Where- und Why-Provenance für syntaktisch reiches SQL durch Kombination von Programmanalysetechniken

Tobias Müller
Universität Tübingen
Tübingen, Deutschland
to.mueller@uni-tuebingen.de

ABSTRACT

Das hier vorgestellte Verfahren ermöglicht die Analyse der Data Provenance von beliebigen SQL-Queries. Von der ebenfalls hier skizzierten Implementierung des Verfahrens werden unter anderem unterstützt: Subqueries, Aggregierungen, rekursive Queries und Window Functions. Eingabequeries werden zunächst in eine imperative Programmiersprache übersetzt. Der Programmcode wird mit einem neuen Verfahren analysiert, das auf bekannte Techniken aus dem Bereich der Programmanalyse aufbaut: Program Slicing, Kontrollflussanalyse und abstrakte Interpretation. Dadurch erhält man eine Berechnung von Where- und Why-Provenance auf der Granularitätsebene einzelner Tabellenzellen.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; H.2.3 [**Database Management**]: Languages—*Query Languages*; H.2.4 [**Database Management**]: Systems—*Relational Databases*

General Terms

Languages

Keywords

Data provenance, SQL, program analysis

1. EINFÜHRUNG

Wir stellen einen neuen Ansatz für die Analyse der *Data Provenance* [5] von SQL-Queries vor sowie eine prototypische Implementierung davon. Der hier präsentierte Ansatz erlaubt eine Analyse beliebiger (lesender) SQL-Queries. Die algebraische Ebene wird nicht berührt, wodurch auch keine algebraischen Restriktionen auftreten. Zum Beispiel ist [1] eingeschränkt auf eine *positive relationale Algebra*.

Die theoretische Anwendbarkeit auf beliebige Queries wird dadurch erreicht, dass wir Eingabequeries zuerst in eine imperative (Turing-vollständige) Programmiersprache übersetzen, das resultierende Programm in eine *linearisierte Form* (erläutert in Abschnitt 4) umwandeln und anschließend die eigentliche Provenance-Analyse durchführen. Dieser von uns entwickelte Ansatz basiert auf dem Prinzip des *Program Slicing* [10, 3].

Von bisherigen Arbeiten wurde sukzessive der Umfang der analysierbaren SQL-Konstrukte erweitert. Dazu zählen beispielsweise geschachtelte Subqueries [6] sowie Aggregierungen [1]. Mit der in der vorliegenden Arbeit vorgestellten Implementierung ist eine Analyse dieser Konstrukte ebenfalls möglich. Nach dem Wissensstand der Autoren erlaubt unsere Implementierung erstmalig auch die Analyse von Window Functions und rekursiven Queries.

1.1 Data Provenance

Das allgemeine Ziel der Berechnung von Data Provenance ist es, den Ursprung und die zurückgelegten Verarbeitungsschritte des Resultats von Datenverarbeitung sichtbar zu machen. Auf dem Gebiet der relationalen Datenbanken haben wir uns konkret mit der Frage beschäftigt, auf welchen Ursprungsdaten (hier: Tabellenzellen) genau das Ergebnis einer SQL-Query beruht.

Seit [2] unterscheidet die wissenschaftliche Literatur zwei Arten von Provenance:

- *Where-Provenance* charakterisiert sowohl eine direkte Abhängigkeit von Werten durch Kopieren sowie zum Beispiel auch bei Aggregierungen, in denen viele Werte zu einem zusammengefasst werden.
- *Why-Provenance* charakterisiert Abhängigkeiten durch Kontrollflussentscheidungen. Diese liegen zum Beispiel vor, wenn die Existenz eines (Teil-)Ergebnisses von einem anderen Wert abhängig ist, der als Filterkriterium dient (= Semantik einer `WHERE`-Klausel in SQL).

Basierend auf dem hier vorgestellten Ansatz sowie mit unserer prototypischen Implementierung können beide Arten von Provenance berechnet werden.

In Abschnitt 2 werden Beispiel-Queries und deren Analyseergebnisse vorgestellt. Die Abschnitte 3 bis 4 erläutern die Grundzüge des Analyseverfahrens. In Abschnitt 5 beschreiben wir eine konkrete Implementierung. Aus Platzgründen ist nur eine verkürzte Darstellung möglich.

2. BEISPIEL-QUERIES

Im Folgenden werden zwei Beispiel-Queries und die Ergebnisse der mit unserer Implementierung durchgeführten Provenance-Analyse vorgestellt. Die erste Query greift ein Beispiel aus der Literatur auf und die zweite wurde gewählt, um die Mächtigkeit unseres Ansatzes zu demonstrieren.

