Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

NedExplain meets Nautilus

Matthias Wagner

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Melanie Herschel
Betreuer/in:	Ralf Diestelkämper, M.Sc.
Beginn am:	30. September 2016
Beendet am:	31. März 2017
CR-Nummer:	H.4

Kurzfassung

Provenance umfasst alle Daten, die während der Berechnung beispielsweise einer SQL-Anfrage gesammelt werden können. Das beinhaltet neben dem eigentlichen Ergebnis auch Zwischenergebnisse und Metadaten, wie Ausführungszeiten oder Umgebungseigenschaften. Zwei bekannte Arten von Provenance sind Why und Why-Not Provenance. Why Provenance erklärt, wieso die Tupel in der Ergebnismenge enthalten sind. Why-Not Provenance beantwortet die Frage, wieso bestimmte Tupel in der Ergebnismenge erscheinen.

Ein Algorithmus zur Berechnung der Why-Not Provenance ist NedExplain. Er generiert Anfrage-basierte Erklärungen für fehlende Tupel in der Ausgabe einer SQL-Anfrage. Dafür sucht der Algorithmus auf die fehlenden Tupel möglicherweise passende Tupel in den Eingaberelationen und generiert die Erklärungen anhand der Stellen in der Anfrage, die diese Tupel entfernen. An Beispielen erkläre ich den detaillierten Ablauf und damit auch die Fähigkeiten des Algorithmus.

Ich implementiere den NedExplain Algorithmus und integriere diesen in den Nautilus Analyzer. Der Nautilus Analyzer bietet eine Reihe von Why-Not Provenance-Algorithmen zur Berechnung von Anfrage-basierten, Modifikation-basierten und Instanz-basierten Erklärungen an. Mit NedExplain erhält der Nautilus Analyzer eine weitere Möglichkeit zur Generierung von Anfrage-basierten Erklärungen. Der Algorithmus erweitert die bisherigen Fähigkeiten des Nautilus Analyzer, da er Aggregationen unterstützt.

Für die Implementierung von NedExplain stelle ich Hilfsdatenstrukturen vor, aus denen sich die Datenstrukturen von NedExplain zusammensetzen. Die Berechnung der Erklärungen in der Implementierung orientiert sich an dem Entwurf von NedExplain von Bidoit et al.. Neben kleinen Veränderungen am Algorithmus enthält die Implementierung eine Nachfolgerberechnung. Ich verzichte hier auf die Verwendung von Bibliotheken Dritter.

In der Evaluation zeige ich die Leistungsfähigkeit der Implementierung. Anhand von Use cases vergleiche ich die Laufzeit von NedExplain mit den anderen Why-Not Provenance Algorithmen mit Anfrage-basierten Erklärungen. Für einen Use cases untersuche ich den Einfluss der Größe des Datensatzes auf die Laufzeit durch Vervielfältigen der Eingabetupel. Die Implementierung ist dabei in allen Use cases schneller als der Why-Not Algorithmus und findet in mehr Use cases die Erklärungen. Die Nachfolgerberechnung ohne Bibliotheken dritter ist sich schneller als die mit externen Bibliotheken von Bidoit et al..

Inhaltsverzeichnis

1 Einleitung **13**
1.1 Nautilus 14
1.2 NedExplain 16
1.3 Übersicht über die Arbeit 17

2 Related Work **19**
2.1 Interaktives Debugging von SQL 19
2.2 Why-Not Provenance-Algorithmen 22

3 Erklärung von NedExplain an Beispielen **31**
3.1 Beispiel ohne Aggregation 33
3.2 Beispiel mit Aggregation 39

4 Implementierung **43**
4.1 Nautilus Architektur 43
4.2 Algorithmusimplementierung 45
4.3 Algorithmusänderungen 50
4.4 Einschränkungen 54
4.5 Zusammenfassung Implementierung 56

5 Evaluation **57**
5.1 Use cases 57
5.2 Use case Performance 60
5.3 Skalierbarkeit 61

6 Zusammenfassung und Ausblick **65**

Literaturverzeichnis **67**

Abbildungsverzeichnis

1.1 Nautilusworkflow 14

2.1 AFT-Zyklus 19
2.2 Einteilung von Why-Not Provenance Antworten 23

3.1 Beispieldatensatz 31
3.2 Beispielszenarien mit Anfrage, von NedExplain generiertem Anfragebaum, Ergebnis und Why-Not Frage ausformuliert und als Prädikat $\mathcal{P}$. Die Beschriftung der Knoten des Anfragebaums dient zur Identifizierung. 32

4.1 Nautilus Architektur (vereinfacht nach [10]). In Grau der im Nautilus Analyzer implementierte Teil mit den Stellen, an denen NedExplain eingefügt wird. . . 43
4.2 Angepasster Startdialog mit Option für NedExplain 45
4.3 UML-Diagramm der Klassen für den Anfragebaum 47

5.1 Performance von NedExplain und Why-Not für die Use cases aus Tabelle 5.2 . 60
5.2 Performance von NedExplain und Why-Not aus Bidoit et al. [3] 61
5.3 Normierte Laufzeiten für skalierte Crime-Relation. Die Referenz entspricht einem linearen Wachstum mit der Größe des Datensatzes. Der Skalierungsfaktor entspricht einer Vervielfachung der Tupel mit diesem Faktor. 62

Verzeichnis der Listings

3.1 Unterabfragen zum Beispiel Abbildung 3.2a 35

4.1 Interface der NedExplain Klasse 49
4.2 Abgeänderte Implementierung für den letzten Eintrag in tab_Q 54

Verzeichnis der Algorithmen

3.1 Ausschnitt aus NedExplain [3] 36

4.1 Prüfung für das Hinzufügen zum PickyMan in FindSuccessors() [3]. Der problematische Teil der Bedingung ist unterstrichen. 53
4.2 Angepasste Bedingung für Algorithmus 4.1 53

1 Einleitung

Bei der Arbeit mit Datentransformationen kann aufgrund der Komplexität leicht ein Fehler bei der Erstellung der Transformation auftreten. Deshalb besteht Nachfrage nach Unterstützung in der Fehlersuche. Für Syntax Verifikation gibt es bereits Programme, für die Untersuchung und das Debuggen der Ausgaben sind wenig oder gar keine Softwareunterstützung vorhanden. Der Entwickler geht typischerweise nach dem Schema *analyze-fix-test (AFT)* die Anfrage durch, bis das gewünschte Ergebnis erreicht ist. Dabei untersucht er das Ergebnis und passt mit den Erkenntnissen daraus die Anfrage an.

Nautilus [10] ist ein Werkzeug, dass Softwareunterstützung für den AFT-Zyklus bietet. Der Entwickler kann damit den vollständigen AFT-Zyklus teilautomatisiert durchführen, was die benötigte Zeit zur Fehlerbehebung reduziert. Während die Phasen fix und test für jeden Durchlauf identisch sind, können in der Analysephase unterschiedliche Algorithmen verwendet werden, um das Ergebnis zu erklären. Diese verwenden sogenannte Provenance Informationen für die Untersuchung der Anfrage und Ergebnistupel.

Provenance umfasst alle Daten, die während einer Berechnung gesammelt werden können. Das beinhaltet neben dem eigentlichen Ergebnis auch Zwischenergebnisse und Metadaten, wie Ausführungszeiten oder Umgebungseigenschaften. Dabei unterscheidet man in zwei Bereiche: Why und Why-Not Provenance. Beide verwenden die Provenance Informationen um Fragen zum Ergebnis zu beantworten: Why Provenance erklärt zu den einzelnen Tupel der Ergebnismenge, warum sie in der Ausgabe enthalten sind. Why-Not Provenance bringt Antworten, weshalb erwartete Tupel nicht erscheinen.

Nautilus verwendet sowohl Why als auch Why-Not Provenance-Algorithmen in der Analysephase zur Erklärung des Ergebnisses. Die Why Provenance-Algorithmen können zu jedem Tupel die Herkunft berechnen. Die Why-Not Provenance-Algorithmen generieren Erklärungen für fehlende Tupel. Sie unterscheiden sich in der Art ihrer Antworten: Instanz-basierte Erklärungen enthalten Änderungsvorschläge für die Dateninstanz, die die fehlenden Tupel dem Ergebnis hinzufügen. Anfrage-basierte Erklärungen beziehen sich auf die Anfrage und kennzeichnen die Stellen, an denen die fehlenden Tupel entfernt werden. Modifikation-basierte Erklärungen generieren bereits Änderungsvorschläge für die Anfrage. Der Nautilus Prototyp für die Analysephase enthält die Why-Not Provenance-Algorithmen Artemis [11], ConQueR [17] und Why-Not [5]. Diese decken alle Möglichkeiten der Erklärungen ab: Artemis generiert Instanz-basierte, ConQueR Modifikation-basierte und Why-Not Anfrage-basierte Erklärungen.

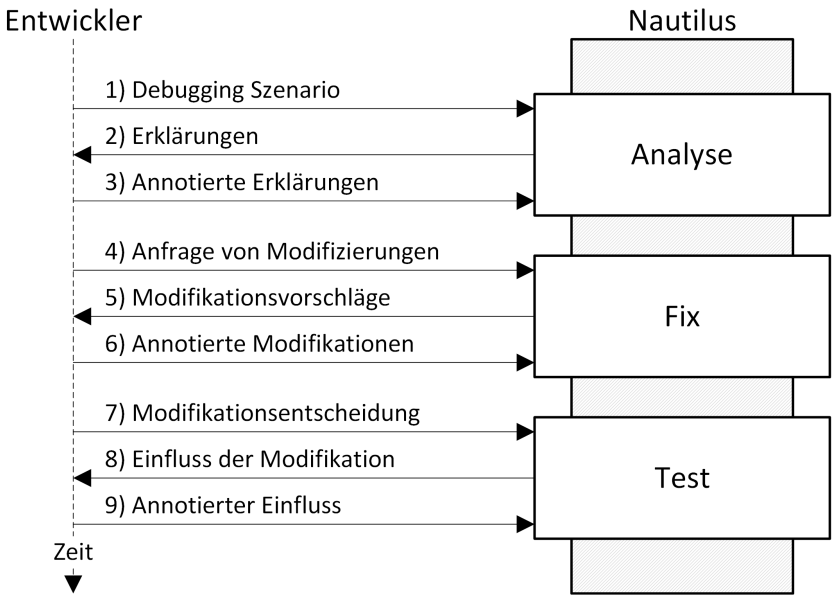


Abbildung 1.1: Der Nautilusworkflow erweitert den AFT-Zyklus auf 9 Schritte [10].

Ein weiterer Why-Not Provenance-Algorithmus ist NedExplain [3]. Die Art der Antworten ist identisch zu dem Why-Not Algorithmus, aber NedExplain unterstützt in der SQL Anfrage zusätzlich zu den von Why-Not unterstützten Operatoren noch Aggregationen. Dazu ist er in der Ausführung in identischen Szenarios mindestens gleich schnell. Bei bestimmten Anfragen, an denen Why-Not scheitert, findet NedExplain die Erklärung.

Die Implementierung von NedExplain in den Nautilus Analyzer erweitert die Fähigkeiten in der Analysephase um einen vielseitigeren Anfrage-basierten Why-Not Provenance-Algorithmus. Der Nautilus Analyzer kann damit in mehr Szenarien vollständig Erklärungen für das Ergebnis berechnen.

Im Folgenden stelle ich kurz Nautilus und NedExplain vor und gebe dann eine Übersicht über die Arbeit.

1.1 Nautilus

Nautilus [10] bildet den AFT-Zyklus in einen Workflow ab. Das System bietet dabei neben einer Unterstützung der einzelnen Phasen auch sogenanntes *Transaction-Lifecycle-Management (TLM)*. Das bedeutet, dass mehrere Zyklen mit einer Anfrage als zusammenhängende Arbeit erkannt werden.

Die unterstützte Sprache für Transformationen ist SQL. Unterabfragen werden nicht unterstützt. In den Anfragen können Mengenoperationen, Selektionen, Projektionen, Joins, Aggregationen und Grouping enthalten sein.

Der *Workflow* erweitert den normalen Analyse-Test-Fix-Zyklus (Abbildung 1.1). Dieser kann, wie ohne Softwareunterstützung, wiederholt durchgeführt werden.

Nautilus verwendet in der Analysephase ein Debuggingszenario, das aus den folgenden Komponenten besteht: Die zu untersuchenden SQL Anfragen, den Quelldaten in einer Datenbank, einer Beschreibung des gewollten Ergebnisses und weiteren Regeln, die das Ergebnis erfüllen soll. Diese weiteren Regeln können z.B. einzelne Anfragen als unveränderbar kennzeichnen.

Für dieses Szenario gibt Nautilus eine Menge an Erklärungen zurück, die beschreiben, weshalb das Ergebnis auf die Anfrage zurückgegeben wurde und was die Unterschiede zwischen erwartetem und erhaltenem Ergebnis verursacht hat. Dafür verwendet Nautilus existierende Provenance-Algorithmen, sowohl für Why als auch für Why-Not Provenance. Die erstellten Erklärungen können dabei Instanz-basiert, Anfrage-basiert oder Modifikation-basiert sein. Es gibt keine Einschränkungen mehrere Algorithmen zu einer Anfrage auszuführen um unterschiedliche Arten von Erklärungen zu erhalten. Dabei wird versucht, das Format zu vereinheitlichen, soweit es die Algorithmen erlauben.

Der Entwickler kann nun auf Basis der Ergebnisse des vorherigen Schritts entscheiden, dass seine Anfrage korrekt ist. Andernfalls kann er nun aus der Menge der Erklärungen annotieren, um zu kennzeichnen, welche bevorzugt oder verworfen werden sollen. Da meistens verschiedene Möglichkeiten vorgeschlagen werden, um eine Problemstelle zu korrigieren, ist dieser Schritt wichtig, um die gewünschte Anfrage zu erhalten. Dazu sammelt Nautilus die gewählten Annotationen über den aktuellen AFT-Zyklus hinaus und versucht die zukünftigen Vorschläge darauf basierend zu verbessern.

Mit den Annotationen der Erklärungen und der Anfrage berechnet Nautilus verschiedene Änderungsvorschläge. Aus diesen priorisiert der Benutzer, wie auch bei den Erklärungen, die passendsten und markiert ungewollte.

In der Testphase wählt der Entwickler aus den Vorschlägen die gewünschten aus, die dann auf die Anfrage angewendet werden. Zur Überprüfung wird die neue Anfrage in das Debuggingszenario eingefügt und die Veränderungen an dem Ergebnis werden zurückgegeben. Dabei erhält der Entwickler nicht nur das neue Ergebnis, sondern einzelne Aussagen, die die Änderungen beschreiben. Davon wählt er nun aus, welche erwünscht und welche unerwünscht sind. Die ungewollten Änderungen werden für den nächsten Zyklus in das Debuggingszenario übernommen.

Im Moment existieren zwei *Prototypen*, die jeweils unterschiedlichen Funktionsumfang mitbringen:

Die Erste ist ein Eclipse-Plug-in Nautilus Analyzer für Eclipse Indigo und verwendet das Eclipse-Framework für die graphische Oberfläche [9]. Es bildet den Analyseteil von Nautilus ab. Für die Erklärungsberechnung stehen die Why-Not Provenance-Algorithmen Why-Not [5], Artemis [11], und ConQuer [17] zur Verfügung, zu denen NedExplain [3] im Rahmen dieser Arbeit hinzugefügt wird. In 4 wird die Implementierung von NedExplain in diesen Prototyp näher erläutert. Dazu ist ein Why Provenance-Algorithmus zur Berechnung der Herkunft

der Ausgabetupel enthalten. Als Datenbanksystem wird PostgreSQL verwendet. Neben der Datenbank greift das Plug-in auf das System Trio zu, das Herkunftsdaten zu den Tupeln zur Verfügung stellt und von den Provenance-Algorithmen benötigt wird. Java Runtime Environment und Python sind dazu installiert als Vorbedingungen für die einzelnen Komponenten.

Die zweite Implementierung basiert auf Webtechnologien. Sie bildet auch die Analysephase ab. Das Backend verwendet Django, das Frontend Bootstrap als Framework. Die Algorithmen sind in Python implementiert und greifen auf eine SQL kompatible Datenbank zurück. Zur Verfügung stehen nur die Why-Not Provenance-Algorithmen Ted++ und ConQueR.

1.2 NedExplain

NedExplain ist ein Why-Not Provenance-Algorithmus. In Nautilus wird er hinzugefügt als weitere Möglichkeit, dem Benutzer im Analyse Schritt Erklärungen zu generieren. Diese sind Anfrage-basiert, d.h. NedExplain untersucht die Anfrage und findet die Stellen, an denen die, im Ergebnis fehlenden Tupel, entfernt werden. Dafür generiert der Algorithmus aus der Anfrage einen Baum aus Unterabfragen und verfolgt Tupel, die auf die fehlenden Tupel passen könnten. Wenn diese in einer Anfrage entfernt werden, wird diese markiert. Dazu werden die jeweils entfernten Tupel gespeichert und können in der vollständigen Antwort eingesehen werden. NedExplain erlaubt dabei unterschiedlich detaillierte Ausgaben zu generieren, nimmt aber keine Priorisierung in der Ausgabe vor. Erlaubte SQL-Anfragen können Vereinigungen, Selektionen, Projektionen, Aggregationen und Joins enthalten.

Als Eingabe erwartet NedExplain eine SQL-Anfrage mit dazugehöriger Datenbankinstanz und eine sogenannte Why-Not-Frage. Diese besteht aus Bedingungstupeln, die jeweils für ein oder mehrere Ausgabespalten des Ergebnisses Regeln enthalten, die eigentlich von der Ausgabe erfüllt sein sollten. Diese Regeln können konstante Werte verlangen oder Abhängigkeiten festlegen. Der Algorithmus beantwortet jedes Bedingungstupel unabhängig von eventuell vorhandenen anderen.

Dafür wird zu Beginn die SQL-Anfrage in einen Anfragebaum umgewandelt, der dann von den Tabelleninstanzen her nach oben zur Aggregation bzw. Projektion verarbeitet wird. Da die Transformation einer SQL-Anfrage in einen Baum nicht eindeutig ist, wurden die folgenden Regeln hinzugefügt: Deshalb werden Selektionen möglichst weit nach unten geschoben, damit diese früh gefunden werden, falls sie Teil der Erklärung sind. Wenn die Anfrage eine Aggregation enthält, wird eine sogenannte *visibility-frontier* berechnet. Dabei handelt es sich um die kleinste Unteranfrage, bei der bereits alle für die Aggregation benötigten Tabellen vorhanden sind. Selektionen darauf werden nicht über diese Grenze nach unten geschoben. Dazu werden kompatible Tupel in der Datenbankinstanz gesucht. Diese Tupel enthalten Werte, die für die Bedingungstupel verlangt werden.

Der berechnete Baum wird in die Tabelle tab_Q übertragen, die dann die Abarbeitungsreihenfolge definiert und im Laufe der Ausführung vervollständigt wird. Dabei werden die Knoten

mit aufsteigender Tiefe eingefügt. Zu jedem Eintrag werden Eingabetupel, Ausgabetupel, kompatible Tupel, das Level des Knotens im Baum und eine Referenz auf den Elternknoten gespeichert. Zu Beginn sind die Ein- und Ausgabe nur für Tabelleninstanzen gesetzt. Für diese wird jeweils auch berechnet, ob sie Tupel enthalten, die die Why-Not-Frage erfüllen. Diese werden dann als sogenannte kompatible Tupel gespeichert. Neben tab_Q werden die globalen Strukturen EmptyOutputMan, Non-PickyMan und PickyMan verwaltet, in denen Unteranfragen gesammelt werden, um diese später in den Erklärungen zu verwenden.

Der Berechnungsschritt des Algorithmus geht nun tab_Q von links nach rechts durch. Dabei wird für jede Unterabfrage die Ausgabe berechnet und zu der Eingabe des Elternknoten hinzugefügt. Falls die Eingaben kompatible Tupel haben, wird überprüft, ob diese valide Nachfolger haben. Weggefallene Tupel werden mit der Unteranfrage gespeichert.

Zur Optimierung der Laufzeit wird vor Beginn der Untersuchung der aktuellen Unterabfrage überprüft, ob es überhaupt noch die Möglichkeit gibt weitere Erklärungen zu finden. Dafür müssen im vorherigen Level, falls dieses gerade geändert wurde, noch kompatible Tupel vorhanden sein oder im noch nicht bearbeiteten Teil der Tabelle ist noch eine Tabelleninstanz, die kompatible Tupel enthält. Wenn das nicht erfüllt ist, wird die Berechnung abgebrochen und die Antwort auf die Why-Not-Frage generiert.

