Ontologie-basierte Fragmentierungs- und Replikationsverfahren für verteilte Datenbanksysteme

Lena Wiese
Forschungsgruppe Knowledge Engineering
Institut für Informatik
Georg-August-Universität Göttingen
wiese@cs.uni-goettingen.de

ABSTRACT

Das Auffinden von semantisch verwandten Daten in großen Datenmengen ist aufwändig und daher zur Laufzeit einer Anfrage nur schwierig durchzuführen. In diesem Artikel stellen wir ein Verfahren vor, dass Datentabellen anhand einer Ontologie in semantisch zusammenhängende Cluster aufteilt. Dadurch ergibt sich eine Ontologie-basierte Fragmentierung der Tabelle, die eine flexible Anfragebeantwortung unterstützt. Bei mehreren derartigen Fragmentierungen überschneiden sich Fragmente; dies wird für ein intelligentes Replikationsverfahren ausgenutzt, um die Anzahl der Replikaserver zu reduzieren.

Keywords

Verteiltes Datenbanksystem, Replikation, Ontologie, Fragmentierung, flexible Anfragebeantwortung

1. EINLEITUNG

Die Verwaltung von großen Datenmengen in Datenbanken erfordert die Verteilung der Daten auf mehrere Datenbank-Server. Dies bietet mehrere Vorteile:

- Lastverteilung: Datenbankanfragen können auf mehrere Server verteilt und damit parallelisiert werden.
- Verfügbarkeit: Durch die Lastverteilung erhöht sich die Verfügbarkeit des Gesamtsystems, da einzelne Anfragen seltener verzögert werden müssen.

Ein geeignetes Verfahren zur Fragmentierung (auch: Partitionierung oder Sharding) der Daten in Teilmengen ist daher notwendig. Die gängigen Fragmentierungsverfahren (Round-Robin, Hash-basiert, Intervall-basiert) ignorieren jedoch semantische Zusammenhänge der Daten.

Aus Gründen der Ausfallsicherheit ist zusätzlich noch die Replikation von Daten (also die Spiegelung desselben Datensatzes auf mehreren Servern) erforderlich. Dies gilt insbesondere für Hauptspeicherdatenbanken, die nicht über einen Hintergrundspeicher verfügen. In dieser Arbeit stellen wir Verfahren zur Ontologie-basierten Fragmentierung und zur intelligenten Replikation vor, womit folgende Eigenschaften sichergestellt werden:

- durch die Fragmentierung wird ein Verfahren der *flexiblen Anfragebeantwortung* unterstützt, die dem Anfrager auch relevante verwandte Werte als Antworten zurückliefert.
- durch die Fragmentierung wird eine Lastverteilung auf mehrere Server möglich.
- durch die Fragmentierung wird die Anzahl der kontaktierten Server pro Anfrage reduziert und damit eine bessere Parallelisierung möglich.
- durch die intelligente Replikation wird die Anzahl der benötigten Replikaserver reduziert.

1.1 Übersicht

Abschnitt 2 beschreibt den Hintergrund zu flexibler Anfragebeantwortung, Fragmentierung und Datenverteilung. Abschnitt 3 stellt die Ontologie-basierte Fragmentierung inklusive Clustering, Fragmentverteilung und Anfragebeantwortung vor. Abschnitt 4 beschreibt darauf aufbauend ein Replikationsverfahren mit überlappenden Fragmenten. Ein Wiederherstellungsverfahren anhand des Replikationsverfahrens wird in Abschnitt 5 dargestellt. Es folgt ein Verfahren für abgeleitete Fragmentierungen in Abschnitt 6 und abschließend eine Zusammenfassung in Abschnitt 7.

2. HINTERGRUND

Wir stellen hier kurz Vorarbeiten zu flexibler Anfragebeantwortung, Fragmentierung und Datenverteilung vor.

2.1 Flexible Anfragebeantwortung

Flexible Anfragebeantwortung ist ein intelligentes Verfahren, um Datenbanknutzern Antworten zu liefern, die zwar nicht exakt der Anfrage entsprechen, die jedoch dennoch interessante Informationen für den Benutzer darstellen. Dabei gibt es syntaktische und semantische Ansätze.

Zum einen gibt es syntaktische Änderungen, die Teile der Anfrage verändern. Dazu zählen [Mic83]:

- Dropping Conditions: Selektionsbedingungen werden aus der Originalanfrage entfernt.
- Goal Replacement: einige Bedingungen der Originalanfrage werden anhand einer Ableitungsregel durch andere Bedingungen ersetzt.

