguration der schemaunabhängigen Aspekte (Verteilung der Duplikatcluster, Verschmutzungsgrad, etc.) werden kausal und zeitlich unabhängig von diesem Vorgang manuell durch den Benutzer vorgenommen.

3. **Ausführung des Verarbeitungsprozesses:** Die Ausführung des eigentlichen Verarbeitungsprozesses geschieht auf Basis der vorgenommenen Konfigurationen der schemaspezifischen und schemaunabhängigen Aspekte. Diese Phase besteht im Wesentlichen aus der Injektion von Duplikaten und Fehlern in das Eingabe-Dataset.

In den folgenden Abschnitten werden die einzelnen Phasen des Workflows im Detail erläutert.

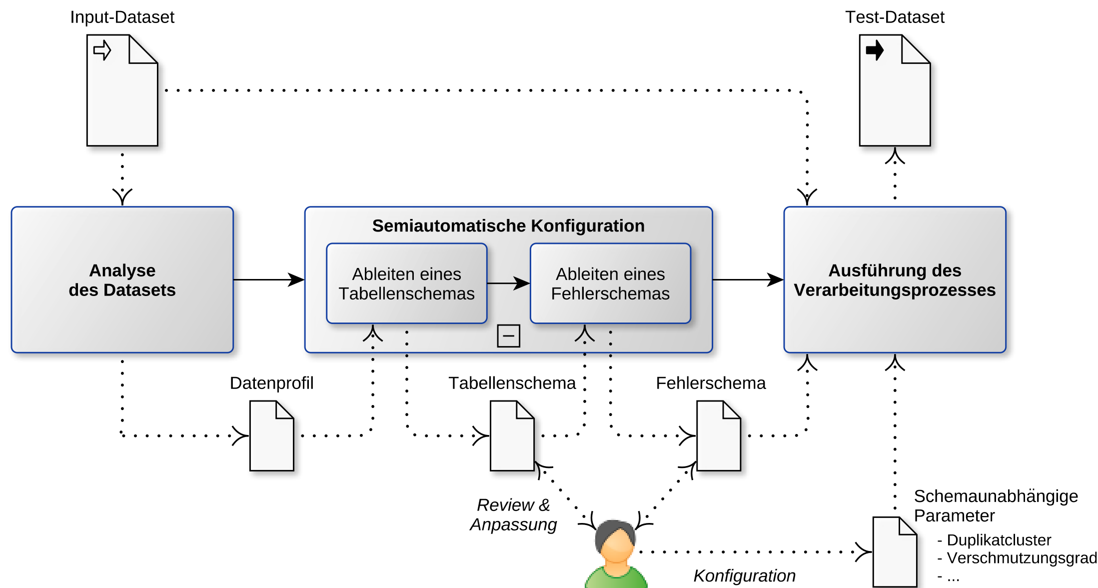


Abbildung 4.1.: Der Workflow des vorgestellten Verfahrens

4.4.1. Analyse des Datasets

In dieser ersten Phase wird das „saubere" Dataset eingelesen und unter Zuhilfenahme von Methoden des attributbezogenen Data Profilings analysiert. Dabei werden zunächst die vorhandenen Schema-Informationen der Datenquelle ermittelt. Anschließend wird für jedes Attribut eine Reihe von (statistischen) Informationen über die zugehörigen Attributwerte erhoben und in einem Datenprofil für das jeweilige Attribut festgehalten. Ein solches Datenprofil enthält folgende Informationen:

- **Verteilung der Basis-Datentypen**: Für jeden Attributwert des untersuchten Attributes wird versucht (unabhängig von den anderen Attributwerten) zu ermitteln, welchen Basis-Datentyp er repräsentiert. Daraufhin wird gezählt und festgehalten, wie oft die verschiedenen Basis-Datentypen innerhalb des Attributes jeweils ermittelt wurden. Zudem wird die Anzahl der enthaltenen NULL-Werte festgehalten. Die Menge der Basis-Datentypen ist abhängig von der Implementation, sollte aber zumindest die vier Typen String, Double, Integer und Boolean abdecken.

- **Basis-Datentyp im Schema**: Falls im Schema des Eingabe-Datasets ein Datentyp für das untersuchte Attribut angegeben ist, wird dieser hier festgehalten. Ist kein Datentyp angegeben, wird als Basis-Datentyp der allgemeinste Datentyp angenommen, welcher in der Verteilung der Basis-Datentypen vorkommt.

- **Individualität der Attributwerte**: Es wird die Anzahl der individuellen Werte (*distinct value count*) gezählt und ihr Anteil an der Gesamtzahl der Attributwerte berechnet (*distinct values ratio*).

- **Datentypspezifische Informationen**: Abhängig vom Basis-Datentyp des betrachteten Attributes, können zusätzlich unterschiedliche Informationen über den Inhalt der Attributwerte interessant sein. Für numerische und String-Attribute könnte das beispielsweise folgendes sein:
  - **Statistiken für numerische Attribute**: Falls es sich um ein numerisches Attribut (z. B. Integer, Double) handelt, werden statistische Basis-Informationen über die Attributwerte erhoben. Dazu gehören u. a. Maße wie Minimum, Maximum, Durchschnitt und Streuung der Attributwerte.
  - **Statistiken für String-Attribute**: Falls es sich um ein String-Attribut handelt, werden statistische Basis-Informationen (die gleichen wie für numerische Attribute) erhoben über:
    - Anzahl der Token pro String
    - Längen der einzelnen Token

Weitere Informationen die erhoben werden könnten, aber in dieser Arbeit nicht berücksichtigt wurden, sind z. B. Abhängigkeiten zwischen Attributen oder Integritätsbedingungen (siehe Abschnitt 7.2).

4.4.2. Semiautomatische Konfiguration

Die aus der Analyse des Datasets gewonnenen Informationen lassen sich nutzen, um in dieser Phase eine sinnvolle Konfiguration der schemaspezifischen Aspekte für die nächste Phase (Ausführung des Verarbeitungsprozess) vorzunehmen. Dafür werden aus den Datenprofilen automatisiert Vorkonfigurationen abgeleitet, welche anschließend vom Benutzer begutachtet und ggf. manuell angepasst werden können.

- **Konfiguration des Tabellenschemas:** Zunächst wird aufbauend auf den Ergebnissen der Analyse-Phase – den Datenprofilen – ein Tabellenschema abgeleitet, welches folgende Aspekte enthält:
  - **Anzahl und Namen der Attribute** entsprechen denen des Eingabe-Datasets. Sind keine Namen vorhanden, werden die Attribute durchnummeriert (d. h. $attribute_1, attribute_2, attribute_3, \ldots$).
  - Falls im Schema eines Attributes ein Datentyp festgelegt ist, wird hieraus ein **Basis-Datentyp** abgeleitet. Andernfalls wird einer aus der Verteilung der Basis-Datentypen abgeleitet werden, welche in der Analyse-Phase erhoben wurde. Sind mehrere Basis-Datentypen in der Verteilung enthalten, ist eine bestimmten Strategie notwendig um einen einzigen Basis-Datentyp auszuwählen. Beispielsweise könnte der speziellste Basis-Datentyp ausgewählt werden, welcher alle Werte des Attributes ohne Informationsverlust darstellen kann.

- Neben dem Basis-Datentyp kann einem Attribut noch ein **Abstrakter Datentyp** zugeordnet werden. Dazu gehören spezifischere Konzepte wie z. B. *Wort*, *Wortfolge* oder *Text* zur Unterscheidung von String-Typen. Die Zuordnung solcher abstrakter Datentypen zu den Attributen kann auf Basis der Datenprofile z. B. mit Hilfe von Reasoning-Systemen geschehen oder manuell durch den Benutzer durchgeführt werden.
- Falls im Schema des Eingabe-Datasets nicht definiert ist, ob ein Attribut **nullable** ist, kann diese Eigenschaft über das Vorhandensein von NULL-Werten innerhalb der jeweiligen Attributwerte festgelegt werden.
- Falls im Schema des Eingabe-Datasets nicht definiert ist, ob ein Attribut **unique** ist, kann diese Eigenschaft auch aus der Individualität seiner Attributwerte (distinct values ratio) abgeleitet werden.
- Über einen booleschen Wert **consider** kann der Benutzer angeben, ob ein Attribut überhaupt bei der Verschmutzung berücksichtigt werden soll.
- Aus den ermittelten Informationen über das Attribut wird schließlich noch ein passendes **Ähnlichkeitsmaß** gefolgert, welches diese Art von Datenwerte sinnvoll vergleichen kann.

Das abgeleitete Tabellenschema ist als Vorschlag anzusehen und kann vom Benutzer begutachtet und manuell angepasst werden.

- **Konfiguration des Fehlerschemas:** Auf Basis des Tabellenschemas wird nun ein Fehlerschema (siehe Abschnitt 4.3.5) erzeugt. Dabei wird gefolgert, welche Attribute mit welchen Fehlertypen versehen werden, da nicht jeder Fehlertyp für jedes Attribut geeignet ist.

  Auch diese Vorkonfiguration ist als Vorschlag zu interpretieren und durch den Benutzer manuell anpassbar.

Zudem müssen noch einige schemaunabhängige Aspekte für den Verarbeitungsprozess festgelegt werden. Diese werden manuell durch den Benutzer konfiguriert, was prinzipiell kausal und zeitlich völlig unabhängig von dieser (und der ersten) Phase geschehen kann. Es handelt sich bei den schemaunabhängigen Aspekten hauptsächlich um folgende:

- **Konfiguration der Duplikatcluster:** In diesem Punkt wird festgelegt, wie viele Duplikatcluster welcher Kardinalität erzeugt werden sollen. Dies kann auf unterschiedliche Weise geschehen:
  - **exakt**, durch die exakte Angabe der gewünschten Anzahlen von Duplikatclustern bestimmter Kardinalitäten.
    Beispiel: *1.000 x 2er, 500 x 3er, 100 x 4er, 10 x 5er*

  - **absolut**, durch die Angabe der absoluten Anzahl gewünschter Duplikate und einer Verteilung zur Berechnung einer exakten Duplikatcluster-Konfiguration.
    Beispiel: *5.000 Duplikate; Normalverteilung*
  - **relativ**, durch die Angabe des relativen Anteils gewünschter Duplikate an der Quellgröße (oder Zielgröße) und einer Verteilung zur Berechnung einer exakten Duplikatcluster-Konfiguration.
    Beispiel: *50% der erzeugten Tupel sollen Duplikate sein; Gleichverteilung*

- **Konfiguration des Verschmutzungsgrades:** Hier wird ein Wert festgelegt, der angibt wie verschmutzt die Tupel innerhalb der Duplikatcluster in etwa sein sollen. Der Verschmutzungsgrad wird über einen Ähnlichkeitswert modelliert. Für die Berechnung eines solchen Wertes sind verschiedene Methoden denkbar. Beispielsweise kann er sich aus allen Durchschnittswerten der Ähnlichkeitswerte aller möglichen Tupel-Paare aller Duplikatcluster ergeben. Die angegebene Größe hängt entsprechend maßgeblich von den Daten und den verwendeten Ähnlichkeitsmaßen ab.

Die gesamte Konfiguration lässt sich abspeichern und später erneut laden, um eine gewisse Reproduzierbarkeit von erzeugten Test-Datasets herzustellen.[2] Dabei werden auch Meta-Daten festgehalten, wie z. B. der Name der Eingabe-Datei oder Datum und Uhrzeit der Erzeugung.

4.4.3. Ausführung des Verarbeitungsprozesses

In dieser Hauptphase wird die Datenverunreinigung und damit die eigentliche Erzeugung des Test-Datasets durchgeführt. Dafür sind drei Schritte notwendig:

- **Injektion von Duplikaten:** Der vorgenommenen Konfiguration der Duplikatcluster folgend, wird für jedes Duplikatcluster C zufällig ein Tupel ausgewählt und $|C| - 1$ mal dupliziert, wobei $|C|$ für die jeweilige Clustergröße steht. Die erzeugten Duplikate werden dem Dataset hinzugefügt. Dabei wird festgehalten, welche Tupel zum selben Duplikatcluster gehören.

- **Injektion von Fehlern:** Das zuvor definierte Fehlerschema wird iterativ auf das Dataset angewendet, bis der gewünschte Verschmutzungsgrad erreicht ist. Dafür wird zwischen den einzelnen Iterationen der erreichte Verschmutzungsgrad berechnet und mit dem gewünschten Zielwert aus der Konfiguration (s.o.) verglichen. Sobald dieser Zielwert ungefähr[3] erreicht ist, wird die Schleife beendet.

  Durch diesen simplen Mechanismus ist die Präzision der Annäherung an den gewünschten Verschmutzungsgrad von der Verschmutzungswirkung des Fehlerschemas abhängig: Je stärker eine Iteration des Fehlerschemas ein Dataset verschmutzt,

[2]Wegen des randomisierten Charakters des Verarbeitungsprozesses, kann keine exakte Reproduzierbarkeit gewährleistet werden.
[3]Wann ein Zielwert "ungefähr erreicht ist", wird in der jeweiligen Implementation festgelegt.

desto größer ist potentiell die Differenz des Ergebnisses zum gewünschten Verschmutzungsgrad.

Zudem ist zu beachten, dass ein Fehlerschema einen definierten Ablauf darstellt. Dadurch haben die Fehleranwendungen innerhalb einer Iteration i. d. R. eine feste Reihenfolge. Auch die Fehlerschemata der verschiedenen Tupelgruppen werden nacheinander ausgeführt, entsprechend der Reihenfolge in der sie definiert wurden.

- **Speicherung:** Zum Schluss wird das modifizierte Dataset als Test-Dataset gespeichert. Dabei wird ihm sein Goldstandard hinzugefügt, indem die Cluster-IDs der Tupel als zusätzliches Attribut des Datasets mit persistiert werden.

5. Implementation: DaPo – Data Polluter

In diesem Kapitel wird der Data Polluter (DaPo) vorgestellt. Bei diesem Tool handelt es sich um eine Implementierung des in Kapitel 4 beschriebenen Verfahrens zum Ableiten großer Test-Datasets für die Duplikaterkennung aus vorhandenen Daten. Zum besseren Verständnis sollte daher Kapitel 4 vor diesem gelesen werden.

Zu Beginn dieses Kapitels wird die Wahl des verwendeten Frameworks als Plattform begründet (Abschnitt 5.1), die verwendete Programmiersprache Scala vorgestellt (Abschnitt 5.2) und kurz die Softwarearchitektur von DaPo umrissen (Abschnitt 5.3). Im Anschluss wird beschrieben wie DaPo benutzt werden kann (Abschnitt 5.5), wie der DaPo-Workflow abläuft (Abschnitt 5.6) und wie das zu Grunde liegende Fehlermodell implementiert wurde (Abschnitt 5.7). Abschließend werden noch einige interessante Aspekte dieser Implementation im Detail erläutert (Abschnitt 5.8).

5.1. Apache Spark als Plattform

Als Plattform für den Data Polluter wurde das Cluster-Computing-Framework Apache Spark (kurz: Spark) ausgewählt, welches in Abschnitt 2.4 vorgestellt wurde. Es eignet sich in besonderem Maße zur unmittelbaren Realisierung von zwei der fünf Instrumente, welche in Abschnitt 4.3 herausgearbeitet wurden:

1. Als Cluster-Computing-Framework gehören Parallelisierung und Verteilung (siehe Abschnitt 4.3.1) zu den Hauptaspekten, welche diese Plattform ermöglicht und zu optimieren versucht.

2. Einer der Hauptzwecke für den Spark konzipiert wurde, ist die Analyse großer Datenmengen [ZCF+10]. Daher eignet es sich auch besonders gut für Data Profiling (siehe Abschnitt 4.3.2). So bietet die Spark-API z. B. von Haus aus Operationen zur (statistischen) Daten-Analyse. Im Übrigen dient das Data Profiling als Grundlage für das angereicherte Tabellenschema sowie für die automatisierte Vorkonfiguration und hilft somit mittelbar auch diese Anforderungen umzusetzen.

Neben den oben genannten Aspekten, welche unmittelbar oder mittelbar die Umsetzung der Anforderungen des Verfahrens unterstützen, werden auch die Anforderungen bietet Spark noch eine Reihe weiterer indirekter Vorteile für die Entwicklung und das Deployment:

Zum einen abstrahiert die Spark-API in hohem Maße von den Parallelisierungs- und Verteilungsaspekten. Zum anderen bietet Spark vielfältige Möglichkeiten des Deployments einer Spark-Applikation. So kann ein und dieselbe Anwendung sowohl auf einem herkömmlichen Desktop-Computer, als auch auf einem professionellen Computercluster ausgeführt werden und skaliert dabei mit der Hardware. Diese Aspekte machen die Entwicklung und das Deployment einer potentiell verteilten, skalierbaren Anwendung sehr bequem. Darüber hinaus eignet sich Spark durch seinen in-memory-Ansatz besonders gut für iterative Prozesse, was für das hier implementierte Verfahren zutrifft.

Eine ausführlichere Vorstellung von Apache Spark kann in Abschnitt 2.4 nachgelesen werden.

