Institut für Parallele und Verteilte Systeme

Masterarbeit Nr. 92

Erklärung fehlender Ergebnisse bei der Verarbeitung hierarchischer Daten in Spark

Karsten Mayer

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Melanie Herschel
Betreuer/in:	Prof. Dr. Melanie Herschel
Beginn am:	21. März 2016
Beendet am:	27. September 2016
CR-Nummer:	H.4

Kurzfassung

Es existieren einige Algorithmen, die Entwicklern bei der Fehlersuche bei einer Datenbankanfrage helfen. Diese Arbeiten beantworten, wieso bestimmte Daten nicht in der Ergebnismenge für eine Anfrage vorhanden sind oder bestimmte nicht erwartete Daten in der Ergebnismenge erscheinen (Why-not-Frage). Für Anfragesprachen, die hierarchische Daten unterstützen, bestehen bisher aber nur wenige Arbeiten.
In dieser Arbeit wird untersucht, welche Besonderheiten es für Why-not-Fragen bei hierarchischen Daten gibt. Dazu wird betrachtet, welche besonderen Fragestellungen dafür möglich sind und wie diese geeignet beantwortet werden können. Dabei wird auch ein konkreter Algorithmus für Python entworfen und implementiert. Anhand von diesem kann mit Hilfe eines Beispiels untersucht werden, ob der Algorithmus effizient und effektiv genug ist Why-not-Fragen zu beantworten.

Inhaltsverzeichnis

1. Einleitung **7**

2. Verwandte Arbeiten **13**
2.1. Why-not-Erklärungen 14
2.2. Why-not-Algorithmen 15
2.3. Datenmodelle 15
2.4. Spark 17

3. Grundlagen und Definitionen **19**
3.1. Besonderheiten durch hierarchische Daten 19
3.2. Besonderheiten durch Spark 19
3.3. Formale Definition 20
3.4. Definitionen für den Algorithmus 21

4. Fragen für hierarchische Daten und Spark **27**
4.1. Klassifizierung der Fragen 27
4.2. Mögliche Fragestellungen 28
4.3. Zusammenfassung 42

5. Algorithmus **45**
5.1. Algorithmus 45
5.2. Abdeckung der Fragestellungen 56
5.3. Einschränkungen des Basisalgorithmus 58

6. Prototyp **63**
6.1. Eingabe 63
6.2. Ausgabe 64
6.3. Vorgehen 64
6.4. Einschränkungen 65

7. Evaluation **67**
7.1. Ziele der Evaluation 67
7.2. Testdaten 68
7.3. Beispiele 68
7.4. Laufzeitanalyse 86

8. Zusammenfassung und Ausblick **95**

A. Dateien **97**
A.1. Beispielanfragen aus der Evaluation 97
A.2. Adressdaten für das erste Beispiel der Evaluation 101
A.3. Personendaten für das erste Beispiel der Evaluation 101
A.4. Seriendaten für das erste Beispiel der Evaluation 101

Abkürzungsverzeichnis **103**

Literaturverzeichnis **105**

1. Einleitung

In der klassischen Softwareentwicklung ist es selbstverständlich, etliche Hilfsmittel für die Fehlerfindung zur Verfügung zu haben. Dazu zählen statische und dynamische Codeanalyse sowie Debugging. Im Bereich der Datenbanken ist dies bisher nicht üblich, obwohl auch bei Daten und Datenbanken schnell komplexe und fehleranfällige Anfragen zustande kommen können.
Um Nutzer bei der Korrektur einer fehlerhaften Datenanfrage zu unterstützen, gibt es verschiedene Ansätze. Dazu zählen die Why-Not-Algorithmen, die das Fehlen von bestimmten erwarteten Daten in der Ausgabe einer Datenanfrage erklären. Bestehende Algorithmen sind, neben Why-Not, [9] z. B. NedExplain [5] und Conseil [19]. Allerdings sind diese auf relationale Datenbankmodelle spezialisiert. Die drei genannten Arbeiten benutzen eine anfragenbasierte Erklärung (Query-based explanation). Dies ist eine Menge von Operatoren, die dafür verantwortlich sind, dass erwartete Tupel nicht in dem Ergebnis vorhanden sind. In dieser Arbeit wird der Why-Not-Algorithmus so erweitert, dass er auch für das hierarchische Datenmodell eingesetzt werden kann. Dazu wird Spark[1] und dessen DataFrames bzw. Resilient Distributed Database (RDD) benutzt.
Ein Beispiel für eine fehlerhafte Anfrage wird im Folgenden gezeigt.

Beispiel 1: Abbildung 1.1 zeigt die hierarchischen Adressdaten. Dabei stellt ein Tupel eine Stadt dar, bestehend aus einem Namen und verschiedenen Adressen. Die Adressen bestehen aus der ID, Straße und Nummer. Die Nummer steht dabei für die Hausnummer, die ID identifiziert die Adresse eindeutig.
Auf der anderen Seite stehen die persönlichen Daten (siehe Abbildung 1.2). Ein Tupel stellt dabei ein Volk dar und hat die zugehörigen Personen als Kinder. Dabei hat eine Person einen Vornamen, einen Nachnamen und eine AdressId.
Die hier benutzte Anfrage ist in Abbildung 1.5 dargestellt. Für alle Bewohner des kleinen gallischen Dorfes werden die Volkszugehörigkeit sowie Vor- und Nachname ausgegeben. Dazu werden die entsprechenden Personen aus JSON-Dateien (JavaScript Object Notation) mit den bereits beschriebenen Eingabedaten gewonnen. Danach wird die Hierarchie der gespeicherten Personen mit der Funktion *explode* des Moduls "pyspark.sql.functions" aufgelöst. Dies bedeutet, dass jede Person, statt als Kind eines Volkes, als extra Tupel dargestellt wird. Diese Personendaten werden dann mit den ebenfalls entschachtelten Adressdaten aus einer JSON-Datei mit dem

[1] http://spark.apache.org/

$root$
$|--Stadt : struct(nullable = true)$
$| \quad |--Adress : string(nullable = true)$
$| \quad |--element : struct(containsNull = true)$
$| \quad | \quad |--ID : string(nullable = true)$
$| \quad | \quad |--Nummer : string(nullable = true)$
$| \quad | \quad |--Stra„e : string(nullable = true)$
$| \quad |--name : string(containsNull = true)$
$| \quad |--zip : string(containsNull = true)$

Abbildung 1.1.: Schema der Adressdaten für das Beispiel

$root$
$|--Volk : struct(nullable = true)$
$| \quad |--Personen : array(nullable = true)$
$| \quad | \quad |--element : struct(containsNull = true)$
$| \quad | \quad | \quad |--AdressID : string(nullable = true)$
$| \quad | \quad | \quad |--Nachname : string(nullable = true)$
$| \quad | \quad | \quad |--Vorname : string(nullable = true)$
$| \quad |--name : string(nullable = true)$

Abbildung 1.2.: Schema der Personendaten für das Beispiel

Schema aus Abbildung 1.1 verknüpft. Dabei werden alle Adressen herausgefiltert, die nicht in dem kleinen gallischen Dorf liegen. Die konkreten Eingabedaten sind in den Abbildungen 1.3 und 1.4 dargestellt.
Aufgrund eines Fehlers innerhalb der *Join*-Operation werden nicht Daten verknüpft, bei denen die Adresse gleich ist, sondern alle Daten, bei denen die AdressID einer Person identisch zu der Hausnummer einer Adresse ist. Somit wird in der Ausgabe nicht, wie erwartet, Asterix und Obelix erscheinen. In diesem Fall könnte ein Algorithmus den Benutzer darauf hinweisen, dass bei dem *Join*-Operator die entsprechenden Daten verloren gehen. Dadurch wird dem Entwickler der Anfrage ein guter Hinweis gegeben, wo er mit der Suche nach dem Fehler beginnen sollte.

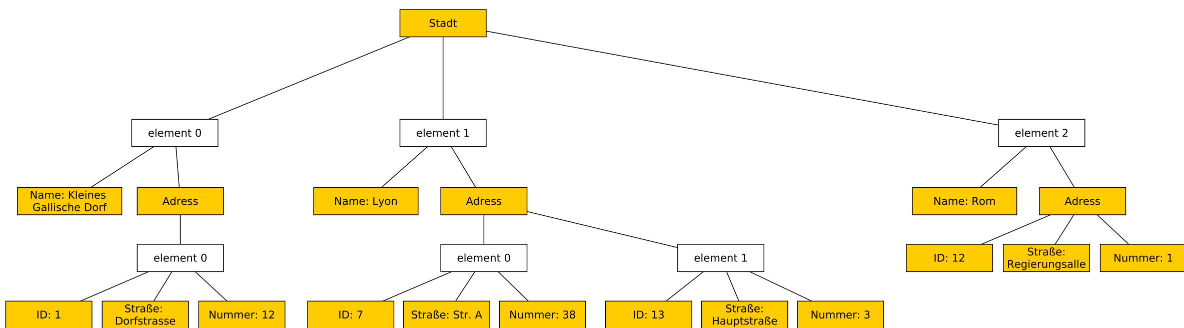

Abbildung 1.3.: Adressdaten für das Beispiel. Die weißen Labels stellen kein eigenes Attribut dar, sondern dienen zur Trennung der verschiedenen Elemente eines ArrayTypes.

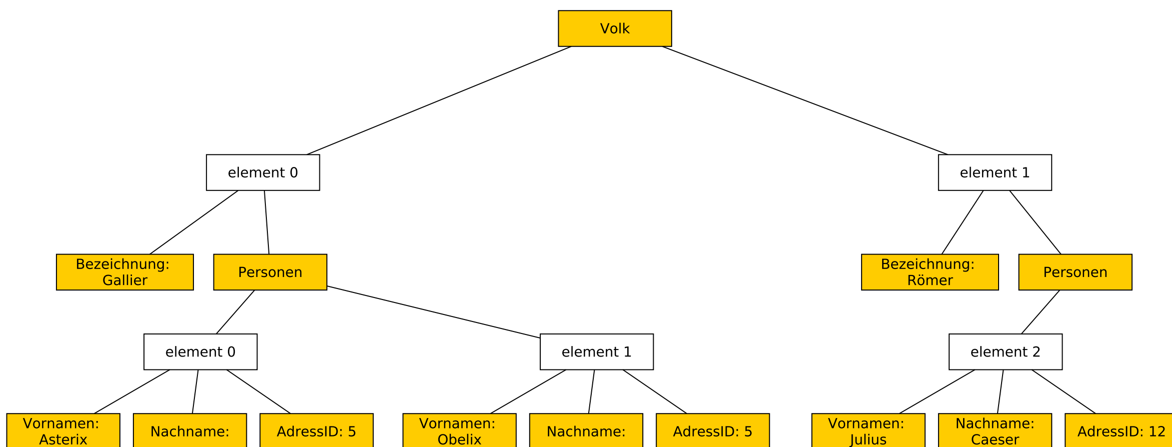

Abbildung 1.4.: Personendaten für das Beispiel. Die weißen Labels stellen kein eigenes Attribut dar, sondern dienen zur Trennung der verschiedenen Elemente eines ArrayTypes.

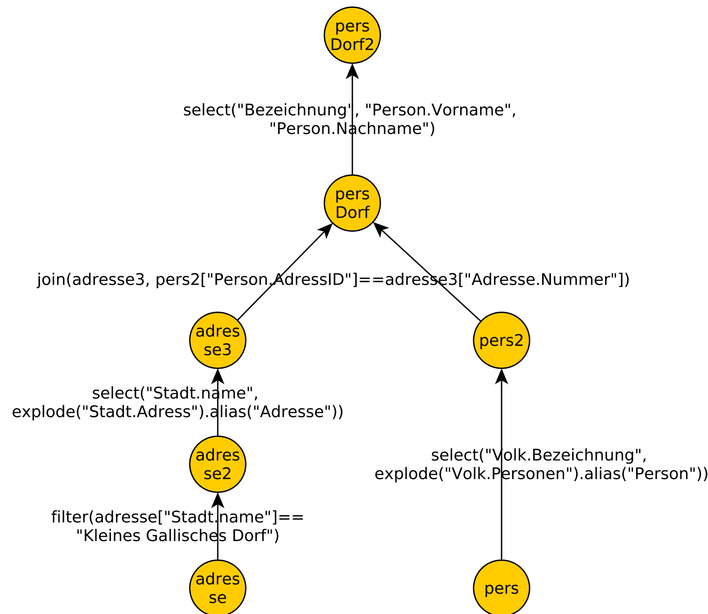

Abbildung 1.5.: Darstellung des Ablaufs der Operationen in Beispiel 3.2 als Baum

Ohne die Unterstützung von Why-not-Algorithmen kann die Suche nach Fehlern in komplexen Anfragen, die viele Operatoren enthalten, sehr zeitaufwändig und fehlerhaft verlaufen. Automatisierung hat den Mehrwert, die Suche zu beschleunigen, und hilft zu einer systematischen Vorgehensweise. Konkret wird dem Entwickler mitgeteilt, wo er mit der Fehlersuche anfangen muss. Bereits bestehende Algorithmen sind zum Beispiel NedExplain und Why-not. Aufbauend auf diese bereits bestehenden werden mit dieser Arbeit folgende Beiträge geleistet:

- Neue Why-Not-Fragestellungen, gegeben durch hierarchische Daten
- Alternative Semantik der Ein- und Ausgabe, gegeben durch ein reicheres Datenmodell
- Erweiterung des Algorithmus, um diese Fragestellungen beantworten zu können
- Evaluation der Effektivität und Effizienz des Algorithmus

In dieser Arbeit werden zuerst in Kapitel 2 verwandte Arbeiten vorgestellt. Im nächsten Kapitel werden wichtige Definitionen eingeführt. Anschließend wird in Kapitel 4 betrachtet, welche neuen Fehlermöglichkeiten und Fragen durch hierarchische Daten und Spark entstehen können. Dabei wurden etliche Fragen aufgestellt, die sich entweder auf das Schema oder die Daten beziehen. Gleichzeitig wurde untersucht, für welche Fragen es sinnvoll und möglich ist, diese durch einen Algorithmus zu beantworten.
Aufbauend auf diesen Grundlagen wird im Kapitel 5 ein Algorithmus erstellt, der die Fragen beantwortet. Dabei muss jede Operation einzeln implementiert werden. Zusätzlich sind noch weitere Einschränkungen vorhanden, hauptsächlich durch die Mächtigkeit von Spark.

Abschließend wird der entwickelte Algorithmus anhand eines Prototyps (Kapitel 6) und eines Beispiels evaluiert.

2. Verwandte Arbeiten

Diese Arbeit gehört in den Bereich der Provenance (deutsch. Herkunft). Eine aktuelle Beschreibung [28] ist, dass Provenance Meta-Daten bezeichnet, die den Entstehungsprozess eines Endprodukts beschreiben. Für diese Arbeit ist besonders die sogenannte Datenherkunft (Engl. Data Provenance) relevant, die sich mit der Beschreibung der Entstehung von Daten im Ergebnis einer Datenmanipulation befasst. Dadurch dass heute viele Internetquellen oder Informationen aus verschiedenen Quellen benutzt werden, bei denen die Vertrauenswürdigkeit nicht immer bekannt ist, hat die Datenherkunft an Bedeutung gewonnen [11]. Für diese Arbeit ist vor allem die Herkunft relevant, die die Beziehung zwischen den Daten in der Datenquelle und den Daten im Ergebnis beschreiben.
Cheney et al. [11] erklären, dass die verbreitetsten Bezeichnungen dieser Datenherkunft die Why-Provenance, How-Provenance und Where-Provenance sind.
Why-Provenance beschreibt, welche Ursprungsdaten für das Erscheinen von bestimmten Daten im Ergebnis verantwortlich sind.
How-Provenance beschreibt, wie bestimmte Daten im Ergebnis aus den Quelldaten erstellt werden.
Where-Provenance beschreibt die Beziehung zwischen der Speicherstelle, den Ergebnisdaten und den Quelldaten.
Eine Arbeit, die sich mit Spark und Datenherkunft beschäftigt, stammt von Interlandi et al [22]. Die Autoren haben die Bibliothek Titian erstellt, diese erlaubt das Zurückverfolgen der Herkunft vorhandener Daten. So können die RDD Operationen nicht nur normal ausgeführt werden, sondern auch rückwärts ausgeführt werden. Dies kann dem Benutzer bei der Suche nach Fehlern in Anfragen helfen.
Ein weiteres Problem, das meist dem Gebiet der Datenherkunft zugeordnet wird, ist die Fragestellung, warum Daten in dem Ergebnis einer Anfrage fehlen, die der Benutzer erwartet hatte. Die Frage wird häufig Why-not-Frage (Why-not Question) genannt und das Ergebnis Why-not Erklärung (Why-not explanation).
In diesem Kapitel werden zuerst Algorithmen zur Berechnung solcher Why-Not-Erklärungen vorgestellt. Für zwei dieser Ansätze (Conseil und Why-not) wird der Algorithmus in Kapitel 2.2 näher beschrieben. Im darauffolgenden Unterkapitel 2.3 werden Datenmodelle eingeführt, dabei liegt der Fokus auf den relationalen und hierarchischen Datenbankmodellen. Als letztes wird Spark vorgestellt im Abschnitt 2.4.

2.1. Why-not-Erklärungen

Bidoit et al. [3] kategorisieren die vorhandenen Algorithmen nach der Art, wie die Antwort auf die Why-not-Frage aussieht. So gibt es Ansätze, die anfragebasiert (Query-based) agieren. Dies bedeutet, dass diese Algorithmen zu erklären versuchen, aufgrund welcher Abschnitte/Bedingungen einer Frage bestimmte Tupel nicht in der Ergebnismenge einer SQL Anfrage erscheinen. Algorithmen mit anfragenbasierten Erklärungen (Query-based explanation) sind TED [4], NedExplain [5], Why-Not [9] und TED++. TED++ wird in [3] beschrieben. Der Algorithmus berechnet eine anfragebasierte Erklärung. TED++ benutzt Polynome zur Darstellung der Gründe für das Fehlen der Tupel einer SQL-Anfrage. TED++ und NedExplain sind für das relationale Datenbankmodell entwickelt worden. Die Beschreibung der Datenherkunft relationaler Daten wird in [15] beschrieben und wird allgemein als How-Provenance bezeichnet. Ted++ ist eine Erweiterung des ursprünglichen Ted Algorithmus, es wurde insbesondere die Laufzeit zur effizienten Berechnung der Polynome optimiert.
Der Why-not-Algorithmus wird in Kapitel 2.2.1 näher erläutert.
Desweiteren gibt es die instanzbasierte Erklärung (Instance-based explanation) (MA [21], Artemis [20], Meliou et al.[27], Calvanese et al. [7]). Dabei soll erklärt werden, wie die Eingabedaten geändert werden müssten, damit die erwarteten Tupel ausgegeben werden.
Weitere Ansätze sind ontologiebasierte Erklärung (Cate et al. [8]) und Refinement-based Erklärung. Bei einer Refinement-based Erklärung wird eine neue Anfrage erstellt, bei deren Ergebnis die fehlenden Tupel vorhanden sind. Dazu zählen die Algorithmen ConQueR [32], TALOS [31], FlexIQ [23], WQRTQ [14], die Arbeit von He und Lo [17], die Arbeit von Islam et al. [24] und die Arbeit von Chen et al. [10]
Cate et al. [8] wollen, mit Hilfe von Ontologien, eine verständliche Erklärung geben, wieso ein Tupel im Ergebnis fehlt. Dadurch kann die Why-not-Frage verständlich und möglichst allgemein gültig beantwortet werden.
Conseil [19] verbindet verschiedene Ansätze und wird deswegen als Hybrid explanation bezeichnet. Es wird in Kapitel 2.2 genauer beschrieben.
Bhowmick et al. [2] haben einen Algorithmus entworfen, um zu erklären, wieso bestimmte Bildmotive bei der Bildersuche mit Tag-based Social Image Retrieval nicht in der Ergebnismenge sind. Damit soll der Benutzer in der Lage sein bestimmte, erwartete Bilder mit einer neuen Anfrage zu finden. Dazu benutzt der Algorithmus verschiedene Ansätze, diese sind das Ändern des Rankings, das Entfernen von Suchbegriffen und das Hinzufügen von Suchbegriffen mit Hilfe einer externen Datenquelle.
Es gibt auch ähnliche Arbeiten, die nicht zu dem Bereich der Why-not-Fragen gehören. Dazu zählt eine Arbeit von Baid et al. [1], bei dieser wird untersucht, wieso bei Schlüsselwortsuche über strukturierte Daten bei einer Anfrage kein Ergebnis gefunden wird.

2.2. Why-not-Algorithmen

Conseil

Conseil wird als Hybrid explanation bezeichnet und in [19] näher beschrieben. Dabei werden instanz- und anfragebasierte Ansätze gemischt.
Als Eingabe wird ein 5-tuple $(E, Q, Q(D), D, C)$ benutzt. E ist das fehlende Ergebnis, Q eine Menge von Anfragen, Q(D) ist das Ergebnis der Anfrage über eine bestimmte Quelle und C sind Beschränkungen über die vier anderen Elemente des Tupels.
Der Algorithmus erstellt dazu als erstes einen Baum für die Anfrage und eine "generic witness". Dies wird im zweiten Schritt mit passing Attributen annotiert. Im nächsten Schritt wird eine Menge von Ableitungen (derivation) erstellt. Im letzten Schritt wird eine Hybride Erklärung für jede Ableitung erstellt. Diese Erklärungen sind gleichzeitig das Ergebnis von Conseil. Falls in einem Zwischenschritt notwendige Daten für das erwartete Ergebnis nicht mehr vorhanden sind, werden diese in dem Schritt automatisch erstellt.
Es werden sowohl anfragebasierte Erklärungen als auch instanzbasierte Erklärungen erstellt. Eine anfragenbasierte Erklärung ist eine Menge von Operatoren, die dafür verantwortlich sind, dass erwartete Tupel nicht in dem Ergebnis vorhanden sind.
Die instanzbasierte Erklärung erstellt Modifikationen in den Daten, die dazu führen, dass das erwartete Ergebnis durch die Anfrage erzeugt wird.

2.2.1. Why-not

Chapman und Jagadish [9] haben ein Why-not-Algorithmus basierend auf Workflows entwickelt. Im relationalen Bereich entspricht dies den Operationen der Datenbankanfrage. Why-not sucht Picky Manipulationen und Frontier Picky Manipulationen. Eine Manipulation ist Picky, wenn ein im Ergebnis fehlender Datensatz oder dessen Nachfolger in der Eingabe vorhanden ist, aber nicht in der Ausgabe. Die Manipulation ist Frontier Picky, wenn mindestens ein Datensatz Picky ist und kein Datensatz existiert, der auch in einem Nachfolger vorhanden ist. In der Arbeit wurden sowohl ein Top-down als auch Bottom-up Ansatz entwickelt, um die Frontier Picky Manipulation zu finden. Diese geben dem Benutzer einen guten Einstieg zur Fehlersuche.

2.3. Datenmodelle

Es gibt für Datenbanken verschiedene Datenbankmodelle. Eines davon ist das relationale Datenbankmodell. In diesem Modell werden die Daten in verschiedenen Relationen gespeichert. Neben dem weit verbreiteten relationalen Modell existieren weitere Datenbankmodelle, dazu

zählt das hierarchische Datenbankmodell. In diesem Modell werden die Daten in hierarchischer Form gespeichert. Weitere Datenmodelle sind das Netzwerkmodell, objektorientiertes Modell und objektrelationale Datenbank [29]. In diesem Kapitel wird das relationale Datenbankmodell, das von den bisherigen Why-not-Algorithmen benutzt wird, und das hierarchische Datenbankmodell, das für diese Arbeit benutzt wird, näher beschrieben.

Relationales Datenbankmodell

Das relationale Datenbankmodell speichert die Daten in verschiedenen Relationen (Tabellen). Dabei besteht eine Relation aus mehreren Datensätzen, sogenannten Tupel (Zeilen). Die Relation legt fest, welche Attribute (Spalten) das Tupel einer Relation jeweils hat. Ein Attribut ist dabei ein atomarer Attributwert. Dieser kann zum Beispiel ein String, ein Datum oder eine Zahl sein. Jede Relation hat einen Schlüssel, dies sind Attribute mit deren Hilfe jedes Tupel eindeutig identifiziert werden kann. Die verschiedenen Relationen können mit Hilfe von Fremdschlüsseln verbunden werden.
Das relationale Datenbankmodell ist heute das weitverbreitetste Datenbankmodell [29]. Als Anfragesprache für eine relationale Datenbank wird meistens SQL benutzt.

Hierarchisches Datenbankmodell

Beim hierarchischen Datenbankmodell werden die Daten baumartig angeordnet. Dies bedeutet, dass alle Attribute, außer die Wurzel, ein Elternelement haben und einem Knoten in einem Baum entsprechen. Ein Knoten kann ein oder mehrere Kinder haben. Es gibt Anwendungen, bei denen das hierarchische Datenbankmodell eine festgelegte Struktur hat. In diesem Fall ist es genau festgelegt, wie der Baum für einen Datensatz aussehen muss. Auch kann es sein, dass der Datentyp für ein Element festgelegt wird.
Ein großer Vorteil der hierarchischen Daten ist der schnelle Datenzugriff durch die Verlinkung zwischen den Daten mit der Eltern-Kind Verbindung. Nachteile kann es bei dem Einfügen von komplexen Daten geben. Zudem kann das Modell zu redundanter Datenspeicherung führen [18].
Das hierachische Datenbankmodell wird heute hauptsächlich in XML eingesetzt. Auch Spark und die Anfragesprache PigLatin [1] unterstützen die Benutzung von geschachtelten bzw. hierarchischen Daten.
In 3.3 werden die hierarchischen Daten für Spark formal definiert.

[1] https://pig.apache.org/docs/r0.14.0/basic.html

2.4. Spark

Spark wurde 2009 durch ein Forschungsprojekt im UC Berkeley RAD LAB gegründet. Inzwischen ist Spark ein Open-Source-Projekt der Apache Software Foundation. Es ist eine Cluster Computing Plattform [25]. Es wurde für die Verarbeitung von großen Datenmengen entworfen [16]. Spark besteht aus verschiedenen Komponenten, diese sind Spark Core, Spark SQL, Spark Streaming, MLib (Machine Learning Library) und GraphX.
Spark Core beinhaltet die Basisfunktionen von Spark. Dazu zählen unter anderem Aufgabenplanung, Interaktion mit dem Speicher und die Programmierschnittstelle der RDDs (resilient distributed datasets).
Spark SQL ist ein Paket, das für strukturierte Daten geeignet ist, und es bietet eine Syntax, die fast identisch zu SQL ist.
Spark kann mit vielen Datenquellen umgehen, dabei hat es kein eigenes Speichersystem. Spark kann zum Beispiel die Daten aus JSON-Dateien einlesen.

Resilient Distributed Datasets (RDD)

ist nach Karau et al. [25], eine verteilte Sammlung von Elementen. Demnach verteilt Spark automatisch die Daten und Operationen auf verschiedene Knoten des Clusters. Zur Erstellung eines RDDs können externe Daten geladen werden. Auf RDDs können zwei Typen von Operationen durchgeführt werden. Dies sind:

- **actions (Aktionen)**, bei diesen wird mit Hilfe einer Operation über ein RDD ein Ergebnis errechnet, welches in ein externes Speichersystem oder im Treiberprogramm gespeichert wird [25].
- **transformations (Transformationen)**, sie erstellen mit dem Ergebnis einer Operation aus einem vorherigen RDD ein neues RDD. Bestehende RDDs werden nicht verändert, es wird immer ein neues erstellt [25]

Die Daten eines RDD werden in verschiedene Partitionen aufgeteilt, diese können dann jeweils auf verschiedenen Cluster ausgeführt werden, falls mehr als ein Rechner zur Verfügung steht. Der Datentyp RDD ist als abstrakte Klasse implementiert. Die RDD Operationen werden nicht direkt ausgeführt, stattdessen werden nur die Transformationen gespeichert. Die eigentliche Ausführung findet erst statt, wenn eine action Operation ausgeführt wird oder die Daten des RDD gespeichert werden [16].
Die Operationen innerhalb der RDD Klasse haben zum Teil auch Funktionen als Übergabeparameter. Spark unterstützt somit Funktionen höherer Ordnung. Diese Funktionen sind in der Abbildung 2.1 dargestellt. Die Daten innerhalb eines RDDs brauchen kein definiertes Schema. [16]

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V,V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T,T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Abbildung 2.1.: Die Funktionen höherer Ordnung, die von Spark unterstützt werden.

Ein RDD ist in Spark ein abstraktes Objekt[2], das Basisoperationen zur Verfügung stellt. Zusätzlich gibt es noch konrekte RDD Objekte, die zusätzlich Spezialfunktionen enthalten. Dazu gehören PairRDDFunctions[3] und DoubleRDDFunctions[4].
PairRDDFunctions enthält spezielle Funktionen für Schlüssel-Wert Paare. Dazu zählt zum Beispiel die Operation groupByKey. Bei dieser muss das RDD genau zwei Spalten haben. Es werden alle Werte der zweiten Spalte (Wert) unter einem neuen Attribut zusammengefasst. Dieses Attribut bildet zusammen mit einem Attribut, das den Schlüssel enthält, ein neues Tupel. Somit existiert für jeden Schlüssel nur noch ein Wert und es wurde bei der Wert Spalte eine neue Hierarchieebene eingeführt.
DoubleRDDFunctions stellt Funktionen zur Verfügung, die nur für RDDs aus doublen Werten zur Verfügung stehen.

DataFrame

DataFrame[5] speichert organisierte Daten in Spalten. Dabei haben die Spalten einen eindeutigen Namen und festgelegte Datentypen.
Data Frames unterstützen einige wichtige Funktionen, die auch von RDDs unterstützt werden. Es ist auch möglich mit Hilfe eines Data Frames ein RDD zu erstellen. Auch umgekehrt ist es möglich aus einem RDD ein Data Frame zu erstellen. Wobei das Schema entweder explizit angegeben werden muss (Operation createDataFrame) oder Spark SQL implizit versucht das Schema zu erstellen (Operation toDF) [16].

[2]http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD
[3]http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions
[4]http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.DoubleRDDFunctions
[5]https://spark.apache.org/docs/1.4.0/api/java/org/apache/spark/sql/DataFrame.html

3. Grundlagen und Definitionen

In diesem Kapitel werden für diese Arbeiten wichtige Grundlagen erörtert und Definitionen eingeführt. Dazu zählt die Betrachtung der Herausforderungen, die sich für das Beantworten von Why-not-Fragen mit hierarchischen Daten und Spark ergeben. Zudem wird das hierarchische Datenmodell von Spark, das bei der Benutzung von JSON entstehen kann, formal definiert. Danach wird das Ziel des zu entwickelnden Algorithmus beschrieben. Dazu kommen die Ein- und Ausgabe des Algorithmus sowie notwendige Definitionen.

3.1. Besonderheiten durch hierarchische Daten

Bei hierarchischen Daten werden die Daten in Baumform und nicht in Tabellenform gespeichert.
Dies führt zu einigen Änderungen für den Algorithmus. Dadurch gibt es bei den Daten auch noch Unterschiede, ob Knoten Kinder haben und wie viele. Dies wirkt sich auch auf die Fragestellung aus. So gibt es auch Fragen, die sich auf die Hierarchie beziehen. So ist nicht nur die Frage wieso ein Tupel nicht existiert, sondern auch wieso existiert ein bestimmtes Kind innerhalb des Datenbaums nicht.