Diese Antwort besteht aus zwei Teilen: Der erste Teil listet zu jedem kompatiblen Tupel, das die Why-Not-Frage erfüllt und es nicht ins Ergebnis geschafft hat, die Unterabfrage auf, bei der es entfernt wurde. Das kann auch indirekt geschehen, wenn eine Selektion nicht alle kompatiblen Tupel entfernt, aber die Aggregation der Anfrage auf der Ausgabe der Selektion nicht mehr erfüllt ist. Da diese Antwort je nach Datensatz unübersichtlich werden kann, gibt es die Möglichkeit, sie auf die Unterabfragen zu reduzieren, die kompatible Tupel entfernen. Der zweite Teil Antwort enthält alle Unterabfragen, bei denen die Ausgabe keine Nachfolger indirekt kompatibler Tupel enthält. Indirekt kompatible Tupel sind alle Tupel aus Tabellen Instanzen, die keine kompatiblen Tupel für die Ausgabe haben, aber auch in der Anfrage verarbeitet werden. Definiert sind beide Antworten nur als Mengen. Eine alternative Darstellung ist ein annotierter Anfragebaum, wie er bereits vom Why-Not Algorithmus [5] im Nautilus Analyzer verwendet wird.

1.3 Übersicht über die Arbeit

Zunächst gehe ich auf Related Work (Kapitel 2) ein. Dabei wird zwischen Arbeiten über das Debugging von SQL-Anfragen (Abschnitt 2.1) und Why-Not Provenance-Algorithmen unterschieden (Abschnitt 2.2). Im Anschluss erkläre ich den NedExplain Algorithmus anhand von zwei Beispielen in Kapitel 3. Kapitel 4 enthält die Dokumentation der Implementierung des Algorithmus in Java und die Einbindung in das Nautilus Eclipse Plug-in. Die Evaluation in Kapitel 5 vergleicht die Implementierung mit anderen Why-Not Provenance-Algorithmen in

ausgewählten Use cases und untersucht die Performance mit unterschiedlich großen Datensätzen. Den Abschluss bildet eine Zusammenfassung der Ergebnisse der Implementierung von NedExplain in Nautilus und ein Ausblick.

2 Related Work

Hier werden inhaltlich ähnliche Forschungsergebnisse vorgestellt. Im Abschnitt 2.1 stelle ich Arbeiten zu SQL Debugging vor. Im Abschnitt 2.2 gehe ich näher auf verschiedene Why-Not Provenance-Algorithmen ein, die zum Teil bereits im Nautilus Analyzer eingesetzt werden.

2.1 Interaktives Debugging von SQL

Nautilus deckt mit seinen Fähigkeiten zum Arbeit mit Datentransformationen den ganzen *analyze-fix-test (AFT)* Zyklus (Abbildung 2.1) ab. Es gibt im Moment kein System, das Unterstützung für alle Phasen dieses Zyklus enthält. Einzelne Phasen des Zyklus sind in spezialisierten Werkzeugen bereits umgesetzt, von denen ich im Folgenden einige vorstelle und mit Nautilus vergleiche.

Für den Analyseteil gibt es Potters Wheel [16], mit dem, neben der Analyse der Ausgabe, auch die Erkennung von Änderungen beim Test möglich ist (Abschnitt 2.1.1). Die nähere Betrachtung der Analyse mit Why-Not Provenance-Algorithmen ist in Abschnitt 2.2 ausgeführt. Sie stehen im direkten Vergleich zu NedExplain und sind deshalb in einem eigenen Kapitel.

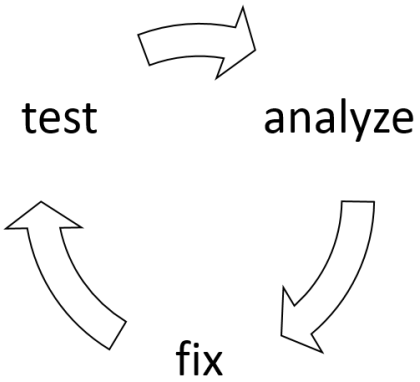


Abbildung 2.1: AFT-Zyklus: Analysephase, um Fehler in der Anfrage auf Basis der Ausgabe zu finden. Fixphase, in der Änderungen angewandt werden. Testphase, bei der ein neues Ergebnis berechnet wird und die Änderungen gekennzeichnet werden.

Im Zusammenhang mit der Fixphase gibt es PRISM [6], das Veränderungen an Datenbankschemata unterstützt (Abschnitt 2.1.2).

Funktionen, die den Testschritt unterstützen, ähneln dem Problem in Datenbanken, die Daten und Views darauf konsistent zu halten (Abschnitt 2.1.3).

2.1.1 Potters Wheel

Potters Wheel [16] ist eine Anwendung zur Entwicklung von Datentransformationen. Der Benutzer arbeitet auf einer Tabelle aus Tupeln und kann diese mit einem graphischen Interface bearbeiten. Die erstellte Transformation kann als ausführbare Datei exportiert werden, die dann auf den Datensatz angewandt werden kann.

Als Input sind unterschiedliche Datenquellen wie Datenbanken oder Dateien möglich, da Potters Wheel selbst definierte Domänen erlaubt. Aus der Eingabe werden automatisiert Muster für die jeweiligen Spalten abgeleitet, die jeweils die passendsten Domänen enthalten. Dazu erkennt das Programm, wenn ein Feld immer konstante Werte oder Längen enthält.

Auf den strukturierten Daten können nun Transformationen entwickelt werden. Definitionen mit Beispielen werden einer Definition über einen regulären Ausdruck vorgezogen. Jede Transformation wird auf den in der graphischen Oberfläche angezeigten Ausschnitt der Daten angewandt und die Änderungen bzw. Probleme werden sichtbar gemacht. Problematisch ist beispielsweise, wenn eine Transformation für einzelne Einträge die Regeln der Domäne verletzt. Wenn eine Aktion wieder rückgängig gemacht wird, werden in gleicher Weise die Veränderungen an dem Datenausschnitt gekennzeichnet. Bei der Kennzeichnung von problematischen Werten wird unterschieden, ob diese durch die Transformation oder durch den Datensatz entstanden sind.

Die Motivation zur Entwicklung von Potters Wheel kommt aus dem Bereich des *data cleaning*. Durch die teilautomatisierte Vorverarbeitung der Datenquelle in strukturierte Daten erlaubt Potters Wheel eine größere Bandbreite an Datenquellen als Nautilus, das sich auf eine relationale Datenbank beschränkt. Beide Systeme haben das gleiche Ziel, eine passende Transformation auf einem exisitierenden Datensatz zu entwickeln. Der Ansatz dafür ist stark unterschiedlich: Potters Wheel setzt auf eine Entwicklung der Transformation über die Veränderung von Beispieltupeln durch den Benutzer. Alle Schritte werden über die graphische Oberfläche definiert und daraus wird dann die Transformation generiert. Nur in Ausnahmefällen bearbeitet der Benutzer an der Transformation selbst. In Nautilus dagegen wählt der Benutzer die Änderungen an der Anfrage selbst aus, während das System Erklärungen und Effekte auf die Tupel zur Verfügung stellt. Diese Möglichkeit der Auswahl der Änderungen fehlt in Potters Wheel ganz, weshalb es nur die Phasen Analyse und Fix aus dem AFT-Zyklus abbildet.

2.1.2 PRISM

PRISM [6] ist ein Programm, das die Weiterentwicklung von Datenbankschemata unterstützt. Ziel ist es, die Ausfallzeit der Anwendung möglichst kurz zu halten, indem in PRISM sowohl Schema als auch Anfragen angepasst und die Effekte untersucht werden können. Die erarbeiteten Änderungen können dann mit den Anfragen exportiert werden.

Der Benutzer spezifiziert seine Änderungen in *Schema Modification Operators (SMO)* auf das alte Datenbankschema und dazugehörige Anfragen. PRISM zeigt die Änderungen am Schema für jeden Schritt und berechnet das entstehende Ausgabeschema. Dabei wird überprüft, ob diese Änderung die Daten erhält oder dabei Redundanzen entstehen. Bei Problemen informiert das System den Benutzer und schlägt Lösungen vor.

Die spezifizierte Sequenz der Schemaänderungen wird im Anschluss invertiert und auf Redundanzen und Informationsverlust geprüft. Wenn beide Sequenzen die Informationen erhalten und keine Redundanzen in die Daten einfügen, dann ist die Änderung insgesamt reversibel.

Darauf aufbauend kann der Benutzer nun die Anfragen an das neue Schema anpassen lassen und das Ergebnis überprüfen. Die Anpassung kann entweder eine Umstellung auf ein View im alten Schema sein oder die Anfrage wird neu für das neue Schema generiert. Dazu ist es möglich, die Komponente zur Anpassung der Anfragen in die Anwendung zu integrieren. Diese prüft alle Anfragen, bevor diese an das Datenbankmanagmentsystem weitergegeben werden. Anfragen auf das alte Schema werden dann im Aufruf angepasst. Dies erlaubt auch Änderungen am Schema vorzunehmen, wenn nicht alle Anfragen vor dem Aufruf angepasst werden können.

Neben der eigentlichen Schemaweiterentwicklung und Anfragenanpassung werden die Änderungen in ein erweitertes Datenbankschema geschrieben, sodass diese auch im Nachhinein nachvollziehbar sind. Das gibt auch die Möglichkeit, die Änderungen zurückzunehmen, falls dies gewünscht ist.

PRISM bringt wie Nautilus einen Workflow mit, anhand dessen die Entwicklung durchgeführt wird. Dieser ist bei PRISM etwas expliziter unterteilt, da für jeden Schritt die Ergebnisse bereits zurückgegeben werden und ein Zurückgehen zum vorherigen Schritt möglich ist. In Nautilus dagegen ist der Workflow als Sequenz von Aktionen entworfen, die in dieser Reihenfolge ausgeführt werden sollen ohne Möglichkeit zum Zurückspringen. Da PRISM annimmt, das der Benutzer weiß, welche Änderungen er an dem Datenbankschema machen möchte, deckt es nur die Fix- und Testphasen des AFT-Zyklus für Schemaweiterentwicklung ab.

2.1.3 View Update Translation

In einer Datenbank müssen die Daten in den Views konsistent zu den Daten in den Ursprungsrelationen gehalten werden. Dies wird View Update Problem genannt. Es lässt sich auf zwei Unterprobleme aufteilen: Die Aufgabe, Änderungen an Views in die Ursprungsrelationen zu

übertragen, wird in [7] betrachtet. Das Gegenstück dazu ist die Aktualisierung der Daten in den Views, wenn Änderungen an den Ursprungsdaten vorgenommen werden.

In [4] werden zwei Algorithmen vorgestellt, die Änderungen aus den Datenbankrelationen in die Views übernehmen. Diese arbeiten auf den Daten, die durch eine Transaktion verändert werden. Dies schließt gelöschte und neu hinzugefügte Daten mit ein.

Der erste Algorithmus untersucht jedes Tupel, ob es die Bedingungen eines Views erfüllt. Falls dies nicht der Fall ist, wird es im zweiten Schritt übersprungen.

Der zweite Algorithmus berechnet für jedes Tupel, den Einfluss auf die Views. Dies kann langsamer sein als eine vollständige Neuberechnung der Views, abhängig davon wie viele Tupel sich geändert haben.

Die Testphase in Nautilus stellt dem Nutzer den Einfluss der Veränderungen an der Anfrage in den Datenbanken bereit. Die Berechnung ist vergleichbar zu diesen Algorithmen, da in beiden Fällen nur die Veränderungen zusätzlich berechnet werden sollen, wenn auch die Motivation unterschiedlich ist. Damit ist [4] zur Unterstützung der Testphase des AFT-Zyklus zu zuordnen.

2.2 Why-Not Provenance-Algorithmen

Why-Not Provenance-Algorithmen sind in Nautilus Teil der Analysephase. Das Problem, das diese erklären sollen, sind fehlende Tupel in der Ausgabe der Anfrage bzw. Anfragen. Als Bedingungstupel können ein oder mehrere fehlende Tupel dargestellt werden. Jedes Bedingungstupel enthält die gleichen Felder, wie die Ausgabetupel und ein weiteres für eine Bedingung. Die einzelnen normalen Felder können entweder einen konstanten Wert oder eine Variable enthalten. In dem Bedingungsfeld können die verwendeten Variablen mit Bedingungen verknüpft werden. Zur Berechnung der Ergebnisse benötigt der Algorithmus noch die Anfragen mit Ergebnis, bei dem die gesuchten Tupel fehlen und die Datenbankinstanz, auf die die Anfragen angewendet werden. Je nach Algorithmus kommen dazu noch Regeln, die die Ausführung bzw. das Ergebnis beeinflussen. Diese Eingabe wird Why-Not Frage oder synonym Debuggingszenario genannt. Diese allgemeine Darstellung der Eingabe für einen Why-Not Provenance-Algorithmus deckt alle spezifischen Eingaben, der hier verglichenen Why-Not Provenance-Algorithmen, ab ([8]).

Die Antwort, die die jeweiligen Algorithmen auf diese Why-Not Frage geben, besteht aus einzelnen Erklärungen. Die Art der Erklärungen erlaubt, die Algorithmen in Kategorien einzuteilen (Abbildung 2.2). *Anfrage-basierte* Erklärungen beziehen sich auf die Anfrage und zeigen die Stellen auf, die zur Entfernung der fehlenden Tupel beitragen. Eine Variante davon sind *Änderungs-basierte* Erklärungen, die Änderungen an der Anfrage vorschlagen, um die fehlenden Tupel dem Ergebnis hinzuzufügen. Beide Variationen beziehen sich auf die Anfrage und sind äquivalent ([8]), weshalb sie in einer Kategorie zusammengefasst sind. Instanz-basierte

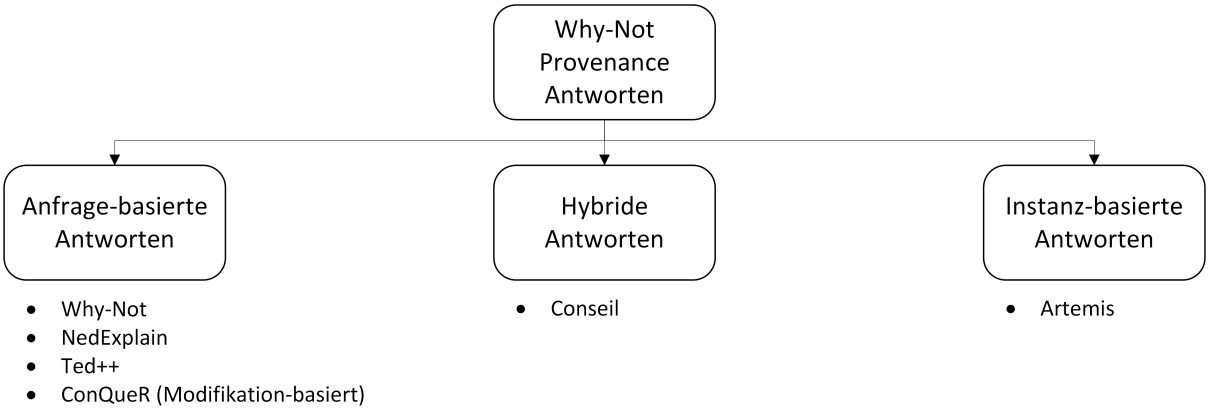


Abbildung 2.2: Die Antworten von Why-Not Provenance-Algorithmen sind in drei Kategorien unterteilt: 1) Anfrage-basiert, 2) Instanz-basiert und die Kombination 3) Hybrid. Modifikation-basiert ist eine Spezialisierung von Anfrage-basiert. Unter den Kategorien stehen jeweils dazugehörige Algorithmen.

Erklärungen enthalten Änderungen an den Ursprungsdaten, um bei den gleichen Anfragen die fehlenden Tupel im Ergebnis zu haben. *Hybride* Antworten können sowohl aus Instanz-basierten als auch aus Anfrage-basierten Erklärungen bestehen, die kombiniert die Frage beantworten.

Die Implementierung von NedExplain erweitert die Why-Not Algorithmen des Nautilus Analyzer. Neben den bereits implementierten Algorithmen Why-Not (Abschnitt 2.2.1), Artemis (Abschnitt 2.2.2) und ConQueR (Abschnitt 2.2.4) stelle ich in diesem Kapitel Weitere aus dem Bereich der Why-Not Provenance vor. Neben der Funktionalität und dem Ablauf des Algorithmus zeige ich die Unterschiede und Gemeinsamkeiten zu NedExplain auf. Als Abschluss erkläre ich die Besonderheiten von NedExplain in diesem Zusammenhang.

2.2.1 Why-Not

Why-Not [5] beantwortet Why-Not Fragen mit Anfrage-basierten Antworten. Die Anfrage wird als strukturiere Abfolge von Manipulationen betrachtet. Der Algorithmus ist für den Einsatz mit Workflowsystemen entworfen und verwendet deshalb andere Begriffe, als die anderen vorgestellten Why-Not Provenance-Algorithmen. Der Algorithmus arbeitet auf einer Baumdarstellung der SQL-Anfrage. Im Nautilus Analyzer werden die Erklärungen von Why-Not als Annotation der Baumdarstellung dargestellt.

Um die fehlenden Tupel zu erklären, sucht der Algorithmus deren mögliche Herkunft in Zwischenergebnissen und Ursprungsdaten. Der Begriff der Herkunft kommt dabei aus der Why Provenance und wird verwendet, um zu erklären, aus welchen vorhergegangenen Tupeln ein Tupel eines Ergebnisses berechnet wurde. Der Algorithmus nimmt an, dass die Herkunft eines fehlenden Tupel außerhalb der Herkunft aller Tupel des Ergebnisses liegen muss. Aus

diesen Tupeln werden zu Beginn die auf die Bedingungstupel passenden herausgefiltert, die *unpicked* Tupel genannt werden. Danach untersucht der Algorithmus die Zwischenergebnisse der Manipulationen und berechnet für jedes Tupel, ob dessen Herkunft in den unpicked Tupeln ist. Damit lassen sich die Manipulationen finden, die in der Ausgabe weniger oder keine passenden Tupel mehr haben. Die Erklärungen, die der Why-Not Algorithmus generiert, sind die Manipulationen, die in der Ausgabe keine passenden Tupel mehr enthalten und am nächsten im Ablauf zum Ergebnis sind. Der Algorithmus verwendet aus der allgemeinen Definition der Why-Not Frage die weiteren Regeln nicht und arbeitet auf einer Anfrage und einem Bedingungstupel. Theoretisch gibt es bei den Manipulationen vom Algorithmus her keine Einschränkungen, solange die Herkunft für die Ausgabe berechnet werden kann.

Für die Implementierung werden zwei Möglichkeiten vorgestellt:

Bottom up Diese Variante orientiert sich an dem theoretischen Entwurf und beginnt bei den Datenquellen. Von dort aus geht der Algorithmus in Breitensuche zur Ausgabe über die Struktur der Manipulationen. Bei SQL-Anfragen ist das eine Baumdarstellung, die in der Implementierung festgelegt wird. Für jede Manipulation wird die Ausgabe berechnet und für alle unpicked Tupel in der Eingabe geprüft, ob ein Nachfolger in der Ausgabe enthalten ist. Falls dies nicht der Fall ist, wird die Manipulation als filternd gespeichert. Die letzten in dem jeweiligen Pfad gefundenen Manipulationen werden als Ergebnis ausgegeben.

Top down In dieser Variante verarbeitet der Algorithmus die Manipulationen von der Ausgabe aus zu den Datenquellen hin. Ziel ist es hier, die ersten Manipulationen zu finden, die noch unpicked Tupel bzw. deren Nachfolger in der Eingabe haben, aber keine Nachfolger davon in der Ausgabe enthalten. Dabei werden jeweils alle Vorgängermanipulationen untersucht und wenn eine filternde gefunden wurde, wird der Pfad abgebrochen. Da nur die Letzten in dem jeweiligen Pfad im Ergebnis gewünscht sind, ist diese Variante möglich. Voraussetzung ist allerdings, dass alle Zwischenergebnisse der Manipulationen vorhanden sind. Dies ist bei klassischen Datenbanksystemen eher selten, für Workflowsysteme allerdings normal.

Im Vergleich kommen beide Varianten zu den gleichen Ergebnissen. Welche besser ist, hängt von dem Einsatzszenario ab: Der Top down Ansatz findet die Manipulationen schnell, die nahe an der Ausgabe sind, Bottom up die zuerst, die in der Nähe der Quellen sind. Bottom up kann mit wenig Speicher auskommen, indem Zwischenergebnisse jedes Mal für den aktuellen Pfad neu berechnet werden. Alternativ können diese auch gespeichert werden. Auch dann braucht er weniger Speicher als Top down, der alle Zwischenergebnisse zur Verfügung haben muss.

Der Algorithmus greift in beiden Varianten auf das Finden der Nachfolger zurück. Die Idee dahinter und die Schnittstelle wird angemessen definiert, aber auf eine Implementierung der Herkunftsverfolgung verzichtet. Bei der Evaluation mit SQL-Anfragen und relationalen Datenbanken wird dafür die Bibliothek Trio [15] verwendet. Diese unterstützt Joins, Selektionen und Projektionen in der Anfrage. Vereinigungen sind durch die Kombination der Ergebnisse möglich.