27^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 26.05.2015 - 29.05.2015, Magdeburg, Germany.

agencies

	name	based_in	phone
t_1	BayTours	San Francisco	415-1200
t_2	HarborCruz	Santa Cruz	831-3000

externaltours

	name	destination	type	price
t_3	BayTours	San Francisco	cable car	$50
t_4	BayTours	Santa Cruz	bus	$100
t_5	BayTours	Santa Cruz	boat	$250
t_6	BayTours	Monterey	boat	$400
t_7	HarborCruz	Monterey	boat	$200
t_8	HarborCruz	Carmel	train	$90

Abbildung 1: Bootstouren-Beispiel: *Where-* und *Why*-Provenance sind markiert mit sowie . Falls beides zutrifft, wird verwendet. Tupel sind mit t_i bezeichnet.

```
SELECT e.name, a.phone
FROM   agencies AS a,
       externaltours AS e
WHERE  a.name = e.name
AND    e.type = 'boat'
```

output

name	phone
HarborCruz	831-3000
BayTours	415-1200
BayTours	415-1200

(a) SFW-Query **(b) Ergebnis**

Abbildung 2: Welche Agenturen bieten Bootstouren an?

2.1 Bootstouren

Diese Beispiel-Query stammt aus [4] und reproduziert die dort gefundene Data Provenance. In Abbildung 1 sind die Eingabetabellen dargestellt: `agencies` enthält Stammdaten von Reiseveranstaltern und `externaltours` enthält deren angebotene Touren. Die Query in Abbildung 2(a) findet diejenigen Veranstalter, die Bootstouren im Angebot haben. Die Ausgaberelation steht in Abbildung 2(b).

Zelle ❶ ist hier als Where-abhängig von t_5:BayTours markiert. Ein Blick auf die SQL-Query (`SELECT e.name`) bestätigt dieses Ergebnis: denn hier werden Werte aus der Eingabetabelle ins Resultat kopiert. Dies entspricht den Kriterien von Where-Provenance, wie wir sie in Abschnitt 1 angegeben haben.

Die Markierungen zeigen außerdem Why-Abhängigkeiten von t_1:BayTours, t_5:BayTours und t_5:boat. Diese drei Werte werden im `WHERE`-Teil der SQL-Query für die Join- und Filterkriterien benutzt.

❶ und ❷ zeigen, dass die wertemäßig nicht unterscheidbaren BayTours und BayTours anhand ihrer jeweiligen Provenance (t_5 oder t_6) unterscheidbar werden.

❸ veranschaulicht, dass Data Provenance entgegen der wörtlichen Bedeutung auch in Vorwärts-Richtung funktioniert. Die Farbmarkierungen besagen, dass t_1:415-1200 mit zwei Werten von der Ausgabetabelle auf Werteebene zusammenhängt. Konkret wird hier die Telefonnummer zwei Mal kopiert.

2.2 Endlicher Automat

Die hier vorgestellte Provenance-Analyse einer rekursiven Query ist nach dem Wissen der Autoren mit keiner anderen existierenden Implementierung möglich. Die Query besteht auch aus einer Anzahl von Funktionsaufrufen. Interessante Ausschnitte des Ergebnisses unserer Provenance-Analyse werden erneut mit farbigen Markierungen dargestellt.

Abbildung 3(a) zeigt die Eingaberelationen. `compounds` enthält chemische Summenformeln und ihre Bezeichnungen. In Tabelle `fsm` ist ein endlicher Automat codiert, der die Syntax dieser Formeln überprüfen kann.

Die SQL-Query in Abbildung 3(b) führt diesen Automaten aus. Da alle Formeln in `compounds` parallel verarbeitet werden, existieren mehrere Automaten parallel. In jedem Schritt eines Automaten wird das erste Zeichen einer Formel abgeschnitten und der entsprechende Zustandsübergang durchgeführt. Aktueller Zustand und "Restformel" sind in der Tabelle `run` gespeichert, die bei jedem Schritt der Automaten neu berechnet wird. In Abbildung 3(c) sind zwei Versionen von `run` abgebildet: direkt nach der Initialisierung (`step`: 0) und nach drei Schritten (`step`: 3). Wenn die Formeln vollständig verzehrt sind, beendet sich der rekursive Teil der Query. Als Endresultat werden die Ableitungsschritte für *citrate* (siehe Abbildung 3(d)) zurückgegeben.

Die Where-Provenance von ❹ zeigt an, von welchen Werten O_7^{3-} abgeleitet wurde. Dazu zählt erst einmal die vollständige Summenformel t_9:$C_6H_5O_7^{3-}$. Aber auch alle Zwischenzustände werden von der Analyse erfasst, wie anhand der Markierungen in Abbildung 3(c) zu erkennen ist.