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

- Anti-Instantiation: Ein Term (eine Variable mit mehreren Vorkommen oder eine Konstante) wird durch eine neue Variable ersetzt.

Diese Operatoren können auch kombiniert werden [IW11].

Diese syntaktischen Änderungen können aber zu weit gehen und zu viele Antworten produzieren – insbesondere beim Ersetzen von Konstanten durch neue Variablen in der Anti-Instantiation. Daher ist es wichtig, dass die Werte, die der neuen Variable zugewiesen werden können, beschränkt werden: es sollen nur semantisch äquivalente oder semantisch nah verwandte Werte zugelassen werden. Diese semantische Ähnlichkeit kann anhand einer Ontologie oder Taxonomie von Werten bestimmt werden. Für Einzelrechner wurden bereits vor einiger Zeit Verfahren vorgeschlagen – ohne jedoch verteilte Datenspeicherung mit einzubeziehen. CoBase [CYC+96] und [SHL07] zum Beispiel benutzten sogenannte Abstraktionshierarchien, um einzelne Werte zu generalisieren. Auch für XML-Anfragen wurden Verfahren entwickelt [HTGC10].

Ein grundlegendes Problem ist dabei jedoch, dass die Bestimmung von ähnlichen Werten zur Laufzeit (während der Anfragebeantwortung) viel zu ineffizient ist [BDIW14]. Dieses Problem wird durch das hier vorgestellte Fragmentierungsverfahren gelöst.

2.2 Fragmentierung

Im relationalen Modell gibt es die Theorie der Datenfragmentierung, die sich aufgrund des strikten Tabellenformats aufteilen lässt in horizontale und vertikale Fragmentierung (siehe zum Beispiel [ÖV11]):

- Vertikale Fragmentierung: Fragmente entsprechen Teilmengen von Attributen (also Tabellenspalten). Die Einträge in den Fragmenten, die zur selben Tabellenzeile gehören, müssen durch einen Tuple-Identifikator verbunden werden. Vertikale Fragmentierung entspricht einer Projektion auf der Tabelle.
- Horizontale Fragmentierung: Fragmente sind Teilmengen von Tupeln (also Tabellenzeilen). Eine horizontale Fragmentierung entspricht einer Selektion auf der Tabelle.
- Abgeleitete Fragmentierung: Eine bereits bestehende „primäre“ horizontale Fragmenation of einer Tabelle induziert eine horizontale Fragmentierung einer weiteren Tabelle; dazu wird der Semi-JOIN auf beiden Tabellen ausgeführt.

Drei grundlegende Eigenschaften sind wichtig für Fragmentierungen:

- Vollständigkeit: Keine Daten dürfen während der Fragmentierung verloren gehen. Bei vertikaler Fragmentierung ist jede Spalte in einem Fragment enthalten; bei horizontaler Fragmentierung ist jede Zeile in einem Fragment enthalten.
- Rekonstruierbarkeit: Daten aus den Fragmenten können wieder zur Originaltabelle zusammengefügt werden. Bei vertikaler Fragmentierung wird der JOIN-Operator benutzt (basierend auf einem zusätzlich eingefügten Tupel-Identifikator), um die Spalten zu verbinden. Bei horizontaler Fragmentierung wird der Vereinigungsoperator zum Zusammenführen der Fragmente verwendet.
- Redundanzfreiheit: Um Duplizieren von Daten zu vermeiden, sollen einzelne Datensätze nur einem Fragment zugewiesen werden. Bei vertikaler Fragmentierung ist jede Spalte nur in einem Fragment enthalten (abgesehen vom Tupel-Identifikator). Bei horizontaler Fragmentierung ist jede Zeile in nur einem Fragment enthalten.

In anderen, nicht-relationalen Datenmodellen (Schlüssel-Wert-Speicher, Dokumentdatenbanken oder Spaltenfamiliendatenbanken) gibt es Fragmentierung meist über den Zugriffsschlüssel entweder Hash-basiert [KLL+97] oder basierend auf Intervallen. Jedoch unterstützt keines dieser Verfahren die flexible Anfragebeantwortung; im Gegensatz dazu hat das im Folgenden vorgestellte Fragmentierungsverfahren den Vorteil, dass eine relaxierte Bedingung aus nur einem Fragment beantwortet werden kann.