NedExplain ist als Verbesserung des Why-Not Algorithmus entworfen und entsprechend ähnlich sind die Algorithmen. Beide arbeiten auf einer Baumdarstellung der Anfrage und

geben die filternden Operatoren zurück. NedExplain gibt dabei alle Operatoren aus, während Why-Not sich auf die letzten in der Verarbeitungsreihenfolge beschränkt. In der Performance ist NedExplain schneller und findet in ausgewählten Szenarien Erklärungen, an denen Why-Not scheitert. Dazu unterstützt NedExplain Aggregationen und mehrere Bedingungstupel gleichzeitig.

2.2.2 Artemis

Artemis [11] ist ein Algorithmus, der Instanz-basierte Antworten auf eine Why-Not Frage errechnet. Er basiert auf dem Why-Not Algorithmus missing-answers [12]. Missing-answers arbeitet auf einer SQL-Anfrage, die aus Selektionen, Projektionen und Joins bestehen kann, und auf einem fehlenden Tupel. Die Antwort enthält nur ein einzelnes Tupel, das zum Datensatz hinzugefügt werden muss, um das fehlende Tupel in der Ausgabe zu erhalten.

In Artemis sind in der einzelnen Anfrage dazu noch Union und Aggregationen möglich und die Ausgabe besteht aus mehreren Vorschlägen, die jeder für sich die fehlenden Tupel der Ausgabe hinzufügen. Neben den erweiterten Möglichkeiten bringt Artemis Neuentwicklungen mit: Artemis beachtet Seiteneffekte und kann diese in der Antwortberechnung beachten. Dazu enthält es eine Regelprüfung, die unerlaubte Antworten herausfiltern kann. Das sind Antworten, die z.B. „null“-Felder einfügen.

Die Why-Not Frage kann mehrere Anfragen und mehrere Bedingungstupel enthalten, die jeweils einer Anfrage zugeordnet sein müssen. Zwischen den Bedingungstupeln können Abhängigkeiten bestehen. Artemis erlaubt zusätzliche Regeln zu definieren, um die Ausgabe zu beeinflussen. Beispielsweise können Relationen oder Views gekennzeichnet werden, die unveränderbar sind oder möglichst wenig Änderungen erhalten sollen.

In der Parameterdefinition des Algorithmus nehmen die Regeln mehr Platz ein als in der allgemeinen Definition der Why-Not Frage. In den Möglichkeiten sind jeweils identisch, wie in [8] gezeigt.

Die Antworten von Artemis bestehen aus einer Menge Erklärungen, wobei jede Erklärung aus Tupeln besteht. Diese sind entweder bereits in der Datenbank vorhanden oder müssen eingefügt werden, um die fehlenden Tupel der Ausgabe hinzuzufügen. Es treten Unterschiede auf, da die Regeln und Anfragen unterschiedliche Lösungen erlauben. Dargestellt werden dann nur die benötigten Tupel um alle fehlenden Tupel in der Ausgabe zu erhalten.

Die Ausführung besteht aus 5 Schritten:

Schritt 1 Aus den Anfragen und den fehlenden Tupeln werden Erklärungsmuster definiert. Jede finale Erklärung muss auf eines dieser Muster passen. Die Schritte 2-4 werden für jedes Muster einzeln wiederholt:

Schritt 2 Zu jedem Tupel der Datenbankinstanz wird ein Bedingungsfeld hinzugefügt, sodass diese die gleiche Struktur haben, wie die fehlenden Tupel. Dazu werden Tupel nach dem Muster eingefügt.

Schritt 3 Die Anfragen aus Q werden auf der modifizierte Datenbankinstanz ausgeführt. Dabei wird die Ausführung auch auf die Bedingung angewandt [13]. Die erhaltenen Ergebnisinstanzen enthalten zu dem eigentlichen Ergebnis ein Bedingungsfeld zu jedem Tupel.

Schritt 4 Aus den Tupeln, die die fehlenden Tupel erfüllen, werden die Erklärungen berechnet. Die Regeln aus der Why-Not Frage fließen hier in die Bedingungen ein. Ein Lösealgorithmus prüft, ob die Bedingungen jeweils erfüllbar sind. Damit gefundene Tupel, also generierte in Kombination mit vorhandenen, werden der Erklärungsmenge hinzugefügt.

Schritt 5 Dieser Schritt nach der Berechnung der einzelnen Muster ausgeführt. Falls eine Union in der Anfrage vorhanden war, werden hier die Erklärungen zusammengeführt Wenn nötig, filtert dieser Schritt die Ausgabe erneut und entfernt redundante Antworten. Danach wird die Erklärungsmenge ausgegeben.

Im Vergleich zu NedExplain unterscheiden sich die Algorithmen in den Möglichkeiten der Eingabe kaum. Beide erlauben Aggregationen, Joins, Vereinigungen, Projektionen und Selektionen. Dazu sind komplexe Bedingungstupel möglich, wobei Artemis Zusammenhänge zwischen den Tupeln erlaubt. In der Antwort verfolgen die Algorithmen konträre Ansätze: Während NedExplain die Datenbank als unveränderlich annimmt, passt Artemis diese an und ändert die Anfragen nicht. In der Laufzeit ist Artemis wesentlich langsamer als Why-Not und damit auch als NedExplain [8].

2.2.3 Ted++

Ted++ [1] gehört wie Why-Not und NedExplain zu den Algorithmen mit Anfrage-basierten Antworten. Die generierten Antworten sind Polynome und nicht nur einzelne Anfrageoperatoren, wie bei den vergleichbaren Algorithmen. Hintergrund dafür ist die Unvollständigkeit von Antworten, die sich auf eine Baumstruktur festlegen. Dabei werden Antworten verworfen, die bei einer anderen Baumdarstellung sichtbar sind. Ted++ vermeidet dieses Problem, indem die Anfrage als Mengen an Operatoren ohne Ordnung betrachtet wird.

Die Why-Not Frage besteht aus einer Anfrage zu der ein Bedingungstupel gehört. Möglich sind sowohl einfache Bedingungen, die ein Feld mit einer Konstante verknüpfen als auch komplexe Bedingungen zwischen zwei Feldern. Kompatible Tupel sind alle Tupel aus dem Kreuzprodukt der Relationen, die die Why-Not Frage erfüllen. Für jedes dieser kompatiblen Tupel wird nun geprüft, welche Bedingung aus den Selektionen oder Joins dieses Tupel filtert.

Damit ergibt sich als Antwort das Polynom PEX (Formel 2.1). Die Menge der Bedingungen $\mathcal{E}$, die das Tupel herausfiltern wird als Multiplikation der einzelnen Bedingungen c dargestellt und steht für die Konjunktion der einzelnen Bedingungen. Der Koeffizient $coef_{\mathcal{E}}$ beschreibt, für wie

viele Tupel, diese Menge an Bedingungen gilt. Die Summe über alle möglichen Kombinationen von Bedingungen steht für eine Disjunktion, denn es genügt eine der Bedingungsmengen anzupassen um die Why-Not Frage zu erfüllen.

$$PEX = \sum_{\mathcal{E} \in E} coef_{\mathcal{E}} \prod_{c \in \mathcal{E}} c \tag{2.1}$$

Die naive Implementierung für polynome Why-Not Antworten ist der Ted [2] Algorithmus. Dessen Laufzeit hängt vom Kreuzprodukt aller kompatiblen Tupel ab. Das ist nicht für einen praktischen Einsatz geeignet, weshalb Ted++ entwickelt worden ist. Ted++ vermeidet das Überprüfen aller kompatiblen Tupel. Das reduziert sowohl Platz- als auch Zeitbedarf. Stattdessen iteriert Ted++ über die Menge der möglichen Antworten, die wesentlich kleiner ist, als die Menge der kompatiblen Tupel. Dazu wird die Arbeit auf Partitionen aufgeteilt und die Anzahl der gefilterten Tupel pro Operator berechnet. Daraus wird dann das Polynom berechnet und ausgegeben.

Im Vergleich zu NedExplain liefert Ted++ differenziertere Ergebnisse und kann mehr Schwachstellen in der Anfrage aufzeigen, da er nicht an den festen Baum von NedExplain gebunden ist. Bidoit et al. zeigen in einen Laufzeitvergleich, bei dem Ted++ in den meisten Szenarien schneller ist [1]. In den für NedExplain besseren Szenarien sind die filternden Operationen nah an den Tabelleninstanzen und deshalb schneller gefunden als bei Ted++, der alle Selektionen überprüft und keinen Vorteil durch die Struktur der Anfrage erhält. Zum aktuellen Zeitpunkt unterstützt NedExplain Aggregationen bereits, während dies für Ted++ für zukünftige Arbeit geplant ist.

2.2.4 ConQueR

ConQueR [17] ist ein Why-Not Provenance-Algorithmus und steht für Constraint-based Query Refinement. Die Antworten sind Anfrage-basiert, wobei eine modifizierte Anfrage erstellt wird. Damit kann ConQueR sowohl zur Suche nach Problemen in der Anfrage als auch zur Generierung von Änderungen verwendet werden. Das erlaubt auch Szenarien, bei denen der Nutzer nur Parameter an der Anfrage ändern kann und die Anfrage selbst nicht kennt. Dabei versucht der Algorithmus gleichzeitig möglichst wenig Änderungen an der Anfrage vorzunehmen und dabei nur die fehlenden Tupel in das Ergebnis einzuschließen.

Die SQL-Anfrage der Why-Not Frage kann Selektionen, Projektionen, Joins und Aggregationen enthalten. Mengenoperatoren werden nicht unterstützt. Es sind mehrere Bedingungstupel möglich, wobei deren Felder neben variabel oder konstant auch noch als unwichtig gekennzeichnet sein können. Als unwichtig oder irrelevant markierte Felder können in der Ausgabe einen beliebigen Wert annehmen und entsprechen variablen Feldern, die nicht im Bedingungsfeld verwendet werden. Die Why-Not Frage kann mehrere Bedingungstupel enthalten, die aber keine Abhängigkeiten untereinander enthalten dürfen.

Als Ausgabe generiert ConQueR die verbesserte Anfrage, die neben allen Tupeln der ursprünglichen Ausgabe, auch für jedes Why-Not Tupel ein passendes Tupel enthält. Dieses stimmt in den konstanten Feldern überein und erfüllt die Bedingungen, die in der Why-Not Frage festgelegt wurden.

Zum internen Vergleich modifizierter Anfragen verwendet ConQueR zwei Metriken:

Die *dissimilarity*-Metrik bildet die Änderungen von der ursprünglichen Anfrage zur modifizierten Version in einem Wert ab. Dabei werden die Veränderungen unterschiedlich gewichtet. Eine Änderung an einer Konstanten einer Selektion wird am schwächsten gewichtet. Danach kommt das Hinzufügen einer neuen Selektionsklausel und dann das Einfügen oder Entfernen eines neuen Join-Prädikats. Als größte Änderung zählt das Verändern der FROM-Klausel. Damit werden kleine und für den Benutzer leicht erkennbare Modifikationen bevorzugt.

Die *imprecision*-Metrik bewertet, wie nah die Ausgabe der neu erstellten Anfrage mit den Anforderungen aus der Why-Not Frage und dem ursprünglichen Ergebnis übereinstimmt. Dabei werden die hinzugekommenen Tupel gezählt, abzüglich der Tupel, die für die Erfüllung der Why-Not Frage benötigt werden. Alle weiteren Tupel sind nicht nötig und sollten im optimalen Fall nicht in der Ausgabe sein.

Diese Metriken werden zur Berechnung der *skyline* Anfragen verwendet. Hier sind alle modifizierten Anfragen enthalten, die in den einzelnen Metriken möglichst niedrige, also gute, Werte erreicht haben. Diese werden nun nach einer Kombination der beiden bisherigen Metriken sortiert: Eine modifizierte Anfrage Q_1 ist besser als eine andere Q_2, wenn Q_1 in allen Metriken mindestens so gleich gute Werte erreicht hat wie Q_2 und Q_1 in mindestens einer Metrik einen besseren Wert, in diesem Fall kleineren Wert, erreicht hat.

Die eigentliche Ausführung des Algorithmus besteht aus zwei Schritten: ConQueR^S und ConQueR^P.

Im ersten Schritt werden aus der eingegebenen Anfrage und der Why-Not Frage möglichst ähnliche modifizierte Anfragen generiert. Der Algorithmus beginnt dabei mit dem aktuellen Schema der Anfrage und passt dieses erst an, falls keine passende modifizierte Anfrage gefunden wurde. Dieser Schritt wird solange wiederholt bis Anfragen gefunden wurden.

Daraus berechnet der ConQueR^P die skyline Anfragen. Dabei wird versucht durch das Hinzufügen von Prädikaten um die Genauigkeit zu verbessern. Diese werden dann als Antwort auf die Why-Not Frage ausgegeben.

Im Vergleich unterstützt NedExplain in den Anfragen zusätzlich noch Vereinigungen. Sonst sind die Möglichkeiten in den Bedingungstupeln gleich. Mit den Metriken führt ConQueR noch eine Gewichtung der Ergebnisse durch. Damit wird indirekt eine Gewichtung der Problemstellen gemacht, die NedExplain nicht enthält. Davon abgesehen ist der Umfang der Ausgabe gleich, nur die Art der Darstellung ist unterschiedlich.

2.2.5 Conseil

Conseil [8] ist ein Why-Not Provenance-Algorithmus der hybride Antworten berechnet. Diese bestehen zum Teil aus Anfrage-basierte Erklärungen und zum Teil aus Instanz-basierten Erklärungen, die zusammen angewendet die fehlenden Tupel erklären. Dabei umgeht der Algorithmus das Problem, keine Erklärungen zu finden, wenn keine passenden Tupel in der Instanz bereits vorhanden sind.

Die hybride Erklärung ist als $\phi_H = \{\phi_{IB}, \phi_{QB}\}$ definiert. Dabei beinhaltet diese sowohl alle Instanz-basierten Erklärungen aus ϕ_{IB} als auch alle Anfrage-basierten Erklärungen aus ϕ_{QB}. Eine Teilmenge davon kann, muss aber nicht die fehlenden Tupel erklären.

Conseil unterstützt als Eingabe Anfragen, die Selektionen, Aggregationen, Projektionen, Joins, karthesische Produkte und maximal einen Mengenoperator enthalten. Damit ist im Vergleich zu den anderen Algorithmen hier die größte Freiheit gegeben. Möglich ist in der Why-Not Frage nur ein Bedingungstupel. Die Erweiterung auf die gleichzeitige Berücksichtigung mehrerer entspricht der wiederholten Ausführung mit jeweils dem nächsten Tupel, solange keine Bedingungen über mehrere Tupel existieren. Dieser Fall wird bisher nicht unterstützt.

Die Ausführung ist in 4 Schritte aufgeteilt:

Schritt 1: Definition generischer Erklärungen Diese teilt sich auf in einen Instanz-bezogenen Teil, der die Abhängigkeiten des fehlenden Tupels auf die Relationen enthält, und auf einen Anfrage-bezogenen Teil. Dieser besteht aus allen Bedingungen aus Selektionen und Joins, die das fehlende Tupel betreffen.

Schritt 2: Annotieren der generischen Erklärungen Jedes Element der generischen Erklärung wird mit einer der Eigenschaften: *passing*, *blocking* oder *ambiguous*. *Passing* steht für einen Operator, für den es sicher ist, dass er keine fehlenden Tupel aus dem Ergebnis entfernt. *Blocking* ist für den Fall, dass der Operator filtert. *Ambiguous* sind Operatoren, die keinen Einfluss auf die generische Erklärung haben.

Schritt 3: Ableitungsberechnung Aus den annotierten generischen Erklärungen werden Ableitungen nach 4 Ableitungsregeln berechnet.

Schritt 4: Berechnung der Erklärung Aus den Ableitungen werden dann die eigentlichen Erklärungen berechnet. Dabei wird jeweils eine Instanz-basierte und eine Anfrage-basierte Erklärung betrachtet und mithilfe einer Kostenfunktion eine ausgewählt. Ziel ist es möglichst wenig Tupel neben den fehlenden Tupeln hinzuzufügen.

Im Vergleich zu NedExplain bringt Conseil einen umfangreicheren Ansatz zur Beantwortung von Why-Not Fragen mit. Dazu wird in [8] ein Framework zur Integration diverser Why-Not Provenance-Algorithmen vorgestellt, das auch für NedExplain verwendbar ist. Damit bringt Conseil in einem vergleichbaren Interface in mehr Szenarien Erklärungen bei vergleichbarer Laufzeit. Die hybride Antwort erlaubt zusätzlich die Erklärungen konstant relativ klein zu halten im Vergleich zu Algorithmen mit nur einem Antworttyp wie NedExplain/Why-Not mit

nur Anfrage-basierten Antworten bzw. Artemis mit nur Instanz-basierten Antworten. Das ist gleichzeitig auch der Nachteil des hybriden Ansatzes, der verlangt, dass der Benutzer sowohl Daten als auch Anfrage modifizieren darf.

2.2.6 Besonderheiten von NedExplain

NedExplain generiert, wie die anderen Why-Not Provenance-Algorithmen, Erklärungen für die Why-Not Frage. Dabei benötigt der Algorithmus nicht alle Felder. Zusätzlich wird allerdings eine abweichende Darstellung und Benennung verwendet.

Die Why-Not Frage von NedExplain entspricht den fehlenden Tupeln aus der allgemeinen Definition der Why-Not Frage aus Abschnitt 2.2. Dateninstanz und die SQL-Anfrage, zu der die Erklärungen berechnet werden, sind weitere Parameter für den Algorithmus. Die Why-Not Frage von NedExplain wird als Prädikat $\mathcal{P}$ dargestellt. Sie enthält die gleichen Informationen, wie diese auch in die fehlenden Tupel der allgemeinen Definition eingetragen werden können. Jedes fehlende Tupel wird aufgeteilt in ein Variablentupel und ein Bedingungstupel. Das Variablentupel enthält alle Attribute des fehlenden Tupels, die entweder auf konstante oder variable Werte gesetzt werden. Attribute, die einen beliebigen Wert erhalten dürfen, kommen nicht in der Why-Not Frage vor. Das Bedingungstupel besteht aus dem Variablentupel und einer logischen Bedingung. Diese kann sich auf Attribute beziehen, die als variabel im Variablentupel gesetzt werden.

Gleichung (2.2) zeigt das Prädikat $\mathcal{P}$ für die folgende Beschreibung des fehlenden Tupels: „Wieso fehlt das Tupel mit dem Namen ‚Max‘ und dem Alter > 5?“.

$$\mathcal{P} = ((Name : Max, Alter : x1), x1 > 5) \tag{2.2}$$

Das Variablentupel ist $(Name : Max, Alter : x1)$. Das Attribut $Name$ hat den konstanten Wert Max. Das Attribut $Alter$ ist variabel und mit einer Bedingung verknüpft. Deshalb steht anstelle des konstanten Wertes ein Variablennamen. Das Bedingungstupel besteht aus dem genannten Variablentupel und der Bedingung $x1 > 5$. Diese nimmt dann Bezug auf die Variable $x1$ und verwendet diese in der Bedingung.

NedExplain verwendet aus der allgemeinen Definition die fehlenden Tupel im Format der Why-Not Frage für NedExplain. Dazu sind die Datenbankinstanz und die Anfrage darauf weitere Parameter. Mehrere Anfragen gleichzeitig werden von NedExplain nicht unterstützt. Genauso sind keine Parameter zur Konfiguration der Ausführung vorgesehen. Im Folgenden beziehe ich mich auf die Definition von NedExplain zur Why-Not Frage, da die Struktur vom Algorithmus erwartet wird.

3 Erklärung von NedExplain an Beispielen

NedExplain sucht für die fehlenden Tupel mögliche kompatible Tupel, die der Algorithmus dann durch die einzelnen Operationen der Anfrage verfolgt. Wie das im Algorithmus abgebildet ist, erkläre ich in diesem Kapitel anhand von zwei Beispielen.

Basis für die Beispiele ist die folgende Datenbankinstanz. Sie besteht aus zwei Tabellen (Abbildung 3.1). Die Produzent-Tabelle hat die Felder Name und ID, wobei die ID der Primärschlüssel der Relation ist und als Fremdschlüssel in der zweiten Tabelle auftaucht. Die Film-Tabelle hat drei Einträge, zu denen jeweils der Name, das Veröffentlichungsjahr, ein fiktiver Preis und der dazugehörige Produzent gespeichert ist. Zur besseren Referenz sind alle Tupel nummeriert.