Die interessantesten Why-Abhängigkeiten von ❹ sind innerhalb der Tupel t_{12} und t_{13} zu finden. Das sind nämlich gerade die Kanten des Automaten, die besucht wurden, um O_7^{3-} abzuleiten.

compounds

	compound	formula
t_9	citrate	$C_6H_5O_7^{3-}$
t_{10}	glucose	$C_6H_{12}O_6$
t_{11}	hydronium	H_3O^+

fsm

	source	labels	target	final
t_{12}	0	A..Za..z	1	false
t_{13}	1	A..Za..z$_{0..9}$	1	true
t_{14}	1	$^{0\ 9}_{..}$	2	true
t_{15}	1	+-	3	true
t_{16}	2	$^{0\ 9}_{..}$	2	false
t_{17}	2	+-	3	false
t_{18}	3	A..Za..z	1	true

(a) Eingabetabellen

```
WITH RECURSIVE
run(compound, step, state, formula) AS (
  SELECT compound, 0, 0, formula
  FROM   compounds
 UNION ALL
  SELECT this.compound,
         this.step + 1 AS step,
         edge.target AS state,
         right(this.formula, -1) AS formula
  FROM   run AS this,
         fsm AS edge
  WHERE  length(this.formula) > 0
  AND    this.state = edge.source
  AND    strpos(edge.labels,
                left(this.formula, 1)) > 0
)
SELECT r.step, r.state, r.formula
FROM   run AS r
WHERE  r.compound = 'citrate'
```

(b) Rekursive Query

run

compound	step	state	formula
citrate	0	0	$C_6H_5O_7^{3-}$
glucose	0	0	$C_6H_{12}O_6$
hydronium	0	0	H_3O^+

⋮

run

compound	step	state	formula
citrate	3	1	$_5O_7^{3-}$
glucose	3	1	$_{12}O_6$
hydronium	3	1	$^+$

(c) Zwischenergebnisse

output

step	state	formula
0	0	$C_6H_5O_7^{3-}$
1	1	$_6H_5O_7^{3-}$
2	1	$H_5O_7^{3-}$
3	1	$_5O_7^{3-}$
4	1	O_7^{3-}
5	1	$_7^{3-}$
6	1	$^{3-}$
7	2	-
8	3	

(d) Ergebnis

Abbildung 3: Endlicher Automat, der die Syntax von chemischen Summenformeln überprüft.

```python
def query(agencies, externaltours):
  #FROM clause: read source tables
  rows = []
  for tupVar2 in agencies:
      for tupVar3 in externaltours:
          rs = {"tupVar2": tupVar2,
                "tupVar3": tupVar3,
                "tmp": {},
                }
          rows.append(rs)
  #WHERE clause: compute where predicate
  rowIdx = 0
  while rowIdx < len(rows):
      rs = rows[rowIdx]
      col4 = rs["tupVar2"]["name"]
      col5 = rs["tupVar3"]["name"]
      res6 = col4 == col5
      col7 = rs["tupVar3"]["type"]
      val8 = "boat"
      res9 = col7 == val8
      res10 = res6 and res9
      rs["tmp"]["where"] = res10
      rowIdx = rowIdx + 1
  #WHERE clause: apply where predicate
  filtered = []
  for rs in rows:
      if rs["tmp"]["where"]:
          filtered.append(rs)
  rows = filtered
  #SELECT clause: compute result columns
  rowIdx = 0
  while rowIdx < len(rows):
      rs = rows[rowIdx]
      col11 = rs["tupVar3"]["name"]
      col12 = rs["tupVar2"]["phone"]
      rs["tmp"]["eval0"] = col11
      rs["tmp"]["eval1"] = col12
      rowIdx = rowIdx + 1
  #SELECT clause: assemble result table
  ship = []
  for rs in rows:
      row = {}
      row["name"] = rs["tmp"]["eval0"]
      row["phone"] = rs["tmp"]["eval1"]
      ship.append(row)
  return ship
```

Listing 1: Übersetzung der Bootstouren-Query. Es findet (noch) keine Code-Optimierung statt.

Zuletzt wird mit ❺ noch die Einsicht transportiert, dass die Schrittzahl des Automaten keinerlei Einfluss auf die Ableitung von *citrate* hat. Die Markierungen zeigen lediglich eine Where-Provenance zwischen den Schritten 0 bis 8 an, die auf das Inkrementieren zurückzuführen ist. Es gibt keine Why-Provenance, das heißt keine (versehentliche) Beeinflussung des Endresultats durch die Schrittzählung.