5.2. Scala als Programmiersprache

Apache Spark bietet APIs für die Programmiersprachen Scala, Java, Python und R. Da Spark selbst in Scala entwickelt wurde (und immer noch entwickelt wird), ist die Scala-API auch am weitesten entwickelt und am besten dokumentiert. Aus diesen Gründen wird in dieser Arbeit auf die "native" Spark-Sprache Scala gesetzt.

Scala ist eine vielfältige Programmiersprache, welche sowohl funktionale als auch objektorientierte Paradigmen vereint. Da Scala-Code zu Java-Bytecode kompiliert wird, werden Scala-Programme auf Java Virtual Machines (JVM) ausgeführt. Diese Tatsache ermöglicht eine gute Integration von Scala und Java. So können z. B. Java-Klassen in Scala und Scala-Klassen in Java benutzt werden.

5.3. Softwarearchitektur

Die Softwarearchitektur wird natürlicherweise durch das zu Grunde liegende Framework Apache Spark geprägt.

Das Entwerfen und Implementieren einer echten Präsentationsschicht in Form einer GUI war nicht Ziel dieser Arbeit und existiert daher bislang nicht. Stattdessen wird die Anwendung über die Kommandozeile ausgeführt. Die weiteren Interaktionen mit dem Benutzer geschehen der Einfachheit halber über Konfigurationsdateien im Dateisystem.[1]

Die Anwendungsschicht wird von der Programmlogik im Driver-Programm gebildet. Den einzigen Eintrittspunkt der Anwendung stellt die `Main`-Klasse dar. Über Argumente bei der Ausführung wird die Eingabe und der genaue Workflow der Anwendung definiert.

Die Persistenzschicht wird über das `FileSystem`-Objekt der Hadoop-API angesteuert. Dies ermöglicht über eine einheitliche Schnittstelle sowohl auf das lokale Dateisystem, als auch auf verschiedene (verteilte) Dateisysteme zuzugreifen. Neben dem *Hadoop*

[1] Eine GUI wäre allerdings durchaus wünschenswert, da sie das Konfigurieren deutlich benutzerfreundlicher gestalten könnte.

Distributed File System (HDFS) ist das z. B. *Amazon S3*[2]. (Verteilte) Datenbanken wie *Cassandra* oder *HBase* werden derzeit zwar nicht unterstützt, eine Erweiterung dürfte aber mit geringem Aufwand zu implementieren sein.

5.4. Basis-Datentypen & Abstrakte Datentypen

Das implementierte Verfahren sieht zwei verschiedene Konzepte für Datentypen vor. Zum einen die sogenannten Basis-Datentypen und zum anderen die Abstrakten Datentypen. Diese Konzepte sind in DaPo als die Enumerations `BasicDataType` und `AbstractDataType` implementiert.

Die existierenden Basis-Datentypen sind `String`, `Double`, `Int` und `Boolean`. Jeder Attributwert kann auf einen dieser Basis-Datentypen zurückgeführt werden, wobei `String` der allgemeinste von ihnen ist.

Die Auswahl der Abstrakten Datentypen für Attribute beschränkt sich bislang auf: `Word` für Wörter, `WordSequence` für Folgen von wenigen Worten, `Text` für ganze Texte und `Year` für Jahreszahlen (bestehend aus vier Ziffern).

Die getroffene Auswahl der Datentypen (sowohl für Basis- als auch Abstrakte Datentypen) ist beispielhaft und ausreichend für die MusicBrainz-Datasets (siehe Abschnitt 6.1), welche in dieser Arbeit überwiegend verwendeten wurden. In Hinblick auf andere Datasets ist es hingegen wünschenswert (wenn auch nicht notwendig) diese Auswahl zu erweitern (siehe Abschnitt 7.2).

5.5. Prolog: Vorbereitung und Ausführung von DaPo

Da in dieser Arbeit auf die Implementation einer aufwendigen Präsentationsschicht in Form einer GUI verzichtet wurde, ist es notwendig schon vor dem Start von DaPo einige Parameter festzulegen. Dazu muss eine Basis-Konfiguration vorgenommen und ein Run-Modus ausgewählt werden.

5.5.1. Basis-Konfiguration

Die Basis-Konfiguration wird in einer Properties-Datei vorgenommen, welche vom Properties-Loader `PropertiesConfiguration`[3] aus der Java-Programmbibliothek *Apache Commons Configuration*[4] gelesen wird. Sie enthält eine Reihe von Parametern, welche für die Ausführung von DaPo notwendig sind. Zu den wichtigsten dieser Parameter gehören:

[2] Amazon S3 (Simple Storage Service) ist ein Online-Speicher, welcher als kostenpflichtiger Webservice zur Verfügung gestellt wird. `http://aws.amazon.com/de/s3/`

[3] eine Erweiterung von `java.util.Properties`

[4] Apache Commons Configuration: `http://commons.apache.org/proper/commons-configuration/`

- Konfigurations-Parameter für Spark (z. B. URL des Spark-Masters und des Dateisystems)
- Dateipfade für Eingabe- und Ausgabe-Datasets
- Angaben zum Eingabe- und Ausgabe-Format
- Dateipfade für Konfigurationsdateien
- Angaben zu den zu erzeugenden Duplikatclustern
  - Anzahl (absolut) oder Anteil (relativ) der Duplikate
  - Verteilung der Duplikatcluster verschiedener Größen
- Konfiguration des Verschmutzungsgrades

In Anhang A.2 ist eine entsprechende Basis-Konfigurationsdatei dargestellt.

Es ist zu beachten, dass diese Parameter auch die schemaunabhängigen Aspekte aus Abschnitt 4.4.2 umfassen. Während die Konfiguration dieser Parameter im Konzept zeitlich und kausal unabhängig von den ersten beiden Phasen ist (vgl. Abbildung 4.1), wird in dieser Implementation die Konfiguration bereits vor dem Programmstart durchgeführt.

5.5.2. Run-Modi

Der Data-Polluter kann in zwei verschiedenen Run-Modi gestartet werden:

auto Im Auto-Modus definiert der Benutzer vor dem Start lediglich die Basis-Konfiguration. Alle übrigen Phasen des Workflows laufen nach dem Start komplett automatisch ab, bis das Test-Dataset erzeugt ist.

managed Im Managed-Modus kann der Benutzer zwischen bestimmten Schritten interaktiv Einfluss auf den weiteren Programmverlauf nehmen. So wird ihm nach der automatisierten Erzeugung des Tabellenschemas Gelegenheit gegeben dieses Schema nach seinen Vorstellungen anzupassen, bevor fortgefahren wird. Gleiches gilt für das automatisiert erzeugte Fehlerschema.

5.5.3. Ausführung

DaPo liegt als Spark-Applikation in einer JAR-Datei vor. Zur Ausführung muss diese JAR-Datei mit dem Befehl `spark-submit` als Spark-Job an den Spark-Master gesendet werden. Dabei erwartet DaPo mindestens zwei Parameter: Die Angabe des Run-Modus und eine Properties-Datei mit der Basis-Konfiguration.

Beispiel 3 (Ausführen von DaPo). *Listing 5.1 zeigt, wie DaPo im (standardmäßigen) Client-Mode vom Master-Knoten eines Computerclusters aus gestartet wird (vgl. Abbildung 2.5b).*

Die Anwendung befindet sich in diesem Fall im Java-Archiv `DaPo-assembly-1.0.jar`. Ihm werden als Argumente der Run-Modus `auto` und die Datei mit der Basis-Konfiguration `dapo-test.properties` mit übergeben. Zudem wird Spark angewiesen auf jedem Worker-Knoten 6 GB Arbeitsspeicher für diesen Spark-Job zu reservieren und als Eintrittspunkt die Klasse `de.unihamburg.vsis.dapo.exe.Main` zu verwenden.

```
kai@spark-master:~$ ~/spark/bin/spark-submit
   --executor-memory 6G
   --class de.unihamburg.vsis.dapo.exe.Main
   DaPo-assembly-1.0.jar auto dapo-test.properties
```

Listing 5.1: DaPo wird von einem Master-Knoten aus gestartet

5.6. Der DaPo-Workflow

Nach dem Start von DaPo werden zunächst einige einleitende Maßnahmen durchgeführt: Es werden die Argumente verarbeitet, die Basis-Konfiguration geladen und das `SparkContext`-Objekt erzeugt. Anschließend wird der Workflow aus Abbildung 4.1 ausgeführt. Listing 5.2 zeigt vereinfacht wie das im Programmcode aussieht.

```
1 ...
2 val config = new Config(argPropertyFile)
3 val dataProfile = analyze(inputDataset)
4 val tableSchema = deriveTableSchema(dataProfile, config)
5 val errorSchema = deriveErrorSchema(tableSchema, config)
6 pollute(inputDataset, tableSchema, errorSchema, config)
```

Listing 5.2: Vereinfachte Darstellung des DaPo-Workflow als Scala-Code

5.6.1. Analyse des Datasets

In dieser ersten Phase des Workflows werden die Eingabe-Daten analysiert und aus den gewonnenen Informationen Datenprofile gebildet. Dabei wird jedes attributbezogene Datenprofil durch ein `AttributeDataProfile`-Objekt repräsentiert, in welchem (statistische) Informationen zu den Attributwerten gespeichert werden. Zur Erzeugung eines solchen Objektes werden alle Attributwerte eines Attributs des Eingabe-Datasets mit Hilfe einiger Spark-Operationen analysiert. Zu diesen Operationen gehören unter anderem `countByValue()`, `stats()` und `count()` (siehe Abschnitt 2.4.6).

Zur Ermittlung der Verteilung der Basis-Datentypen wird in dieser Implementation wie folgt vorgegangen: Weil die Daten in DaPo aus CSV-Dateien gelesen werden, liegen alle Attributwerte zunächst als Strings vor. Um nun für jeden dieser Werte zu ermitteln, welchen Basis-Datentypen er repräsentiert, wird mit Hilfe der Scala-String-Operationen `toBoolean()`, `toInt()`, `toDouble()` nacheinander (in dieser Reihenfolge) versucht den Attributwert in den entsprechenden Datentyp zu überführen. Der erste Datentyp,

bei dem das gelingt[5], wird als sein Basis-Datentyp deklariert. Ist keine dieser Operationen erfolgreich, wird der Basis-Datentyp `String` angenommen. Ist der String leer oder besteht aus dem Wort "`NULL`" (case-insensitive) ist der Basis-Datentyp undefiniert.

Ein `DataProfile`-Objekt dient als Container für ein Array aus `AttributeDataProfile`-Objekten, wobei die Array-Elemente mit den Attributen des Eingabe-Datasets korrespondieren. Die Signaturen der beiden Klassen sind in Listing 5.3 festgehalten. Die Bedeutungen der einzelnen Klassenvariablen decken sich im Wesentlichen mit den Angaben in Abschnitt 4.4.1.

```
1  case class DataProfile(attrProfiles: Array[AttributeDataProfile])
2
3  case class AttributeDataProfile(
4    name: String,
5    basicDataType: Option[BasicDataType.Value],
6    types: Map[Option[BasicDataType.Value], Long],
7    count: Long,
8    distinctValuesCount: Long,
9    distinctValuesRatio: Double,
10   nullValuesCount: Long,
11   stats: Map[StatsType.Value, StatCounter]
12 )
```

Listing 5.3: Die Signaturen der beiden Klassen zur Repräsentation der Datenprofile: `AttributeDataProfile` und `DataProfile`

Um weiter zu veranschaulichen, wie Informationen in einem Datenprofil gespeichert werden, wird folgendes Beispiel angeführt.

Beispiel 4 (Statistische Informationen in Attribut-Datenprofilen)**.** *Listing 5.4 zeigt wie statistische Informationen über die Anzahl der Token (pro Tupel) für ein Attribut ausgegeben werden, welches Namen von Musik-Interpreten als Strings enthält.*

```
1 // dp is a DataProfile-Objekt
2 // 5 is the index of the attribute "interpret"
3 println(dp.attrProfiles(5).stats(AttributeStatsType.TokenCounts))
4 >> (count: 10000, mean: 2.25, stdev: 1.32, max: 17.00, min: 1.00)
```

Listing 5.4: Die statistischen Informationen eines Attributes befinden sich im Feld `stats` des entsprechenden `AttributeDataProfile`-Objektes

Die Ausgabe ist wie folgt zu interpretieren: Es wurden 10.000 *Attributwerte untersucht. Die durchschnittliche Token-Anzahl der Musiker-Namen ist* $2,25$*, die entsprechende Standardabweichung beträgt* $1,32$ *und jeder Eintrag besteht aus mindestens einem bzw. maximal* 17 *Token.*

```
kai@spark-master:~$ ~/spark/bin/spark-submit ...
A table schema has been generated. You can review and customize the
 xml file at:
  file:///path/to/file/musicbrainz-unique-10000.csv.tableschema.xml

Press Enter to continue ...
```

Listing 5.5: Im managed-Modus interagiert das Tool mit dem Benutzer

5.6.2. Konfiguration des Tabellenschemas

Da die Eingabe-Daten aus CSV-Dateien gelesen werden, ist über das Tabellenschema der Eingabe-Relation nicht viel mehr bekannt, als die Anzahl der Attribute, die Attributwerte und ggf. die Attributnamen. Um die Daten allerdings mit noch realistischeren Fehlern anreichern zu können, sind weitere Informationen über das zu Grunde liegende Tabellenschema hilfreich. Korrespondierend zu den Angaben in Abschnitt 4.4.2 ist in Listing 5.6 zu sehen, welche Informationen für jedes Attribut eines Tabellenschemas festgelegt werden sollen.

```
1  case class TableSchema(name: String, attributes: Array[Attribute])
2
3  case class Attribute(
4    name: String,
5    basicDataType: BasicDataType.Value,
6    abstractDataType: AbstractDataType.Value,
7    isNullable: Boolean,
8    isUnique: Boolean,
9    simMeasure: () => StringMetric,
10   var consider: Boolean
11 )
```

Listing 5.6: Die Signaturen der Klassen `TableSchema` und `Attribute` zur Repräsentation des Tabellenschemas

Für die Ableitung eines `Attribute`-Objektes aus einem `DataProfile`-Objekt stellt DaPo das Interface[6] `AttributeReasoner` bereit, sowie eine einfache Implementation dieses Interfaces `AttributeReasonerSimple`. Wie diese Implementation aus einem Datenprofil ein Attribut mit seinen verschiedenen Eigenschaften ableitet, wird in Abschnitt 5.8.1 genauer erläutert.

Das gefolgerte `TableSchema`-Objekt wird nun mit Hilfe der Bibliothek XStream[7] als XML-Datei serialisiert und gespeichert. Im managed-Modus beginnt jetzt eine Interaktion mit dem Benutzer: Der Prozess pausiert und der Benutzer wird aufgefordert die erzeugte XML-Datei manuell im Dateisystem zu begutachten und ggf. anzupassen. An-

[5] Bei Misslingen werfen die Operationen entsprechende Exceptions.

[6] Da es in Scala keine Interfaces gibt, werden hier stattdessen abstrakte Klassen verwendet.

[7] XStream ist eine einfache Java-Bibliothek, um Objekte als XML-String zu serialisieren und wieder zu deserialisieren. `http://x-stream.github.io/`

schließend kann der Prozess durch betätigen der Eingabetaste fortgesetzt werden (siehe Listing 5.5). Im auto-Modus wird das abgeleitete Tabellenschema ohne Feedback durch den Benutzer für den weiteren Prozess übernommen.

5.6.3. Konfiguration des Fehlerschemas

Das Tabellenschema wird nun genutzt, um hieraus ein "realistisches" Fehlerschema abzuleiten. Ein Fehlerschema wird durch die Klasse `ErrorSchema` repräsentiert, welche die Wurzel des baumartig aufgebauten Fehlerschemas und einen Wert für den Grad der Fehlervererbung (siehe Abschnitt 5.7.3) enthält.

```
1 case class ErrorSchema(
2   rootComponent: BlockErrorComponent,
3   errorInheritance: Double
4 )
```

Listing 5.7: Die Signatur der Klasse `ErrorSchema` zur Repräsentation eines Fehlerschemas

Wie ein "realistisches" Fehlerschema aussieht, ist von einigen Faktoren abhängig, wie z. B. der Domäne oder des Anwendungskontextes. Daher stellt DaPo das Interface `ErrorSchemaReasoner` bereit, welches ermöglicht anwendungsspezifische Regeln zur Ableitung eines `ErrorSchema`-Objektes zu implementieren. Zudem enthält DaPo eine allgemeine Beispiel-Implementation dieses Interfaces, welche mit dem Namen `ErrorSchemaReasonerSimple` bezeichnet wird und in Abschnitt 5.8.2 detailliert beschrieben ist. Sollte diese Implementierung nicht den Benutzer-Ansprüchen genügen oder nicht zum verwendeten Eingabe-Dataset passen, kann eine eigene Implementation des `ErrorSchemaReasoner` geschrieben werden.

Analog zum Tabellenschema wird das gefolgerte `ErrorSchema`-Objekt mit Hilfe der Bibliothek XStream als XML-Datei serialisiert und gespeichert. Im managed-Modus kann die erzeugte XML-Datei nun manuell im Dateisystem begutachtet und angepasst werden (vgl. Listing 5.5). Im auto-Modus wird das automatisch erzeugte Fehlerschema ohne Feedback durch den Benutzer für den weiteren Prozess übernommen.