2.3 Datenverteilung

In einem verteilten Datenbanksystem müssen Daten auf die verschiedenen Server verteilt werden. Wichtige Eigenschaften sind

- Datenlokalität: Daten, auf die innerhalb einer Anfrage oder Transaktion zugegriffen wird, sollten möglichst auf einem Server sein. Dadurch verringert sich die Anzahl der kontaktierten Server und somit auch die Dauer der Anfragebeantwortung. Außerdem kann so die Parallelisierung der Anfragebeantwortung verbessert werden, da die anderen Server neue Anfragen annehmen können.
- Lastverteilung: Daten sollen so verteilt werden, dass parallele Anfragen möglichst auch von verschiedenen Servern beantwortet werden können, damit nicht einzelne Server unter Lastspitzen („hot spots“) leiden.

Einige Arbeiten befassen sich mit horizontaler Fragmentierung und Verteilung; jedoch unterstützen diese nur exakte Anfragebeantwortung und keine flexible Anfragebeantwortung wie in unserem Ansatz. Die meisten Ansätze gehen von einer vorgebenen Menge von Anfragen oder Transaktionen aus („workload“) und optimieren die Lokalität der Daten, die innerhalb der Anfrage beziehungsweise Transaktion benötigt werden. Dies ist für die Anwendung der flexiblen Anfragebeantwortung jedoch nicht anwendbar, da hier auch Werte zurückgegeben werden, die nicht explizit in einer Anfrage auftauchen.

[CZJM10] stellen Tupel als Knoten eines Graphen dar. Für eine vorgegebene Menge von Transaktionen fügen sie Hyperkanten in den Graph ein, wenn die Tupel in derselben Transaktion benötigt werden. Mit einem Graphpartitionierungsalgorithmus wird dann eine Datenfragmentierung ermittelt, die Zahl der zerschnittenen Kanten minimiert. In einer zweiten Phase benutzen die Autoren einen Klassifizierer aus dem Maschinellen Lernen, der eine Intervall-basierte Fragmentierung herleitet. Experimentell vergleichen sie dann das Graph-basierten mit Intervall-basierten und Hash-basierten Verfahren. Im Gegensatz zu unserem Ansatz wenden sie jedoch volle Replikation an, bei der alle Server alle Daten vorhalten; dies ist wenig realistisch in großen verteilten Systemen. Zudem vergleichen sie drei verschiedene Arten von Lookup-Tabellen, die Tupel-Identifikatoren für jedes Fragment abspeichern: Indexe, Bitarrays und Bloomfilter. Bei der Analyse unseres Systems stellt sich jedoch her-

aus, dass Lookup-Tabellen ineffizienter sind als das Einführen einer zusätzlichen Spalte mit der Cluster-ID in anderen Fragmentierungen.

Auch [QKD13] modellieren das Fragmentierungsproblem durch Minimierung zerschnittener Hyperkanten in einem Graphen. Zur Verbesserung der Effizienz komprimieren sie den Graphen und erhalten so Gruppen von Tupeln. Die Autoren kritisieren dabei auch den tupelweisen Ansatz von [CZJM10] als unpraktisch für große Tupelmengen. Die Autoren vergleichen ihren Ansatz mit zufälligen und tupelweisen Fragmentierungen und betrachten auch Änderungen der vorgegebenen Transaktionen.

[TCJM12] gehen von drei existierenden Fragmentierungen aus: Hash-basiert, Intervall-basiert und Lookup-Tabellen auf einzelnen Zugriffsschlüsseln. Sie vergleichen diese drei bezüglich Kommunikationskosten und Anfragendurchsatz. Zur Effiziensteigerung analysieren sie diverse Komprimierungstechniken. Sie beschreiben Hash-basierte Fragmentierung als zu ineffizient. Die Autoren beschreiben jedoch nicht, wie die Fragmentierungen für die Lookup-Tabellen berechnet werden; im Gegensatz dazu stellen wir ein Ontologie-basiertes Fragmentierungsverfahren vor.

Im Gegensatz zu den meisten anderen Ansätzen gehen wir nicht von einer vorgegebenen Menge von Anfragen oder Transaktionen aus sondern schlagen ein allgemein anwendbares Clusteringverfahren vor, dass die flexible Anfragebeantwortung zum Auffinden semantisch ähnlicher Antworten ermöglicht. Unsere Ergebnisse zeigen, dass Lookup-Tabellen (selbst wenn sie auf allen Servern repliziert werden) für unseren Ansatz zu ineffizient sind, dadurch dass viele JOIN-Operationen durchgeführt werden müssen.

3. ONTOLOGIE-BASIERTE FRAGMENTIERUNG

Unser Verfahren der Ontologie-basierten Fragmentierung beruht darauf, dass

- zur Anti-Instantiierung ein Attribut (also eine Tabellenspalte) ausgewählt wird.
- ein Clusteringverfahren auf dieser Tabellenspalte ausgeführt wird, um die ähnlichen Werte innerhalb dieser Spalte zu gruppieren.
- anhand der Cluster die Tabelle zeilenweise fragmentiert wird.

Wie in Abbildung 1 dargestellt werden dann die Anfragen so weitergeleitet, dass zu einer Konstante aus der Originalanfrage das semantisch ähnlichste Fragment ermittelt wird und anschließend alle Werte des Fragments als relevante Antworten zurückgeliefert werden. Daher werden zum Beispiel bei einer Anfrage nach Husten auch die ähnlichen Werte Asthma und Bronchitis gefunden.

3.1 Clustering

Zu dem zur Anti-Instantiierung ausgewählten Attribut A in der gegebenen Tabelle F werden alle in der Tabelle vorhandenen Werte (die sogenannte aktive Domäne) ausgelesen durch Projektion $\pi_A(F)$. Anhand einer gegebenen Ontologie werden Ähnlichkeitswerte *sim* zwischen jeweils zwei Termen a, $b \in \pi_A(F)$ bestimmt im Wertebereich 0 (keine Ähnlichkeit) bis 1 (volle Ähnlichkeit). Dazu gibt es verschiedene Metriken, die meist auf der Berechnung von Pfaden zwischen den Termen in der Ontologie beruhen [Wie14, Wie13].

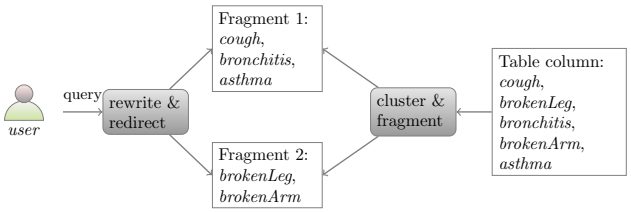

Figure 1: Fragmentation and query rewriting

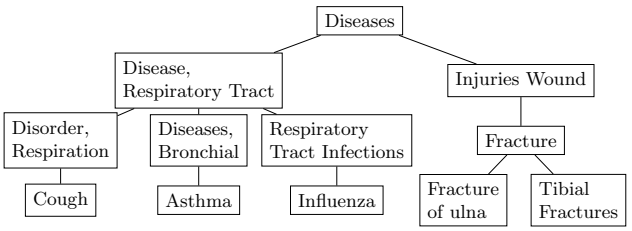

Figure 2: Beispieltaxonomie für Krankendaten

Ein Clustering-Verfahren benutzt diese Ähnlichkeitswerte um Cluster zu bestimmen (in Anlehnung an [Gon85]). Dazu wird in jedem Cluster ein prototypisches Element *head* bestimmt, das das jeweilige Cluster repräsentiert. Zusätzlich gibt es einen Schwellenwert α und das Verfahren unterteilt Cluster solange in Teilcluster, bis für jedes Element innerhalb eines Clusters die Ähnlichkeit zu *head* mindestens α ist. Das heißt, für jedes Cluster c_i und $head_i \in c_i$ gilt für jeden anderen Wert $a \in c_i$ (mit $a \neq head_i$), dass $sim(a, head_i) \geq \alpha$. Dieses Verfahren wird in Auflistung 1 dargestellt.

Listing 1 Clustering procedure

Input: Set $\pi_A(F)$ of values for attribute A, similarity threshold α
Output: A set of clusters $c_1, \ldots, c_f$
1: Let $c_1 = \pi_A(F)$
2: Choose arbitrary $head_1 \in c_1$
3: $sim_{min} = min\{sim(a, head_1) \mid a \in c_1; a \neq head_1\}$
4: $i = 1$
5: **while** $sim_{min} < \alpha$ **do**
6: Choose $head_{i+1} \in \bigcup_{1 \leq j \leq i}\{b \mid b \in c_j; b \neq head_j; sim(b, head_j) = sim_{min}\}$
7: $c_{i+1} = \{head_{i+1}\} \cup \bigcup_{1 \leq j \leq i}\{c \mid c \in c_j; c \neq head_j; sim(c, head_j) \leq sim(c, head_{i+1})\}$
8: $i = i + 1$
9: $sim_{min} = min\{sim(d, head_j) \mid d \in c_j; d \neq head_j; 1 \leq j \leq i\}$
10: **end while**

Zum Beispiel kann eine Taxonomie wie in Abbildung 2 benutzt werden, um die Tabellenspalte aus Abbildung 1 zu clustern.

3.2 Fragmentierung

Ein Clustering der aktiven Domäne von A induziert eine horizontale Fragmentierung der Tabelle F in Fragmente $F_i \subseteq F$. Jede aktive Domäne eines Fragments F_i entspricht genau den Werten in einem Cluster: $c_i = \pi_A(F_i)$. Die grundlegenden Eigenschaften einer Fragmentierung (Vollständig-

keit, Rekonstruierbarkeit, Redundanzfreiheit) sollen auch bei einer Clustering-basierten Fragmentierung gelten. Auch das Clustering muss vollständig sein: bei einer aktiven Domäne $\pi_A(F)$ muss dann für ein Clustering $C = c_1, \ldots, c_n$ gelten, dass es die ganze aktive Domäne umfasst und kein Wert verloren geht: $c_1 \cup \ldots \cup c_n = \pi_A(F)$. Die Eigenschaften einer Clustering-basierten Fragmentierung werden in Definition 1 zusammengefasst.

DEFINITION 1 (CLUSTERING-BAS. FRAGMENTIERUNG). *Für ein Attribut A einer Tabelle F (eine Menge von Tupeln t) und ein Clustering $C = \{c_1, \ldots c_n\}$ der aktiven Domäne $\pi_A(F)$ und für $head_i \in c_i$ gibt es eine Menge von Fragmenten $\{F_1, \ldots, F_n\}$ (definiert über denselben Attributen wie F), so dass folgende Eigenschaften gelten:*

- *Horizontale Fragmentierung: für jedes Fragment F_i gilt $F_i \subseteq F$*
- *Clustering: für jedes F_i gibt es in Cluster $c_i \in C$ so dass $c_i = \pi_A(F_i)$*
- *Schwellenwert: für jedes $a \in c_i$ (wobei $a \neq head_i$) gilt $sim(a, head_i) \geq \alpha$*
- *Vollständigkeit: für jedes Tupel t in F gibt es ein Fragment F_i, in dem t enthalten ist*
- *Rekonstruierbarkeit: $F = F_1 \cup \ldots \cup F_n$*
- *Redundanzfreiheit: für jedes $i \neq j$, $F_i \cap F_j = \emptyset$ (oder auch $c_i \cap c_j = \emptyset$)*

3.3 Fragmentverteilung

In verteilten Datenbanken müssen Fragmente verschiedenen Servern zugewiesen werden. Dieses Fragmentverteilungsproblem kann als ein Bin-Packing-Problem dargestellt werden:

- K Server entsprechen K Bins
- Jedes Bin hat maximale Kapazität W
- n Fragmente entsprechen n Objekten
- Jedes Objekt/Fragment hat ein Gewicht (oder Kapizitätsverbrauch) $w_i \leq W$
- alle Objekte müssen auf möglichst wenig Bins verteilt werden, wobei die Gewichtsgrenze W beachtet werden muss.

Dieses Bin-Packing-Problem kann auch als Problem der ganzzahligen linearen Optimierung dargestellt werden:

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{1}$$

$$\text{s.t.} \quad \sum_{k=1}^{K} x_{ik} = 1, \qquad i = 1, \ldots, n \tag{2}$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W y_k, \qquad k = 1, \ldots, K \tag{3}$$

$$y_k \in \{0, 1\} \qquad k = 1, \ldots, K \tag{4}$$

$$x_{ik} \in \{0, 1\} \quad k = 1, \ldots, K, \; i = 1, \ldots, n \tag{5}$$

Dabei entspricht x_{ik} einer Binärvariable die dann wahr (1) ist, wenn Fragment/Objekt i Server/Bin k zugewiesen wird; und y_k bedeutet, dass Server/Bin k belegt ist (also nicht leer). Gleichung (1) fordert, dass die Anzahl der belegten Server minimiert wird; Gleichung (2) erzwingt, dass jedes Fragment einem Server zugewiesen wird; Gleichung (3) bedeutet, dass die Kapazitätsgrenzen nicht überschritten werden; die letzten beiden Gleichungen stellen sicher, dass die Variablen binär sind.

Zusätzlich können noch die Eigenschaften der Datenlokalität und der Lastverteilung optimiert werden. Datenverteilung mit einer guten Datenlokalität platziert die Fragmente zusammen auf einen Server, die häufig gemeinsam innerhalb einer Datenbanktransaktion (oder auch innerhalb einer Anfrage) benutzt werden. Lastverteilung sorgt dafür, dass alle Server ungefähr dieselbe Anzahl von Anfragen beantworten müssen.

3.4 Metadaten

Für die Durchführung des Clustering, der Fragmentverteilung und der Anfrageumschreibung und -umleitung werden ein paar Metadaten benötigt.

Eine Tabelle **root** speichert einen Identifikator für jedes Cluster (Spalte ID), den Namen des Fragments (Name), den Repräsentaten des Clusters (Head), die Größe des Clusters (S) sowie den Server (Host), auf dem das Fragment gespeichert wird. Eine Beispieltabelle sieht dann so aus:

ROOT	*ID*	*Name*	*Head*	*S*	*Host*
	101	Respiratory	Flu	4	S1
	107	Fracture	brokenArm	2	S2

Eine Tabelle **similarities** speichert die Ähnlichkeiten aller Werte des betrachteten Attributes zu den jeweiligen *head*-Werten der Cluster.

3.5 Anfragebeantwortung

Für die flexible Anfragebeantwortung wird die Konstante im entsprechenden Attribut A in der Anfrage anti-instantiiert und die entstehende Anfrage dann aus dem semantisch ähnlichsten Fragment beantwortet.

Als Beispiel betrachten wir eine Anfrage nach Husten in einer Tabelle *ill* mit Patienten IDs und Krankheiten:

```
SELECT patientid, disease
 FROM ill WHERE disease='cough'
```

Über die Tabelle similarities wird das Fragment F_i ausgewählt, dessen *head* am ähnlichsten zu der anti-instantiierten Konstante (im Beispiel `cough`) ist.

```
SELECT TOP 1 root.name
 FROM root, similarities
 WHERE similarities.term='cough'
 AND similarities.head = root.head
 ORDER BY similarities.sim DESC
```

Der Name der Originaltabelle F wird durch den Namen des Fragmentes F_i ersetzt und die geänderte Anfrage an den entsprechenden Server gesendet. Dadurch werden alle Werte aus dem Fragment als relevante Antworten zurückgeliefert. Als Beispiel sei der Fragmentname Respiratory. Daher wird die Anfrage geändert zu:

```
SELECT patientid, disease  FROM respiratory
```

und and den entsprechenden Server (hier S1) weitergeleitet.

Ähnliches gilt beim Einfügen neuer Zeilen:

```
INSERT INTO ill  VALUES (349, 'asthma')
```

wird umgeschrieben zu:

```
INSERT INTO respiratory  VALUES (349, 'asthma').
```

Auch das Löschen von Werten wird so umgesetzt:

```
DELETE FROM ill WHERE disease='cough'
```

wird zu:

```
DELETE FROM respiratory WHERE mesh='cough'
```

4. INTELLIGENTE REPLIKATION

Bisherige Replikationsverfahren kopieren die vorhandenen Fragmente auf andere Server; bei einer m-fachen Replikation verteilen sich so m verschiedene Kopien jedes Fragments auf m verschiedenen Servern. Dabei wird davon ausgegangen, dass die Fragmentierung redundanzfrei ist und sich die Fragmente daher nicht überschneiden: jedes Tupel ist in genau einem Fragment enthalten.

Bei der Ontologie-basierten Fragmentierung kann es jedoch sein, dass mehrere Attribute zur Anti-Instantiierung ausgewählt werden. Dadurch ergeben sich mehrere Fragmentierungen derselben Tabelle. Fragmente aus unterschiedlichen Fragmentierungen überschneiden sich deswegen. Bei α redundanzfreien Fragmentierungen ist daher jedes Tupel in α verschiedenen Fragmenten enthalten.

Beispielsweise kann unsere Beispieltabelle anhand eines Clusterings der Krankheitsdiagnose fragmentiert werden; so ergeben sich zwei Fragemente: Respiratory und Fracture.

Respiratory	*PatientID*	*Disease*
	8457	Cough
	2784	Flu
	2784	Asthma
	8765	Asthma

Fracture	*PatientID*	*Diagnosis*
	2784	brokenLeg
	1055	brokenArm

Zum Zweiten kann dieselbe Tabelle auch anhand der IDs der Patienten fragmentiert werden.

IDlow	*PatientID*	*Diagnosis*
	2784	Flu
	2784	brokenLeg
	2784	Asthma
	1055	brokenArm

IDhigh	*PatientID*	*Diagnosis*
	8765	Asthma
	8457	Cough

Dabei gilt, dass *Respiratory* $\cap$ *IDlow* $\neq \emptyset$, *Respiratory* $\cap$ *IDhigh* $\neq \emptyset$ und *Fracture* $\cap$ *IDlow* $\neq \emptyset$.

Im Sinne der Minimierung der benutzten Server sollten nicht alle α Fragmentierungen m-fach kopiert werden, da dies zu $\alpha \cdot m$ Kopien jedes Tupels führt, obwohl m Kopien ausreichen würden. Das bedeutet, dass das hier vorgestellte Verfahren weniger Speicherplatzbedarf hat als eine konventionelle Replikation aller ontologie-basierter Fragmente.

Um eine m-fache Replikation pro Tupel zu erreichen, werden daher die Überschneidungen berücksichtigt. Wir gehen im Folgenden (ohne Beschränkung der Allgemeinheit) davon aus, dass $\alpha = m$ – andernfalls werden einige Fragmentierungen dupliziert bis die entsprechende Anzahl erreicht ist. Damit ist also jedes Tupel in m Fragmenten enthalten.

Das Fragmentverteilungsproblem lässt sich daher erweitern um die Bedingung, dass überlappende Fragmente ($i \cap i' \neq \emptyset$) auf verschiedenen Server platziert werden (Gleichung (9)). Im Beispiel kann also *Fracture* nicht mit *IDlow* auf einem Server platziert werden; jedoch können *Fracture* und *IDhigh* auf demselben Server liegen. Generell handelt es sich dann dabei um ein Bin Packing Problem with Conflicts (BPPC):

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{6}$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik} = 1, \quad i = 1, \ldots, n \tag{7}$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W y_k, \quad k = 1, \ldots, K \tag{8}$$

$$x_{ik} + x_{i'k} \leq y_k \quad k = 1, \ldots, K, \quad i \cap i' \neq \emptyset \tag{9}$$

$$y_k \in \{0, 1\} \quad k = 1, \ldots, K \tag{10}$$

$$x_{ik} \in \{0, 1\} \quad k = 1, \ldots, K, \quad i = 1, \ldots, n \tag{11}$$

Um diese Konflikte (überlappende Fragmente) zu identifizieren werden Fragmente aus verschiedenen Fragmentierungen verglichen. Im Beispiel gibt es Fragmente c_i mit einer Cluster-ID (Fragmentierung wie im obigen Beispiel über den Werten der Krankheiten) und Fragmente r_j mit einer Range-ID (Fragmentierung über den Werten der Patienten-ID):

```
SELECT DISTINCT clusterid, rangeid
 FROM c_i  JOIN r_j  ON (r_j.tupleid=c_i.tupleid)
```

für jedes Cluster-Fragment c_i und jedes Range-Fragment r_j. Danach wird das resultierende BPPC gelöst und die Fragmente auf entsprechend viele Server verteilt mittels:

```
ALTER TABLE c_i  MOVE TO 'severname' PHYSICAL.
```

5. WIEDERHERSTELLUNG

Passend zum Replikationsverfahren müssen im Falle eines Serverausfalls einige Fragmente wiederhergestellt werden. Dazu werden der Originaltabelle Spalten für die jeweiligen Cluster-Identifikatoren hinzugefügt. Anhand der IDs können die entsprechenden Fragmente rekonstruiert werden:

```
INSERT INTO c_i SELECT * FROM  r_j  WHERE clusterid=i
```

Alternativ ist die Erstellung einer sogenannten Lookup-Tabelle [TCJM12] möglich, die zu jeder Cluster-ID die beteiligten Tupelidentifikatoren abspeichert. Diese benötigt jedoch einen JOIN-Operator:

```
INSERT INTO c_i SELECT * FROM ill  JOIN lookup
 ON (lookup.tupleid= r_j.tupleid)
 WHERE lookup.clusterid=i
```

Die Lookup-Tabelle hat sich daher als ineffizienter herausgestellt.