3. SQL-ÜBERSETZUNG

Die zu analysierenden SQL-Queries werden in unserer Implementierung zunächst in Python-Programme übersetzt. Python hat als Zwischensprache den Vorteil, sehr leichtgewichtig und daher für die weitere Analyse gut zugänglich zu sein. Es wird außerdem von Kooperationspartnern eingesetzt. Ein Nachteil von Python ist die schlechtere Performance (gegenüber Sprachen wie C). Es gibt jedoch keinen Hinderungsgrund, die mit Hilfe von Python erarbeiteten Techniken der Provenance-Analyse nicht auf andere Sprachen wie LLVM zu übertragen.

Damit würden wir einem aktuellen Forschungstrend in der Datenbank-Community folgen, SQL nicht länger mit dem Volcano-Iterator-Model zu implementieren, sondern Queries just-in-time zu kompilieren (siehe [9]).

Aus Platzgründen wird der von uns verwendete Übersetzer nur anhand eines Beispiels vorgestellt. Listing 1 zeigt, wie die Übersetzung von Query 2(a) aussieht. Je Query oder Sub-Query wird eine eigene Python-Funktion erzeugt. Die Argumente und Rückgabewerte sind Tabellen, die in Python als Listen von Dictionaries implementiert sind. Die SQL-Klauseln einer Query werden in eine Reihenfolge gebracht, die eine Berechnung im imperativen Paradigma erlaubt: SFW wird beispielsweise zu FWS.

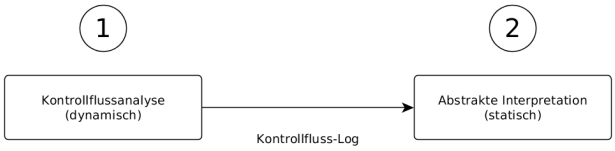

Abbildung 4: Hauptelemente der Provenance-Analyse

4. PROVENANCE-ANALYSE IN ZWEI STUFEN

Wie in Abbildung 4 dargestellt, teilt sich die Provenance-Analyse im Wesentlichen auf zwei Schritte auf: ① *Kontrollflussanalyse* (dynamisch, zur Laufzeit) und ② *abstrakte Interpretation* (statisch, zur Kompilierzeit).

Die Kontrollflussanalyse ist dafür zuständig, alle für den Kontrollfluss benötigten Prädikate zu bestimmen und deren Werte zu speichern. Beispielsweise würde bei der Ausführung einer `if`-Anweisung die zugehörige Kontrollflussinformation darin bestehen, ob entweder der `if`- oder der `else`-Rumpf ausgeführt wird. Die Information kann durch einen einzelnen booleschen Wert codiert werden.

In Phase ② findet die eigentliche Provenance-Analyse statt. Hier wird das Kontrollfluss-Log benutzt, um den tatsächlichen Ausführungspfad nachvollziehen zu können, den das Programm zur Laufzeit genommen hat.

In Abschnitt 4.1 wird die Motivation für die soeben skizzierte Struktur geschildert sowie ein Bezug zu Ergebnissen der theoretischen Informatik hergestellt, die einer Programmanalyse harte Grenzen setzt. Abschnitt 5 erläutert die beiden Analyseschritte genauer sowie deren Implementierung in Python.

4.1 Linearisierung

Eine rein statische Provenance-Analyse ist im Allgemeinen für Python-Programme nicht möglich, denn Python ist eine Turing-vollständige Programmiersprache. Der *Satz von Rice* besagt, dass nicht-triviale Laufzeiteigenschaften (wie Data Provenance) für allgemeine Turing-Maschinen nicht algorithmisch entscheidbar sind.

Um den Konsequenzen des Satzes von Rice zu entgehen, ändert dieses Verfahren die Voraussetzungen. Wie in Abbildung 4 dargestellt, wird der zur Laufzeit aufgezeichnete Kontrollfluss an die abstrakte Interpretation übergeben.

Zu Kontrollflussanweisungen zählen unter anderem `if`- und `while`-Anweisungen. Für diese Konstrukte besteht das zugehörige Kontrollfluss-Log lediglich aus einer Folge von booleschen Werten:

- Wird entweder `if` oder `else` ausgeführt?
- Wird der Rumpf von `while` (nochmal) ausgeführt oder wird die Schleife beendet?

Dieses Log steht also während der statischen Analyse zur Verfügung. Das heißt, die statische Analyse weiß für jedes `if x:`, ob es in Wirklichkeit entweder ein `if True:` oder `if False:` ist - je nachdem, ob `True` oder `False` im Log steht.

Mit dem Kontrollfluss-Log findet deshalb eine Linearisierung des Python-Programms statt. Die Abfolge der Anweisungen im Programm ist statisch festgelegt, weil der Kontrollfluss festgelegt ist. Kontrollfluss-Konstrukte verhalten sich nun transparent und im einfachsten Fall besteht die restliche Analyse des Programms nur noch darin, Zuweisungen an Variablen zu betrachten.