Die dazugehörigen Beispiele sind in Abbildung 3.2 dargestellt. Jedes Beispiel besteht aus der SQL-Anfrage, für die dann mit NedExplain nach den Erklärungen für die fehlenden Tupel gesucht werden soll. Diese wird im Verlauf der Berechnung des Algorithmus als Anfragebaum dargestellt. Zu jedem Beispiel ist das Ergebnis angegeben und die Why-Not Frage nach fehlenden Tupeln. Diese steht einmal ausformuliert und einmal als Why-Not Frage für NedExplain in Form des Prädikats $\mathcal{P}$ bei jedem Beispiel am Schluss.

Die beiden Beispielszenarien überschneiden sich an vielen Stellen, da das Beispiel Abbildung 3.2b die Anfrage des Beispiels Abbildung 3.2a nur um eine Aggregation mit Gruppierung erweitert. Dies hat Einfluss auf die Berechnung von NedExplain und wird beispielsweise an der anderen Struktur des Anfragebaumes bereits sichtbar. Im folgenden Abschnitt 3.1 werde ich die Ausführung von NedExplain am Beispiel ohne die Aggregation erklären und dann im Anschluss in Abschnitt 3.2 die Unterschiede im Ablauf auf einer Anfrage mit Aggregation herausarbeiten.

Produzent

	Name	ID
t1	Disney	p1
t2	Pixar	p2
t3	Illumination Entertaiment	p3

Film

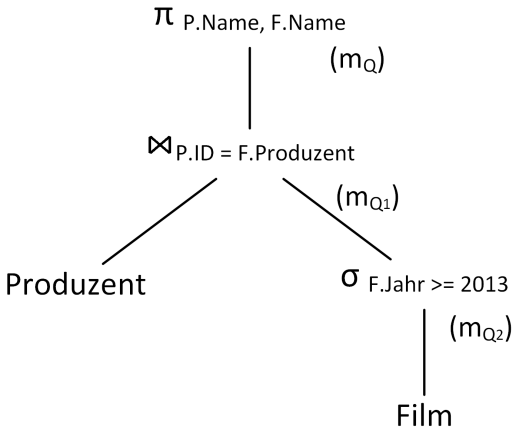
	Name	Jahr	Preis	Produzent
t4	Ich - einfach unverbesserlich	2010	14	p3
t5	Die Eiskönigin	2013	15	p1
t6	Alles steht Kopf	2015	18	p2
t7	Zoomania	2016	22	p1

Abbildung 3.1: Beispieldatensatz

SQL-Anfrage

```
Q = SELECT P.Name, F.Name
FROM Produzent P, Film F
WHERE P.ID = F.Produzent
  AND F.Jahr >= 2013;
```

Anfragebaum



Ergebnis

	P.Name	F.Name
t8	Disney	Die Eiskönigin
t9	Pixar	Alles steht Kopf
t10	Disney	Zoomania

Why-Not Frage

Wieso kommt hier „Ich – einfach unverbesserlich“ nicht in den Titeln vor?

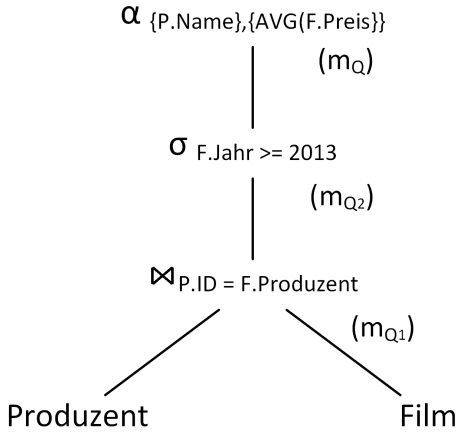
$\mathcal{P}$ = (F.Name:Ich – einfach unverbesserlich)

(a) Beispielszenario ohne Aggregation

SQL-Anfrage

```
Q = SELECT P.Name, AVG(F.Preis)
FROM Produzent P, Film F
WHERE P.ID = F.Produzent
  AND F.Jahr >= 2013
GROUP BY P.Name;
```

Anfragebaum



Ergebnis

	P.Name	AVG(F.Preis)
t11	Disney	18,5
t12	Pixar	18

Why-Not Frage

Wieso ist „Illumination Entertaiment“ nicht im Ergebnis?

$\mathcal{P}$ = (P.Name:Illumination Entertaiment)

(b) Beispielszenario mit Aggregation

Abbildung 3.2: Beispielszenarien mit Anfrage, von NedExplain generiertem Anfragebaum, Ergebnis und Why-Not Frage ausformuliert und als Prädikat $\mathcal{P}$. Die Beschriftung der Knoten des Anfragebaums dient zur Identifizierung.

3.1 Beispiel ohne Aggregation

Im Beispiel (Abbildung 3.2a) werden aus der Datenbank alle Filme mit den dazugehörigen Produzenten abgefragt, die im Jahr 2013 oder später veröffentlicht wurden. Das Ergebnis besteht aus den Tupeln t8-t10. Der Film „Ich - einfach unverbesserlich“, das Tupel t4 in der Datenbank, taucht nicht in der Ergebnismenge auf. In diesem kleinen Beispiel ist die Antwort mit Blick auf die Datengrundlage und Anfrage ersichtlich: Die Selektion über das Jahr entfernt den Film, der bereits 2010 veröffentlicht worden ist.

Um dies durch NedExplain zu erhalten, formuliert man die „Wieso kommt hier ‚Ich - einfach unverbesserlich‘ nicht in den Titeln vor?“ Frage als Why-Not Frage. Die Formulierung bezieht sich nur auf das Attribut $F.Name$, das den Wert $Ich - einfachunverbesserlich$ haben soll. Das Prädikat $\mathcal{P}$ aus Abbildung 3.2a bildet diese Frage als Why-Not Frage ab. Das Attribut $P.Name$ des fehlenden Tupels ist für die Frage egal und taucht deshalb auch nicht im Prädikat auf.

NedExplain erhält nun als Eingabe die Datenbank mit den Relationen „Produzent“ und „Film“ mit der Anfrage Q_1 und und dem Prädikat $\mathcal{P}$. Der Algorithmus teilt sich in zwei Phasen auf: Vorbereitung (Abschnitt 3.1.1) und Berechnung der Erklärungen (Abschnitt 3.1.2).

3.1.1 Vorbereitung

In der Vorbereitung werden die globalen Datenstrukturen initialisiert und die Eingabe wird für den Berechnungsschritt vorbereitet. Die Schritte entsprechen den Schritten aus Kapitel 3.1 von [3].

Auflösung von Umbenennungen NedExplain benötigt zur Suche nach den kompatiblen Tupeln absolute Referenzen in der Why-Not Frage. Deshalb macht dieser Schritt Umbenennungen durch die Anfrage im Prädikat P rückgängig, sodass eine Why-Not Frage übrig bleibt, die nur noch Referenzen auf Felder in Relationen hat. Dabei werden einerseits neue Namen, die in der Aggregation oder Projektion beispielsweise mit „AS“ gesetzt worden sind, entfernt. Andererseits werden Bedingungen, die sich auf eine mit einem Join zusammengeführte Spalte beziehen, auf die Ursprungsrelationen aufgelöst.

Im Beispiel ist die einzige Referenz $F.Name$ bereits absolut, weshalb keine Änderungen am Prädikat nötig sind.

Finden von kompatiblen Tupeln NedExplain sucht in diesem Schritt in allen Relationen nach kompatiblen Tupel, die in der Berechnung durch die Operatoren verfolgt werden. Gleichzeitig werden alle Relationen gesammelt, die keine kompatiblen Tupel direkt enthalten können. Kompatible Tupel können aus Relationen kommen, die in der Why-Not Frage mit im Variablentupel verwendet werden. Ein Tupel ist kompatibel, wenn es alle Bedingungen der Why-Not Frage auf die Relation des Tupels erfüllt.

Im Beispiel bezieht sich das Prädikat mit Attribut $F.Name$ auf die Relation „Film“. Darin findet sich das kompatible Tupel t_4, das im Feld Name den gesuchten Wert enthält. Alle kompatiblen Tupel werden in der Menge Dir^{tc} gespeichert. Die Relationen, die keine kompatiblen Tupel enthalten, sind in $inDir^{tc}$ gesammelt. Diese Menge wird indirekt kompatible Tupel genannt, da sie beispielsweise als Joinpartner zum fehlenden Tupel beitragen können. Das ist in diesem Fall die Relation „Produzent“.

Kanonisieren der Anfrage NedExplain berechnet eine Baumdarstellung der Anfrage, die dann in der Berechnung traversiert wird. Eine vorhandene Vereinigung bildet immer die Wurzel. Darunter wird die Projektion bzw. Aggregation der vereinigten Abfrage angeordnet. Wenn keine Vereinigung vorhanden ist, beginnt der Baum bei der Projektion bzw. Aggregation. Die Relationen, auf die sich die Anfrage bezieht, bilden die Blätter der Baumdarstellung. Dazwischen werden vorhandene Joins und Selektionen eingeordnet.

Die Abbildung einer SQL-Anfrage als Baum ist normalerweise nicht eindeutig. Deshalb verwendet NedExplain zwei Regeln, die die Darstellung eindeutig machen. Die erste Regel verlangt, dass Selektionen möglichst weit weg von der Wurzel im Anfragebaum angeordnet werden sollen. Damit werden diese als Erklärung schneller gefunden, da der Algorithmus von den Blättern aus den Baum verarbeitet. Die zweite Regel bezieht sich auf Anfragen mit Aggregationen und wird in Abschnitt 3.2 erklärt.

Jeder Knoten im Baum steht für eine SQL-Anfrage. Diese enthält den Operatoren des Knotens und aller Kinder. Diese Anfrage wird *Unteranfrage* des Knotens genannt. Die Wurzel entspricht der SQL-Anfrage, die als Eingabe mitgegeben wird.

Für das Beispiel wird die Projektion $\pi_{P.Name,F.Name}$ als Wurzel des Baumes gesetzt. Die Blätter sind die Relationen „Produzent“ und „Film“. Die Selektion $\sigma_{F.Jahr \geq 2013}$ (m_{Q_2}) bezieht sich nur auf die Relation „Film“ und wird direkt darüber angeordnet. Der Join $\bowtie_{P.ID=F.Produzent}$ (m_{Q_1}) wird noch zwischen Selektionen, Relation „Produzent“ und der Projektion eingefügt. Auf welcher Seite des Joins die Kinder angeordnet sind, spielt keine Rolle, da der Algorithmus die Knoten mit absteigendem Abstand zur Wurzel durchgeht und beide Kinder den gleichen Abstand haben.

Die Unterabfragen zu den Knoten sind in Listing 3.1 aufgelistet. Die Unterabfrage zu m_{Q_2} bzw. m_{Q_1} enthält dabei jeweils den eigenen Operator zu der bzw. den Unterabfragen der Kinder.

Initialisieren von tab_Q Die Tabelle tab_Q enthält den Berechnungsfortschritt von NedExplain. In diesem Schritt wird diese mit den bisher vorhandenen Informationen initialisiert.

tab_Q speichert zu jeder Unterabfrage Eingabe, Ausgabe, kompatible Tupel, den Abstand zu der Wurzel, eine Referenz auf den Elternknoten und die Art des Operators, also beispielsweise Selektion, Relation etc.. Diese wird nun mit dem Baum gefüllt. Dabei werden die Knoten mit absteigendem Abstand eingefügt. Für jede Relation werden die Eingabe und die kompatiblen Tupel gesetzt, da diese bereits berechnet worden sind.

Listing 3.1 Unterabfragen zum Beispiel Abbildung 3.2a

```
1 -- Unteranfrage fuer den Knoten Film
2 SELECT * FROM Film;
3 -- Unteranfrage fuer den Knoten Produzent
4 SELECT * FROM Produzent;
5 -- Unteranfrage fuer den Knoten mQ2 erweitert die Unteranfrage zum Film
6 SELECT * FROM Film WHERE F.Jahr >= 2013;
7 -- Unteranfrage fuer den Knoten mQ1 erweitert und kombiniert die Unteranfrage zur
      Relation Produzent und mQ2
8 SELECT * FROM Film, Produzent WHERE F.Jahr >= 2013 AND P.ID = F.Produzent;
```

	Film	**Produzent**	$\mathbf{m_{Q_2}}$	$\mathbf{m_{Q_1}}$	$\mathbf{m_Q}$
Eingabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$			
Kompatible Tupel	$\{t_4\}$	$\emptyset$			
Ausgabe					
Abstand	3	2	2	1	0
Eltern	m_{Q_2}	m_{Q_1}	m_{Q_1}	m_Q	
Art	Relation	Relation	Selektion	Join	Projektion

Tabelle 3.1: tab_Q für Szenario mit den Kürzeln für die Knoten des Anfragebaumes aus Abbildung 3.2a

Für das Beispiel ergibt sich die folgende Initialisierung von tab_Q Tabelle 3.1. Der erste Eintrag ist die Relation „Film“, da diese als einzige den Abstand 3 zur Wurzel hat. Eingabe aller Relationen ist deren Datenbankinstanz und ergibt dann auch deren Ausgabe, die aber erst in der zweiten Phase gesetzt wird. Für bessere Übersicht sind die Elternoperatoren mit dem Kürzel angegeben. Die zweite Spalte enthält die zweite Relation, die aber eine leere Menge an kompatiblen Tupeln mitbringt. Der Rest der Tabelle enthält nur Informationen zur Struktur der Einträge, da die Daten dazu erst in Abschnitt 3.1.2 berechnet werden.

Initialisierung der anderen globalen Strukturen Neben tab_Q hält NedExplain weitere globale Strukturen vor, die im Laufe der Berechnung gefüllt werden.

- *EmptyOutputMan* ist eine Menge an Unterabfragen des Anfragebaumes, die eine leere Ausgabe generieren. Diese werden benötigt, um sekundäre Erklärungen zu berechnen, die indirekt kompatible Tupel berücksichtigt. Anfragen mit leerer Ausgabe entfernen neben allen kompatiblen Tupeln auch mögliche indirekt kompatible Tupel aus ihrer Eingabe.
- *Non-PickyMan* ist die Menge der Unterabfragen, die in der Ausgabe kompatible Tupel enthalten.
- *PickyMan* ist eine Menge von Tupeln, die jeweils eine Unterabfrage und eine Menge an Tupeln enthalten. Diese Tupel sind kompatible Tupel, die in der Eingabe der Unterabfrage vorhanden sind, aber in der Ausgabe keinen Nachfolger haben.

Algorithmus 3.1 Ausschnitt aus NedExplain [3]

```
1:  for int i = 0, ..., Tab_Q.size() − 1 do
2:      m = Tab_Q[i]
3:      if checkEarlyTermination(m) then
4:          return DetailedAnswer();
5:      end if
6:      m.Ausgabe ← m(m.Eingabe);
7:      p ← m.Eltern;
8:      p.Eingabe ← p.Eingabe ∪ m.Ausgabe;
9:      if m.Output = ∅ then
10:         EmptyOutputMan ← EmtpyOutputMan ∪{m}
11:         if m.KompatibleTupel ≠ ∅ then
12:             PickyMan ← PickyMan ∪{m, m.KompatibleTupel}
13:         end if
14:     end if
15:     if m.Art ≠ "Relation" then
16:         p.KompatibleTupel ← p.KompatibleTupel ∪FindSuccessors(m);
17:     else
18:         if m.KompatibleTupel ≠ ∅ then
19:             p.KompatibleTupel ← p.KompatibleTupel ∪ m.KompatibleTupel;
20:             Non-PickyMan ← Non-PickyMan ∪{m};
21:         end if
22:     end if
23: end for
24: return null;
```

Zusätzlich wird aus den vorherigen Schritten noch $inDir^{tc}$, die Menge aller indirekt kompatiblen Tupel, im Algorithmus verwendet. Im Gegensatz zu den anderen globalen Strukturen, wird $inDir^{tc}$ aber nicht mehr verändert.

3.1.2 Berechnung der Erklärungen

NedExplain verfolgt in der Berechnung die gefundenen kompatiblen Tupel durch die einzelnen Operatoren. Gefundene Nachfolger kompatibler Tupel werden zu den kompatiblen Tupeln hinzugefügt. Operatoren, die kompatible Tupel entfernen, werden in den globalen Strukturen gespeichert. Wenn alle möglichen kompatiblen Tupel aus den Zwischenergebnissen entfernt werden, bricht der Algorithmus ab und berechnet auf Basis der globalen Strukturen die Erklärungen. Basis für die Verfolgung der kompatiblen Tupel ist die Tabelle tab_Q, die vom Algorithmus von links nach rechts ausgefüllt wird. Die einzelnen Schritte stelle ich nun anhand des Pseudocodes (Algorithmus 3.1) vor.

Zu Beginn setzt der Algorithmus die Variable m auf die aktuelle Unterabfrage aus tab_Q. Danach lässt sich die Berechnung in drei Teile aufteilen:

Zeilen 3-5 Am Anfang überprüft der Algorithmus, ob noch kompatible Tupel zur Nachverfolgung vorhanden sind. Wenn nicht bricht er ab und berechnet mit $DetailedAnswer()$ die Erklärungen, die dann ausgegeben werden.

Die Funktion $checkEarlyTermination()$ überprüft unterschiedliche Abbruchkriterien: Das erste Feld von tab_Q wird immer verarbeitet. Sonst kann abgebrochen werden, wenn die aktuelle Unterabfrage einen noch nicht bearbeiteten Abstand zur Wurzel hat. Es wird also sichergestellt, dass die Berechnung für alle Knoten mit gleichem Abstand entweder gar nicht oder ganz durchgeführt wird. Wenn nun der zu prüfende Knoten zu einem noch nicht berechneten Abstand gehört, werden noch zusätzliche Prüfungen gemacht: Es genügt um die Berechnung fortzusetzen, dass eine der Unterabfragen des noch nicht berechneten Abstandes kompatible Tupel in der Eingabe hat. Zum anderen wird nicht abgebrochen, wenn es noch nicht bearbeitete Relationen in tab_Q gibt.

Damit wird garantiert, dass nur abgebrochen wird, wenn keine Möglichkeit besteht, in den nicht ausgefüllten Feldern von tab_Q kompatible Tupel zu finden.

Im Beispiel wird beim Join m_{Q_1} abgebrochen, da dieser die erste Unterabfrage mit dem Abstand 1 ist. Zusätzlich hat sowohl die Relation „Produzent“ als auch die Unterabfrage m_{Q_2} keine kompatiblen Tupel und die noch nicht bearbeiteten Felder enthalten keine Relation. Zu den vorherigen Spalten von tab_Q gibt die Funktion „falsch“ zurück, da die Relation „Film“ an erster Stelle steht, die Relation „Produzent“ eine Relation ist und deshalb überprüft werden muss und bei m_{Q_2} bereits eine Unterabfrage mit Abstand 2 berechnet worden ist und deshalb die Berechnung für diesen Abstand vervollständigt wird.

Zeilen 6-14 Wenn die Berechnung fortgesetzt wird, berechnet der Algorithmus in diesem Teil die Ausgabe der Unterabfrage und trägt diese in tab_Q als Ausgabe und als Eingabe für den Elternknoten ein. Damit wird sichergestellt, dass die Eingabe für die später zu berechnenden Unterabfragen gesetzt sind. Die berechnete Ausgabe der Unterabfrage wird der Eingabe des Elternknoten hinzugefügt.

Im Sonderfall, dass die Ausgabe leer ist, wird die Unterabfrage im EmptyOutputMan gespeichert. Wenn es kompatible Tupel in der Eingabe bei leerer Ausgabe gibt, werden diese offensichtlich herausgefiltert und werden mit der Anfrage in den PickyMan eingefügt.

Im Beispiel ist die Ausführung dieser Schritte für die Relationen „Film“ und „Produzent“ trivial. Die Unterabfrage m_{Q_2} enthält danach als Eingabe die Tupelmenge $\{t_4, t_5, t_6, t_7\}$ und das kompatible Tupel $\{t_4\}$. Die Anwendung der Selektion auf die Eingabe ergibt die Ausgabe $\{t_5, t_6, t_7\}$, die nicht leer ist (Tabelle 3.2). Damit fallen die weiteren Überprüfungen weg.

	Film	**Produzent**	$\mathbf{m_{Q_2}}$	$\mathbf{m_{Q_1}}$	$\mathbf{m_Q}$
Eingabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3, t_5, t_6, t_7\}$	
Kompatible Tupel	$\{t_4\}$	$\emptyset$	$\{t_4\}$	$\emptyset$	
Ausgabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$	$\{t_5, t_6, t_7\}$		
Abstand	3	2	2	1	0
Eltern	m_{Q_2}	m_{Q_1}	m_{Q_1}	m_Q	
Art	Relation	Relation	Selektion	Join	Projektion

Tabelle 3.2: tab_Q für Szenario Abbildung 3.2a nach der Ausführung

Zeilen 15-22 In diesem Schritt überprüft NedExplain der Einfluss des Operators der Unterabfrage auf die kompatiblen Tupel. Nachfolger der kompatiblen Tupel werden zu den kompatiblen Tupel des Elternknotens hinzugefügt. Dabei werden Relationen gesondert behandelt, da für diese bereits in der Vorbereitung die kompatiblen Tupel gesucht werden.

