Anfragen an Ontologien in relationalen Datenbanken

Silke Trißl
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin
trissl@informatik.hu-berlin.de

Abstract

Ontologien und Taxonomien sind kontrollierte, strukturierte Vokabulare für eine Domäne um Objekte darin zu beschreiben. Ontologien in der Biologie, wie beispielsweise die NCBI Taxonomie, die Klassifizierung von Lebewesen, oder die Gene Ontology sind als Bäume bzw. gerichtete, azyklische Graphen aufgebaut. Taxonomien werden häufig zusammen mit den annotierten Objekten in relationalen Datenbanken gespeichert.

Durch die Struktur der Ontologien bzw. Taxonomien können Beziehungen zwischen Objekten angefragt und erkannt werden. Diese Anfragen innerhalb eines relationalen Datenbanksystems auszuführen setzt entweder die Verwendung von rekursiven Funktionen oder die Indizierung des Graphen voraus. Ich stelle 2 verschiedene Indizierungsmöglichkeiten vor, die transitive Hülle und Pre- und Postorder Ranking, und vergleiche die Anfragezeiten für beide Indexstrukturen mit Anfragezeiten für die rekursive Funktion.

1 Einleitung und Motivation

Ontologien und Taxonomien spielen eine wichtige Rolle in der Biologie und Medizin. Die vielleicht älteste Taxonomie in der Biologie ist die Klassifizierung von Flora und Fauna. In ihr spiegelt sich die evolutionäre Abstammung von verschiedenen Organismen wieder. Die NCBI Taxonomy ist als Baum mit gerichteten Kanten aufgebaut [1]. In den letzten Jahren wurden weitere Ontologien entwickelt um biologische Objekte und Vorgänge zu beschreiben, beispielsweise die Gene Ontology [2]. Sie stellt ein strukturiertes Vokabular zur Verfügung, um Proteine in ihren Funktionen und Wirkorten zu beschreiben. Die Gene Ontology (GO) ist ein gerichteter, azyklischer Graph (directed acyclic graph, DAG) mit einem Wurzelknoten.

Betrachtet man folgende Fragestellung 'Zeige mir alle biologischen Proben, die laut NCBI Taxonomy Eukaryoten sind?', so erwartet ein Biologe nicht nur die Proben, die direkt mit dem Konzept 'Eukaryot' beschrieben sind, sondern auch Proben von Menschen, Mäusen, Pantoffeltierchen und weiteren Eukaryoten.

Diese Frage in einer relationalen Datenbank zu beantworten ist mit Standard-SQL nicht möglich, da nicht nur nach Kindern eines Knotens gesucht wird, sondern auch nach deren Nachfahren. Da diese beliebig weit vom Startknoten entfernt sein können, ist es nur möglich die Frage mit Hilfe einer rekursiven Funktion zu beantworten.

Ein großer Nachteil der rekursiven Funktion ist die Abhängigkeit der Antwortzeit von der Anzahl der traversierten Knoten. Je nach Größe des Ergebnissets kann dies einige Zeit dauern. Eine Alternative dazu ist einen vorberechneten Index in nahezu konstanter Zeit anzufragen. Indizes benötigen allerdings Speicherplatz und Zeit für die Vorberechnung, was viele Indexstrukturen für große Ontologien ungeeignet macht.

In dieser Arbeit präsentiere ich eine Indexstruktur für Bäume und gerichtete azyklische Graphen, die Anfragen in konstanter Zeit beantwortet, aber wesentlich weniger Speicherplatz für baumähnliche Graphen benötigt als die transitive Hülle.