Dadurch liegt in ② keine Turing-Vollständigkeit mehr vor. Der Satz von Rice gilt nicht mehr und eine Provenance-Analyse ist (quasi-statisch) möglich.

Das Beispiel in Abschnitt 5 wird das verdeutlichen.

4.2 Granularität und Auflösung

Als *Granularität* (oder *level of detail*) wird in [7] bezeichnet, was die kleinstmöglichen Datenstrukturen sind, die in einer Provenance-Analyse berücksichtigt werden. Meist sind das entweder Tupel oder Tabellenzellen. In dem hier besprochenen Ansatz besteht die Granularität in Zellen.

Orthogonal dazu wollen wir den Begriff *Auflösung* verwenden, um damit die Größe der Programmfragmente zu bezeichnen, für die am Ende der Analyse eine Data Provenane ausgegeben wird.

Beispielsweise könnte es in einer niedrigen Auflösung so sein, dass das Programm nur als Ganzes analysiert wird. Das heißt, die Data Provenance bezieht sich gerade auf die Ein/Ausgabedaten des Programms selbst und zeigt an, wie die Ausgabedaten von den Eingabedaten abhängig sind. Eine andere Variante bestünde darin, für jeden einzelnen Ausdruck in diesem Programm eine Data Provenance auszugeben. Das heißt, für einen Ausdruck wie `b+c` wird als Ergebnis die Abhängigkeit von den Einzelwerten `b` sowie `c` ausgegeben. Hier ist die Auflösung sehr hoch.

Von uns wurde als Auflösung die Ebene von Funktionsaufrufen benutzerdefinierter Funktionen gewählt. Im Ergebnis der Provenance-Analyse sind deshalb die Parameter und Rückgabewerte von Funktionsaufrufen aufgeführt sowie die dazugehörende Data Provenance.

5. IMPLEMENTIERUNG

Als Grundlage für die Implementierung der Provenance-Analyse dient *CPython* in Version 3.4. Dabei handelt es sich um die stabile und zum Zeitpunkt des Schreibens dieses Artikels aktuelle Referenzimplementation von Python. Intern übersetzt sie Quelltext in Bytecode, der anschließend in der zugehörigen virtuellen Maschine (VM) ausgeführt wird. Als Zwischenschritt während der Übersetzung erzeugt CPython einen Abstract Syntax Tree (AST), auf dessen Basis wir das Analyseverfahren implementiert haben.

In Abbildung 5 ist dargestellt, wie die einzelnen Python-Komponenten zusammenarbeiten. Mit weißem Hintergrund dargestellt sind die ein/ausgehenden Datensätze (Relationen) sowie der Python Programmcode, der analysiert werden soll. Grau hinterlegt sind Komponenten der CPython-Implementierung selbst, die unmodifiziert verwendet werden. In schwarz ist, was wir zusätzlich implementiert haben.

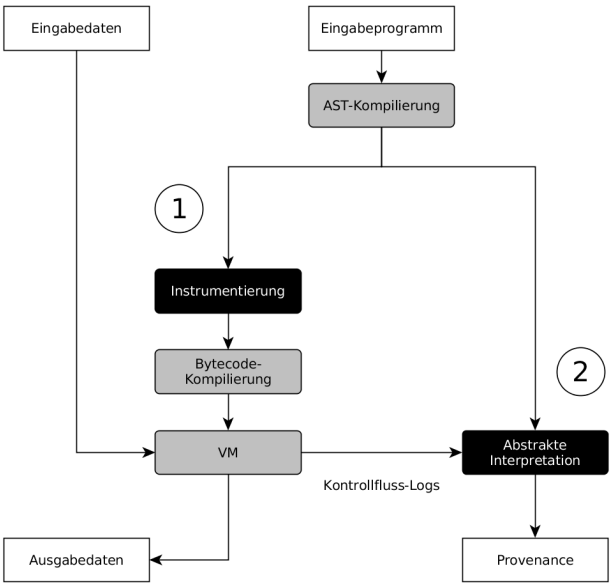

Abbildung 5: Komponenten der Python-Implementierung

Während der Analyse wird das zu einem AST kompilierte Eingabeprogramm zwei Mal verarbeitet. In Schritt ① wird es zunächst instrumentiert. Dabei werden zusätzliche Python-Anweisungen eingefügt, die einerseits die vorhandene Funktionalität nicht verändern und andererseits für das Schreiben des Kontrollfluss-Log zuständig sind. Das so modifizierte Programm wird zu Bytecode kompiliert und anschließend mit der VM ausgeführt. Während der Ausführung werden Eingabe/Ausgabedaten gelesen/geschrieben sowie das Kontrollfluss-Log produziert. Schritt ① wird in Abschnitt 5.1 genauer beschrieben.