3.2. Besonderheiten durch Spark

Spark ist nicht speziell als reine Datenbankanfragesprache gedacht. Es werden auch andere Funktionen für Daten zur Verfügung gestellt, wie zum Beispiel Map Reduce.
RDD und DataFrames bieten eine Reihe von Funktionen. Einige davon (z. B.: map) erlauben das Übergeben von Funktionen, die dann auf die Daten angewandt werden. Spark unterstützt mehr Funktionen als relationale Datenbanken, darunter auch komplexe Funktionen. Auch das Schema kann sich häufig während einer Anfrage ändern. Zusätzlich erschwert wird es durch die Funktionen höherer Ordnung (siehe Kapitel 2.4) bei denen der Benutzer beliebige Funktionen einsetzen kann.
Problematisch ist dabei die Zurückverfolgung der Daten (Lineage Tracing). Die Daten müssen von den Ergebnisdaten über die Zwischenergebnisse bzw. Operatoren zu Eingabedaten zurückverfolgt werden. Um das Problem der Funktionen höherer Ordnung zu lösen, könnte eine inverse Funktion genutzt werden. Dies wird von Cui und Widom [12] beschrieben. Eine

zweite Möglichkeit wäre es, bei der Ausführung der Anfrage entsprechende Informationen zu sammeln und mit Hilfe von diesen die Herkunft zu verfolgen. Diesen Ansatz verfolgt Interlandi et al. [22]. Eine weitere Herausforderung ist, dass durch diese Funktionen eventuell sehr viele unterschiedliche Eingaben möglich sind. (Bsp. Wörter eines Textes zählen).

3.3. Formale Definition

In dieser Arbeit wird JSON als Eingabe benutzt. Es hat Ähnlichkeiten mit dem Schema der DataFrames. Eine genaue Definition [6] für JSON-Objekte wurde von dem Internet Engineering Task Force (IETF) 2014 veröffentlicht. Allerdings gibt es im Vergleich zwischen dem Schema von DataFrames in Spark und dem Schema von JSON kleine Unterschiede. Die Datentypen, die von DataFrames unterstützt werden, werden in [30] beschrieben.
Im Folgenden wird das Schema formal definiert. Es werden die Unterschiede zwischen JSON-Eingabedateien und DataFrames erklärt, sowie auf ihre Auswirkungen untersucht.

Das DataFrame besteht dabei aus $T_1, T_2, ..., T_n$ Tupel.
Jedes der Tupel hat einen Wurzelknoten **root**, der n $\in \mathbb{N}$ Typen als Kinder hat.
Formal gilt:
Datenbank :== $\emptyset | r_1, r_2, ..., r_n \ mit \ n \in \mathbb{N}$
root :== $RecordType$
Type :== *label*, *childAttribut* | *childAttribut* $\in$ {**Simple Type**, **ArrayType**, **RecordType**}
Simple Type :== *wert*|*null*, *nullable* $mit \ wert \in$ **Datentyp** $\wedge$ *nullable* $\in$ boolean
ArrayType :== *containsNull*, $\emptyset$ | {*childelement*} mit *containsNull*, *nullable* $\in$ boolean, *childElement* $\in$ **RecordType**
RecordType :== $t_1, t_2, ..., t_n \mid n \in \mathbb{N} \wedge$ schema$(t_o) =$ schema(t_u) mit $o \neq u \wedge o, u \in \mathbb{N} \wedge \forall k \in \mathbb{N} \ t_k \in$ **Type** $mit \ k \leq n \wedge \forall i, j \in \mathbb{N} \ Type(label \ t_i) \neq Type(label \ t_j) \ mit \ (i \neq j \wedge i, \ j \leq n)$
Datentyp :== Numeric Datentyp | StringType | BinaryType | BooleanType | DateTimeType
Numeric Datentyp :== ByteType | ShortType | IntegerType | LongType | FloatType | DoubleType | DecimalType
schema(r) :== (label$_1$, **Type**(label$_1$)), (label$_2$, **Type**(label$_2$)), ..., (label$_n$, **Type**(label$_n$)) | $r \in$ **RecordType**

Die Tupel eines DataFrames müssen alle dasselbe Schema besitzen. Es ist zwar, wie im JSON-Standard festgelegt [6], möglich, dass in einer JSON-Eingabedatei für ein RecordType verschiedene Typen existieren (verschiedene RecordTypes), allerdings ergänzt Spark bei allen Elementen des Arrays die entsprechenden fehlenden Typen und füllt diese mit dem Wert null. Somit besitzen alle Elemente des DataFrames dasselbe Schema. Auch die Reihenfolge der Eingabedatei spielt keine Rolle.
DataFrames erlauben zwar zusätzlich MapTypes, die in dieser Arbeit allerdings nicht berücksichtigt werden, weil JSON als Eingabedatei vorausgesetzt ist. (**MapType** :==

$\{keyType, valueType, valueContainsNull\}|keyType \in \mathbf{Type} \wedge valueType \in \mathbf{Type} \wedge valueContainsNull \in$ boolean)
Der JSON-Standard erlaubt für values (hier wäre es RecordType) auch die Werte *false*, *true*, *null* sowie ein String oder Integer. Eine solche Eingabedatei wird von Spark nicht korrekt eingelesen und deswegen hier nicht weiter berücksichtigt.
Die Typen des root, ArrayType und RecordType, sind somit durch das Schema vorgegeben. Bei dem ArrayType werden die Elemente in einer Sequenz gespeichert, sodass die Reihenfolge relevant ist.
Außerdem unterstützt JSON nur weniger Datentypen, diese sind String (entspricht StringType) und Number (entspricht Numeric Datentyp).

3.4. Definitionen für den Algorithmus

In dieser Arbeit wird ein anfragenbasierter Why-not-Algorithmus für hierarchische Daten entwickelt. Der Algorithmus soll einem Benutzer zeigen, welcher Operator bei einer Anfrage dafür verantwortlich ist, dass das Ergebnis anders ist als erwartet. Eine Anfrage, für die untersucht wird, wieso das Ergebnis nicht wie erwartet ist, wird Debugging-Szenario genannt. Die Anfrage ist dabei eine Sparkanfrage, bestehend aus einer Reihe von DataFrame-Operationen bzw. RDD Operationen, die implizit oder explizit in ein DataFrame umgewandelt werden. Funktionen höherer Ordnung und Funktionen mit nicht deterministischem Ergebnis dürfen dabei nicht in der Anfrage vorkommen.

3.4.1. Eingabe

Die Ein- und Ausgabe eines Debugging-Szenarios soll auf eine bestehende Ein- und Ausgabe-Definition aufbauen. Dies bedeutet, dass die Ein- und Ausgabe eines in Kapitel 2 vorgestellten Algorithmus für hierarchische Daten angepasst werden soll. Die Eingabe ist hierbei die Anfrage, bei der das erwartete Ergebnis fehlt. Dabei muss es sich um eine Sparkanfrage aus einer Reihe von Operatoren auf DataFrames handeln. Auch Operationen auf RDD sind erlaubt, solange diese explizit oder implizit in ein DataFrame umgewandelt werden können. Zur Eingabe gehört zusätzlich die Fragestellung.

Definition 3.4.1 (Eingabe eines Debugging-Szenarios)
Die Eingabe eines Debugging-Szenarios ist ein 2-Tupel (E, Q(D)). Dabei steht E für das fehlende erwartete Ergebnis, dargestellt durch ein geschachteltes Tupel.
Für die Darstellung der erwarteten Tupel E wird die Syntax aus Definition 3.4.2 benutzt.
Q ist die Anfrage, bestehend auf Operationen über RDD und DataFrames. Die Datenquelle muss implizit in der Anfrage Q definiert werden. Dies geschieht dadurch, dass jede Operation, die keine Vorgänger hat, eine Operation sein muss, die Daten einliest. Für diesen Algorithmus ist JSON als Eingabe vorgesehen.

Da das erwartete Ergebnis baumförmig ist, wird eine Notation eingeführt, die es ermöglicht entsprechende Tupel an query tree pattern angelehnt zu definieren.
Nach Lakshmanan et al. [26] besteht ein tree pattern query aus einem Teil von XPath. Tree pattern query unterstützt child(/), descendant(//) und branching ([]).
Child (a / b) gibt an, dass b ein Kind von a ist. Descendant (a // b) bedeutet, dass b ein Nachkomme von a ist, unabhängig davon an welcher Stelle er dies im Anfragebaum ist [33]. Dieser Operator wird von dem Why-not query tree nicht unterstützt. Ein Anfragebaum stellt den Ablauf der DataFrame in Abhängigkeit zu den entsprechenden Operationen dar. Ein Anfragebaum ist in Abbildung 3.3 dargestellt. Branching ([]) bedeutet, dass definiert wird, welchen Wert ein Attribut des Baumes haben muss, was mit eckigen Klammern geschieht. Dabei wird der Wert für ein Kindattribut mit bestimmten Namen definiert, z. B Stadt[name='Kirchheim' AND Einwohnerzahl>2000]/Strasse. Letzteres gibt an, dass der Name der Stadt Kirchheim sein muss und es mindestens 2000 Einwohner haben muss.

Definition 3.4.2
*[Why-not tree pattern query] Why-not tree pattern query ist eine Darstellung für hierarchische Tupel. Dabei werden einzelne Pfade von der Wurzel bis zu einem Blatt definiert und mit AND zusammengefügt. Für jeden Pfad werden die Eltern-Kind Attribute mit / getrennt. Dabei werden jeweils die Namen des Attributs angegeben. Falls ein Attribut ein SimpleType ist und einen Wert haben kann, wird der Wert mit [] direkt hinter dem Attributnamen angegeben. Der Wert kann dabei ein Integer, ein String, eine Variable oder eine Bedingung sein. Falls ein Attribut ein ArrayType ist, muss dies vor dem Attribut mit einem zusätzlichen / gekennzeichnet sein. Falls die Anzahl der Kinder beliebig ist muss ein neuer Pfad eingesetzt werden, der nach dem ArrayType * in eckigen Klammern angegeben wird. Mit runden Klammern wird festgelegt, dass die Kinder eines ArrayTypes zu demselben childElement gehören. Dabei können geschachtelte Klammern benutzt werden, falls es mehrere Ebenen eines ArrayTypes gibt. Verschiedene Tupel werden durch geschweifte Klammern voneinander getrennt.*

Für Attributwerte können auch Variablen definiert werden. Das bedeutet, dass das Attribut einen beliebigen Wert haben kann. Dabei kann auch eine Variable für mehrere Werte benutzt werden.
Zusätzlich wird * als Wert für ein childElement eines ArrayTypes eingeführt. Es steht für eine beliebige Anzahl von Attributen, wobei mögliche zusätzliche Geschwister dadurch nicht mehr relevant sind.
Für SimpleTypes kann entweder ein konkreter Wert oder eine Bedingung vorgegeben werden (=, <, > und !=). Der Wert wird in eckigen Klammern hinter den Attributen spezifiziert (Attribut[«Wert»]).
Es wird davon ausgegangen, dass das Attribut nur mit den angegebenen Werten vorkommen darf. Wenn ein Attribut auch noch andere Werte haben darf, muss ein neues erwartetes Tupel mit AND festgelegt werden. Allerdings wird bei fehlenden Werten für SimpleTypes davon ausgegangen, dass der zugehörige Typ einen beliebigen Wert annehmen kann. Die Operatoren sind in der Tabelle 3.1 dargestellt.

Tabelle 3.1.: Operatoren der Why-not tree pattern query

Operator	Bedeutung
/	Trennt ein Kindattribut von dessen Elternattribut (das Elternattribut ist ein RecordType)
//	Trennt ein Kindattribut von dessen Elternattribut (das Elternattribut ist ein ArrayType)
AND	Verbindet zwei Pfade eines Baums miteinander oder trennt zwei verschiedene Tupel
[bzw.]	Gibt direkt hinter einen Attribut (SimpleType) dessen Wert an
(bzw.)	Trennt verschiedene Elemente eines ArrayTypes
{ bzw. }	Trennt verschiedene Tupel voneinander
*	Gibt in eckigen Klammern direkt hinter einem ArrayType an, dass es nicht relevant ist, wie viele Kinder für dieses ArrayType existieren

Mit dieser Eingabe ist es möglich, alle gefundenen relevanten Fragen zu beantworten. Dies wird in Kapitel 4 näher beschrieben.

Beispiel 2: In Abbildung 3.2 wird die Anfrage aus Abbildung 1.5 erweitert, indem die Personen nicht als einzelne Tupels ausgegeben werden, sondern als ArrayType für jedes Dorf.
In dem Dorf sollten zumindest Asterix und Obelix wohnen, wobei noch beliebig viele andere Personen möglich sind. Die Syntax dieses erwarteten Ergebnisses sieht wie folgt aus:

```
Dorf["Kleines Gallisches Dorf"] AND Einwohner//data/Vorname["Asterix]
AND Einwohner//data/Vorname["Obelix"] AND Einwohner//data[*]
```

In Abbildung 3.1 ist das erwartete Ergebnis als Baum dargestellt.

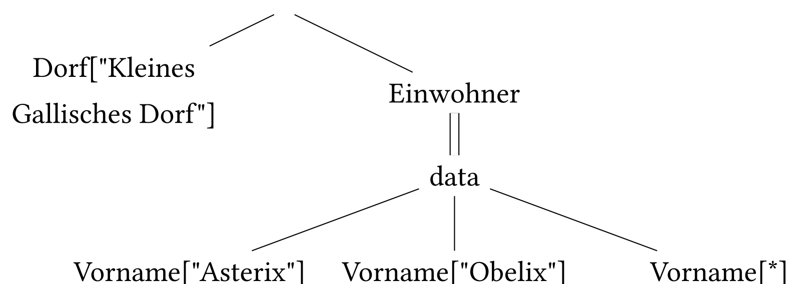

Abbildung 3.1.: Erwartetes Ergebnis als Baum

Das Dorf ist ein direktes Kind des roots und soll den Wert kleines, gallisches Dorf haben. Danach wird festgelegt, dass Asterix und Obelix in den Daten existieren sollen. Dabei ist data ein ArrayType und deswegen wird dem ersten Schrägstrich (/) vor dem Attribut ein zweiter hinzugefügt. Der Kindknoten dieses Attributs Vorname soll Asterix bzw. Obelix sein. In diesem Fall sind der Nachname und die AdressID nicht definiert, dies hat zur Folge, dass die

entsprechenden Attribute beliebige Werte haben dürfen. Zudem wird ein Pfad erstellt, der data * als Inhalt zuweist. Dies bedeutet, dass es beliebig viele Kinderknoten in dem data ArrayType geben darf.

```
pers = sqlContext.read.json("/Spark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("/Spark/files/sources/AdressenRA.json");
adresse2=adresse.filter(adresse["Stadt.name"]=="Kleines Gallisches Dorf")
adresse3=adresse2.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=pers2.join(adresse3, pers2["Person.AdressID"]==adresse3["Adresse.ID"])
persDorf2 = persDorf.select("name", "Person")
persDorf3=persDorf2.rdd.groupByKey().toDF()
persDorf4 = persDorf3.select(persDorf3["_1"].alias("Dorf"), persDorf3["_2"].alias("Einwohner"))
```

Abbildung 3.2.: Gruppierte Ausgabe aller Bewohner des kleinen gallischen Dorfs

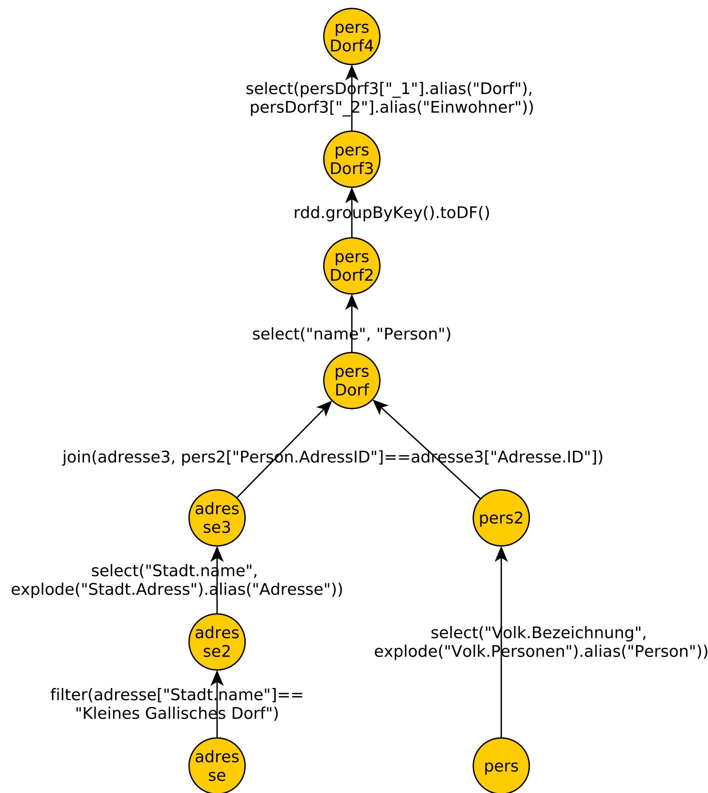

Abbildung 3.3.: Darstellung des Ablaufs der Operationen einer Anfrage als Baum

3.4.2. Ausgabe

Bei den bisherigen Ansätzen für relationale Daten existieren bereits verschiedene Möglichkeiten für die Antwort. In dieser Arbeit wird jedoch ein Ansatz ähnlich zu Why-not und Conseil benutzt, da dieser besonders gut geeignet ist für die Untersuchung der Why-not-Fragen mit einem Anfragebaum. Dem Benutzer wird durch einen Hinweis auf einen Operator, der für den Fehler verantwortlich ist oder mit diesem in Zusammenhang steht, ein guter Ansatzpunkt gegeben, wo er mit der Fehleranalyse beginnen sollte.
Die Ausgabe des Algorithmus wird als Debugging Result bezeichnet (Definition 3.4.7). Da diese Arbeit JSON-Dateien als Eingabe voraussetzt, muss jeder Datensatz mit Hilfe einer Leseoperation in ein RDD/DataFrame umgewandelt werden. Da dieses DataFrame die unveränderten Daten der Datei enthält, zeigt die Annotation dieser Operation an, ob die Eingabedaten vorhanden sind, die ein korrektes Ergebnis ermöglichen.
Formal definiert erfolgt die Ausgabe durch eine Menge von Debugging Explanations (siehe Definition 3.4.7), wobei in der Menge für jeden Operator bzw. DataFrame eine Debugging Explanation vorhanden ist.

Definition 3.4.3 (Required Post Operator Attribut (R-tupel))
Ein R-tupel entspricht der formalen Definition in Kapitel 3.3. Damit das erwartete Ergebnis einer Anfrage zustande kommen kann, muss gelten R-tuples $\subseteq$ des DataFrame der Sparkanfrage.
*Der Wert des Attributs muss nicht unbedingt mit dem exakten Wert angegeben werden. Es können auch Variablen (var), statt einem korrekten Wert, benutzt werden. Zudem müssen nicht alle Werte eines Attributes des hierarchischen Tupels definiert werden. Falls der Wert (w) eines SimpleTypes nicht festgelegt ist, wird davon ausgegangen, dass der Wert beliebig sein darf. Allerdings wird bei einem ArrayType davon ausgegangen, dass, sobald ein childElement definiert ist, nur die angegebenen childElemente existieren. Falls es beliebig viele andere Elemente geben darf, muss ein childElement mit * als Inhalt eingesetzt werden. Für RecordTypes wird der content auf null gesetzt.* Ein R-tupel entspricht einem einzelnen Tupels eines erwarteten Ergebnisses.

Definition 3.4.4 (Required Post Operator DataFrame (R-DataFrame))
Ein R-DataFrame ist ein 2-Tupel (t, C). Dabei ist t eine Menge von R-Tupels $t =< r_1, ..., r_n >$ mit $n \in \mathbb{N}$ und $r \in$ R-Tupel. C ist ein Cardinality Range(min, max) für die Anzahl der notwendigen Tupels. Diese R-Tupels sind für ein bestimmtes DataFrame in einer Anfrage notwendig, damit die Anfrage das erwartete Ergebnis zurückgeben kann. Allerdings können auch mehrere R-Tupels durch ein Tupel in dem DataFrame abgedeckt werden.
Das erwartete Ergebnis entspricht dem R-DataFrame der letzten Operation. Der einzigste Unterschied ist, dass der Algorithmus, je nachdem ob für ein ArrayType ein * angegeben ist, die Cardinality Range dementsprechend festlegen muss.

Definition 3.4.5 (Cardinality Range)
Der Cardinality Range (Kardinalitätsbereich) besteht aus dem 2 -Tupel(min, max). Die Elemente min und max geben an, wie viele Elemente (Kinder eines Knotens/Tupels eines DataFrame) mindestens bzw. maximal notwendig sind, damit die richtige Anzahl an Kindern bzw. Elementen im Ergebnis vorhanden sein kann. Also formal:

AnzahlRichtig $\rightarrow$ #Kinder bzw. Element $<$ *max* $\wedge$ #Kinder bzw. Element $>$ *min*
Zusätzlich gilt: min $=$ *max* $=$ #Kinder bzw. Element $\longleftrightarrow$ *AnzahlRichtig*
Für einen Cardinality Range, dessen Minimal- und Maximalwert nicht gleich ist, lässt sich nur sagen, dass der Wert innerhalb des Wertebereiches zwischen dem minimalen und maximalen Wert liegen muss. Es ist aber nicht sicher, dass die Anzahl der Elemente richtig ist, wenn der Wert in dem Bereich liegt (notwendige Bedingung, aber nicht hinreichend). Falls es egal ist wie viele Elemente bzw. Kinder existieren, ist *min* und *max* null.
Beispiel 3: Ein Cardinality Range kann aus dem 2-Tupel (2, 2) bestehen, also dass sowohl der *min* als auch *max* Wert des Cardinality Range 2 beträgt. Das gerade beschriebene Cardinality Range gehört zu dem ArrayType *data* des folgenden erwarteten Ergebnisses.

```
Dorf["Kleines Gallisches Dorf"] AND Einwohner//data/Vorname["Asterix] AND Einwohner//data/Vorname["Obelix"]
```

Der Benutzer erwartet exakt zwei Kinder (Asterix und Obelix).

Definition 3.4.6 (TracingRDD)
Ein TracingRDD-Objekt stellt eine RDD/DataFrame-Operation dar und enthält ein 6-Tupel (name, P, Oper, R, V, N). name steht dabei für den Namen des RDD/DataFrame, P für die Parameter des RDD/DataFrame. Oper für die Operation des RDD/DataFrame. R für das R-DataFrame, V für die Precedessors (alle Vorgängerknoten im Anfragebaum) und N für die Successors (alle Nachfolgerknoten im Anfragebaum).
Beispiel 4: Ein Beispiel für ein TracingRDD ist das 6-Tupel (persDorf4, {persDorf3["_1"].alias("Dorf"), persDorf3["_2"].alias("Einwohner")}, select, R, {persDorf3}, { }). R ist ein R-DataFrame das als einzigstes R-Tupel das erwartete Ergebnis aus Beispiel 3.4.2 hat. Die Sparkanfrage, die zusammen mit dem erwarteten Ergebnis zu dem TracingRDD führt, ist in Abbildung 3.3 dargestellt.

Definition 3.4.7 (Debugging Explanation)
Eine Debugging Explanation ist ein 2-Tupel (name, annotation). name ist der Name des zugehörigen RDDs. Annotation gibt den Status für das RDD an. Es gilt annotation $\in \{\mathrm{PP}, \mathrm{BB}, \mathrm{BB}_{\mathrm{AE}}\}$. *Dabei steht PP für Passing, BB für Blocking und* $\mathrm{BB}_{\mathrm{AE}}$ *für AnzahlElement. Die genauere Bedeutung der verschiedenen Annotationen wird in 5.1.3 näher beschrieben.*

4. Fragen für hierarchische Daten und Spark

Bisher wurden verwandte Arbeiten angesehen, sowie wichtige Definitionen eingeführt. Im Folgenden wird betrachtet, welche Arten von Fragen ein Why-not-Algorithmus für hierarchische Daten berücksichtigt werden sollte. Dazu werden zuerst alle Fragen betrachtet, die sich stellen, wenn das Ergebnis einer Anfrage anders ausfällt als erwartet. Durch die Benutzung hierarchischer Daten kommen zusätzliche Fehlermöglichkeiten zustande. Dadurch können auch die weiteren Arten von Fragen gestellt werden.
Zuerst wird in dem Teilkapitel 4.1 eine Übersicht und Klassifizierung aller Fragestellungen vorgestellt. Im zweiten Unterkapitel werden dann die möglichen Fragen für hierarchische Daten erforscht und einzeln erklärt. Am Ende wird im dritten Unterkapitel betrachtet, welche Fragen für den Algorithmus relevant und lösbar sind.

4.1. Klassifizierung der Fragen

Es gibt verschiedene Arten möglicher Fragen, die für einen Entwickler von Datenanfragen interessant sein können. Für relationale Daten sind diese die Why-not-Fragen und Why-provenance-Fragen. Die erste Frage beantwortet, wieso bestimmte Daten nicht vorhanden sind. Die zweite Frage, wieso bestimmte Daten existieren. Bei hierarchischen Daten geht es, bedingt durch die Eltern-Kind Beziehung, zusätzlich um die Frage, wieso sich ein Teilbaum in einem hierarchischen Ergebnis nicht an einer anderen Stelle befindet. Diese wird hier Why-here-Frage genannt. Diese Frage ist sowohl für Daten (erwartetes Tupel und tatsächliches Tupel haben dasselbe Schema) als auch für Fragen, die sich auf das Schema beziehen, relevant. Dadurch ergeben sich folgende drei Fragestellungen.

1. Why-not-Frage?
2. Why-provenance-Frage?
3. Why-here-Frage?

Im Gegensatz zu den ersten beiden Fragestellungen ist die letzte nicht relevant für relationale Daten. Die ersten beiden Fragen wurden im relationalen Kontext sehr ausführlich in der Literatur behandelt.

Neben der Einteilung in der Art der Frage kann noch unterschieden werden, ob die Daten oder das Schema betrachtet werden. Daraus ergeben sich folgende Unterscheidungsmöglichkeiten:

1. Es werden nur die Daten beachtet. Dabei wird davon ausgegangen, dass das Schema des erwarteten Ergebnisses und das tatsächliche Schema identisch sind.
2. Es wird nur betrachtet, wieso das Datenschema des Ergebnisses anders als erwartet ist, die Daten sind nicht relevant.
3. Sowohl die Daten als auch das Schema sind anders als erwartet. Dieser Fall kombiniert somit die beiden ersten Fälle und stellt die allgemeinste aller Fragestellungen dar.

Bei der Anfrage des Beispeils 1 ist die Frage, wieso fehlen Asterix und Obelix. Das Schema ist korrekt und wie erwartet. Dadurch handelt es sich um eine Why-not-Frage, die sich nur auf Daten bezieht.
Falls bei dem Ergebnis der Anfrage Julius Caeser in dem kleinen gallischen Dorf wohnt, handelt es sich um eine Why-here-Frage. Wieso wohnt Julius Caeser in dem kleinen gallischen Dorf und nicht in Rom?

4.2. Mögliche Fragestellungen

Wie in Kapitel 4.1. soeben besprochen, sind durch hierarchische Daten zusätzliche Fragestellungen möglich. Alle Fragestellungen, die bei relationalen Datenbankmodellen relevant waren, bleiben aber auch relevant.
Why-provenance-Fragen können aufgrund den in Unterkapitel 4.2.1 vorgestellten Problemen nicht berücksichtigt werden. Die einzelnen Fragestellungen werden im Folgenden näher erläutert. Dabei wird zwischen Fragen zum Schema (Unterkapitel 4.2.2) und Fragen zu den Daten (Unterkapitel 4.2.3) unterschieden. Auch wird diskutiert, ob es möglich ist diese zu beantworten.

Um die möglichen Fragen zu erhalten, wurden für die Elemente aus 3.3 systematisch Einfüge-, Lösch- und Aktualisierungsaktionen durchgeführt. Dabei wird, durch das Einfügen eines Elements in das formale Schema (siehe Kapitel 3.3), dargestellt, dass dieses erwartet war, aber nicht in dem Ergebnis vorhanden ist. Die zugehörige Frage ist dann, wieso fehlt das neu erwartete (das neu eingefügte) Element (**Expect**).
Eine Löschaktion stellt durch das Wegnehmen eines vorhanden Elements dar, dass dies zwar im Ergebnis der Anfrage vorhanden ist, aber nicht erwartet wurde. Dadurch ist die zugehörige Frage wieso ist das nicht erwartete Element vorhanden (**Unexpected**).
Eine Aktualisierungsaktion steht dafür, dass erwartet wurde, dass ein vorhandenes Element anders aussieht (**Update**).
Durch diese Vorgehensweise soll eine vollständige Abdeckung der möglichen Fragen erreicht werden.

Alle Fragen, die aus Einfüge Operationen (**Expect**) entstehen sind Why-provenance-Fragen. Alle Fragen, die aus Löschoperationen (**UnExpected**) entstehen sind Why-not-Fragen. Aus den **Update** Operationen können Fragen aus allen drei Kategorien erstellt werden.
In Tabelle 4.1 ist die systematische Analyse zusammengefasst. Dabei gibt die Operationsspalte an, ob die Frage durch eine Einfüge-, Lösch oder Updateoperation abgeleitet wurde und auf welchen Typ sie sich bezieht. In der darauffolgenden Spalte wird jeder Frage eine Nummer zugewiesen. In der dritten Spalte befindet sich die Definition und eine Erklärung gibt es in Spalte 4. In der letzten Spalte wird angegeben, ob es sich um eine Why-not- oder Why-provenance-Frage handelt und ob sich die Frage auf das Schema oder die Daten bezieht.