2 Speichern und Anfragen von Ontologien

Datenmodell. Wir betrachten Ontologien die die Form von gerichteten Bäumen oder azyklischen Graphen haben. Ein Pfad in einer solchen Struktur ist eine Folge von Knoten, die durch gerichtete Kanten verbunden sind, wobei die Länge des Pfades die Anzahl an Knoten in dem Pfad ist. In einem Baum kann jeder Knoten auf genau einem Pfad von der Wurzel aus erreicht werden. In DAGs können Knoten mehr als einen Elternknoten haben, daher kann mehr als ein Pfad zwischen zwei Knoten existieren. In einem Baum bzw. gerichteten Graphen ohne Zyklen (DAG) sind alle Knoten w Nachfahren von v wenn ein Pfad zwischen v und w existiert. Alle Knoten w, die von v aus zu erreichen sind, bilden die Menge an Nachfahren.

Graphen werden häufig als Sammlung von Knoten und Kanten gespeichert. Die Tabelle `node` enthält als eindeutige Kennung das Attribut `node_name`, in der Tabelle `EDGE` werden die gerichteten Kanten des Graphen in Form von Paaren von Knoten gespeichert, `from_node` und `to_node`.

Rekursive Datenbankfunktion. Alle Nachfahren eines gegebenen Knotens v können durch eine Tiefensuche über einem Baum oder DAG gefunden werden. Algorithmus 1 sucht zuerst alle Kinder des Startknotens v und für jedes der Kinder ruft sich die Funktion mit dem Kindknoten als neuen Startknoten selbst wieder auf. Können keine weiteren Kinder in dem Subgraphen mehr gefunden werden, gibt die Funktion alle traversierten Knoten zurück. Der Aufruf der Funktion erfolgt mit `SELECT * FROM successorSet(`v`);`.

Algorithmus 1 Rekursiver Algorithmus um alle Nachfahren von v zu erhalten.

```
FUNCTION successorSet(v) RETURNS nachfahren
  FOR EACH kind IN SELECT to_node FROM EDGE WHERE from_node = v DO
    FOR EACH enkel IN SELECT nachfahre FROM successorSet(kind) DO
      nachfahren := nachfahren + enkel;
    nachfahren := nachfahren + kind;
  RETURN nachfahren;
```

3 Indizierung von Baum- und DAG-Strukturen

In diesem Abschnitt erkläre ich Möglichkeiten die Beziehungen zwischen Knoten in Bäumen und DAGs vorzuberechnen. Die erste Möglichkeit ist die Berechnung der transitiven Hülle, während die zweite die Indizierung der Struktur mit Pre- und Postorder Ranks ist.

3.1 Berechnung der transitiven Hülle

Die transitive Hülle eines Graphen ist eine Menge von Knotenpaaren. Jedes Paar (v, w) wird in die transitive Hülle aufgenommen, wenn entweder eine Kante oder ein Pfad zwischen den Knoten v und w existiert.

In der Vergangenheit wurden verschiedene Algorithmen entwickelt um die transitive Hülle in relationalen Datenbanksystemen zu berechnen. Wir fanden heraus, dass der so genannte 'Logarithmic algorithm' [3] für Bäume und DAGs die geringste Zeit benötigt.

Dieser Algorithmus fügt zuerst alle Tupel der ursprünglichen Tabelle `EDGE` in die Tabelle für die transitive Hülle, `TC` mit der Distanz 1 zwischen zwei Knoten ein. In Schritt 2 werden alle Tupel mit maximaler Distanz mit `TC` gejoint. Die Bedingung für diese Operation ist, dass der Nachfolger der ersten Relation, $\texttt{TC}_1$, gleich dem Vorgänger der zweiten Relation, $\texttt{TC}_2$ sein muß. In der Tabelle `TC` wird $\texttt{TC}_1$`.VORG`, $\texttt{TC}_2$`.NACHF`, sowie `min(`$\texttt{TC}_1$`.DIST+`$\texttt{TC}_2$`.DIST)` eingefügt. Schritt 2 wird so lange wiederholt, bis keine weiteren Tupel mehr in `TC` eingefügt werden können.