In Schritt ② wird das unmodifizierte (nicht instrumentierte) Eingabeprogramm erneut hergenommen und mit Hilfe des Kontrollfluss-Logs die eigentliche Provenance-Analyse durchgeführt. Eine genaue Beschreibung dieses Teils folgt in Abschnitt 5.2. Bemerkenswert ist, dass hier keine Eingabedaten benötigt werden. Die abstrakte Interpretation findet auf symbolischer Ebene statt, das heißt es wird lediglich ermittelt, wie die im Programm benutzten Variablen voneinander abhängig sind. Dies genügt, um sowohl Why- als auch Where-Provenance zu berechnen.

5.1 Instrumentierung

Allgemein gesprochen müssen diejenigen Sprachkonstrukte instrumentiert werden, deren Verhalten bezüglich Data Provenance sich nicht durch rein statische Analyse bestimmen lässt. Bisher instrumentieren wir zwei Kategorien von Sprachkonstrukten: (i) Kontrollfluss und (ii) Indexzugriff.

Listing 2 zeigt ein bereits instrumentiertes Programmfragment, das je ein Beispiel für beide Fälle enthält. Die Funktion `dow()` erhält als Argumente den Wochentag als Zahlenwert (`num = 0...6`) und eine Liste mit Namen von Wochentagen. Sie liefert eine String-Repräsentation zurück. Sie unterstützt außerdem zwei verschiedene Datumsformate: Wochenbeginn am Montag (`fmt = True`) oder am Sonntag (`fmt = False`).

```
1  days = ["Sun","Mon","Tue",
2          "Wed","Thu","Fri","Sat"]
3  def dow(num, fmt, days):
4      if fmt:
5          #true: week starts on Monday
6          logCtrl(True)
7          pos = (num+1) % 7
8      else:
9          #false: week starts on Sunday
10         logCtrl(False)
11         pos = num
12     logIdx(pos)
13     res = days[pos]
14     return res
15 dow(4, True, days) #returns: "Fri"
```

Listing 2: Beispiel für instrumentierten Programmcode. Die Instrumentierungsanweisungen sind umrandet.

Die beiden instrumentierten Anweisungen sind: (i) das `if`/`else`-Statement und (ii) der Ausdruck `days[pos]` (Zeile 13). An beiden Stellen wäre eine rein statische Analyse nicht ausreichend. Bei (i) ist nicht bekannt, welcher Rumpf überhaupt ausgeführt wird und bei (ii) ist unbekannt, auf welches Listenelement von `days` zugegriffen wird.

Im Gegenzug ist die Data Provenance einer Anweisung wie zum Beispiel `pos = (num+1) % 7` schon bei statischer Analyse klar: der Wert von `pos` ist Where-abhängig von `num`. In diesem Fall ist keine Instrumentierung erforderlich.

Wird `dow()` mit den Parametern von Zeile 15 aufgerufen, werden folgende Logs produziert:

- *Kontrollfluss*: `[True]`
- *Indices*: `[5]`

In diesem bewusst kurz gehaltenem Beispiel bestehen beide Logs aus jeweils nur einem Element. Für längere Programme würden weitere Einträge einfach in der Reihenfolge hinzugefügt, in der die Instrumentierungsanweisungen aufgerufen werden. In Schritt ② werden die Einträge in genau derselben Reihenfolge wieder gelesen. Deshalb braucht das Log nur sequenziell zu sein. Die Zuordnung eines Log-Eintrags zur zugehörigen Anweisung im analysierten Programm ist implizit gegeben.

5.2 Abstrakte Interpretation

Nachdem die Kontrollfluss-Logs erzeugt worden sind, kann jetzt die eigentliche Ableitung der Data Provenance stattfinden. Dazu werden wie in Abbildung 5 dargestellt, lediglich das Eingabeprogramm zusammen mit den Logs verwendet. Daten sind an dieser Phase nicht beteiligt. Dementsprechend werden auch keine Berechnungen von Werten ausgeführt, was Systemressourcen spart.

Die *abstrakte Interpretation* wird durchgeführt, indem alle Anweisungen des Programms in derselben Reihenfolge nachvollzogen werden, in der sie auch zur Laufzeit ausgeführt wurden. Die richtige Reihenfolge einzuhalten ist dank des aufgezeichneten Kontrollflusses sehr einfach. Immer dann, wenn eine Kontrollflussentscheidung benötigt wird (zum Beispiel: `if`- oder `else`-Block ausführen), kann diese Information im Kontrollfluss-Log nachgeschlagen werden.

Where-Provenance.