Tabelle 4.1.: Herleitung der Fragen

Operation		Formale Frage	Informale Frage	Kategorisierung
Datenbank				
Expect root	1	Wieso $\nexists\, r \in$ Datenbank $\mid r \in$ root	Wieso fehlt ein erwartetes Tupel?	Why not – Daten
	2	Wieso $\lvert$root$\rvert$ < x $\mid$ root $\in$ Datenbank, $x \in \mathbb{N}$	Wieso existieren weniger als x Tupel?	Why not –Daten
Unexpected root	3	Wieso $\exists\, r \in$ Datenbank $\mid r \in$ root	Wieso existiert ein nicht erwartetes Tupel?	Why provenance – Daten
	4	Wieso $\lvert$*root*$\rvert$ > x $\mid$ *root* $\in$ Datenbank, $x \in \mathbb{N}$	Wieso existieren mehr als x Tupel?	Why provenance – Daten
Update root	5	Wieso $\lvert$root$\rvert$ = x $\mid$ *root* $\in$ Datenbank, $x \in \mathbb{N}$	Wieso existieren x Tupels?	Why provenance – Daten
	6	Wieso $\lvert$root$\rvert$ $\neq x$ $\mid$ root $\in$ Datenbank, $x \in \mathbb{N}$	Wieso existieren nicht x Tupels?	Why provenance – Daten
Type				
Update label	7	Wieso *label*=x $\mid$ $x \in$ String $\wedge$ *label* = label(Type)	Warum ist der Name des Typs x?	Why provenance – Daten
	8	Wieso label $\neq x$ $\mid$ $x \in$ String $\wedge$ *label* = label(Type)	Warum ist der Name des Typs nicht x?	Why not – Daten
Update Type	9	Wieso *childType* $\in x$ $\mid$ $x \in$ (Simple Type $\mid$RecordType $\mid$ Array Type)	Warum ist das childAttribut t vom Typ x?	Why provenance – Schema
	10	Wieso *childType* $\notin x$ $\mid$ $x \in$ (Simple Type $\mid$RecordType $\mid$ Array Type)	Warum ist das childAttribut t nicht vom Typ x?	Why not – Schema
SimpleType				
Update value	11	Wieso w = wert(Simple Type) $\mid$ $w \in$ Datentyp	Wieso hat ein Element den Wert w?	Why provenance – Daten
	12	Wieso $w \neq$ wert(Simple Type) $\mid$ $w \in$ Datentyp	Wieso hat ein Element nicht den Wert w?	Why not – Daten

Update value	13	Wieso $w \in$ Datentyp d | $w \in$ wert(Simple Type)	Wieso hat der Wert den Datentyp d?	Why provenance – Schema
	14	Wieso $w \notin$ Datentyp d | $w \in$ wert(Simple Type)	Wieso hat der Wert nicht den Datentyp d?	Why not – Schema
Update nullable	15	Wieso n=*true*/*false* | n=*nullable*(Simple Type)	Wieso darf ein Element null oder keine null Werte enthalten	
ArrayType				
Update containsNull	16	Wieso c=true/false | c=conainsNull(Array Type)	Wieso darf ein ArrayType nullWerte enthalten oder nicht	Why provenance – Daten
Expect childElement	17	Wieso $\nexists\, c \in$ wertChild| $c \in$ RecordType	Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden?	Why not – Daten
	18	Wieso |childElement| < x | childElement $\in$ ArrayType $\wedge\, x \in \mathbb{N}$	Wieso hat ein Knoten weniger als x Kinder?	Why not – Daten
Unexpected childElement	19	Wieso $\exists\, c \in$ *wertChild* | $c \in$ RecordType $\wedge$ *wertChild* $\in$ childelement(ArrayType)	Wieso ist ein nicht erwartetes Kind eines Knotens (Teilbaum) vorhanden?	Why provenance – Daten
	20	Wieso |*childElement*| > x | *childElement* $\in$ ArrayType $\wedge\, x \in \mathbb{N}$ Wieso existieren mehr als x Elemente im ArrayType?	Wieso hat ein Knoten mehr als x Kinder?	Why provenance – Daten
Update childElement	21	Wieso |*childElement*| = x mit *childElement* $\in$ ArrayType $\wedge\, x \in \mathbb{N}$	Wieso hat ein Knoten x Kinder?	Why provenance – Daten
	22	Wieso |*childElement*| $\neq x$ | *childElement* $\in$ ArrayType $\wedge\, x \in \mathbb{N}$	Wieso hat ein Knoten nicht x Kinder?	Why not – Daten
	23	Wieso $i = j$ | $c_i \in$ childElement(ArrayType)	Wieso ist das childElement c an der Stelle i?	Why provenance – Daten
	24	Wieso $i \neq j$ | $c_i \in$ childElement(ArrayType)	Wieso ist childElement c nicht an der Stelle j statt i?	Why not – Daten

	25	Wieso $c \in$ childelement(a) $\wedge c \notin$ *childelement*(b) mit $c \in$ Type $\wedge$ $a \in$ RecordType $\wedge$ $b \in$ RecordType $\wedge$ $c \in$ Type	Wieso gehört der Typ a zum RecordType T statt zu dem RecordType R?	Why-here – Daten
RecordType				
Expect Type	26	Wieso $\nexists x \in r \mid x \in$ Type $\wedge r \in$ RecordType	Wieso fehlt ein Teil der Datenstruktur, die erwartet worden ist?	Why not – Schema
Unexpected Type	27	Wieso $\exists x \in r \mid x \in$ Type $\wedge r \in$ RecordType	Wieso ist ein Teil der Datenstruktur vorhanden, die nicht erwartet wurde?	Why – Schema
UPTDATE Type	28	Wieso $i = j \mid t_i \in$ RecordType	Wieso ist das childAttribut t eines Typs an der Stelle i?	Why – Schema
	29	Wieso $i \neq j \mid t_i \in$ RecordType	Wieso ist das childAttribut t eines Typs nicht an der Stelle i?	Why not- Schema
	30	Wieso $c \in a$ mit $c \in$ Type $\wedge a \in$ RecordType $\wedge$ $c \in$ Type	Wieso ist der Typ a nicht an dieser Stelle in der Hierarchie?	Why-here – Daten
	31	Wieso $c \notin b$ mit $c \in$ Type $\wedge b \in$ RecordType $\wedge c \in$ Type Type.childElement	Wieso ist der Typ a nicht an dieser Stelle in der Hierarchie?	Why-here – Daten
	32	Wieso $c \in a \wedge c \notin b$ mit $c \in$ Type $\wedge a \in$ RecordType $\wedge b \in$ RecordType $\wedge c \in$ Type	Wieso gehört der Typ a zum RecordType T statt zu dem RecordType R?	Why-here – Daten

Fragen, die sich nur auf eine MapType (Zuweisung von einem Schlüssel zu einem bestimmten Wert) beziehen, wurden nicht aufgenommen, da dazu eine andere Eingabe als JSON-Dateien notwendig wäre. Dies betrifft beispielsweise folgende Fragen:

- Wieso hat der KeyType (nicht) den Datentyp x?
- Wieso hat der ValueKeyType (nicht) den Datentyp x?
- Wieso gehört x (value) (nicht) zu y (key)?
- Wieso fehlt ein Key-Value Pair im MapType b?
- Wieso ist containsNull in MapType/ArrayType (keine) true oder false?

Es ist auch möglich, die verschiedenen Fragen zu verbinden oder eine Frage mehrmals zu stellen. So könnte eine Frage sein, wieso zwei bestimmte RecordTypes unter *root* fehlen und wieso unter einem existierenden RecordType in einem Nachfolger Array das SimpleType x fehlt.

4.2.1. Einschränkungen bei Why-provenance-Fragen

Eine Why-provenance-Frage kann nicht beantwortet werden, da bei dieser Frage der Fehler nicht lokalisiert werden kann. Bei einer Why-provenance-Frage muss es also keine eindeutige Fehlerstelle mehr geben. Die vorliegende Arbeit soll dem Ersteller der Datenbankanfrage aber einen konkreten Einstiegspunkt für das Debuggen zur Verfügung stellen.
Es ist möglich, dass die nicht erwarteten Tupels fälschlicherweise schon in den Eingabedaten vorhanden sind. Gleichzeitig kann jede Funktion, die Tupel filtert, so verändert werden, dass das nicht erwartete Ergebnis nicht mehr vorkommt. Auch können beliebig neue Operationen eingefügt werden, um das gewünschte Resultat zu erhalten.
Dadurch gibt es auch beliebig viele Möglichkeiten, das Erscheinen der nicht erwarteten Ergebnisse zu verhindern. Somit können nur Vorschläge gemacht werden, wie die Anfrage abgeändert werden kann, damit das erwartete Ergebnis erzielt wird. Dieser Ansatz wird von Algorithmen benutzt, die auf einer Refinement-based Erklärung aufbauen. Im Gegensatz dazu verfolgt diese Arbeit den Ansatz, den Fehler in der Anfrage zu finden.
Es gibt auch die Möglichkeit eine Why-provenance-Frage mit Datenherkunft zu beantworten. Dabei wird erklärt, wieso diese Daten existieren, indem das Tupel der Anfrage zurückverfolgt wird. Dem Benutzer kann durch Datenherkunft allerdings nicht aufgezeigt werden, wo der Fehler liegt (bei welchem Operator).
Aus diesen Gründen werden die Why-provenance-Fragen zwar in diesem Kapitel behandelt, später allerdings nicht mehr berücksichtigt.

4.2.2. Fragen zum Schema

Für Fragen, die sich auf das Schema beziehen, ist es nicht relevant, welche Eingabedaten vorhanden sind. Es ist nur das Schema der Eingabe sowie die Umwandlung des Schemas durch die Anfrage relevant. Im Unterschied zu relationalen Daten ist es möglich, dass das Schema substantiell durch die Anfrage verändert wird. Es sind nicht nur Umbenennungen möglich, sondern vor allem Änderungen der Hierarchie (grouping und cogroup). Dadurch können Typen ihre Eltern verlieren oder neue bekommen.
Diese Fehler können unter anderem dadurch auftreten, dass die erwartete Datenstruktur, anders als vorgesehen, (nicht) durch eine Projektion gefiltert wird. Es ist aber auch möglich, dass die Typen nicht an die richtigen Stellen gekommen sind oder von diesen getrennt wurden. Auch können die Typen schon in den Eingabedateien fehlen. Diese Fälle werden in dieser Arbeit allerdings nicht weiter behandelt.

4.2.3. Fragen zu den Daten

Bei den Fragen zu den Daten ist nicht das Schema relevant, sondern nur die Daten. Es wird davon ausgegangen, dass das Schema in dem Ergebnis und der gesamten Anfrage korrekt ist. Die einzelnen Fragen, die sich auf Daten beziehen, werden im Folgenden näher beschrieben. Dabei entspricht die Nummer vor der Frage, der Nummer in der Tabelle 4.1.

1. Wieso fehlt ein erwartetes Tupel?

Schema: Wieso existiert das Tupel r nicht in der Menge der roots?
Formal: Wieso $\nexists r \in$ Datenbank | $r \in$ root
Kategorisierung: Es handelt sich um eine Why-not-Frage.

Diese Frage existiert bereits für relationale Daten. Bei dieser Frage fehlt ein Tupel, das der Benutzer erwartet hat, in der Ergebnismenge der Anfrage. Dabei ist das hierarchische Tupel genau festgelegt und der Benutzer erwartet das angegebene Element. Es kann aber sein, dass einzelne Werte nicht exakt definiert sein müssen.

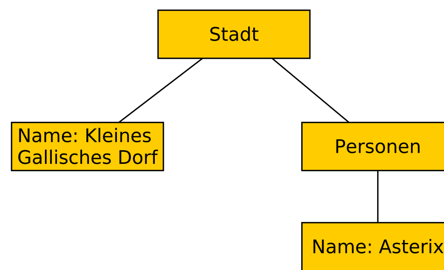

Abbildung 4.1.: Beispiel für ein fehlendes Element: Wieso gibt eine Datenbankabfrage über Adressdaten nicht an, dass Asterix in dem kleinen gallischen Dorf lebt.

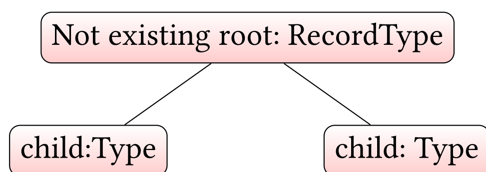

Abbildung 4.2.: Grafische Darstellung der Frage

2. Wieso existieren weniger als x Tupel?

Schema: Wieso existieren weniger als x roots?
Formal: Wieso |root| < x | root $\in$ Datenbank, $x \in \mathbb{N}$
Kategorisierung Es ist eine Why-not-Frage, da es darum geht, dass eine beliebige Anzahl erwarteter Knoten nicht existieren. Dabei ist es egal, welche Werte diese Knoten haben.

Der Benutzer hätte erwartet, dass in der Ergebnismenge mehr als x Tupel existieren.
Es sei beispielsweise von 10 Personen bekannt, dass sie in dem kleinen gallischen Dorf von Asterix leben. Bei einer Abfrage über alle Personen in dem kleinen gallischen Dorf werden allerdings nur Asterix und Obelix zurückgegeben. Dann weiß der Benutzer sicher, dass in dem Ergebnis zu wenig Personen existieren. Er könnte fragen, wieso nicht mindestens 10 Personen existieren (wieso existieren weniger als 10 Personen).

3. Wieso existiert ein nicht erwartetes Tupel?

Schema: Wieso existiert das Tupel r in der Menge der roots?
Formal: Wieso $\exists r \in$ Datenbank | $r \in$ root
Kategorisierung: Es handelt sich um eine Why-provenance-Frage.

Bei dieser Frage existiert ein Tupel, das nicht erwartet worden war.
Ein Beispiel dafür wäre, falls bei der Ausgabe von Städten in Deutschland New York als Tupel existieren würde. Dies ist ein Tupel, das nicht erwartet wurde.
Der Fehler kann durch ein Tupel in den Eingabedaten, das nicht vorhanden sein sollte, ent-

stehen. Es könnte aber auch das Tupel durch eine Operation fälschlicherweise nicht gefiltert worden sein oder es wurde fälschlicherweise durch eine Operation erstellt.

4. Wieso existieren mehr als x Tupel?

Schema: Wieso existieren mehr als x roots?
Formal: Wieso |root| > x | root $\in$ Datenbank, $x \in \mathbb{N}$
Kategorisierung: Diese Frage ist eine Why-provenance-Frage, da gefragt wird, wieso eine unbekannte Anzahl von Tupel existieren, die nicht erwartet wurden.

Der Benutzer hätte erwartet, dass weniger Tupel existieren.
Ein Beispiel dafür ist eine Abfrage über alle Spieler, die an einem bestimmten Fußballspiel teilgenommen haben, bei der mehr als 28 Spieler zurückgegeben werden. Der Benutzer könnte sich wundern, wieso mehr Spieler an dem Spiel teilgenommen haben, als es nach der Regel mit 11+3 Spielern pro Mannschaft möglich ist.

5. Wieso existieren x Tupel?

Schema: Wieso existieren x roots?
Formal: Wieso |root| = x | root $\in$ Datenbank, $x \in \mathbb{N}$
Kategorisierung: Diese Frage ist eine Why-provenance-Frage.

Bei dieser Frage geht der Benutzer davon aus, dass mehr oder weniger Tupel existieren sollen, als in dem Ergebnis einer Anfrage vorhanden sind.

6. Wieso existieren nicht x Tupel?

Schema: Wieso existieren nicht x roots?
Formal: Wieso |root| $\neq x$ | root $\in$ Datenbank, $x \in \mathbb{N}$
Kategorisierung Bei dieser Frage handelt es sich um eine Why-provenance-Frage.

Bei der Frage hat der Benutzer eine bestimmte Anzahl an Ergebnistupel erwartet, die Anfrage liefert aber mehr oder weniger Tupel zurück.
Ein Beispiel dafür wäre eine Anfrage über verschiedene Fußballweltmeisterschaften zwischen 1745 und 2016, bei denen Deutschland den Titel gewonnen hat. Der Benutzer erwartet, dass es vier Weltmeisterschaften als Ergebnistupel gibt, da es der Anzahl der Titel entspricht. Falls die Anfrage ein anderes Ergebnis liefert, wird der Benutzer fragen, wieso die Anzahl der Titel nicht vier ist.

11. Wieso hat ein Element den Wert w?

Schema: Wieso hat ein SimpleType den Wert w?
Formal: Wieso w = wert(SimpleType) | $w \in$ Datentyp.
Kategorisierung: Die Frage ist eine Why-provenance-Frage.

Die Frage hat Ähnlichkeiten mit der Frage, wieso hat ein Kindelement nicht den Wert W. Es wird allerdings gefragt, wieso der Typ innerhalb des Baumes seinen Wert hat.
In der Abbildung 4.3 ist die Frage, wieso ist der Name des Dorfes kleines germanisches Dorf, obwohl bekannt sein sollte, dass Asterix und Obelix nicht in Germanien wohnen.

12. Wieso hat ein Element nicht den Wert w?

Schema: Wieso hat ein SimpleType nicht den Wert w?
Formal: Wieso $w \neq$ wert(Simple Type) | $w \in$ Datentyp.
Kategorisierung: Es handelt sich um eine Why-not-Frage

Die Frage ist, wieso ein Typ innerhalb eines Tupels nicht den erwarteten Wert hat.
Es gibt zwei Möglichkeiten für die Frage.
1. Der Benutzer erwartet, dass der Wert bei einem bestimmten Tupel anders ist, aber dass alle anderen Werte des Tupels gleich bleiben.
2. Der Benutzer erwartet nur, dass der Wert in einem nicht näher spezifizierten Tupel vorkommt.
Das Beispiel 4.3 bezieht sich auf den 1. Fall. Es würde sich auf den 2. Fall beziehen, wenn es nur wichtig wäre, dass ein gallisches Dorf existiert, egal wer dort wohnt.

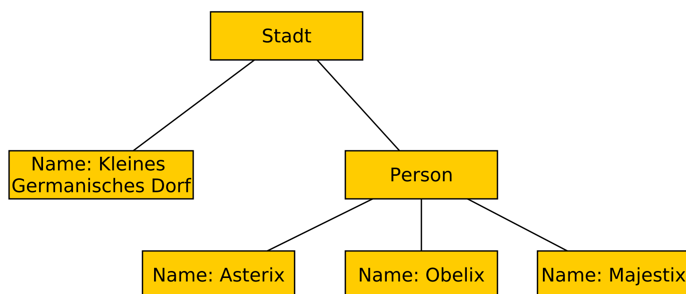

Abbildung 4.3.: Beispiel für einen nicht erwarteten Wert. Eigentlich wohnen Asterix, Obelix und Majestix in einem gallischen Dorf.

17. Wieso ist ein erwartetes Kind eines Arrays (Teilbaum) nicht vorhanden?

Schema: Wieso fehlt das Element w im Array?
Formal: Wieso $\nexists c \in$ wertChild | $c \in$ RecordType
Kategorisierung: Die Frage bezieht sich auf die Daten, da die Elemente, die ein Array als Kinder hat, dort definiert werden. Es handelt sich um eine Why-not-Frage.

Diese Frage bezieht sich darauf, dass sich ein Teilbaum eines hierarchischen Tupels in dem Ergebnis der Anfrage von dem erwarteten unterscheidet. Dadurch handelt es sich um eine Frage, die für relationale Daten nicht von Relevanz ist.
Bei dieser Frage fehlen bei einem vorhandenen hierarchischen Tupel einzelne childElemente eines ArrayType. Es kann aber durchaus sein, dass Geschwisterknoten existieren, wobei es verschiedene Möglichkeiten für deren Zulässigkeit gibt. So kann es bei der Frage, wieso Asterix in dem Beispiel 4.4 nicht in dem kleinen gallischen Dorf lebt, folgende Unterschiede geben.

1. Es ist egal, ob andere Personen in diesem Dorf leben.
2. Es muss eine bestimmte Personenmenge in dem Dorf leben (zum Beispiel Asterix und Obelix).
3. Neben dem fehlenden Kind muss eine bestimme Anzahl von Personen im Dorf leben, es ist aber nicht von allen bekannt, wer diese sind.

Zudem gibt es verschiedene Möglichkeiten, ob der Rest des Baums relevant ist. Diese werden im Folgenden aufgelistet.

1. Der Benutzer erwartet, dass sich nur der Teilbaum bei einem bestimmten Tupel vom vorhandenen unterscheidet. Also bleiben alle anderen Werte des Tupels gleich.

2. Der Benutzer erwartet nur, dass der Teilbaum in irgendeinem Tupel vorkommt. Beispiel 4.4 bezieht sich auf den 1. Fall, würde sich aber auf den 2. Fall beziehen, wenn es nur wichtig wäre, dass Asterix irgendwo wohnt, egal in welchem Dorf und mit welchen Nachbarn.

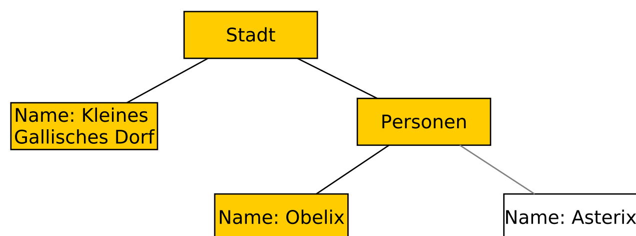

Abbildung 4.4.: Beispiel für ein fehlendes Kind: Wieso gibt eine Datenbankabfrage über Adressdaten nur an, dass Obelix in dem kleinen gallischen Dorf lebt?

18. Wieso hat ein Knoten weniger als x Kinder?

Schema: Wieso existieren weniger als x childElemente in einem ArrayType?
Formal: Wieso $|\text{childElement}| < x \mid \text{childElement} \in \text{ArrayType} \land x \in \mathbb{N}$
Kategorisierung: Es handelt sich um eine Why-not-Frage, da hier gefragt ist, wieso x Kinder

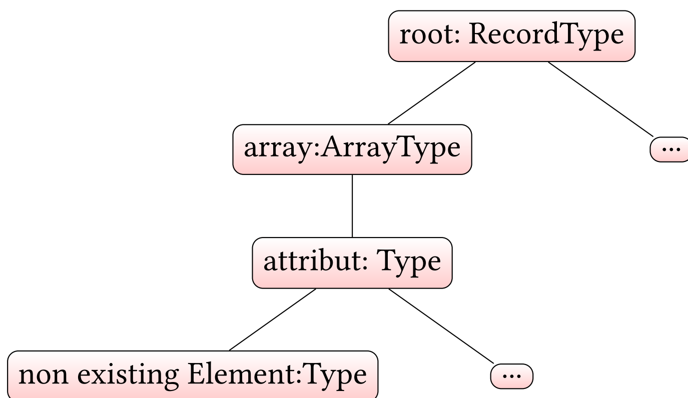

Abbildung 4.5.: Grafische Darstellung der Frage

fehlen.

Diese Frage ist nur bei verschachtelten Daten möglich. Es wird gefragt, wieso ein Array-Type weniger Kinder hat als erwartet.

19. Wieso ist ein nicht erwartetes Kind eines Knotens (Teilbaum) vorhanden?

Schema: Wieso existiert root r?
Formal: Wieso $\nexists r \in$ Datenbank | $r \in$ root
Kategorisierung: Es handelt sich um eine Why-provenance-Frage.

Diese Frage existiert nicht für relationale Daten. Bei dieser Frage hat ein ArrayType des Ergebnisses ein Kind, das der Benutzer nicht erwartet hat.
Ein Beispiel dafür ist eine Anfrage, die für das kleine gallische Dorf alle Bewohner als Teilbaum des Dorfes zurückgibt und dabei Julius Caeser enthält, obwohl er eigentlich nicht als Bewohner zu diesem Dorf gehört.
Für die Geschwisterknoten gilt dieselbe Aufteilung wie in Frage 17. (Entweder alle anderen Geschwisterknoten müssen trotzdem vorhanden sein, während die anderen Geschwisterknoten nicht relevant sind oder eine bestimmte Anzahl an Geschwistern muss existieren)

20. Wieso hat ein Knoten mehr als x Kinder?

Schema: Wieso existieren mehr als x childElemente im ArrayType?
Formal: Wieso |childElement| > x | childElement $\in$ ArrayType $\wedge\ x \in \mathbb{N}$
Kategorisierung: Es handelt sich um eine Why-provenance-Frage, da hier gefragt wird, wieso x Kinder zu viel vorhanden sind.

Es wird gefragt wieso ein ArrayType mehr Kinder hat als erwartet.
Ein Beispiel dafür ist eine Abfrage über alle Personen, die in dem kleinen gallischen Dorf von

Asterix leben (die Personen werden als Kinder des ArrayTypes Einwohner zurückgegeben). Dem Benutzer seien 10 Personen bekannt, die in dem Dorf leben. Dadurch weiß er, dass mindestens 10 Ergebnistupel vorhanden sein sollten. Wenn die Abfrage nur Asterix und Obelix zurückgibt, könnte der Benutzer fragen, wieso nicht mindestens 10 Personen existieren (wieso existieren weniger als 10 Personen).

21. Wieso hat ein Knoten x Kinder?

Schema: Wieso existieren x Elemente im ArrayType?
Formal: Wieso |childElement| = x mit childElement ∈ ArrayType $\wedge x \in \mathbb{N}$
Kategorisierung: Diese Frage existiert nur bei verschachtelten Daten, bei denen von einem Typ mehrere Instanzen für einen Elternknoten existieren können. Es ist eine Why-provenance-Frage und bezieht sich auf die Daten.

Es kann vorkommen, dass es bei einer Datenbankabfrage nicht wichtig ist, welche Kinder ein Element hat, sondern wieso es keine oder eine bestimmte Anzahl an Kindern hat. In Abbildung 4.6 ist bekannt, dass Obelix eigentlich keine drei Haustiere hat, deswegen ist die Frage, wieso existieren drei Kinder in dem Datenbaum als Haustiere. Diese Frage ist equivalent zu der Frage wieso existiert das Tupel mit dem Knoten, der drei beliebige Kinder hat. Diese Frage kann genauso wie die Frage, wieso existiert ein nicht erwartetes Tupel, nicht beantwortet werden.

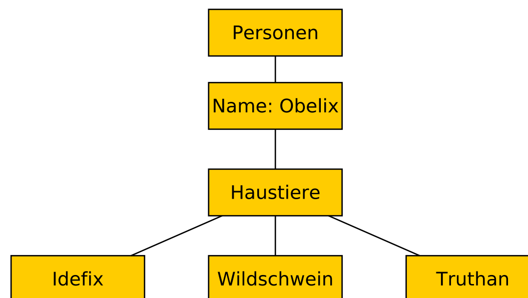

Abbildung 4.6.: Beispiel für die falsche Anzahl von Kindern: Wieso gibt eine Datenbankabfrage über Adressdaten an, dass Obelix drei Haustiere besitzen soll, obwohl er nur Idefix als Haustier hat.

22. Wieso hat ein Knoten nicht x Kinder?

Schema: Wieso existieren nicht x Elemente im ArrayType?
Formal: Wieso |childElement| $\neq x$ | childElement ∈ Array Type $\wedge x \in \mathbb{N}$
Kategorisierung: Es ist eine Why-not-Frage.

Diese Frage ist nur bei verschachtelten Daten möglich. Es wird gefragt, wieso ein ArrayType nicht die erwartete Anzahl an Kindern hat.

Bei dem Beispiel 4.6 ist die Frage, wieso hat Obelix nicht genau ein Haustier. Es ist bekannt, dass Obelix einzigstes Haustier sein Hund Idefix ist.

23. Wieso ist das childElement c an der Stelle i?

Formal: Wieso $i = j \mid c_i \in$ childElement(ArrayType)
Kategorie: Diese Frage ist eine Why-provenance-Frage.

Da es sich bei dem ArrayType, um eine Sequenz handelt ist die Reihenfolge relevant. Dadurch wird auch die Frage möglich, wieso ein Element innerhalb des ArrayTypes an einer bestimmten Stelle steht.

24. Wieso ist das childElement c an der Stelle j statt i?

Formal: Wieso $i \neq j \mid c_i \in$ childElement(ArrayType)
Kategorie: Diese Frage ist eine Why-not-Frage.

Da es sich bei dem ArrayType, um eine Sequenz handelt, ist die Reihenfolge relevant. Dadurch ist auch eine mögliche Frage, wieso ein Element innerhalb des ArrayTypes sich an der Stelle j und nicht an der Stelle i befindet.

25. Wieso hat das Element X nicht den Elternknoten Y statt den Elterknoten Z? (3)

Schema: Wieso ist das childElement x nicht Kind des RecordType Y statt des RecordType Z?
Formal: Wieso $c \in$ childelement(a) $\wedge$ $c \notin$ childelement(b) mit $c \in$ Type $\wedge$ $a \in$ RecordType $\wedge$ $b \in$ RecordType $\wedge$ $c \in$ Type
Kategorisierung: Bei dieser Frage handelt es sich um eine Why-here-Frage.
Die Frage ist, wieso hat ein Type nicht einen anderen Elternknoten, der allerdings an derselben Stelle in der Hierarchie stehen muss. Diese Frage existiert auch nicht für relationale Daten, da dort keine Elternknoten vorhanden sind. In Beispiel 4.7 kann gefragt werden, wieso Idefix nicht das Haustier von Obelix ist.
Diese Frage ist schwierig zu beantworten, da es beliebig viele Möglichkeiten gibt, wieso der Knoten zu dem erwarteten Elternknoten kommen kann. Die RDD/Data Frame Operationen können beliebig ergänzt und verändert werden, sodass X den Elternknoten Y hat. Der Algorithmus soll allerdings die fehlerhaften Operationen finden und keine Alternativvorschläge machen. Deswegen kann diese Frage nicht beantwortet werden.

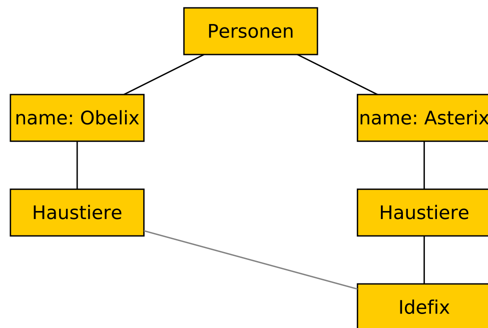

Abbildung 4.7.: Beispiel für ein Element mit dem falschen Elternknoten: Wieso ist Idefix das Haustier von Asterix und nicht von Obelix?

4.3. Zusammenfassung

Dieses Kapitel verdeutlicht, dass es viele Fragen geben kann, die ein Benutzer an eine fehlerhafte Datenbankanfrage in Zusammenhang mit der Ausgabe stellen kann. Viele davon sind für relationale Daten nicht vorhanden. Der Algorithmus dieser Arbeit beantwortet allerdings nicht alle Fragen. So werden keine Why-provenance-Frage betrachtet, aus den in 4.1 beschriebenen Gründen. Dadurch werden die Fragen 3–5, 7, 9, 13, 15, 19, 20–21, 23, 27–28 nicht weiter behandelt.
Auch Fragen, die sich auf das Schema beziehen, werden in dieser Arbeit nicht weiter berücksichtigt, da fehlende Daten erklärt werden sollen. Dazu gehören folgende Fragen 10, 14, 26, 29. Fragen bei denen verschiedene Tupel miteinander verglichen werden, werden aus Zeitgründen nicht berücksichtigt, dazu zählen die Fragen 30, 31, 32. Auch die Reihenfolge von Elementen und ob null Werte existieren können, wird in dieser Arbeit nicht weiter berücksichtigt. Dies betrifft die Fragen 24, 15 und 16. Dadurch werden die folgenden Fragen in dieser Arbeit und für den Algorithmus in Kapitel 5 weiter berücksichtigt.

- 1. Wieso fehlt ein erwartetes Tupel?
- 2. Wieso existieren weniger als x Tupel?
- 6. Wieso existieren nicht x Tupels?
- 12. Wieso hat ein Element nicht den Wert w?
- 17. Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden?
- 18. Wieso hat ein Knoten weniger als x Kinder?
- 22. Wieso hat ein Knoten nicht x Kinder?

Die Fragen 17, 18 und 22 existieren nicht für relationale Daten, sondern stellen sich nur bei hierarchischen Daten.
Alle Fragen, mit Ausnahme der Frage 2 und 18, können damit beantwortet werden, dass der Benutzer ein Tupel angibt, das komplett oder als Teil eines anderen Tupel vorhanden ist. Dies wird jeweils bei den Erläuterungen der einzelnen Fragen im Teilkapitel 4.2.2 gezeigt. Die Fragen 2 und 18 würden eine große Erweiterung des Algorithmus und damit stark erhöhte Komplexität bedeuten. Zudem sind diese Fragen sehr ähnlich zu den Fragen 6 und 22. Aus diesen Gründen werden auch diese nicht weiter berücksichtigt.

5. Algorithmus

Der Algorithmus soll einem Benutzer zeigen, welcher Operator bei einer Anfrage dafür verantwortlich ist, dass das Ergebnis anders ist als erwartet. Eine solche Frage an den Algorithmus wird Debugging-Szenario genannt.
Die Fragen, die der Algorithmus beantworten soll, werden in Kapitel 4 genau beschrieben. Alle Fragen, die nach der dortigen Analyse betrachtet werden, können dadurch beantwortet werden, dass geprüft wird, wieso ein Tupel mit bestimmten Attributen nicht existiert. Abbildung 4.4 zeigt die Ein- und Ausgabe des Algorithmus. Der Algorithmus benötigt die fehlerhafte Sparkanfrage und das erwartete Ergebnis als Eingabe. Nach der Ausführung des Algorithmus wird zurückgegeben, welche Operatoren für das fehlerhafte Ergebnis verantwortlich sein könnten. In diesem Kapitel werden zuerst die Eingabe 3.4.1 und Ausgabe 3.4.2 definiert. Anschließend wird der eigentliche Algorithmus im Unterkapitel 5.1 vorgestellt. Am Ende dieses Kapitels werden in 5.3 die Einschränkungen und Probleme des Algorithmus diskutiert.