Die transitive Hülle hat einen theoretischen Speicherplatzverbrauch von $O(|V|^2)$, doch der in Kapitel 4 gemessene Bedarf ist wesentlich geringer. Alle Nachfahren eines Knotens erhält man durch folgenden Ausdruck: `SELECT nachfahre FROM TC WHERE vorgaenger =` v`;`

3.2 Pre- und Postorder Ranking Modell

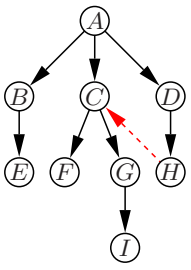
Das Pre- und Postorder Ranking Modell für Bäume ist gut untersucht. Verschiedene Gruppen haben dieses Modell vorgeschlagen, um XML Dokumente in relationalen Datenbanken zu indizieren [4].

Algorithmus 2 zeigt die Funktion, um Knoten in Bäumen als auch in DAGs Pre- und Postorder Ranks zuzuordnen. Die Knoten werden während einer Tiefensuche, angefangen am Wurzelknoten, markiert. Den Preorder Rank bekommt ein Knoten sobald er während der Tiefensuche gefunden wurde. Der Postorder Rank wird einem Knoten erst zugeteilt, wenn alle Nachfolger einen Postorder Rank haben, aber noch bevor ein Elternknoten in dem Pfad einen solchen besitzt.

Algorithmus 2 Pre- und Postorder Rank-Zurordnung, angefangen beim Wurzelknoten, r.

```
var pr:=0; var post:=0;
FUNCTION prePostOrder(r) RETURNS void
  FOR EACH kind IN SELECT to_node FROM EDGE WHERE from_node = r DO
    pr:=pr+1; pre:=pr;
    prePostOrder(kind);
    post:=post+1;
    s:= pr-pre;
    INSERT kind, pre, post, s INTO prePostOrder;
```

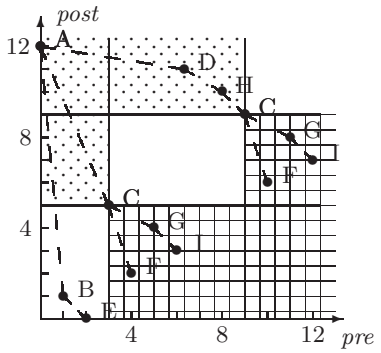
Wie in Abbildung 1(b) dargestellt, werden die Pre- und Postorder Ranks zusammen mit der eindeutigen Kennung des Knotens und der Anzahl an Kindern, `s`, in einer separaten Tabelle gespeichert. Für Bäume ist der Platzbedarf gleich der Anzahl an Knoten, in DAGs können jedem Knoten mehrere Ranks zugeordnet werden, abhängig von der Zahl an Wegen, auf denen dieser Knoten von der Wurzel aus erreichbar ist.



(a) DAG

ID	*pre*	*post*	*s*
A	0	12	12
B	1	1	1
C	3	5	3
D	7	11	5
E	2	0	0
F	4	2	1
G	5	4	1
H	8	10	4
I	6	3	0
C	9	9	3
F	10	6	0
G	11	8	1
I	12	7	0

(b) Tabelle mit den Pre-/ Postorder Ranks.



(c) Pre- /Postorder Ranks im Koordinatensystem. ⊞: Nachfolger von C, ∴: Vorgänger.

Abbildung 1: Pre- und Postorder Ranks in einem gerichteten azyklischen Graphen.

Der Nutzen von Pre- und Postorder Ranks wird deutlich, wenn wie in Abbildung 1(c) die Knoten mit den beiden Ranks in einem zweidimensionalen Koordinatensystem aufgetragen werden. Wie für Knoten C dargestellt, kann die Fläche in vier Regionen eingeteilt werden. Für Ontologien sind nur die Region oben links mit allen Vorfahren von C und die Region unten rechts mit allen Nachfolgern des Knotens C interessant.