Bei einer Relation werden die eventuell in der Vorbereitung gefundenen Tupel zu den kompatiblen Tupeln der Elternabfrage gespeichert. Die Relation wird zu den Unterabfragen im Non-PickyMan hinzugefügt, wenn sie kompatible Tupel in der Ausgabe hat. Für alle anderen Arten der Unterabfrage berechnet die Funktion $FindSuccessors()$ die Nachfolger der kompatiblen Tupel der Eingabe der Unterabfrage:

Diese geht jedes Ausgabetupel der Unterabfrage durch und prüft, ob es sich um einen validen Nachfolger der kompatiblen Tupel handelt. Valide bedeutet in diesem Fall, dass die Herkunft des Tupels vollständig in den kompatiblen und indirekt kompatiblen Tupeln liegt. Abhängig nachdem ob Nachfolger gefunden werden, wird m zum Non-PickyMan bzw. m mit den blockierten kompatiblen Tupeln zum PickyMan hinzugefügt.

Im Beispiel sind die ersten zwei Spalten Relationen, die zum $else$ Zweig der Verzweigung gehören. Die Relation „Film“ hat die kompatiblen Tupel t_4, weshalb dieses zu den kompatiblen Tupeln von m_{Q_2} hinzugefügt wird und die Relation zum Non-PickyMan gehört. Die zweite Relation „Produzent“ hat keine kompatiblen Tupel und setzt damit auch keine für die Elternabfrage. Für m_{Q_2} wird $FindSuccessors(m)$ ausgeführt. Danach ist m_{Q_2} im PickyMan mit dem entfernten kompatiblen Tupel: $\{\mathrm{m}_{Q_2}, \{t_4\}\}$.

Nach der Ausführung ist tab_Q nicht ganz ausgefüllt, da die Berechnung endet, wenn keine kompatiblen Tupel zur Nachverfolgung mehr vorhanden sind. Der Non-PickyMan besteht aus der Relation „Film“, der PickyMan enthält $\{m_{Q_2}, \{t_4\}\}$ und der EmptyOutputMan ist leer.

Daraus kann NedExplain mit $DetailedAnswer()$ unterschiedliche Erklärungen ausgeben. Die *vollständige Beantwortung* besteht aus allen Einträgen des PickyMan, die je nach Anzahl der kompatiblen Tupel unübersichtlich werden kann. Die *reduzierte Beantwortung* enthält nur die Unterabfragen des PickyMan mit dem Nachteil, dass der Benutzer nicht mehr erkennen

kann, wie groß der Einfluss der Unterabfrage auf die kompatiblen Tupel ist. Die *sekundäre Beantwortung* bezieht sich auf die indirekt kompatiblen Tupel und gibt Unterabfragen aus, die alle indirekt kompatible Tupel aus ihrer Eingabe entfernt haben. Basis dafür ist der EmptyOutputMan.

Im Beispiel enthält die vollständige Beantwortung das Tupel $\{\mathrm{m}_{\mathrm{Q}_2}, \{t_4\}\}$, die reduzierte Beantwortung besteht nur aus $\{\mathrm{m}_{\mathrm{Q}_2}\}$ und die sekundäre Beantwortung ist leer, da der EmptyOutputMan leer ist. Ausformuliert erklärt NedExplain das fehlende Tupel durch eine zu restriktive Bedingung bei $\mathrm{m}_{\mathrm{Q}_2}$, die das einzige kompatible Tupel t_4 entfernt.

3.2 Beispiel mit Aggregation

Das zweite Beispiel erweitert die Anfrage des ersten Beispiels um eine Aggregation der Filmpreise gruppiert um die Produzenten (Abbildung 3.2b). Dafür fällt die Spalte mit dem Name der Filme weg. Da die Selektion- und Joinbedingungen gleich bleiben, sind vor der Aggregation die gleichen Tupel für das Ergebnis vorhanden, die dann auf zwei Ergebnistupel aggregiert werden. Die Knoten des Anfragebaumes sind wie im ersten Beispiel benannt, was durch die andere Baumstruktur dazu führt, dass die Nummerierung nicht in aufsteigender Reihenfolge von oben nach unten ist. Die Why-Not Frage bezieht sich auf das gleiche Tupel vor der Anwendung der Aggregation wie im ersten Beispiel. Die Referenz ist in diesem Fall der Name des Produzenten „Illumination Entertaiment“.

Die Struktur des Beispiels ist übernommen von Abschnitt 3.1, verzichtet aber auf die Erklärung der Schritte, sofern keine Unterschiede vorliegen. Ziel dieses Beispiels ist es, die Möglichkeiten von NedExplain mit Aggregationen zu zeigen und die Unterschiede im Vergleich zum normalen Ablauf hervorzuheben.

3.2.1 Vorbereitung

In der Vorbereitung werden wieder die globalen Datenstrukturen initialisiert. Die Unterschiede zeigen sich bei der Suche nach kompatiblen Tupeln und bei der Generierung des Anfragebaumes.

Auflösung von Umbenennungen Wie im Beispiel ohne Aggregationen sind alle Referenzen bereits absolut, weshalb keine Änderungen am Prädikat nötig sind.

Finden von kompatiblen Tupeln Bedingungen auf aggregierte Spalten beachtet NedExplain hier bei der Suche nach kompatiblen Tupeln nicht. Allerdings versucht der Algorithmus möglichst früh im Verlauf der Berechnung eine Bedingung auf die Aggregation zu verifizieren und generiert deshalb im nächsten Schritt einen anderen Anfragebaum als im Beispiel ohne Aggregation.

	Film	**Produzent**	$\mathbf{m_{Q_1}}$	$\mathbf{m_{Q_2}}$	$\mathbf{m_Q}$
Eingabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$			
Kompatible Tupel	$\emptyset$	$\{t_3\}$			
Ausgabe					
Abstand	3	3	2	1	0
Eltern	m_{Q_1}	m_{Q_1}	m_{Q_2}	m_Q	
Art	Relation	Relation	Join	Selektion	Aggregation

Tabelle 3.3: tab_Q für Szenario Abbildung 3.2b

Die Berechnung der kompatiblen Tupel unterscheidet sich im Beispiel nicht von der Berechnung bei der Anfrage ohne Aggregation, da sich die Why-Not Frage nicht auf eine aggregierte Spalte bezieht. Das gefundene kompatible Tupel ist t_3. Die Menge aller Relationen der indirekt kompatiblen Tupel $inDir^{tc}$ enthält die Relation „Film“.

Kanonisieren der Anfrage Mit einer Aggregation ändern sich die Regeln zu Generierung des Anfragebaumes. Gleich bleibt, dass die Aggregation die Wurzel bildet und die Relationen dazu die Blätter sind. Die Selektionen sollen wieder möglichst früh als Erklärungen gefunden werden, aber gleichzeitig soll die Aggregation möglichst oft angewandt und eventuelle Bedingungen darauf überprüft werden. Dafür werden die vorhandenen Joins so angeordnet, dass die Unterabfrage minimal ist, die alle für die Aggregation benötigten Felder enthält. Diese Anfrage zusammen mit den übrigen Relationen bildet die *visibility frontier* V. Alle Selektionen werden darüber im Baum angeordnet nach der bisherigen Regel möglichst weit entfernt von der Wurzel. Theoretisch gilt diese Regel auch für die Berechnung des Baumes ohne Aggregationen. Sie hat aber da keinen Einfluss und wird deshalb in der Erklärung des ersten Beispiels vernachlässigt.

Für das Beispiel wird die Aggregation $\alpha_{\{P.Name\},\{AVG(F.Preis)\}}$ (m_Q) als Wurzel des Baumes gesetzt. Die Blätter sind die Relationen „Produzent“ und „Film“. Der Join $\bowtie_{P.ID=F.Produzent}$ (m_{Q_1}) ist die minimale Unteranfrage, die die Felder $P.Name$ und $F.Preis$ der Aggregation enthält. Damit bildet dieser die visibility frontier V. Die Selektion $\sigma_{F.Jahr\geq 2013}$ (m_{Q_2}) wird direkt darüber angeordnet.

Initialisieren von tab_Q tab_Q wird initialisiert wie im ersten Beispiel und hat dann die Belegung wie in Tabelle 3.3 gezeigt.

Initialisierung der anderen globalen Strukturen Die übrigen Strukturen Non-PickyMan, PickyMan und EmptyOutputMan werden wie im ersten Beispiel initialisiert. Zu $inDir^{tc}$ greift der Algorithmus auf V lesend im Verlauf der Berechnung zu.

3.2.2 Berechnung der Erklärungen

Die Berechnung läuft über den gleichen Algorithmus (Algorithmus 3.1). Änderungen gibt es nur bei der Funktion $FindSuccessors(m)$, die die Nachfolger der kompatiblen Tupel sucht. Der Einfluss der einzelnen Abschnitte auf die globalen Strukturen wird jeweils kurz erläutert.

Zeilen 3-5 Die Funktion $checkEarlyTermination()$ macht keinen Unterschied, ob die Unterabfrage zu einer Aggregation gehört oder nicht.

Im Beispiel werden die Relationen jeweils nicht abgebrochen, da dort kompatible Tupel dazukommen können bzw. die Relation „Film“ der erste Eintrag von tab_Q ist. Die Unterabfrage m_{Q_1} wird berechnet, da noch kompatible Tupel zur Weiterverfolgung vorhanden sind. Das gilt auch für m_{Q_2}. Abgebrochen wird erst die Evaluation von m_Q, da keine Unterabfragen mit Abstand 1 im Non-PickyMan enthalten ist, da hier das kompatible Tupel entfernt wird.

Zeilen 6-14 Die Eingabe der Relationen wird jeweils in die Ausgabe der Relation und in die Eingabe des Joins m_{Q_1} übertragen. Dieser bekommt als Ausgabe gejointe Tupel aus den Eingabemengen, die die Joinbedingung erfüllen. Die Selektion m_{Q_2} entfernt das Tupel $t_4 \bowtie t_3$ aus den Eingaben, die Übrigen erfüllen die Bedingung. Die Aggregation wird nicht mehr berechnet, da $checkEarlyTermination()$ davor abbricht.

Zeilen 15-22 Die Nachfolger der kompatiblen Tupel werden wie im ersten Beispiel durch Iteration über alle Ausgabetupel berechnet. Wenn die Ausgabe leer ist, wird die Unterabfrage auch zum PickyMan hinzugefügt mit den kompatiblen Tupeln der Eingabe.

Die Überprüfung bei nicht leerer Ausgabe, ob die Unterabfrage kompatible Tupel entfernt hat, wird mit einer Aggregation erweitert. Dafür muss die visibility frontier V eine Unterabfrage der aktuellen Unterabfrage sein, sonst verhält sich die Funktion wie im ersten Beispiel. Dann wird nicht nur geprüft, dass jedes kompatible Tupel ein Nachfolgertupel hat, sondern auch noch, dass Ein- und Ausgabe die Bedingung erfüllen, die in der Why-Not Frage auf die Aggregation gesetzt ist. Wenn eine dieser Prüfungen verletzt ist, wird die Anfrage dem PickyMan mit den eventuell entfernten kompatiblen Tupeln hinzugefügt.

Im Beispiel wird das kompatible Tupel t_3 der Relation „Produzent“ zu den kompatiblen Tupeln von m_{Q_1} hinzugefügt. Valider Nachfolger ist das Tupel $t_4 \bowtie t_3$, da dessen Herkunft in den kompatiblen Tupeln ($\{t_3\}$) und inDir^{tc} ($\{t_4, t_5, t_6, t_7\}$) liegt. Die Aggregation m_Q hat kein kompatibles Tupel mehr, da dieses von $_{\text{Q}_2}$ herausgefiltert wird. Dabei wird sowohl das einzige kompatible Tupel entfernt und die Why-Not Frage ist nicht mehr erfüllt, wenn die Aggregation auf die Ausgabe angewendet wird. Das Endergebnis ist in Tabelle 3.4 gezeigt.

Nach der Ausführung ist tab_Q nicht ganz ausgefüllt, da die Berechnung beendet wird, wenn keine kompatiblen Tupel mehr vorhanden sind. Der Non-PickyMan besteht aus der Relation „Produzent“, der PickyMan enthält $\{m_{Q_2}, \{t_4 \bowtie t_3\}\}$ und der EmptyOutputMan ist leer.

	Film	**Produzent**	$\mathbf{m_{Q_1}}$	$\mathbf{m_{Q_2}}$	$\mathbf{m_Q}$
Eingabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$	$\{t_4, t_5, t_6, t_7\}$, $\{t_1, t_2, t_3\}$	$\{t_4 \bowtie t_3$, $t_5 \bowtie t_1$, $t_6 \bowtie t_2$, $t_7 \bowtie t_1\}$	$t_5 \bowtie t_1$, $t_6 \bowtie t_2$, $t_7 \bowtie t_1\}$
Kompatible Tupel	$\emptyset$	$\{t_3\}$	$\{t_3\}$	$\{t_4 \bowtie t_3\}$	$\emptyset$
Ausgabe	$\{t_4, t_5, t_6, t_7\}$	$\{t_1, t_2, t_3\}$	$\{t_4 \bowtie t_3$, $t_5 \bowtie t_1$, $t_6 \bowtie t_2$, $t_7 \bowtie t_1\}$	$\{t_5 \bowtie t_1$, $t_6 \bowtie t_2$, $t_7 \bowtie t_1\}$	
Abstand	3	3	2	1	0
Eltern	m_{Q_1}	m_{Q_1}	m_{Q_2}	m_Q	
Art	Relation	Relation	Join	Selektion	Aggregation

Tabelle 3.4: tab_Q für Szenario Abbildung 3.2b

Im Beispiel enthält die vollständige Beantwortung das Tupel $\{m_{Q_2}, \{t_4 \bowtie t_3\}\}$, die reduzierte Beantwortung besteht wie im ersten Beispiel nur aus $\{m_{Q_2}\}$ und die sekundäre Beantwortung ist leer, da der EmptyOutputMan leer ist. Ausformuliert erklärt NedExplain das fehlende Tupel durch die Selektion m_{Q_2}, die das einzige kompatible Tupel $t_4 \bowtie t_3$ entfernt.

4 Implementierung

NedExplain wird im Rahmen dieser Arbeit in Java implementiert und in das Eclipse Plug-in Nautilus Analyzer [9] eingebunden.

Die Implementierung nimmt dabei Rücksicht auf die zeitnahe Implementierung von Nautilus als Webapplikation [14]. Die Abhängigkeiten von Eclipse und den Bibliotheken werden gering gehalten und der Algorithmus möglichst eigenständig implementiert.

Zuerst stelle ich kurz die Architektur von Nautilus und die Einordnung von NedExplain darin vor. Im Anschluss gehe ich auf Implementierung des Algorithmus in Java ein und erkläre die Stellen, an denen die Implementierung vom Entwurf abweicht. Abschluss des Kapitels bilden die Einschränkungen dieser Implementierung und eine kurze Zusammenfassung.

4.1 Nautilus Architektur

Die Architektur (Abbildung 4.1) orientiert sich mit den einzelnen Modulen am AFT-Zyklus. Die Arbeit mit Erklärungen in der Analysephase wird vom *Explaination manager* abgedeckt. Der *Query modification manager* enthält die Methoden und Module um die Änderungen an der Anfrage in der Fixphase zu verwalten. Neben den für den Benutzer benötigten Funktionen

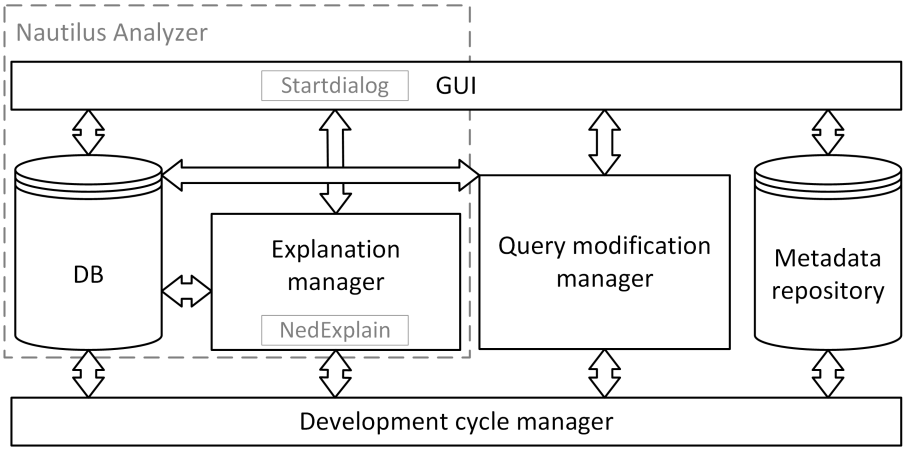


Abbildung 4.1: Nautilus Architektur (vereinfacht nach [10]). In Grau der im Nautilus Analyzer implementierte Teil mit den Stellen, an denen NedExplain eingefügt wird.

enthalten die Komponenten ein Modul, um Vorschläge gewichten zu können. Basis dafür sind Nutzerannotationen und Metadaten aus vorherigen Durchläufen aus dem *Metadata repository*. Dieses ist gleichzeitig Speicher für Daten, die die Provenance-Algorithmen permanent speichern. Deshalb enthält es keine feste Struktur. Der Zugriff läuft über den *Development cycle manager*. Neben der Schnittstelle auf das Metadata repository stellt dieses Modul die Methoden für die Testphase des AFT-Zyklus bereit. Dazu speichert es den globalen Zustand im Workflow.

Die Dateninstanz auf der Nautilus arbeitet ist eine relationale Datenbank (*DB*). Alle Vorschläge und Aussagen über Änderungen beziehen sich nur auf die Daten in der Datenbank, die darin zum Zeitpunkt der Analyse existieren. Eine Überprüfung der Anfrage auf Seiteneffekte, die nicht mit diesem Datensatz sichtbar werden, ist nicht in Nautilus enthalten und geplant, da diese den AFT-Zyklus ergänzen, aber kein Teil davon sind.

Das Userinterface (*GUI*) erlaubt dem Benutzer Nautilus zu bedienen. Der Aufbau und die Möglichkeiten hängen dabei von der Implementierung ab.

Der Nautilus Analyzer enthält das der Analysephase zugeordneten Modul Explaination manager und zusätzlich die Datenbank und eine GUI. NedExplain wird als Why-Not Provenance-Algorithmus dem Explaination manager hinzugefügt. Der Algorithmus braucht zur Berechnung der Erklärungen neben dem Debuggingszenario nur Zugriff auf die Datenbank.

Über einen Startdialog kann die Analyse des Debuggingszenarios konfiguriert werden (Abbildung 4.2). Neben der Auswahl des bzw. der anzuwendenden Why-Not Provenance-Algorithmen können diese auch konfiguriert werden. Für NedExplain wird die Option zum Ausführung auf diesem Debuggingszenario hinzugefügt. NedExplain hat keine Konfigurationsmöglichkeiten für die Berechnung, weshalb ein Button mit Optionen fehlt. Mit dem Start der Berechnung werden von den ausgewählten Algorithmen Erklärungen für das Debuggingszenario berechnet.

Die Ausgabe beschränkt sich auf die Konsole. Eine Ansicht mit einer Baumdarstellung der Anfrage ist für den Why-Not Algorithmus bereits vorhanden. Diese unterstützt aber keine Aggregationen und eine neue oder kombinierte Ansicht ist nicht Teil dieser Arbeit.

Intern erstellt der Startdialog ein Konfigurationsobjekt, dass alle ausgewählten Optionen enthält. Dieses gibt dann Eclipse an einen Thread weiter, der darauf dann die einzelnen Why-Not Provenance-Algorithmen startet. Der Startdialog ist in der Klasse `LaunchTab1` definiert. Hier werden eine Checkbox für NedExplain hinzugefügt und die Änderungs-Listener angepasst. Eine Auswahl der Checkbox durch den Benutzer übertragen die Listener in das Konfigurationsobjekt, das in einem Feld den Status der Checkbox speichert.