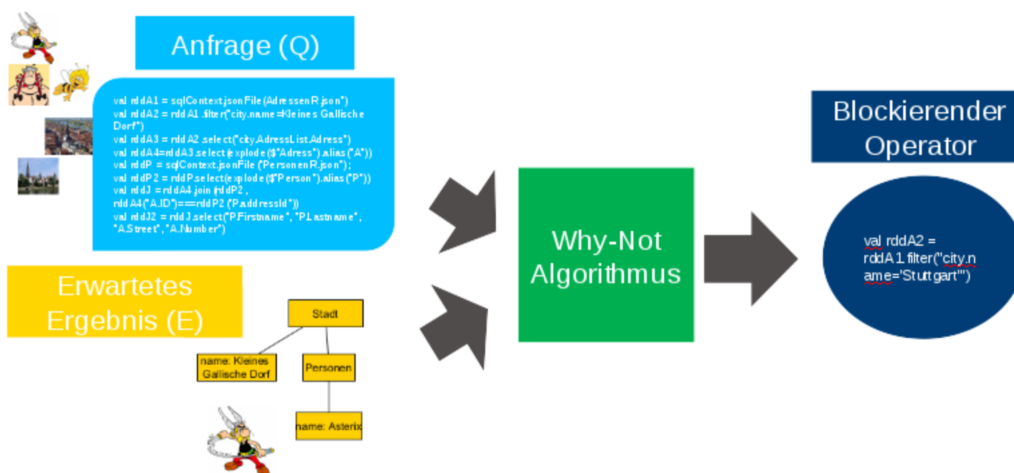

Abbildung 5.1.: Schematische Darstellung des Algorithmus

5.1. Algorithmus

Der Algorithmus erklärt, wieso ein erwartetes Ergebnis nicht in der Ergebnismenge vorkommt, stellt aber auch fehlende Eingangsdaten fest. Er lässt sich in drei Teile gliedern: GenerateRe-

questTree, TraceAttribut und SetAnnotation. Abbildung 5.2 zeigt den Aufbau dieser Schritte mit Eingabe und Ausgabe, der Quellcode wird in Algorithmus 1 als Pseudocode dargestellt. Für die Durchführung des Algorithmus werden verschiedene Datenstrukturen gebraucht, diese werden im Kapitel "Grundlagen" vorgestellt werden.

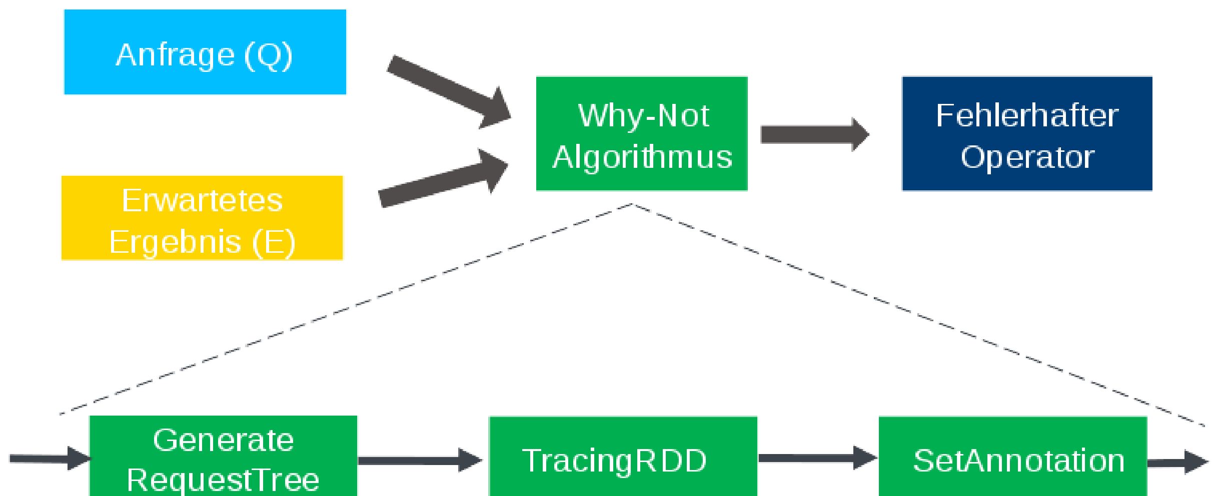

Abbildung 5.2.: Aufbau des Algorithmus

ALGORITHM 1: BasisAlgorithmus

input : Debugging Szenario(rdd0, query)
output : Debugging Explanation e

1. *treeRoot* ← GenerateRequestTree(*anfrage*);
2. *tracingRDDRoot* ← TraceAttribut(*rdd0*, *treeRoot*);
3. *e* ← SetAnnotation(*tracingRDDRoot*, R);
4. **return** *e*;

5.1.1. GenerateRequestTree

GenerateRequestTree erstellt einen Baum der Anfrageoperationen auf RDDs und DataFrames. Dabei werden die RDD Operationen in TracingRDD Objekte gespeichert. Diese enthalten unter anderem die Vorgänger und Nachfolger. In Abbildung 5.3 wird ein Anfragebaum grafisch dargestellt. Der entsprechende Algorithmus wird in Pseudocode in Algorithmus 2 dargestellt.

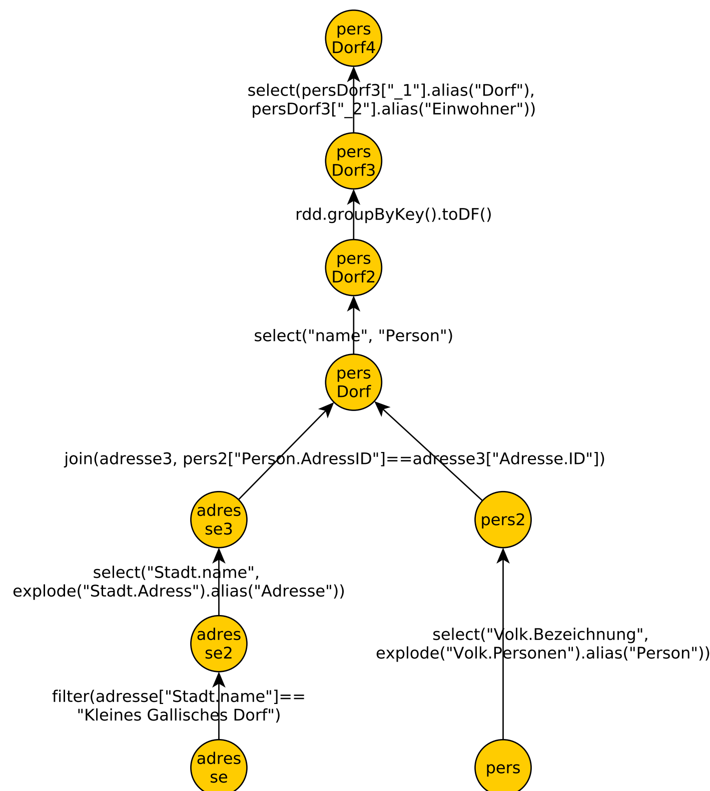

Abbildung 5.3.: Darstellung des Ablaufs der Operationen des Beispiels 3.2 als Baum

ALGORITHM 2: GenerateRequestTree

input : RDDOperations operations
output : TracingRDD rootOfTree

1 **for** Operation o : *operations* **do**
2   generiere TracingRDD r für o;
3   setze Vorgänger und Nachfolger für r;
4   **if** r hat kein Nachfolger **then**
5         $rootOfTree \leftarrow r$;
6   **return** *rootOfTree*

5.1.2. TraceAttribut

In diesem Schritt, wird ausgehend von dem erwarteten Ergebnis, für jede RDD/DataFrame Operation das entsprechende R-DataFrame berechnet. Dadurch wird ermittelt welche Tupel

nach dem Ausführen dieser Operationen vorhanden sein müssen. Dies ist ähnlich zu dem Why-not-Algorithmus. Für die hierarchischen Daten ist allerdings eine spezielle Speicherstruktur erforderlich, und es müssen zusätzliche Informationen erfasst werden (min, max Anzahl der Tupel und min, max Anzahl der Kinder der ArrayTypes). Dabei wird für jedes TracingRDD aus dem vorher erstellten Baum errechnet, welche Daten innerhalb des entsprechenden RDDs vorhanden sein müssen. Dabei wird ein Top-down Ansatz verfolgt. Bei dem zuerst aus dem vom Benutzer erwarteten Ergebnis das entsprechende R-DataFrame erstellt wird. Aus diesen werden rekursiv die R-DataFrames für die vorherigen RDD errechnet. Dies wird fortgeführt bis die notwendigen Daten in der Datenquelle errechnet worden sind.
In diesem Schritt ist das Berechnen der notwendigen Daten eines RDDs aus den Daten des Nachfolger RDDs und der Operation die größte Herausforderung. Dies gehört zu dem Bereich der Datenherkunft, das einzeln für jede mögliche Operation auf den RDDs implementiert wird. In dieser Arbeit wird nur ein Teil der RDD-Operationen berücksichtigt. Dazu zählen auch Operatoren, die bei relationalen Datenbanken vorhanden sind. Mit Ausnahme des letzten Operators, der Sparkanfrage, ist es nur sinnvoll RDD Operationen zu berücksichtigen, die auch wieder einen RDD- oder DataFrame Objekt zurückgeben. Teile der Operatoren werden als Pseudocode näher beschrieben. In Algorithmus 4 wird der entsprechende Algorithmus für GroupByKey beschrieben und in Algorithmus 3, derjenige für select.

ALGORITHM 3: calculateSelectTraceAttributForPrecedessors

input : R-DataFrame attribut, TracingRDD traceRDD
output : TracingRDD des Baums mit TraceAttributen

```
1  res ← new R-tupel;
2  res.addChilds(SelektierterType)
3  for traceAttribut ∈ attribut.t do
4      for param ∈ Parameter des traceRDD do
5          if param contains explode then
6              if attribut.min! = null then
7                  res.selektiertesArrayType.min = 1;
8              if attribut.max! = null then
9                  res.selektiertesArrayType.max = attribut.max;
10         //Eine select Funktion muss genau einen Vorgänger haben;
11         //Dessen R-DataFrame wird zu res gesetzt;
12         traceRDD.V.get(0).R ← res;
```

In dem Pseudocode 5 des TraceAttributs werden diese Operationen in der Zeile 7 aufgerufen. Die auftretenden Probleme werden in Kapitel 5.3 näher beschrieben.
Die Funktion getChild(*param*) liefert das Kind der roots des R-tupel zurück, dessen label param ist. Die Funktion addChild(*type*) fügt type zu dem root des R-tupels hinzu, addChilds(*types*) ordnet dem root des R-tupels eine Liste von Typen zu.

ALGORITHM 4: calculateGroupByKeyTraceAttributForPrecedessors

```
input  : R-DataFrame attribut, TracingRDD traceRDD
output : TracingRDD des Baums mit TraceAttributen

1  resFrame ← new R − DataFrame;
2  for traceAttribut ∈ attribut.t do
3      for secondValue ∈ traceAttribut.getChild("_2.data") do
4          newTraceAttribut ← newtraceAttribut;
5          newTraceAttribut.addChilds(traceAttribut.getChild("1.data");
6          newTraceAttribut.addChilds(traceAttribut);
7          ? resFrame.add(newTraceAttribut);
8      resFrame.anzahlMinTupelDataFrame ← 1;
9      resFrame.anzahlMinTupelDataFrame ← ∞;
10     traceRDD.precedessors.R ← resFrame;
```

ALGORITHM 5: TraceAttribut

```
input  : R-DataFrame c, TracingRDD rootRDD
output : TracingRDD des Baums mit TraceAttributen

1  stack ← new list of TracingRDD;
2  stack.add(rootRDD);
3  rootRDD.R ← c;

4  for aktKnoten ∈ stack do
5      while aktKnoten ist nicht root do
6          aktTraceAttribut ← aktKnoten.R;
7          aktKnoten.calculateTraceAttributForSuccessors(aktTraceAttribut);
8          stack ← aktT.getSuccessors;
```

In diesem Schritt ist es entscheidend, die Hierarchie zu berücksichtigen. Es muss sichergestellt werden, dass die geforderte Anzahl von Geschwisterknoten vorhanden ist. Dazu wird die minimale und maximale Anzahl der Tupels in dem R-DataFrame gespeichert und die minimale und maximale Anzahl von Kindern eines ArrayTypes. Dies geschieht in den Cardinality Range Objekten.
Dies muss für jede Operation einzeln implementiert werden, beispielsweise ändert eine select-Operation ohne explode die Werte nicht. Während eine Filterfunktion die Anzahl der maximalen Tupel auf unendlich setzt und die anderen Werte lässt.

Beispiel 5: Abbildung 5.4 zeigt den Anfragebaum zu Abbildung 3.2, ergänzt um das erwartete Ergebnis der Eingabe des Debugging Szenarios (hellblauer Kasten 1). Es wird beschrieben, dass der Benutzer Asterix und Obelix als Bewohner des Dorfes erwartet hätte. Wobei der Nachname nicht relevant ist, er kann einen beliebigen Wert annehmen.

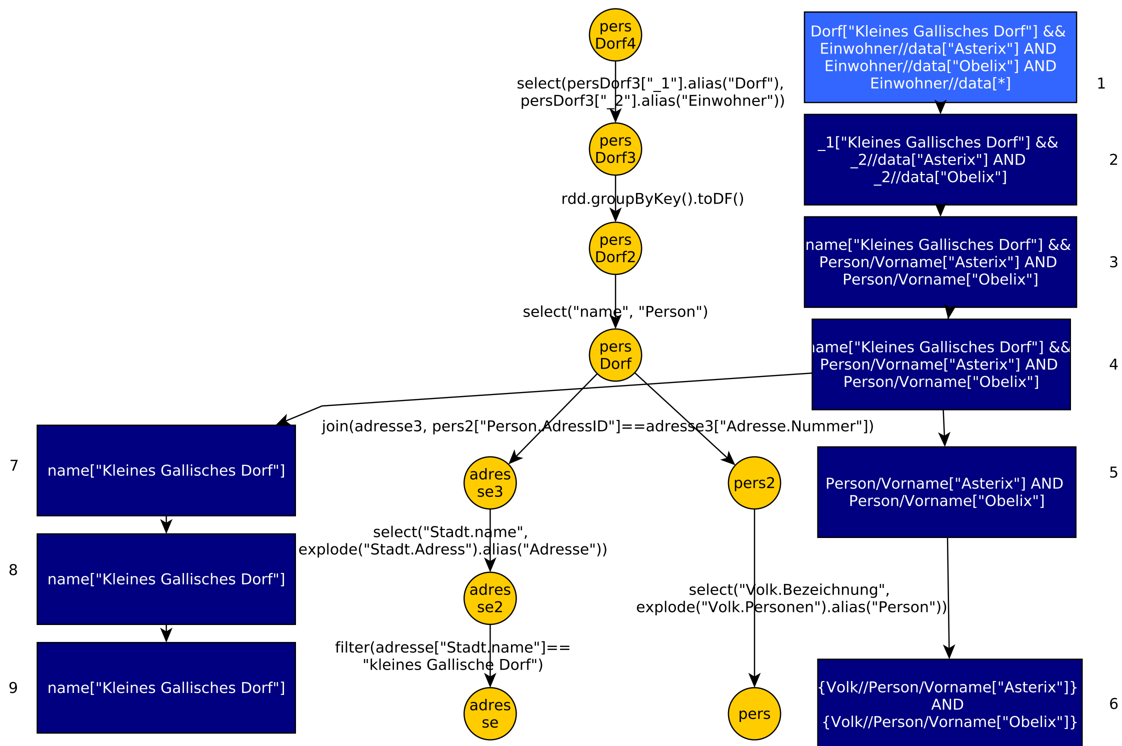

Abbildung 5.4.: Darstellung des Baums und die zugehörigen zurückverfolgten Werte.

Die dunkelblauen Kästen stellen die Elemente des einzigen R-Tupels des R-DataFrame dar (nur Berücksichtigung des Namens ohne cardinality ranges). Diese werden von dem Algorithmus im TraceAttribut errechnet. Da in diesem Fall eine beliebige Anzahl von Menschen in dem Dorf leben dürfen, wird der Algorithmus die cardinality ranges nicht weiter berücksichtigen. An den Labels der Kästen ist die Top-down Reihenfolge zur der Berechnung des Algorithmus angegeben.
Der Algorithmus erstellt jeweils aus dem R-DataFrame, das nach der Durchführung einer Operation notwendig ist, das R-DataFrame, das vor der Operation notwendig ist. Dies ist gleichzeitig das Ergebnis der vorherigen Operation. In diesem Fall muss dort für das *adresse* RDD/DataFrame das kleine gallische Dorf vorhanden sein. Bei *pers* sind dies Asterix und Obelix, wobei der Algorithmus durch die explode Funktion nicht weiß, ob diese zu demselben ArrayType gehören. Es könnte sowohl sein, dass Asterix zu dem selben Volk als auch zu unterschiedlichen Völkern gehören, wobei beide Möglichkeiten zu demselben Ergebnis in *pers2* führen. Dieses Problem wird in 5.3 näher erläutert.

5.1.3. SetAnnotation

SetAnnotation benutzt die TracingRDD mit dem erwarteten Ergebnis, um festzustellen, welche RDD Operationen verhindern, dass erwartete Daten in der Ausgabe erscheinen. Dazu werden die vom Benutzer vorgegebenen Spark Befehle ausgeführt. Um festzustellen, ob die entsprechende RDD Operation für das Fehlen verantwortlich sein kann, wird überprüft, ob die Daten aus dem TraceAttribut Schritt in den Zwischenergebnissen vorhanden sind. Dies ist dasselbe wie bei Why-not. Allerdings muss bei hierarchischen Daten auch berücksichtigt werden, dass die Anzahl der Geschwisterknoten für alle ArrayTypes und Anzahl der Tupel eines Attributs so sind wie gefordert. Deshalb sind drei Bedingungen für die Anzahl der Gesamt Tupel und alle ArrayTypes zu berücksichtigen.

- Es existieren nicht alle TraceAttribute aus dem Schritt SetAnnotation (Element fehlt)
- Die Anzahl der Elemente für die Tupel und alle ArrayTypes sind richtig (Anzahl kann richtig sein)
- Es ist sicher, dass die Anzahl der Elemente für die Tupel und alle ArrayTypes richtig/falsch sind (Anzahl unsicher)

Falls bei mehreren ArrayTypes bzw. der Anzahl der Element relevant ist wie viele Kinder bzw. Elemente existieren, wird der schlimmste Fall ausgewählt. Dadurch ist die Reihenfolge:

1. Anzahl kann richtig sein = false und Anzahl unsicher = false;
2. Anzahl kann richtig sein = true und Anzahl unsicher = false;
3. Anzahl kann richtig sein = true und Anzahl unsicher = true;

Dies führt dazu, dass, anders als bei Why-not, nicht nur die Operatoren als Schuld markiert werden, bei denen ein Ergebnis fehlt, sondern ähnlich zu Conseil (siehe Kapitel 2.2) eine Kennzeichnung der Operationen bzw. DataFrames erfolgt. Es gibt PP, BB und BB_{AE}. Dabei steht PP für Passing (blockiert keine Tupel), BB für Blocking (blockiert vorhandene Tupel) und BB_{AE} (Blocking Anzahl Element). Letzteres wird benutzt, falls im Ergebnis ein ArrayType zu viele childElemente hat. Es gibt an, dass es bis zu diesen Tupel durch die Operationen noch möglich ist, dass überflüssige childelements bzw. Elemente durch die Anfrage aus dem Ergebnis gefiltert werden. Dadurch wird die erste Funktion der Anfrage als BB_{AE} gekennzeichnet, bei der es sicher ist, dass die Anzahl der Elemente falsch ist. Der Algorithmus beantwortet diese Frage nicht näher. Es handelt sich nämlich um die Frage, wieso dieses Element vorhanden ist. Dies entspricht allerdings einer Why-provenance-Frage und kann, aus den in Kapitel 4.1 genannten Gründen, von dem Algorithmus nicht exakt berücksichtigt werden.
Ein DataFrame kann sowohl die Annotation BB_{AE} als auch die Annotation BB gleichzeitig haben. Ansonsten sind aber keine Kombinationen von Annotation möglich. Zu Beginn hat die Annotation der R-DataFrames den Zwischenstand ND (Nicht Definiert). ND weist auf einen noch nicht festgelegten Werte hin.

In der folgenden Tabelle wird dargestellt, welche Fälle notwendig sind. Ein Beispiel des Algorithmus wird in Abbildung 5.5 gezeigt, der Pseudocode in den Algorithmen 6 bis 8.

Anzahl kann richtig sein	Anzahl unsicher	Element fehlt	Reaktion
0	0	0	Blocking Anzahl Element
0	0	1	Blocking, Blocking Anzahl Element
0	1	0	Kann nicht eintreten
0	1	1	Kann nicht eintreten
1	0	0	Passing
1	0	1	Blocking
1	1	0	Passing
1	1	1	Blocking

Falls ein Operator blockierend ist und ein oder mehrere Elemente fehlen, wird dies nach dem Operator genauso wie bei Conseil zu den Daten hinzugefügt, bevor der nächste Operator dies als Eingabe erhält.

ALGORITHM 6: SetAnnotation

```
input  : R-DataFrame rootRDD, Operationen rddO
output : Debugging Explanation e
1  res ← new Debugging Explanation;
2  Map < name, DataFrame > rddResults ← executeOperations(rddO);
3  stack ← new Stack < TracingRDD >;
4  stack.push(rootRDD);
5  rddWithAnnotation = new Menge < TracingRDD >;
6  while stack.hasNext() do
7      curR ← stack.pop();
8      if curR hat kein Vorgänger, dessen Annotation ND ist then
9          SetAnnotationTraceingRDD(curRDD, curRDD.R, rddResults);
10         rddWithAnnotation.add(curR);
11         res.add(curR.name, curRDD.annotation)
12 return res;
```

ALGORITHM 7: SetAnnotationTraceingRDD

```
   input  : TracingRDD kn, R-DataFrame R, Map<name, DataFrame> rddContent
   output : annotated TracesRDD
 1 ElementFehlt ← false;
 2 for tupel ∈ RDDTupels do
 3 |   tupelExist ← false;
 4 |   for r ∈ rddContent do
 5 |   |   if tupel entspricht r then
 6 |   |   |   tupelExist ← true;
 7 |   if tupelExist == false then
 8 |   |   ElementFehlt = true;
 9 (AnzahlRichtig, AnzahlUnsicher) ← isAnzahlKorrekt()
10 if !AnzahlRichtig()∧!AnzahlUnsicher()∧!ElementFehlt then
11 |   kn.annotation == BB_AE;
12 if !AnzahlRichtig()∧!AnzahlUnsicher() ∧ ElementFehlt then
13 |   kn.annotation == BB, BB_AE;
14 if AnzahlRichtig()∧!AnzahlUnsicher()∧!ElementFehlt then
15 |   kn.annotation == PP
16 if AnzahlRichtig()∧!AnzahlUnsicher() ∧ ElementFehlt then
17 |   kn.annotation == BB
18 if AnzahlRichtig() ∧ AnzahlUnsicher()∧!ElementFehlt then
19 |   kn.annotation == PP
20 if AnzahlRichtig() ∧ AnzahlUnsicher() ∧ ElementFehlt then
21 |   kn.annotation == BB
22 if ElementFehlt then
23 |   rddContent.get(kn.name).add(missingTupels)
   |   rddContent = rexecuteRDDOperations(operations, kn.name)
```

ALGORITHM 8: IsAlgorithmusKorrekt

```
   input  : R-tupel tupel, DataFrame f
   output : (Boolean isAnzahlFalsch, Boolean isAnzahlKorrekt)
 1 isAnzahlFalsch ← false;
 2 isAnzahlKorrekt ← true;
 3 if Anzahl der Elemente von tupel relevant then
 4 |  if Anzahl Elemente in f < tupel.C.minAND Anzahl Elemente inf > tupel.C.min then
 5 |  |  isAnzahlFalsch ← true;
 6 |  if tupel.C.min <> tupel.C.max then
 7 |  |  isAnzahlKorrekt ← false;
 8 for Durchlaufe alle Typen t von tupel do
 9 |  if t ∈ ArrayType then
10 |  |  if Anzahl der Elemente von t relevant then
11 |  |  |  if Anzahl Elemente in f < t.C.min AND Anzahl Elemente inf > t.C.min
   |  |  |  then
12 |  |  |  |  AFalsch ← true;
13 |  |  |  if t.C.min! = t.C.max then
14 |  |  |  |  AKorrekt ← false;
15 |  |  |  if
   |  |  |  (AFalsch, AKorrekt) hat höher Priorität als (isAnzahlFalsch, isAnzahlKorrekt)
   |  |  |  then
16 |  |  |  |  isAnzahlFalsch ← AFalsch;
17 |  |  |  |  isAnzahlKorrekt ← AKorrekt;
18 return (isAnzahlFalsch, isAnzahlKorrekt);
```

Beispiel 6: In der Abbildung 5.5 wird gezeigt, welche Annotation der Algorithmus für die einzelnen Operatoren in Abhängigkeit der R-DataFrames gibt. Es handelt sich dabei um die in Abbildung 3.2 dargestellte Anfrage. Die Daten entsprechen denen, die in Abbildung 1.3 und 1.4 eingeführt wurden. Im Unterschied zum Beispiel 18 ist dabei die Anzahl der Elemente und Kinder des ArrayTypes Einwohner nicht relevant.
Der Algorithmus überprüft, ob die notwendigen Elemente, die sich aus den dargestellten R-DataFrames ergeben, vorhanden sind. Da Asterix, Obelix und das kleine gallische Dorf in den Eingabedaten vorhanden sind, sind die Operationen *adresse* und *pers* PP. Diese beiden enthalten jeweils unverändert die Eingabedaten. Auch *addresse2*, *addresse3* und *pers2* sind PP, denn die entsprechenden Daten wurden nicht gefiltert oder so verändert, dass nicht korrekte Daten entstehen. Gleichzeitig benutzt der join-Operator aber einen falschen Parameter. *Nummer* steht für die Hausnummer und nicht für die *ID* der Adresse. Dadurch werden Asterix und Obelix herausgefiltert, womit dieser Operator die Eigenschaft blocking erhält. Anschließend fügt der Algorithmus dem DataFrame das notwendige Tupel hinzu und führt die verbleibende Anfrage erneut aus. Dadurch kann der Algorithmus erkennen, dass die anderen Operatoren keine weiteren Fehler mehr verursachen.

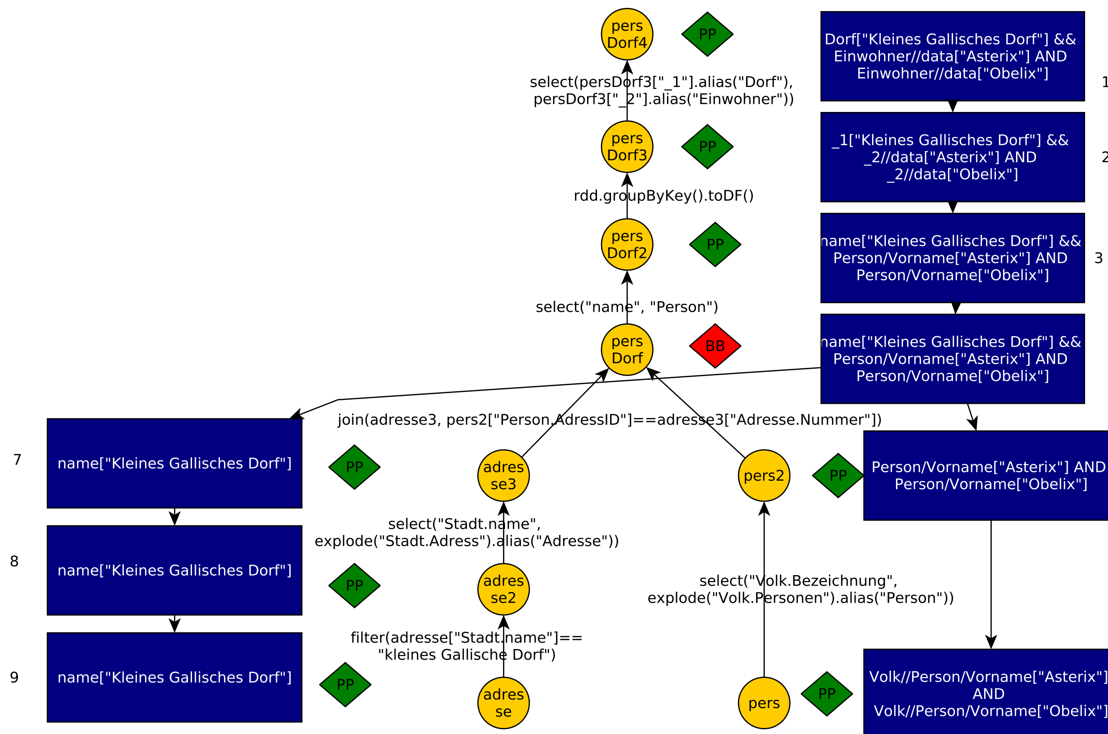

Abbildung 5.5.: Beispiel für einen Anfragebaum mit zugehörigen Annotations ohne Berücksichtung der Anzahl der Elemente.

Beispiel 7: Dieses Beispiel ist eine Erweiterung des vorherigen. Diesmal werden die Anzahl der Tupel berücksichtigt und der vorherige Fehler ist verbessert, d. h. der join-Operator ist korrekt. Der Benutzer möchte jetzt wissen, wieso genau dieses Tupel fehlt. Dadurch dürfen keine weiteren Geschwister entstehen, obwohl in den Eingabedaten auch noch einige andere Bewohner für das Dorf aufgelistet sind. Deswegen sind zusätzlich die erlaubte minimale bzw. maximale Anzahl an Elementen und Kindern der ArrayTypes des Schrittes TraceAttribut dargestellt. Diese werden zusätzlich von dem SetAnnotation Schritt berücksichtigt. In den Operatoren *pers*, *pers2*, *adresse*, *adresse2* und *adresse3* ist die Anzahl zwar entscheidend, aber es ist nicht sicher, dass die Anzahl richtig (variable: AnzahlRichtig) ist, da minimale und maximale Anzahl nicht übereinstimmen. Dies ändert sich nach dem join-Operator, da die minimale und maximale Anzahl der Tupel dort genau zwei entsprechen muss. Danach kommen nur Operatoren, bei denen die Anzahl sicher zurückverfolgt werden kann. Tatsächlich gibt es aber drei Personen, womit die tatsächliche Anzahl der Tupels nicht in dem geforderten Bereich liegt. Deswegen ist diese Operation BB_{AE}, da es die erste Operation ist, bei der die Anzahl sicher nicht korrekt ist.

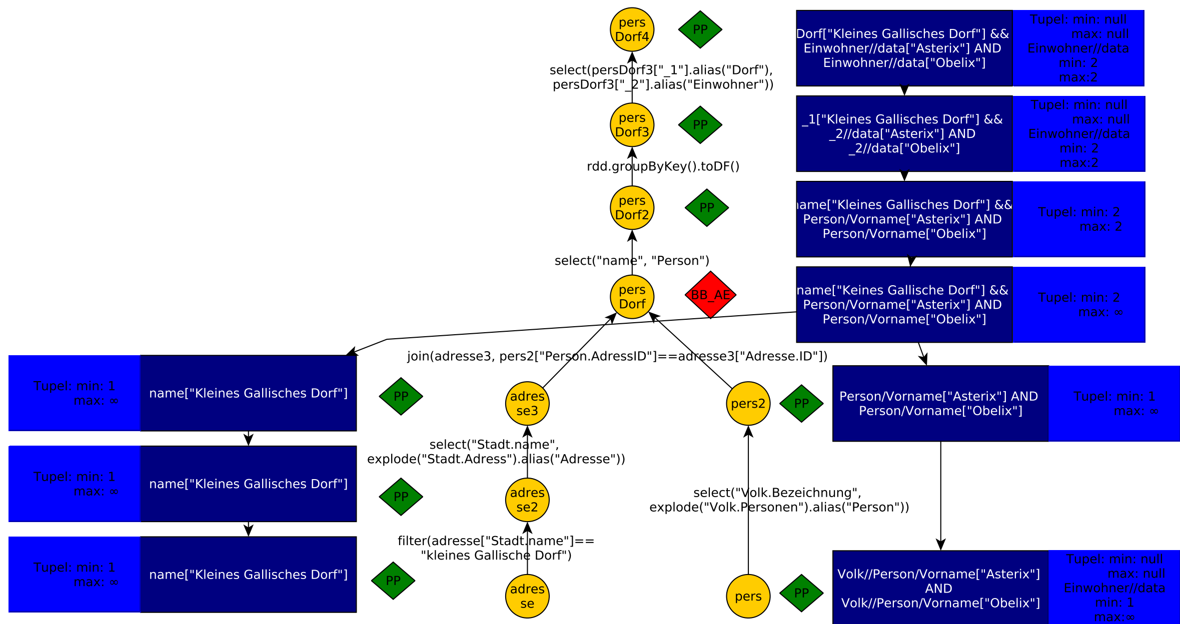

Abbildung 5.6.: Beispiel für einen Anfragebaum mit zugehörigen Annotations unter Berücksichtigung der Anzahl der Elemente

5.2. Abdeckung der Fragestellungen

Die Fragen, die der Algorithmus abdecken kann sind:

1. Wieso fehlt ein erwartetes Tupel?
6. Wieso existieren nicht x Tupel?
12. Wieso hat ein Element nicht den Wert w?
22. Wieso hat ein Knoten nicht x Kinder?
17. Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden?