5.6.4. Ausführung des Verarbeitungsprozesses

In der letzten Phase wird der eigentliche Verarbeitungsprozess ausgeführt. Dieser Prozess erzeugt aus dem Eingabe-Dataset ein Test-Dataset, indem er ersteres mit Duplikaten und Fehlern anreichert und zusammen mit dem Goldstandard abspeichert.

Injektion der Duplikate

Beim Laden der Basis-Konfiguration wird aus den Angaben zur Konfiguration der Duplikatcluster ein `ClusterDistribution`-Objekt erzeugt, welches eine Verteilung von Duplikatclustern repräsentiert. Es enthält für jede mögliche Clustergröße die Anzahl der

Cluster, die es von dieser Größe geben soll. Für den Fall, dass die Verteilung in der Basis-Konfiguration exakt angegeben wurde, kann sie unmittelbar übernommen werden. Andernfalls muss aus den Angaben zu Verteilungsfunktion, Clustergrößen und der absoluten Anzahl der Duplikate bzw. dem relativen Anteil an Duplikaten eine Duplikatcluster-Verteilung berechnet werden (vgl. Abschnitt 4.4.2).

Auf Grundlage des `ClusterDistribution`-Objektes wird ein `ClusterAllocation`-Objekt erzeugt, welches Informationen darüber enthält, welche Tupel wie oft dupliziert werden sollen. Dafür wird für jedes der Duplikatcluster aus der Cluster-Verteilung zufällig ein Tupel des Eingabe-Datasets als Repräsentant ausgewählt. Festgehalten wird diese Zuordnung in einem Mapping `HashMap[K,V]`, in dem jeweils die ID des zufällig ausgewählten Tupels auf die Größe des zugehörigen Duplikatclusters abgebildet wird. Anschließend wird für jede Tupel-ID `K` aus der Cluster-Verteilung das korrespondierende Tupel `V-1` mal dupliziert und in das Datasets eingefügt. Dabei wird jedem Tupel folgendes zugewiesen:

- ein zufälliger Wert aus dem Wertebereich der Tupel-IDs des Eingabe-Datasets als temporäre Tupel-ID,
- die ID des Ursprungstupels als Cluster-ID und
- eine Duplikatcluster-interne ID.

Die Duplikatcluster-interne ID findet bislang zwar keine Verwendung, es ist aber denkbar sie z. B. dafür zu benutzen Duplikate in irgendeiner Art und Weise an den Tupelgruppen auszurichten (siehe Abschnitt 7.2).

Injektion der Fehler

Nachdem die Duplikat-Tupel in das Dataset injiziert wurden, kann nun das zuvor festgelegte Fehlerschema auf die Eingabe-Daten angewendet werden. Wichtig ist zu beachten, dass nicht nur die Duplikate mit Fehlern versehen werden, sondern potentiell alle Tupel. Das Fehlerschema wird iterativ auf die Daten angewendet, bis der in der Basis-Konfiguration festgelegte Zielwert für die durchschnittliche Duplikatähnlichkeit erreicht ist. Hierfür wird nach jeder Iteration i die momentan erreichte durchschnittliche Duplikatähnlichkeit sim_i berechnet und mit dem Zielwert sim_{target} aus der Basis-Konfiguration verglichen.

Da es äußerst unwahrscheinlich ist, dass eine errechnete durchschnittliche Duplikatähnlichkeit exakt mit dem Zielwert übereinstimmt, wird ein Toleranzwert definiert. Dieser Toleranzwert t_i wird dynamisch zu Beginn jeder Iteration $i > 1$ als die Hälfte der Verschmutzungswirkung δ_{i-1} der letzten Iteration $i-1$ definiert. Die Verschmutzungswirkung δ_i ist definiert als Differenz zwischen der durchschnittlichen Duplikatähnlichkeit der letzten Iteration $i-1$ und der aktuellen Iteration i. Daraus ergibt sich:

$$t_1 = 0 \text{ , und}$$

$$t_i = \frac{\delta_{i-1}}{2} = \frac{sim_{i-2} - sim_{i-1}}{2} \quad \text{mit } sim_0 = 1 \quad \text{für } i > 1$$

Der iterative Verschmutzungsprozess wird so lange fortgeführt bis die Bedingung

$$sim_i > sim_{target} + t_i$$

nicht mehr erfüllt ist. Diese Strategie beruht auf der Annahme, dass die Verschmutzungswirkung einer Iteration ungefähr so stark ist, wie die der vorangegangenen. Die Bedingung prüft also, ob die aktuelle durchschnittliche Duplikatähnlichkeit sim_i bereits so nahe am Zielwert sim_{target} liegt, dass eine weitere Iteration (unter obiger Annahme) zu einer durchschnittlichen Duplikatähnlichkeit führen würde, die weiter vom Zielwert entfernt ist, als die aktuell vorliegende.

Da es sich bei der Injektion der Fehler um einen iterativen Prozess handelt, welcher den Hauptteil der Arbeit von DaPo verrichtet, ist dieser Teil besonders kritisch bezüglich seiner Performanz zu betrachten. Daher wurden zur Verringerung der Laufzeit zwei Maßnahmen getroffen: Zum einen werden die Daten nach ihrer Duplikatcluster-Zugehörigkeit partitioniert und zum anderen wird an den geeigneten Stellen explizites Caching eingesetzt. Details zu diesen Maßnahmen können in den Abschnitten 5.8.3 und 5.8.4 nachgelesen werden.

Speicherung

Da bei der Duplikat-Erzeugung neue Tupel-IDs aus dem Wertebereich der bereits existierenden Tupel-IDs an die Duplikate verteilt wurden, existieren nun unter Umständen einige Tupel-IDs mehrfach. Dies zieht allerdings keine negativen Konsequenzen mit sich, da dieses Vorgehen lediglich der zufälligen Einsortierung der neu erzeugten Tupel in die Ordnung aller betrachteten Tupel dient. Vor dem endgültigen Speichern werden nämlich zunächst die Tupel nach ihren Tupel-IDs sortiert und anschließend durch Durchnummerieren mit neuen Tupel-IDs versehen. So bleibt die alte Ordnung bestehen, die neuen Tupel werden zufällig in diese Ordnung eingefügt und zum Schluss besitzt jedes Tupel eine neue, eindeutige Tupel-ID.

Erst jetzt liegt die Menge der Tupel in der Form vor, in der sie gespeichert werden soll. Die von Spark zum Speichern von RDDs in Text-Dateien vorgesehene Operation `saveAsTextFile(...)` speichert RDDs automatisch korrespondierend zu seinen Partitionen in mehreren durchnummerierten Dateien ab. Da das Ergebnis (ebenso wie die Eingabe-Datei) aber als eine einzelne CSV-Datei vorliegen soll, werden diese Dateien zum Schluss noch mit Hilfe der Funktion `FileUtil.copyMerge(...)` aus der HDFS-API zu einer einzelnen Ergebnis-Datei zusammengeführt.

5.7. Fehlermodell

Das in Abschnitt 4.3.5 vorgestellte Fehlermodell sieht eine Menge verschiedener Fehlertypen vor, welche nach ihrem Wirkungs-Skopus klassifiziert in eine Typ-Hierarchie eingeordnet sind. Zudem können diese Fehlertypen über Meta-Fehler flexibel miteinander verknüpft und verschachtelt werden. In DaPo werden alle Fehlertypen und Meta-Fehler unter dem Begriff *Fehler-Komponenten* zusammengefasst.

5.7.1. Typ-Hierarchie der Fehler-Komponenten

Die Typ-Hierarchie zur Klassifizierung der Fehler-Komponenten ist in DaPo über eine Hierarchie aus abstrakten Klassen realisiert (siehe Abbildung 5.1). Die Oberklasse aller Fehler-Komponenten ist die abstrakte Klasse `ErrorComponent[T]`. Darin ist der Parameter `score` für eine mögliche Gewichtung, sowie die Methode `applyErr(obj: T)` zur Anwendung der Fehler-Komponente auf ein entsprechendes Objekt des generischen Typs `T` definiert. Von dieser Oberklasse erbt die spezialisierte abstrakte Klasse `UncertainErrorComponent[T]`. Diese definiert den Parameter `prob` und eine konkretisierte `applyErr(obj: T)`-Methode, welche die Fehler-Komponente nur mit einer Wahrscheinlichkeit von `prob` tatsächlich anwendet.

Eine ausführbare Fehler-Komponente stellt immer eine Konkretisierung einer der folgenden vier abstrakten Fehler-Klassen dar:

- `FieldErrorComponent` für Feld-Fehler konkretisiert `UncertainErrorComponent` und definiert die Typvariable T als `String`.
- `RowErrorComponent` für Zeilen-Fehler konkretisiert `UncertainErrorComponent` und definiert die Typvariable T als `Tupel`.
- `ColErrorComponent` für Spalten-Fehler konkretisiert `UncertainErrorComponent` und definiert die Typvariable T als `RDD[(Long,Tupel)]`[8].
- `BlockErrorComponent` für Tupelgruppen-Fehler konkretisiert unmittelbar `ErrorComponent` und definiert die Typvariable T als `RDD[(Long,Tupel)]`[8]. Zudem definiert sie den Parameter `condition` und konkretisiert die Methode `applyErr(obj: T)` derart, dass die enthaltenen Fehler-Komponenten nur auf diejenigen Tupel angewendet werden, welche die Tupelgruppen-Bedingung `condition` erfüllen.

Bei Feld-Fehlern kann noch zwischen weiteren Subklassen unterschieden werden, welche sich an den Datentypen der Felder orientieren. In DaPo gibt es bisher die Subklassen `NumericErrorComponent` für Fehler in numerischen Werten, `WordErrorComponent` für Fehler, die sich auf einzelne Wörtern beziehen und `WordSeqComponent` für Fehler,

[8]Der Typ `RDD[(Long,Tupel)]` wird in Abschnitt 5.8.3 erläutert.

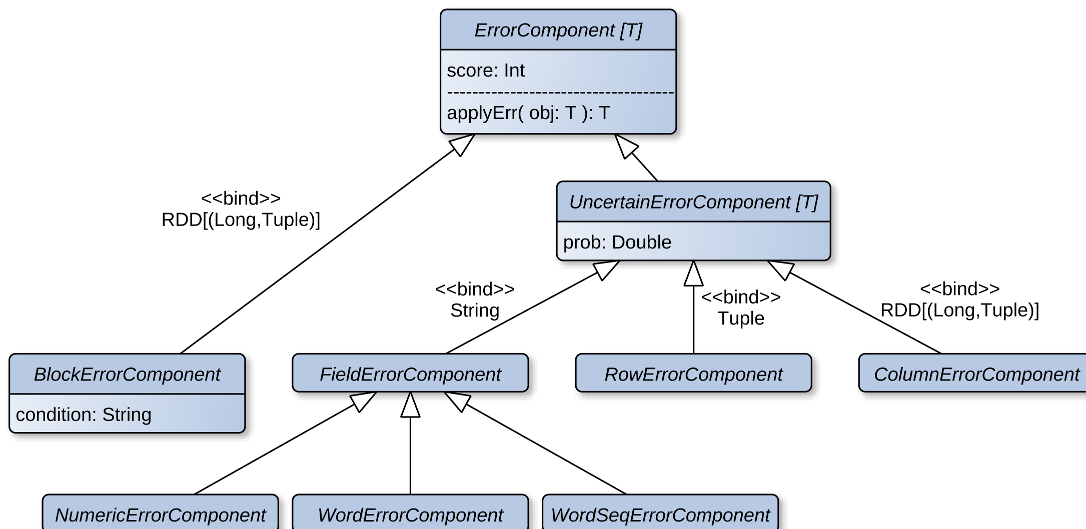


Abbildung 5.1.: Die Typ-Hierarchie der Fehler-Komponenten

die auf Wortfolgen abzielen. Darüber hinaus sind noch diverse weitere Subklassen denkbar, wie z. B. für Datums-Felder oder für Felder, welche menschliche Namen beinhalten (siehe Abschnitt 7.2).

5.7.2. Die verschiedenen Fehler-Komponenten

DaPo enthält bisher eine Menge von rund 50 verschiedenen Fehler-Komponenten. Da es nicht sinnvoll ist an dieser Stelle jede einzelne Fehler-Komponenten zu erläutern, werden in diesem Abschnitt nur die verschiedenen Fehler-Klassen anhand einiger Stellvertreter charakterisiert. Eine Liste aller implementierten Fehler-Komponenten findet sich in Anhang A.4.

Feld-Fehler

Feld-Fehler werden durch die Klasse `FieldErrorComponent` repräsentiert. Unabhängig von dem zu Grunde liegenden Datentyp, verarbeiten Feld-Fehler immer Strings, da die Daten in DaPo als String aus der Datenquelle gelesen und auch als solche im Speicher gehalten werden. Das bedeutet also, dass auch numerische oder boolesche Fehler durch String-Manipulationen erzeugt werden.

DaPo enthält eine Reihe von einfachen Feld-Fehlern. Dazu gehören u. a. die in solchen Tools gängigen Fehler zur Simulation von Rechtschreib-, phonetischen oder OCR-Fehlern. Zudem können Feld-Fehler in DaPo noch weiter spezifiziert werden:

- Wort-Fehler werden durch die Klasse `WordErrorComponent` repräsentiert. Bei dieser Klasse handelt es sich um eine Untermenge von allgemeinen Feld-Fehlern, welche sich besonders für die Anwendung auf einzelne Wörter eignen, wie z. B. eine Abkürzung oder ein Rechtschreibfehler.

- Wortfolgen-Fehler werden durch die Klasse `WordSeqErrorComponent` repräsentiert. Hierbei wird der zu Grunde liegende String in einzelne Wörter (*Token*) und Trennzeichen (*Delimiter*) aufgesplittet. Auf dieser Grundlage können nun einzelne Wörter oder Trennzeichen der Wortfolge vertauscht, ersetzt, gelöscht oder zusammengeführt werden.

- Numerische Fehler werden durch die Klasse `NumericErrorComponent` repräsentiert. In DaPo umfassen sie typische Eingabe-Fehler für numerische Werte, wie sie von Verhoeff spezifiziert wurden [Ver69, Cor12]. Dazu gehören das Vertauschen zweier Ziffern (z. B. 312 $\rightarrow$ 321), die Ersetzung einzelner Ziffern (z. B. 124 $\rightarrow$ 324), der sogenannte "Jump Twin"-Fehler[9] (z. B. 131 $\rightarrow$ 232) und Homophone (z. B. 2 $\rightarrow$ 3 oder 17 $\rightarrow$ 70).

Feld-Fehler verarbeiten einzelne Felder unabhängig voneinander und resultieren daher immer in reinen Narrow-Dependency-Operationen.

Zeilen-Fehler

Zeilen-Fehler werden durch die Klasse `RowErrorComponent` repräsentiert. Sie verarbeiten einzelne `Tupel`-Objekte und wirken somit über mehrere Felder eines Tupels hinweg. Beispielsweise können die Felder vertauscht, ersetzt, verschoben oder zusammengeführt werden. Außerdem können einzelne Feld-Fehler auf alle Felder eines Tupels angewendet werden, um dem Tupel ein einheitliches Fehler-Muster bzw. eine einheitliche Darstellung zu geben (so könnten z. B. alle Felder eines Tupels in Großbuchstaben oder in einen anderen Zeichensatz umgewandelt werden).

Da RDDs in Spark horizontal partitioniert werden und Zeilen-Fehler Tupel unabhängig voneinander verarbeiten, resultieren Zeilen-Fehler immer in reinen Narrow-Dependency-Operationen. Das bedeutet, dass die einzelnen Partitionen bzgl. dieser Operationen unabhängig voneinander (und damit nebenläufig) verarbeitet werden können.

Spalten-Fehler

Spalten-Fehler werden durch die Klasse `ColErrorComponent` repräsentiert. Technisch gesehen verarbeiten sie zwar RDDs mit `Tupel`-Objekten[10], wirken aber in jedem Tupel-Objekt immer nur auf ein bestimmtes Feld (Spalte), welches über den Parameter `col: Int` festgelegt wird. Ähnlich wie bei den Zeilen-Fehlern können diese Felder innerhalb der Spalte Tupel-übergreifend vertauscht, von einem zum anderen übernommen oder verschoben werden. Über den Parameter `fraction: Double` kann festgelegt werden, wie groß der Anteil der von diesem Fehler betroffenen Tupel sein soll.

[9] Der "Jump Twin"-Fehler ersetzt zwei identische Ziffern durch dieselbe andere Ziffer.

[10] Genau genommen werden Objekte vom Typ `RDD[(Long,Tupel)]` verarbeitet. Details dazu können in Abschnitt 5.8.3 nachgelesen werden.

Da RDDs in Spark horizontal partitioniert werden und Spalten-Fehler i. d. R. mehrere Tupel gemeinsam verarbeiten, resultieren sie potentiell in Wide-Dependency-Operationen. Das bedeutet, dass ggf. einzelne Partitionen abhängig voneinander verarbeitet werden müssen, und somit zusätzlich Synchronisationsaufwand entsteht.