Für alle Nachfolger eines Knotens v gilt, dass der Preorder Rank jedes Knotens w größer sein muß als der von v, während der Postorder Rank kleiner sein muß. Die Position der Nachfolger kann noch weiter eingeschränkt werden, denn alle Nachfahren von v besitzen einen Preorder Rank zwischen pre_v und $pre_v + s$. Beide Möglichkeiten sind in folgenden Anfragen dargestellt:

Option 1:

```
SELECT DISTINCT p1.node_name AS w
FROM prePostOrder p1, prePostOrder p2
WHERE p2.node_name = v
  AND p1.pre > p2.pre
  AND p1.post < p2.post;
```

Option 2:

```
SELECT DISTINCT p1.node_name AS w
FROM prePostOrder p1, prePostOrder p2
WHERE p2.node_name = v
  AND p1.pre > p2.pre
  AND p1.pre ≤ p2.pre + p2.s;
```

In Bäumen kommt jeder Knoten nur einmal vor, in DAGs kann ein Knoten jedoch mehrmals vorkommen. Wie man aber in Abbildung 1(c) sehen kann, ist die Menge an Nachfahren von jeder Instanz des Knotens C gleich, da Teilbäume von Knoten mit mehr als einem Elternknoten vervielfacht werden. Um nun jeden Knoten nur einmal in der Ergebnismenge zu erhalten ist `DISTINCT` in den Anfragen notwendig.

4 Ergebnisse

In diesem Abschnitt vergleiche ich die Indizierungsmethoden mit dem rekursiven Algorithmus. Ich stelle sowohl Ergebnisse von generierten Bäumen und DAGs als auch von existierenden Ontologien vor.

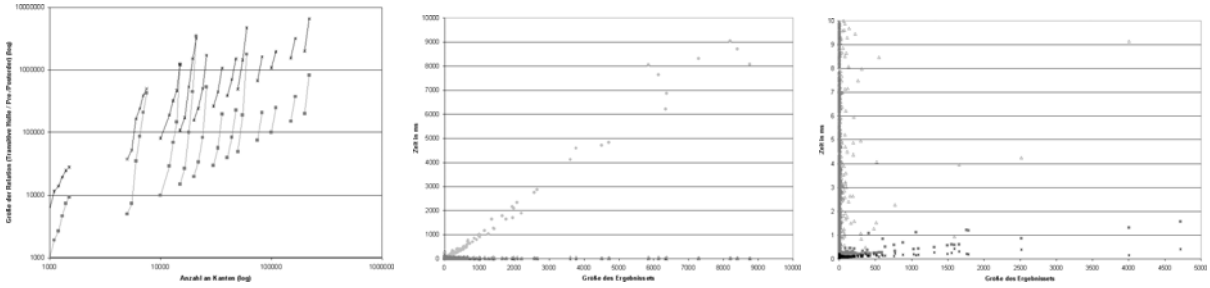
Speicherplatzbedarf. Um systematisch den Speicherplatzbedarf zu messen, habe ich Bäume und DAGs mit gegebener Zahl an Knoten erstellt. Abbildung 2(a) zeigt den Speicherplatzbedarf für die beiden Indexstrukturen. Der Startpunkt jeder Kurve stellt den Baum mit der entsprechenden Zahl an Kanten dar, während jeder weitere Punkt für einen DAG mit jeweils 10 % mehr Kanten steht.

Für einen Baum ist die Anzahl an Tupel, die in `prePostOrder` eingefügt werden, gleich der Anzahl an Knoten. Mit steigender Anzahl an Kanten im Graphen nähert sich die Anzahl an Tupeln in beiden Indextabellen an. Allerdings enthält für DAGs mit bis zu 30 % zusätzlichen Kanten die Tabelle `TC` noch über 50 % mehr Tupel als `prePostOrder` in den untersuchten Graphen. Werden 40 % oder mehr Kanten zusätzlich eingefügt, so kehrt sich die Situation um. Der Grund für dieses Verhalten ist die mehrfache Traversierung von Teilbäumen bei der Pre- /Postorder Ranking-Methode mit jeder zusätzlich eingefügten Kante. Auch die transitive Hülle wächst, doch je mehr Kanten bereits existieren, desto seltener wird eine neue Verbindung zwischen zwei Knoten erstellt, sondern nur alternative Pfade eingeführt.