Im Folgenden wird im Einzelnen erläutert, wie diese Fragen durch den bestehenden Algorithmus beantwortet werden können.

Ein Algorithmus für die Frage **Wieso fehlt ein erwartetes Tupel? (1)** kann stark an den bereits bestehenden relationalen Algorithmen, wie Why-not [9] angelehnt werden. Es kann festgestellt werden, welche Daten in welchem Zwischenschritt notwendig sind und an welcher Stelle dies nicht mehr der Fall ist. Daher ist der Algorithmus in der Lage diese Fragestellung zu beantworten.

Die Frage **Wieso existieren nicht x Tupel? (6)** kann auch mit der Frage wieso existieren nicht x variable Tupel zurückgeführt werden. Dadurch kann der Algorithmus die Frage indirekt, mithilfe der Frage 1, beantworten.

Wieso hat ein Element nicht den Wert w? (11) ist eine Frage, für die zwei Möglichkeiten bestehen, die unterschiedlich behandelt werden müssen:,
Zum einen erwartet der Benutzer vielleicht, dass ein bestimmter Wert bei einem bestimmten Tupel anders ist, während alle anderen Werte des Tupels gleich bleiben. Dazu muss der restliche Teil des Tupels mit angegeben werden, damit nicht im Anfragebaum der Wert von anderen SimpleTypes fälschlicherweise zu Passing führen kann. Diese Frage kann demzufolge indirekt mit der Frage, wieso fehlt ein bestimmtes erwartetes Tupel, beantwortet werden.
Zum anderen könnte der Benutzer erwarten, dass der Wert in einem nicht näher spezifizierten Tupel vorkommt.
In diesem Fall ist die Frage auch gleichbedeutend mit der Frage, wieso ein bestimmtes erwartetes Tupel fehlt. Allerdings können dabei alle Werte einen beliebigen Wert annehmen, abgesehen von dem SimpleType, das einen bestimmten Wert haben soll.

Die zweiundzwanzigste Frage (**Wieso hat ein Knoten nicht x Kinder**) kann so umformuliert werden, dass sie equivalent zu der Frage ist, wieso bestimmte Kinder existieren. Deshalb kann diese indirekt durch den Algorithmus beantwortet werden.

Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden? (27) Für diese Frage bestehen zwei Möglichkeiten, die unterschiedlich behandelt werden müssen:

1. Der Benutzer erwartet, dass nur der Teilbaum bei einem bestimmten Tupel sich von vorhandenen unterscheidet. Also bleiben alle anderen Werte des Tupels gleich. Dann muss der restliche Teil des Tupels mit angegeben werden, da sonst eventuell im Anfragebaum der Wert von anderen SimpleTypes fälschlicherweise zu Passing führen könnte. Damit wird diese Frage dann mit der Frage beantwortet, wieso ein bestimmtes erwartetes Tupel fehlt.
2. Der Benutzer erwartet nur, dass der Teilbaum in irgendeinem Tupel vorkommt.
   Dann ist die Frage auch gleichbedeutend mit der Frage, wieso ein bestimmtes erwartetes Tupel fehlt. Allerdings können dabei alle Typen einen beliebigen Wert annehmen, außer dem SimpleType, der einen bestimmten Wert haben soll.
   Beispiel 4.4 bezieht sich auf den 1. Fall, würde sich aber auf den 2. Fall beziehen, wenn es nur wichtig wäre, dass Asterix irgendwo wohnt, egal in welchem Dorf und mit welchen Nachbarn.

5.3. Einschränkungen des Basisalgorithmus

In diesem Kapitel werden die Einschränkungen des Basisalgorithmus beschrieben, die dazu führen, dass er für bestimmte Sparkanfragen nicht benutzt werden kann oder dass bestimmte Fehler und Probleme nicht erkannt werden können. Diese Probleme werden im Folgenden aufgezählt und beschrieben.

Nicht korrekte Operation Der Algorithmus benutzt die RDD-Operationen, um ein kompatibles Element zu erhalten. Dabei werden diese invers von dem erwarteten Ergebnis berechnet. Dies führt allerdings dazu, dass Fehler nicht erkannt werden können, die durch Umwandlung der Datenstruktur oder Anwendung von Funktionen entstehen.

Beispiel 8: In der Abbildung 5.7 ist ein Anfragebaum dargestellt, bei dem eine solche falsche Operation vorhanden ist. Der Anfragebaum ist identisch zu der Anfrage in 1.5. Es wird lediglich in dem letzten select Vor- und Nachname nicht berücksichtigt, da Asterix und Obelix keinen Nachnamen haben, sondern nur einen Vornamen. Trotzdem sollten Asterix und Obelix in den Ergebnisdaten vorhanden sein. Allerdings wird nach den Nachnamen gesucht, statt nach den Vornamen, sodass die letzte Operation falsch ist. Der Algorithmus wird dadurch bei dem Zurückverfolgen des Ergebnisses davon ausgehen, dass die Selektion korrekt ist und den Fehler in den Daten vermuten.

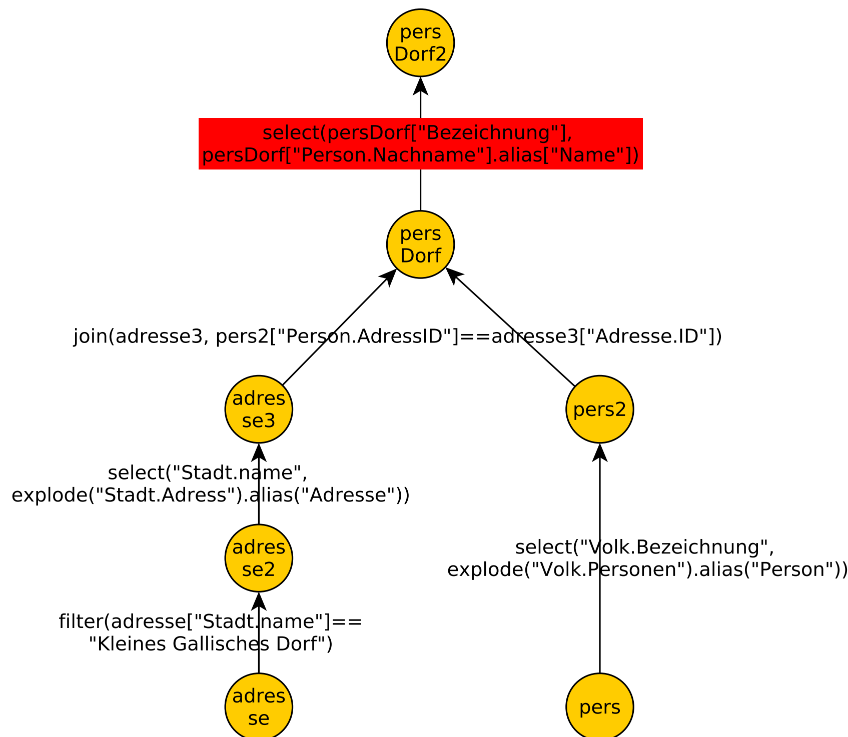

Abbildung 5.7.: R-DataFrame nach der Operation

TraceAttribut Lineage Tracing: Ein schwieriger Teil des Algorithmus ist das Zurückverfolgen (Lineage Tracing) der R-DataFrames von der Wurzel bis zu den Blättern, da der Algorithmus keine Funktionen als Übergabeparameter unterstützt. Dies ist prinzipiell nicht möglich, weil die Funktion nicht bekannt ist und daher beliebig viele Möglichkeiten entstehen.
Cui und Widom [13] unterscheiden bei dem Lineage Tracing zwischen drei Transformationsklassen dispatcher, aggregator und black box.
Ein Dispatcher ist eine Transformation, bei der jedes Eingabetupel mehrere Ausgabetupel erzeugen kann, unabhängig von den anderen Eingabetupels.
Bei einer Blackbox-Transformation gibt es keine Funktion, die die Lineage Tracing durchführen kann. Dadurch könnte jede Teilmenge der Eingabe dazu notwendig sein, die Ausgabe zu berechnen. Es ist aber auch möglich, dass kein Ausgabetupel erzeugt wird.
Für eine Blackbox-Transformation ist es unmöglich, die Daten zurückzuverfolgen, wenn die Eingabedaten nicht bekannt sind und damit unendlich viele Möglichkeiten bestehen. Da bei der Why-not-Frage von bestimmten fehlenden Eingabedaten ausgegangen wird, kann eine solche Transformation nicht vollständig und höchstens unzureichend durchgeführt werden.

Einschränkung von Operationen: Eine Annahme in dieser Arbeit ist, dass die Transformationen stabil und deterministisch sind. Dies trifft nicht auf alle RDD Operationen zu. So gibt die

sample-Funktion ein RDD mit einer zufälligen Teilmenge des ursprünglichen RDDs zurück.[1] Dieser Fall wird in dieser Arbeit, aufgrund der erhöhten Komplexität nicht näher behandelt. Auch Operationen, die beliebige Funktionen auf die Daten anwenden, werden nicht betrachtet. Zudem muss der Rückgabewert einer Operation ein DataFrame sein.

Implementation des TraceAttribut Schrittes für jede Operation: In dem Schritt TraceAttribut muss für jede RDD/DataFrame Operation extra ein Algorithmusteil entwickelt werden, der die Ausgabedaten eines DatenSchema/RDDs zu den Eingabeteilen zurückverfolgt. Dadurch werden nicht alle Funktionen unterstützt.

Doppelte Attributnamen: Jedes Attribut muss einen individuellen Namen haben. Dies gilt für alle RDDs, die bei dem vom Benutzer vorgegebenen Sparkcode erstellt werden. Dadurch wird die Aussagekraft des Algorithmus jedoch nicht eingeschränkt. Es wäre nämlich möglich, die RDDs auf Duplikate zu überprüfen und entsprechende Umbenennungen vorzunehmen. Diese könnten dann bei der Ausgabe wieder zurückverfolgt und in die ursprüngliche Version umbenannt werden.

Select: Um in dem aktuellen Algorithmus die Daten der Select-Operation in dem Schritt TraceAttribut zurückzuverfolgen, können nur Felder selektiert werden. Zusätzliche Änderungen der Spalten sind nicht erlaubt. Zu den nicht erlaubten Operationen zählen zum Beispiel $select(, df["age"] + 1)$.

Keine Unterstützung von MapTypes: Der Algorithmus unterstützt keine MapTypes als Datentypen.

Schema: Der Algorithmus setzt, anders als RDDs, ein festdefiniertes Schema voraus. Diese Voraussetzungen erfüllen DataFrames, allerdings nicht, wenn sich verschiedene Tupel in dem Schema unterscheiden.

Schema des erwarteten Ergebnisses: Das in dem erwarteten Ergebnis benutzte Schema muss der Ausgabe entsprechen oder ein Teil dieser Ausgabe sein. Es dürfen keine Attribute in dem erwarteten Schema vorkommen, die in dem Ergebnis der Anfrage nicht vorhanden sind.

Duplikate in den Daten: Falls in den Daten Duplikate notwendig sind, wird der Algorithmus einen Fehler nicht mehr finden, sobald eins dieser Objekte vorhanden ist. Es kann außerdem nicht festgestellt werden, ob zwei ChildElemente eines ArrayTypes Geschwister sind oder zwei unterschiedliche Eltern haben. Prinzipiell wäre eine Überprüfung mit ausreichend vielen Duplikaten möglich. Dazu müsste mit Zählern sichergestellt werden, dass in einem R-DataFrame mindestens genauso viele Duplikate in den Daten der Ausgabe vorhanden sind.
Allerdings kann bei einer Operation, die eine Datenhierarchie auflöst, nicht sicher festgestellt werden, ob mehrere R-DataFrames nicht zu ein und demselben vorherigen Tupel gehören. Es wäre nicht klar, ob Tupels, die sich nach der Operation nur in einem oder keinem Attribut

[1]https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/rdd/RDD.html#sample%28boolean,%20double,%20long%29

unterscheiden, vor der Operation zu demselben Tupel gehört haben oder nicht. Dadurch wäre eine Überprüfung Top-down höchstens bis zur ersten Auflösung einer Hierarchieebene sinnvoll, bei der zuvor Duplikate vorhanden sein können. Es wäre maximal möglich, zu berücksichtigen, bei welchen Attributen es nicht festgestellt werden kann, ob es sich um Duplikate handelt. Dies würde allerdings zu einer sehr großen Komplexität des Algorithmus führen. Dieses Problem wird in Beispiel 5.3 verdeutlicht.

Beispiel 9: Abbildung 5.8 zeigt die Daten nach der folgenden select-Operation:
$select(``Stadt.name", explode(``Stadt.Person"))$.
Der Algorithmus kann dabei nicht feststellen, ob die Daten vor der Operation denen in Abbildung 5.9 oder 5.10 entsprechen, da beide Eingabedaten zu demselben Ergebnis nach der select-Operation führen.

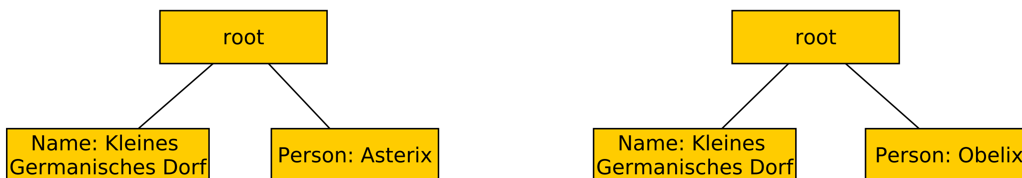

Abbildung 5.8.: R-DataFrame nach der Operation

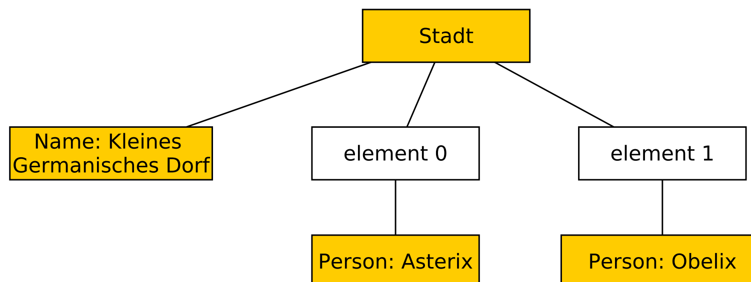

Abbildung 5.9.: Erste Option für R-DataFrame vor der Operation

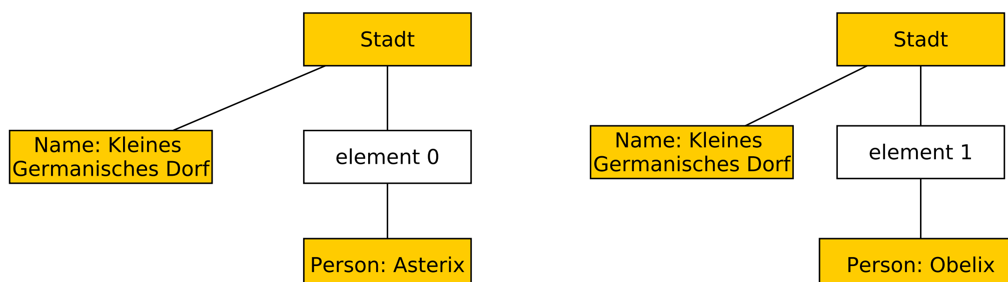

Abbildung 5.10.: Zweite Option für R-DataFrame vor der Operation

6. Prototyp

Um den Algorithmus zu testen, wurde ein Prototyp in Scala entwickelt, während die Oberfläche in JavaFX implementiert wurde.

6.1. Eingabe

Die Eingabe erfolgt über ein Textfeld mit der Syntax, die in Kapitel 3.4.1 definiert wurde. Die Oberfläche der Eingabe ist in Abbildung 6.1, die der Ausgabe in 6.2 dargestellt.

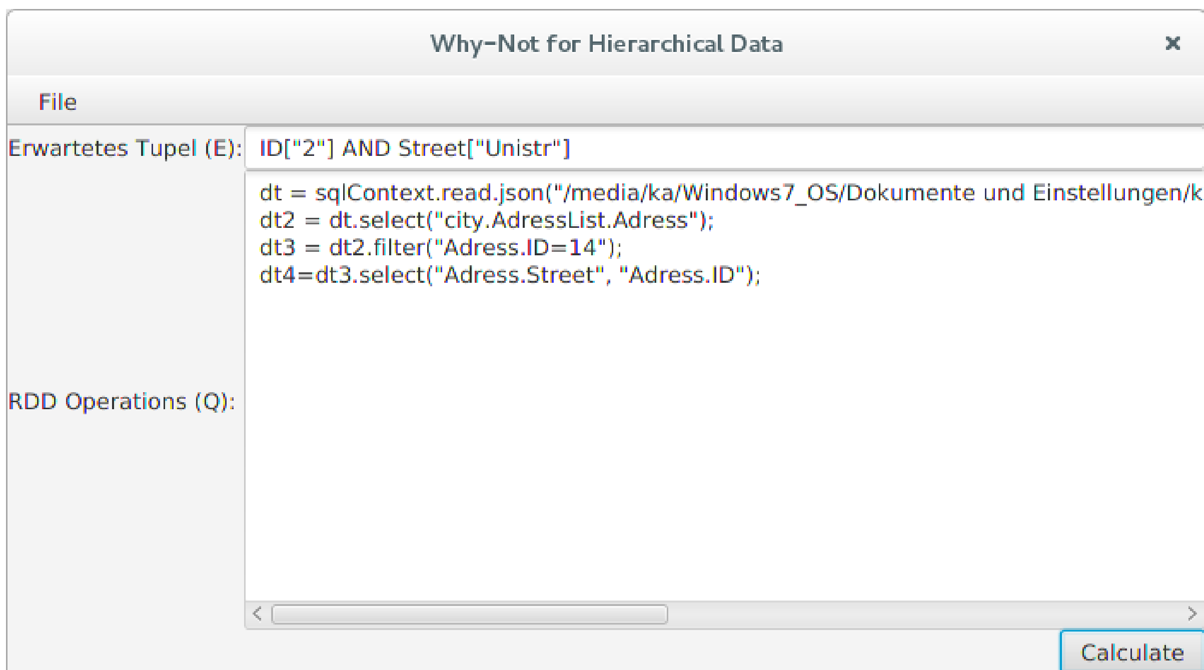

Abbildung 6.1.: Oberfläche für die Eingabe des Algorithmus

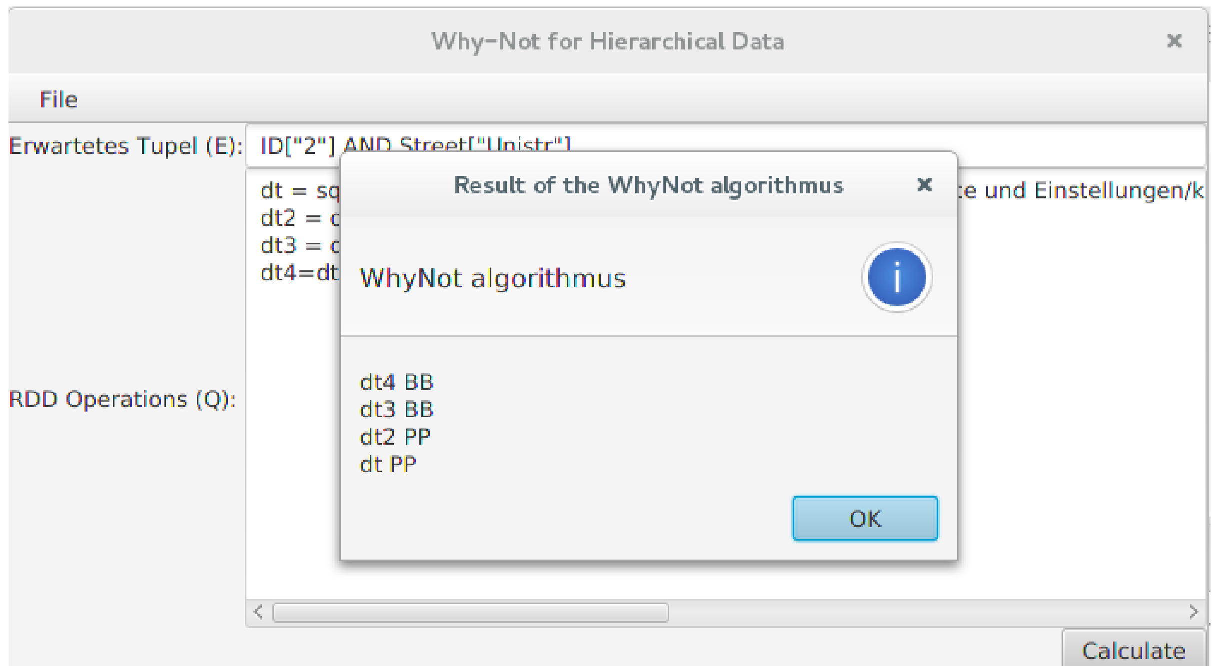

Abbildung 6.2.: Oberfläche für die Ausgabe des Algorithmus

6.2. Ausgabe

Bei der Ausgabe wird für jede DataFrame-Operation in einem Dialog die Annotation angezeigt und damit dargestellt, ob das DataFrame für den Fehler in der Anfrage verantwortlich sein kann. Die Ausgabe wird in Kapitel 3.4.2 genauer beschrieben.

6.3. Vorgehen

Im GenerateRequestTree wird der Baum in Objektform dargestellt. Dabei wird jeder Operationsknoten zu einem Objekt mit allen relevanten Eigenschaften umgewandelt.
Im Schritt TraceAttribut werden die notwendigen R-DataFrames als Objekte in den Operationsknoten aus dem GenerateRequestTree gespeichert. Dazu muss bei bestimmten Operatoren (z. B: join, groupBy) zuvor die Anfrage ausgeführt werden, um das Schema des vorherigen DataFrame zu bekommen. Für einen groupBy-Operator ist aus dem nachfolgenden DataFrame und dessen Operation ohne dies nicht klar, welche Typen das vorherige Schema hat. Bei der Ausführung wird die Anfrage so verändert, dass nach jeder Operation das DataFrame auf die Festplatte gespeichert und anschließend eingelesen wird. Die Ausführung der Sparkanfrage erfolgt allerdings nur einmal, dann werden die Ergebnisse der vorhandenen Ausführung benutzt.

In dem letzten Schritt (SetAnnotation) werden die Operationen ausgeführt, falls dies im vorherigen Schritt nicht erfolgt ist (vgl. 5.1.3).

6.4. Einschränkungen

Der Prototyp implementiert den in Kapitel 5 vorgestellten Algorithmus mit den im Folgenden dargestellten Einschränkungen.

Bedingungen: Der Prototyp unterstützt keine Bedingungen in dem erwarteten Ergebnis der Eingabe.
Datentypen: Der Basisalgorithmus unterstützt bisher nicht alle Datentypen, die in der formalen Definition von Kapitel 3.3 eingeführt wurden. Die unterstützten Datentypen sind Boolean, Integer und String.
JSON als Eingabe: Der Prototyp ist dafür ausgelegt, Daten aus einer JSON-Datei auszulesen. Bei anderen Eingabequellen kann es bei dem Prototypen zu Problemen kommen. Dies beschränkt aber nicht die Allgemeingültigkeit des Algorithmus, da dies nur von den Eingabemethoden abhängt.
Mehrere Tupels: Es wird nicht unterstützt, dass der Benutzer mehr als ein Tupel in dem erwarteten Ergebnis angibt.
GroupByKey: Bei dem Zurückverfolgen eines Tupels kann es bei dem GroupByKey-Operator zu Problemen kommen, da der Prototyp eventuell nicht korrekt feststellen kann, welcher Typ des R-DataFrame der Vorgänger des Schlüssels ist.
Zudem darf der Werttyp (ein ArrayType) als childAttribut nur ein SimpleTyp enthalten.
Erzeugen von korrekten Zwischenergebnissen: Der Algorithmus sieht bestimmte Daten auch dann als Eingabe für nachfolgende Schritte vor, wenn sie durch einen Operator verloren gehen. Diese werden für die Eingabe der nachfolgenden Schritte erzeugt. Dazu ist der Prototyp allerdings nicht in der Lage, weshalb nach einem solchen Fehler für jeden Nachfolger Blocking notiert wird, unabhängig von dessen wahrer Annotation.

7. Evaluation

Die Evaluation des Algorithmus wird mit Beispielen und verschiedenen Eingabedaten anhand des entwickelten Prototyps durchgeführt. Zuerst werden in diesem Kapitel Ziele und Testvoraussetzungen erläutert. Danach verschiedene Beispiele in Kapitel 7.3 erklärt, deren Laufzeiten Kapitel 7.4 evaluiert. In diesem Kapitel wird zusätzlich die Laufzeit durch mehrmaliges Ausführen einer Anfrage mit steigender Eingabemenge überprüft.

7.1. Ziele der Evaluation

Ziel der Evaluation ist es festzustellen, ob der Algorithmus innerhalb der in den Kapiteln 5.3 und 6.4 beschriebenen Einschränkungen, den Benutzer auf die fehlerhafte Stelle verweist, damit der Benutzer den eigentlichen Fehler schneller finden kann. Zusätzlich soll festgestellt werden, wie effizient die Implementierung in Bezug auf die Laufzeit ist.
Dazu werden verschiedene Beispielanfragen erstellt. Bei diesen wird darauf geachtet, dass sich die Anfragen unterscheiden und verschiedene Schwerpunkte haben. So gibt es Anfragen, die sehr viele Filteroperationen haben und andere, die Hierarchieebenen entschachteln und wieder verschachteln. Auch werden zwischen einer und drei Eingabedateien benutzt. Dabei werden auch verschiedene von dem Algorithmus unterstützte Fragen berücksichtigt. Allerdings können (siehe 6.4) im Prototyp nicht alle Fragen abgedeckt werden, da der Prototyp bisher keine Variablen unterstützt. Diese Fragen sind:

6. Wieso existieren nicht x Tupel?
22. Wieso hat ein Knoten nicht x Kinder?

Die anderen Fragen des Algorithmus können beantwortet werden:

1. Wieso fehlt ein erwartetes Tupel?
12. Wieso hat ein Element nicht den Wert w?
17. Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden?

7.2. Testdaten

Da es bisher keine Arbeiten gibt, die sich mit Why-not-Fragen für hierarchische Daten beschäftigen, wird es keine Benchmarking-Daten geben, die eins zu eins für diese Arbeit benutzt werden können. Somit werden für die Tests eigene Anfragen erstellt oder bestehende verändert. Teilweise wurden auch die Eingabedateien selbst erstellt.

7.3. Beispiele

In diesem Kapitel wird mit den verschiedenen Beispielen untersucht, wie akkurat der durch den Prototyp implementierte Algorithmus Operatoren identifiziert, die zum Fehlen erwarteter Daten bei der Why-Not-Frage führen.

7.3.1. Beispiel 1

Dieses Beispiel ist dem aus der Einleitung ähnlich. In der Anfrage werden alle Personen des kleinen, gallischen Dorfs ausgegeben, allerdings nur mit Vornamen. Dieses Beispiel gehört zu der siebzehnten Frage: Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden? Hier ist die konkrete Frage, wieso Asterix und Obelix in dem ArrayType Einwohner mit dem kleinen gallischen Dorf fehlen? Dabei sind die restlichen Geschwisterknoten nicht relevant. Im Test wird zuerst die fehlerhafte Anfrage und anschließend die korrigierte ausgeführt.
Die Anfrage ist als Baum in der Abbildung 7.1 dargestellt, während die Anfrage in Abbildung A.2 zu finden ist.

Eingabe

Die Eingabedateien, die implizit geladen werden, sind in Anhang A.2 und A.3 dargestellt.

Das erwartete Tupel für die Eingabe des Debugging-Szenarios ist:

```
Dorf["Kleines Gallisches Dorf"] AND Einwohner//data["Asterix"]
AND Einwohner//data["Obelix"] AND Einwohner//data[*]
```

Ausgabe des Prototyps

Es wird erwartet, dass alle Operatoren bis zu dem join Passing und danach Blocking sind, da der join fehlerhaft ist und statt der *AdressID* der Personen die Hausnummer benutzt wird. Es wird dabei nicht nur der join-Operator selbst als Blocking markiert, sondern auch alle nachfolgenden. Der Grund dafür liegt darin, dass der Prototyp die Erstellung von korrekten

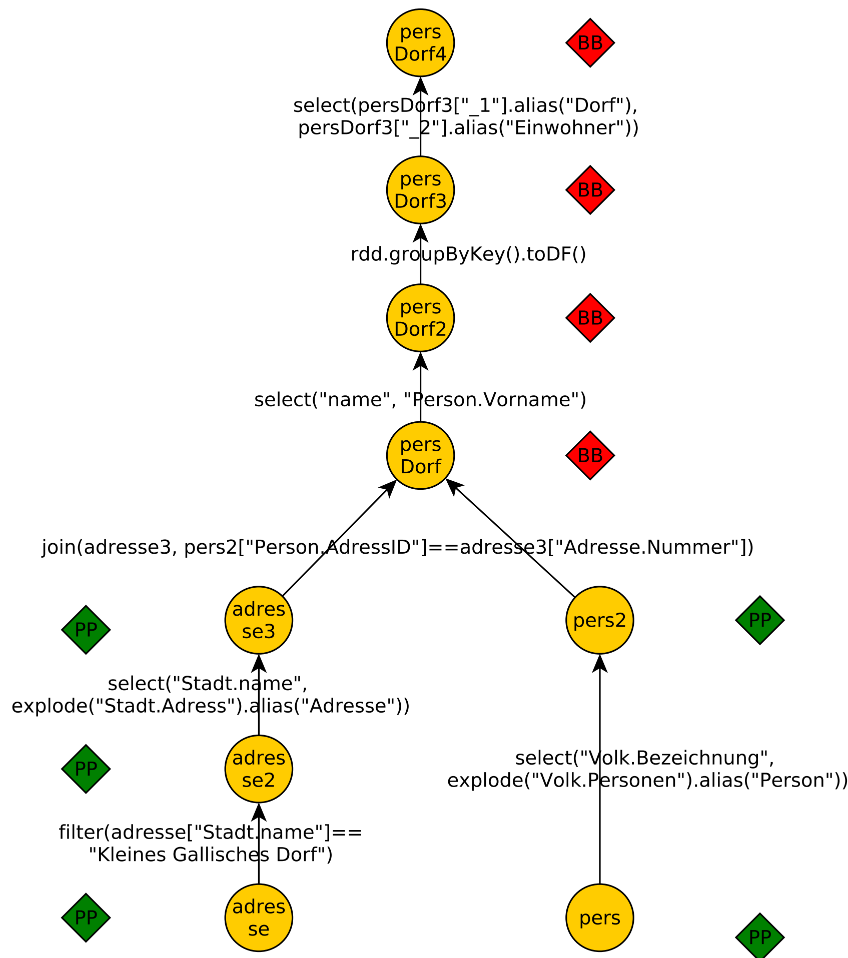

Abbildung 7.1.: Anfrage als Baum des ersten Evaluationsbeispiels

Zwischenergebnissen nach Blockierung durch einen Operator bisher nicht unterstützt (siehe 6.4).
Das Ergebnis des Prototyps ist in dem Anfragebaum in Abbildung 7.1 zu sehen. Es zeigt, dass der Algorithmus, wie erwartet, den join ebenso wie die nachfolgenden DataFrames als Blocking kennzeichnet.