Die Ausführung der Algorithmen wird von der Klasse `LaunchConfigDelegate` gesteuert. Diese wird in einem eigenen Thread von Eclipse gestartet und erhält als Parameter das vom Startdialog erstellte Konfigurationsobjekt. Jeder Why-Not Provenance-Algorithmus hat eine eigene Verzweigung, die zu Beginn überprüft, ob dieser Algorithmus zur Ausführung ausgewählt ist. Für NedExplain wird hier eine Verzweigung mit Überprüfung des Status der Checkbox

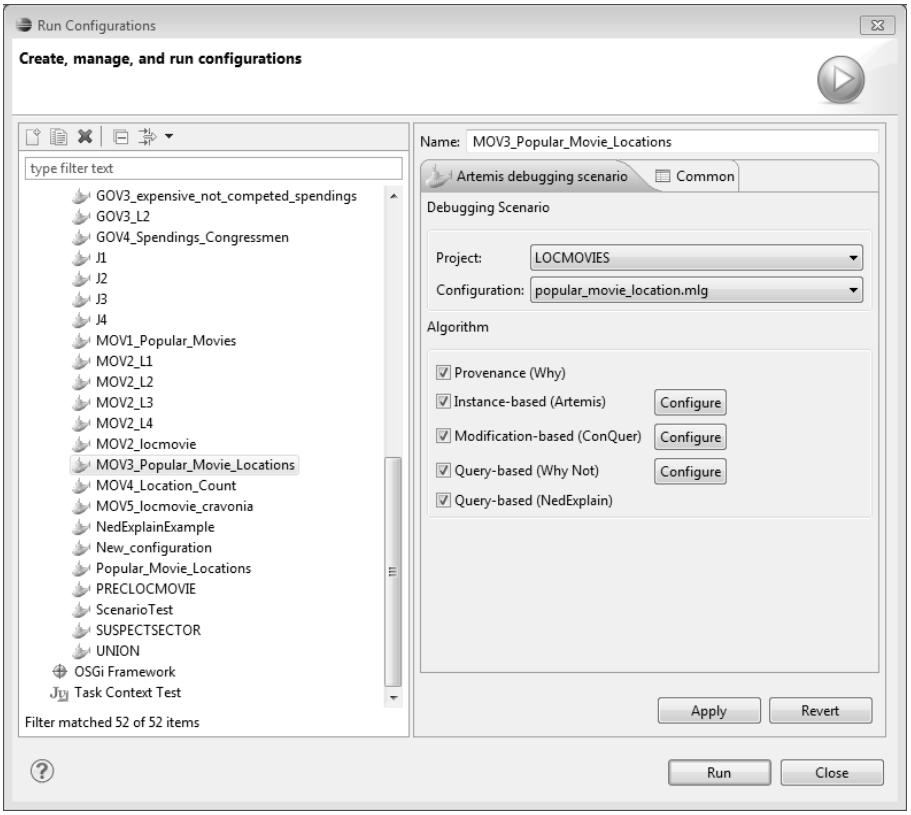


Abbildung 4.2: Angepasster Startdialog mit Option für NedExplain

hinzugefügt. Wenn NedExplain zur Ausführung ausgewählt ist, generiert eine Methode aus den fehlenden Tupeln des Debuggingszenario eine Why-Not Frage für NedExplain. Diese erstellt im Anschluss eine `NedExplainConfiguration` aus den Why-Not Frage, der Verbindung zur Datenbank und der SQL-Anfrage und startet NedExplain damit.

Der Aufruf mit der `NedExplainConfiguration` hat den Vorteil, die Implementierung (Abschnitt 4.2) näher am Pseudocode von Bidoit et al. orientieren zu können [3]. Damit ist NedExplain unabhängig von der Implementierung der fehlenden Tupel in Java in Nautilus. Die Möglichkeiten der fehlenden Tupel wird dabei nicht eingeschränkt, da die Methode zur Erstellung der Why-Not Frage für NedExplain nur die Darstellung verändert. Die Konfiguration bietet dazu ein klares Interface, beispielsweise für die Integration von NedExplain in Nautilus Web.

4.2 Algorithmusimplementierung

Die Implementierung des Algorithmus folgt dem Entwurf von Bidoit et al. (Listing 4.1). Schnittstelle für die Eingabe ist ein Objekt der Klasse `NedExplainConfiguration`, das den Zugang

zur Datenbank, die Why-Not Frage als Text und die SQL-Abfrage enthält. Im Folgenden stelle ich zuerst die verwendeten Datenstrukturen vor und gehe dann die Ausführungsschritte, wie in Abschnitt 3.1, durch.

4.2.1 Datenstrukturen

Die Implementierung der Datenstrukturen von NedExplain basiert auf Hilfsdatenstrukturen. Diese stelle ich zuerst vor in Abschnitt 4.2.1 vor und erkläre dann in Abschnitt 4.2.1 die Datenstrukturen von NedExplain darauf.

Hilfsdatenstrukturen

Zu jeder Datenstruktur erkläre ich die Funktion und die Klassen in denen diese implementiert ist.

SQL-Anfragen sind als Objekt gespeichert um Zusatzinformationen über Anfrage zu speichern und Hilfsmethoden zur Verfügung zu stellen. Anfragen werden in `SQLQuery` als einfacher String gespeichert. Zusätzlich enthält das Objekt eine Liste der verwendeten Relationen und Methoden für den Zugriff auf die Anfrage und die Anfrage ohne Semikolon. Die Klasse `Subquery` erweitert das Interface von `SQLQuery` und speichert die Anfrage in einzelnen Klauseln ab. Jede Klausel besteht aus einer Liste der Elemente, beispielsweise enthält die Liste für die Gruppierung die Spalten, nach denen gruppiert wird, als Listeneinträge. Die auf Klauseln aufgeteilte Implementierung erlaubt es die Unterabfrage eines Operators im Anfragebaum aus den Unterabfragen des bzw. der Kinder zu erstellen. Zu der Kopie des bzw. der Kinder wird nur der eigene Operator hinzugefügt. Selektionen beispielsweise nehmen die Anfrage des vorherigen Operators und fügen die Selektionsbedingung den Bedingungen der Anfrage hinzu. Joins führen die Anfragen der Kinder zusammen, indem die Listen zusammengeführt werden, und fügen die Joinbedingung hinzu. Die SQL-Anfrage wird aus den Listen zusammengesetzt, wenn sie benötigt wird. Die Aufteilung in Klauseln ermöglicht eine vereinfachte Generierung der Anfragen zur Berechnung der Nachfolger und blockierten kompatiblen Tupel. Die Methoden dafür sind auch in `Subquery` eingefügt.

Tupelmengen werden in der `Tuples` Klasse gespeichert. Diese besteht aus einer SQL-Anfrage entweder als String oder `Subquery` und einem Feld, das anzeigt, ob die Tupelmenge leer ist. Um dies zu setzen muss die Anfrage auf die Datenbank ausgeführt werden. Die Tupel selbst sind nicht in Java gespeichert. Diese Implementierung erlaubt die häufige Prüfung, ob eine Tupelmenge leer ist, durchzuführen ohne die Anfrage dafür erneut auszuführen. Damit sind mehr Anfragen nötig, als bei einer Implementierung, die Tupelmengen wie beispielsweise Zwischenergebnisse in Java speichert. Dafür skaliert die Implementierung besser mit der Menge der kompatiblen Tupel, da die Datenbank auf den Umgang mit Tupeln optimiert ist.

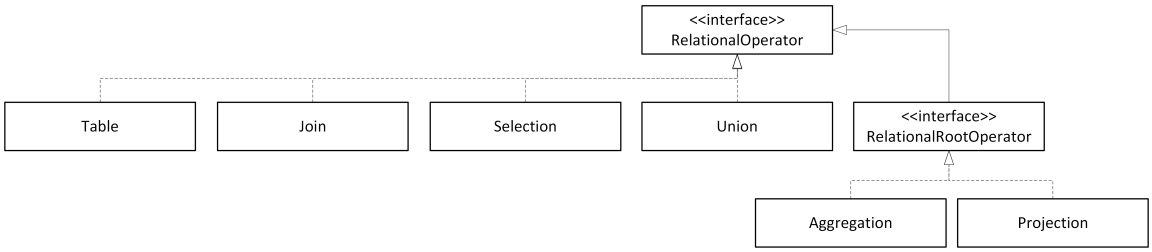


Abbildung 4.3: UML-Diagramm der Klassen für den Anfragebaum

Bedingungen werden als Objekte gespeichert, da im Verlauf der Berechnung Aliase von Operanden umbenannt werden müssen. Die dafür benötigten Methoden in den Klassen implementiert. Die Objekte, die Bedingungen verwenden, benutzen Listen mit Elementen vom Typ `Predicate`, die mit „und“ verknüpft sind. `Predicate` ist ein Interface, dass von `Condition` für binäre Bedingungen und `ORCondition` für, mit „oder“ verknüpfte, `Conditons` implementiert ist. Diese Implementierung erlaubt eine Kombination von „und“- und „oder“-Verknüpfungen in Bedingungen. Da die Verknüpfung mit „und“ wesentlich häufiger in Anfragen vorkommt, ist diese Standard.

Why-Not Frage wird als Objekt der Klasse `WhyNotQuestion` mit einer Menge an Bedingungstupeln (`CTuple`) gespeichert, die jeweils ein Variablentupel (`VTuple`) haben. Jede dieser Klassen erlaubt eine Initialisierung über die Darstellung des Objekts als String. Das Why-Not Frage Objekt teilt die Why-Not Frage aus der Eingabe in Blöcke auf, die jeweils einem Bedingungstupel entsprechen. Das Bedingungstupel teilt den String in die Bedingung und den Teil des Variablentupels auf, das dann damit initialisiert wird. Umbenennungen in der Vorbereitung für die Berechnung von NedExplain werden über ein Interface gesetzt und dann auf alle passenden Felder angewandt. Alle Bedingungen zu einem Alias, beispielsweise für eine Relation, können zu einer SQL-Bedingung zusammengestellt werden zur Berechnung der kompatiblen Tupel.

Baumdarstellung der SQL-Anfrage ist die größte Datenstruktur. Sie bildet die Anfrage in einzelne Objekte ab, die jeweils einer Unterabfrage entsprechen. Zusätzlich bringt jedes Objekt im Baum die Methoden mit, die später für die Berechnung von tab_Q benötigt werden. Ausgenommen sind die Felder Abstand zur Wurzel und die Reihenfolge, die in Listen gespeichert werden, wie in Abschnitt 4.2.2 erklärt.

Alle Knoten des Baumes sind von der abstrakten Klasse `RelationalOperator` abgeleitet (Abbildung 4.3). Diese verwaltet die Doppelverkettung der Einträge und stellt Felder für den Zugriff auf u.A. Ausgabe, Eltern, Kinder, Unteranfrage und kompatible Tupel. `RelationalOperator` definiert damit die Schnittstelle, mit der der Algorithmus später auf ein Feld in tab_Q zugreift. Das Feld Eingabe in der Tabelle wird nicht implementiert, da dieses den Ausgaben der Kinder entspricht.

Die Operatoren für Vereinigung, Projektion und Aggregation können die Wurzel eines Anfragebaumes sein und erhalten über die abstrakte Klasse `RelationalRootOperator`

zusätzliche Felder. Jede Projektion bzw. Aggregation speichert für sich und alle Unterabfragen darunter das dazugehörige Variablentupel der Why-Not Frage, da das Rückgängigmachen der Umbenennungen spezifisch für jede Projektion bzw. Aggregation um Unterabfragen ist. Davon hängt auch die Berechnung der kompatiblen Tupel in den Relationen ab, die von hier aus angestoßen wird.

NedExplain-Datenstrukturen

Die Datenstrukturen für NedExplain bauen auf den Hilfdatenstrukturen und den in Java standardmäßig vorhandenen Datenstrukturen auf. Im Folgenden gehe ich die einzelnen Datenstrukturen durch und erkläre den Aufbau.

Why-Not Frage wird in der gleichnamigen Datenstruktur gespeichert. Die Werte dafür stammen aus dem Prädikat $\mathcal{P}$. Dieses ist die Darstellung der Why-Not Frage als String aus der `NedExplainConfiguration`. Die Bedingungen im Bedingungstupel sind in `Predicates` gespeichert.

$\mathbf{tab_Q}$ verwendet die Klassen der Baumdarstellung der SQL-Anfrage um auf die Werte Eingabe, Ausgabe, kompatible Tupel, Elternoperator und Art zuzugreifen. Die Reihenfolge wird ein einer Liste gespeichert, die die einzelnen `RelationalOperatoren` in der Reihenfolgen enthält, in der sie dann abgearbeitet werden sollen. Der Abstand zur Wurzel ist in einer zweiten Liste gespeichert. Der Index eines Operators in der Reihenfolgeliste ist auch der Index des Abstandes dieses Operators in der zweiten Liste. Dieser Zugriff über den Index bietet sich an, da NedExplain bei der Berechnung über den Index iteriert.

PickyMan ist eine Map, die für den Schlüssel `RelationalOperator` eine Liste von blockierten `Tuples` als Wert enthält. Die Liste ist notwendig, da eine Join-Unterabfrage kompatible Tupel aus beiden Eingaben blockieren kann.

NonPickyMan und EmptyOutputMan sind Mengen an Unterabfragen, also Mengen an `RelationalOperator`.

$\mathbf{inDir^{tc}}$ wird als Menge von `Table` gespeichert.

4.2.2 Algorithmus

Kern der Implementierung ist der NedExplain Algorithmus in der Klasse `NedExplain` (Listing 4.1). Sie bring statische Methoden mit zur Ausführung von NedExplain auf der `NedExplainConfiguration` aus Datenbankverbindung, SQL-Anfrage und Why-Not Frage. Die Methode *calcProvenance()* enthält die Initialisierung und den Algorithmus selbst, die Hilfsmethoden *checkEarlyTermination()*, *detailedAnswer()* und *findSuccessors()* entsprechen dem Entwurf von Bidoit et al. [3]. Im Vergleich zum Entwurf erwarten die Hilfsmethoden mehr Parameter, da die globalen Strukturen von NedExplain in Java nicht global sichtbar sind.

Listing 4.1 Interface der NedExplain Klasse

```
package de.uni_stuttgart.provenance.nedexplain.algorithm;

public class NedExplain {
      public static ArrayList<HashMap<String, Figure>>
          calcProvenance(NedExplainConfiguration config);
      private static void unrename(...);
      private static void compatibleFinder(...);
      private static boolean checkEarlyTermination(RelationalOperator m, ...);
      private static ArrayList<HashMap<String, Figure>> detailedAnswer(...);
      private static Tuples findSuccessors(RelationalOperator m, ...) throws SQLException;
      private static void calculateVisibilityFrontier(...);
}
```

Die Vorbereitung der Ausführung in *calcProvenance()* verwendet die Methoden *compatibleFinder()*, *unrename()* und *calculateVisibilityFrontier()*. Die Methode *compatibleFinder()* sucht in den Relationen nach kompatiblen Tupeln, *unrename()* macht eventuelle Umbenennungen von Feldern rückgängig und teilt Join-Bedingungen auf die Join-Partner auf und *calculateVisibilityFrontier()* berechnet für eine Aggregation die dazugehörige Menge an Unterabfragen, die die *visibility frontier* V bilden.

Die Ausführung des Algorithmus beginnt auch in der Implementierung in Java mit der Initialisierung der globalen Strukturen. Basis für die Ausführung ist der Parameter mit dem `NedExplainConfiguration` Objekt, das die Why-Not Frage, die Verbindung zur Datenbank und die Anfrage enthält.

NedExplain erstellt eine leere Why-Not Frage von der Klasse `WhyNotQuestion` und initialisiert diese mit der Why-Not Frage aus der `NedExplainConfiguration`. Das erste Variablentupel wird als aktuell fehlendes Tupel vermerkt, für das in diesem Durchlauf Erklärungen gesucht werden.

Mit der Anfrage aus der Eingabekonfiguration erstellt NedExplain mit dem `Parser` eine Baumstruktur aus `RelationalOperatoren`. Diese benötigt der Algorithmus, da er darauf basierend die Suche nach kompatiblen Tupeln und das Rückgängigmachen der Umbenennungen ausführt. Als SQL-Parser verwendet NedExplain die Bibliothek ZQL[1]. Diese erlaubt dann Zugriff auf die einzelnen Elemente der Anfrage. Daraus erstellt der Algorithmus eigene Objekte von den entsprechenden Klassen `Projection`, `Join`, `Aggregation`, `Selection`, `Table` und `Union`. Anschließend verbindet NedExplain diese zu einem Anfragebaum und speichert eine Referenz auf die Wurzel ab. Je nachdem, ob eine Aggregation in der Anfrage vorhanden ist, verwendet der Algorithmus unterschiedliche Methoden zur Generierung der Baumstruktur.

Für jede Aggregation bzw. Projektion im Baum macht NedExplain die Umbenennungen in einer Kopie des aktuellen Variablentupel rückgängig. Diese speichert er bei der Projektion bzw.

[1] http://zql.sourceforge.net

Aggregation. Mehrere Projektionen bzw. Aggregationen sind möglich, wenn sie von Vereinigungen zusammengeführt werden. Die vereinigten Unterabfragen können unterschiedliche Umbenennungen enthalten.

Die kompatiblen Tupel sucht NedExplain auch für jede Unterabfrage ab Projektion bzw. Aggregation separat, da die Suche von dem umbenannten Variablentupel abhängt. Die gefundenen kompatiblen Tupel speichert der Algorithmus als `Tuples` an der Relation, aus der die kompatiblen Tupel stammen. Alle Relationen, die keine kompatible Tupel enthalten können, fügt NedExplain inDir^{tc} hinzu. Diese ist auch spezifisch für die Projektion bzw. Aggregation und speichert der Algorithmus an der Projektion bzw. Aggregation.

Als nächstes initialisiert NedExplain tab_Q. Eine Methode *createBottomUpOrder()* aus `NedExplainUtil` füllt die beiden Listen mit Referenzen auf den Anfragebaum. Die Wurzel mit dem Abstand 0 steht am Ende der Listen. Die anderen globalen Strukturen EmptyOutputMan, Non-PickyMan und PickyMan initialisiert der Algorithmus leer.

Die Berechnung der Erklärungen orientiert sich an den Pseudocode-Blöcken aus Bidoit et al. [3]. Die Implementierung von *checkEarlyTermination()* übernimmt den Code ohne Änderungen. Die Hauptfunktion und die Nachfolgerberechnung mit *FindSuccessors()* sind abgeändert implementiert, wie im Abschnitt 4.3 erklärt wird.

4.3 Algorithmusänderungen

Im Folgenden stelle ich Änderungen am Algorithmus vor, die über die reine Umsetzung in Java hinausgehen. Die Begründungen sind unterschiedlich und jeweils bei der Änderung ausgeführt.

4.3.1 Nachfolgerberechnung

Im Entwurf findet die Nachfolgerberechnung über die Funktion $lineage()$ statt. Diese berechnet für ein gegebenes Tupel die Herkunft. Liegt diese in den direkt und indirekt kompatiblen Tupeln (Dir^{tc} und $InDir^{tc}$), ist das gefundene Tupel ein kompatibles Tupel. Es wird mit seiner Herkunft zu den kompatiblen Tupeln hinzugefügt. In der Implementierung des Why-Not Algorithmus [5] und in der Implementierung von Bidoit et al. [3] wird auf Trio [15] zurückgegriffen. Dieses Programm kann für eine PostgreSQL Datenbank die Herkunft der Tupel berechnen. Es wird über Python angesprochen.

Bei der Implementierung von NedExplain verzichte ich auf Trio. Damit benötigt der Algorithmus zur Ausführung nur die Java Runtime. Als Datenbank genügt dem Algorithmus eine SQL-kompatible Datenbank, während die Implementierung von Why-Not eine Trio-kompatible Datenbank voraussetzt. Die Berechnung der Nachfolger übernimmt die Datenbank. NedExplain generiert nur die SQL-Anfragen für die Nachfolger, die dann von der Datenbank ausgeführt

werden. Dies erlaubt eine Berechnung der Nachfolger ohne die Betrachtung der einzelnen Tupel und ohne eine externe Bibliothek. Als Nebeneffekt skaliert dieser Ansatz gut mit großen Mengen an kompatiblen Tupeln, weil die Berechnung auf Datenbankfunktionen abgebildet wird, die vom Datenbankmanagementsystem optimiert zur Verfügung gestellt werden.

Im Folgenden stelle ich für alle Operatoren geltende Regeln vor. Im Anschluss gehe ich die einzelnen Operatoren ein und erkläre jeweils, wie mit den Möglichkeiten von SQL die validen Nachfolger der kompatiblen Tupel der Eingabe gefunden werden.

Ein Operator kann in seiner Ausgabe entweder nur indirekt kompatible Tupel oder eine Kombination aus kompatiblen und nicht kompatiblen Tupeln enthalten. Das beginnt bei den Relationen, die entweder in den indirekt kompatiblen Tupeln liegen oder zu den kompatiblen Tupeln beitragen können. Jeder unäre Operator danach verändert die Zuordnung der Tupel nicht. Er kann aus der Sicht von NedExplain nur eventuell vorhandene kompatible Tupel aus der Eingabe entfernen. Liegt die Eingabe in den inkompatiblen Tupeln, so ist auch die Ausgabe in den inkompatiblen Tupeln. Den binären Operator Join gilt das nur, wenn beide Eingaben entweder aus indirekt kompatiblen Tupeln oder aus der Kombination von kompatiblen und indirekt kompatiblen Tupeln bestehen. Bei einer Eingabe aus den indirekt kompatiblen Tupeln und einer Eingabe aus kompatiblen bzw. nicht kompatiblen Tupeln erstellt der Join eine Ausgabe aus kompatiblen bzw. nicht kompatiblen Tupeln. Jedes Tupel der Ausgabe liegt nicht mehr ganz in den indirekt kompatiblen Tupeln, weshalb die Ausgabe keine mehr enthält.