Tupelgruppen-Fehler

Eine Tupelgruppe wird in DaPo mit Hilfe der Klasse `BlockCondition` als eine Bedingung modelliert, welche über der Menge aller Tupel des Eingabe-Datasets definiert ist. Tupelgruppen-Fehler werden durch die Klasse `BlockErrorComponent` repräsentiert, welche ein `BlockCondition`-Objekt (also die Definition einer Tupelgruppe) enthält[11][12] und RDDs mit `Tupel`-Objekten verarbeitet.

Neben der Modellierung von Tupelgruppen haben Tupelgruppen-Fehler noch eine weitere wichtige Funktion. Das Wurzel-Element eines Fehlerschemas ist immer ein Tupelgruppen-Fehler. Sie dienen also als Einstiegspunkt für die Ausführung des Fehlerschemas und enthalten daher auch die Basis-Logik dafür. Dabei werden Zeilen- und Spalten-Fehler grundsätzlich unterschiedlich und daher auch getrennt voneinander verarbeitet. Dadurch ergeben sich drei Varianten:

- Der `StandardBlockError` führt erst eine `RowErrorComponent` und anschließend eine `ColErrorComponent`[13] aus. Die Menge der resultierenden Operationen enthält daher potentiell Wide-Dependency-Operationen.
- Der `ColBlockError` führt nur eine `ColErrorComponent` aus und resultiert daher in einer Menge von Operationen, welche potentiell Wide-Dependency-Operationen enthält.
- Der `RowBlockError` führt nur eine `RowErrorComponent` aus und resultiert daher in einer Menge von Operationen, welche ausschließlich Narrow-Dependency-Operationen enthält.

Meta-Fehler

Für jede der vier Fehler-Klassen gibt es zudem noch die sogenannten Meta-Fehler, welche die oben erwähnten flexiblen Verknüpfungen und Verschachtelungen erst ermöglichen.

Der Meta-Fehler `ErrorSequence` besteht aus einer Liste von Fehler-Komponenten-Objekten einer einheitlichen Fehler-Klasse, welche entsprechend ihrer Reihenfolge nacheinander ausgeführt werden.

[11] Genaugenommen wird die Bedingung im Objekt als String gehalten und nur bei Bedarf und on-the-fly in ein `BlockCondition`-Objekt überführt.

[12] Eine leere BlockCondition bedeutet, dass die Tupelgruppe alle Tupel des Eingabe-Datasets umfasst.

[13] Die hier betrachteten Fehler-Komponenten sind i. d. R. Meta-Fehler, in welche eine Reihe weiterer Fehler gekapselt sind.

Der Meta-Fehler `SelectByScore` besteht aus einer Liste von Fehler-Komponenten-Objekten einer einheitlichen Fehler-Klasse, aus der genau ein Objekt zufällig ausgewählt und ausgeführt wird. Bei der zufälligen Auswahl werden die Objekte nach ihrem individuellen `score`-Parameter gewichtet.

5.7.3. Fehlervererbung

In realen Daten ist es möglich, dass Duplikate z. T. dieselben Fehler enthalten. Dies kann zum Beispiel darauf zurückzuführen sein, dass bei der Daten-Eingabe dieser Tupel dieselben Fehler gemacht wurden, ein Tupel durch Kopieren eines anderen Tupel entstanden ist, ein Tupel aus der Fusion mehrerer anderer Tupel resultiert oder die Datenwerte ursprünglich fehlerfrei waren, aber inzwischen veraltet und dadurch nun fehlerhaft sind [DBES09, DBEHS10a, DBEHS10b].

In DaPo ist es möglich, solche Fehler zu produzieren. Dafür werden zwischen den einzelnen Iterationen randomisiert innerhalb einiger Duplikatcluster die Attributwerte eines Tupels mit den Attributwerten eines anderen Tupels desselben Duplikatclusters überschrieben.

Um zu steuern ob und wie häufig so eine Fehlervererbung vorgenommen wird, ist im Fehlerschema der Parameter `errorInheritance: Double` mit einem Wert zwischen 0 und 1 definiert. Je größer der Wert, desto mehr Fehlervererbungen finden statt. Für den Wert 0 findet keinerlei Fehlervererbung statt. Außerdem nimmt die Wahrscheinlichkeit für eine Fehlervererbung mit sinkender Anzahl der voraussichtlichen Rest-Iterationen ab, d. h. je fortgeschrittener der Verschmutzungsprozess, desto weniger Fehlervererbungen werden noch durchgeführt.

5.7.4. Beispiel: Anwendung eines Fehlerschemas

Im folgenden Beispiel wird zunächst ein einfaches Fehlerschema definiert und anschließend auf eine kleine Menge von Personendaten angewendet (eine Iteration).

Listing 5.8 zeigt das Fehlerschema, dessen Wurzelelement aus einem `RowBlockError` besteht. Das bedeutet, dass dieses Fehlerschema nur Zeilen-Fehler (und darin geschachtelte Feld-Fehler), aber keine Spalten-Fehler enthalten kann. Der Parameter `condition` definiert eine Tupelgruppe, welche aus allen Tupeln besteht, dessen sechstes Feld (Index 5) dem String "Hamburg" entspricht. Für diese Tupelgruppe wird der Meta-Fehler `RowErrorSequence`, also eine Sequenz von Zeilen-Fehlern, auf jedes Tupel angewendet: Zuerst wird jeweils mit 50%iger Wahrscheinlichkeit der Fehler `TransposeFieldsInRow` ausgeführt, welcher zwei zufällige Attributwerte innerhalb des Tupels vertauscht. Anschließend wird auf jedem Tupel für jedes Feld (`ApplyFieldErrorOnRow`) der Fehler `ToLower` angewendet, welcher alle Strings in Kleinbuchstaben überführt. Eine Fehlervererbung wird in diesem Beispiel nicht durchgeführt (siehe Zeile 16).

ID	*first name*	*second name*	*street*	*zip*	*city*	*email*
5321	Kai	Hildebrandt	Hafenstraße 8	20535	Hamburg	6hildebr@...
5322	Jule	Fuchs	Domstraße 22	50668	Köln	J.Fuchs@...
5323	Helmut	Schmidt	Rathausmarkt 1	20095	Hamburg	Schmidt@...
5324	Sybille	Sauber	Mauerstraße 89	10117	Berlin	s_sauber@...
5325	Kai	Hildebrandt	Hafenstraße 8	20535	Hamburg	6hildebr@...
5326	Fiete	Fisch	Fischmarkt 11	20535	Hamburg	FiFi@...

Anwendung des Fehlerschemas

ID	*first name*	*second name*	*street*	*zip*	*city*	*email*
5321	kai	hildebrandt	hafenstraße 8	20535	hamburg	6hildebr@...
5322	Jule	Fuchs	Domstraße 22	50668	Köln	J.Fuchs@...
5323	schmidt	helmut	rathausmarkt 1	20095	hamburg	schmidt@...
5324	Sybille	Sauber	Mauerstraße 89	10117	Berlin	s_sauber@...
5325	hildebrandt	kai	hafenstraße 8	20535	hamburg	6hildebr@...
5326	fiete	fisch	fischmarkt 11	20535	hamburg	fifi@...

Tabelle 5.1.: Beispiel für die Anwendung des Fehlerschemas aus Listing 5.8

```
1  <ErrorSchema>
2    <root class="RowBlockError" condition="t.fields(5) == 'Hamburg'">
3      <rowErrors class="RowErrorSequence">
4        <prob>1.0</prob>
5        <errors class="list">
6          <TransposeFieldsInRow>
7            <prob>0.5</prob>
8          </TransposeFieldsInRow>
9          <ApplyFieldErrorOnRow>
10           <prob>1.0</prob>
11           <error class="ToLower" />
12         </ApplyFieldErrorOnRow>
13       </errors>
14     </rowErrors>
15   </root>
16   <errorInheritance>0.0</errorInheritance>
17 </ErrorSchema>
```

Listing 5.8: Beispiel für ein einfaches Fehlerschema

Tabelle 5.1 zeigt wie eine Relation mit Personendaten durch einmalige Anwendung des Fehlerschemas verändert wurde. Die grau hinterlegten Zeilen enthalten Tupel, welche die Tupelgruppen-Bedingung erfüllt haben, d. h. welche verändert wurden. Dabei wurden die hellgrau hinterlegten Tupel nur mit dem Fehler `ToLower` versehen, wohingegen auf den dunkelgrau hinterlegten Tupel zusätzlich der Fehler `TransposeFieldsInRow`

angewendet wurde. Die beiden Tupel mit den IDs 5321 und 5325 sind offensichtlich Duplikate. In der oberen Tabelle sind diese noch (bis auf die ID) übereinstimmend, und daher sehr leicht zu identifizieren. Durch Anwendung des Fehlerschemas wurden in dem Tupel mit der ID 5325 der Vor- und der Nachname miteinander vertauscht, so dass diese Duplikate jetzt eine geringere Ähnlichkeit zueinander aufweisen als zuvor.

5.8. Weitere Implementationsdetails

In den vorangegangen Abschnitten konnten einige interessante Implementationsdetails von DaPo aus Platzgründen nicht vertieft werden. Daher werden im Folgenden einige dieser Aspekte im Detail erläutert.

5.8.1. Die Klasse AttributeReasonerSimple

Die Klasse `AttributeReasonerSimple` ist eine einfache Beispiel-Implementation des Interfaces `AttributeReasoner` und leitet aus einem `AttributeDataProfile`-Objekt ein `Attribute`-Objekt ab (siehe Abschnitt 5.6.2). Die einzelnen Eigenschaften dieses Objektes ergeben sich dabei wie folgt:

- `name`: Der Name eines Attributes ergibt sich i. d. R. direkt aus dem CSV-Header des Eingabe-Datasets. Ist kein CSV-Header vorhanden, werden die Attribute-Namen auf "Attribute 1", "Attribute 2", . . . , "Attribute n" gesetzt.

- `basicDataType`: Da das Schema von CSV-Dateien keine Typ-Informationen beinhaltet, muss der Datentyp aus dem Datenprofil abgeleitet werden. Hierzu wird aus der Verteilung der Basis-Datentypen `types` der speziellste Datentyp ausgewählt, welcher alle Werte des Attributes ohne Informationsverlust darstellen kann. Für die vorhandenen Datentypen ergibt sich dabei die Hierarchie: `String` ist allgemeiner als `Double`, `Double` ist allgemeiner als `Int` und `Int` ist allgemeiner als `Boolean`.

- `abstractDataType`: Die Ableitung eines abstrakten Datentypen erfolgt bisher über eine handvoll einfacher Regeln:

  Für Attribute mit dem Basis-Datentyp `String` wird anhand der durchschnittlichen Token-Anzahl seiner Attributwerte entschieden, ob es sich um ein Attribut für Texte, Wortfolgen oder einzelne Wörter handelt. Ist dieser Durchschnittswert größer als 10 wird der abstrakte Datentyp `Text` gewählt. Ist der Durchschnittswert 10 oder kleiner und größer als 1,2 wird angenommen, dass der abstrakte Datentyp `WordSequence` am geeignetsten ist. Ist der Wert kleiner als 1,2 wird der abstrakte Datentyp `Word` verwendet. Für Attribute mit dem Basis-Datentyp `Int` ist bisher nur der abstrakte Datentyp *Year* definiert. Liegt der kleinste Attributwert eines solchen Attributes über 1500 und der größte unter 2500, wird angenommen, dass es

sich um Jahreszahlen handelt[14]. Für Attribute mit den Basis-Datentypen `Double` und `Boolean` sind bisher keine abstrakten Datentypen implementiert.

Ein Listing des Codes, welcher die genannten Regeln in DaPo umsetzt, ist in Anhang A.3 zu finden.

- `isNullable`: Wenn das Attribut NULL-Werte enthält, wird dieser Wert auf `True` gesetzt, andernfalls auf `False`.
- `isUnique`: Wenn das Attribut keine doppelte Werte enthält, wird dieser Wert auf `True` gesetzt, andernfalls auf `False`.
- `consider`: Dieser Wert wird standardmäßig immer auf `True` gesetzt. Ein Ändern auf `False` ist nur durch den Benutzer bei der Begutachtung möglich.
- `simMeasure`: In der aktuellen Version von DaPo wird der Einfachheit halber immer dasselbe Ähnlichkeitsmaß verwendet (*Jaro-Winkler* [Win90]). Prinzipiell soll allerdings aus den Datentyp-Informationen des Attributes ein Ähnlichkeitsmaß gefolgert werden, welches diese Art von Datenwerten sinnvoll vergleichen kann.

5.8.2. Die Klasse ErrorSchemaReasonerSimple

Die Klasse `ErrorSchemaReasonerSimple` ist eine einfache Beispiel-Implementation des Interfaces `ErrorSchemaReasoner` und leitet dementsprechend aus einem `TableSchema`-Objekt ein `ErrorSchema`-Objekt ab (siehe Abschnitt 5.6.3). Abbildung 5.2 zeigt das Ergebnis einer Ableitung eines solchen Fehlerschemas in einer baumartigen Darstellung. Die Wurzelkomponente ist ein `StandardBlockError`, d. h. dass als erstes eine `RowErrorComponent` und danach eine `ColErrorComponent` ausgeführt wird.

Die `RowErrorComponent` entspricht in diesem Fall einer `ErrorSequence`, welche zunächst mit einer Wahrscheinlichkeit von 50% ein `FieldErrorSchema` auf ein Tupel anwendet und anschließend mit 5%iger Wahrscheinlichkeit mit Hilfe einer `SelectByScore`-Komponente zufällig genau einen Fehler aus einer definierten Menge von Zeilen-Fehler-Komponenten auswählt und anwendet. Das `FieldErrorSchema` ist dabei abhängig vom `TableSchema`-Objekt konstruiert und enthält für jedes Attribut (mit `consider=true`) eine Menge von Feld-Fehlern von denen jeweils genau einer ausgeführt wird. Die einzelnen Mengen von Feld-Fehlern sind abhängig vom Basis-Datentypen, vom abstrakten Datentypen und vom `isNullable`-Wert des jeweiligen Attributes. Für numerische Attribute kommt der Feld-Fehler `NumericTypo` in Frage, welcher Tippfehler in Zahlen produziert. Für Strings ohne abstrakten Datentypen und Wörter existiert

[14] Das ist selbstverständlich eine gewagte Annahme, da es sich auch um Zahlen mit einer ganz anderen Semantik handeln könnte. Allerdings soll es sich hier zum einen eher um ein Beispiel handeln und zum anderen mündet diese Folgerung ja lediglich in einer Vorkonfiguration, welche vom Benutzer angepasst werden kann.

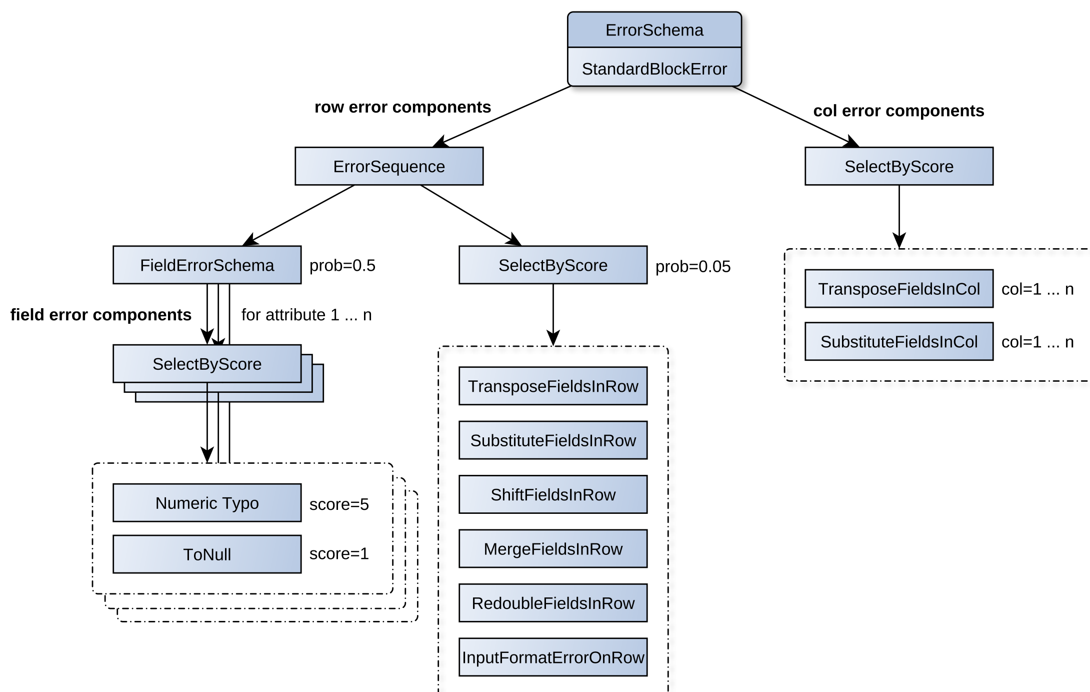


Abbildung 5.2.: Baumartige Darstellung eines Fehlerschemas, welches durch die Klasse `ErrorSchemaReasonerSimple` erzeugt wurde. Knoten repräsentieren Fehler-Komponenten, wobei innere Knoten Meta-Fehlern entsprechen. Die Kästen mit gestricheltem Rahmen repräsentieren Mengen von Fehler-Komponenten, von denen jeweils genau einer zufällig ausgewählt wird. Die Fehler-Komponenten werden durch Depth-First-Traversierung des Baumes nacheinander ausgeführt. Für Knoten an denen keine Parameter stehen, gelten die Standardwerte `prob=1.0` und `score=1`. Die Feld-Fehler (links-unten) sind datentypabhängig (hier ist Attribut 1 vom Typ `Int` und `nullable`).

eine Menge von passenden Fehlern, welche unter dem Namen `WordError` zusammengefasst sind. Für Wortfolgen existiert zudem die Menge `WordSeqError`, welche spezielle Fehler für Wortfolgen enthält, was auch die Anwendung von Wortfehlern auf einzelne Wörter mit einschließt. Für Attribute mit der Eigenschaft `isNullable=true` wird die Fehler-Menge noch um die Komponente `ToNull` ergänzt.

