Seminar: XML-basierte Datenbanken WS 1999/2000
Leitung: Prof. Dr.-Ing. Gerhard Weikum, Dipl.-Inform. Anja Theobald
Datum: 10.12.1999

Ausarbeitung zum Thema: Query-Ausführung

von Jens Thelemann

Inhalt

1 Einleitung ____ *3*

1.1 Was sind semistrukturierte Daten? ____ **3**

2 Ansatz 1 [BF99a] ____ *3*

2.1 Das 3-Schichten-Modell ____ **3**

2.2 Formale Darstellung der Daten ____ **4**

2.2.1 Schemata ____ 5

2.2.2 Anfragetypen ____ 7

2.2.3 Verarbeitung der Anfragen mittels Constraints ____ 8

2.3 Performanz und Optimierung ____ **11**

2.3.1 Reduzierung des Suchraumes mittels Schema Containment ____ 11

2.3.2 Effizientes Lösen eines CSP ____ 11

3 Ansatz 2 [FPS97] ____ *12*

3.1 Grundlagen ____ **12**

3.1.1 Die Abfragesprache UnQL (UnCAL) ____ 12

3.1.2 Datalog ____ 12

3.2 Datenmodell ____ **13**

3.3 Erläuterung des Ansatzes anhand eines Beispiels ____ **13**

3.3.1 Die Beispielanfrage Q1 in Datalog ____ 14

3.3.2 Entfalten der Anfrage Q1 zur Eliminierung der Rekursion ____ 14

3.4 Graphenschema ____ **15**

3.5 Speicherschema ____ **16**

3.6 Konjunktive Anfragen ____ **16**

3.7 Übersetzen positiver UnQL-Anfragen in konjunktive Anfragen ____ **17**

4 Vergleich der Ansätze ____ *19*

4.1 Gemeinsamkeiten ____ **19**

4.2 Unterschiede ____ **19**

4.3 Praxistauglichkeit ____ **19**

5 Fazit ____ *20*

Literaturverzeichnis ____ *21*

1 Einleitung

In dieser Arbeit werden die Kernpunkte zweier formaler Ansätze zur Handhabung von Anfragen auf semistrukturierten Daten vorgestellt. Danach werden die beiden Ansätze miteinander verglichen und Gemeinsamkeiten und Unterschiede hervorgehoben.
Der erste Ansatz arbeitet mit Schema Matching und löst intern zur Auswertung von Anfragen Constraint Erfüllungsprobleme.
Der zweite Ansatz benutzt Speicherschemata (eine Aufspaltung der Kanten des ursprünglichen Datengraphen in Relationen) zur Speicherung semistrukturierter Daten und zur Ausführung von Suchanfragen. Es wird gezeigt, wie Anfragen, die in UnQL formuliert sind, in konjunktive Anfragen übersetzt werden, die direkt auf den Relationen des Speicherschemas arbeiten. Erklärungen erfolgen an entsprechender Stelle.

1.1 Was sind semistrukturierte Daten?

Semistrukturierte Daten sind Daten, die weder Rohdaten (zum Beispiel Bild- und Videodaten) sind, noch streng getypt. Semistrukturierte Daten verfügen i. allg. über eine unregelmäßige, teilweise implizite, teilweise explizite Struktur, die sich leicht und häufig ändern kann. XML stellt eine syntaktische Möglichkeit zur Beschreibung semistrukturierter Daten dar.

2 Ansatz 1 [BF99a]

2.1 Das 3-Schichten-Modell

Das 3-Schichten-Modell stellt die Basis für den Aufbau von semistrukturierten Datenquellen dar.

Das Modell besteht aus folgenden drei Schichten (siehe Abbildung 1)

- *Operationale Schicht*. Diese enthält Anfragen, Sichten (Views), Constraints, die auf die folgende Schicht angewendet werden.
- *Schemaschicht*. Sie ist vergleichbar mit den Tabellen in einer relationalen Datenbank.
- *Instanzschicht*. Diese steht für einzelne Ausprägungen der Schemaschicht und ist vergleichbar mit den Tupeln einer relationalen Datenbank.

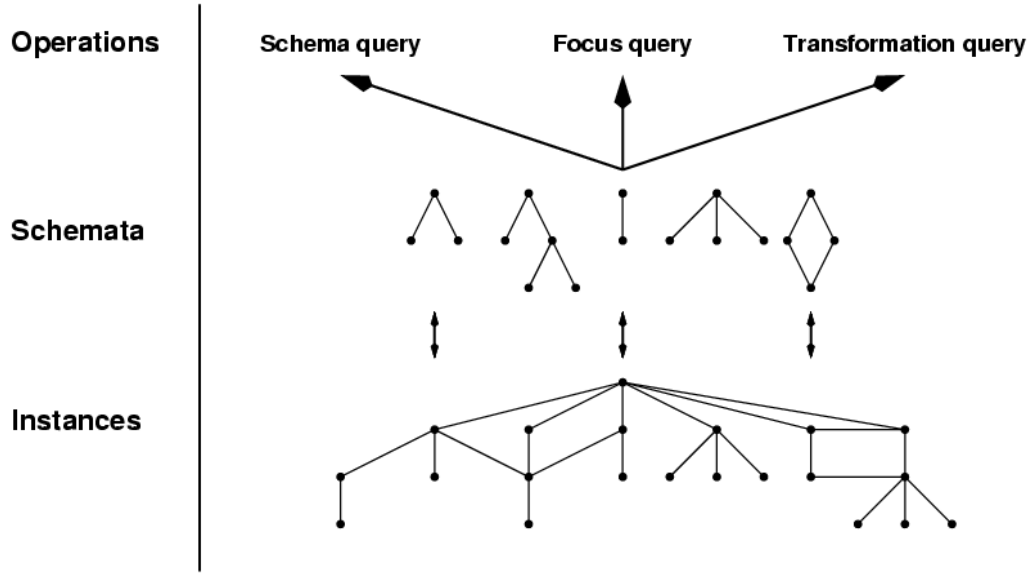

Abb. 1: Das 3-Schichten-Modell [BF99a]

Mit Hilfe dieser Veranschaulichung wird klar, dass die Herausforderung beim Umgang mit semistrukturierten Daten im „Herausfinden" der Schemaschicht liegt. Dies versucht man durch eine Analyse der Anfragen an ein System zu erreichen. Die Anfragen bestehen aus zwei Teilen:

- „Was"-Teil, der der Selektion der Anfrage entspricht
- „Wie"-Teil, der angibt, wie die selektierten Daten zu projizieren sind

Die Verarbeitung dieser implizit gelieferten Strukturinformationen ermöglicht es, den Teil der Schemainformation, den die semistrukturierten Daten enthalten, mit jeder weiteren Anfrage weiter zu modellieren.

2.2 Formale Darstellung der Daten

Datenquellen werden in Form von markierten gerichteten Graphen dargestellt, die nun definiert werden.