Während die Anweisungen nacheinander interpretiert werden, wird eine Variablenumgebung gepflegt und mit jeder interpretierten Anweisung gegebenenfalls aktualisiert. Die Umgebung beinhaltet alle derzeit sichtbaren Variablen zusammen mit ihren jeweiligen Abhängigkeiten von den Eingabedaten.

```
1  days = [(),(),(),
2          (),(),(),(),()]
3  def dow(num, fmt, days):
4      if fmt: #fmt: True
5          pos = (num+()) % ()
6      else:
7          pos = num
8      res = days[pos] #pos: 5
9      return res
10 dow((), (), days) #returns: ()
```

Listing 3: Pseudocode aus Sicht der abstrakten Interpretation.

Der Aufbau dieser Umgebung ist ein inkrementeller Prozess. Jede Zuweisung einer Variablen, wie zum Beispiel in `a = b`, wird eine entsprechende Aktualisierung der Umgebung nach sich ziehen. In diesem Beispiel müssten alle Abhängigkeiten von `b` nach `a` kopiert werden.

Auf diese Weise wird die Umgebung ständig aktuell gehalten und referenziert in ihren Abhängigkeiten stets die Eingabedaten. Am Ende der Analyse braucht nur noch die gewünschte Variable, zum Beispiel `res`, in der Umgebung nachgeschlagen zu werden.

Why-Provenance.

Die Ausführung von Anweisungen in einem `if`- oder `else`-Rumpf sind abhängig vom zugehörigen Prädikat des `if`/`else`-Konstrukts. Diese Abhängigkeit modellieren wir als Why-Provenance.

In der Analyse wird dazu eine Menge an Abhängigkeiten gepflegt, die dem Kontrollfluss selbst zugeordnet ist. Bei Eintritt in den Rumpf einer `if`-Anweisung wird dem Kontrollfluss eine Abhängigkeit vom Prädikat dieser `if`-Anweisung zugeordnet. Allen Zuweisungen, die in diesem Rumpf ausgeführt werden, werden dann wiederum die Abhängigkeiten des Kontrollflusses in Form von Why-Provenance hinzugefügt.

Nachdem der `if`-Rumpf abgearbeitet ist, werden die Abhängigkeiten des Kontrollflusses wieder zurückgesetzt. Ob `if`- oder `else`-Rumpf ausgeführt werden, macht hier keinen Unterschied: in beiden Fällen gilt dasselbe Prädikat. Die Interpretation von Schleifen funktioniert analog.

Beispiel-Analyse.

In Listing 3 ist abgedruckt, wie sich das aus dem Instrumentierungsschritt bereits bekannte Programmfragment in Schritt ② der Provenance-Analyse darstellen würde. Alle vorkommenden atomaren Werte sind durch `()` ersetzt worden. Code in dieser Form wird nicht erzeugt, doch das Listing veranschaulicht, auf welcher Basis die abstrakte Interpretation arbeitet.

Anhand dieses Beispiels wird nun dargestellt, wie die Ableitung der Data Provenance funktioniert. Das Ergebnis dieser Analyse besteht gemäß der gewählten Granularität und Auflösung (siehe Abschnitt 4.2) darin, wie die Variable `res` von den Funktionsparametern abhängig ist.

Um die Analyse zu initialisieren, werden die in den Funk-

days: [0_e, 1_e, 2_e, 3_e, 4_e, 5_e, 6_e]
num: 7_e
fmt: 8_e

Abbildung 6: Initialisierung

tionsparametern vorkommenden Werte durch künstliche *ids* repräsentiert. Abbildung 6 zeigt die im Beispiel gewählte Repräsentation, die aus ansteigenden natürlichen Zahlen besteht. Die Variable `days` ist eine Liste und enthält dementsprechend für jedes Listenelement einen anderen Repräsentanten. Das tiefgestellte $_e$ soll anzeigen, dass eine Where-Abhängigkeit besteht (= Abhängigkeit vom Wert an dieser Stelle). Why-Abhängigkeiten kommen später hinzu und werden entsprechend durch $_y$ gekennzeichnet.

Diese Repräsentanten ersetzen während der abstrakten Interpretation die tatsächlichen Werte. Am Ende der Analyse von `dow()` wird die Variable `res` nicht mit dem Wert `"Fri"` belegt sein, sondern mit einer Menge von Repräsentanten.

In Tabelle 1 sind die einzelnen Analyseschritte der abstrakten Interpretation aufgeführt. In der ersten Spalte steht die Zeilennummer des Python-Statements, das soeben interpretiert wurde. Die mittlere Spalte enthält die Abhängigkeiten des Kontrollflusses. Die letzte Spalte zeigt den Inhalt der Variablenumgebung. Um die Übersichtlichkeit zu verbessern, wird hier die Variable `days` nicht dargestellt.

In *Zeile 3* wird die Funktion betreten. Hier findet die Initialisierung der Datenstrukturen statt, das heißt die ids 0..8 werden vergeben. `num`, `fmt` und `days` sind die Parameter der Funktion und werden der Umgebung hinzugefügt.