Die `ColErrorComponent` der Wurzelkomponente ist eine `SelectByScore`-Komponente, welche genau einen Spalten-Fehler (`SubstituteFieldsInCol` oder `TransposeFieldsInCol`) für genau ein Attribut (mit `consider=true`) ausführt. Für den Parameter `fraction` wird der Standardwert (`0.001`) verwendet, so dass bei jeder Ausführung jeweils ca. 0,1% der Tupel betroffen sind.

Auf eine Tupelgruppen-Bedingung wurde im `ErrorSchemaReasonerSimple` verzichtet, so dass alle Tupel zusammen die einzige Tupelgruppe bilden. Der Parameter für die Fehlervererbung `errorInheritance` wurde auf den Wert `0.1` gesetzt.

5.8.3. Partitionierung nach Duplikatcluster-ID

Eine gängige Maßnahme, um die Performanz von verteilten Anwendungen zu erhöhen, ist es die Datenlokalität zu verbessern. Einige Vorgänge im DaPo-Workflow verarbeiten die Tupel eines Duplikatclusters gemeinsam. Hierbei ist es von Vorteil, wenn die Tupel eines Duplikatclusters in derselben Partition (d. h. auf dem selben Worker-Knoten) vorliegen. Zu den betroffenen Vorgängen im DaPo-Workflow gehören:

- Injektion der Duplikate (siehe Abschnitt 5.6.4)
- Berechnung der Duplikatähnlichkeit (siehe Abschnitt 5.6.4)
- Fehlervererbung (siehe Abschnitt 5.7.3)

Hierfür werden die Tupel eines Datasets im DaPo-Code mit ihrer Cluster-ID als Schlüssel `K` und dem eigentlichen Tupel-Objekt als Value `V` in einem `RDD[(K,V)]`-Objekt gehalten, so dass sich der Typ `RDD[(Long,Tupel)]` ergibt. Solche speziellen Key-Value-RDDs haben in Spark den impliziten Typen `PairRDDFunctions`, welcher bestimmte Eigenschaften besitzt und zusätzliche Operationen zur Verfügung stellt. Dazu gehört die Möglichkeit die Daten nach ihrem Schlüssel mit einem benutzerdefinierten Partitioner zu partitionieren und die Eigenschaft, dass diese Partitionierung bei Transformations implizit an das Ergebnis-RDD vererbt wird. Durch diese Maßnahme liegen die Tupel des Eingabe-Datasets während des Verschmutzungsprozesses also immer partitioniert nach ihrer Cluster-ID vor.

Im Übrigen ist es denkbar, dass auch eine Partitionierung nach weiteren bzw. anderen Aspekten (z. B. nach Tupelgruppen) unter Umständen Vorteile mit sich bringen kann. Hierfür wären allerdings weitere Nachforschungen notwendig (siehe Abschnitt 7.2).

5.8.4. Explizites Caching

Abbildung 5.3 zeigt, dass jeweils an dem Punkt, an dem zwischen zwei Iterationen die durschnittliche Duplikatähnlichkeit berechnet wird, der Lineage-Graph eine Verzweigung aufweist. Ohne weitere Maßnahmen würde zusätzlich zum Hauptpfad (die Iterationen bis zur Speicherung) für jede dieser Verzweigungen die komplette Lineage bis zu der jeweiligen Verzweigung neu berechnet werden. Damit jedoch jeder Abschnitt der Lineage nur genau einmal berechnet wird, bietet es sich an die RDDs jeweils an diesen Punkten zu cachen, d. h. im Arbeitsspeicher zu persistieren (grün hervorgehoben). Diese Maßnahme ist zwar mit zusätzlichem Speicheraufwand verbunden, ermöglicht aber eine erhebliche Reduzierung der Laufzeit. Für DaPo ist sie zudem logisch von essentieller Bedeutung: Da dieser randomisierte Prozess nicht-deterministisch ist, würde ohne Caching die Berechnung der durchschnittlichen Duplikatähnlichkeit auf anderen Daten beruhen, als im Hauptpfad tatsächlich verarbeitet werden.

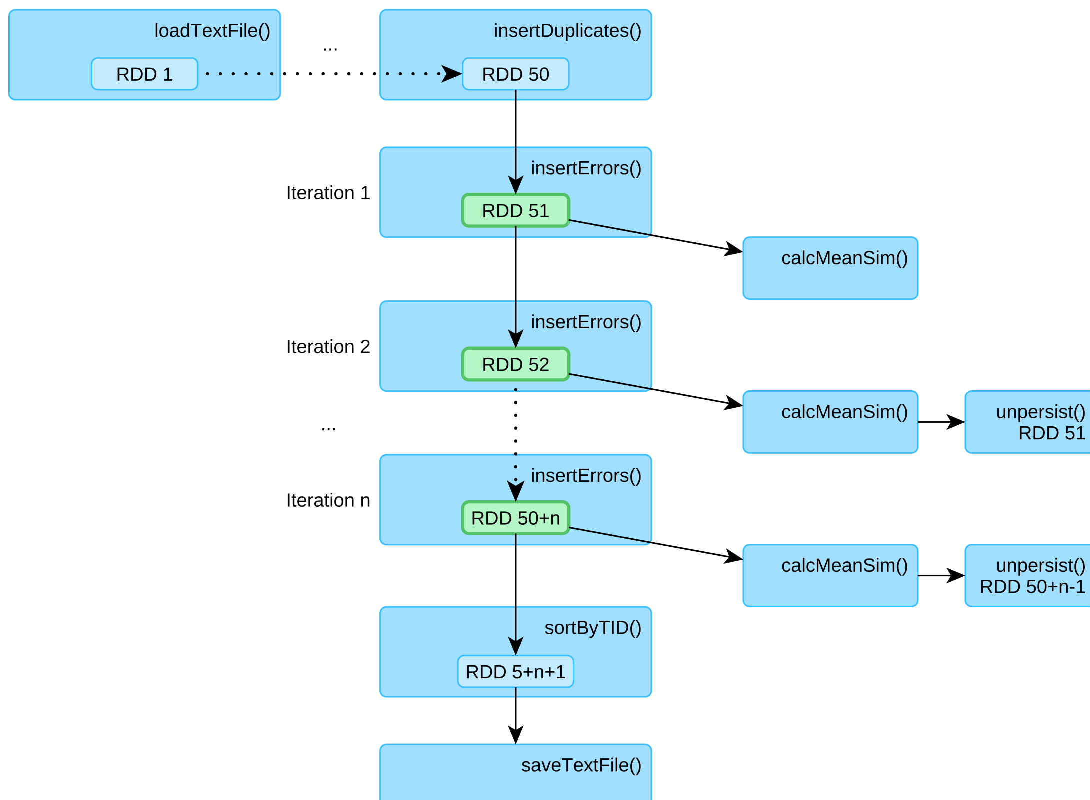


Abbildung 5.3.: Vereinfachte Lineage des DaPo-Workflows

Nach Aussage der Entwickler[15] versucht Spark zwar selbstständig herauszufinden, welche RDDs nicht mehr verwendet werden um sie dann aus dem Cache zu entfernen, allerdings greift dieser Mechanismus in diesem Fall offenbar nicht. Ohne weitere Maßnahmen würde das dazu führen, dass sich mit zunehmender Anzahl an Iterationen der Arbeitsspeicher immer weiter füllt, bis Spark schließlich auf die Festplatte ausweichen muss, wodurch sich die Laufzeit für jede neue Iteration deutlich erhöht.

Daher werden in DaPo die nicht mehr benötigten RDDs aus den vorangegangenen Iterationen explizit mit der Operation `unpersist()` aus dem Cache entfernt. Wichtig ist hierbei zu beachten, dass der Aufruf zum Entfernen eines alten RDDs zum richtigen Zeitpunkt erfolgen muss. Die Operation `unpersist()` darf nämlich erst an dem alten RDD aufgerufen werden, wenn das neue RDD bereits materialisiert im Cache vorliegt, was erst nach Ausführung der Action der Fall ist. In der Abbildung ist dies daran zu erkennen, dass `unpersist()` für RDD 51 erst aufgerufen wird, nachdem RDD 52 durch die Action `calcMeanSim()` materialisiert wurde. Da bis zum Aufruf von `unpersist()` beide RDDs im Cache sind, sollte während des Verarbeitungsprozesses also immer möglichst

[15] siehe:
`https://github.com/apache/spark/pull/126`
`https://github.com/apache/spark/commit/11eabbe125b2ee572fad359c33c93f5e6fdf0b2d`
`https://mail-archives.apache.org/mod_mbox/spark-user/201403.mbox/`
`<467E4BA7-7817-4137-AA81-F1EF0897796F@gmail.com>`

mindestens die doppelte Größe des Datasets als freier Arbeitsspeicher zur Verfügung stehen.

6. Experimentelle Evaluation

In diesem Kapitel wird die experimentelle Evaluation des in Kapitel 5 vorgestellten Tools DaPo beschrieben. Dabei werden insbesondere das Laufzeitverhalten und die Skalierbarkeit bzgl. der Eingabe-Größe sowie der Anzahl der Cluster-Knoten betrachtet.

Die Erzeugung der verwendeten Datasets und das Setup des Computerclusters wurden in Kooperation mit Niklas Wilcke durchgeführt, welcher in seiner Master-Thesis [Wil15b] dieselben Daten und dasselbe Computercluster verwendet hat.

6.1. Datasets

Um DaPo bezüglich der Skalierbarkeit und Verarbeitbarkeit großer Datenmengen evaluieren zu können, werden entsprechend große Datasets im CSV-Format benötigt.

In dieser Arbeit wurde dafür auf die frei verfügbare *MusicBrainz*-Datenbank[1] zurückgegriffen. Diese PostgreSQL-Datenbank enthält mehrere Gigabyte an Meta-Daten über Musik (Interpreten, Songs, Aufnahmen, Veröffentlichungen, Musiklabels, etc.) in einem komplexen relationalen Schema (siehe Anhang A.1) und wird als ca. 11 GB große virtuelle Maschine (VM) zum Download angeboten.

Aus dieser Datenbank wurde ein CSV-Dataset extrahiert, welches Informationen über 16 Millionen Musik-Stücke (*Tracks*) enthält. Wegen des komplexen Datenbank-Schemas war hierfür eine entsprechend komplexe SQL-Abfrage nötig, welche in Listing 6.1 dargestellt ist.

```sql
set client_encoding='utf8';
\copy (
  SELECT tr.id, tr.number, tr.name AS title, tr.length, ac.name
  AS artist, rl.name AS album, rc.date_year AS year, lg.name AS
  language
  FROM track AS tr
  LEFT JOIN artist_credit AS ac ON tr.artist_credit = ac.id
  LEFT JOIN medium AS md ON tr.medium = md.id
  LEFT JOIN release AS rl ON md.release = rl.id
  LEFT JOIN release_country AS rc ON rl.id = rc.release
  LEFT JOIN language AS lg ON rl.language = lg.id
  WHERE ac.name != 'Various Artists'
) To '/tmp/musicbrainz-raw.csv' With CSV HEADER;
```

Listing 6.1: SQL-Abfrage zur Abfrage von Tracks aus der MusicBrainz-DB

[1] MusicBrainz-Datenbank: https://musicbrainz.org/doc/MusicBrainz_Database

Name	# Tupel	Größe (MB)
musicbrainz-unique-10000.csv	10.000	0,89
musicbrainz-unique-100000.csv	100.000	8,95
musicbrainz-unique-1000000.csv	1.000.000	89,71
musicbrainz-unique-10000000.csv	10.000.000	896,75
musicbrainz-unique-full.csv	15.567.518	1396,11

Tabelle 6.1.: Die verwendeten Datasets in der Übersicht

Da das extrahierte CSV-Dataset noch eine Reihe exakter Duplikate enthielt, wurden diese anschließend noch mit folgendem Bash-Script entfernt:

```
1 #!/bin/sh
2 sort -k2 -t, -u $1 > $1.unique.sorted
3 sort --random-sort $1.unique.sorted > $1.unique.random
```

Das resultierende CSV-Dataset enthält 15.567.518 Tupel und ist ca. 1,4 GB groß. Das impliziert eine durchschnittliche Tupel-Größe von ca. 89 Bytes. Aus diesem Dataset wurden zudem als weitere Datasets noch Untermengen mit zehntausend, hunderttausend, einer Million und zehn Millionen Tupeln extrahiert. Tabelle 6.1 zeigt alle Datasets, welche in den folgenden Experimenten als Eingabe-Datasets verwendet wurden.

6.2. Setup des Computerclusters

Für die Experimente wurde ein Computercluster eingerichtet, welches aus fünf gleichartigen Desktop-Computern als Knoten besteht. Jeder dieser Desktop-Computer ist vom Typ *Dell Optiplex 980* und verfügt über einen Intel Core i5-750 mit vier Kernen à 2,76 GHz, 8 GB Arbeitsspeicher und einer 500 GB SATA-II-Festplatte mit 7200 rpm. Die Knoten sind in einer Stern-Topologie über einen 1-Gigabit-Switch mit Full-duplex Ethernet verbunden.

Auch die Software auf allen Knoten ist einheitlich und besteht im Wesentlichen aus:

- Ubuntu Server 12.04
- Apache Spark 1.5.1
- Apache Hadoop 2.6.0

Um Konfigurations- und Performanz-Overhead zu reduzieren, wurde auf zusätzliche Software oder Virtualisierungsschichten zur bequemeren Verwaltung der Knoten verzichtet. Apache Spark läuft daher im *standalone mode* und verwendet entsprechend seinen integrierten Standalone-Cluster-Manager (siehe Abschnitt 2.4.4). Von Apache Hadoop wird lediglich das verteilte Dateisystem HDFS eingesetzt. Dies ist dabei derart konfiguriert, dass alle Daten auf jedem Knoten vorliegen und somit volle Redundanz bietet.

Name	Deployment	HDFS	Knoten	Spark-Executor		
	mode		Anzahl	Anzahl	Kerne	RAM
local	local	✗	1	1	4[1]	5 GB
cluster-1	client	✓	2	1	4	6 GB
cluster-2	client	✓	3	2	8	12 GB
cluster-3	client	✓	4	3	12	18 GB
cluster-4	client	✓	5	4	16	24 GB

[1] Die vier Kerne werden sich mit dem Driver-Prozess geteilt.

Tabelle 6.2.: Die verschiedenen Cluster-Konfigurationen in der Übersicht

Einer der fünf Knoten ist explizit als Spark-Master und HDFS-NameNode konfiguriert. Die übrigen Knoten sind als Spark-Worker und HDFS-DataNodes konfiguriert.

6.2.1. Konfigurationen der Cluster-Architektur

Die Experimente wurden mit fünf Konfigurationen hinsichtlich der Cluster-Architektur der darauf laufenden Spark-Plattform durchgeführt (siehe Abschnitt 2.4.4). Die verschiedenen Konfigurationen sind in Tabelle 6.2 aufgelistet und werden im folgenden als *Cluster-Konfigurationen* bezeichnet.

Bei vier der Cluster-Konfigurationen wird der *client mode* verwendet und die Spark-Applikation auf dem Master-Knoten gestartet (vgl. Abbildung 2.5b). Hinzu kommen entweder 1, 2, 3 oder 4 Worker-Knoten auf denen jeweils 6 GB Arbeitsspeicher für den Executor-Prozess reserviert sind. Für den Driver-Prozess auf dem Master-Knoten sind 2 GB Arbeitsspeicher reserviert. Als Persistenz-Schicht dient das HDFS.

Bei der fünften Cluster-Konfiguration wird gar kein richtiges Spark-Cluster gestartet, weil die Spark-Applikation auf einem der Cluster-Knoten im *local mode* ausgeführt wird (vgl. Abbildung 2.5a). Da sich in diesem Modus Driver- und Executor-Prozess die Ressourcen eines Knotens teilen, muss der explizit reservierte Speicher reduziert werden[2]. Während für den Driver-Prozess 2 GB reserviert bleiben (wie bei den anderen Konfigurationen), werden für den Executor-Prozess nur 5 GB reserviert. Als Persistenz-Schicht dient hier das lokale Dateisystem.

6.3. Experimente

An dieser Stelle werden einige Experimente dokumentiert, deren Ergebnisse Aufschluss über das Laufzeitverhalten von DaPo geben sollen. Den Experimenten lag eine voreingestellte Basis-Konfiguration zu Grunde, wobei einige dieser Parameter in den Experimenten gezielt variiert wurden. Das folgende Listing zeigt die wichtigsten voreinge-

[2] Auf jedem Knoten sollte Spark nicht mehr als 7 der 8 GB Arbeitsspeicher belegen, damit das Betriebssystem und andere Prozesse noch störungsfrei arbeiten können.