Definition: **Markierter gerichteter Graph**
Ein *markierter gerichteter Graph* ist ein Tupel G = (V, A, s, t, *l*) mit

- V Menge der Knoten
- A Menge der gerichteten Kanten
- s einstellige Funktion die jedem a $\in$ A einen Quellknoten v $\in$ V zuordnet
- t einstellige Funktion die jedem a $\in$ A einen Quellknoten v $\in$ V zuordnet
- *l* einstellige Markierungsfunktion *l*: V$\cup$ A $\rightarrow$ L, L Menge der Markierungen

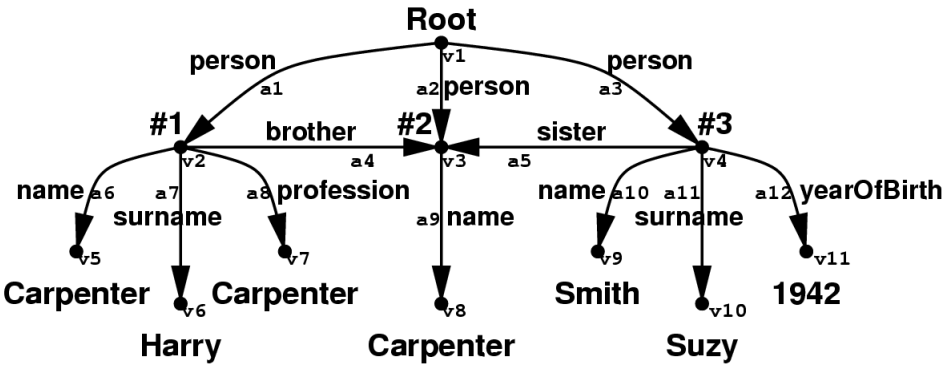

Abb. 2: Beispiel für einen Graphen [BF99a]

Abbildung 2 zeigt den Graphen einer semistrukturierten Datenquelle, die Informationen über Personen enthält. Man sieht, dass jede der drei Ausprägungen unterschiedlich ist. So hat zum Beispiel Ausprägung #2 das Attribut *‚name'* als einziges Attribut, Ausprägung #3 enthält das Geburtsjahr, Ausprägung #1 hat hingegen die Berufsbezeichnung an dieser Stelle.

2.2.1 Schemata

Die Definition der Markierungen von Knoten und Kanten wird nun erweitert, sodass Knoten und Kanten auch mit einstelligen Prädikaten markiert werden können. So können Suchanfragen ebenfalls in Form von Graphen (*Prädikatenschemata*) gestellt werden. Der Graph der Datenquelle, mit dem die Suchanfrage verglichen wird, muss bei einem Treffer die Prädikate erfüllen. Beispiele für Prädikate sind

- *isstring()*: Ist erfüllt, wenn der Graph, mit dem verglichen wird, an entsprechender Stelle eine Markierung vom Typ *String* hat.
- *isinteger()*: Das gleiche wie oben, nur mit dem Datentyp *Integer*.
- *true()*: Ist immer erfüllt, vorausgesetzt der Knoten oder die Kante ist vorhanden.
- *false()*: Ist nie erfüllt.

Definition: **Prädikatenschema**

Sei P eine Menge einstelliger Prädikate. Ein *Prädikatenschema* (über P) ist ein gerichteter Graph $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$, dessen Elemente mit Prädikaten $l : V^{(s)} \cup A^{(s)} \rightarrow P$ markiert sind.

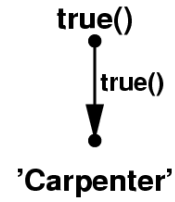

Abb. 3: Einfaches Prädikatenschema [BF99a]

Abbildung 3 zeigt ein einfaches Prädikatenschema. Es verlangt einen beliebig markierten Knoten, gefolgt von einer beliebig markierten Kante, gefolgt von einem Knoten mit der Markierung *‚Carpenter'*.

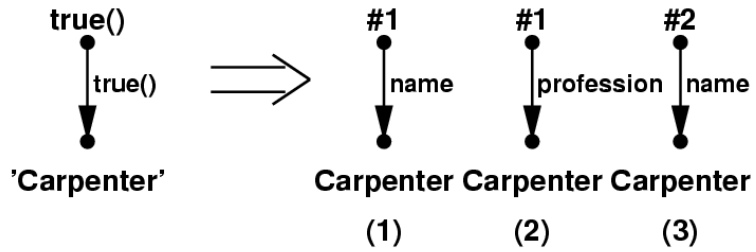

Abb. 4: Anfrage als Prädikatenschema mit Treffern [BF99a]

Abbildung 4 zeigt links die Suchanfrage in Form des Prädikatenschemas aus Abbildung 3 und rechts die drei gefundenen Treffer, die beim Vergleich des Graphen aus Abbildung 2 gefunden wurden.

2.2.1.1 Hinzufügen von Variablen:

Dem Ansatz wird nun eine partielles Abbildung der Knoten und Pfeile auf Variablen hinzugefügt:

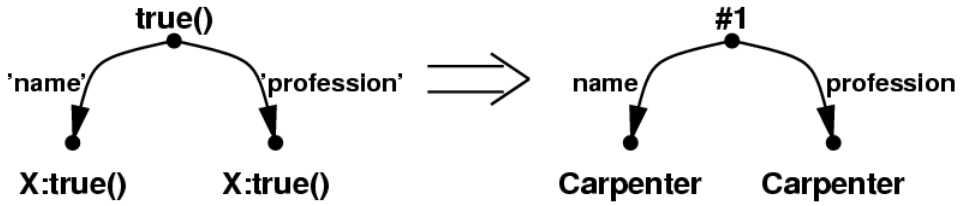

Abb. 5: Anfrage mit Variablen [BF99a]

Die Einführung von Variablen ermöglicht eine Verknüpfung von Knoten und Kanten sowie Verweise auf Strukturen.
Abbildung 5 verdeutlicht dies durch eine Verknüpfung der beiden benachbarten Knoten, die hierdurch beliebig aber gleich sein müssen. Als Treffer ergibt sich bei Anwendung auf den Graphen aus Abbildung 2 der rechte Teil der Abbildung.

2.2.1.2 Hinzufügen von Längenbeschränkungen:

Zur Einführung von Längenbeschränkungen auf Pfaden, die zum Beispiel die Suchtiefe im Graphen einschränken sollen, muss zunächst der Begriff des Spurgraphen eingeführt werden.

Der Spurgraph enthält zusätzlich zu den Kanten des ursprünglichen Graphen die Spuren (in Form von neuen Kanten) aller Pfade des ursprünglichen Graphen. Intern enthalten die Spuren die Länge der Pfade, für die sie stehen.

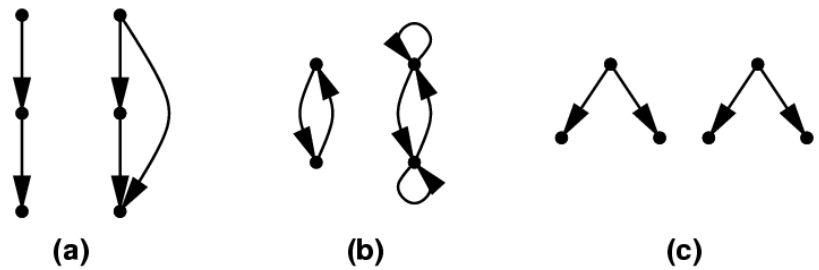

Abb. 6: Spurgraphen [BF99a]

Formal kann nun die Längenbeschränkung wie folgt eingeführt werden:

Definition: **Schema mit Längenbeschränkungen**
Das Tupel eines Graphen $G = (V^{(G)}, A^{(G)}, s^{(G)}, t^{(G)})$ wird erweitert auf $G' = (V^{(G)}, A^{(G)}, s^{(G)}, t^{(G)}, q_{\min}^{(s)}, q_{\max}^{(s)})$, wobei $q_{\min}^{(s)} : A^{(s)} \to N^+$ und $q_{\max}^{(s)} : A^{(s)} \to N^+ \cup \{+\infty\}$ *Längenbeschränkungen* sind.