6. DATENLOKALITÄT FÜR ABGELEITETE FRAGMENTIERUNGEN

Wenn auf mehrere Tabellen innerhalb einer Anfrage zugegriffen wird und diese Tabellen Join-Attribute gemeinsam haben, kann durch abgeleitete Fragmentierung die Datenlokalität für die Anfragen erhöht werden. Zum Beispiel sei zusätzlich zur Tabelle *ill* eine Tabelle *info* gegeben, die zu jeder Patienten-ID Adressangaben enthält. Eine mögliche Anfrage wäre daher, die Angaben zu Krankheiten und Adressen zu kombinieren:

```
SELECT a.disease, a.patientid, b.address
 FROM ill AS a,info AS b  WHERE disease='cough'
 AND b.patientid= a.patientid
```

Anhand der vorgegebenen primären Fragmentierung der Tabelle *ill* wird dann auch die Tabelle *info* fragmentiert, zum Beispiel für das Fragment Respiratory:

```
INSERT INTO inforesp
 SELECT a.patientid, b.address
 FROM respiratory AS a, info AS b
 WHERE b.patientid = a.patientid
```

Daher kann dann im Folgenden auch die Anfrage, die Angaben zu Krankheiten und Adressen kombiniert, entsprechend umgeschrieben werden:

```
SELECT a.disease, a.patientid, b.address
 FROM respiratory AS a
 JOIN inforesp AS b  ON (a.patientid=b.patientid)
```

7. ZUSAMMENFASSUNG UND AUSBLICK

Flexible Anfragebeantwortung unterstützt Benutzer bei der Suche nach relevanten Informationen. In unserem Verfahren wird auf eine Ontologie zurückgegriffen aufgrund derer semantisch ähnliche Werte in einem Cluster zusammengefasst werden können. Eine Fragmentierung der Originaltabelle anhand der Cluster ermöglichst ein effizientes Laufzeitverhalten der flexiblen Anfragebeantwortung. Durch einige Metadaten (Root-Tabelle, Similarities-Tabelle, zusätzliche Spalte für Cluster-ID) werden alle typischen Datenbankoperationen unterstützt.

Zukünftig soll insbesondere das dynamische Anpassen der Fragmente untersucht werden: da sich durch Einfügungen und Löschungen die Größen der Fragmente stark ändern können, müssen zur Laufzeit Fragmente verschoben werden, sowie gegebenenfalls zu kleine Fragmente in ein größeres vereinigt werden beziehungsweise zu große Fragmente in kleinere aufgespalten werden.

8. REFERENCES

[BDIW14] Maheen Bakhtyar, Nam Dang, Katsumi Inoue, and Lena Wiese. Implementing inductive concept learning for cooperative query answering. In *Data Analysis, Machine Learning and Knowledge Discovery*, pages 127–134. Springer, 2014.

[CYC+96] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. CoBase: A scalable and extensible cooperative information system. *JIIS*, 6(2/3):223–259, 1996.

[CZJM10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1):48–57, 2010.

[Gon85] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[HTGC10] J. Hill, J. Torson, Bo Guo, and Zhengxin Chen. Toward ontology-guided knowledge-driven xml query relaxation. In *Computational Intelligence, Modelling and Simulation (CIMSiM)*, pages 448–453, 2010.

[IW11] Katsumi Inoue and Lena Wiese. Generalizing conjunctive queries for informative answers. In *Flexible Query Answering Systems*, pages 1–12. Springer, 2011.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[Mic83] Ryszard S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, Berlin/Heidelberg, Germany, 2011.

[QKD13] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences*, pages 430–441, New York, NY, USA, 2013. ACM.

[SHL07] Myung Keun Shin, Soon-Young Huh, and Wookey Lee. Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy. *Expert Systems with Applications*, 32(2):469–484, 2007.

[TCJM12] Aubrey Tatarowicz, Carlo Curino, Evan P. C. Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012)*, pages 102–113, Washington, DC, USA, 2012. IEEE Computer Society.

[Wie13] Lena Wiese. Taxonomy-based fragmentation for anti-instantiation in distributed databases. In *3rd International Workshop on Intelligent Techniques and Architectures for Autonomic Clouds (ITAAC'13) collocated with IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 363–368, Washington, DC, USA, 2013. IEEE.

[Wie14] Lena Wiese. Clustering-based fragmentation and data replication for flexible query answering in distributed databases. *Journal of Cloud Computing*, 3(1):1–15, 2014.