In *Zeile 4* beginnt das `if/else`-Konstrukt. Hier werden zunächst die Abhängigkeiten des `if/else`-Prädikats `fmt` den Kontrollfluss-Abhängigkeiten hinzugefügt. Dabei findet eine Transformation von 8_e zu 8_y statt. Die Umgebung bleibt unverändert. Als dritter Punkt wird das Kontrollfluss-Log gelesen, um festzustellen, ob der `if`- oder `else`-Rumpf besucht werden soll. Das Log liefert ein `True` zurück, also wird eine Analyse des `if`-Rumpfs durchgeführt.

Zeile 5 enthält eine Zuweisung an die Variable `pos`. Dazu wird zunächst der Ausdruck `(num+()) % ()` analysiert. Die beiden `()` haben hier keine Abhängigkeiten, werden also einfach ignoriert. Die Abhängigkeiten von `num` werden kopiert: 7_e. Außerdem werden die Kontrollflussabhängigkeiten übernommen: 8_y.

Zuletzt findet in *Zeile 8* ein Listenzugriff statt. Hier wird das Indices-Log bemüht, das den Index `5` enthält. An 5. Stelle von *days* befindet sich 5_e und wird nach `res` kopiert. Darüber hinaus hat der Index wie auch der Kontrollfluss eine steuernde Funktion. Deshalb wird der Inhalt von `pos` als Why-Provenance kopiert.

Das Ergebnis der Analyse für `res` ist: [8_y, 7_y, 5_e], also Where-Abhängigkeit von `"Fri"` und Why-Abhängigkeit von `fmt: True` und `num: 4`.

6. ZUSAMMENFASSUNG UND AUSBLICK

Unser hier vorgestellte Ansatz und seine konkrete Implementierung erweitert die bisherigen Grenzen der Provenance-Analyse von SQL. Window Functions und rekursive Queries sind Bestandteile aktueller DBMS-Implementierungen und können von unserem Prototypen analysiert werden.

Derzeit wird im Rahmen einer studentischen Arbeit eine zusätzliche Python-Implementierung der hier vorgestellten Provenance-Analyse entwickelt. Ihr Merkmal besteht darin, dass sie Bytecode direkt analysiert (statt den Python-AST).

Zeile	Abhängigkeiten: Kontrollfluss	Abhängigkeiten: Variablen
3 (`dow()`)		`num`: 7_e `fmt`: 8_e
4 (`if fmt:`)	8_y	`num`: 7_e `fmt`: 8_e
5 (`pos=`)	8_y	`num`: 7_e `fmt`: 8_e `pos`: 8_y, 7_e
8 (`res=`)		`num`: 7_e `fmt`: 8_e `pos`: 8_y, 7_e `res`: 8_y, 7_y, 5_e

Tabelle 1: Ableitung der Provenance

Wir versprechen uns eine Performanceverbesserung von dieser neuen Implementierung.

Als weitere Implementierung ist LLVM geplant, um mittels [9] kompilierte Queries analysieren zu können.

Habitat [8] ist ein SQL-Debugger, der eingesetzt wird, um potentiell fehlerhafte Queries direkt auf SQL-Sprachebene zu untersuchen. Dazu wird die verdächtige Query von Habitat instrumentiert und vom RDBMS ausgeführt. Die instrumentierte Query *beobachtet* (= zeichnet auf), wie die potentiell fehlerhafte Ergebnisrelation berechnet wird und präsentiert dem Benutzer diese Beobachtungen. Bei großen Eingabetabellen besteht das Problem, dass man vielleicht nur an der Beobachtung der Berechnung eines einzelnen Ergebnistupels interessiert ist, aber auch tausende andere Tupel zusätzlich beobachtet. Um genau die für ein bestimmtes Ergebnistupel relevanten Eingabetupel herauszufinden, ist Data Provenance genau das richtige Werkzeug. Eine Kombination von diesen beiden Techniken wird von uns angestrebt.

7. REFERENCES

[1] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for Aggregate Queries. In *Proc. PODS*, pages 153–164. ACM, 2011.

[2] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *Proc. ICDT*, pages 316–330. Springer, 2001.

[3] J. Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.

[5] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM TODS*, 25(2), 2000.

[6] B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *Proc. EDBT*, pages 982–993. ACM, 2009.

[7] B. Glavic and K. Dittrich. Data Provenance: A Categorization of Existing Approaches. In *BTW*, volume 7, pages 227–241. Citeseer, 2007.

[8] T. Grust and J. Rittinger. Observing SQL Queries in their Natural Habitat. *ACM TODS*, 38(1), 2013.

[9] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *Proc. VLDB*, 2011.

[10] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 1984.