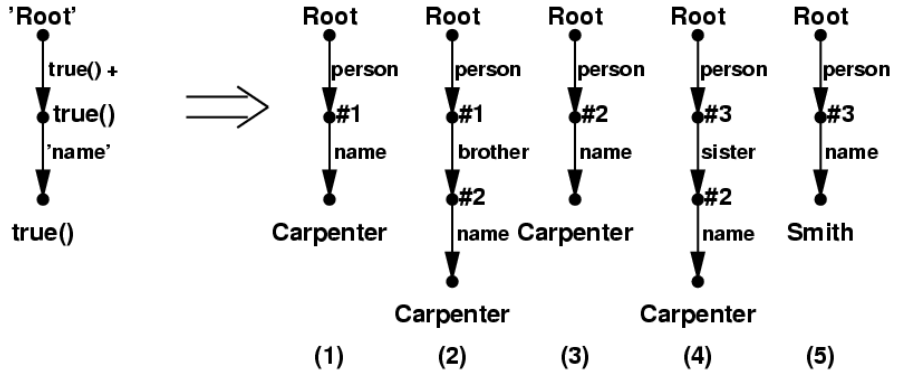

Abb. 7: Anfrage mit Pfaden [BF99a]

Abbildung 7 zeigt nun, wie sich die Längenbeschränkung des ersten Pfades auf eine Länge von 1 bis unendlich auf das Suchergebnis auswirkt.
Bei Betrachtung der fünf Treffer stellt sich die Frage, ob alle *minimale Treffer* sind (betrachte Treffer (1) und (2)). Minimale Treffer sind wichtig, weil nur sie als Antwort auf eine Suchanfrage zurückgeliefert werden sollen. Bei diesem Beispiel sind alle Treffer minimale Treffer und zwar jeweils von unterschiedlichen Teilgraphen des Graphen der Datenquelle. Daher sind sie alle relevant. Die Berechnung der minimalen Treffer erfolgt mit Hilfe einer Glättungsfunktion, die angewendet auf alle Treffer des Anfrageschemas den Treffer in die Menge der minimalen Treffer abbildet. Informell zerlegt die Glättungsfunktion die Spuren des Spurgraphen in normale Kanten und fügt alle so neu gewonnenen Quell- und Zielknoten zu den entsprechenden Mengen hinzu. Details hierzu sind in [BF99a] zu finden.

2.2.2 Anfragetypen
Das Vorangegangene bezog sich auf den „Was"-Teil einer Anfrage, der folgende Teil befasst sich mit dem „Wie"-Teil, also wie Ergebnisse von Anfragen zurückgeben werden. Es folgen verschiedene Arten von Anfragen.

2.2.2.1 Schemaanfrage
Bei einer Schemaanfrage liegt ein Schema als Anfrage vor. Die Antwort ist eine Menge von Treffern, die das Schema erfüllen (vgl. Abbildung 7).

2.2.2.2 Fokusanfrage
Bei einer Fokusanfrage liegen zwei Schemata vor. Nach dem ersten Schema wird gesucht, das zweite Schema gibt an, was zurückgegeben werden soll. Die Antwort enthält somit lediglich den Fokus, alle übrigen Informationen fallen weg.

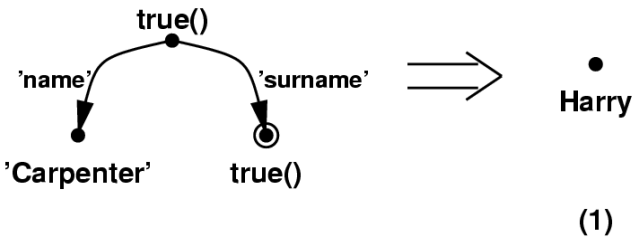

Abb. 8: Eine Fokusanfrage [BF99a]

Abbildung 8 zeigt eine einfache Fokusanfrage. Das Fokusschema wird hierbei lediglich durch eine Umkreisung des Fokus markiert (hier der rechte Knoten). Diese einfache und praktische Notation ergibt sich aus der Tatsache, dass der Fokus nur ein Teilgraph des Anfragegraphen sein kann.

2.2.2.3 Transformationsanfrage

Hier werden ein Schema *s*, nach dem gesucht wird und ein Schema *t* mitgegeben. Die Antwort wird erzeugt, indem aus jedem Treffer von *s* in der Datenquelle ein zu *t* isomorphes Schema kreiert wird, das mit den ausgewerteten Termen von *t* markiert ist, wobei zur Auswertung die Markierungen des jeweiligen Treffers als Instanz in die entsprechenden Terme von *t* eingesetzt werden.

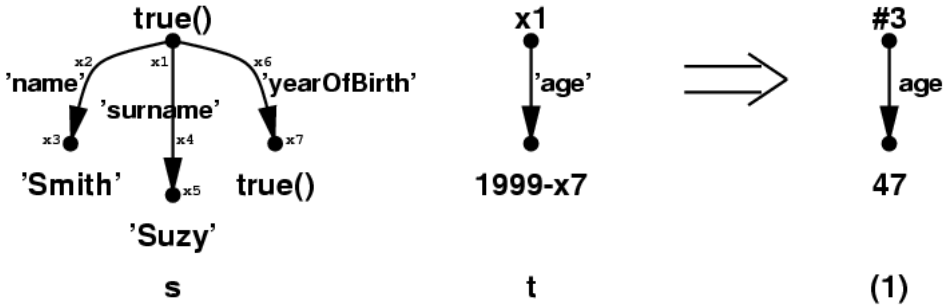

Abb. 9: Eine Transformationsanfrage [BF99a]

Abbildung 9 illustriert die Anwendung einer einfachen Transformationsanfrage auf die Datenquelle aus Abbildung 2. Die Markierung des oberen Knoten wird in x1 eingesetzt und evaluiert zu dem ursprünglichen Wert, die Markierung der Kante wird von *‚yearOfBirth'* in *‚age'* geändert und der Wert des unteren Knoten wird von der Zahl 1999 subtrahiert, wodurch sich das aktuelle Alter der gefundenen Person ergibt.

2.2.3 Verarbeitung der Anfragen mittels Constraints

Die Ausführung der Anfragen wird schließlich mit Hilfe von Constraints durchgeführt. Zunächst wird daher folgender Begriff eingeführt.

Ein *Constraint Erfüllungsproblem* (englische Abkürzung: *CSP*) ist ein Tupel (*X*, *D*, *C*) wobei

- *X* die Menge der *Variablen* $\{x_1,...,x_m\}$ ist
- *D* ist die Menge der endlichen *Domänen* D_i für jede Variable $x_i \in X$
- *C* ist die Menge der *Constraints* $\{C_{S_1},...,C_{S_n}\}$, die die Werte einschränken, die die Variablen gleichzeitig annehmen können. Hier sind die $S_i(x_{S_{i_1}},...,x_{S_{i_k}})$ beliebige Tupel

von Variablen aus X und jedes C_{S_i} ist eine Relation über dem Kreuzprodukt der Domänen dieser Variablen ($C_{S_1} \subseteq D_{S_{i_1}} \times ... \times D_{S_{i_k}}$).

Zur Lösung eines CSP wird eine Variablenbelegung aus den Domänen herausgesucht, die alle Constraints erfüllt. Man beachte, dass zur korrekten Beantwortung einer Anfrage alle möglichen Lösungen gefunden werden müssen.

Der Graph der Datenquelle wird in Domänen unterteilt und für die Elemente des Schemas werden Variablen eingeführt. Die Semantiken der Treffer werden durch zwei Kategorien von Constraints dargestellt. Die eine beschreibt die Markierungen, die andere beschreibt den strukturellen Teil der Matchsemantiken.