Bei derselben verbesserten Anfrage (bei dem join Operator wird Adresse.Nummer durch Adresse.ID ersetzt) findet der Algorithmus korrekterweise keinen Fehler.

7.3.2. Beispiel 2

Dieses Beispiel unterscheidet sich von 7.3.1 darin, dass jetzt nur Asterix und Obelix in dem Dorf wohnen sollen. Zudem wurde der Fehler innerhalb der Anfrage behoben.
Durch dieses Beispiel wird getestet, dass der Algorithmus nicht fälschlicherweise einen fehlerhaften Operator identifiziert, den es nicht gibt. Diese ist in der Abbildung A.3 abgebildet. Als Baum befindet sich die Anfrage in Abbildung 7.2

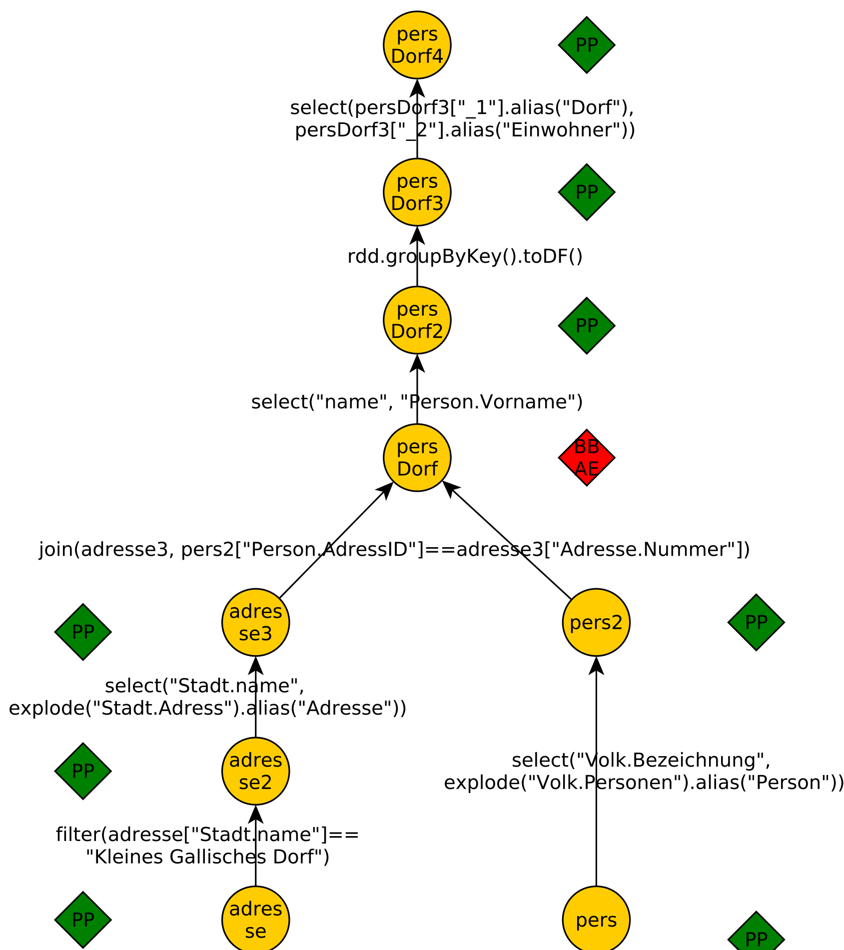

Abbildung 7.2.: Anfrage als Baum des ersten Evaluationsbeispiels

Eingabe

Die implizit geladene Eingabedatei wird in Anhang A.2 und A.3 dargestellt.

Das erwartete Tupel der Eingabe des Debugging Szenarios ist:

Dorf["Kleines Gallisches Dorf"] AND Einwohner//data["Asterix"]
AND Einwohner//data["Obelix"]

Ausgabe des Prototyps

Es wird erwartet, dass alle Operatoren, außer dem join, Passing sind und der join BB_{AE}. Für das Ergebnis in den Zwischenschritten sind nämlich alle notwendigen Daten vorhanden, am Ende existieren aber zu viele Bewohner. Der join Operator ist (Top-down) der erste Operator, bei dem dies sicher ist. Deswegen sollte er als BB_{AE} gekennzeichnet werden.
Das Ergebnis des Prototyps ist in den Anfragebaum (Abbildung 7.2) integriert. Es zeigt, dass der Algorithmus, wie erwartet, den join Operator als BB_{AE} kennzeichnet und die restlichen Operatoren als Passing.

7.3.3. Beispiel 3

In diesem Beispiel werden alle Autoren von wissenschaftlichen Arbeiten ausgegeben, deren Namen mit “Sch” anfangen. Für jeden dieser Autoren werden diejenigen mit aufgeführt, die mindestens eine Arbeit zusammen mit diesen veröffentlicht haben.
An diesem Fall ist, im Vergleich zum vorherigen, außergewöhnlich, dass es nur eine Eingabedatei gibt, die in dem Anfragebaum zuerst in zwei verschiedene Pfade aufgeteilt und später wieder zusammengeführt wird. Es erfolgen sowohl die Entschachtelung von Daten (z. B. in der Operation des DataFrames *authors*) als auch eine Verschachtelung durch groupByKey in der Operation des DataFrames *aPair1*. Diese Beispiel dient zur Kontrolle, dass keine Operatoren falsch gekennzeichnet werden.
Die Anfrage ist als Baum in der Abbildung 7.3 abgebildet, die Anfrage in der Abbildung A.4.

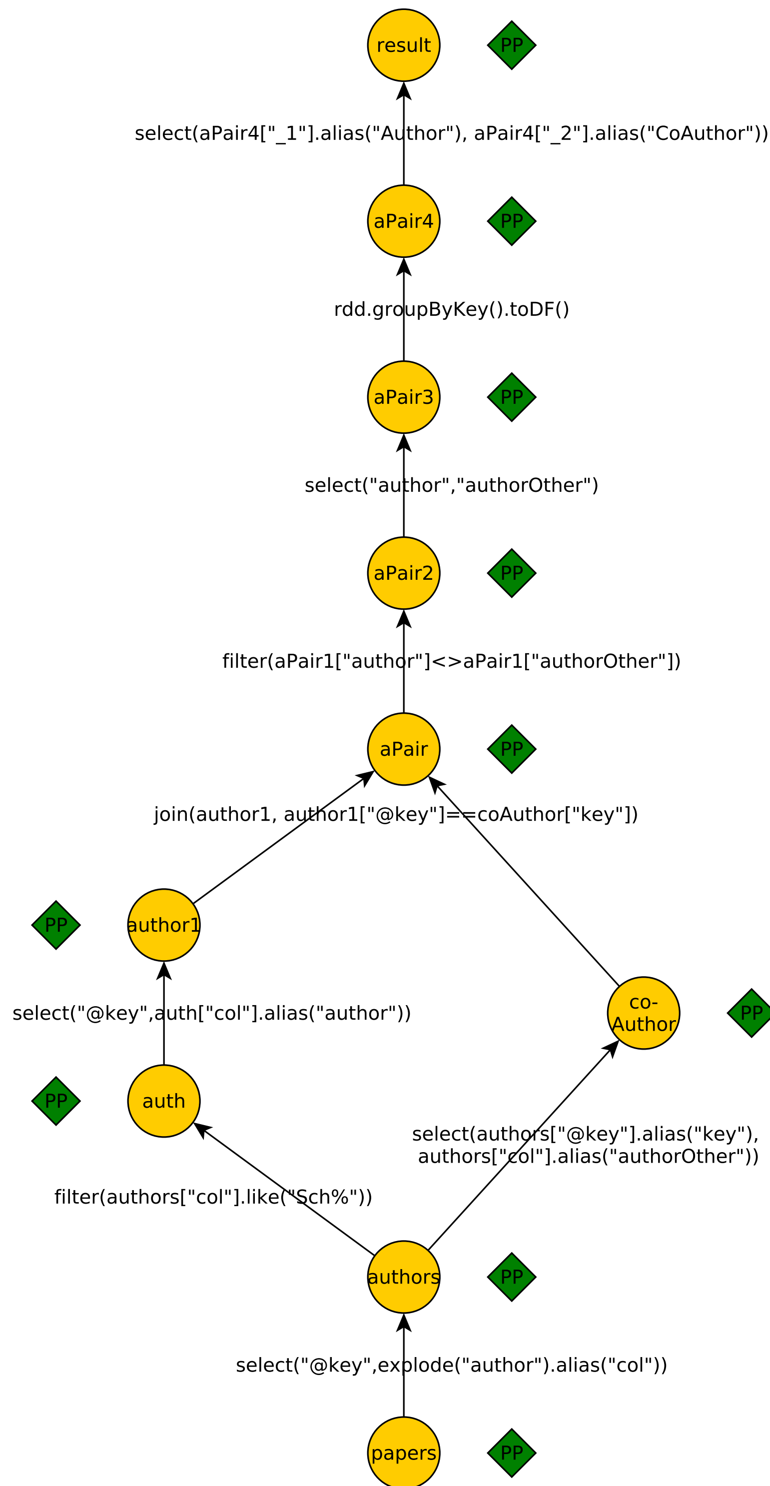

Abbildung 7.3.: Anfrage als Baum des dritten Evaluationsbeispiels

Eingabe

Als Datenquelle wurden Daten der dblp computer science bibliography[1] genutzt, die eine XML-Datei[2] mit bibliographischen Arbeiten im Informatikbereich zur Verfügung stellt. Damit Spark eine XML-Datei einlesen kann, wird für diese Anfrage zusätzlich die spark-xml Bibliothek[3] benötigt.
Die XML-Datei ist ca. 2 GB groß und enthält laut der Webseite[4] mehr als 3 Millionen wissenschaftliche Arbeiten von mehr als 1,5 Millionen Autoren.
Allerdings werden nicht alle Eingabedaten korrekt eingelesen, da manche Tupel durch die Bibliothek nicht berücksichtigt werden.

Das erwartete Tupel der Eingabe des Debugging Szenarios ist:

```
Author["Schuyler Erle"] AND CoAuthor//data["Rich Gibson"]
```

Ausgabe des Prototyps

Es wird erwartet, dass alle Operationen Passing sind, da das erwartete Ergebnis der Anfrage vorhanden ist.
Das Ergebnis des Prototyps ist in Abbildung 7.3 dargestellt. Es zeigt, dass der Algorithmus, wie erwartet, alle DataFrames als PP kennzeichnet.

[1] http://dblp.uni-trier.de/
[2] http://dblp.dagstuhl.de/xml/release/dblp-2016-08-03.xml.gz
[3] https://github.com/databricks/spark-xml
[4] http://dblp.uni-trier.de/

7.3.4. Beispiel 4

Dieses Beispiel entspricht dem Beispiel in Kapitel 7.3.3, allerdings ist die Filterfunktion der Operation des DataFrames aPair2 falsch. Es werden nicht alle Co-Autoren aussortiert, die nicht in einer Arbeit Mitautoren sind. Stattdessen werden alle Autoren ausgefiltert, die nicht dem Autor selbst entsprechen. Dadurch wird im Ergebnis als Co-Autor für die Autoren nur ihr eigener Name stehen.
Es handelt sich um die Frage wieso fehlt ein erwartetes Tupel (1). Denn es sind alle Attribute definiert und es dürfen keine zusätzlichen Geschwister existieren. Die Anfrage ist als Baum in der Abbildung 7.4 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.5 dargestellt.

Eingabe

Die Eingabe entspricht der Eingabe des Beispiels 3 in Kapitel 7.3.3. Als Datenquelle wurden Daten der dblp computer science bibliography[5] genutzt. Dazu wurde eine XML-Datei mit bibliographischen Arbeiten im Informatikbereich von dblp computer science bibliography benutzt.[6]. Das erwartete Tupel der Eingabe des Debugging Szenarios ist:

```
Author["Schuyler Erle"] AND CoAuthor//data["Rich Gibson"]
```

Ausgabe des Prototyps

Es wird erwartet, dass alle DataFrames vor *aPair2*, Passing sind, da die Operation, die zu *aPair2* führt, der fehlerhafte filter Operator ist.
Das Ergebnis des Prototyps ist in Abbildung 7.4 dargestellt. Es zeigt, dass der Algorithmus, wie erwartet, alle DataFrames bis zu der Operation *aPair2* als PP kennzeichnet und anschließend als BB.

[5] http://dblp.uni-trier.de/
[6] http://dblp.dagstuhl.de/xml/release/dblp-2016-08-03.xml.gz

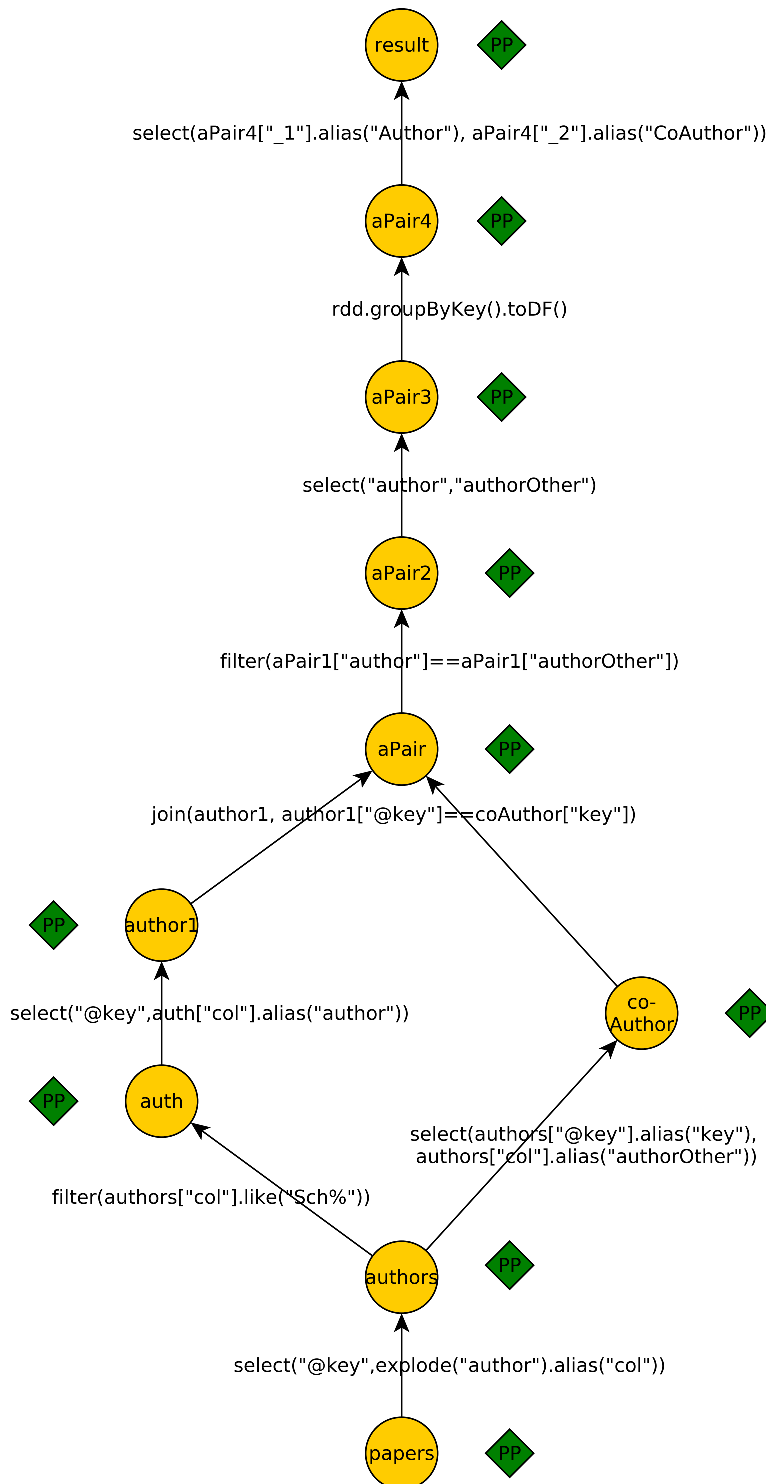

Abbildung 7.4.: Anfrage als Baum des vierten Evaluationsbeispiels

7.3.5. Beispiel 5

Dieses Beispiel entspricht dem Beispiel in Kapitel 7.3.3, allerdings wird in diesem Beispiel getestet, ob der Algorithmus auch, falls die Daten schon in der Eingabe fehlen, das richtige DataFrame als Blocking erkennt. Dies entspricht der Operation, die die Daten einliest.
Die Anfrage ist als Baum in der Abbildung 7.5 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.4 dargestellt.

Eingabe

Die Eingabedatei entspricht der des Beispiels 3 in Kapitel 7.3.3.
Das erwartete Ergebnis ist:

Author["Schuyler Erle"] AND CoAuthor//data["Max Mustermann"].

Max Mustermann ist allerdings in keiner Publikation Mitautor von Schuyler Erle.

Ausgabe des Prototyps

Es wird erwartet, dass bereits der DataFrame, der die Daten einliest, als Blocking gekennzeichnet ist.
Das Ergebnis des Prototyps ist in Abbildung 7.5 dargestellt. Es zeigt, dass bereits das Paper DataFrame Blocking ist. Dies entspricht dem erwarteten Resultat des Prototyps. Da keine kompatiblen Tupel durch den Prototyp erstellt werden, ist nicht nur das erste DataFrame Blocking, sondern alle bis auf *auth* und *author1*, da von diesen beiden Operatoren nur Schuyler Erle für die Ausgabe als Datensatz benötigt wurde.

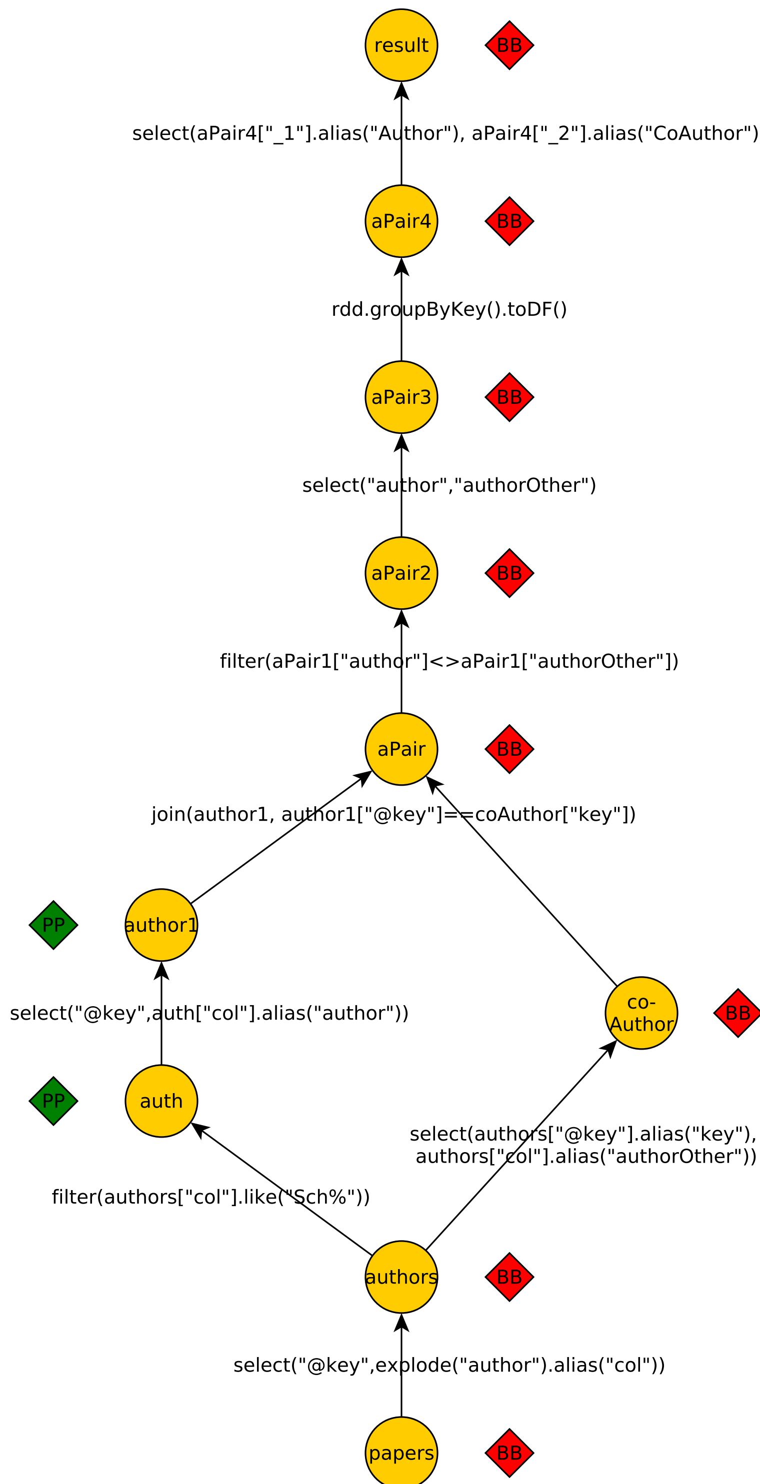

Abbildung 7.5.: Anfrage als Baum des fünften Evaluationsbeispiels

7.3.6. Beispiel 6

Dieses Beispiel besteht, außer den Leseoperationen, nur aus filter-Funktionen. Zudem hat die Anfrage keine Verzweigungen. Es werden alle Arbeiten ausgegeben, die nach dem Jahr 2000 veröffentlicht wurden, im Titel das Wort Provenance enthalten und als Buchtitel keinen leeren String haben.

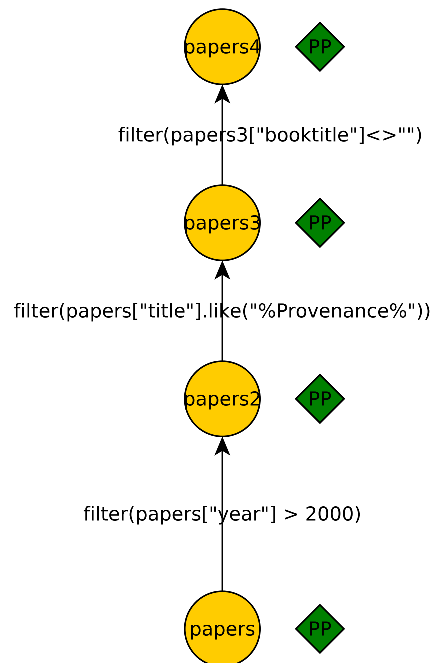

Abbildung 7.6.: Anfrage als Baum des sechsten Evaluationsbeispiels

Eingabe

Die Eingabedatei entspricht der Eingabe des Beispiels 3 in Kapitel 7.3.3.
Das erwartete Ergebnis ist:

```
title["Provenance: An Introduction to PROV"].
```

Ausgabe des Prototyps

Es wird erwartet, dass es keine Fehler gibt, da der Titel in den Eingabedaten vorhanden ist und alle Filter erfüllt.

Das Ergebnis des Prototyps ist in Abbildung 7.6 dargestellt. Es zeigt, dass, wie erwartet, alle Operatoren PP sind.

7.3.7. Beispiel 7

Dieses Beispiel entspricht dem vorherigen Beispiel, nur dass der erste Filteroperator falsch ist. Es wird fälschlicherweise nach Arbeiten gesucht, die vor 2000 veröffentlicht wurden.
Dieses Beispiel kann der Frage, wieso hat ein Element den Wert w, entsprechen. Es handelt sich um den zweiten Fall der Frage. Es wird erwartet, dass der Wert in einem beliebigen Tupel vorkommt. Die Anfrage ist als Baum in der Abbildung 7.7 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.7 dargestellt.

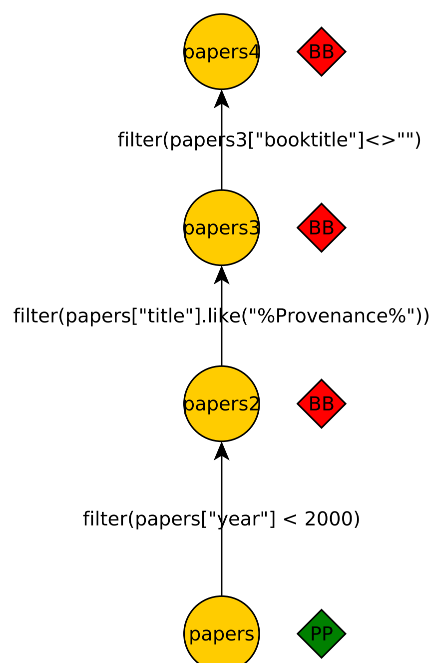

Abbildung 7.7.: Anfrage als Baum des siebten Evaluationsbeispiels

Eingabe

Die Eingabedatei entspricht der Eingabe des Beispiels 3 in Kapitel 7.3.3.
Das erwartete Ergebnis ist:

```
title["Provenance: An Introduction to PROV"].
```

Ausgabe des Prototyps

Es wird erwartet, dass alle DataFrames ab dem ersten Filter BB sind, da dort das Größer- und Kleinerzeichen falsch ist, also nur der erste DataFrame PP ist.
Das Ergebnis des Prototyps ist in Abbildung 7.7 dargestellt. Es zeigt, dass, wie erwartet, der erste Operator PP ist und alle restlichen BB sind.

7.3.8. Beispiel 8

In diesem Beispiel kommen keine filter-Funktionen vor, jedoch gibt es drei Eingabedateien und dadurch zwei joins. Es wird für jedes Buch der Serie Asterix ausgegeben, welche Personen mitspielen. Dabei werden nur Personen berücksichtigt, die keinen Nachnamen haben. Ausgegeben wird, neben der Person und dem Buch, das Dorf und die Adresse der Person. Dieses Beispiel dient zur Kontrolle, dass bei korrekter Anfrage und Eingabe keine Fehler erkannt werden.
Die Anfrage ist als Baum in der Abbildung 7.8 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.8 dargestellt.

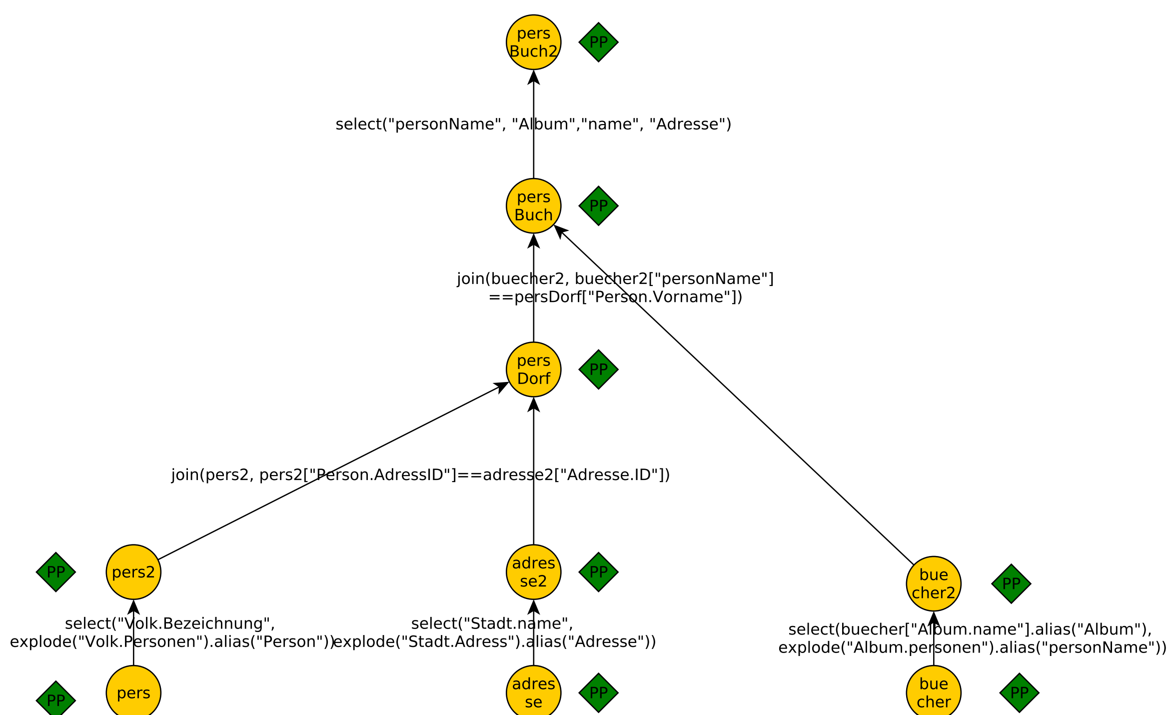

Abbildung 7.8.: Anfrage als Baum des achten Evaluationsbeispiels

Eingabe

Die Eingabedateien, die implizit geladen werden, sind in den Anhängen A.2, A.3 und A.4 dargestellt.
Es wird erwartet, dass es mindestens ein Buch gibt, in dem Asterix in dem kleinen gallischen Dorf in der Dorfstrasse wohnt. Formal sieht das erwartete Ergebnis wie folgt aus:

personName["Asterix"] AND name ["Kleines Gallisches Dorf”] AND Adresse/Straße["Dorfstrasse"]

Ausgabe des Prototyps

Da die Anfrage korrekt ist und alle Daten im Ergebnis existieren, wird erwartet, dass alle DataFrames PP sind.
Das Ergebnis des Prototyps ist in Abbildung 7.8 dargestellt. Es zeigt, dass, wie erwartet, alle Operatoren PP sind.

7.3.9. Beispiel 9

Dieses Beispiel entspricht dem vorherigen, nur ist der erste join fehlerhaft. Statt der *AdressID* wird nach der Hausnummer gefragt.
Es handelt sich um die Frage wieso fehlt ein erwartetes Tupel (1). Die Anfrage ist als Baum in der Abbildung 7.9 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.9 dargestellt.

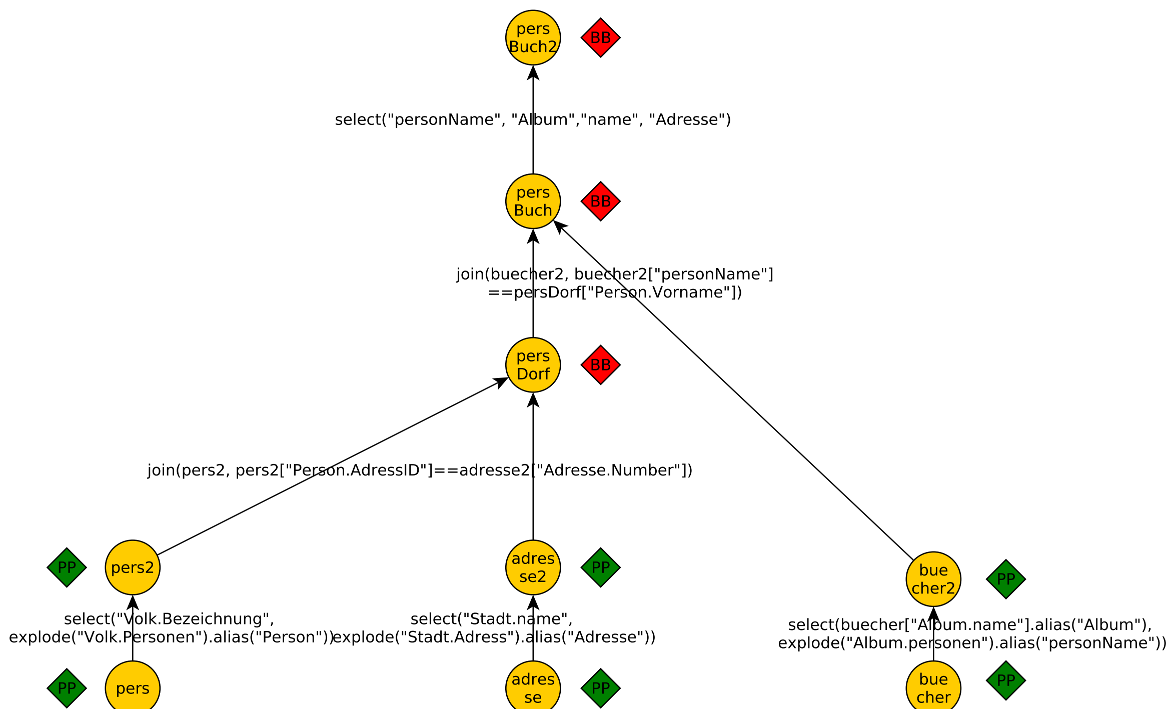

Abbildung 7.9.: Anfrage als Baum des neunten Evaluationsbeispiels

Eingabe

Die Eingabe entspricht der Eingabe aus Beispiel 8.