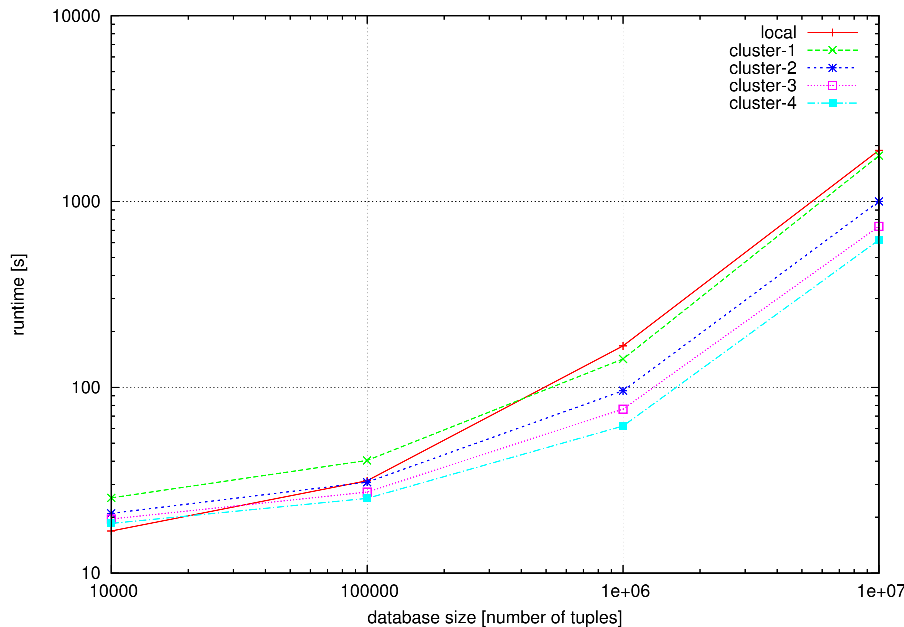


Abbildung 6.1.: Das Laufzeitverhalten der verschiedenen Cluster-Konfigurationen für unterschiedlich große Datasets

stellten Parameterwerte. Die Voreinstellungen der übrigen Parameter finden sich in Anhang A.2. Jede einzelne Messung wurde fünf mal durchgeführt und als Messpunkt die durchschnittliche Laufzeit aller fünf Durchläufe berechnet.

```
1 dapo.input.file = musicbrainz-unique-1000000.csv
2 dapo.duplicates.relative = 0.1 # Anteil der Duplikate
3 dapo.duplicates.clustersizes = 2 to 20 # Mögliche Clustergrößen
4 dapo.duplicates.distribution = equalpair # Dupl'cluster-Verteilung
5 dapo.duplicates.clustersim = 0.85 # Ziel-Ähnlichkeit
```

6.3.1. Variieren der Tupel-Menge

Im ersten Experiment wurde die Größe des zu verarbeitenden (und zu erzeugenden) Datasets variiert, um das Laufzeitverhalten und die Skalierbarkeit bzgl. der Eingabe-Größe sowie der Anzahl der Cluster-Knoten von DaPo zu untersuchen. Hierfür wurden die Laufzeiten für die MusicBrainz-Datasets der Größen zehntausend, hunderttausend, einer Million und zehn Millionen Tupel gemessen. Der Wert für die Ziel-Ähnlichkeit wurde auf 0,85 gesetzt, was in diesem Setup immer zu genau zwei Iterationen im Verschmutzungsprozess führte. Die Experimente wurden für alle Cluster-Konfigurationen durchgeführt.

Die Messergebnisse zeigen, dass DaPo problemlos alle verwendeten Dataset verarbeiten kann, sowohl lokal auf einem herkömmlichen Desktop-Computer, als auch auf den verschiedenen Konfigurationen des Computerclusters. So konnten lokal hunderttausend Tupel in einer halben Minute verarbeitet werden und zehn Millionen Tupel in etwas mehr als einer halben Stunde. Auf einem Spark-Cluster mit vier Worker-Knoten benötigte die

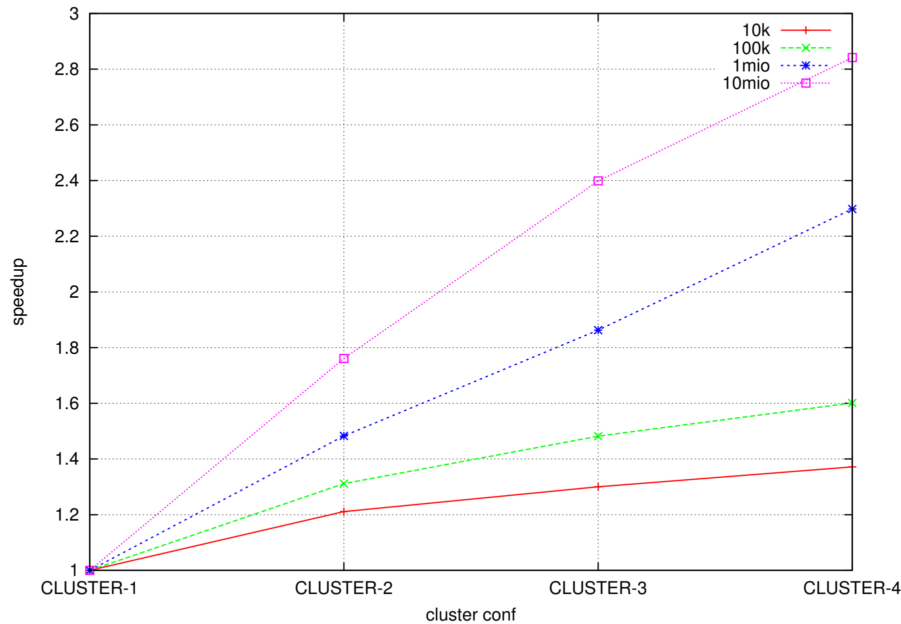


Abbildung 6.2.: Der Speedup von DaPo auf einem Spark-Cluster bei Hinzunahme zusätzlicher Worker-Knoten für unterschiedlich große Datasets

Verarbeitung von zehn Millionen Tupeln sogar nur 10,4 Minuten.

In Abbildung 6.1 sind die Größen der Datasets gegen die Laufzeiten aufgetragen. Dabei steht jede Linie im Graphen für eine Cluster-Konfiguration. Da sich die Tupel-Mengen in jedem Schritt um eine Zehnerpotenz erhöhen, werden logarithmische Skalen zur Basis 10 verwendet.

Für alle Cluster-Konfigurationen zeigt sich wie erwartet, dass sich bei Erhöhung der Tupel-Menge auch die Laufzeit erhöht. Bis zu einer Datenmenge von einer Million Tupel ist diese Steigerung noch sub-linear. Ab dem Schritt zum Dataset mit zehn Millionen Tupeln ist die Laufzeit-Erhöhung allerdings z. T. super-linear.

Beim Vergleich zwischen den verschiedenen Cluster-Konfigurationen lässt sich erkennen, dass sich bis hunderttausend Tupel der Einsatz eines Computerclusters noch nicht substanziell positiv auf die Laufzeit auswirkt – zum Teil ist das Gegenteil der Fall: Die Konfiguration *local* ist für den kleinsten Datensatz sogar am schnellsten. Erst ab einer Million Tupel sind deutliche Laufzeit-Gewinne durch zusätzliche Worker-Knoten zu erkennen.

Dieses Verhalten lässt sich auch gut in Abbildung 6.2 erkennen. Dieser Graph zeigt den jeweiligen Speedup bzgl. des Clusters mit einem Worker-Knoten bei Hinzunahme zusätzlicher Worker-Knoten. Die Linien repräsentieren die verschiedenen Eingabe-Datasets. Es zeigt sich: Je größer der Datensatz, desto größer der Speedup und folglich: desto lohnender die Hinzunahme weitere Worker-Knoten. Für die Datensätze mit zehn- und hunderttausend Tupeln ist nur noch ein relativ geringer Gewinn durch einen zusätzlichen Worker-Knoten zu erkennen.

Für den größten Datensatz ergeben sich zufriedenstellende Speedup-Werte von 1,8 für

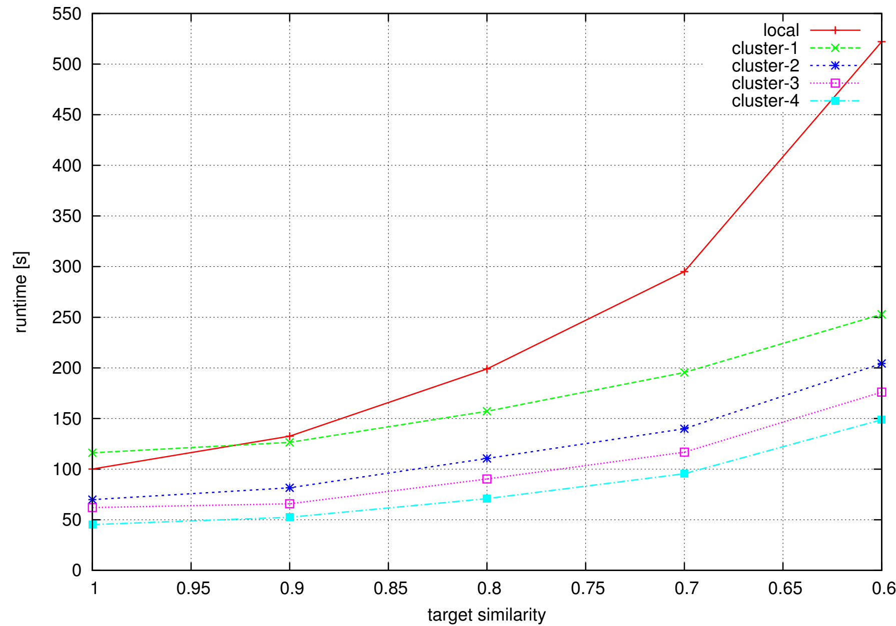


Abbildung 6.3.: Das Laufzeitverhalten der verschiedenen Cluster-Konfigurationen für unterschiedliche Werte der Ziel-Ähnlichkeit

zwei Knoten, 2,4 für drei Knoten und 2,8 für vier Knoten. Im Übrigen haben weitere (undokumentierte) Experimente angedeutet, dass der Speedup noch deutlich verbessert werden kann, wenn auf Spalten-Fehler verzichtet wird. Das ist auch naheliegend, da Spalten-Fehler zu Wide-Dependency-Operationen führen können, welche zusätzlichen Synchronisationsaufwand erzeugen.

6.3.2. Variieren der Ziel-Ähnlichkeit

In diesem Experiment wurde der Wert für die Ziel-Ähnlichkeit variiert. Dabei wurden Messungen für die Werte 1,0, 0,9, 0,8, 0,7 und 0,6 auf dem Dataset mit einer Million Tupel durchgeführt.

In Abbildung 6.3 ist die Ziel-Ähnlichkeit gegen die Laufzeiten aufgetragen. Erneut steht jede Linie im Graphen für eine der fünf Cluster-Konfiguration. Es wird deutlich, dass die Laufzeit überproportional gegenüber der sinkenden Ziel-Ähnlichkeit ansteigt. Das ist wie folgt zu erklären: Um die verschiedenen Ähnlichkeitswerte erzielen zu können, durchläuft DaPo unterschiedlich viele Iterationen des Fehlerschemas. Mit zunehmender Iterationszahl sinkt die Verschmutzungswirkung einer Iteration[3]. Aus diesem Grund steigt die Anzahl benötigter Iterationen überproportional gegenüber der sinkenden Ziel-Ähnlichkeit. Da die Laufzeit von der Anzahl der Iterationen abhängt, ist die Steigung im Graph also zunehmend. Auffällig ist, dass die Steigung bei der Cluster-Konfiguration *local* deutlich stärker ausfällt als bei den übrigen vier Konfigurationen, bei denen die Steigung jeweils sehr ähnlich ist.

[3] An diesem Effekt können verschiedene Aspekte beteiligt sein, wie z. B. der Charakter des Fehlerschemas, der Charakter der Daten und die verwendeten Ähnlichkeitsmaße.

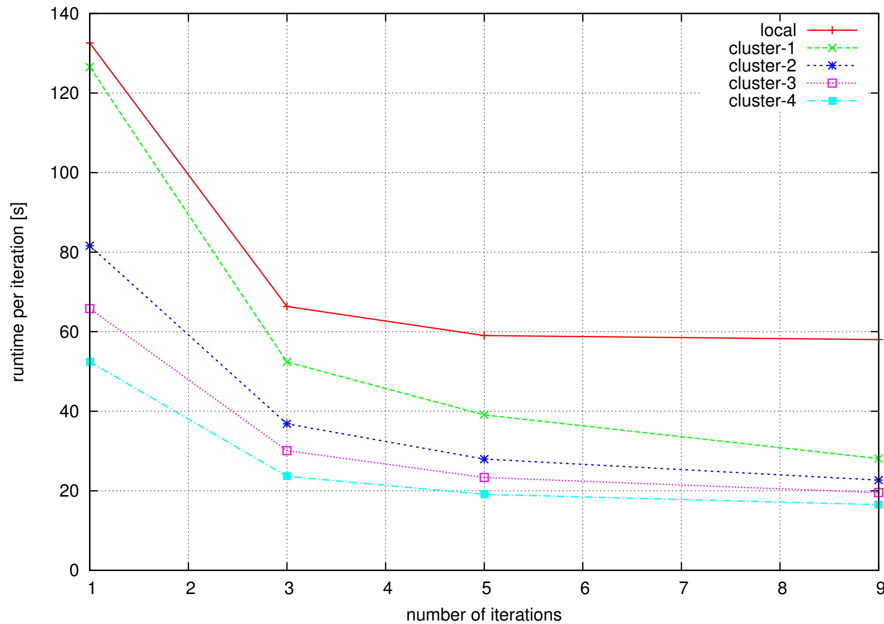


Abbildung 6.4.: Die Laufzeiten aus Abbildung 6.3 im Verhältnis zur Anzahl der jeweils benötigten Iterationen für die verschiedenen Cluster-Konfigurationen

In Abbildung 6.4 ist die Anzahl der benötigten Iterationen gegen die Laufzeiten pro Iteration aufgetragen. Hier wird deutlich, dass sich der Aufwand für eine zusätzliche Iteration einem konstanten Niveau (entsprechend der Leistung der Cluster-Konfiguration) annähert. Die Laufzeit pro Iteration ist bei Prozessen mit wenigen Iterationen deutlich größer als bei Prozessen mit vielen Iterationen, weil hier der konstante Overhead (Laden und Analyse des Datasets, Automatische Konfiguration, Speicherung) stärker ins Gewicht fällt.

6.3.3. Fazit

DaPo kann auf einem herkömmlichen Desktop-Computer problemlos Datasets mit zehn Millionen Tupeln verarbeiten. Um eine bessere Laufzeit zu erzielen, lohnt sich der Einsatz eines Computerclusters erst für Datasets ab einer Million Tupel[4]. Für kleinere Datasets reicht es aus DaPo einfach im *localmode* auf einem einzelnen Desktop-Computer zu betreiben.

Das Laufzeitverhalten von DaPo ist in etwa proportional zur Anzahl der durchlaufenen Iterationen. Da der maximale Speicherbedarf hauptsächlich abhängig von der Größe der Eingabe-Daten ist und daher während der Iterationen ungefähr konstant bleibt, können theoretisch beliebig viele Iterationen durchlaufen werden. Wie viele Iterationen überhaupt benötigt werden, hängt von dem verwendeten Fehlerschema und der gewählten Ziel-Ähnlichkeit ab.

[4]Diese Erkenntnis bezieht sich natürlich auf das verwendete Dataset, die verwendeten Reasoner (siehe Abschnitt 5) und die verwendete Basis-Konfiguration. Die Größe ab der sich ein Cluster lohnt, ist daher selbstverständlich von diesen Rahmenbedingungen abhängig.

6.4. Vergleich mit anderen Tools

In Kapitel 3 wurden einige Tools zur Erzeugung von Test-Daten für die Duplikaterkennung vorgestellt. Als Ergebnis ist dort eine Tabelle entstanden, welche diese Tools anhand bestimmter Aspekte gegenüberstellt. An dieser Stelle wird das in dieser Arbeit entstandene Tool DaPo in diesen Vergleich eingeordnet (siehe Tabelle 6.3). Die Messwerte entstammen der Cluster-Konfiguration *local*, so dass bezüglich der Hardware kein Vorteil entstanden ist. Zudem umfassen die Messwerte nicht nur die eigentliche Verschmutzung, sondern den gesamten Prozess, inklusive der Analyse-Phase und der Phase der automatisierten Vorkonfiguration.

Es wird deutlich, dass DaPo den anderen Tools in fast allen betrachteten Aspekten überlegen ist. Lediglich die fehlende GUI kann DaPo zum Nachteil gereicht werden. Zu beachten ist auch, dass DBGen zwar deutlich schneller ist als DaPo, dabei allerdings sehr unflexibel ist (nicht schemaunabhängig, wenig konfigurierbar) und zudem unrealistische Daten erzeugt (siehe Abschnitt 3.1). Vor allem ist DaPo aber das erste Tool, welches horizontal skalierbar ist und somit auf einem entsprechend großen Computercluster noch deutlich größere Datasets verarbeiten kann.

Tool	Erscheinungsjahr	Interaktion	Konfigurierbarkeit	Konf'aufwand (mind.)	Schemaunabhängigkeit	Ziel-Ähnlichkeit	horizontale Skalierbarkeit	Laufzeit 100k	Laufzeit 1mio
DBGen	1997	GUI/CLI	gering	gering	✗	✗	✗	5 s	7 s
Febrl	2008	CLI	gering	gering	✗	✗	✗	2 min	(4,2 min)[1]
GeCo	2012	Lib/Web-UI	hoch	hoch	✓	✗	✗	80 min	13,4 h
TDGen	2012	GUI/(CLI)	hoch	mittel	✓	✗	✗	4,5 min[2]	✗
ProbGee	2012	GUI	hoch	mittel	✓	✓	✗	5,9 h	✗
DaPo	2015	CLI	hoch	gering	✓	✓	✓	31 s	2,8 min

[1] terminiert nicht mit phonetischen Fehlern, daher ohne phonetische Fehler gemessen
[2] Um das Dataset überhaupt verarbeiten zu können, waren einige manuelle Optimierungen notwendig.

Tabelle 6.3.: Die Übersicht der in Kapitel 3 vorgestellten Tools, ergänzt um einen Eintrag für DaPo. Die Laufzeiten sind mit der Cluster-Konfiguration *local* gemessen.

7. Abschluss

In diesem Kapitel wird zunächst der Kern der Arbeit zusammengefasst und ein Fazit gezogen. Abschließend wird noch ein Ausblick auf mögliche zukünftige Erweiterungen gegeben.

7.1. Zusammenfassung & Fazit

Ziel dieser Arbeit war es, ein Verfahren zum Ableiten großer Test-Datasets aus vorhandenen Daten zu entwickeln, welches effizient, skalierbar, schemaunabhängig sowie flexibel konfigurierbar ist und dabei realistische Fehler produziert.

Zunächst wurde in Kapitel 3 eine Reihe von Tools zur Erzeugung von Test-Daten für die Duplikaterkennung vorgestellt, untersucht und verglichen. Dabei wurden insbesondere die Stärken und Schwächen dieser Tools herausgearbeitet. Es wurde deutlich, dass keines der untersuchten Tools den gestellten Anforderungen gerecht wird. Mit Blick auf diese Stärken und Schwächen wurde in Kapitel 4 ein allgemeines Verfahren konzipiert, welches die oben formulierten Anforderungen erfüllt. In Kapitel 5 wurde der *Data Polluter (DaPo)* vorgestellt, welcher eine Implementation des Verfahrens auf Basis von Apache Spark darstellt. Um Skalierbarkeit und Laufzeit-Verhalten von DaPo zu untersuchen, wurde eine experimentelle Evaluation des Tools durchgeführt und in Kapitel 6 beschrieben.

Zum Abschluss dieser Arbeit kann festgestellt werden, dass es gelungen ist ein Verfahren zum Ableiten großer Test-Datasets aus vorhandenen Daten zu entwickeln, welches die getroffenen Anforderungen erfüllt:

- **Effizienz & Skalierbarkeit:** Mit DaPo ist es möglich auf einem handelsüblichen Desktop-Computer Test-Datasets mit zehn Millionen Tupeln und einer Größe von ca. 1 GB in einer guten halben Stunde zu erzeugen. Zudem ist das Tool horizontal skalierbar, so dass die Hinzunahme weiterer Computer die Leistungsfähigkeit bezüglich Laufzeit und Datenmenge steigert.

- **Schemaunabhängigkeit:** Es ist möglich relationale Datasets mit beliebigem Schema zu verarbeiten. In DaPo müssen die Datasets derzeit als CSV-Datei vorliegen.

- **flexible Konfigurierbarkeit:** Auf der einen Seite bietet DaPo viele Möglichkeiten der Konfiguration. Auf der anderen Seite sorgen die Mechanismen der automati-

sierten Vorkonfiguration dafür, dass der benötigte Mindestaufwand für die Konfiguration äußerst gering ist.

- **realistische Fehler:** Das Verfahren bietet viele Ansatzpunkte, um die Auswahl und Verteilung der injizierten Fehler möglichst realistisch zu gestalten. In DaPo sind diese Mechanismen bisher nur beispielhaft implementiert und bieten somit noch Raum für Verbesserungen.

7.2. Ausblick

Bei der Erarbeitung des Verfahrens und der Implementation von DaPo sind eine Reihe von Aspekten in Erscheinung getreten, deren Erforschung bzw. Umsetzung für dieses Thema zwar interessant sind, jedoch den Rahmen dieser Arbeit gesprengt hätten. Im Folgenden werden einige dieser Aspekte vorgestellt, welche als Ansatzpunkte für zukünftige Maßnahmen dienen könnten.

7.2.1. Ansätze zur Verbesserung des Verfahrens

Um noch realistischere Fehler ableiten zu können, wäre es hilfreich bereits beim Data Profiling Abhängigkeiten zwischen Attributen erkennen zu können und Integritätsbedingungen aus dem Ursprungsschema zu übernehmen.

Das Konzept des abstrakten Datentyps soll helfen, ein Attribut über den Basis-Datentyp hinaus zu charakterisieren. Es hat sich allerdings herausgestellt, dass es schwierig ist sinnvolle abstrakte Datentypen zu definieren deren Charakteristika nicht überlappen, so dass auch eine eindeutige Typ-Zuordnung erfolgen kann. Aus diesem Grund ist es möglicherweise besser dieses Konzept durch eine flexiblere Lösung zu ersetzen. Eine Liste von typischen Charakteristika für jedes Attribut könnte hier Abhilfe schaffen. Bei der Ableitung eines Fehlerschemas könnte dann individuell auf diese Charakteristika eingegangen werden.

Bisher ist das Verfahren auf einzelne relationale Tabellen beschränkt. Es ist aber denkbar es auf komplexere relationale Datenbankschemata mit mehreren Tabellen auszuweiten oder auf nicht-relationale Datenformate wie z. B. XML oder JSON.

7.2.2. Ansätze zur Verbesserungen von DaPo

In DaPo sind bisher nur wenige Basis- & abstrakte Datentypen implementiert. Eine Ausweitung dieser Aspekte würde helfen die Eingabe-Daten differenzierter zu charakterisieren.

Um eine solche differenzierte Charakterisierung nutzen zu können, müssten auch die vorhandenen Fehler-Komponenten stärker ausdifferenziert sein. Weitere Subklassen für Feld-Fehler und generell weitere durchdachte Fehler-Komponenten würden dabei helfen die injizierten Fehler noch realistischer zu machen.

In DaPo sind an mehreren Stellen Reasoning-Systeme vorgesehen, die ausgehend vom Datenprofil verschiedene Parameter für den Verschmutzungsprozess vorkonfigurieren. Bisher existieren hier nur zwei sehr rudimentäre Beispiel-Implementierungen (`ErrorSchemaSimple`, `AttributeReasonerSimple`). Diese Stellen könnten prinzipiell auch mit ausgeklügelten Klassifizierern versehen werden, die z. B. zuvor durch Techniken des maschinellen Lernens trainiert wurden.

Die erzeugten Duplikate werden bislang zufällig in die Menge aller Tupel eingeordnet. Es ist allerdings denkbar die cluster-internen IDs der Duplikat-Tupel zu nutzen, um sie in irgendeiner Art gezielt an den Tupelgruppen auszurichten, um z. B. das Zugrundeliegen mehrerer Datenquellen zu simulieren.

In DaPo werden die Tupel nach ihrer Duplikatcluster-ID partitioniert. Es ist vorstellbar, dass die Partitionierung nach weiteren bzw. anderen Aspekten (z. B. nach Tupelgruppen) unter Umständen Vorteile mit sich bringen kann.

DaPo kann bisher nur CSV-Dateien lesen und erzeugen, aber gerade für den Umgang mit großen Datenmengen in verteilten Umgebungen wäre es wünschenswert auch mit (verteilen) Datenbanksystemen wie Cassandra oder HBase umgehen zu können. Der Aufwand für eine Unterstützung wäre vermutlich überschaubar.

Zum Abschluss ist anzumerken, dass die Verwendung des managed-Modus von DaPo alles andere als komfortabel ist, da manuell XML-Dateien manipuliert werden müssen. Für eine bessere Benutzbarkeit würde es sich daher anbieten eine graphische Benutzeroberfläche zu entwickeln, wie z. B. ein Web-Interface.

Literaturverzeichnis

[AGK10] ARASU, Arvind ; GÖTZ, Michaela ; KAUSHIK, Raghav: On active learning of record matching packages. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* ACM, 2010, S. 783–794

[BCC03] BAXTER, Rohan ; CHRISTEN, Peter ; CHURCHES, Tim: A comparison of fast blocking methods for record linkage. In: *ACM SIGKDD* Bd. 3 Citeseer, 2003, S. 25–27

[Bil03] BILENKO, Misha: *RIDDLE: Repository of Information on Duplicate Detection, Record Linkage, and Identity Uncertainty*. `http://www.cs.utexas.edu/users/ml/riddle/data.html`. Version: August 2003

[BR12] BACHTELER, Tobias ; REIHER, Jörg: Tdgen: A test data generator for evaluating record linkage methods / Technical Report wp-grlc-2012-01, German Record Linkage Center. 2012. – Forschungsbericht

[CC04] CHURCHES, Tim ; CHRISTEN, Peter: Blind data linkage using n-gram similarity comparisons. In: *Advances in Knowledge Discovery and Data Mining*. Springer, 2004, S. 121–126

[CG91] CHURCH, Kenneth W. ; GALE, William A.: Probability scoring for spelling correction. In: *Statistics and Computing* 1 (1991), Nr. 2, S. 93–103

[Chr09] CHRISTEN, Peter: Development and user experiences of an open source data cleaning, deduplication and record linkage system. In: *ACM SIGKDD Explorations Newsletter* 11 (2009), Nr. 1, S. 39–48

[Chr12] CHRISTEN, Peter: *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Science & Business Media, 2012

[Cor12] CORDTS, Sönke: *Datenqualität in Datenbanken*. mana-Buch, 2012

[CP09] CHRISTEN, Peter ; PUDJIJONO, Agus: Accurate synthetic generation of realistic personal information. In: *Advances in Knowledge Discovery and Data Mining*. Springer, 2009, S. 507–514

[CV13] CHRISTEN, Peter ; VATSALAN, Dinusha: Flexible and extensible generation and corruption of personal data. In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* ACM, 2013, S. 1165–1168

[DBEHS10a] DONG, Xin L. ; BERTI-EQUILLE, Laure ; HU, Yifan ; SRIVASTAVA, Divesh: Global detection of complex copying relationships between sources. In: *Proceedings of the VLDB Endowment* 3 (2010), Nr. 1-2, S. 1358–1369

[DBEHS10b] DONG, Xin L. ; BERTI-EQUILLE, Laure ; HU, Yifan ; SRIVASTAVA, Divesh: Solomon: Seeking the truth via copying detection. In: *Proceedings of the VLDB Endowment* 3 (2010), Nr. 1-2, S. 1617–1620

[DBES09] DONG, Xin L. ; BERTI-EQUILLE, Laure ; SRIVASTAVA, Divesh: Truth discovery and copying detection in a dynamic world. In: *Proceedings of the VLDB Endowment* 2 (2009), Nr. 1, S. 562–573

[DG08] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: simplified data processing on large clusters. In: *Communications of the ACM* 51 (2008), Nr. 1, S. 107–113

[DHI12] DOAN, AnHai ; HALEVY, Alon ; IVES, Zachary: *Principles of data integration*. Elsevier, 2012

[FW12] FRIEDRICH, Steffen ; WINGERATH, Wolfram: *Evaluation of tuple matching methods on generated probabilistic data*. Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, Universität Hamburg, Fachbereich Informatik, Masterarbeit, November 2012

[GH07] GUILLET, Fabrice ; HAMILTON, Howard J.: *Quality measures in data mining*. Bd. 43. Springer, 2007

[HKZ+15] HAMSTRA, M. ; KARAU, H. ; ZAHARIA, M. ; KONWINSKI, A. ; WENDELL, P.: *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Incorporated, 2015. – ISBN 9781449358624

[HM09] HASSANZADEH, Oktie ; MILLER, Renée J: Creating probabilistic databases from duplicated data. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 18 (2009), Nr. 5, S. 1141–1166

[HS95] HERNÁNDEZ, Mauricio A. ; STOLFO, Salvatore J.: The merge/purge problem for large databases. In: *ACM SIGMOD Record* Bd. 24 ACM, 1995, S. 127–138

[HS98] HERNÁNDEZ, Mauricio A. ; STOLFO, Salvatore J.: Real-world data is dirty: Data cleansing and the merge/purge problem. In: *Data mining and knowledge discovery* 2 (1998), Nr. 1, S. 9–37

[HSW07] HERZOG, Thomas N. ; SCHEUREN, Fritz J. ; WINKLER, William E.: *Data quality and record linkage techniques*. Springer Science & Business Media, 2007

[Jar89] JARO, Matthew A.: Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. In: *Journal of the American Statistical Association* 84 (1989), Nr. 406, S. 414–420

[Kau11] KAUL, Oliver B.: *Skalierbarkeit – Kurz & Gut*. 2011 `http://www.se.uni-hannover.de/pub/File/kurz-und-gut/ws2011-labor-restlab/RESTLab-Skalierbarkeit-Oliver-Beren-Kaul-kurz-und-gut.pdf`

[KKH+10] KIRSTEN, Toralf ; KOLB, Lars ; HARTUNG, Michael ; GROSS, Anika ; KÖPCKE, Hanna ; RAHM, Erhard: Data partitioning for parallel entity matching. In: *arXiv preprint arXiv:1006.5309* (2010)

[KTR12] KOLB, Lars ; THOR, Andreas ; RAHM, Erhard: Dedoop: efficient deduplication with Hadoop. In: *Proceedings of the VLDB Endowment* 5 (2012), Nr. 12, S. 1878–1881

[Lev66] LEVENSHTEIN, Vladimir I.: Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet physics doklady* Bd. 10, 1966, S. 707–710

[ME+96] MONGE, Alvaro E. ; ELKAN, Charles u. a.: The Field Matching Problem: Algorithms and Applications. In: *KDD*, 1996, S. 267–270

[Nau07] NAUMANN, Felix: Datenqualität. In: *Informatik-Spektrum* 30 (2007), Nr. 1, S. 27–31

[Nau14] NAUMANN, Felix: Data profiling revisited. In: *ACM SIGMOD Record* 42 (2014), Nr. 4, S. 40–49

[NH10] NAUMANN, Felix ; HERSCHEL, Melanie: *An Introduction to Duplicate Detection*. Bd. 3. Morgan & Claypool Publishers, 2010. – ISBN 978–1–608–45220–0

[Pan15] PANSE, Fabian: *Duplicate Detection in Probabilistic Relational Databases*. Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, Universität Hamburg, Fachbereich Informatik, Online Publication, 2015. `http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:gbv:18-228-7-211`. – URN urn:nbn:de:gbv:18–228–7–211

[RD00] RAHM, Erhard ; DO, Hong H.: Data cleaning: Problems and current approaches. In: *IEEE Data Eng. Bull.* 23 (2000), Nr. 4, S. 3–13

[RLOW15] RYZA, S. ; LASERSON, U. ; OWEN, S. ; WILLS, J.: *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, Incorporated, 2015. – ISBN 9781491912768

[SB02] SARAWAGI, Sunita ; BHAMIDIPATY, Anuradha: Interactive deduplication using active learning. In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2002, S. 269–278

[Sha05] SHAFRANOVICH, Yakov: Common Format and MIME Type for Comma-Separated Values (CSV) Files / RFC Editor. Version: October 2005. `http://www.rfc-editor.org/rfc/rfc4180.txt`. RFC Editor, October 2005 (4180). – RFC. – 1–8 S.. – ISSN 2070–1721

[Tal11] TALBURT, John R.: *Entity resolution and information quality*. Elsevier, 2011

[TKM02] TEJADA, Sheila ; KNOBLOCK, Craig A. ; MINTON, Steven: Learning domain-independent string transformation weights for high accuracy object identification. In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2002, S. 350–359

[TVC13] TRAN, Khoi-Nguyen ; VATSALAN, Dinusha ; CHRISTEN, Peter: GeCo: an online personal data generator and corruptor. In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* ACM, 2013, S. 2473–2476

[VCL10] VERNICA, Rares ; CAREY, Michael J. ; LI, Chen: Efficient parallel set-similarity joins using MapReduce. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* ACM, 2010, S. 495–506

[Ver69] VERHOEFF, J: Error detecting decimal codes, vol. 29, Math. In: *Centre Tracts. Math. Centrum Amsterdam* (1969)

[Wil15a] WILCKE, Niklas: DduP–Towards a Deduplication Framework utilising Apache Spark. In: *BTW workshops*, 2015, S. 253–262

[Wil15b] WILCKE, Niklas: *Scalable Duplicate Detection utilizing Apache Spark*. Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, Universität Hamburg, Fachbereich Informatik, Masterarbeit, September 2015

[Win90] WINKLER, William E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990)