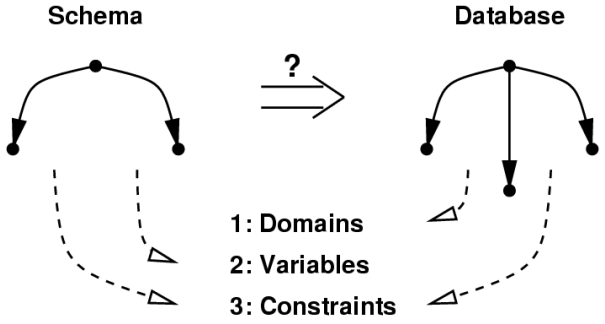

Abb. 10: Schema Matching durch Constrainterfüllung [BF99a]

Beispiel [BF99a]:
Der Graph aus Abbildung 2 wird hier zur Erhöhung der Übersichtlichkeit wieder eingefügt. Auf ihn bezieht sich das Beispiel.

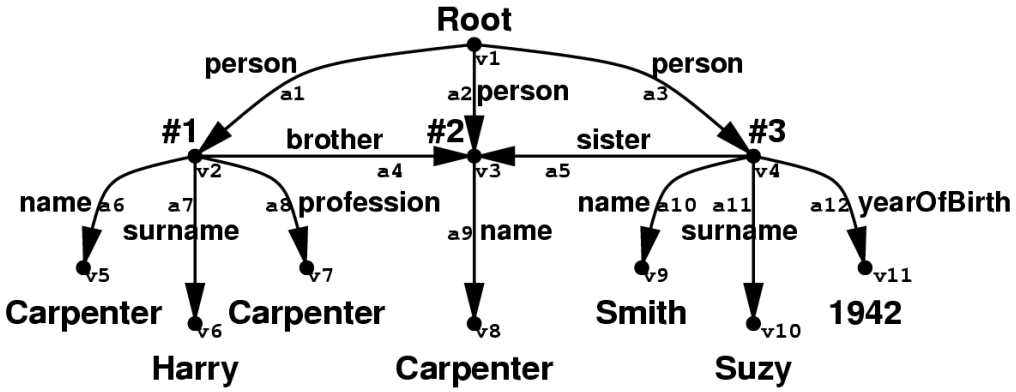

Abb. 11: Beispielgraph [BF99a]

Es werden zunächst Domänen für die Pfeile und die Knoten eingeführt.

$$D_V = \{v_1, v_2, v_3, ..., v_{11}\}$$
$$D_A = \{a_1, a_2, a_3, ..., a_{12}\}$$

Die folgende Abbildung eines einfachen Prädikatenschemas liefert die Zuordnung der Variablen und der Domänen für dieses Beispiel.

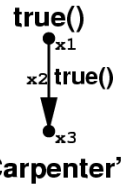

Abb. 12: Ein einfaches Prädikatenschema [BF99a]

$$X = \{x_1, x_2, x_3\}$$
$$D_1 = D_3 = D_V$$
$$D_2 = D_A$$

Die Constraints leitet man nach dem folgenden Schema ab. Zunächst von den Markierungen der Knoten und Kanten für die jeweiligen Variablen

$$C_{x_1}^{lab} = \{(v_1),(v_2),(v_3),...,(v_{11})\}$$
$$C_{x_2}^{lab} = \{(a_1),(a_2),(a_3),...,(a_{12})\}$$
$$C_{x_3}^{lab} = \{(v_5),(v_7),(v_8)\}$$

und dann von der Struktur des Schemas, definiert durch die Quellen und Ziele. Die Paare in der Schreibweise unten können wie folgt in die schon bekannten Funktionen

$s(a_i) = v_j$ bzw. $t(a_k) = v_l$

eingesetzt werden.

$$\begin{aligned} C_{x_2,x_1}^{src} = \{&(a_1,v_1),(a_2,v_1),(a_3,v_1),(a_4,v_2),\\ &(a_5,v_4),(a_6,v_2),(a_7,v_2),(a_8,v_2),\\ &(a_9,v_3)(a_{10},v_4)(a_{11},v_4)(a_{12},v_4)\} \end{aligned}$$
$$\begin{aligned} C_{x_2,x_3}^{tar} = \{&(a_1,v_2),(a_2,v_3),(a_3,v_4),(a_4,v_3),\\ &(a_5,v_3),(a_6,v_5),(a_7,v_6),(a_8,v_7),\\ &(a_9,v_8)(a_{10},v_9)(a_{11},v_{10})(a_{12},v_{11})\} \end{aligned}$$

Die Lösungen des CSP dieses Beispiels sind (v_2, a_6, v_5), (v_2, a_8, v_7), (v_3, a_9, v_8) für die Variablen (x_1, x_2, x_3). Bei diesem einfachen Beispiel kann man zu der Lösung durch „scharfes Hinsehen“ kommen. Da eine naive schematische Lösung des Problems exponentielle Laufzeit hat, liegt gerade hier der Hauptansatzpunkt für Optimierungen.

Eine Möglichkeit der Laufzeitverbesserung ist zum Beispiel die Wiederverwendung schon verarbeiteter Anfragen, wie beschrieben im folgenden Kapitel 2.3.1.

2.3 Performanz und Optimierung

2.3.1 Reduzierung des Suchraumes mittels Schema Containment (Wiederverwendung schon gewonnener Ergebnisse)

Definition Schema Containment:
Ein Schema s_1 enthält ein Schema s_2 wenn für alle Datenbanken o alle Treffer von s_2 auch Treffer von s_1 sind.
Wenn s_2 in s_1 enthalten ist,

- sind alle Treffer von s_2 auch Treffer von s_1. Wenn man die Treffer von s_1 finden will und schon die von s_2 hat, kann man *die ersten paar Treffer sofort* präsentieren.
- können Treffer von s_2 nur unter Treffern von s_1 gefunden werden. Wenn man die Treffer von s_2 finden will und schon die für s_1 haben, kann man *den Suchraum reduzieren.*

Die folgende Abbildung zeigt Schema Containment und zwar paarweise von links nach rechts. Jedes neue Schema stellt eine Spezialisierung des Vorangegangenen dar, wodurch weniger Treffer möglich sind, die zudem nur eine Teilemenge der Treffer des vorangegangenen (allgemeineren) Schemas sein können.

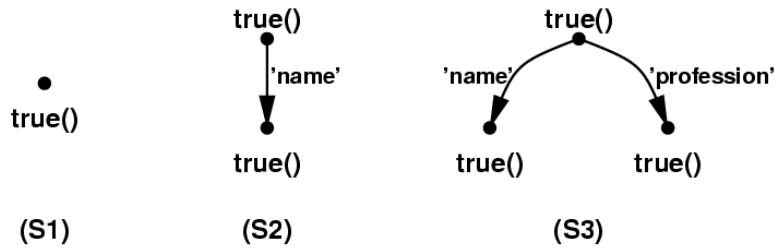

Abb. 13: Schema Containment [BF99a]

Wenn man also zum Beispiel alle Treffer zu S2 schon hat, so muss man zur Ausführung der Anfrage S3 nur noch diese Treffer durchsuchen und nicht mehr den gesamten Graphen der Datenquelle.

2.3.2 Effizientes Lösen eines CSP

In [BF99b] werden vier Ansatzpunkte für Optimierungen genannt. Diese sind

1.) Injektivität der Lösung nicht durch zusätzliche Constraints, sondern durch einen zusätzlichen Test sichern.
2.) Die Labelconstraints als Domänen benutzen (statische Domänenreduzierung).
3.) Variablen nach aufsteigenden Domänen und nach fallender Anzahl von Constraints sortieren.
4.) Dynamische Domänenreduzierung durchführen abhängig von dem gerade instantiierten Knoten bzw. der Kante.

Details und entsprechende Verweise können aus [BF99b] entnommen werden.