Ausgabe des Prototyps

Da das erwartete Ergebnis und die Anfrage, bis auf den ersten join, korrekt sind, wird erwartet, dass bis zu dem ersten join alle DataFrames PP sind.
Das Ergebnis des Prototyps ist in Abbildung 7.9 dargestellt. Es zeigt das erwartete Ergebnis.

7.3.10. Beispiel 10

Die Anfrage entspricht der Anfrage L3 des Benchmarks PigMix. Allerdings wurde die Anfrage von PigLatin zu Spark übertragen und die letzte SUM-Operation weggelassen.
Die Anfrage ist als Baum in der Abbildung 7.10 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.10 dargestellt.

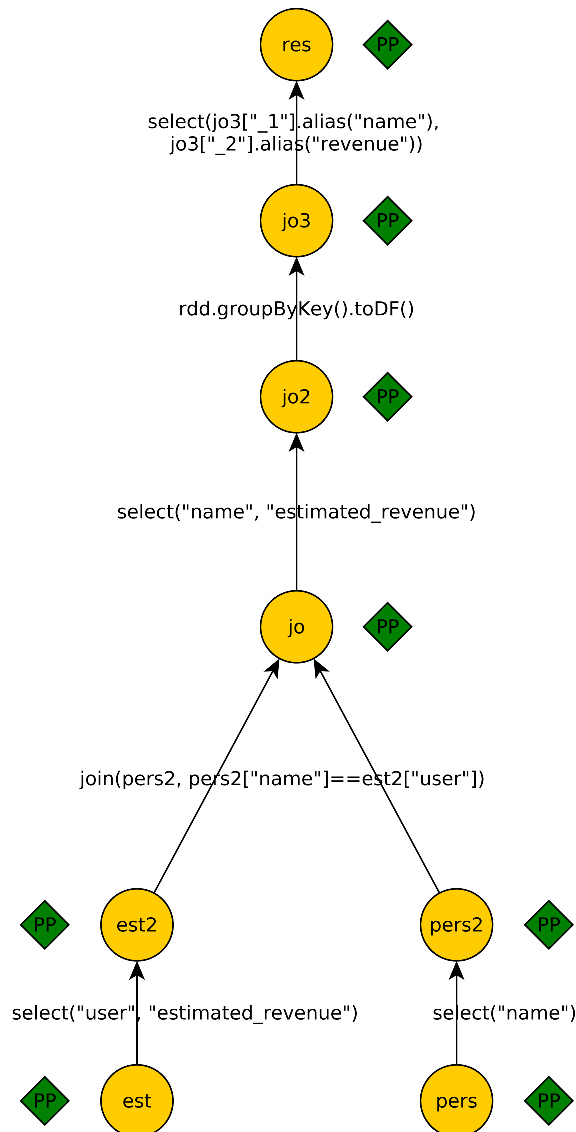

Abbildung 7.10.: Anfrage als Baum des zehnten Evaluationsbeispiels

Eingabe

Die Eingabedateien, die implizit geladen werden, sind keine Original Benchmark-Testdateien. Es wird erwartet, dass bei der Ausgabe ein Datensatz mit dem user Michael und ein Gehalt von 3000 existiert. Das erwartete Ergebnis ist formal:

name["Michael"] AND revenue//data[3000]

Ausgabe des Prototyps

Es wird erwartet, dass alle Operatoren korrekt sind, da der Datensatz mit der Anfrage wirklich aus den Eingabedaten generiert werden kann.
Das Ergebnis des Prototyps ist in Abbildung 7.10 bei den Anfragebaum dargestellt. Es bestätigt das korrekte Ergebnis.

7.3.11. Beispiel 11

Die Anfrage entspricht der Anfrage L3 des Benchmarks PigMix. Allerdings wurde die Anfrage von PigLatin zu Spark übertragen und die letzte SUM-Operation weggelassen.
Es handelt sich um die Frage, wieso hat ein Element nicht den Wert w? (12). Es existiert das erwartete Tupel, allerdings ist ein Wert falsch. Die Frage ist equivalent zu der Frage, wieso fehlt ein erwartetes Tupel (1). Nur dass in diesem Fall der Benutzer erwartet, dass das alte Tupel weiterhin besteht. Dadurch das der Algorithmus zwischen den Fragestellungen nicht näher differenzieren kann, ist es auch denkbar, dass der Fehler in bestimmten Fällen nicht korrekt erkannt wird (siehe Kapitel 5.3). Die Anfrage entspricht der aus dem Beispiel 10. Sie ist als Baum in der Abbildung 7.11 abgebildet und als Text in der Abbildung A.10.

Eingabe

Die Eingabedateien, die implizit geladen werden, sind keine Orginal Benchmark-Testdateien. Das erwartete Ergebnis ist ein Datensatz mit dem user Michael und ein Gehalt von 5000. Das erwartete Ergebnis ist formal:

name["Michael"] AND revenue//data[5000]

Ausgabe des Prototyps

In den Eingabedateien existiert der Benutzer Michael und auch ein Gehalt von 5000. Allerdings gehört keins dieser Gehälter zu dem Benutzer Michael. Deswegen wird erwartet, dass der join Blocking ist. Alle notwendigen Datensätze für die Ausgabe sind bis zu dem *join* vorhanden,

bei diesem Operator werden diese jedoch nicht zu einem Datensatz zusammengefügt.
Das Ergebnis des Prototyps ist in Abbildung 7.11 dargestellt. Es zeigt das erwartete Ergebnis.

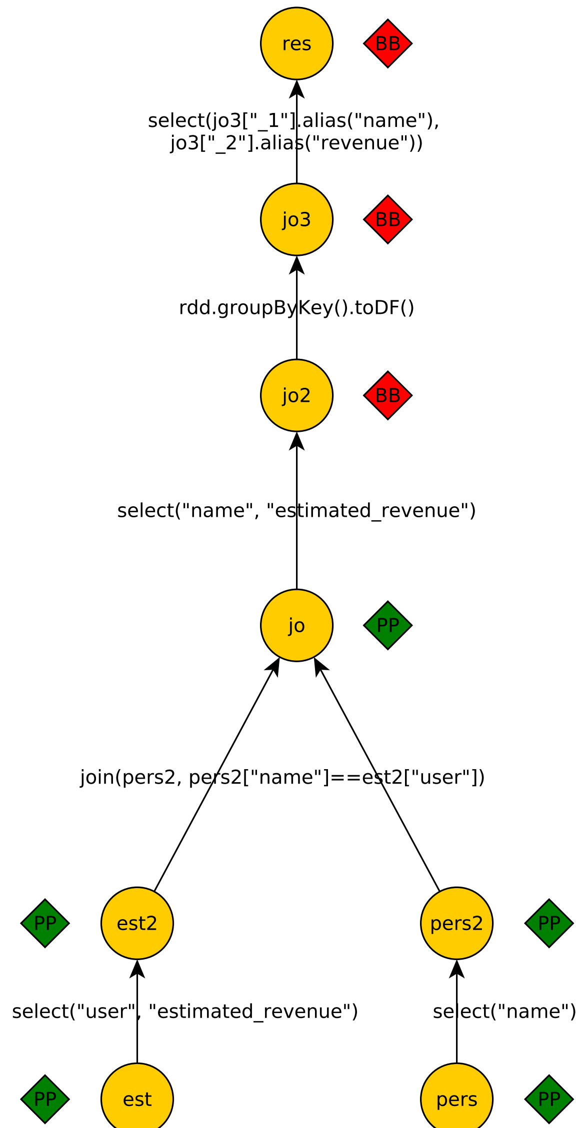

Abbildung 7.11.: Anfrage als Baum des zehnten Evaluationsbeispiels

7.4. Laufzeitanalyse

In der Laufzeitanalyse wurde für die vorherigen Beispiele die Laufzeit der verschiedenen Teile aus dem Algorithmus gemessen. Die Tests wurden mit Openstack und einer virtuellen Maschine durchgeführt, die 16 virtuelle CPUs und 16 GB RAM zur Verfügung stellt.

7.4.1. Laufzeit der Beispiele

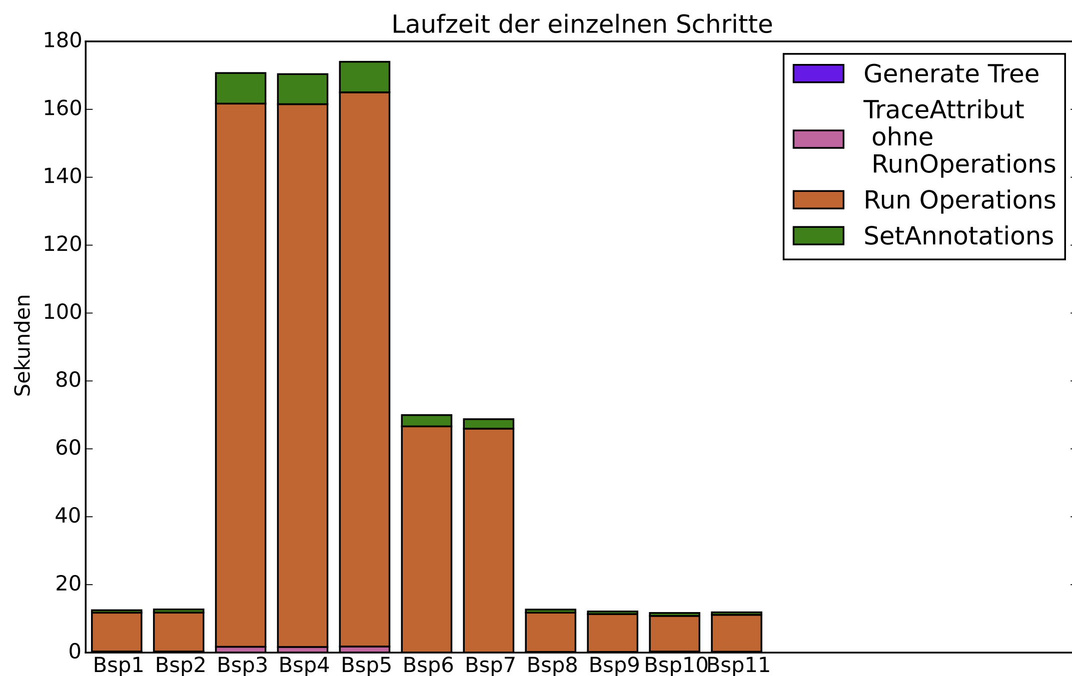

Abbildung 7.12.: Laufzeit der Evaluierungsbeispiele, aufgeteilt in die einzelnen Schritte

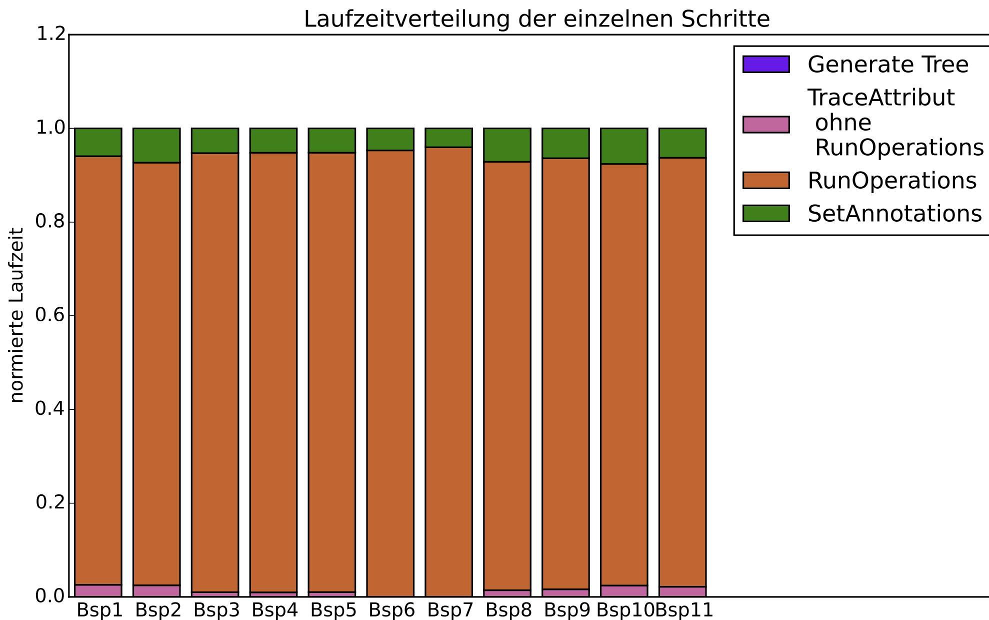

Abbildung 7.13.: Verteilung der Laufzeit der Evaluierungsbeispiele, aufgeteilt in die einzelnen Schritte.

Abbildung 7.12 zeigt für die verschiedenen Beispiele die Laufzeit der einzelnen Schritte des Algorithmus, während Abbildung 7.13 die Verteilung der Laufzeit der verschiedenen Schritte des Algorithmus zeigt.
Die beiden Diagramme zeigen den Mittelwert aus jeweils mindestens 25 Durchläufen.
Die Laufzeit der veränderten Datenbankanfragen (RunOperations) des Benutzers ist extra aufgelistet. Hierzu wird die Anfrage so verändert, dass die einzelnen DataFrames auf die Festplatte gespeichert werden, was in allen Beispielen im TraceAttribut-Schritt inbegriffen ist. Bei allen Beispielen nimmt dieser Teil der Ausführung fast die gesamte Laufzeit in Anspruch. Dies liegt daran, dass die Ausführung der Sparkanfrage, je nach Operatoren und Eingabedaten, einige Zeit in Anspruch nehmen kann. Zusätzlich verursacht das Abspeichern und das Einlesen der Zwischenergebnisse auf die Festplatte einen sehr großen Zeitverlust. Diese beiden Punkte führen auch zu der hohen Gesamtlaufzeit in den verschiedenen Beispielen, während die anderen Schritte im Verhältnis dazu sehr wenig Zeit in Anspruch nehmen.
Die Laufzeit des Schrittes GenerateTree beträgt in allen Beispielen nur den Bruchteil einer Sekunde, da die Sparkanfrage lediglich in eine Datenstruktur umgewandelt wird. Sie ist so gering, dass sie in den Abbildungen nicht sichtbar ist.
Auch der TraceAttribut-Schritt ohne die Ausführung der Anfrage fällt bei den Beispielen praktisch nicht ins Gewicht und liegt in der Größenordnung von wenigen Sekunden. Gleiches gilt für die Eingabe des erwarteten Ergebnisses, die ebenfalls keine Größe erreichen wird, bei der sich die Gesamtlaufzeit merklich vergrößert.

Der SetAnnotation Schritt nimmt, im Vergleich zu der Ausführung der Anfrage, nur einen kleinen Teil der Laufzeit in Anspruch.

7.4.2. Laufzeit mit verschiedenen Datengrößen

Zur genaueren Laufzeitanalyse wird eine Anfrage mit verschieden großen Eingabedaten getestet. Die Anfrage (Abbildung 7.14) ist angelehnt an die Anfrage des Beispiels in Teilkapitel 7.3.1. Es werden hierarchisch gespeicherte Personen eingelesen und mit ihren Adressdaten verknüpft. Am Ende werden die Vornamen der Personen nach Wohnort gruppiert.
Im Unterschied zur Anfrage in Kapitel 7.3.1 sind die Personen in den Eingabedaten nicht nach ihrem Volk sortiert. Zusätzlich ist der Filter nicht vorhanden, der nur Bewohner des kleinen gallischen Dorfes zulässt.

```
pers = sqlContext.read.json("dir2/testP1.json");
pers2 = pers.select(explode("Person").alias("Person"))
adresse = sqlContext.read.json("dir2/testA1.json");
adresse2=adresse.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=pers2.join(adresse2, pers2["Person.addressId"]==adresse2["Adresse.ID"])
persDorf2 = persDorf.select("name", "Person.Firstname")
persDorf3=persDorf2.rdd.groupByKey().toDF()
persDorf4 = persDorf3.select(persDorf3["_1"].alias("Dorf"), persDorf3["_2"].alias("Einwohner"))
```

Abbildung 7.14.: Anfrage für die Laufzeitanalyse

Eingabedaten und erwartetes Ergebnis

Das erwartete Ergebnis ist:

Dorf["Duesseldorf"] AND Einwohner//data["Thies"] AND
Einwohner//data["Max"] AND Einwohner//data["Klara"] AND Einwohner//data[*]

Es wird demzufolge erwartet, dass die Stadt Duesseldorf existiert und sie unter anderem Einwohner mit den Vornamen Thies, Max und Klara hat.
Das Schema der zugehörigen Adressdaten wird in Abbildung 7.15 dargestellt, die Personendaten in Abbildung 7.16. Es wurden Eingabedaten mit unterschiedlicher Anzahl an Personen und Adressen erstellt, jeweils bestehend aus 10 Eingabedateien für beide Teile. Für den ersten Test sind 50000 Personen bzw. Adressen gespeichert, bei jedem weiteren kommen 5000 Einträge pro Datei hinzu. Die Personen sind auf ein hierarchisches Tupel aufgeteilt. Die Adressdaten werden beliebig auf eine zufällige Anzahl von Städten verteilt.

$root\ |--\ Stadt:\ struct\ (nullable\ =\ true)$
$|\quad |--\ Adress:\ array\ (nullable\ =\ true)$
$|\quad |\quad |--\ element:\ struct\ (containsNull\ =\ true)$
$|\quad |\quad |\quad |--\ ID:\ string\ (nullable\ =\ true)$
$|\quad |\quad |\quad |--\ Nummer:\ string\ (nullable\ =\ true)$
$|\quad |\quad |\quad |--\ Stra„e:\ string\ (nullable\ =\ true)$
$|\quad |--\ name:\ string\ (nullable\ =\ true)$
$|\quad |--\ zip:\ string\ (nullable\ =\ true)$

Abbildung 7.15.: Schema der Adressdaten für die Laufzeitanalyse

$root$
$|--\ Person:\ array\ (nullable\ =\ true)$
$|\quad |--\ element:\ struct\ (containsNull\ =\ true)$
$|\quad |\quad |--\ Firstname:\ string\ (nullable\ =\ true)$
$|\quad |\quad |--\ Gender:\ string\ (nullable\ =\ true)$
$|\quad |\quad |--\ Lastname:\ string\ (nullable\ =\ true)$
$|\quad |\quad |--\ addressId:\ string\ (nullable\ =\ true)$
$|\quad |\quad |--\ isChild:\ string\ (nullable\ =\ true)$

Abbildung 7.16.: Schema der Personendaten für die Laufzeitanalyse

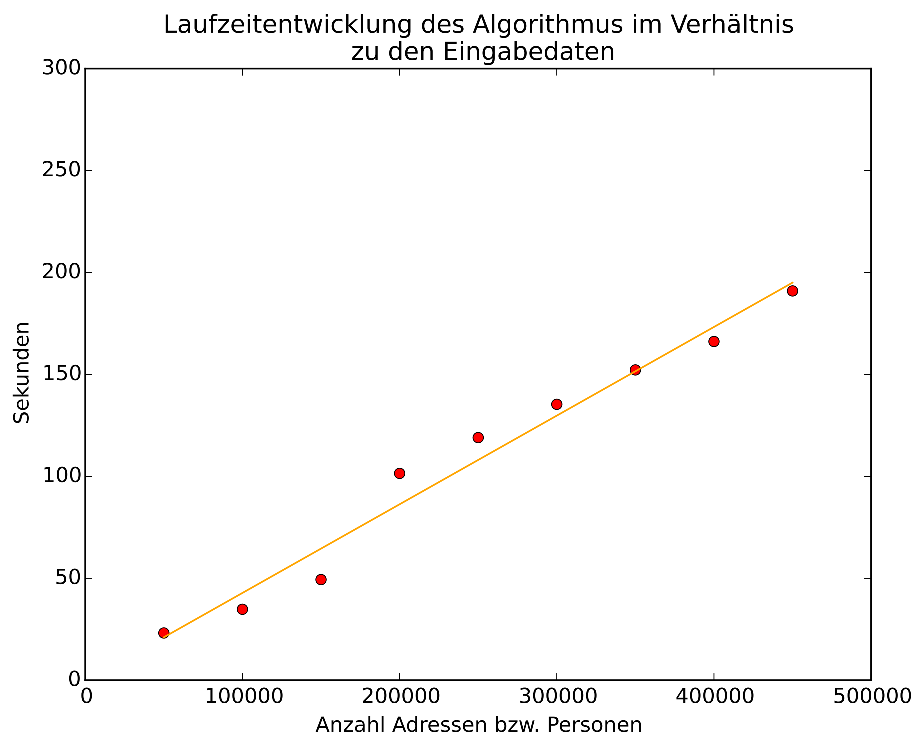

Abbildung 7.17.: Gesamtlaufzeit

Für die Eingabe werden immer Daten mit derselben Anzahl von Personen und Adressen benutzt. Dabei werden die Tests für alle Paare von Eingabedateien jeweils 10 Mal durchgeführt und die Laufzeiten gemittelt. Das Ergebnis für die Gesamtlaufzeit ist in Abbildung 7.17 dargestellt und zeigt einen linear ansteigenden Verlauf. Allerdings ist die Gesamtlaufzeit mit etwa drei Minuten bei jeweils 500000 Personen und 500000 Adressen sehr hoch. Um die Ursache dafür zu untersuchen, wird im Folgenden die Laufzeit der einzelnen Schritte des Algorithmus untersucht.

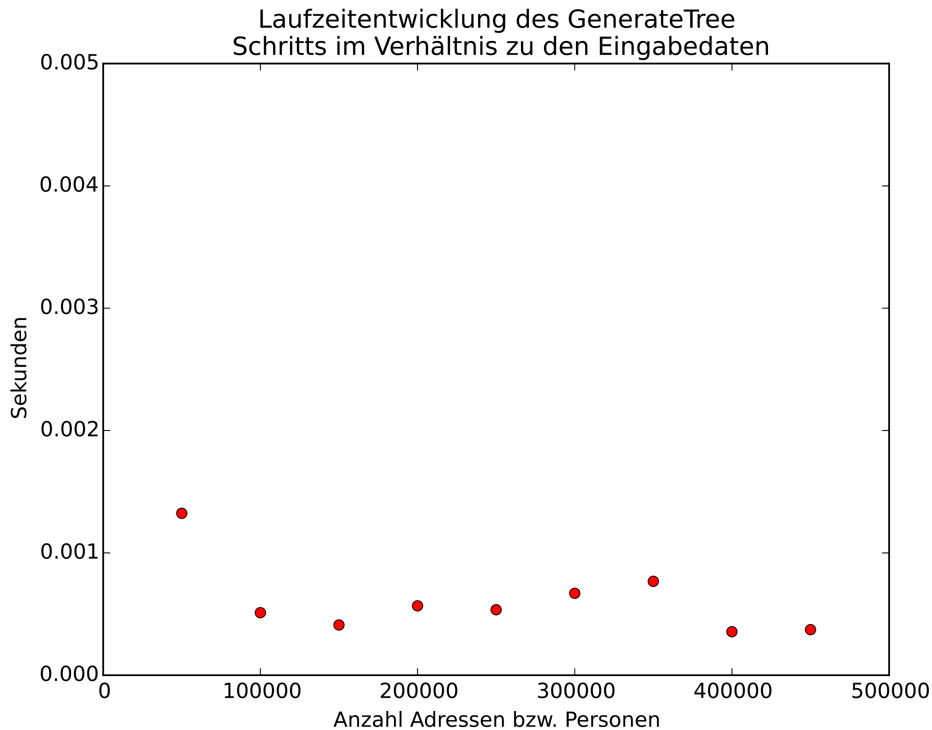

Abbildung 7.18.: Laufzeit des GenerateTree Schrittes des Algorithmus

Abbildung 7.18 zeigt die Laufzeit für das Erzeugen des Anfragebaums. Da sich die Anfrage zwischen den einzelnen Ausführungen nicht ändert, wird bei jeder Ausführung exakt dieselbe Berechnung durchgeführt. Deshalb ist auch kein Anstieg bzw. keine eindeutige Tendenz der Laufzeit in dem Diagramm zu erkennen. Die kleinen Unterschiede in der Laufzeit stammen von Einflüssen wie zum Beispiel unterschiedliche Zuteilung der Rechenleistung.
Die Laufzeit dieses Teils des Algorithmus entspricht dem Bruchteil einer Sekunde, womit das Erstellen des Anfragebaums nicht relevant für die Gesamtlaufzeit ist.

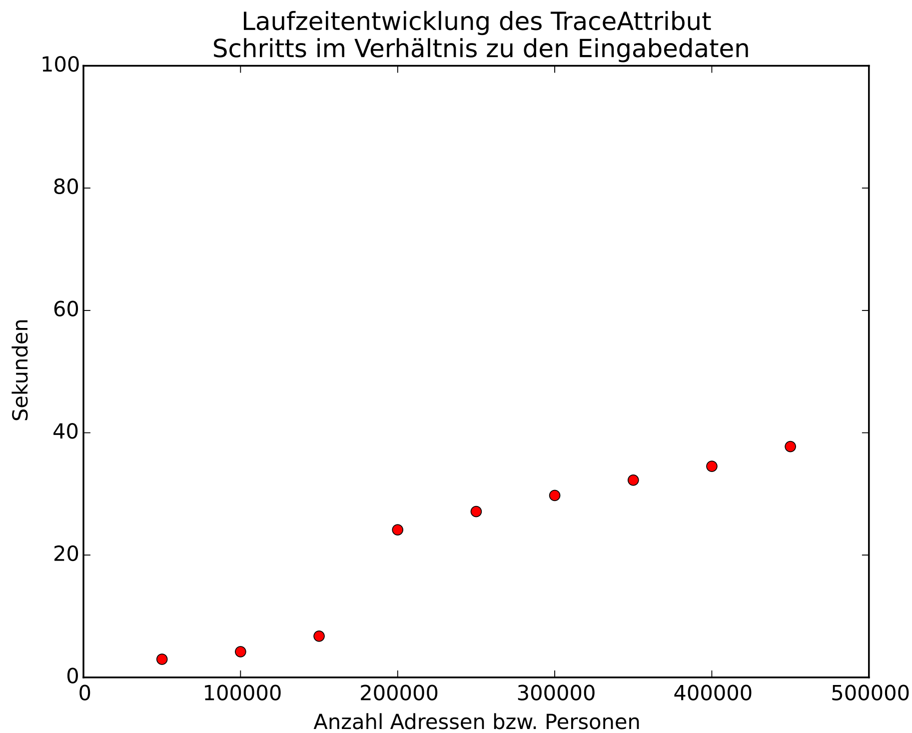

Abbildung 7.19.: Laufzeit des TraceAttributs Schrittes

Im nächsten Schritt des Algorithmus werden die notwendigen R-DataFrames für jeden Zwischenschritt ermittelt. Das Diagramm 7.19 zeigt die entsprechende Laufzeit für die Eingabedateien, die linear mit der Anzahl der Eingabedaten steigt. Allerdings gibt es zwischen dem dritten und vierten Test einen kleinen Sprung.
Dieser Schritt nimmt einen großen Teil der Laufzeit in Anspruch, im letzten Test etwa ein Fünftel. Die Laufzeit ist nicht konstant, obwohl immer dasselbe erwartete Ergebnis zurückverfolgt wird. Dies liegt an dem Zurückverfolgen des groupByKey- und des join-Operators. Bei diesen Operatoren ist es notwendig, das Schema des vorherigen DataFrame zu kennen. Dazu lädt der Prototyp die DataFrames, die als Eingabe für den join- bzw. groupByKey-Operator dienen. Die Größe dieser DataFrames nimmt zwischen den Tests zu und somit auch die benötigte Zeit, diese zu laden.
Das kann optimiert werden, da nur das Schema relevant ist und nicht die Daten, die für die lange Laufzeit verantwortlich sind.

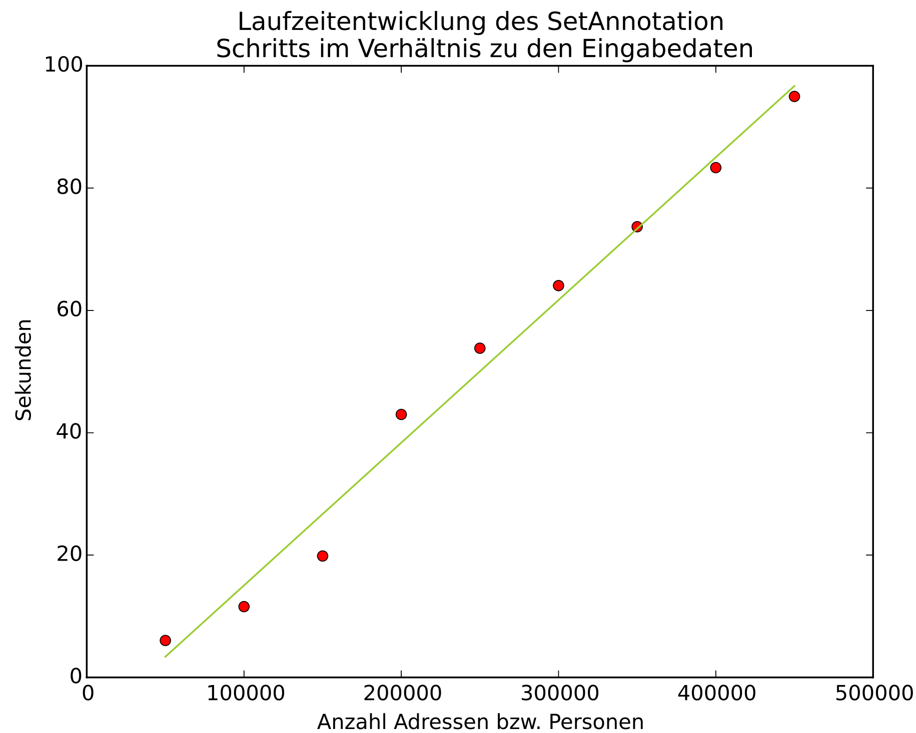

Abbildung 7.20.: Laufzeit des SetAnnotation Schrittes

Im letzten Schritt (SetAnnotation) des Algorithmus wird überprüft, welche DataFrames bzw. Operationen für das fehlerhafte Ergebnis der Anfrage verantwortlich sein können. Auch in diesem Schritt steigt die Laufzeit linear an, wobei es zwischen dem dritten und vierten Messwert einen Sprung gibt, dessen Grund nicht bekannt ist.
In diesem Schritt wird überprüft, ob die notwendigen Daten aus dem TraceAttribut-Schritt in den DataFrames der Benutzeranfrage vorhanden sind. Dazu werden diese DataFrames jeweils von der Festplatte eingelesen und nach den notwendigen Daten durchsucht, was der Grund für den linearen Anstieg in der Laufzeit ist.