Für die einzelnen Operatoren betrachte ich deshalb nur für den Join den Fall, dass in einer Eingabe indirekt kompatible Tupel sind. Für das Ziel das Finden von Nachfolgern der kompatiblen Tupel genügt für die anderen Operatoren die Betrachtung einer Eingabe, die kompatible Tupel und evtl. indirekt kompatible Tupel enthält.

Selektion Ein kompatibles Tupel in der Ausgabe einer Selektion hat als Herkunft ein kompatibles Tupel in der Eingabe, auf das die Bedingung der Selektion angewandt wird. Da die Bedingung der Selektion die Tupel nicht verändert, können die Nachfolger der kompatiblen Tupel der Eingabe in der Ausgabe identifiziert werden, indem man die Selektion auf die kompatiblen Tupel anwendet. Die kompatiblen Tupel liegen in der Implementierung bereits als SQL-Abfrage vor, die zum Finden der Nachfolger um die Selektionsbedingung erweitert die kompatiblen Tupel in der Ausgabe ergibt. Der Schnitt der kompatiblen Tupel aus der Eingabe mit den kompatiblen Tupeln nach der Anwendung des Operators entspricht den von der Selektion blockierten kompatiblen Tupel.

Projektion Eine Projektion verändert nur die Anzahl der angezeigten Felder und hat damit keinen Einfluss auf die kompatiblen Tupel. Alle kompatiblen Tupel der Eingabe finden sich auch in der Ausgabe der Projektion.

Vereinigung Vereinigungen entfernen, wie Projektionen auch, keine Tupel. Damit sind die Nachfolger kompatibler Tupel in der Eingabe die gleichen Tupel in der Ausgabe.

Join Join hat zwei Eingaben, die mit der Joinbedingung verknüpft werden. Wenn beide Eingaben kompatible Tupel enthalten, sind valide Nachfolger nur Tupel, die aus dem

Verknüpfen von zwei kompatiblen Tupel entstanden sind. Diese können mit einem Join der kompatiblen Tupel der Eingaben berechnet werden. Alle anderen Tupel in der Ausgabe enthalten mindestens ein nicht kompatibles Tupel in der Herkunft und sind deshalb keine validen Nachfolger. NedExplain führt die Abfragen der kompatiblen Tupel der Eingabe in eine SQL-Anfrage zusammen und ergänzt die Joinbedingung. Wenn eine Eingabe kompatible Tupel enthält und die Andere aus indirekt kompatiblen Tupeln besteht, sind alle Tupel in der Ausgabe valide Nachfolger, die aus einem kompatiblen Tupel verknüpft mit einem indirekt kompatiblen Tupeln bestehen. Hier führt der Algorithmus die Ausgabe mit den indirekt kompatiblen Tupeln mit den kompatiblen Tupeln zusammen und fügt die Joinbedingung dazu. Wenn eine Eingabe kompatible Tupel enthält und die Andere keine kompatiblen Tupel mitbringt und nicht in $inDir^{tc}$ liegt, enthält die Ausgabe keine kompatiblen Tupel. NedExplain macht in diesem Fall keine weiteren Berechnungen für die Nachfolger. Für jedes Szenario lassen sich die Indizes der blockierten kompatiblen Tupel über den Schnitt zwischen den kompatiblen Tupeln der Eingabe und ihren Nachfolgern berechnen.

Aggregation Aggregationen lassen sich aufteilen in Aggregationen mit und ohne Gruppierung. Bei Aggregationen ohne Gruppierung, wie beispielsweise `COUNT`$(*)$, werden alle Tupel der Eingabe auf ein Ausgabetupel aggregiert. Wenn die Eingabe mindestens ein Tupel enthält, das nicht zu den kompatiblen Tupeln gehört, dann ist auch das Ausgabetupel kein kompatibles Tupel. Bei Aggregationen mit Gruppierung muss für jedes Tupel der Ausgabe überprüft werden, ob alle Tupel, die hier zusammengeführt werden, kompatible Tupel sind. Für beide Szenarien generiert NedExplain die SQL-Anfragen zur Suche der kompatiblen Tupel in der Ausgabe und zur Berechnung der blockierten kompatiblen Tupel.

4.3.2 Prüfung für PickyMan in FindSuccessors

In der Methode $FindSuccessors(m)$ gibt es die Möglichkeit, dass der Algorithmus eine Unterabfrage m, die kompatible Tupel entfernt, nicht zum PickyMan hinzufügt. Zuerst ordne ich die Stelle im Pseudocode in den Zusammenhang der Methode ein. Danach erkläre ich das problematische Szenario und stelle im Anschluss die Lösung dazu vor.

Algorithmus 4.1 zeigt einen Ausschnitt aus der $FindSuccessors(m)$-Methode von NedExplain. In diesem Abschnitt fügt NedExplain gefundene blockierte kompatible Tupel mit ihrer Unterabfrage m zum PickyMan hinzu. Diese verwendet der Algorithmus bei der Generierung der Erklärungen. Die Bedingung unterschiedet dabei zwischen zwei Szenarien, je nachdem ob die visibility frontier V unter m im Baum steht oder nicht. NedExplain beachtet dies, um die Bedingungen der Why-Not Frage auf die Aggregation zu überprüfen, sobald diese angewandt werden kann.

Im Entwurf des Pseudocode wird die Möglichkeit nicht beachtet, dass in NedExplain blockierte Tupel findet und diese nicht zum PickyMan hinzufügt, weil er die Bedingung zu falsch evaluiert.

Algorithmus 4.1 Prüfung für das Hinzufügen zum PickyMan in FindSuccessors() [3]. Der problematische Teil der Bedingung ist unterstrichen.

9: **if** $(Blocked \neq \emptyset \wedge \underline{V\ not\ subquery\ of\ m}) \vee (V\ subquery\ of\ m \wedge \alpha_{G,F}(m.Input) \models t_c.cond_\alpha \wedge \alpha_{G,F}(m.Output) \not\models t_c.cond_\alpha)$ **then**
10: PickyMan ← PickyMan ∪ {m,Blocked}
11: **end if**

Ein Beispiel wäre eine Anfrage mit Aggregation, bei der die aktuelle Unterabfrage m eine Unterabfrage von V ist und die Bedingung auf die Aggregation sowohl für Ein- als auch für die Ausgabe erfüllt ist. In diesem Fall sollte die Unterabfrage zum PickyMan hinzugefügt werden, da es blockierte Tupel gibt. Dies geschieht nicht, da die Bedingung zu falsch evaluiert wird. Die erste Hälfte ergibt falsch, da V eine Unterabfrage von m ist. Sonst könnte man die Aggregation nicht anwenden. Die zweite Hälfte ergibt auch falsch, da die Ausgabe in diesem Szenario auch die Bedingung auf die Aggregation erfüllt. Damit wird die Unterabfrage m nicht zum PickyMan hinzugefügt und fehlt in den Erklärungen. Bei einem passenden Datensatz fallen in diesem Schritt alle kompatiblen Tupel weg und NedExplain findet keine Erklärung für die fehlenden Tupel.

Das Entfernen des unterstrichenen Teils aus Algorithmus 4.1 aus der Bedingung vermeidet dieses Szenario (Algorithmus 4.2). Damit wird die Bedingung dem Ansatz dahinter gerecht, der

Algorithmus 4.2 Angepasste Bedingung für Algorithmus 4.1

9: **if** $(Blocked \neq \emptyset) \vee (V\ subquery\ of\ m \wedge \alpha_{G,F}(m.Input) \models t_c.cond_\alpha \wedge \alpha_{G,F}(m.Output) \not\models t_c.cond_\alpha)$ **then**
10: PickyMan ← PickyMan ∪ {m,Blocked}
11: **end if**

das Finden von filternden Operatoren über beide Varianten erlaubt. Damit findet NedExplain sowohl Operatoren, die kompatible Tupel über ihre Funktion entfernen, als auch Operatoren, bei denen die Funktion die Tupel entfernen, die zur Erfüllung der Bedingung auf die Aggregation benötigt werden. Eine Implementierung mit lazy Evaluation der Bedingung genügt, da die Generierung der Antwort nicht mehr nachvollzieht, weshalb die Tupel zu der Unterabfrage im PickyMan sind.

4.3.3 Kompatible Tupel in der Ausgabe

Es kann vorkommen, dass der Benutzer einen Fehler in der Definition der Why-Not Frage macht oder sich der Datensatz ändert seit der Definition der Why-Not Frage. Dann ist es, dass Tupel in der Ausgabe die Bedingung in der Why-Not Frage erfüllen. Das ist ein Problem, da NedExplain in diesem Fall keine Erklärungen ausgibt. Der Algorithmus erreicht nie

$detailedAnswer()$, da $checkEarlyTermination()$ immer kompatible Tupel zur Nachverfolgung findet. Im Pseudocode ist vorgesehen, dass im Fall, dass alle Felder von tab_Q durchlaufen werden, $null$ zurückgegeben wird. Zusätzlich stürzt die Berechnung davor schon bei der Berechnung des letzten Feldes ab. Dieses entspricht der Wurzel in der Baumdarstellung und hat keinen Elterneintrag p. Der Zugriff auf die Felder $p.Eingabe$ in Zeile 8 bzw. $p.KompatibleTupel$ in Zeile 16 von Algorithmus 3.1 erzeugt einen Fehler.

Eine Möglichkeit, mit diesem Fall umzugehen, ist zu Beginn eine Prüfung der Ausgabe auf kompatible Tupel zu machen. Das würde dem Benutzer schnell Feedback geben, dass in dieser Eingabe ein Problem ist.

Die Alternative ist, den Algorithmus auszuführen und die Probleme beim Zugriff auf den Elterneintrag abzufangen. In der Implementierung wird dieser Ansatz gewählt und die Funktion $detailedAnswer()$ angepasst, um dem Benutzer neben den filternden Operatoren auch darauf hinzuweisen, dass die Ausgabe kompatible Tupel enthält. Der Zugriff in Zeile 8 findet in der Implementierung nicht statt, da die Eingabe eines Operators immer die Ausgabe seiner Kinder ist. Listing 4.2 zeigt den geänderten Abschnitt für das Hinzufügen der gefundenen Nachfolger zum Elterneintrag. Die Zuweisung auf $p.KompatibleTupel$ fällt für den letzten Operator weg, da p $null$ ist (Zeile 17). Stattdessen speichert NedExplain die kompatiblen Tupel der Ausgabe in $compatiblesLeft$ (Zeile 20-22). Die Methode $detailedAnswer()$ beachtet die $compatiblesLeft$

Listing 4.2 Abgeänderte Implementierung für den letzten Eintrag in tab_Q

```
15 if (!(m instanceof Table)) {
16       // check if last item in tree produces compatibles
17       if (p != null)
18             p.addCompatibleTuples(findSuccessors(m, stmt, pickyMan, non_pickyMan, tabQ,
                   v, aggregation), m);
19       else {
20             compatiblesLeft = findSuccessors(m, stmt, pickyMan, non_pickyMan, tabQ, v,
                   aggregation);
21             detailedAnswer(stmt, compatiblesLeft, pickyMan);
22             return;
23       }
24 }
```

bei der Generierung der Erklärung beachtet. Gleichzeitig wird die Berechnung damit beendet und der $null$-Fall aus dem Pseudocode vermieden, bei dem der Benutzer keine Erklärung erhält.

4.4 Einschränkungen

Im Folgenden stelle ich Implementierungsteile vor, die anders als ursprünglich geplant, in der Implementierung vorhanden sind.

Tests sind nur für sehr kleine Abschnitte des Codes implementiert. Vorhanden sind Tests für den Parser und die Generierung des Baumes. Dazu wird das Übertragen der Daten aus der schriftlichen Darstellung der Why-Not Frage in das dazugehörige Objekt getestet.

Tests für eine vollständige Ausführung des Algorithmus sind nicht enthalten, da dieser von der Datenbank und dem Debuggingszenario abhängt. Die Datenbank ist in den Start des Plug-ins eingebunden und damit steht sie für Tests nicht allein zur Verfügung. Damit beschränken sich die Tests auf einen kleinen Teil der Datenstrukturen.

Generierung der Baumstruktur wird mit einem eigenen Parser durchgeführt, der die Daten aus den SQL-Parser in die Baumobjekte überträgt und im Anschluss eine Baumstruktur generiert. Die vorgesehene Erweiterung des Why-Not Parsers, der eine sehr ähnliche Aufgabe für den Why-Not Algorithmus übernimmt, ist nicht implementiert. Die benötigten Änderungen sind eine Unterstützung von Aggregationen und eine Generierung eines zweiten Baumes, dass Why-Not bzw. NedExplain jeweils auf einer eigenen Baumstruktur arbeiten können. Diese unterscheidet sich bei Aggregationen in der Eingabe.

Diese Erweiterung ist aufgrund die Architektur des Why-Not Parsers nicht möglich. Dazu lassen sich die Baumklassen nicht gut an einen anderen Algorithmus anpassen. Der neu geschriebene Parser für NedExplain orientiert sich an der Funktionalität des Why-Not Parsers und verwendet eine anderen Aufbau, der die gefundenen Probleme vermeidet. Zusätzlich erlaubt die Neuimplementierung eine Unterstützung von Vereinigungen im Baum.

SQL-Anfragen dürfen nicht alle Möglichkeiten nutzen, die syntaktisch korrekt sind. Anforderungen stellen der SQL-Parser, NedExplain oder die Why-Not Provenance-Algorithmen.

Die Parserbibliothek ZQL hat Probleme mit dem Schlüsselwort „AS“ bei den ausgewählten Spalten der Projektion bzw. Aggregation. Aliase müssen hier mit einem Leerzeichen hinter die Spalte geschrieben werden, auf die sie sich beziehen. Das Schlüsselwort „JOIN“ ist für das Zusammenführen von Tabellen in der FROM-Klausel der SQL-Anfrage nicht erlaubt. Das wird auch NedExplain selbst nicht unterstützt. Joins sind über Bedingungen in der WHERE-Klausel der SQL-Anfrage möglich.

Allgemein dürfen Bedingungen in der WHERE-Klausel nur mit „AND“ verknüpft werden für die Implementierung für NedExplain. Die Unterstützung von Mengenoperatoren beschränkt sich auf Vereinigungen. Die Operatoren „INTERSECT“ und „MINUS“ werden nicht korrekt erkannt.

Darüber hinaus benötigen alle Why-Not Provenance-Algorithmen eine absolute Benennung der Spalten in der Anfrage und Relationen müssen mindestens 3 Zeichen im Namen haben.

4.5 Zusammenfassung Implementierung

Die Implementierung von NedExplain enthält alle Fähigkeiten, die der Entwurf von Bidoit et al. vorsieht [3]. Die vorgegebenen Operatoren Selektion, Projektion, Join, Vereinigung und Aggregation können in SQL-Anfragen enthalten sein. Die Abhängigkeiten der Implementierung sind auf die Bibliothek ZQL reduziert und Abhängigkeiten zu Klassen von Eclipse sind nicht enthalten. Das erlaubt den Algorithmus für einen Einsatz ohne Eclipse anzupassen. Im Gegenzug ist der Parser selbst implementiert und es ist keine graphischen Ausgabe der Ergebnisse enthalten.

5 Evaluation

In der Evaluation vergleiche ich die Ausführung von NedExplain mit anderen Why-Not Provenance-Algorithmen in Nautilus. Implementiert sind neben NedExplain die Algorithmen Artemis, Why-Not und ConQueR. Naheliegend ist der Vergleich mit dem Why-Not Algorithmus, da dieser, wie NedExplain, Anfrage-basierte Erklärungen berechnet. Der qualitative Vergleich der Erklärungen der beiden Algorithmen ist in Bidoit et al. zu finden [3]. Ich werde mich hier auf Aspekte der Implementierung beschränken: Auf Basis einer Auswahl von Use cases (Abschnitt 5.1) vergleiche ich die Laufzeiten der verschiedene Why-Not Algorithmen in Abschnitt 5.2. Im Anschluss untersuche ich in Abschnitt 5.3 die Performance von NedExplain in einem Use case mit unterschiedlich großen Datensätzen.

Nautilus ist in einer virtuellen Maschine in VirtualBox[1] installiert. Der Host ist ein Windows 8.1 Laptop mit 1,9 GHz Intel i5 mit zwei Kernen und vier logischen Kernen. Die virtuelle Maschine hat Zugriff auf einen logischen Kern und 4GB RAM. Darin läuft Nautilus auf Windows 7 und der JRE 1.7. Die Datenbanken sind in PostgreSQL 9.2 gespeichert, auf das auch Trio zugreift.

5.1 Use cases

Die Datensätze für die Use cases sind die Datenbanken *crime*, *imdb* und *gov*. Sie sind, wie die Use cases, Teil der Evaluation der ersten Implementierung von NedExplain [3]. Die *crime* Datenbank stammt aus den Tests für den Why-Not Algorithmus. Sie enthält neben Verbrechen Tabellen zu Zeugen und Tätern. Sie besteht aus 3217 Tupeln in 4 Relationen und ist insgesamt 75 KB groß. Die *imdb* Datenbank basiert auf den Daten, die vom Filmportal IMDB[2] zur Verfügung gestellt werden. Sie enthält diverse Informationen über die Filme und Drehorte. Die Informationen sind in 3988 Tupeln in 5 Relationen gespeichert, die zusammen 318 KB groß sind. Der größte Datensatz ist die *gov* Datenbank, die aus den Veröffentlichungen der US Behörden zusammenstellt ist. Sie enthält Informationen über die Abgeordneten und Ausgaben der USA. Der Datensatz ist auf 7 Relationen mit zusammen 15997 Tupel verteilt und benötigt 1,17 MB.

Zu jeder Datenbank gehören eine Auswahl von Use cases (Tabelle 5.2). Sie bestehen jeweils aus einer Why-Not Frage und einer SQL-Anfrage, auf die sich die Frage bezieht. Die SQL-Anfragen

[1] https://www.virtualbox.org
[2] http://www.imdb.com

sind in Tabelle 5.1 aufgelistet. Zu jeder Anfrage ist kurz beschrieben, was diese aus der Datenbank abfragt. Die einzelnen Tabellennamen sind abgekürzt und bei mehrfacher Benutzung einer Relation sind die Instanzen nummeriert. Die Anfragen decken dabei die unterschiedlichen Möglichkeiten der Why-Not Provenance-Algorithmen und SQL ab. Q3 und Q4 enthalten einen Self-Join. Q8 und Q9 sind Aggregationen und auf denen der Why-Not Algorithmus keine Erklärungen generieren kann. Q12 bringt eine Vereinigung von zwei Unterabfragen mit. In den Use cases werden sowohl konstante als auch komplexere Bedingungen auf die Felder der Ausgabe gesetzt. Nicht erwähnte Felder dürfen beliebige Werte in den fehlenden Tupeln enthalten.