Anfragezeiten. Die Anfragezeiten an Ontologien habe ich auf real existierenden Ontologien, dem Baum der NCBI Taxonomy und dem DAG der Gene Ontology, gemessen. Der Index für Pre-/ Postorder enthält in beiden Fällen weniger Tupel als die transitive Hülle, bei der NCBI Taxonomy 230 550 im Gegensatz zu 3 583 760. Auch die Gene Ontology (16 859 Knoten, 23 526 Kanten) hat nur 76 734 Tupel in dem Pre-/ Postorder Index im Gegensatz zu 178 033 für die transitive Hülle, obwohl bereits etwa 40 % zusätzliche Kanten eingefügt worden sind.

Die Suche nach allen Nachfahren für eine Menge zufällig ausgewählter Knoten zeigt in Abbildung 2(b) deutlich, dass die Anfragezeit der rekursiven Funktion linear von der Größe des Ergebnissets abhängt, während die Zeiten für die Indexstrukturen nahezu konstant bleiben. Abbildung 2(c) zeigt, dass für die Anfrage an die Pre-/ Postorder Indexstruktur die Option 2 aus 3.2 die bessere der beiden ist. Obwohl der Ausführungsplan für beide Anfragen jeweils einen Nested Loop-Join verwendet und in Oracle 9i identische Kosten vorhersagt, hängt die Anfragezeit der Option 1 von dem Pre- bzw. Postorder Rank – je nach verwendetem Index – ab. Je höher

dieser ist, desto länger dauert die Anfrage, da zuerst alle Tupel selektiert werden, die einen Pre- bzw. Postorder Rank haben, der kleiner ist als der des gegebenen Knotens. Bei Option 2 werden nur Tupel selektiert, die einen Preorder Rank innerhalb der gegebenen Grenzen besitzen. Die Antwortzeit für eine Anfrage mit einem Blattknoten variiert von 0,1 bis 2000 ms bei Option 1, während bei Option 2 die Zeiten unter 0,8 ms bleiben. Die Anfrage an die transitive Hülle liefert für viele der gewählten Knoten am schnellsten alle Nachfahren, wobei sie maximal um Faktor 4 schneller ist, was bei Zeiten um 1 ms nicht stark ins Gewicht fällt.



(a) Größe (log) der Indizes für generierte Bäume / DAGs.

(b) Anfragezeit an NCBI Taxonomy.

(c) Anfragezeit an Gene Ontology.

Abbildung 2: Größe der Indizes und Laufzeiten für Nachfahren eines Knotens v. $*$: transitive Hülle, $\triangle$: Pre-/ Post Option 1, $\blacksquare$: Pre-/ Post Option 2, $\bullet$: rekursive Funktion. Messungen mit Oracle 9i, auf DELL dual Xeon mit 4 GB RAM.

5 Zusammenfassung und Ausblick

Der Pre- und Postorder Index benötigt für Bäume und baumähnliche DAGs deutlich weniger Speicherplatz als die transitive Hülle. Anfragen nach allen Nachfolgern eines Knotens können mit beiden Indexstrukturen in konstanter Zeit beantwortet werden, wobei Anfragen an die transitive Hülle geringfügig schneller beantwortet werden. Auch andere Speicherstrukturen, die sowohl speicherplatzeffizient sind als auch Anfragen in konstanter Zeit erlauben, sollten nicht aus den Augen gelassen werden, wie der 2-hop-cover [5].

Literatur

[1] DL Wheeler, C Chappey, AE Lash, DD Leipe, TL Madden, GD Schuler, TA Tatusova, and BA Rapp. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 28(1):10 – 14, Jan 2000.

[2] Gene Ontoloy Consortium. The Gene Ontology (GO) database and inforamtics resource. *Nucleic Acids Research*, 32:D258 – D261, 2004. Database issue.

[3] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In L. Kerschberg, editor, *First International Conference on Expert Database Systems*, pages 271–293, Redwood City, CA, 1986. Addison-Wesley.

[4] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM Press, 2002.

[5] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, 2005.