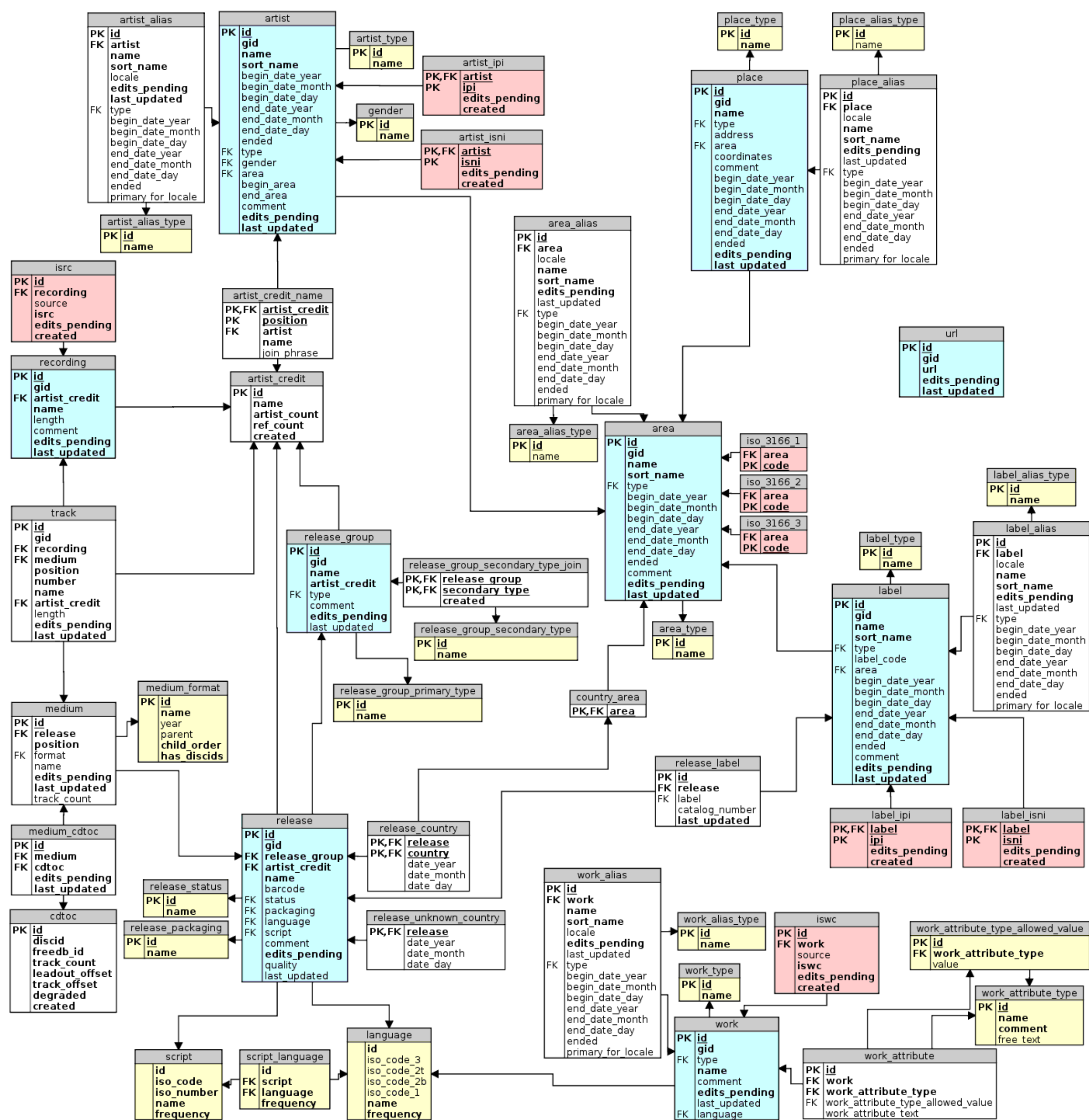
[ZCD+12] ZAHARIA, Matei ; CHOWDHURY, Mosharaf ; DAS, Tathagata ; DAVE, Ankur ; MA, Justin ; MCCAULEY, Murphy ; FRANKLIN, Michael J. ; SHENKER, Scott ; STOICA, Ion: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX*

conference on Networked Systems Design and Implementation USENIX Association, 2012, S. 2–2

[ZCF+10] ZAHARIA, Matei ; CHOWDHURY, Mosharaf ; FRANKLIN, Michael J. ; SHENKER, Scott ; STOICA, Ion: Spark: cluster computing with working sets. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* Bd. 10, 2010, S. 10

A. Anhang

A.1. Schema der MusicBrainz-Datenbank



A.2. Beispiel einer Basis-Konfiguration

```
# Spark Config
##############
# Options: spark://... | local[4]
spark.conf.master=spark://vsispool20:7077

# General
##############
# Options: hdfs://vsispool20:9000 ... | file:///...
dapo.basePath = hdfs://vsispool20:9000
dapo.verbose = true
# Options: target | source
dapo.tupleCount.of = target
dapo.tupleCount = 1000000

# Input
##############
dapo.input.file = ${dapo.basePath}/musicbrainz-unique-1000000.csv
dapo.input.firstLine = 1

# Duplicates
##############
# Options: relative | absolute | determined
dapo.duplicates.method = relative
dapo.duplicates.relative = 0.1
# if 'absolute': dapo.duplicates.absolute = 100
# if 'determined': dapo.duplicates.determined = 100x2 10x3 1x4
# Options: equaltuple | equalpair | equalcluster
dapo.duplicates.distribution = equalpair
dapo.duplicates.clustersizes = 2 to 20
dapo.duplicates.clustersim = 0.85

# Output
##############
dapo.output.tableSchemaFile = ${dapo.input.file}.tableschema.xml
dapo.output.errorSchemaFile = ${dapo.input.file}.errorschema.xml
dapo.output.tmpPath = ${dapo.basePath}/_tmp
dapo.output.file = ${dapo.input.file}.dapo
dapo.output.separator = ,
dapo.output.header = true
dapo.output.prepedTid = true
dapo.output.prepedCid = true
```

A.3. Ableiten eines abstrakten Datentyps

```
// adp is the examined AttributeDataProfile object
val abstractType: AbstractDataType.Value = basicDataType match {
  case BasicDataType.String =>
    if(adp.stats(StatsType.TokenCounts).mean >= 10){
      AbstractDataType.Text
    }else if(adp.stats(StatsType.TokenCounts).mean >= 1.2){
      AbstractDataType.WordSequence
    }else{
      AbstractDataType.Word
    }
  case BasicDataType.Int =>
    if(adp.stats(StatsType.AttributeValues).min > 1500 &&
       adp.stats(StatsType.AttributeValues).max < 2500){
      AbstractDataType.Year
    }else{
      null // dummy: space to add more logic...
    }
  case BasicDataType.Double =>
    null // dummy: space to add more logic...
  case BasicDataType.Boolean =>
    null // dummy: space to add more logic...
  case _ =>
    null
}
```

Listing A.1: Die Ableitung des abstrakten Datentypen in der Klasse AttributeReasonerSimple (siehe Abschnitt 5.8.1)

A.4. Liste der implementierten Fehler-Komponenten

Im folgenden sind die bisher in DaPo implementierten Fehler-Komponenten dokumentiert. Zum besseren Verständnis wurden einige der Fehler-Komponenten (insbesondere die Feld-Fehler) mit Beispielen versehen.

A.4.1. BlockErrorComponents

Ein `BlockErrorComponent`-Objekt repräsentiert einen Tupelgruppen-Fehler. Die Tupelgruppe ist im Parameter `condition: String` durch eine Bedingung über das Tabellenschema definiert.

StandardBlockError führt erst eine `RowErrorComponent` und anschließend eine `ColErrorComponent` aus.

RowBlockError führt eine `RowErrorComponent` aus.

ColBlockError führt eine `ColErrorComponent` aus.

A.4.2. ColErrorComponents

Ein `ColErrorComponent`-Objekt repräsentiert einen Spalten-Fehler. Über den Parameter `fraction: Double` kann festgelegt werden, wie groß der Anteil der von diesem Fehler betroffenen Tupel sein soll.

SubstituteFieldsInCol ersetzt die Attributwerte zufällig ausgewählter Tupel durch andere zufällig ausgewählte Werte dieses Attributs.

TransposeFieldsInCol vertauscht die Attributwerte zufällig ausgewählter Tupel miteinander.

ShiftFieldsInCol verschiebt die Werte dieses Attributes um eine "Zeile" nach unten. Der Attributwert des ersten Tupels wird auf `null` gesetzt.

A.4.3. RowErrorComponents

Ein `RowErrorComponent`-Objekt repräsentiert einen Spalten-Fehler.

FieldErrorOnRow wendet eine `FieldErrorComponent` auf alle Attributwerte dieses Tupels an.

InputFormatErrorOnRow wendet einen `InputFormatError` auf alle Attributwerte dieses Tupels an.

TransposeFieldsInRow vertauscht zwei zufällig ausgewählte Felder innerhalb einer Tupels.

SubstituteFieldsInRow ersetzt den Wert eines zufällig ausgewählten Attributes durch den Wert eines anderen zufällig ausgewählten Attributes desselben Tupels.

ShiftFieldsInRow verschiebt die Attributwerte innerhalb des Tupels um ein Feld nach rechts. Der erste Wert wird `null`.

RedoubleFieldsInRow überschreibt den Wert eines zufällig ausgewählten Feldes durch den Wert eines direkten Nachbarfeldes.

MergeFieldsInRow konkateniert zwei benachbarte Attributwerte, speichert den neuen Wert in dem einen Feld und leert das andere Feld.

FieldErrorSchema erlaubt für jedes Attribut eine individuelle `FieldErrorComponent` zu definieren (Meta-Fehler).

A.4.4. FieldErrorComponents

Ein `FieldErrorComponent`-Objekt repräsentiert einen Feld-Fehler und verarbeitet somit einen String.

ConfusionMatrixTypo fügt einen typischen Tippfehler ein, welcher auf (austauschbaren) Konfusionsmatrizen [CG91, FW12] basiert. Solch ein Fehler setzt sich aus dem Entfernen, Einfügen, Ersetzen und Vertauschen von Buchstaben zusammen.
Beispiel.

eigentlich ⟶ *eingentlich (insertion)*
eigentlich ⟶ *eigenlich (deletion)*
eigentlich ⟶ *eigemtlich (substitution)*
eigentlich ⟶ *eigenltich (transposition)*

HomophoneError fügt einen typischen phonetischen Fehler ein. Basiert auf einem (austauschbaren) Konfusionsset aus [FW12].
Beispiel.

Schmitt ⟶ *Schmidt (char substitution)*
Mainz ⟶ *meins (word substitution)*

OCRError fügt einen typischen Fehler der optischen Texterkennung (OCR) ein. Basiert auf einem (austauschbaren) Konfusionsset aus [BR12].
Beispiel. *Müller* → *Mü11er*

InsertRandomBlank fügt an zufälliger Position der Zeichenkette ein Leerzeichen ein.
Beispiel. *Schmidt* → *Sc hmitd*

RemoveDigits entfernt alle numerischen Zeichen aus der Zeichenkette.
Beispiel. *Mauerweg 43* → *Mauerweg*

RemoveNonDigits entfernt alle nicht-numerischen Zeichen aus der Zeichenkette.
Beispiel. *Mauerweg 43 → 43*

ToLower wandelt alle Buchstaben in der Zeichenkette in Kleinbuchstaben um.
Beispiel. *Mauerweg 43 → mauerweg 43*

ToUpper wandelt alle Buchstaben in der Zeichenkette in Großbuchstaben um.
Beispiel. *Mauerweg 43 → MAUERWEG 43*

Abbreviate kürzt eine Zeichenkette ab, indem nur das erste Zeichen beibehalten und ein Punkt angehängt wird.
Beispiel. *Schmidt → S.*

ToASCII wandelt die Zeichenkette ins ASCII-Format um. Über den Parameter `normalize: Boolean` kann angegeben werden, ob dabei Zeichen (wenn möglich) normalisiert[1] werden sollen (Default: `true`).
Beispiel.

1. *Müller → Muller (mit `normalize=true`)*
2. *Müller → Mller (mit `normalize=false`)*

EncodingError kodiert die Zeichenkette mit einem Charset, welches über den Parameter `encoding: String` festgelegt wird.
Beispiel. *Müller → MÃ¼ller*

ToEmptyString ersetzt die Zeichenkette durch eine leere Zeichenkette.
Beispiel. *Müller →*

ToNull ersetzt die Zeichenkette durch `null`.
Beispiel. *Müller → `null`*

InputFormatError ist ein zusammengesetzter Fehler, welcher genau einen der Feld-Fehler aus folgender Menge ausführt:
{`OCRError`, `ToASCII`, `EncodingError`, `RemoveDigits`, `RemoveNonDigits`, `ToLower`, `ToUpper`, `OmitAllDelimiters`, `ReplaceDelimiters`}

WordErrorComponents

WordError ist zusammengesetzt aus Feld-Fehlern, welche sich besonders für die Ausführung auf einzelnen Wörtern eignen. Diese Menge sieht wie folgt aus:
{`Abbreviate`, `ConfusionMatrixTypo`, `EncodingError`, `HomophoneError`, `InsertRandomBlank`, `OCRError`, `RemoveDigits`, `RemoveNonDigits`, `ToASCII`, `ToLower`, `ToUpper`}

[1] siehe https://docs.oracle.com/javase/7/docs/api/java/text/Normalizer.html

WordSequenceErrorComponents

Die folgenden Fehler behandeln Zeichenketten als Wortfolgen. Dafür wird jede Zeichenkette an bestimmten Trennzeichen (Leerzeichen, Bindestrich und Unterstrich) aufgeteilt und in eine Menge aus Wörtern und Trennzeichen (*Delimiter*) überführt. Diese Menge wird nun (dem Fehler entsprechend) manipuliert und zurück in eine Zeichenkette umgewandelt.

OmitRandomWord löscht ein zufälliges Wort der Wortfolge
Beispiel. *Hans Peter Müller → Hans Müller*

OmitFirstWord löscht das erste Wort der Wortfolge
Beispiel. *Hans Peter Müller → Peter Müller*

KeepFirstWord behält nur das erste Wort der Wortfolge bei.
Beispiel. *Hans Peter Müller → Hans*

KeepLongestWord behält nur das längste Wort der Wortfolge bei.
Beispiel. *Hans Peter Müller → Müller*

SubstituteNeighborWords ersetzt ein Wort durch ein Nachbarwort
Beispiel. *Hans Peter Müller → Hans Hans Müller*

TransposeNeighborWords vertauscht zwei benachbarte Wörter
Beispiel. *Hans Peter Müller → Peter Hans Müller*

RedoubleWord verdoppelt ein zufälliges Wort der Wortfolge
Beispiel. *Hans Peter Müller → Hans Peter Peter Müller*

ForEachWord wendet einen Wort-Fehler auf jedes einzelnes Wort der Wortfolge an.
Beispiel. *Hans Peter Müller → H. P. M. (hier:* `Abbreviate`*)*

ForRandomWord wendet einen Wort-Fehler auf ein zufälliges Wort der Wortfolge an.
Beispiel. *Hans Peter Müller → Hans P. Müller (hier* `Abbreviate` *auf das zweite Wort)*

OmitRandomDelimiter löscht einen zufälligen Delimiter.
Beispiel. *Hans-Peter Müller → HansPeter Müller*

OmitAllDelimiters löscht alle Delimiter.
Beispiel. *Hans-Peter Müller → HansPeterMüller*

ReplaceDelimiters ersetzt einen Delimiter-Typ durch einen anderen Delimiter-Typ.
Beispiel. *Hans-Peter Müller → Hans Peter Müller*

ShiftRandomDelimiter verschiebt die Delimiter.
Beispiel. *Hans-Peter Müller → Hans Peter-Müller*

WordSeqError ist zusammengesetzt aus allen übrigen Wortfolgen-Fehlern.

NumerErrorComponents

Die folgenden numerischen Fehler behandeln Zeichenketten als `Int`. Die Behandlung von weiteren numerischen Typen wie `Double` oder `Float` ist noch nicht implementiert.

NumericTransposition wählt zufällig zwei Ziffern aus und vertauscht sie miteinander. Hat die Zahl mehr als zwei Ziffern sind die beiden Zahlen mit einer Wahrscheinlichkeit von 92% Nachbarn. Mit einer Wahrscheinlichkeit von 8% liegt eine weitere Ziffer dazwischen.
Beispiel. *123 → 213*

NumericSubstitution wählt zufällig eine Ziffer oder Ziffernfolge aus und ersetzt sie durch eine weitere. Dabei sind kürzere Ziffernfolgen wahrscheinlicher als längere.
Beispiel. *123 → 122*

NumericHomophone Ersetzt eine Ziffer oder Ziffernfolge durch eine ähnlich klingende Ziffer bzw. Ziffernfolge.
Beispiel. *17 → 70*

NumericJumpingTwin ersetzt zwei identische Ziffern durch dieselbe andere Ziffer
Beispiel. *131 → 232*

NumericTypo ist zusammengesetzt aus den übrigen numerischen Fehlern und folgt der Verteilung aus [Cor12].

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den __________ Unterschrift: _________________________