Q1	Die Anfrage wählt die Personen zu den Verbrechenstypen aus, bei denen sie von Zeugen gesehen wurden.
	$\pi_{P.name,C.type}(C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P)$
Q2	Die Anfrage sucht die Personen zu den Verbrechenstypen aus den Sektoren über 99 heraus, bei denen sie von Zeugen gesehen wurden.
	$\pi_{P.name,C.type}((\sigma_{C.sector>99}(C)) \bowtie_{sector} W \bowtie_{witnessname} S \bowtie_{hair,clothes} P)$
Q3	Die Anfrage wählt alle Zeugen aus Sektoren aus, in denen Beihilfe geleistet wurde.
	$\pi_{W.name,C2.type}(W \bowtie_{sector} C2 \bowtie_{sector} (\sigma_{C.type=Aiding}(C)))$
Q4	Die Anfrage listet alle Personen mit Anfangsbuchstabe A, deren Haarfarbe mehrmals in der Datenbank vorkommt.
	$\pi_{P2.name}(\sigma_{P1.name \neq P2.name}(P2 \bowtie_{hair} (\sigma_{P1.name<B}(P1))))$
Q5	Die Anfrage zeigt die Filmtitel und Drehorte für Filme, die nach 2005 gedreht wurden und ein Rating von mindestens 8 haben.
	$\pi_{M.moviename,L.locationid}(L \bowtie_{movieid} ((\sigma_{M.year>2005}(M)) \bowtie_{moviename} (\sigma_{R.rating \geq 8}(R))))$
Q6	Die Anfrage listet die Abgeordneten der Republikaner, die nach 1970 geboren sind.
	$\pi_{C.lastname,C.firstname}((\sigma_{AA.party=Republican}(AA)) \bowtie_{id} (\sigma_{Co.Byear>1970}(Co)))$
Q7	Die Anfrage listet den Namen der Sponsoren, seine Identifikationsnummer und die ausgegebene Summe für Sponsoren der Republikaner aus dem Senat auf.
	$\pi_{S.sponsorid,S.sponsorln,E.camount}(E \bowtie_{earmarkId} (\sigma_{ES.sponsorsubstagename= Senate\ committee}(ES)) \bowtie_{id} (\sigma_{SPO.sponsorparty=Republican}(SPO)))$
Q8	Die Anfrage zählt wie oft eine Person Verbrechen begangen hat.
	$\alpha_{\{P.name\}\{count(C.type) \rightarrow ct\}}(\sigma_{sector>60}(C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P))$
Q9	Die Anfrage gibt zu dem Sponsor die ausgegebene Summe aus. Es werden nur Ausgaben der Republikaner aus dem Senat beachtet.
	$\alpha_{\{S.sponsorln\}\{sum(E.camount \rightarrow am\}}(\sigma_{sponsorsubstage=Senatecommittee}(\sigma_{sponsorparty=Republican}(E \bowtie_{earmarkid} ES \bowtie_{id} SPO)))$
Q10	Die Anfrage wählt die Nachnamen der Demokraten aus, die in New York aktiv sind oder waren.
	$\pi_{C.lastname}(Co \bowtie_{id} (\sigma_{AA.state=NY}(\sigma_{AA.party=Democrat}(AA))))$

Q11	Die Anfrage listet die Nachnamen der Sponsoren, die aus New York kommen und den Demokraten angehören.
	$\pi_{S.sponsorln}(\sigma_{SPO.sponsorstate=NY}(\sigma_{SPO.sponsorparty=Democrat}(SPO)))$
Q12	Die Anfrage vereinigt die gefundenen Nachnamen der vorherigen Anfragen Q11 und Q10.
	$Q10 \cup Q11$

Tabelle 5.1: Anfragen für die Evaluation mit kurzer Erklärung zur Funktion (aus [3]). Q5 hat eine angepasste Bedingung und mehrere Spaltenreferenzen sind an den Datensatz angepasst.

Use case	Anfrage	Why-Not Frage
Crime1	Q1	(P.Name:Hank, C.Type:Car theft)
Crime2		(P.Name:Roger, C.Type:Car theft)
Crime3	Q2	(P.Name:Roger, C.Type:Car theft)
Crime4		(P.Name:Hank, C.Type:Car theft)
Crime5		(P.Name:Hank)
Crime6	Q3	(C2.Type:kidnapping)
Crime7		(W.Name:Susan, C2.Type:kidnapping)
Crime8	Q4	(P2.Name:Audrey)
Crime9	Q8	((P.Name:Betsy, ct:x), x>8)
Crime10		(P.Name:Roger)
Imdb1	Q5	(name:Batman Begins)
Imdb2		(name:Avatar, L.locationId:USANew York)
Gov1	Q6	(Co.firstname:Christopher)
Gov2		(Co.firstname:Christopher, Co.lastname:MURPHY)
Gov3		(Co.firstname:Christopher, Co.lastname:GIBSON)
Gov4	Q7	(sponsorId:467)
Gov5		((SPO.sponsorln:Lugar, E.camount:x), x>=1000)
GOV6	Q9	((name:Bennett, am:x), x=18700)
Gov7	Q12	(name:JOHN)

Tabelle 5.2: Use cases [3]

Da der Datensatz nicht ganz auf die Use cases gepasst hat, sind zu Bidoit et al. kleine Unterschiede an den Anfragen und Use cases vorhanden. Diese ändern keine Funktionen und beziehen sich auf Spaltenreferenzen und Schreibweisen von Feldern. In Q5 ist die Bedingung auf das Jahr geändert, da sonst Imdb2 nicht funktioniert. Imdb1 und Imdb2 sind entsprechend angepasst.

5.2 Use case Performance

Bei der Ausführung der Use cases wird versucht, Daten von den Why-Not Provenance-Algorithmen mit Anfrage-basierten Erklärungen zu erhalten. In Abbildung 5.1 sind die Laufzeiten der Algorithmen zu den jeweiligen Use cases dargestellt. ConQueR fehlt in Abbildung 5.1, da nur für Crime5 Erklärungen generiert werden. In den anderen Use cases findet der Algorithmus entweder keine Erklärungen oder die Berechnung läuft in einen Fehler und wird nicht beendet. Im Folgenden gehe ich näher auf die Laufzeiten von NedExplain und Why-Not ein und vergleiche sie mit den Ergebnissen von Bidoit et al. [3].

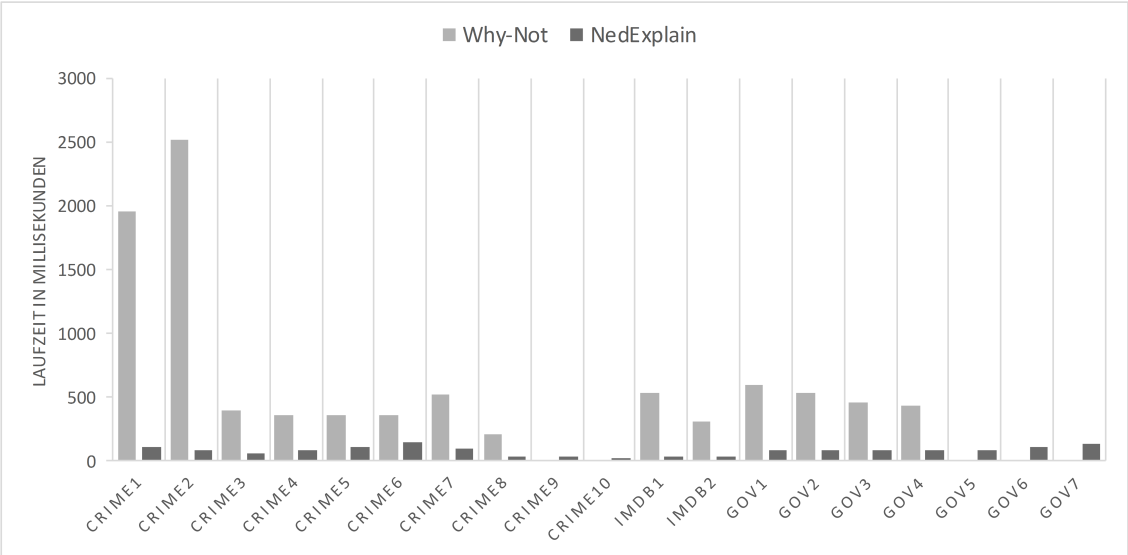


Abbildung 5.1: Performance von NedExplain und Why-Not für die Use cases aus Tabelle 5.2

Why-Not kommt mit den meisten der Use cases zurecht. Dabei werden nicht in allen Fällen Erklärungen gefunden, wie Bidoit et al. feststellen [3]. Grund dafür ist auch der Entwurf der Use cases, die Unterschiede in den Möglichkeiten aufzeigen sollen. Auch ohne Erklärungen berechnet der Why-Not Algorithmus die Herkunft und verfolgt kompatible Tupel durch die Berechnung. Deshalb sind auch die Laufzeiten für Fälle ohne Erklärung angegeben, da die Berechnung vollständigt durchgeführt wird. In den Use cases Crime9, Crime10 und Gov6 kann der Algorithmus keine Erklärungen finden, da er nicht mit Aggregationen umgehen kann.

Bei Gov5 ist die Ausführung reproduzierbar abgestürzt, weshalb auch hier keine Laufzeit für Why-Not vorhanden ist.

In allen anderen Use cases ist der Why-Not Algorithmus langsamer als NedExplain. Während NedExplain immer weniger als 150 Millisekunden zur Berechnung benötigt, läuft Why-Not bis zu 2 Sekunden in Crime2, bis die Erklärungen gefunden sind. Dabei benötigt die Herkunftsberechnung den größten Anteil der Laufzeit. Das Verfolgen der kompatiblen Tupel durch die Operatoren benötigt in den Use cases nie mehr als 100 Millisekunden. Für NedExplain lässt sich diese Aufteilung nicht so einfach untersuchen, da die Berechnung der Nachfolger als Teil der Berechnung des Algorithmus immer wieder für einen Operator durchgeführt wird.

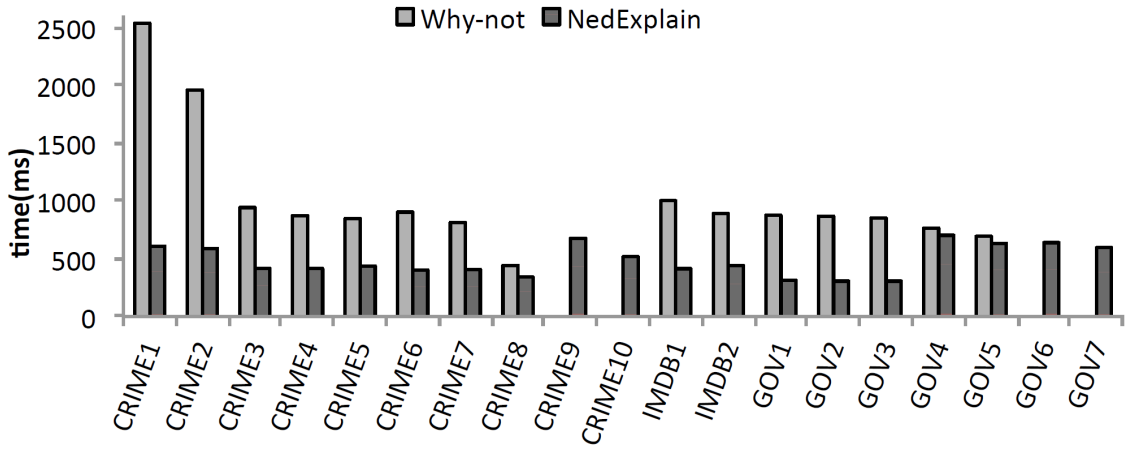


Abbildung 5.2: Performance von NedExplain und Why-Not aus Bidoit et al. [3]

Im Vergleich zu den Ergebnissen von Bidoit et al. (Abbildung 5.2) sind die Unterschiede zwischen Why-Not und NedExplain mit dieser Implementierung größer. Bidoit et al. verwenden, wie Why-Not, Trio zur Berechnung der Nachfolger [3]. Die Differenzen in den Laufzeiten von Why-Not in Abbildung 5.1 und Abbildung 5.2 ergeben sich durch die unterschiedliche Ausführungsumgebungen. Ich nehme bei der Implementierung von NedExplain außer der Nachfolgerberechnung keine weiteren Veränderungen am Algorithmus vor, die die Performance beeinflussen. Damit hängen die Unterschiede in der Laufzeit der NedExplain Implementierungen an der Nachfolgerberechnung, die ohne Trio messbar schneller ist.

5.3 Skalierbarkeit

Im zweiten Abschnitt der Evaluation gehe ich näher auf die Skalierbarkeit der Implementierung von NedExplain ein. Dafür stelle zuerst ich das Szenario vor und gehe im Anschluss auf die Ergebnisse ein.

Dafür vervielfältige ich die Daten der Relation *crime* in der *crime* Datenbank mit 603 Tupeln. In jedem Schritt wird die Anzahl der Tupel der Relation verzehnfacht. Dass keine doppelten

Tupel in der Relation entstehen, verändere ich das *sector* Feld. Bei der größten getesteten Skalierung enthält die Relation *crime* 6 Millionen Tupel. Ein weitere Vergrößerung ist mit dem vorhandenen Speicher der Java VM nicht möglich. Zur Messung der Laufzeit verwende ich den Use case Crime1. Die Relation *crime* enthält kompatible Tupel für den Use case, da dieser die Bedingung *C.Type:Car theft* auf die Relation enthält. Die Laufzeit von NedExplain wird normiert durch Vergleich mit der Laufzeit von Crime1 auf der unveränderten Datenbank.

Bei der Vervielfältigung unterscheide ich, ob die kompatiblen Tupel mit vervielfältigt werden. Im ersten Szenario werden nur alle nicht kompatiblen Tupel skaliert. Die Anzahl der kompatiblen Tupel bleibt in diesem Fall immer gleich. Im zweiten Szenario werden alle Tupel ohne Rücksicht auf die kompatiblen Tupel vervielfacht. Nach jedem Skalierungsschritt sind auch 10-mal so viele kompatible Tupel wie vorher in der Relation vorhanden. Da NedExplain die nur kompatiblen Tupel verfolgt, sollte das zweite Szenario langsamer berechnet werden als das erste.

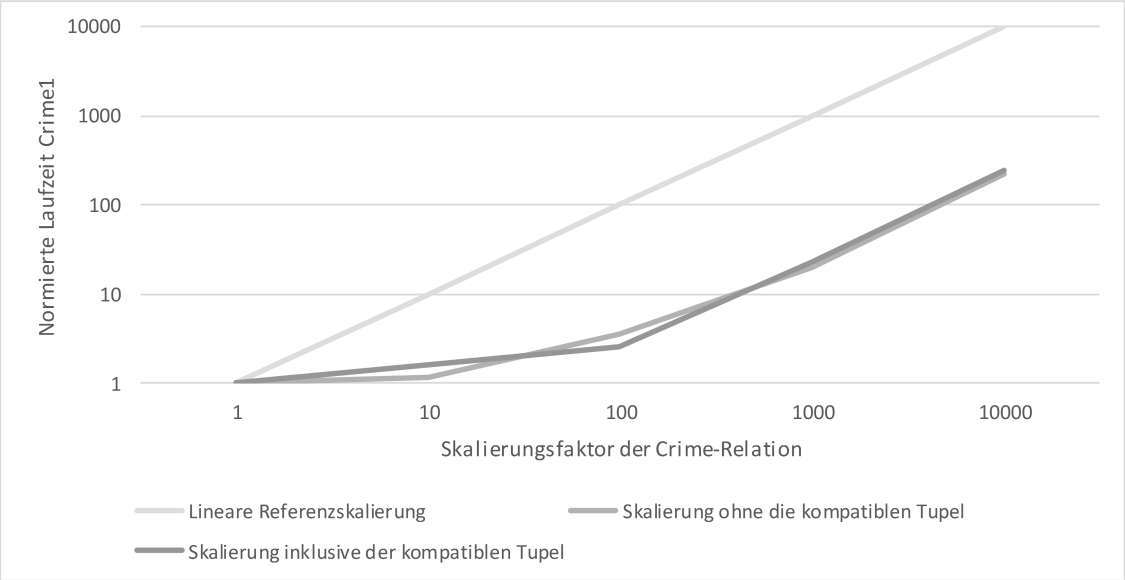


Abbildung 5.3: Normierte Laufzeiten für skalierte Crime-Relation. Die Referenz entspricht einem linearen Wachstum mit der Größe des Datensatzes. Der Skalierungsfaktor entspricht einer Vervielfachung der Tupel mit diesem Faktor.

Abbildung 5.3 zeigt die normierten Laufzeiten für die Implementierung von NedExplain. Laufzeit der Implementierung wächst bis zum Faktor 1000 wesentlich langsamer als die lineare Referenz. Ab da skaliert sie linear mit dem Datensatz. Insgesamt wächst die Laufzeit im getesteten Bereich nie schneller als der Datensatz. In absoluten Werten steigt die Laufzeit auch für den größten getesteten Skalierungsfaktor nicht über 25 Sekunden für Crime1.

Die Unterschiede zwischen den Szenarien mit und ohne Skalierung der kompatiblen Tupel ist vorhanden. Tendenziell ist die Variante mit weniger kompatiblen Tupeln schneller, da

NedExplain weniger Tupel verfolgen muss. Bei den größeren Datensätzen wird der Unterschied stärker sichtbar.

6 Zusammenfassung und Ausblick

NedExplain findet Anfrage-basierte Erklärungen für fehlende Tupel in der Ausgabe einer SQL-Anfrage. Anhand von zwei Beispielanfrage habe ich gezeigt, wie der Algorithmus die kompatiblen Tupel findet und durch die Operatoren verfolgt.

Ich habe NedExplain implementiert und in den Nautilus Analyzer integriert. Dies erlaubt den NedExplain zur Analyse des Debuggingszenarios auszuwählen. Der Algorithmus gibt die gefundenen Erklärungen über die Konsole aus. Die aufrufende Methode generiert aus dem Debuggingszenario eine Why-Not Frage für NedExplain. Die Implementierung von NedExplain enthält ein Interface aus Why-Not Frage für NedExplain, Verbindung zur Datenbank und SQL-Anfrage. Dabei entstehen keine Abhängigkeiten zu Klassen des Nautilus Analyzer.

Die Implementierung enthält Veränderungen gegenüber dem Entwurf von Bidoit et al.. NedExplain findet jetzt alle filternden Operatoren in der Anfrage. Die Suche nach Nachfolgern findet nicht mehr in Trio statt. Stattdessen generiert die Implementierung SQL-Anfragen der Nachfolger und blockierten kompatiblen Tupel, die dann von der Datenbank berechnet werden. Bei fehlerhafter Eingabe mit fehlenden Tupeln in der Ausgabe generiert der Algorithmus jetzt erfolgreich Erklärungen.

Im Vergleich zu den anderen Why-Not Provenance-Algorithmen mit Anfrage-basierten Erklärungen ist die Implementierung von NedExplain vielseitiger und schneller in der Berechnung. Die Suche nach Nachfolgern mit der Datenbank ist schneller als die Suche mit Trio. Außerdem skaliert die Implementierung mit großen Datensätzen ab einer Grenze linear.

Ausblick

Die bisherige Laufzeitevaluation betrachtet den Algorithmus als ein Modul. Eine Untersuchung, welchen Anteil die einzelnen Komponenten an der Gesamtlaufzeit haben, kann mögliche Optimierungsbereiche aufzeigen. Zusätzlich kann damit die eigentliche Laufzeit von NedExplain ohne die Berechnung der Nachfolger mit Why-Not verglichen werden.

Die Evaluation im Bereich der Skalierbarkeit enthält nur einen Use case, dessen Daten vergrößert werden. Die Untersuchung weiterer Use cases mit unterschiedlichen Datengrößen ist möglich.

NedExplain ist vorausschauend mit möglichst wenigen Abhängigkeiten zu Eclipse und dem Nautilus Plug-in entwickelt. Möglichkeiten zur Weiterentwicklung der Implementierung sind eine Standalone-Version und eine Integration in die Webanwendung.

Literaturverzeichnis

[1] 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.* 2015.

[2] N. Bidoit, M. Herschel, K. Tzompanaki. „Immutably answering why-not questions for equivalent conjunctive queries“. In: *Earth* (2014).

[3] N. Bidoit, M. Herschel, K. Tzompanaki. „Query-based why-not provenance with nedexplain“. In: *Extending Database Technology (EDBT).* 2014.

[4] J. A. Blakeley, P.-A. Larson, F. W. Tompa. „Efficiently Updating Materialized Views“. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data.* New York, NY, USA, 1986.

[5] A. Chapman, H. Jagadish. „Why not?“ In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 2009.

[6] C. A. Curino, H. J. Moon, C. Zaniolo. „Graceful Database Schema Evolution: The PRISM Workbench“. In: *Proc. VLDB Endow.* (Aug. 2008).

[7] U. Dayal, P. A. Bernstein. „On the Correct Translation of Update Operations on Relational Views“. In: *ACM Trans. Database Syst.* (Sep. 1982).

[8] M. Herschel. „A hybrid approach to answering why-not questions on relational query results“. In: *Journal of Data and Information Quality (JDIQ)* (2015).

[9] M. Herschel, H. Eichelberger. „The Nautilus Analyzer: understanding and debugging data transformations“. In: *Proceedings of the 21st ACM international conference on Information and knowledge management.* 2012.

[10] M. Herschel, T. Grust. „Transformation lifecycle management with nautilus“. In: *VLDB Workshop on the Quality of Data (QDB).* 2011.

[11] M. Herschel, M. A. Hernández. „Explaining missing answers to SPJUA queries“. In: *Proceedings of the VLDB Endowment* (2010).

[12] J. Huang, T. Chen, A. Doan, J. F. Naughton. „On the provenance of non-answers to queries over extracted data“. In: *Proceedings of the VLDB Endowment* (2008).

[13] T. Imieliński, W. Lipski Jr. „Incomplete information in relational databases“. In: *Journal of the ACM (JACM)* (1984).

[14] N. Mendel, B. Caddell, B. Ast. *Nautilus Web - Visualisierung aktueller Data Provenance Algorithmen im Web*. Forschungsber. Institut für Parallele und Verteilte Systeme, Universität Stuttgart, 17. Jan. 2017.

[15] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. Das Sarma, R. Murthy, T. Sugihara. „Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS". In: *Proc. of Conference on Innovative Data Systems Research, Monterey, USA*. Online publication, Jan. 2007.

[16] V. Raman, J. M. Hellerstein. „Potter's wheel: An interactive data cleaning system". In: *VLDB*. 2001.

[17] Q. T. Tran, C.-Y. Chan. „How to ConQueR why-not questions". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010.

Alle URLs wurden zuletzt am 29. 03. 2017 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