3 Ansatz 2 [FPS97]

Wie der erste beschriebene Ansatz benutzt auch dieser markierte Graphen zur Darstellung von Datenquellen. Allerdings arbeitet dieser Ansatz nicht direkt auf den Graphen, sondern leitet aus ihnen zunächst ein *Speicherschema* ab, das die Kanten des ursprünglichen Graphen in Relationen aufteilt, die teilweise mit Markierungen bekannten Typs versehen sind, teilweise aber auch dynamisch getypt sind.
Zulässig sind dann *UnQL-Anfragen*, die in *konjunktive Anfragen* übersetzt werden, die direkt auf den Relationen des Speicherschemas arbeiten.

3.1 Grundlagen

3.1.1 Die Abfragesprache UnQL (UnCAL)

UnCAL ist das zu UnQL äquivalente Kalkül mit dem hier hauptsächlich gearbeitet wird. UnCAL eignet sich zur Beschreibung von Graphen und unterstützt Vereinigung, Konkatenation, das Konditional if und das gext-Konstrukt, eine einfache Form der Rekursion auf Bäumen, das bei der Übersetzung positiver UnQL-Anfragen in konjunktive Anfragen benötigt wird. Ein Beispiel hierzu ist in Kapitel 3.7 zu finden. In UnQL werden bei diesem Ansatz die Anfragen formuliert. UnCAL wird benutzt, um die Anfrage auf das gegebene Speicherschema anzuwenden und so das Speicherschema zu finden, dessen Relationen die Anfrage in Form von konjunktiven Anfragen ermöglichen (Kapitel 3.7).

3.1.2 Datalog

Datalog ist eine einfache nicht prozedurale Sprache zur Beschreibung und Verarbeitung großer persistenter Sammlungen von Fakten und Regeln.
Ein Datalog-Programm besteht aus

- einer Menge von Fakten der Form $P_i(V_1,V_2,...,V_k)$ mit einem *k*-stelligen Prädikat P_i und Konstanten $V_1,V_2,...,V_k$ und
- einer Menge von Regeln in Form von Hornklauseln (Klauseln mit nur einem positiven Literal) über den gegebenen Prädikaten

Beispiel [Wei99]:
Fakten:

Flüge(Frankfurt, Chicago)
Flüge(Chicago, Phoenix)
...

Regeln:

Flüge(a,z) $\Rightarrow$ Flugverbindungen(a,z)
Flugverbindungen(a,o) $\wedge$ Flüge(o,z) $\Rightarrow$ Flugverbindungen(a,z)

Anfrage:

Flugverbindungen(Frankfurt, x)

Das Beispiel zeigt eine Sammlung von „atomaren“ Flügen, aus denen man mit Hilfe der Regeln Flugverbindungen errechnen kann. Die beispielhafte Anfrage sucht nach allen von Frankfurt aus möglichen Flugverbindungen.

Datalog wird bei diesem Ansatz dazu verwendet, die konjunktiven Anfragen über den Relationen des Speicherschemas zu formulieren.

3.2 Datenmodell

Wie in Ansatz 1 wird die Datenquelle dargestellt als Graph mit markierten gerichteten Kanten. Knoten sind hier allerdings nicht markiert. Dadurch gibt es keine direkte Unterscheidung zwischen Attributbezeichnung und Wert des Attributs. Bezeichnung und Wert sind einfach aufeinanderfolgende Kanten.
Weitere Merkmale des Datenmodells sind

- Der Graph hat genau eine Wurzel
- Markierungen sind Elemente der Vereinigungsmenge der atomaren Datentypen (String, Int, Bool, ...)
- Der Graph wird gesehen als eine Menge von Markierungs-Graph-Paaren (rekursive Definition)

Beispiel:
$t = \{a \Rightarrow \{c, d \Rightarrow \{e\}\}, b \Rightarrow \{f\}\}$ (a)
$t' = \{a \Rightarrow \{b\} \cup T\}$ mit $T = \{c \Rightarrow T\}$ (b)
$t \cup t'$ (c)

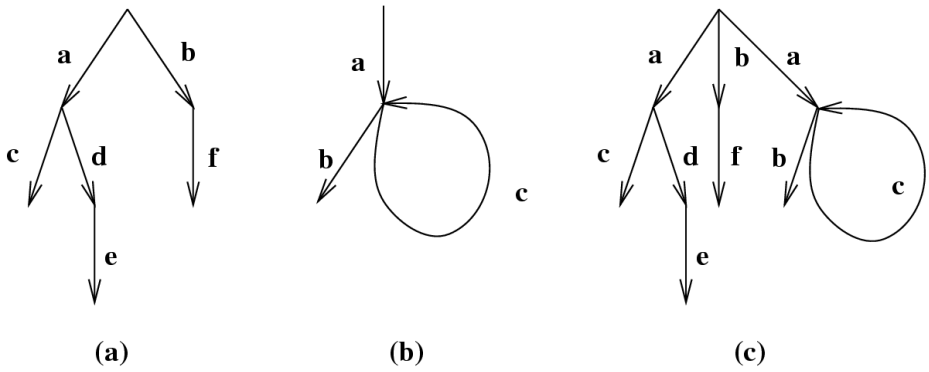

Abb. 14: Beispielgraphen [FPS97]

Das Beispiel zeigt in (a) einen einfachen Graphen, in (b) einen Graphen mit Zyklus und in (c) die Vereinigung der Graphen aus (a) und (b).

3.3 Erläuterung des Ansatzes anhand eines Beispiels

Die Anfrage wird zunächst in UnQL formuliert.

$Q1$: $select\{x \Rightarrow \{y \Rightarrow t\}\}$
$where * \Rightarrow x \Rightarrow y \Rightarrow t\ in\ DB,$
$y\ matcht\ "Security"$

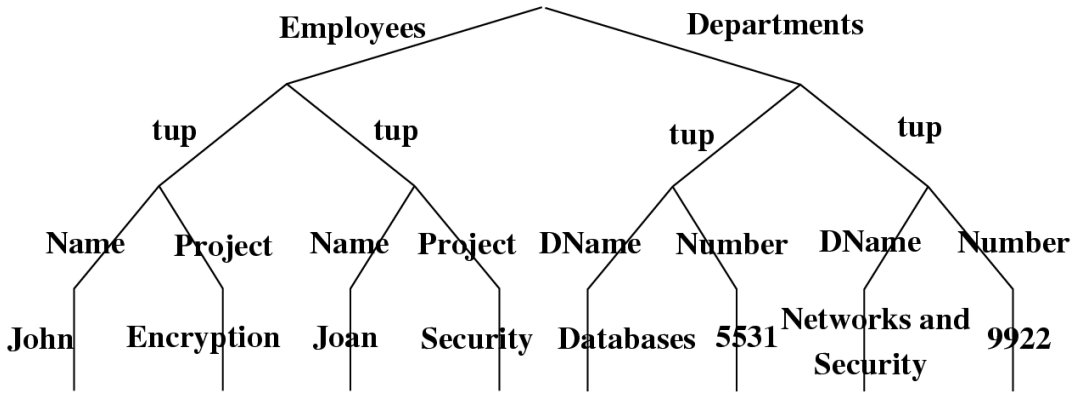

Abb. 15: Graph der Beispieldatenquelle [FPS97]

Abbildung 16 zeigt den Graphen der Datenquelle, auf den die Anfrage geschickt wird. Die Auswertung erfolgt nun zunächst durch Umformulierung der Anfrage in Datalog.

3.3.1 Die Beispielanfrage Q1 in Datalog

Die folgende Schreibweise arbeitet mit dreistelligen Relationen der Form E(Oid, Label, Oid) und der einstelligen Relation Root(Oid), wobei Oid ein eindeutiger interner Knotenidentifier ist und Label eine Kantenmarkierung.
Die „direkte“ Übersetzung in Datalog ergibt ein rekursives Programm. Die erste Zeile beschreibt die Wurzel, die zweite eine Kante von einem Knoten zu einem anderen. Die dritte Zeile beschreibt die Konstruktion der Antwort ab dem Wurzelknoten. Zeile vier beschreibt den Rekursionsschritt mit dessen Hilfe die Suche nach Treffern durch den gesamten Graphen hindurch fortgesetzt wird.

```
C(r)            :- Root(r)
C(n')           :- C(n), E(n,_,n')
Antw(r,x,n')    :- Root(r), C(n), E(n,x,n'), E(n',y,n''), matcht(y,"Security")
Antw(n',y,n'' ) :- C(n), E(n,x,n'), E(n',y,n''), matcht(y,"Security")
```

3.3.2 Entfalten der Anfrage Q1 zur Eliminierung der Rekursion

Im folgenden wird die Anfrage Q1 in eine Version ohne Rekursion umgeformt (entfaltet) und zwar für jede der drei Graphenebenen einzeln. Dies ist natürlich nur möglich, weil die Tiefe des Graphen bekannt ist.

Antwort der ersten Graphenebene:

```
Antw(r,x,n')    :- Root(r),
                   E(r,x,n'), E(n',y,n''), matcht(y,"Security")
Antw(n',y,n'')  :- Root(r),
                   E(r,x,n'), E(n',y,n''), matcht(y,"Security")
```

Antwort der zweiten Graphenebene:

```
Antw(r,x,n')    :- Root(r), E(r,_,n),
                   E(r,x,n'), E(n',y,n''), matcht(y,"Security")
Antw(n',y,n'')  :- Root(r), E(r,_,n),
                   E(r,x,n'), E(n',y,n''), matcht(y,"Security")
```

Antwort der dritten Graphenebene:

Antw(r,x,n') :- Root(r), E(r,_,n_1), E(n_1,_,n),
E(r,x,n'), E(n',y,n''), matcht(y,"Security")

Antw(n',y,n'') :- Root(r), E(r,_,n_1), E(n_1,_,n),
E(r,x,n'), E(n',y,n''), matcht(y,"Security")

Die Antworten für die Ebenen eins und zwei sind leer. Dennoch kann man sie nicht sofort wegfallen lassen, da man die Datentypen der Kanten nicht kennt. Das in Kapitel 3.4 eingeführte Graphenschema hebt diesen Nachteil auf.

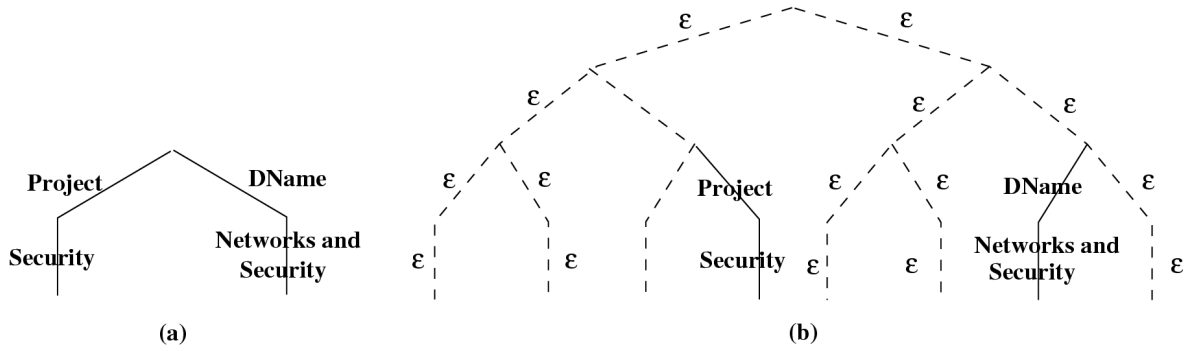

Abb. 16: Antwort auf Anfrage Q1 [FPS97]

Abbildung 16 (a) zeigt den errechneten Treffer. (b) zeigt das Ergebnis mit ε-Übergängen wie es bei korrekter Auswertung der Datalogprogramme entsteht. Intuitiv repräsentieren diese Überbleibsel des Graphen, an den die Anfrage gestellt wurde. Sie markieren die Teile des Graphen, die nicht Teil des oder der Treffer sind. Bei Zykelfreiheit können die ε-Übergänge durch Bildung der transitiven Hülle des Ergebnisses eliminiert werden.

3.4 Graphenschema

Beim Graphenschema sind ausgehend vom Graphen der Datenquelle die Pfade um eine Kante erweitert, die als Markierung den Datentypen der vorangegangenen Kante hat (vgl. Abbildung 17 (a), Graphenschema der Datenquelle). Dadurch ist nun eine eindeutige Typisierung sofort ablesbar. Der Nachteil des Graphenschemas ist allerdings ein Verlust der Schemainformation der Datenquelle, was in Abbildung 17 (b) sichtbar wird, die das Graphenschema der Antwort auf Q1 zeigt.

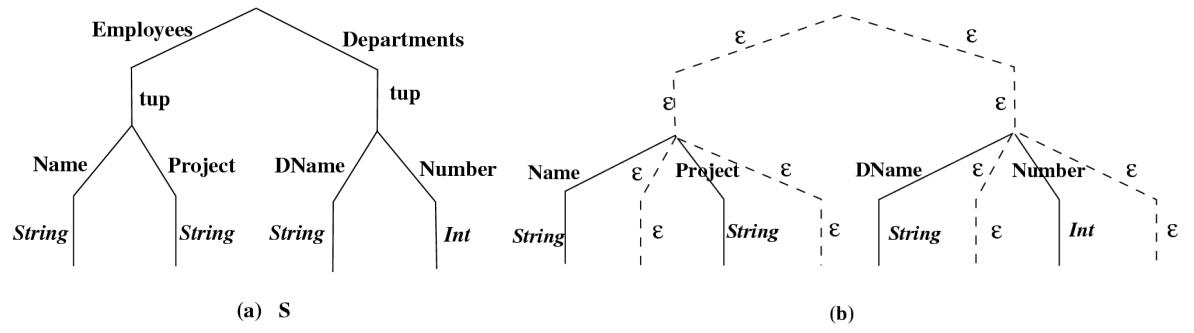

Abb. 17: (a) Graphenschema der Datenquelle und (b) der Antwort [FPS97]

3.5 Speicherschema

Das Speicherschema ist eine Erweiterung des Graphenschemas. Das Speicherschema führt für jede Kante des Graphenschemas eine dreistellige Relation der Form $\{s \times b \times s'\}$ mit *s*, *b*, *s'* sind atomare Typen, ein und definiert so eindeutig die Menge aller Datenkanten. Die Wurzel eines Speicherschemas ist per Definition der leere Knoten $\langle\ \rangle$. ε-Übergänge sind erlaubt. Ihre Relationenschreibweise ist $\{s \times \varepsilon \times s'\}$.

Im Gegensatz zum Graphenschema gehen beim Speicherschema die Schemainformationen des ursprünglichen Graphen der Datenquelle nicht verloren. Die Kanten des ursprünglichen Graphen erhält man aus der disjunkten Vereinigung aller Relationen des Speicherschemas. Die disjunkte Vereinigung aller Relationen aus Abbildung 18 (b) würde zum Beispiel Q_3 und Q_3', Q_4 und Q_4' usw. wieder zusammenfassen, wodurch das Ausgangsschema wieder hergestellt wäre.

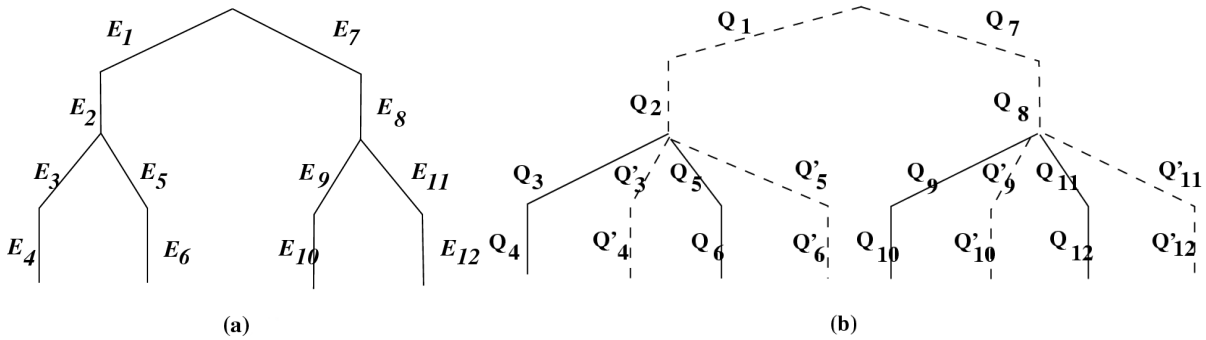

Abb. 18: (a) Speicherschema der Datenquelle und (b) der Antwort [FPS97]

Abbildung 18 (a) zeigt das Speicherschema der Datenquelle aus Abbildung 15, Abbildung 18 (b) zeigt das Speicherschema der Antwort auf Q1.

3.6 Konjunktive Anfragen

Der hier verwendete Formalismus CONJ stellt eine Mengeschreibweise zur Verfügung mit deren Hilfe syntaktische Beziehungen zwischen Relationen beschrieben werden können. Verwendet wird CONJ zur Formulierung der konjunktiven Anfragen auf den Relationen eines Speicherschemas. Ein einfaches Beispiel für CONJ ist

$$\{\langle x, y\rangle \mid \langle x, v\rangle \in R, \langle w, y\rangle \in R', v = w\}$$

Es berechnet die Join-Anfrage zwischen den Relationen *R* und *R'*.

$\langle x, y\rangle, \langle x, v\rangle$ und $\langle w, y\rangle$ sind Beispiele für skalare Ausdrücke, diese können bestehen aus

- Tupeln von Skalaren
- Projektionen von Skalaren
- skalaren Variablen
- Konstanten eines Typs der Vereinigungsmenge der atomaren Typen (Int, String, ε, ...)

Die folgenden Ausdrücke beschreiben jede der Relationen des Speicherschemas aus Abbildung 18 (b) als konjunktive Anfrage über den Relationen des Speicherschemas aus Abbildung 18 (a). Zur Erinnerung: 18 (a) zeigt das Speicherschema der Datenquelle, 18 (b) zeigt das Speicherschema der Antwort auf Q1 angewendet auf die Datenquelle.

$$
\begin{aligned}
Q_3(n,x,n') &:- E_3(n,x,n'), E_4(n',y,n''), matcht(y,"Security") \\
Q_5(n,x,n') &:- E_3(n,x,n'), E_6(n',y,n''), matcht(y,"Security") \\
Q_9(n,x,n') &:- E_3(n,x,n'), E_{10}(n',y,n''), matcht(y,"Security") \\
Q_{11} &:- \varnothing \\
Q_i(n',y,n'') &:- E_i(n',y,n''), i = 4,6,10,12 \\
Q_i(n,\varepsilon,n') &:- E_3(n,_,n'), i = 1,2,7,8 \\
Q'_i(n,\varepsilon,n') &:- E_3(n,_,n'), i = 3,4,5,6,9,10,11,12
\end{aligned}
$$

Bei Q_{11} konnten die aus dem Speicherschema bekannten Typinformationen benutzt werden, um eine Optimierung vorzunehmen. Da Q_{11} nicht den richtigen Datentyp hat, um die Anfrage beantworten zu können, konnte die konjunktive Anfrage zu $\varnothing$ vereinfacht werden.

3.7 Übersetzen positiver UnQL-Anfragen in konjunktive Anfragen

Es wird nun an einem Beispiel beschrieben, wie eine rekursive Anfrage in ein Speicherschema übersetzt wird, mit dessen Relationen die ursprüngliche Anfrage durch konjunktive Anfragen auf Relationen des Speicherschemas ersetzt werden kann.

Wir gehen direkt von einer in UnCAL formulierten Anfrage aus, die äquivalent zu einer in UnQL formulierten Anfrage ist. Sei

$Q2(DB) = g_1(DB)$ mit

$$
\begin{aligned}
g_1(\{\}) &= \{\} \\
g_1(\{l \Rightarrow t\}) &= \{l \Rightarrow \{l \Rightarrow g_1(t)\}\} \\
g_1(t_1 \cup t_2) &= g_1(t_1) \cup g_1(t_2)
\end{aligned}
$$

Der formale UnCAL-Ausdruck mit korrekt formuliertem gext-Ausdruck lautet:

$$Q2(DB = (gext_Z(\lambda(l,t).(Z := \{l \Rightarrow \{l \Rightarrow Z\}\})))(DB)$$

Bei kurzer Betrachtung des Ausdrucks wird klar, dass dieser eine Verdoppelung der Kanten vornimmt. Es folgt die Anwendung von Q2 auf den Speichergraphen von DB aus Abbildung 19 (a). Der interessante Teil, nämlich die Anwendung auf den Zyklus mit der Markierung E2 wird gesondert hervorgehoben. E2 sei nun bevölkert von den sechs Kanten aus Abbildung 19 (b), also

$$E2 = \{\langle x,a,u\rangle, \langle x,b,v\rangle, \langle v,c,w\rangle, \langle z,d,w\rangle, \langle z,e,p\rangle, \langle z,f,q\rangle\}$$

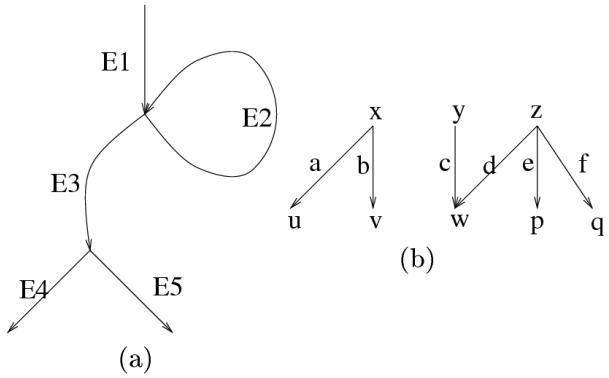

Abb. 19: (a) Ein Speicherschema und (b) sechs Kanten, die E2 bevölkern [FPS97]

Nach Anwendung von Q2 sind die Kanten verdoppelt, wie in Abbildung 20 (a) gezeigt.

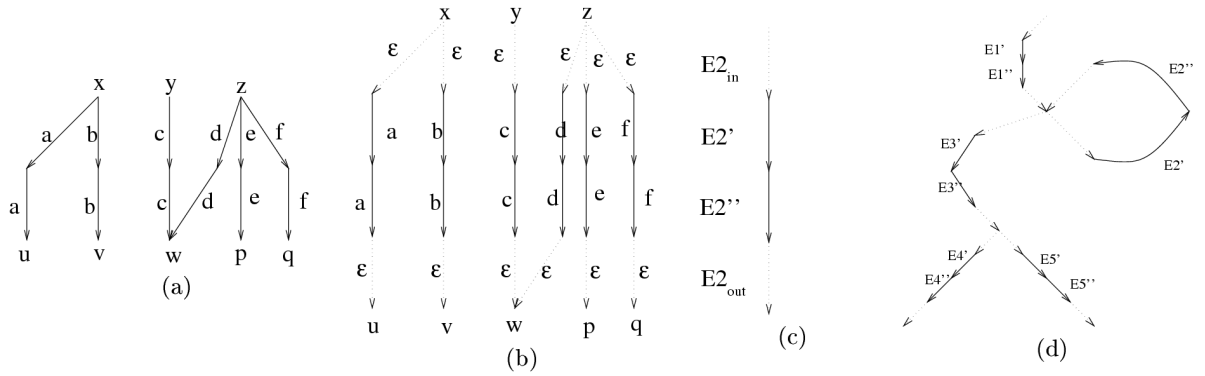

Abb. 20: (a) und (b) Ergebnisse der Anfrage Q2. (c) Speicherschema des Ergebnisses beschränkt auf E2. (d) Vollständiges Speicherschema des Ergebnisses von der Anfrage Q2 [FPS97]

Dieser Teilgraph enthält nun sechs neue Zwischenknoten, für die neue Oids eingeführt werden müssen. Da die Knoten zu den sechs Kanten von E2 gehören, werden sie nach ihren Kanten benannt. Daraus ergibt sich für die neuen Knoten folgende Tripelschreibweise

$$\langle n,l,n'\rangle \in E2$$

Da in E2 allerdings noch atomare Werte wie x, y, z, u, ... vorhanden sind, werden zur Vereinheitlichung ε-Übergänge, wie in Abbildung 20 (b) illustriert, hinzugefügt. Nun können alle internen Knoten einheitlich mit der Tripelschreibweise beschrieben werden. Die hereinkommenden und die herausgehenden Knoten behalten die Bezeichnungen des ursprünglichen Graphen. Abbildung 20 (c) beschreibt diesen Teil des Speicherschemas. $E2_{in}$ und $E2_{out}$ beinhalten die ein- und ausgehenden ε-Übergänge, E2' und E2'' speichern die internen Kanten. Es ergeben sich folgende vier Ausdrücke für die neuen Relationen

$$E2_{in} = \{\langle n,\varepsilon,\langle n,l,n'\rangle\rangle \mid \langle n,l,n'\rangle \in E2\}$$

$$E2_{out} = \{\langle\langle n,l,n'\rangle,\varepsilon,n'\rangle \mid \langle n,l,n'\rangle \in E2\}$$

$$E2' = E2'' = \{\langle\langle n,l,n'\rangle,l,\langle n,l,n'\rangle\rangle \mid \langle n,l,n'\rangle \in E2\}$$

Insgesamt erhält man das Speicherschema aus Abbildung 20 (d) [FPS97].

4 Vergleich der Ansätze

4.1 Gemeinsamkeiten

Generell sind sich die beiden Ansätze sehr ähnlich, hinsichtlich der Darstellung der Daten in markierten Graphen.
Ansatz 1 benutzt die Graphen um aus Ihnen sehr direkt die Mengen von Variablen, Domänen und Constraints für ein CSP abzuleiten, das die Antwort auf eine Anfrage berechnet.
Bei Ansatz 1 werden Anfragen direkt in Form eines Schemas als Graphen gestellt, Ansatz 2 erwartet Anfragen in UnQL, was im Endeffekt auch nichts anderes als eine Schemabeschreibung ist.
Beide Ansätze arbeiten vorrangig auf den Strukturen der zugrundeliegenden Daten und erst zweitrangig auf den Inhalten der Daten. Zum Vergleich: Herkömmliche Suchmaschinen im WWW arbeiten hingegen ausschließlich auf den Inhalten ohne Berücksichtigung einer eventuell vorhandenen Struktur.
Ein weiteres gemeinsames Merkmal ist die Fähigkeit beider Ansätze die Ergebnisse schon bearbeiteter Anfragen weiterzuverwenden, zum Beispiel zur schnelleren Beantwortung weiterer Anfragen.

4.2 Unterschiede

Trotz der Ähnlichkeit der Datenmodelle benutzen beide Ansätze recht unterschiedliche algorithmische Herangehensweisen zur Berechnung von Antworten auf Anfragen.
Ansatz 1 arbeitet mit Constraints zur Suche nach Treffern, Ansatz 2 arbeitet mit konjunktiven Anfragen auf den Speicherschemata der Datenquellen.
Nichtsdestotrotz leisten beide Herangehensweisen prinzipiell dasselbe und zwar die strukturbasierte Suche nach Antworten auf eine Anfrage, gestellt an eine semistrukturierte Datenquelle.

4.3 Praxistauglichkeit

Ansatz 1 wurde zunächst auf einem Prolog basierten System teilweise implementiert. Die Entwicklung wurde umgestellt auf das kommerzielle Constraint Lösungssystem ECLiPSe. Die Praxistauglichkeit des Ansatzes hängt zum einen vom Erfolg der in 2.3.2 aufgezählten Optimierungsansätze und zum anderen von der Performanz von ECLiPSe ab, denn die größten Zweifel an der Einsetzbarkeit einer derartigen Lösung in der Praxis werfen die exponentiellen Laufzeiten von CSPs auf.

Die Implementierung von Ansatz 2 ist laut [FPS97] nicht abgeschlossen. Es existiert ein UnQL-Interpreter, implementiert in ML, ein ebenfalls in ML geschriebener Übersetzer, der die UnQL-Anfragen in konjunktive Anfragen übersetzt sollte folgen. Außerdem sind/waren noch Optimierungsarbeiten an dem Übersetzungsverfahren nötig, da zum Stand der Veröffentlichung von [FPS97] noch viele redundante Ausdrücke und unnötige Joins berechnet wurden.
Auch wenn Ansatz 2 noch in einem frühen Entwicklungsstadium zu sein scheint, so sind schon Erkenntnisse aus diesem Ansatz in die Entwicklung der XML Anfragesprache XML-QL eingeflossen, wie aus [FFL+98] zu entnehmen ist.

5 Fazit

Da beide Ansätze sich von der Herangehensweise an das Problem der Beantwortung von Anfragen an semistrukturierte Datenquellen sehr ähneln und Erkenntnisse aus Ansatz 2 schon in die Entwicklung von XML-QL eingeflossen sind, ist zu vermuten, dass auf dieser theoretischen Basis eines strukturbasierten Ansatzes weiterentwickelt wird. Welches algorithmische Verfahren im Endeffekt vorrangig eingesetzt wird, hängt letztendlich von der jeweiligen Geschwindigkeit und Flexibilität des Ansatzes ab.

Literaturverzeichnis

[BF99a] **Bergholz, A., Freytag, J. C.** *Querying Semistructured Data based on Schema Matching*. Workshop on Database Programming Language (DBPL), Kinloch Rannoch, September 1999.

[BF99b] **Bergholz, A., Freytag, J. C.** *Matching Schemata by utilizing Constraint Satisfaction Techniques*. Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats (in conjunction with ICDT'99), Jerusalem, Januar 1999.

[Ecl] ECLiPSe – The ECRC Constraint Logic Parallel System, http://www.ecrc.de/eclipse, Zugriff 10.12.1999.

[FFL+98] **Fernandez, M., Florescu, D., Levy, A., Suciu, D., Deutsch, A.** *XML-QL: A Query Language for XML* World Wide Web Consortium Technical Report, 1998, http://www.w3.org/TR/NOTE-xml-ql/, Zugriff: 10.12.1999.

[FPS97] **Fernandez, M., Popa, L., Suciu, D.** *A structure-based approach to querying semistructured data*. Workshop on Database Programming Language (DBPL), 1997.

[Wei99] **Weikum, G.** *Datenbanksysteme (WS 1999/2000) – Skriptum zur Vorlesung*. Universität des Saarlandes, Fachbereich 14 Informatik, Lehrstuhl Prof. Dr.-Ing. Gerhard Weikum, 1999, http://paris.cs.uni-sb.de/dbs99/, Zugriff 10.12.1999.