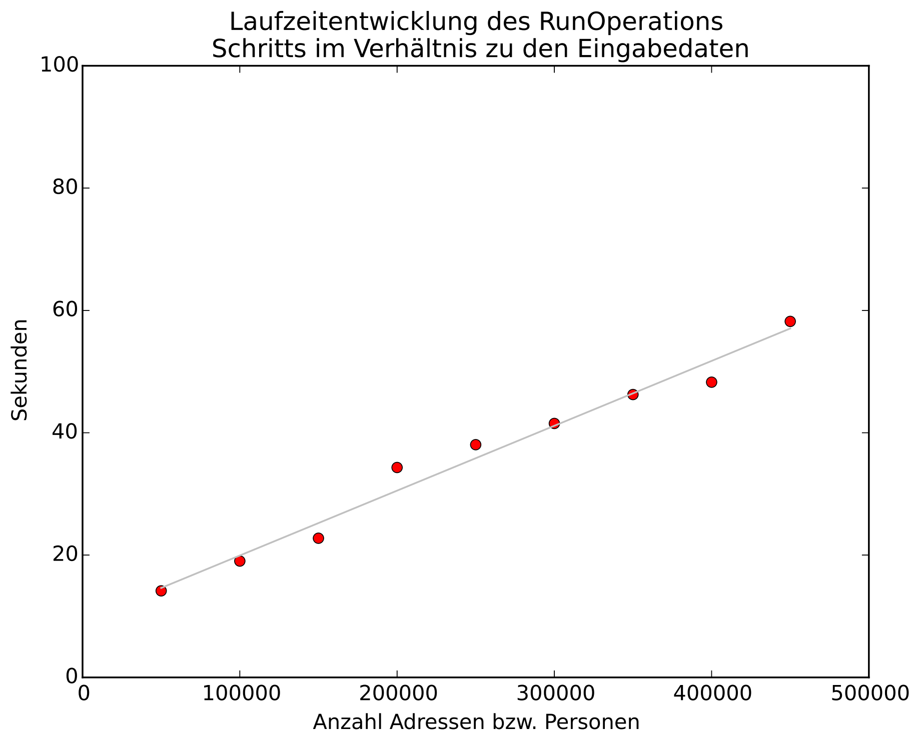

Abbildung 7.21.: Laufzeit der Ausführung der veränderten Benutzeranfrage

Gesondert wird hier das Ausführen der veränderten Spark-Anfragen betrachtet, bei denen neben der ursprünglichen Anfrage jedes DataFrame auf die Festplatte gespeichert wird. Dies führt, zusammen mit der Ausführung selbst, zu einer hohen Laufzeit der veränderten Anfrage. Diese steigt linear mit den Eingabedaten an. Von den verschiedenen Bereichen des Algorithmus nimmt der RunOperations-Schritt am Anfang am meisten Zeit in Anspruch. Daher ist es eine gute Stelle für Optimierungen. Insbesondere das Speichern der Zwischenergebnisse auf die Festplatte könnte eingespart werden.

Der Vergleich der verschiedenen Schaubilder zeigt, dass in den einzelnen Schritten die Laufzeit linear ansteigt. Allerdings unterscheiden sich die verschiedenen Steigungen. So steigt die Laufzeit des SetAnnotation-Schritts am schnellsten, darauf folgt RunOperation und mit der kleinsten Steigung TraceAttribut. Allerdings benötigt der RunOperation-Schritt bei sehr wenigen Daten eine längere Laufzeit als die beiden anderen Schritte. Dies verdeutlichen auch die Laufzeiten der Beispiele aus Kapitel 7.4.1, wo dieser Schritt die meiste Zeit in Anspruch nimmt. Es handelt sich nämlich jeweils um kleine Eingabedateien. Bei dem Beispieltest mit verschiedenen Eingabedateien dauert der Schritt RunOperation ab 200000 Adressen und Personen länger als SetAnnotation und TraceAttribut. Dies ist auch die Stelle, an der bei den meisten Diagrammen ein Sprung in der Laufzeit vorhanden ist, was daraufhin deutet, dass es für den Sprung und den Übergang zur längeren Laufzeit einen, damit zusammenhängenden

Grund gibt. Dies könnte mit Beschränkungen der virtuelle Maschine (z. B.: Arbeitsspeicher) zusammen hängen. Dies könnte mit Beschränkungen der virtuelle Maschine (z. B. Arbeitsspeicher) zusammen hängen. Beispielsweise könnte Spark ab dieser Datenmenge eine andere Optimierungs- oder Speicherverwaltung einsetzten. An dieser Stelle besteht noch Bedarf für nähere Analysen.

8. Zusammenfassung und Ausblick

Aufbauend auf bestehende Algorithmen für Why-not-Fragen wurde untersucht, ob es möglich ist, diese auf hierarchische Daten mit Spark zu übertragen. Zudem wurde ermittelt, welche neue Fragestellungen sich ergeben, wenn das erwartete Ergebnis nicht vorhanden ist. Während der Arbeit wurde festgestellt, dass es möglich ist, NedExplain auf hierarchische Daten zu erweitern. Dabei kann der Algorithmus auch die meisten wichtigsten Why-not-Fragen beantworten:

- Wieso fehlt ein erwartetes Tupel?
- Wieso existieren nicht x Tupels?
- Wieso hat ein Element nicht den Wert W?
- Wieso hat ein Knoten nicht x Kinder?
- Wieso ist ein erwartetes Kind eines Knotens (Teilbaum) nicht vorhanden?

Zur Beantwortung der Frage überprüft der Algorithmus, ob die für das erwartete Ergebnis notwendigen Zwischenergebnisse in jedem DataFrame der Anfrage existieren. Dabei ist das erwartete Ergebnis ein hierarchisches Tupel bzw. der Teil eines solchen Tupels, das bei einer Datenanfrage erwartet wurde, aber im Ergebnis nicht vorhanden ist.
Falls ein notwendiges Zwischenergebnis nicht vorhanden ist, ist dieses DataFrame für das Fehlen des erwarteten Ergebnisses verantwortlich. Es kann in dem Ergebnis auch ein Tupel geben, dass identisch mit dem erwarteten ist und nur zusätzliche Daten enthält. In diesem Fall verweist der Algorithmus auf das erste DateFrame der Anfrage, bei dem es sicher ist, dass zu viele Daten in dem Ergebnis vorhanden sind. Dies liegt daran, dass der Fehler nicht näher lokalisiert werden kann, da es sich um eine Why-provenance-Frage handelt.
Bei der Beantwortung von Fragen gibt es einige Einschränkungen bei dem entwickelten Algorithmus. Diese kommen insbesondere durch die Besonderheiten von Spark zustande. Das gilt vor allem aufgrund der Mächtigkeit von Spark, wie die Möglichkeit beliebige Funktionen als Übergabeparameter zu verwenden. Es gibt Sonderfälle, in denen der Algorithmus Fehler nicht richtig zuordnen kann. Die meisten Einschränkungen beziehen sich allerdings auf die Anfragen, die bearbeitet werden können.
Der Algorithmus musste die neuen Verbindungen berücksichtigen, die aufgrund von Hierarchie zwischen den Attributen entstehen. So müssen mehr Operationen berücksichtigt werden. Diese haben zum Teil auch eine höhere Komplexität, die eine neue Hierarchieebene entstehen lassen (z. B.: groupByKey) oder diese aufheben (z. B. explode innerhalb eines selects). Diese müssen einzeln in den Algorithmus eingefügt werden. Aus Zeitgründen war es nur für einen

Teil möglich, die Operationen in den Algorithmus zu integrieren.
Aufgrund der möglichen Hierarchie in dem Ergebnis der Anfrage, wurde für das erwartete Ergebnis eine neue Darstellung der Daten erstellt. Bei dieser wird auch berücksichtigt, dass der Benutzer eventuell nur Interesse an einem Teil der hierarchischen Daten hat. Dies wurde auch in dem Algorithmus entsprechend beachtet.
Um den Algorithmus zu evaluieren, wurde ein Prototyp mit dem Ziel möglichst hoher Effektivität entwickelt. Dieser ist in der Lage, die Fehler von unterstützten Anfragen richtig zu lokalisieren.

Ausblick

In zukünftigen Arbeiten empfiehlt es sich den Algorithmus weiter zu verbessern. Sowohl in der Effektivität als auch der Effizienz sind weitere Verbesserungen möglich. So ist es notwendig, den Algorithmus zu erweitern, um weitere Operationen zu unterstützen. Zudem kann dem Benutzer zusätzlich angezeigt werden, welche Attribute innerhalb der hierarchischen Daten dafür verantwortlich sind, dass der Operator fehlerhaft ist. Falls in einem Tupel zu viele Daten existieren, ist es möglich zu überprüfen, ob nicht einzelne Pfade für den Fehler verantwortlich sein können. Auch sollte die Unterstützung von Variablen in den Prototyp implementiert werden.
Bei der Effizienz kann der Prototyp verbessert werden. So kann das Abspeichern und anschließende Einlesen der Daten eingespart werden. Eine Möglichkeit dazu ist, die Datenanfrage so zu verändern, dass direkt bei dieser Anfrage getestet wird, ob die notwendigen Daten für einen Operator vorhanden sind.
Auch ist es möglich zu untersuchen, inwieweit die Entwickler bei der Fehlersuche für andere Datenbanksysteme unterstützt werden können (z. B. das grafische Datenbankmodell).

A. Dateien

```
pers = sqlContext.read.json("/Spark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("/Spark/files/sources/AdressenRA.json");
adresse2=adresse.filter(adresse["Stadt.name"]=="kleines Gallische Dorf")
adresse3=adresse2.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=pers2.join(adresse3, pers2["Person.AdressID"]==adresse3["Adresse.Nummer"])
persDorf2 = persDorf.select("Bezeichnung", "Person.Vorname", "Person.Nachname")
```

Abbildung A.1.: Beispiel Anfrage mit Spark für hierarchische Daten

A.1. Beispielanfragen aus der Evaluation

A.1.1. Beispiel 1

```
pers = sqlContext.read.json("/Spark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("/Spark/files/sources/AdressenRA.json");
adresse2=adresse.filter(adresse["Stadt.name"]=="kleines Gallische Dorf")
adresse3=adresse2.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=pers2.join(adresse3, pers2["Person.AdressID"]==adresse3["Adresse.Nummer"])
persDorf2 = persDorf.select("name", "Person")
persDorf3=persDorf2.rdd.groupByKey().toDF()
persDorf4 = persDorf3.select(persDorf3["_1"].alias("Dorf"), persDorf3["_2"].alias("Einwohner"))
```

Abbildung A.2.: Gruppierte Ausgabe aller Bewohner (Vorname) des kleinen, gallischen Dorfs

A.1.2. Beispiel 2

```
pers = sqlContext.read.json("/Spark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("/Spark/files/sources/AdressenRA.json");
adresse2=adresse.filter(adresse["Stadt.name"]=="kleines Gallische Dorf")
adresse3=adresse2.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=pers2.join(adresse3, pers2["Person.AdressID"]==adresse3["Adresse.ID"])
persDorf2 = persDorf.select("name", "Person")
persDorf3=persDorf2.rdd.groupByKey().toDF()
persDorf4 = persDorf3.select(persDorf3["_1"].alias("Dorf"), persDorf3["_2"].alias("Einwohner"))
```

Abbildung A.3.: Gruppierte Ausgabe aller Bewohner (Vorname) des kleinen, gallischen Dorfs

A.1.3. Beispiel 3 und Beispiel 5

```
papers = sqlContext.load(source="com.databricks.spark.xml", rowTag = 'book', path = '/home/ka/Dokumente/-
Masterarbeit/daten/dblp.xml')
authors = papers.select("@key",explode("author").alias("col"))
auth = authors.filter(authors["col"].like("Sch%"))
author1=auth.select("@key",auth["col"].alias("author"))
coAuthor=authors.select(authors["@key"].alias("key"), authors["col"].alias("authorOther"))
aPair1=coAuthor.join(author1, author1["@key"]==coAuthor["key"])
aPair2 = aPair1.filter(aPair1["author"]<>aPair1["authorOther"])
aPair3=aPair2.select("author","authorOther")
aPair4 = aPair3.rdd.groupByKey().toDF()
result=aPair4.select(aPair4["_1"].alias("Author"), aPair4["_2"].alias("CoAuthor"))
```

Abbildung A.4.: Ausgabe von allen Mitautoren der Autoren, deren Namen mit “Sch“ anfängt.

A.1.4. Beispiel 4

```
papers = sqlContext.load(source="com.databricks.spark.xml", rowTag = 'book', path = '/home/ka/Dokumente/-
Masterarbeit/daten/dblp.xml')
authors = papers.select("@key",explode("author").alias("col"))
auth = authors.filter(authors["col"].like("Sch%"))
author1=auth.select("@key",auth["col"].alias("author"))
coAuthor=authors.select(authors["@key"].alias("key"), authors["col"].alias("authorOther"))
aPair1=coAuthor.join(author1, author1["@key"]==coAuthor["key"])
aPair2 = aPair1.filter(aPair1["author"]==aPair1["authorOther"])
aPair3=aPair2.select("author","authorOther")
aPair4 = aPair3.rdd.groupByKey().toDF()
result=aPair4.select(aPair4["_1"].alias("Author"), aPair4["_2"].alias("CoAuthor"))
```

Abbildung A.5.: Ausgabe von allen Mitautoren der Autoren, deren Namen mit "Sch" anfängt.

A.1.5. Beispiel 6

Operation

Die Anfrage ist als Baum in der Abbildung 7.6 abgebildet. Die Anfrage des Beispiels wird in der Abbildung A.6 dargestellt.

```
papers = sqlContext.load(source="com.databricks.spark.xml", rowTag = 'book', path = '/home/ka/Dokumente/-
Masterarbeit/daten/dblp.xml')
papers2 = papers.filter(papers["year"] > 2000)
papers3 = papers2.filter(papers["title"].like("%Provenance%"))
papers4 = papers3.filter(papers3["booktitle"]<>"")
```

Abbildung A.6.: Ausgabe von allen Arbeiten, die nach 2000 veröffentlicht wurden, das Wort Provenance enthalten und keinen leeren Buchtitel haben.

A.1.6. Beispiel 7

```
papers = sqlContext.load(source="com.databricks.spark.xml", rowTag = 'book', path = '/home/ka/Dokumente/-
Masterarbeit/daten/dblp.xml')
papers2 = papers.filter(papers["year"] < 2000)
papers3 = papers2.filter(papers["title"].like("%Provenance%"))
papers4 = papers3.filter(papers3["booktitle"]<>"")
```

Abbildung A.7.: Ausgabe von allen Arbeiten, die vor 2000 veröffentlicht wurden, das Wort Provenance enthalten und keinen leeren Buchtitel haben.

A.1.7. Beispiel 8

```
pers = sqlContext.read.json(SSpark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("Spark/files/sources/AdressenRA.json");
buecher = sqlContext.read.json("Spark/files/sources/BuecherRA.json");
buecher2=buecher.select(buecher["Album.name"].alias("Album"), explode("Album.personen").alias("personName"))
adresse2=adresse.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=adresse2.join(pers2, pers2["Person.AdressID"]==adresse2["Adresse.ID"])
persBuch=persDorf.join(buecher2, buecher2["personName"]==persDorf["Person.Vorname"])
persBuch2 = persBuch.select("personName", "Album","name", "Adresse")
```

Abbildung A.8.: Ausgabe von allen Personen mit Herkunftsort und Adresse für alle Bücher.

A.1.8. Beispiel 9

```
pers = sqlContext.read.json(SSpark/files/sources/PersonenRA.json");
pers2 = pers.select("Volk.Bezeichnung", explode("Volk.Personen").alias("Person"))
adresse = sqlContext.read.json("Spark/files/sources/AdressenRA.json");
buecher = sqlContext.read.json("Spark/files/sources/BuecherRA.json");
buecher2=buecher.select(buecher["Album.name"].alias("Album"), explode("Album.personen").alias("personName"))
adresse2=adresse.select("Stadt.name", explode("Stadt.Adress").alias("Adresse"))
persDorf=adresse2.join(pers2, pers2["Person.AdressID"]==adresse2["Adresse.Nummer"])
persBuch=persDorf.join(buecher2, buecher2["personName"]==persDorf["Person.Vorname"])
persBuch2 = persBuch.select("personName", "Album","name", "Adresse")
```

Abbildung A.9.: Ausgabe von allen Personen mit Herkunftsort und Adresse für alle Bücher.

A.1.9. Beispiel 10 und Beispiel 11

```
est = sqlContext.read.json("Spark/files/sources/PigMix/GehaltL3.json");
est2 = est.select("user", "estimated_revenue")
pers = sqlContext.read.json("Spark/files/sources/PigMix/userL3.json");
pers2 = pers.select("name")
jo = est2.join(pers2, pers2["name"]==est2["user"])
jo2 = jo.select("name", "estimated_revenue")
jo3 = jo2.rdd.groupByKey().toDF()
res=jo3.select(jo3["_1"].alias("name"), jo3["_2"].alias("revenue"))
```

Abbildung A.10.: Ausgabe von Gehältern für Benutzer.

A.1.10. Beispiel 11

A.2. Adressdaten für das erste Beispiel der Evaluation


```
{"Stadt":{"name": "kleines Gallische Dorf", "zip": "54323", "Adress": [{
    "ID": 5,"Strae": "Dorfstrasse", "Nummer": 12}]}}
{"Stadt":{"name": "Massila", "zip": "54323", "Adress": [{"ID": 6,"Strae":
    "Hauptstrasse", "Nummer": 423}]}}
{"Stadt":{"name": "Lyon", "zip": "32645", "Adress": [{"ID": 7,"Strae":
    "Strasse A", "Nummer": 423},{"ID": 13,"Strae": "Hauptstrae", "Nummer": 3
    }]}}
{"Stadt":{"name": "Rom", "zip": "32645", "Adress": [{"ID": 12,"Strae":
    "Regierungsviertel", "Nummer": 423},{"ID": 4,"Strae": "Olgastae",
    "Nummer": 2},{"ID": 1,"Strae": "Hauptstrae", "Nummer": 1}]}}
```


A.3. Personendaten für das erste Beispiel der Evaluation


```
{"Volk":{"Bezeichnung": "Gallier", "Personen":[{"Vorname": "Asterix",
    "Nachname": "", "AdressID" :5},{"Vorname": "Obelix", "Nachname": "",
    "AdressID" :5},{"Vorname": "Troubadix", "Nachname": "", "AdressID" :5},{
    "Vorname": "Idefix", "Nachname": "", "AdressID" :5},{"Vorname":
    "Automatix", "Nachname": "", "AdressID" :5},{"Vorname": "Miraculix",
    "Nachname": "", "AdressID" :5},{"Vorname": "Minna", "Nachname": "",
    "AdressID" :6},{"Vorname": "Schnfix", "Nachname": "", "AdressID" :7}]}}
{"Volk":{"Bezeichnung": "Rmer", "Personen":[{"Vorname": "Julius",
    "Nachname" :"Caeser", "AdressID" :12}]}}
```


A.4. Seriendaten für das erste Beispiel der Evaluation


```
{"Album": {"name": "Geheimnisse der Druiden", "personen": ["Asterix",
    "Automatix", "Caligula Minux", "Caliguliminix", "Carolus Stachus",
    "Denkdirnix", "Gaius Barfus", "Gaius Bonus", "Gracchus Torschus",
    "Julius Csar", "Julius Pompilius", "Majestix", "Marcus Ecus",
    "Miraculix", "Obelix", "Troubadix", "Tullius Octopus"]}}
{"Album": {"name": "Die goldene Sichel", "personen": ["Asterix", "Bossix",
    "Claudius Metrobus", "Gracchus berdrus", "Majestix", "Miraculix",
    "Obelix", "Saingesix", "Stupidix", "Talentix", "Troubadix"]}}
{"Album": {"name": "Asterix und die Goten", "personen": ["Appelmus",
    "Asterix", "Asterus", "Barometrix", "Bartrik", "Cholerik", "Elektrik",
```

```
    "Fidibus", "Florix", "Historik", "Holperik", "Julius Bazillus",
    "Lyrik", "Majestix", "Marcus Konfus", "Mickerik", "Miraculix",
    "Obelix", "Obelus", "Pampelmus", "Periferik", "Praktifix", "Rheotorik",
    "Satirik", "Sprnix", "Strategus", "Symmetrik", "Theorik", "Troubadix"]}}
4 {"Album": {"name": "Asterix als Gladiator", "personen": ["Asterix",
    "Baba", "Briseradius", "Brutus", "Caligula Alavacomgetepus", "Der Rote
    Korsar", "Dreifuss", "Epidemais", "Gaius Obtus", "Gracchus Nenjetepus",
    "Julius Csar", "Keskonrix", "Lupus", "Majestix", "Miraculix", "Obelix",
    "Plaintcontrix", "Rictus", "Troubadix", "Zigepus"]}}
5 {"Album": {"name": "Tour de France", "personen": ["Asterix", "Automatix",
    "Baba", "Cosinus", "Csar Kneipix", "Der Rote Korsar", "Dreifuss",
    "Erix", "Excus", "Flavia", "Gaudeamus", "Gracchus Nenjetepus",
    "Heuchlerix", "Lucius Nichtsalsverdrus", "Majestix", "Meister Panix",
    "Minna", "Miraculix", "Mitgenus", "Motus", "Numalfix", "Obelix",
    "Odalix", "Omnibus", "Petilarus", "Possiamus", "Quintilius", "Schnfix",
    "Sinus", "Troubadix", "Unnutzus"]}}
6 {"Album": {"name": "Asterix und Kleopatra", "personen": ["Abstosis",
    "Asterix", "Automatix", "Chorus", "Der Rote Korsar", "Der Vorkoster von
    Kleopatra", "Dreifuss", "Erix", "Ginfiz", "Grobianus", "Julius Csar",
    "Kleopatra", "Majestix", "Miraculix", "Numerobis", "Obelix",
    "Schraubzieris", "Sekretaris", "Troubadix"]}}
7 {"Album": {"name": "Der Kampf der Huptlinge", "personen": ["Amnesix",
    "Asterix", "Augenblix", "Berlix", "Cathedralgotix", "Gibtermine",
    "Majestix", "Miraculix", "Obelix", "Troubadix"]}}
8 {"Album": {"name": "Asterix bei den Briten", "personen": ["Asterix",
    "Automatix", "Baba", "Der Rote Korsar", "Dreifuss", "Julius Csar",
    "Majestix", "Miraculix", "Obelix", "Troubadix"]}}
9 {"Album": {"name": "Asterix und die Normannen", "personen": ["Asterix",
    "Automatix", "Der Rote Korsar", "Dompfaf", "Majestix", "Miraculix",
    "Obelix", "Olaf Maulaf", "Stenograf", "Telegraf", "Troubadix"]}}
10 {"Album": {"name": "Asterix als Legionr", "personen": ["Asterix",
    "Automatix", "Baba", "Der Rote Korsar", "Dreifuss", "Hotelterminus",
    "Julius Csar", "Majestix", "Miraculix", "Obelix", "Tennisplatzis",
    "Troubadix"]}}
11 {"Album": {"name": "Der Arvernerschild", "personen": ["Asterix",
    "Automatix", "Diagnostix", "Julius Csar", "Majestix", "Miraculix",
    "Obelix", "Prognostix", "Troubadix"]}}
12 {"Album": {"name": "Asterix bei den Olympischen Spielen", "personen": [
    "Asterix", "Automatix", "Baba", "Claudius Musculus", "Der Rote Korsar",
    "Dreifuss", "Majestix", "Miraculix", "Obelix", "Troubadix"]}}
13 {"Album": {"name": "Asterix und der Kupferkessel", "personen": ["Asterix",
    "Automatix", "Baba", "Der Rote Korsar", "Dreifuss", "Majestix",
    "Miraculix", "Obelix", "Troubadix"]}}
```

```
14 {"Album": {"name": "Geheimnisse der Druiden zum Album Asterix in Spanien",
     "personen": ["Asterix", "Automatix", "Baba", "Begonia", "Costa y
     Bravo", "Der Rote Korsar", "Dreifuss", "Hohlenus", "Julius Csar",
     "Majestix", "Miraculix", "Obelix", "Troubadix"]}}
```

Literaturverzeichnis

[1] A. Baid, W. Wu, C. Sun, A. Doan, J. F. Naughton. „On Debugging Non-Answers in Keyword Search Systems“. In: *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.* 2015, S. 37–48. DOI: 10.5441/002/edbt.2015.05. URL: http://dx.doi.org/10.5441/002/edbt.2015.05.

[2] S. S. Bhowmick, A. Sun, B. Q. Truong. „Why not, WINE?: towards answering why-not questions in social image search“. In: *ACM Multimedia Conference, MM ’13, Barcelona, Spain, October 21-25, 2013.* 2013, S. 917–926. DOI: 10.1145/2502081.2502098. URL: http://doi.acm.org/10.1145/2502081.2502098.

[3] N. Bidoit, M. Herschel, A. Tzompanaki. „Efficient Computation of Polynomial Explanations of Why-Not Questions“. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015.* 2015, S. 713–722. DOI: 10.1145/2806416.2806426. URL: http://doi.acm.org/10.1145/2806416.2806426.

[4] N. Bidoit, M. Herschel, K. Tzompanaki. „Immutably answering Why-Not questions for equivalent conjunctive queries“. In: *Ingénierie des Systèmes d’Information* 20.5 (2015), S. 27–52. DOI: 10.3166/isi.20.5.27-52. URL: http://dx.doi.org/10.3166/isi.20.5.27-52.

[5] N. Bidoit, M. Herschel, K. Tzompanaki. „Query-Based Why-Not Provenance with NedExplain“. In: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.* 2014, S. 145–156. DOI: 10.5441/002/edbt.2014.14. URL: http://dx.doi.org/10.5441/002/edbt.2014.14.

[6] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 7159. RFC Editor, 24.2014. URL: https://tools.ietf.org/html/rfc7159%5C#section-5.

[7] D. Calvanese, M. Ortiz, M. Simkus, G. Stefanoni. „Reasoning about Explanations for Negative Query Answers in DL-Lite“. In: *CoRR* abs/1402.0575 (2014). URL: http://arxiv.org/abs/1402.0575.

[8] B. ten Cate, C. Civili, E. Sherkhonov, W. Tan. „High-Level Why-Not Explanations using Ontologies“. In: *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 2015, S. 31–43. DOI: 10.1145/2745754.2745765. URL: http://doi.acm.org/10.1145/2745754.2745765.

[9] A. Chapman, H. V. Jagadish. „Why not?“ In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 2009, S. 523–534. DOI: 10.1145/1559845.1559901. URL: http://doi.acm.org/10.1145/1559845.1559901.

[10] L. Chen, X. Lin, H. Hu, C. S. Jensen, J. Xu. „Answering why-not questions on spatial keyword top-k queries“. In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 2015, S. 279–290. DOI: 10.1109/ICDE.2015.7113291. URL: http://dx.doi.org/10.1109/ICDE.2015.7113291.

[11] J. Cheney, L. Chiticariu, W. C. Tan. „Provenance in Databases: Why, How, and Where“. In: *Foundations and Trends in Databases* 1.4 (2009), S. 379–474. DOI: 10.1561/1900000006. URL: http://dx.doi.org/10.1561/1900000006.

[12] Y. Cui, J. Widom. „Lineage tracing for general data warehouse transformations“. In: *VLDB J.* 12.1 (2003), S. 41–58. DOI: 10.1007/s00778-002-0083-8. URL: http://dx.doi.org/10.1007/s00778-002-0083-8.

[13] Y. Cui, J. Widom. „Lineage Tracing in a Data Warehousing System“. In: *ICDE*. 2000, S. 683–684. DOI: 10.1109/ICDE.2000.839493. URL: http://dx.doi.org/10.1109/ICDE.2000.839493.

[14] Y. Gao, Q. Liu, G. Chen, B. Zheng, L. Zhou. „Answering Why-not Questions on Reverse Top-k Queries“. In: *PVLDB* 8.7 (2015), S. 738–749. URL: http://www.vldb.org/pvldb/vol8/p738-gao.pdf.

[15] T. J. Green, G. Karvounarakis, V. Tannen. „Provenance semirings“. In: *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. 2007, S. 31–40. DOI: 10.1145/1265530.1265535. URL: http://doi.acm.org/10.1145/1265530.1265535.

[16] M. Guller. *Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large-Scale Data Processing, Machine Learning, and Graph Analytics, and High-Velocity Data Stream Processing*. Berkeley, CA: Apress, 2015, Online–Ressource (XXIII, 277 p. 64 illus, online resource). ISBN: 978-1-4842-0965-3 (Druckausgabe). URL: http://dx.doi.org/10.1007/978-1-4842-0964-6.

[17] Z. He, E. Lo. „Answering Why-not Questions on Top-k Queries“. In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. 2012, S. 750–761. DOI: 10.1109/ICDE.2012.8. URL: http://dx.doi.org/10.1109/ICDE.2012.8.

[18] M. J. Hernandez. *Database design for mere mortals: a hands-on guide to relational database design*. Pearson Education, 2003.

[19] M. Herschel. „A Hybrid Approach to Answering Why-Not Questions on Relational Query Results“. In: *J. Data and Information Quality* 5.3 (2015), 10:1–10:29. DOI: 10.1145/2665070. URL: http://doi.acm.org/10.1145/2665070.

[20] M. Herschel, M. A. Hernández. „Explaining Missing Answers to SPJUA Queries“. In: *PVLDB* 3.1 (2010), S. 185–196. URL: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R16.pdf.

[21] J. Huang, T. Chen, A. Doan, J. F. Naughton. „On the provenance of non-answers to queries over extracted data“. In: *PVLDB* 1.1 (2008), S. 736–747. URL: http://www.vldb.org/pvldb/1/1453936.pdf.

[22] M. Interlandi, K. Shah, S. D. Tetali, M. Gulzar, S. Yoo, M. Kim, T. D. Millstein, T. Condie. „Titian: Data Provenance Support in Spark“. In: *PVLDB* 9.3 (2015), S. 216–227. URL: http://www.vldb.org/pvldb/vol9/p216-interlandi.pdf.

[23] M. S. Islam, C. Liu, R. Zhou. „FlexIQ: A flexible interactive querying framework by exploiting the skyline operator“. In: *Journal of Systems and Software* 97 (2014), S. 97–117.

[24] M. S. Islam, R. Zhou, C. Liu. „On answering why-not questions in reverse skyline queries“. In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* 2013, S. 973–984. DOI: 10.1109/ICDE.2013.6544890. URL: http://dx.doi.org/10.1109/ICDE.2013.6544890.

[25] H. Karau, A. Konwinski, P. Wendell, M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis. [lightning-fast data analysis].* eng. 1. ed. Hier auch später erschienene unveränderte Nachdrucke. Beijing ; Köln[u.a.]: O'Reilly, 2015, XVI, 254 S. ISBN: 978-1-449-35862-4.

[26] L. V. S. Lakshmanan, W. H. Wang, Z. (Zhao. „Answering Tree Pattern Queries Using Views“. In: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006.* 2006, S. 571–582. URL: http://dl.acm.org/citation.cfm?id=1164177.

[27] A. Meliou, W. Gatterbauer, K. F. Moore, D. Suciu. „The Complexity of Causality and Responsibility for Query Answers and non-Answers“. In: *PVLDB* 4.1 (2010), S. 34–45. URL: http://www.vldb.org/pvldb/vol4/p34-meliou.pdf.

[28] L. Moreau, P. Missier. *PROV-DM: The PROV Data Model.* W3C Recommendation. https://www.w3.org/TR/2013/REC-prov-dm-20130430/. W3C, Apr. 2013.

[29] R. Ramakrishnan, J. Gehrke. *Database management systems.* Osborne/McGraw-Hill, 2000.

[30] *Spark SQL, DataFrames and Datasets Guide.* http://spark.apache.org/docs/1.6.1/sql-programming-guide.html. [Online].

[31] Q. T. Tran, C. Y. Chan, S. Parthasarathy. „Query reverse engineering“. In: *VLDB J.* 23.5 (2014), S. 721–746. DOI: 10.1007/s00778-013-0349-3. URL: http://dx.doi.org/10.1007/s00778-013-0349-3.

[32] Q. T. Tran, C. Chan. „How to ConQueR why-not questions“. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010.* 2010, S. 15–26. DOI: 10.1145/1807167.1807172. URL: http://doi.acm.org/10.1145/1807167.1807172.

[33] *XPath Syntax*. http://www.w3schools.com/xsl/xpath_syntax.asp. [Online].

Alle URLs wurden zuletzt am 21. 09. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift