Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Clustering-based fragmentation and data replication for flexible query answering in distributed databases

Lena Wiese

Abstract

One feature of cloud storage systems is data fragmentation (or sharding) so that data can be distributed over multiple servers and subqueries can be run in parallel on the fragments. On the other hand, flexible query answering can enable a database system to find related information for a user whose original query cannot be answered exactly. Query generalization is a way to implement flexible query answering on the syntax level. In this paper we study a clustering-based fragmentation for the generalization operator Anti-Instantiation with which related information can be found in distributed data. We use a standard clustering algorithm to derive a semantic fragmentation of data in the database. The database system uses the derived fragments to support an intelligent flexible query answering mechanism that avoids overgeneralization but supports data replication in a distributed database system. We show that the data replication problem can be expressed as a special Bin Packing Problem and can hence be solved by an off-the shelf solver for integer linear programs. We present a prototype system that makes use of a medical taxonomy to determine similarities between medical expressions.

Keywords: Fragmentation; Distributed database; Flexible query answering; Clustering; Load balancing; Data replication; Bin packing with conflicts

Introduction

In the era of "big data" huge data sets usually cannot be stored on a single server any longer. Cloud storage (where data are stored in a cloud infrastructure) offers the advantage of flexibly adapting the amount of used storage based on the growing or shrinking storage demands of the data owners. In a cloud storage system, a distributed database management system (DDBMS) can be used to manage the data in a network of servers. This allows for load balancing (data can be distributed according to the capacities of servers) and higher availability (servers can process user requests in parallel). In particular, when data are distributed over a wider area in different data centers, it is important that only few servers have to be contacted to answer user queries in order to reduce network delays; in the ideal case, these servers are also geographically close to the user.

Correspondence: wiese@cs.uni-goettingen.de
Institute of Computer Science, Georg-August-Universität Göttingen, Goldschmidtstraße 7, Göttingen, Germany

Depending on the data structure used in the DDBMS a variety of distribution models are possible. For relational data, the theory of fragmentation has a long history (see for example [1]) and several procedures have been analyzed for splitting tabular data into fragments and subsequently assigning fragments to servers. Other database systems with key-based access (like key-value stores, document databases, or column family stores) use range-based partitioning or consistent hashing to distribute data.

On the other hand, *flexible query answering* offers mechanisms to intelligently answer user queries going beyond conventional exact query answering. If a database system is not able to find an exactly matching answer, the query is said to be a *failing* query. Conventional database systems usually return an empty answer to a failing query. In most cases, this is an undesirable situation for the user, because he has to revise his query and send the revised query to the database system in order to get some information from the database. In contrast, flexible query answering systems internally revise failing user queries themselves and – by evaluating the revised query – return answers to the user that are more informative for the user than just an empty answer. *Query generalization* is one way to implement flexible query answering.

This paper revises and extends the previous results presented in [2]. In this paper we make the following additional contributions:

- We study how a standard clustering heuristic on a single relaxation attribute (that is, table column) can induce a horizontal fragmentation of a database table; in [2] a taxonomy-based fragmentation was used instead of a clustering-based fragmentation.
- We formally study the *data replication problem* for these fragments by representing it as a variant of the bin packing problem and solve it using an integer linear programming solver. This was not discussed in [2].
- We present a detailed query rewriting and query redirecting method that allows access to the distributed fragments. This was discussed in [2] only briefly.

The paper is organized as follows. Section Background provides background on data fragmentation, query generalization (in particular anti-instantiation) and data replication. Section Clustering-based fragmentation presents the main contribution on clustering-based fragmentation and its management with a lookup table; whereas Section Query rewriting talks about how to decompose a query to be distributed among the servers. Section Improving data locality with derived fragmentations extends the basic approach by allowing derived fragmentation in order to facilitate joins over multiple tables. Section Implementation and example presents the components of our prototype implementation. Section Related work surveys related work and Section Discussion and conclusion concludes the paper.

Background

In the following subsections we present prior work on **data fragmentation**, flexible query answering (with a focus on **anti-instantiation**) and **data replication**. These three techniques will be combined to obtain an intelligent distributed database system that can autonomously configure its replication mechanism while at the same time support users in finding relevant information by flexible query answering.

Data fragmentation

As the basic data model, we consider the case of data stored in relational tables. The relational data model is still widely applied today although alternatives exist (like tree- or graph-structured data or data stored in a simple key-value format).

Example 1. *As a running example, we consider a hospital information system that stores illnesses and treatments of patients as well as their personal information (like address and age) in the following three database tables:*

Ill	PatientID	Diagnosis
	8457	*Cough*
	2784	*Flu*
	2784	*Asthma*
	2784	*brokenLeg*
	8765	*Asthma*
	1055	*brokenArm*

Treat	PatientID	Prescription
	8457	*Inhalation*
	2784	*Inhalation*
	8765	*Inhalation*
	2784	*Plaster bandage*
	1055	*Plaster bandage*

Info	PatientID	Name	Address
	8457	*Pete*	*Main Str 5, Newtown*
	2784	*Mary*	*New Str 3, Newtown*
	8765	*Lisa*	*Main Str 20, Oldtown*
	1055	*Anne*	*High Str 2, Oldtown*

In relational database theory, several alternatives of splitting tables into fragments have been discussed (see for example [1]), for example:

- Vertical fragmentation: Subsets of attributes (that is, columns) form the fragments. Rows of the fragments that correspond to each other have to be linked by a tuple identifier. A vertical fragmentation corresponds to projection operations on the table.
- Horizontal fragmentation: Subsets of tuples (that is, rows) form the fragments. A horizontal fragmentation can be expressed by a selection condition on the table.
- Derived fragmentation: A given horizontal fragmentation on a primary table (the *primary* fragmentation) induces a horizontal fragmentation of another table based on the semijoin with the primary table. In this case, the primary and derived fragments with matching values for the join attributes can be stored on the same server; this improves efficiency of a join on the primary and the derived fragments.

The following three properties are considered the important *correctness properties* of a fragmentation:

- Completeness: No data should be lost during fragmentation. For vertical fragmentation, each column can be found in some fragment; in horizontal fragmentation each row can be found in a fragment.
- Reconstructability: Data from the fragments can be recombined to result in the original data set. For vertical fragmentation, the join operator is used on the tuple identifier to link the columns from the fragments; in horizontal fragmentation, the union

operator is used on the rows coming from the fragments.

- Non-redundancy: To avoid duplicate storage of data, data should be uniquely assigned to one fragment. In vertical fragmentation, each column is contained in only one fragment (except for the tuple identifier that links the fragments); in horizontal fragmentation, each row is contained in only one fragment.

In this paper we will compute semantically-guided horizontal fragmentations of a primary table. Each of these fragmentations will be based on clustering an attribute for which values should be relaxed to allow for flexible query answering. In contrast to the conventional applications of fragmentation, the clustering-based fragmentations will support flexible query answering in an efficient manner.

For other tables (those that can be joined with the primary table) a derived fragmentation will be computed that allows for data locality in a distributed database system.

Anti-instantiation

In this paper we focus on flexible query answering for conjunctive queries expressed as logical formulas. That is, we assume a logical language $\mathscr{L}$ consisting of a finite set of predicate symbols (denoting the table names; for example, *Ill*, *Treat* or *P*), a possibly infinite set *dom* of constant symbols (denoting the values in table cells; for example, *Mary* or *a*), and an infinite set of variables (x or y). A term is either a constant or a variable. The capital letter X denotes a vector of variables; if the order of variables in X does not matter, we identify X with the set of its variables and apply set operators – for example we write $y \in X$. We use the standard logical connectors conjunction $\wedge$, disjunction $\vee$, negation $\neg$ and material implication $\rightarrow$ and universal $\forall$ as well as existential $\exists$ quantifiers. An atom is a formula consisting of a single predicate symbol only; a literal is an atom (a "positive literal") or a negation of an atom (a "negative literal"); a clause is a disjunction of atoms; a ground formula is one that contains no variables; the existential (universal) closure of a formula ϕ is written as $\exists\phi$ ($\forall\phi$) and denotes the closed formula obtained by binding all free variables of ϕ with the respective quantifier.

A query formula Q is a conjunction of literals with some variables X occurring freely (that is, not bound by variables); that is, $Q(X) = L_{i_1} \wedge \ldots \wedge L_{i_n}$. By abuse of notation, we will also write $L_{ij} \in Q$ when L_{ij} is a conjunct in formula Q. A query $Q(X)$ is sent to a knowledge base Σ (a set of logical formulas) and then evaluated in Σ by a function *ans* that returns a set of answers containing instantiations of the free variables (in other words, a set of formulas that are logically implied by Σ); as we focus on the generalization of queries, we assume the *ans* function and an appropriate notion of logical truth given. A special case of a knowledge base can be a relational database with database tables as in Example 1.

Example 2. *Query $Q(x_1, x_2, x_3) = Ill(x_1, Flu) \wedge Ill(x_1, Cough) \wedge Info(x_1, x_2, x_3)$ asks for all the patient IDs x_1 as well as names x_2 and addresses x_3 of patients that suffer from both flu and cough. This query fails with the given database tables as there is no patient with both flu and cough. However, the querying user might instead be interested in the patient called Mary who is ill with both flu and asthma. Query generalization will enable an intelligent database system to find this informative answer.*

As in [3] we apply a notion of generalization based on a model operator $\models$.

Definition 1 (Deductive generalization [3]). *Let Σ be a knowledge base, $\phi(X)$ be a formula with a tuple X of free variables, and $\psi(X, Y)$ be a formula with an additional tuple Y of free variables disjoint from X. The formula $\psi(X, Y)$ is a deductive generalization of $\phi(X)$, if it holds in Σ that the less general ϕ implies the more general ψ where for the free variables X (the ones that occur in ϕ and possibly in ψ) the universal closure and for free variables Y (the ones that occur in ψ only) the existential closure is taken:*

$$\Sigma \models \forall X \exists Y \, (\phi(X) \rightarrow \psi(X, Y))$$

The CoopQA system [4] applies three generalization operators to a conjunctive query (which – among others – can already be found in the seminal paper of Michalski [5]): **Dropping Condition** (*DC*) removes one conjunct from a query; **Anti-Instantiation** (*AI*) replaces a constant (or a variable occurring at least twice) in Q with a new variable y; **Goal Replacement** (*GR*) takes a rule from Σ, finds a substitution θ that maps the rule's body to some conjuncts in the query and replaces these conjuncts by the head (with θ applied). In this paper we focus only on the AI operator.

Example 3. *For query $Q(x_1, x_2, x_3) = Ill(x_1, Flu) \wedge Ill(x_1, Cough) \wedge Info(x_1, x_2, x_3)$ an example generalization with AI is $Q^{AI}(x_1, x_2, x_3, y) = Ill(x_1, Flu) \wedge Ill(x_1, y) \wedge Info(x_1, x_2, x_3)$. A non-empty answer (and hence informative answer) $Ill(2748, Flu) \wedge Ill(2748, Asthma) \wedge Info(2748, Mary, \text{'New Str 3, Newtown'})$ is returned as an answer saying that Mary suffers from flu and asthma at the same time. However, another obtained answer is $Ill(2748, Flu) \wedge Ill(2748, brokenLeg) \wedge Info(2748, Mary, \text{'New Str 3, Newtown'})$ saying that Mary suffers from flu and a broken leg.*

AI applies to constants and to variables and covers these special cases:

- turning constants into variables: $P(a)$ is converted to $P(x)$ (see [5])
- breaking joins: $P(x) \wedge S(x)$ is converted to $P(x) \wedge S(y)$ (introduced in [3])

- naming apart variables inside atoms: $P(x, x)$ is converted to $P(x, y)$

For each constant a all occurrences can be anti-instantiated one after the other; the same applies to variables x – however, with the exception that if x only occurs twice, one occurrence of x need not be anti-instantiated due to equivalence. For logical queries, anti-instantiation can be implemented as shown in the listing in Listing 1.

Listing 1 Anti-instantiation (AI)

```
Input: Query $Q(X) = L_1 \wedge \ldots \wedge L_n$ of length $n$
Output: Generalized query $Q^{AI}(X, Y)$ with $Y$ containing
    one new variable
1: From $Q(X)$ choose a term $t$ such that $t$ is
     • either a variable occurring in $Q(X)$ at least twice
     • or a constant
2: Choose one literal $L_j$ where $t$ occurs
3: Let $L'_j$ be the literal with one occurrence of $t$ replaced
   with a new variable
4: return $L_1 \wedge \ldots \wedge L_{j-1} \wedge L'_j \wedge L_{j+1} \wedge \ldots \wedge L_n$
```

In this paper, we focus on the first application of anti-instantiation: turning constants into variables. In the following section, we present an approach that identifies those tuples in a relational table that are good candidates for answers to such an anti-instantiated query; these candidates are put into one fragment for storage in a distributed database system.

Data replication

To achieve fault tolerance, reliability and high availability, data in a distributed database system should be copied (that is, *replicated*) to different servers. Whenever one of the database servers fails, if it is too overloaded or geographically too far away from the requesting user, a data copy (that is, a *replica*) can be retrieved from one of the other servers.

The data replication problem (DRP; see [6]) is a formal description of the task of distributing copies of data records (that is, database fragments) among a set of servers in a distributed database system. The data replication problem is basically a Bin Packing Problem (BPP) in the following sense:

- K servers correspond to K bins
- bins have a maximum capacity W
- n fragments correspond to n objects
- each object has a weight (a capacity consumption) $w_i \leq W$
- objects have to be placed into a minimum number of bins without exceeding the maximum capacity

This BPP can be written as an integer linear program (ILP) as follows – where x_{ik} is a binary variable that denotes whether fragment/object i is placed in server/bin k; and y_k denotes that server/bin k is used (that is, is non-empty):

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

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

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

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

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

To explain, Equation 1 means that we want to minimize the number of servers/bins used; Equation 2 means that each object is assigned to exactly one bin; Equation 3 means that the capacity of each server is not exceeded; and the last two equations denote that the variables are binary – that is, the ILP is a so-called 0-1 linear program.

An extension of the basic BPP will be used to ensure that replicas will be placed on distinct servers: the Bin Packing with Conflicts (BPPC; [7-9]) problem allows constraints to be expressed on pairs of objects that should not be placed in the same bin. That is, one adds a conflict graph $G = (V, E)$ where the node set $V = \{1, \ldots, n\}$ corresponds to the set of objects. A binary edge $e = (i, j)$ exists whenever the two incident nodes i and j must not be placed in the same bin; note that (i, j) is meant to be undirected and hence identical to (j, i). In the ILP representation, a further constraint is added to avoid conflicts in the placements.

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

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

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

$$x_{ik} + x_{jk} \leq y_k \quad (i, j) \in E, \; k = 1, \ldots, K \tag{9}$$

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

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

Equation 9 ensures that no conflicting objects i and j are placed in the same bin k because otherwise the sum of the two x-variables x_{ik} and x_{jk} would be 2 and hence exceed y_k which is 1.

In this paper, we will extend the BPPC to ensure that a certain replication factor m for each fragment of the relational table is obeyed; that is, for each fragment stored at one server there are at least $m - 1$ other servers storing a copy of this fragment, too.

Clustering-based fragmentation

We now present our intelligent fragmentation and replication procedure that will support flexible query answering with anti-instantiation.

The anti-instantiation operator as stated above is a purely syntactic operator. For the application of turning constants into variables, any constant can be inserted in the answer. This syntactic operator is oblivious of whether the obtained answer is *semantically* close to the replaced constant in the original query or not. For example in Example 3, the two diseases cough and asthma are semantically closer to each other than the two diseases cough and broken leg. That is, the generalization operators can sometimes lead to overgeneralization where the generalized queries (and hence the obtained answers) are too far away from the user's original query intention. To avoid this overgeneralization and the overabundance of answers, a semantic guidance has to be added to the process. This semantic guidance can for example be given by a taxonomy on constants.

As an extension to [2], we will present a clustering heuristics attributes on which anti-instantiation should be applied. We call such attribute a *relaxation attribute*. The *domain* of an attribute is the set of values that the attribute may range over; whereas the *active domain* is the set of values actually occuring in a given table. For a given table instance F (a set of tuples ranging over the same attributes) and a relaxation attribute A, the active domain can be obtained by a projection π to A on F: $\pi_A(F)$. In our example the relaxation attribute is the attribute *Diagnosis* in table *Ill*. From a semantical point of view, the domain of *Diagnosis* is the set of strings that denote a disease; the active domain is the set of terms $\{Cough, Flu, Asthma, brokenArm, brokenLeg\}$.

Wiese 2013 [2] assumes a tree-shaped taxonomy on the active domain of a relaxation attribute where the active domain values can be found in the leave nodes connected by some intermediary nodes serving as a classification of the values. As an alternative, in this paper we only rely on the specification of a similarity value $sim(a, b)$ between any two values a and b in the active domain of a relaxation attribute. These similarity values, however, can indeed be calculated by using a taxonomy; we will briefly survey some of such similarity measures below when describing the prototype. Based on this similarity specification, we derive a clustering of the active domain of each relaxation attribute A in a relation instance F. We rely on a very general definition of a clustering as being a set of subsets (the *clusters*) of a larger set of values. For a clustering to be reasonable, similarities of any two values inside one cluster should somehow be larger than between any two values from different clusters. This will be ensured below by relying on so-called *head* elements in the clusters and on a threshold value α that restricts the minimal similarity allowed inside a cluster: if c_i is a cluster, then $head_i \in c_i$ and for any other value $a \in c_i$ (with $a \neq head_i$) it holds that $sim(a, head_i) \geq \alpha$ The clustering of the active domain of A induces a horizontal fragmentation of F into fragments $F_i \subseteq F$ such that the active domain of each fragment F_i coincides with one cluster; more formally, $c_i = \pi_A(F_i)$. For the fragmentation to be complete, we also require the clustering C to be complete; that is, if $\pi_A(F)$ is the active domain to be clustered, then the complete clustering $C = c_1, \ldots, c_n$ covers the whole active domain and no value is lost: $c_1 \cup \ldots \cup c_n = \pi_A(F)$. These requirements are summarized in the definition of a *clustering-based fragmentation* as follows.

Definition 2 (Clustering-based fragmentation). *Let A be a relaxation attribute; let F be a table instance (a set of tuples); let $C = \{c_1, \ldots c_n\}$ be a complete clustering of the active domain $\pi_A(F)$ of A in F; let $head_i \in c_i$; then, a set of fragments $\{F_1, \ldots, F_n\}$ (defined over the same attributes as F) is a clustering-based fragmentation if*

- *Horizontal fragmentation: for every fragment F_i, $F_i \subseteq F$*
- *Clustering: for every F_i there is a cluster $c_i \in C$ such that $c_i = \pi_A(F_i)$ (that is, the active domain of F_i on A is equal to a cluster in C)*
- *Threshold: for every $a \in c_i$ (with $a \neq head_i$) it holds that $sim(a, head_i) \geq \alpha$*
- *Completeness: For every tuple t in F there is an F_i in which t is contained*
- *Reconstructability: $F = F_1 \cup \ldots \cup F_n$*
- *Non-redundancy: for any $i \neq j$, $F_i \cap F_j = \emptyset$ (or in other words $c_i \cap c_j = \emptyset$)*

Approximation algorithm for clustering

We use and adapt an established approximation algorithm for clustering originally presented by Gonzalez [10]. Its original presentation relies on a notion of distance between any two values. It has a running time of $O(kf)$ for clustering a set of k objects into f clusters. Each cluster is represented by one or more so-called head values; and each value is assigned to the cluster represented by a head with minimal distance to the value. In case the distance measure is metric (in particular, satisfies the triangular inequation), Gonzalez showed that the number of heads obtained by his algorithm is at most twice as much as the optimal number of heads (in other words, it is a 2-approximation of the optimal solution).

Rieck et al. [11] apply this algorithm to malware detection. Instead of fixing the number f of clusters, they use a threshold for the distances of values inside a cluster to the cluster head; hence the number of obtained clusters can differ. This functionality is also needed in our application. We however rely on the notion of similarity between two values (instead of distance) and provide a reformulation of the clustering algorithm here based on [10,11]. The algorithm starts by assigning all values of the active domain to an initial cluster c_1, choosing an arbitrary element of it as $head_1$ and then step by step choosing other head elements $head_2, \dots, head_f$ that have lowest similarity to all other heads and moving other elements to the new clusters $c_2, \dots, c_f$; an element is moved to a new cluster when it has higher similarity to the new head element than to the old head element. The algorithm continues finding new heads until a threshold α is reached; α limits the minimum similarity that elements inside a cluster may have to their cluster heads. Listing 2 shows a pseudocode for the clustering procedure.

Listing 2 Clustering procedure

```
Input: Set $\pi_A(F)$ of values for attribute $A$, similarity threshold $\alpha$
Output: A set of clusters $c_1, \dots, c_f$
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 $head_{i+1} \in \{b \mid b \in c_j; b \neq head_j; sim(b, head_j) = sim_{min}; 1 \leq j \leq i\}$
7:     $c_{i+1} = \{head_{i+1}\} \cup \{c \mid c \in c_j; c \neq head_j; sim(c, head_j) \leq sim(c, head_{i+1}); 1 \leq j \leq i\}$
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
```

Note that the clustering obtained by this heuristic is always complete: any value of $\pi_A(F)$ is assigned to some cluster c_i. And we also have the property that clusters do not overlap: $c_i \cap c_j \neq \emptyset$ for each $i \neq j$.

Example 4. *In our example, we assume that the pairwise similarities for the values in the active domain of the relaxation attribute Diagnosis are given. We assume further that the pairwise similarities in the subset* $\{Cough, Flu, Asthma\}$ *and in the subset* $\{brokenArm, brokenLeg\}$ *are higher than any similarity in between these two subsets. In the first clustering step, we choose* $head_1$ *arbitrarily – let us assume Flu – and the entire active domain forms cluster* c_1. *Now as* $head_2$ *the value with the lowest similarity is chosen – let us assume brokenArm. Now, all values with higher similarity to brokenArm (than to Flu) are moved to cluster* c_2 *– which will then consist of* $\{brokenArm, brokenLeg\}$. *If we choose threshold* α *to lie in between the minimum intra-cluster (of both* c_1 *and* c_2*) similarity and the maximum inter-cluster similarity (between pairs of values from* c_1 *and* c_2*), we will stop after this second iteration.*

Fragmentation and lookup table

When considering only a single relaxation attribute A, we obtain a fragmentation of the corresponding table: a set of fragments F_i – each corresponding to a cluster c_i. A relational algebra expression for each fragment can be stated as follows (using the selection operator σ and a disjunction of equality conditions on A for each value a contained in the cluster):

$$F_i = \sigma_{condition(c_i)}(F)$$
$$\text{where } condition(c_i) = \bigvee_{a \in c_i}(A = a)$$

The selection operator results in a set of rows – hence a horizontal fragmentation is obtained. Because the clustering is complete, the fragmentation itself will also be complete; hence, in addition a reconstruction of the original instance F is possible by the union operator. Moreover, because clusters do not overlap, we also achieve non-redundancy in this fragmentation. Hence, all properties of Definition 2 will be ensured.

Example 5. *Based on the above clustering, we obtain two fragments of the Ill table.*

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

Fracture	PatientID	Diagnosis
	2784	*brokenLeg*
	1055	*brokenArm*

Fragmentation and replica management are usually supported by lookup tables [12] (also called root tables [13]) that store metadata – for example, information about in which fragment to look for matching tuples when a query arrives. In our case, as we enable flexible query answering in distributed database systems, we create a lookup table that contains:

- the fragment ID F_i that is used to solve the data replication problem
- the fragment name that will be used in queries to the fragment
- the head $head_i$ of the cluster c_i that was used to obtain the fragment F_i as a semantic representative of the values for relaxation attribute A inside fragment F_i
- the size w_i of fragment F_i that is used in the data replication problem; for simplicity, in this paper we

only count the number of rows – but more advanced size measures can be used, too
- an array of the IP addresses or names of the database servers that fragment F_i is assigned to

Example 6. *We insert the following data into the ROOT lookup table where ID is the fragment identifier, Name is the fragment name, S is the fragment size in number of tuples, and Host is the name of the server where the fragment is assigned to.*

ROOT	ID	Name	Head	S	Host
	F1	*Respiratory*	*Flu*	*4*	*NULL*
	F2	*Fracture*	*brokenArm*	*2*	*NULL*

The last missing information – identifying the database server hosting the fragment – is computed by solving a bin packing problem with conflicts (BPPC). The basic idea is that for f fragments we want to replicate m times, each fragment F_i is copied $m-1$ times: for F_i (and $1 \leq i \leq f$) we obtain the copies $F_{f+i}, F_{2f+i}, \ldots, F_{(m-1)f+i}$ so that the total number of fragments will be $n = f \cdot m$. Furthermore any two copies of fragment F_i (and including F_i itself) must not be placed on the same server; this will be ensured by a conflict graph where there exist edges between all pairs of copies of F_i.

Example 7. *In our example, when we assume a replication factor of $m = 2$, we have to copy each fragment once. Hence we have that $F_1 = F_3 =$ Respiratory each with a size of 4; and $F_2 = F_4 =$ Fracture each with a size of 2. The conflict graph then consists of nodes $V = \{F_1, F_2, F_3, F_4\}$ and edges $E = \{(F_1, F_3), (F_2, F_4)\}$.*

As input information for the BPPC we hence need:

- the capacity W of each of the database servers based on some configuration information of the distributed database system
- the replication factor m based on some configuration information of the distributed database system
- F_i as well as $m-1$ copies $F_{f+i}, F_{2f+i}, \ldots, F_{(m-1)f+i}$ of each F_i (where $1 \leq i \leq f$)
- the sizes w_i for each F_i (where $1 \leq i \leq n$) where the copies of a fragment have the same size as the fragment itself
- the conflict graph G where the set of $n = f \cdot m$ nodes is the set of fragments and their copies – that is, $V = \{F_1, F_2, \ldots, F_f, F_{f+1}, \ldots, F_n\}$ – and the set of undirected binary edges E consists of the sets E_i (where $1 \leq i \leq f$) of pairs (X, Y) of a fragment F_i and all its copies – that is, $E = \bigcup_{i=1}^{f} E_i$ where $E_i = \{(X, Y) \mid X, Y \in \{F_i, F_{f+i}, F_{2f+i}, \ldots, F_{(m-1)f+i}\}; 1 \leq i \leq f\}$.

When solving the usual ILP formulation of BPPC (as shown in Equations 6 to 11) with these inputs, we obtain a solution that occupies the minimal number of servers (bins) while respecting the different sizes w_i that the fragments (for a single relaxation attribute A) may have as well as ensuring the replication factor. An example with the ILP solver *lpsolve* is provided in an upcoming section.

As opposed to lookup tables for individual tuples [12], we only store a row per fragment (and only the appropriate cluster head). Due to this, the lookup tables are small and lookups can be faster. That is why we assume that there is only a master server for the lookup table; one hot backup server can be used that can take over the task of the master server in case of a failure. Alternatively, distribution of the lookup table to all replica servers can be used; however, this incurs extra overhead and consistency problems [12].

Query rewriting

Flexible query answering can now be executed on the obtained clustering-based fragmentation. Queries are rewritten and redirected to the appropriate fragment with the help of the lookup table as follows:

1. The user sends a query to the database system with a selection condition containing a constant a for the relaxation attribute A.
2. The database system checks if there is a head value $head_i$ in the lookup table such that $head_i = a$. Then the appropriate fragment F_i is already identified and the next three steps can be skipped.
3. Otherwise the database system reads all f head values from the lookup table.
4. The database system computes all similarities $sim(a, head_i)$ (for $1 \leq i \leq f$).
5. The database system chooses a head $head_i$ with maximum similarity to a and thereby identifies appropriate fragment F_i. A threshold β can be provided by the user to limit this similarity divergence.
6. The database system rewrites the query by replacing the original table name with the identified fragment name and removes the selection condition containing a for the relaxation attribute.
7. The rewritten query is redirected to the server that hosts the identified fragment.
8. The server can return the entire fragment for the rewritten query with the assertion that the distance threshold β is not exceed and hence the answers are relevant for the user.
9. If the query contains multiple selection conditions for the relaxation attribute, several query rewritings will be executed and theses queries can be redirected to different servers.

Example 8. *In the example query $Q(x_1, x_2, x_3) = Ill(x_1, Flu) \wedge Ill(x_1, Cough) \wedge Info(x_1, x_2, x_3)$ the constant Cough is anti-instantiated. The fragment matching the*

Cough constant is the one containing respiratory diseases because we assume that it holds that sim(Flu, Cough) > sim(brokenArm, Cough). The second constant for the relaxation attribute in this query is Flu; however, Flu is a head element of the corresponding fragment and hence no similarities have to be computed. The anti-instantiated query is

$$Q^{AI}(x_1, x_2, x_3, y, y') = Respiratory(x_1, y)$$
$$\wedge Respiratory(x_1, y') \wedge Info(x_1, x_2, x_3) \wedge y \neq y'$$

The inequality condition on the new variables is necessary to only obtain answers where the two disease values found in the Respiratory fragment differ. A distributed join on x_1 has to be executed to combine the data from the Info table with the data from the Respiratory fragment; we will later on discuss how this overhead can be avoided by using derived fragmentation. Because the query is redirected to the fragment with highest similarity, in this case only the first informative answer (see Example 3) with the disease asthma $Ill(2748, Flu) \wedge Ill(2748, Asthma) \wedge Info(2748, Mary, \text{'New Str 3, Newtown'})$ is returned. In contrast, the answer for the disease brokenLeg is suppressed because it resides in the Fracture fragment.

The computation of distributed joins cannot be avoided if subqueries must be redirected to different server. We argue however, that with any other conventional data replication scheme (like [12,14]), distributed joins have to be processed, too; while with our scheme we have added support for flexible query answering.

Example 9. *Consider the example query*

$$Q(x_1, x_2, x_3) = Ill(x_1, brokenLeg)$$
$$\wedge Ill(x_1, Cough) \wedge Info(x_1, x_2, x_3)$$

The query has to be rewritten into the query

$$Q^{AI}(x_1, x_2, x_3, y, y') = Fracture(x_1, y)$$
$$\wedge Respiratory(x_1, y') \wedge Info(x_1, x_2, x_3)$$

which has to be answered by both Fracture the and the Respiratory fragment. It may happen that the Respiratory, Fracture and Info tables all reside on different servers and so we would have to compute a three-way distributed join on x_1.

Improving data locality with derived fragmentations

Apart from failure tolerance and load balancing, another important issue for cloud storage is *data locality*: Data that are often accessed together should be stored on the same server in order to avoid excessive network traffic and delays. That is why we propose to compute a *derived fragmentation* for each table that shares join attributes with the primary table (for which the clustering-based fragmentation was computed). Each derived fragment should then be assigned to the same database server on which the primary fragment with the matching join attribute values resides.

Hence for a given fragmentation $\{F_1, \ldots, F_f\}$ of a primary table F we compute the corresponding fragmentation $\{G_1, \ldots, G_f\}$ of any table G sharing join attributes with F as a semijoin of G with each fragment F_i: $G_i = G \ltimes F_i$ – which is equivalent to the projection on the attributes of G of the natural join of G and F_i: $\pi_{Attr(G)}(G \bowtie F_i)$.

Example 10. *In our example we can join both the Treat as well as the Info table with the Ill table. Because we have two fragments of Ill, we obtain two derived fragments of Treat and Info as well: the first set of derived fragments is called Treat_resp and Info_resp based on a join on patient IDs occurring in the primary Respiratory fragment.*

Respiratory	PatientID	Diagnosis
	8457	Cough
	2784	Flu
	2784	Asthma
	8765	Asthma

Treat_resp	PatientID	Prescription
	8457	Inhalation
	2784	Inhalation
	8765	Inhalation
	2784	Plaster bandage

Info_resp	PatientID	Name	Address
	8457	Pete	Main Str 5, Newt.
	2784	Mary	New Str 3, Newt.
	8765	Lisa	Main Str 20, Oldt.

The second set of derived fragments is Treat_frac and Info_frac based on a join on patient IDs occurring in the primary Fracture fragment.

Fracture	PatientID	Diagnosis
	2784	brokenLeg
	1055	brokenArm

Treat_frac	PatientID	Prescription
	2784	Inhalation
	2784	Plaster bandage
	1055	Plaster bandage

Info_frac	PatientID	Name	Address
	2784	Mary	New Str 3, Newt.
	1055	Anne	High Str 2, Oldt.

Note that non-redundancy of derived fragments is difficult to achieve (this is also discussed in [1]). We opt for having some redundancy in the derived fragments for sake of better data locality and hence better performance of query answering. That is why the information for patient

Mary occurs in both derived fragments; the same applies to the treatment fragments.

Data replication for derived fragments

We maintain separate lookup tables for each (primary and derived) fragmentation of each table. Hence, the sizes of the derived fragments are also computed and stored in the corresponding lookup table. These sizes of the derived fragments must be taken into account for the data replication procedure and are encoded in the BPPC as follows. The capacity W, the replication factor m, the primary fragments and their $m-1$ copies as well as the conflict graph stay the same as before; the only input that changes is sizes w_i assigned to the fragments:

- the sizes w_i are now computed as the sum of the size of the primary fragment F_i plus the size of any derived fragment G_i.
- solving the BPPC results in a placement where the primary fragment fits on the server together with all its derived fragments.
- the primary fragment and its derived fragments are hence assigned to the same server and the server information in the lookup tables is inserted accordingly.

Implementation and example

Our prototype implementation is based on PostgreSQL and the UMLS::Similarity implementation. In the following subsections we describe the steps that the prototype executes.

UMLS and its similarity measures

The Unified Medical Language System incorporates several taxonomies from the medical domain like the Systematized Nomenclature of Medicine–Clinical Terms (SNOMED CT), or Medical Subject Headings (MeSH). It unifies these taxonomies assigning Concept Unique Identifiers (CUI) to terms so that shared terms in the different taxonomies have the same identifier.

The Perl program UMLS::Similarity [15] offers an implementation of several standard similarity measures. They can be differentiated into measures based solely on path lengths in a taxonomy and measures taking the so-called information content [16] into account. The information content (IC) is computed from a pre-assigned estimated probability $p(c)$ of each leave term in the taxonomy (assuming a parent-child or is-a relationship in the taxonomy); for inner nodes that subsume other terms, this probability must be larger than for any child node (for example, by summing over all child nodes) because this concept covers all its child concepts. The information content is then defined as the negative log likelihood: $-\log p(c)$. In this way, the higher a term is located in taxonomy, the more abstract the term it is, and the lower is its information content; where the unique root node of the taxonomy has IC 0 (or in other words, its probability is 1) – that is, no information content.

UMLS::Similarity offers implementations of the following measures based on path lengths:

- Path length (path) counts the nodes occurring on a path between two terms a and b and takes the inverse: $sim(a,b) = \frac{1}{length(path(a,b))}$.
- Leacock and Chodorow (lch) [17] use the length of the shortest path between two terms but also consider the overall maximum depth d_{max} of the taxonomy: $sim(a,b) = -\log \frac{length(path(a,b))}{2 \cdot d_{max}}$
- Wu and Palmer (wup) [18] consider the depth of terms – that is, the length of the path from the root node to the term. It first calculates the depths of the two terms and the depth of their least common subsumer (LCS) and then calculates similarity as twice the lcs depth divided by the sum of the depths of the two terms: $sim(a,b) = \frac{2 \cdot depth(lcs(a,b))}{depth(c)+depth(b)}$
- Conceptual distance (cdist) refers to the path length between two terms; while in the original case paths between terms were defined with respect to whether a meaning was narrower or broader ([19]), later on the paths in a parent-child (is-a) relationship were considered [20] – that is why in the latter case cdist coincides with path.
- Al-Mubaid and Nguyen (nam) [21] combine path length and depth into one measure; they consider the overall maximum depth d_{max} of the taxonomy, the depth of the least common subsumer of the two comparison terms, the shortest path length between the two terms. UMLS::Similarity returns the inverse of this distance measure, that is: $\frac{\log 2}{(length(path(a,b))-1)\cdot(d_{max}-depth(lcs(a,b)))+1}$

UMLS::Similarity offers implementations of the following measures incorporating information content (IC):

- Resnik (res) [16] proposed to use the information content of the least common subsumer (LCS): $IC(lcs(a,b))$
- Jiang and Conrath (jcn) [22] use the inverse of a distance that is based on the IC of the two terms and the IC of the least common subsumer: $sim(a,b) = \frac{1}{IC(a)+IC(b)-2 \cdot IC(lcs(a,b))}$
- Lin (lin) [23] takes twice the IC of the LCS and divides it by the sum of the ICs of the two terms: $sim(a,b) = \frac{2 \cdot IC(lcs(a,b))}{IC(a)+IC(b)}$

We used the UMLS::Similarity web interface with the MeSH taxonomy to obtain the pair-wise similarity of the set of terms *asthma*, *cough*, *influenza*, *tibial fracture* and *ulna fracture*. Figure 1 shows how the terms are related by a is-a relationship in the MeSH taxonomy. Table 1 shows the similarity values obtained. Due to symmetry of the terms in the taxonomy (the path lengths and LCSs are mostly identical), the similarity values do not differ much in the two subsets *asthma*, *cough* and *influenza*, as opposed to *tibial fracture* and *ulna fracture*; the only difference is obtained with the two measures where the IC of the respective terms *a*, *b* are taken into account – namely jcn and lin.

Clustering and fragmentation

The clustering heuristics has been implemented as a Java module that calls the UMLS::Similarity web interface. When using the clustering heuristics with the given similarities, regardless of which heads we choose, after two steps we obtain the two clusters {*asthma*, *cough*, *influenza*}, and {*tibial fracture*, *ulna fracture*}: let us assume, we choose *asthma* as $head_1$, then we compute all similarities to *asthma*. The one with the lowest similarity is *ulna fracture* – which is taken to be $head_2$. Because *tibial fracture* has lower similarity to *ulna fracture* than to *asthma*, it is assigned to c_2. For an appropriate threshold α (depending on the similarity measure chosen) the process could stop here. If instead we now continue the clustering, we would eventually obtain a total clustering consisting of only singleton sets: *tibial fracture* would become $head_3$ (because it has minimal distance to $head_2$); later on, *cough* would become $head_4$ and *influenza* would be $head_5$.

Choosing the path similarity and a threshold $\alpha = 0.15$ results in the two mentioned clusters. A fragmentation of the base table *Ill* can hence be obtained by computing the following materialized views in the Postgres database.

```
CREATE MATERIALIZED VIEW Respiratory AS
 SELECT * FROM Ill WHERE Diagnosis
 IN ('Cough', 'Influenza', 'Asthma')

CREATE MATERIALIZED VIEW Fracture AS
 SELECT * FROM Ill WHERE Diagnosis
 IN ('Tibial fracture', 'Ulna Fracture')
```

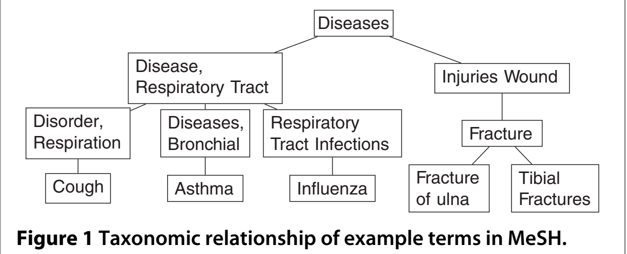

Figure 1 Taxonomic relationship of example terms in MeSH.

Sophisticated size estimations for these fragments might be possible as stated previously. We obtain the sizes of the fragment by counting the number of rows:

```
SELECT count(*) FROM Respiratory
```

and

```
SELECT count(*) FROM Fracture
```

Next, we fill a lookup table containing information as.

```
INSERT INTO root_ill VALUES
('F1', 'Respiratory', 'asthma', 4, NULL);
  INSERT INTO root_ill VALUES
('F2', 'Fracture', 'ulna fracture', 2,
  NULL);
```

To obtain the placement of the fragments to servers we model the corresponding BPPC and use the solver lp_solve [24]. lp_solve has a simple human-readable syntax and can be accessed by a Java program via the Java Native Interface (JNI). An example input for $K = 5$ (maximum number of servers), $W = 5$ (capacity per server), $m = 2$ (replication factor) looks as shown in Listing 3.

Listing 3 Input to ILP solver

```
min: y1 + y2 + y3 + y4 + y5;

x11 + x12 + x13 + x14 + x15 = 1;
x21 + x22 + x23 + x24 + x25 = 1;
x31 + x32 + x33 + x34 + x35 = 1;
x41 + x42 + x43 + x44 + x45 = 1;

4 x11 + 2 x21 + 4 x31 + 2 x41 <= 5 y1;
4 x12 + 2 x22 + 4 x32 + 2 x42 <= 5 y2;
4 x13 + 2 x23 + 4 x33 + 2 x43 <= 5 y3;
4 x14 + 2 x24 + 4 x34 + 2 x44 <= 5 y4;
4 x15 + 2 x25 + 4 x35 + 2 x45 <= 5 y5;

x11 + x31 <= y1; x21 + x41 <= y1;
x12 + x32 <= y2; x22 + x42 <= y2;
x13 + x33 <= y3; x23 + x43 <= y3;
x14 + x34 <= y4; x24 + x44 <= y4;
x15 + x35 <= y5; x25 + x45 <= y5;

bin x11,x12,x13,x14,x15;
bin x21,x22,x23,x24,x25;
bin x31,x32,x33,x34,x35;
bin x41,x42,x43,x44,x45;

bin y1,y2,y3,y4,y5;
```

The solution uses four servers (out of the five available ones): one for each of the two fragments and their copy. If the capacity is increased to $W = 6$, only two servers are used: the two fragments now fit on one server and the two copies on another server.

Table 1 Sample similarity obtained with UMLS::Similarity

	Asthma	Cough	Influenza	Tibial fracture	Ulna fracture
Asthma		(jcn) 0.3109	(jcn) 0.3844	(jcn) 0.1405	(jcn) 0.1282
	max	(cdist) 0.2	(cdist) 0.2	(cdist) 0.1429	(cdist) 0.1429
		(lin) 0.6175	(lin) 0.6662	(lin) 0.2116	(lin) 0.1968
		(wup) 0.6667	(wup) 0.6667	(wup) 0.5556	(wup) 0.5556
		(path) 0.2	(path) 0.2	(path) 0.1429	(path) 0.1429
		(res) 2.5963	(res) 2.5963	(res) 0.9555	(res) 0.9555
		(lch) 2.0794	(lch) 2.0794	(lch) 1.743	(lch) 1.743
		(nam) is 0.1621	(nam) 0.1621	(nam) 0.1483	(nam) 0.1483
Cough			(jcn) 0.2958	(jcn) 0.1266	(jcn) 0.1166
		max	(cdist) 0.2	(cdist) 0.1429	(cdist) 0.1429
			(lin) 0.6057	(lin) 0.1948	(lin) 0.1822
			(wup) 0.6667	(wup) 0.5556	(wup) 0.5556
			(path) 0.2	(path) 0.1429	(path) 0.1429
			(res) 2.5963	(res) 0.9555	(res) 0.9555
			(lch) 2.0794	(lch) 1.743	(lch) 1.743
			(nam) 0.1621	(nam) 0.1483	(nam) 0.1483
Influenza				(jcn) 0.1373	(jcn) 0.1256
			max	(cdist) 0.1429	(cdist) 0.1429
				(lin) 0.2079	(lin) 0.1936
				(wup) 0.5556	(wup) 0.5556
				(path) 0.1429	(path) 0.1429
				(res) 0.9555	(res) 0.9555
				(lch) 1.743	(lch) 1.743
				(nam) 0.1483	(nam) 0.1483
Tibial fracture					(jcn) 0.243
				max	(cdist) 0.3333
					(lin) 0.6295
					(wup) 0.7778
					(path) 0.3333
					(res) 3.4961
					(lch) 2.5903
					(nam) 0.1867
Ulna fracture					
					max

For improved efficiency, the root table is currently stored as a hash map in the Java frontend (instead of stored in a separate database table). The hash map is keyed by the head element, because the heads are necessary for the query rewriting module.

Query rewriting

The query rewriting procedure has to parse the SQL string inserted by the user. If a selection condition is given for the relaxation attribute, the root table is consulted to check for a matching head element. If none is found, again the UMLS::Similarity interface is consulted to obtain the similarities between head elements and the selection condition.

As a simple example considers the SQL query

```
SELECT * FROM Ill
WHERE Diagnosis = 'Bronchitis'
```

When comparing bronchitis to the first head (that is, asthma), UMLS::Similarity gives the following similarity results: 'The similarity of bronchitis (C0006277) and asthma (C0004096) using (jcn) is 1.0305, (cdist) is 0.3333, (lin) is 0.881, (wup) is 0.8, (path) is 0.3333, (res) is 3.5921, (lch) is 2.5903, (nam) is 0.1867'

Whereas comparing bronchitis to the second head (that is, tibial fracture), UMLS::Similarity gives the following similarity results: 'The similarity of bronchitis (C0006277) and tibial fracture (C0040185) using (jcn) is 0.1308, (cdist) is 0.1429, (lin) is 0.2, (wup) is 0.5556, (path) is 0.1429, (res) is 0.9555, (lch) is 1.743, (nam) is 0.1483'

Hence, asthma is more similar to bronchitis in every measure. The SQL query is rewritten by retrieving the appropriate fragment name and redirected to the appropriate server identified from the root table:

```
SELECT * FROM Respiratory
```

That is, the entire fragment is returned as it is the one with the most relevant answers for the user.

Experimental analysis

In general, any flexible query answering approach incurs a certain performance overhead compared to exact query answering. In our case, the clustering and fragmentation have to be compute but also the query answering incurs some extra overhead due to the fact that the appropriate fragment has to be identified and multiple answers are returned whereas exact query answering would simply have returned an empty answer set. With our approach however we aim to reduce this overhead by locating all related answers in the same fragment; any other fragmentation approach would need to recombine related answers from different fragments. For a performance evaluation of our prototype we used a test dataset consisting of values taken from the list of Medical Subject Headings (MeSH) [25]. The similarity computation during the clustering constituted an extreme overhead. That is why we computed pairwise similarities for 300 sample headings and stored these similarity values in a separate table. With these 300 values we randomly filled the disease column of a test table. We varied the row count between 10 and 1000 rows. Another parameter to vary is the threshold α for the intra-cluster similarity: the maximum similarity that values in a cluster may have to their respective head. For a higher threshold, more clusters are computed (and hence more similarity computations are executed) than for a lower threshold. We tested similarity thresholds 0.1, 0.125, 0.3 and 0.5. We ran the clustering and fragmentation algorithm on a PC with 1.73 GHz and 4GB RAM. The observed runtimes and number of obtained fragments are reported in Table 2. For the lower threshold values 0.1 and 0.125 runtimes are in the range of some seconds up to around 17 minutes. For a row count of 1000 rows the higher similarity values lead to a high number of fragments and runtimes are hence prohibitively high. Due to the high amount of pairwise comparisons, obtaining the similarity values is still the bottleneck of the clustering procedure. In future work we will follow two ways of improving scalability of the clustering: optimizing the access to the similarity values and investigating implications of a parallel implementation of the clustering procedure.

Related work

We divide the related work survey into approaches for flexible query answering and approaches for data fragmentation and replication.

Flexible query answering

The area of flexible query answering (sometimes also called cooperative query answering) has been studied extensively for single server systems. Some approaches have used taxonomies or ontologies for flexible query answering but did not consider their application for distributed storage of data: CoBase [26] used a type abstraction hierarchy to generalize values; Shin et al. [27] use some specific notion of metric distance in a knowledge abstraction hierarchy to identify semantically related answers; Halder and Cortesi [28] define a partial order between cooperative answers based on their abstract interpretation framework; Muslea [29] discusses the relaxation of queries in disjunctive normal form. Ontology-based query relaxation has also been studied for non-relational data (like XML data in [30]).

All these approaches address query relaxation at runtime while answering the query. This is usually prohibitively expensive. In contrast, our approach

Table 2 Runtime and fragment count obtained for MeSH dataset

	10 rows		100 rows		1000 rows	
α	Runtime (ms)	Fragment count	Runtime (ms)	Fragment count	Runtime (ms)	Fragment count
0.1	971	2	18200	4	709627	12
0.125	1211	3	32900	7	1038085	17
0.3	2658	8	201959	45	4132254	94
0.5	2415	10	244161	69	7428473	233

precomputes the clustering and fragmentation so that query answering does not incur a performance penalty.

Data fragmentation and replication

There are some approaches for fine-grained fragmentation and replication on object/tuple level; however none of these approaches support the flexible query answering application aimed at in this paper. In contrast they are mostly workload-driven and try to optimize the locality of data that are covered in the same query. However, they only support exact query answering. In contrast to this, we do not consider workloads but a generic clustering approach that can work with arbitrary workloads providing the feature of flexible query answering by finding semantically related answers. While some approaches are adaptive to updates, no quality guarantee after an update is reported. We intend to extend our approach in the future by bringing robust optimization to the data replication area. Loukopoulos and Ahmad [6] describe data replication as an optimization problem; they focus on fine-grained geo-replication for individual objects. They include an assumed number of reads and writes for each site as well as communication costs between sites. They reduce their problem to the Knapsack problem. In particular, they devise an adaptive genetic algorithm that can reallocate data to different sites. We aim to follow a different path to support this adaptive behavior: the notion of robust optimization is briefly discussed in Section Discussion and conclusion.

Curino et al. [14] represent database tuples as nodes in a graph. They assume a given transaction workload and add hyperedges to the graph between those nodes that are accessed by the same transactions. By using a standard graph partitioning algorithm, they find a database fragmentation that minimizes the number of cut hyperedges. In a second phase, they use a machine learning classifier to derive a range-based fragmentation. Then they make an experimental comparison between the graph-based, the range-based, a hash-based fragmentation on tuple keys and full replication. Lastly, they also compare three different kinds of lookup tables to map tuple identifier to the corresponding fragment: indexes, bit arrays and Bloom filters. Similar to them, we apply lookup tables to locate the replicated data; however we apply this to larger fragments and not to individual tuples.

Quamar et al. [31] also model the fragmentation problem as minimizing cuts of hyperedges in a graph; for efficiency reasons, their algorithm works on a compressed representation of the hypergraph which results in groups of tuples. In particular, the authors criticize the fine-grained (tuple-wise) approach in [14] to be impractical for large number of tuples which is similar to our approach. The authors propose mechanisms to handle changes in the workload and compare their approach to random and tuple-level partitioning.

Tatarowicz et al. [12] assume three existing fragmentations: hash-based, range-based and lookup tables for individual keys and compare those in terms of communication cost and throughput. For an efficient management of lookup tables, they experimented with different compression techniques. In particular they argue that for hash-based partitioning, the query decomposition step is a bottleneck. While we apply the notion of lookup tables, too, the authors do not discuss how the fragments are obtained, whereas we propose a semantically guided fragmentation approach here.

Discussion and conclusion

In this paper we proposed an intelligent fragmentation and replication approach for a distributed database system; with this approach, cloud storage can be enhanced with a semantically-guided flexible query answering mechanism that will provide related but still very relevant answers for the user. The approach combines fragmentation based on a clustering with data replication. For the user, this approach is totally invisible: he can send queries to the database system unchanged. The distributed database system autonomously computes the fragmentation (where the only additional information needed is the clustering backed by a taxonomy specific to the domain of the anti-instantiation column) and can use an automatic data replication mechanism that relies on the size information of each fragment and generates a bin packing input for an Integer Linear Programming (ILP) solver. As most of the related approaches, we assume a static dataset with mostly read-only accesses. When receiving a user query, the database system can autonomously rewrite the query and redirect subqueries to the appropriate servers based on the maintenance of a root table. The proposed method hence offers novel self-management and self-configuration techniques for a user-friendly query handling. While the user provides the original table and the desired similarity threshold as input, the database system can autonomously distribute the data while minimizing the amount of database servers. Hence we see our approach as a first step towards an intelligent cloud database system. For full applicability in a cloud database, automatic reconfiguration after updates, failure-tolerance as well as parallelization of our clustering approach (for example with map-reduce) will be necessary; these topics will be handled in future work.

The work presented in this paper can be extended in various research directions. We give a brief discussion of possible extensions.

- So far, the fragmentation process is only centered around a single relaxation attribute. The current approach can of course be executed in parallel for several relaxation attributes in parallel (with separate fragmentations and root tables for each relaxation attribute); however, this will lead to a massive (possibly unnecessary) replication of the data. We are currently investigating a more fine-grained support for multiple relaxation attributes with a more sophisticated data replication approach that can also be stated as a bin packing problem.

- In order to have a full-blown distributed flexible query answering system, the interaction of the proposed fragmentation with other generalization operators (like dropping condition and goal replacement) must be elaborated.
- When multiple fragments are assigned to one server, data locality can be improved by assigning fragments that are semantically close to each other to the same server.
- Our main focus for future work is to study the effect of updates on data (deletions and insertions) in the fragments: it must be studied in detail how fragments can be reconfigured and probably migrated to other server without incurring too much transfer cost.

Regarding the update problem, we plan to apply a special optimization approach to database replication: the notion of recovery robust optimization [32] describes optimization methods that compute a solution that can later on adapt to changing conditions which so far have been used mostly for timetabling applications [33] or job sequencing [8] or telecommunication networks [34]; in this respect it is important to ensure a worst case guarantee as in [8]. This is a different approach than presented here and its implications will hence be the topic of future work.

Competing interests
The author declares that she has no competing interests.

Acknowledgements
I would like to thank Anita Schöbel and Marie Schmidt for helpful discussions and Florian Henke for implementation and evaluation of the prototype. I acknowledge support by the German Research Foundation and the Open Access Publication Funds of Göttingen University.

Received: 3 June 2014 Accepted: 10 October 2014

References

1. Özsu MT, Valduriez P (2011) Principles of distributed database systems, Third Edition. Springer, Berlin/Heidelberg
2. Wiese L (2013) 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. IEEE, Washington, DC. pp 363–368
3. Gaasterland T, Godfrey P, Minker J (1992) Relaxation as a platform for cooperative answering. JIIS 1(3/4):293–321
4. Inoue K, Wiese L (2011) Generalizing conjunctive queries for informative answers. In: Flexible query answering systems. Springer, Berlin/Heidelberg. pp 1–12
5. Michalski RS (1983) A theory and methodology of inductive learning. Artif Intell 20(2):111–161
6. Loukopoulos T, Ahmad I (2004) Static and adaptive distributed data replication using genetic algorithms. J Parallel Distributed Comput 64(11):1270–1285
7. Gendreau M, Laporte G, Semet F (2004) Heuristics and lower bounds for the bin packing problem with conflicts. Comput OR 31(3):347–358
8. Epstein L, Levin A, Marchetti-Spaccamela A, Megow N, Mestre J, Skutella M, Stougie L (2010) Universal sequencing on a single machine. In: Integer programming and combinatorial optimization. Springer, Berlin/Heidelberg. pp 230–243
9. Sadykov R, Vanderbeck F (2013) Bin packing with conflicts: a generic branch-and-price algorithm. INFORMS J Comput 25(2):244–255
10. Gonzalez TF (1985) Clustering to minimize the maximum intercluster distance. Theor Comput Sci 38:293–306
11. Rieck K, Trinius P, Willems C, Holz T (2011) Automatic analysis of malware behavior using machine learning. J Comput Secur 19(4):639–668
12. Tatarowicz A, Curino C, Jones EPC, Madden S (2012) Lookup tables: Fine-grained partitioning for distributed databases. In: Kementsietsidis A, Salles MAV (eds). IEEE 28th International Conference on Data Engineering (ICDE 2012). IEEE Computer Society, Washington, DC. pp 102–113
13. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst 26(2):4:1–4:26
14. Curino C, Zhang Y, Jones EPC, Madden S (2010) Schism: a workload-driven approach to database replication and partitioning. Proc VLDB Endowment 3(1):48–57
15. McInnes BT, Pedersen T, Pakhomov SVS, Liu Y, Melton-Meaux G (2013) Umls::similarity: Measuring the relatedness and similarity of biomedical concepts. In: Vanderwende L, Daumé H III, Kirchhoff K (eds). Human language technologies: conference of the North American chapter of the association of computational linguistics. The Association for Computational Linguistics, Stroudsburg. pp 28–31
16. Resnik P (1999) Semantic similarity in a taxonomy: an information-based measure and its application to problems of ambiguity in natural language. J Artif Intell Res (JAIR) 11:95–130
17. Leacock C, Chodorow M (1998) Combining local context and wordnet similarity for word sense identification. WordNet: Electron Lexical Database 49(2):265–283
18. Wu Z, Palmer MS (1994) Verb semantics and lexical selection. In: Pustejovsky J (ed). 32nd annual meeting of the association for computational linguistics. Morgan Kaufmann Publishers/ACL, Stroudsburg. pp 133–138
19. Rada R, Mili H, Bicknell E, Blettner M (1989) Development and application of a metric on semantic nets. IEEE Trans Syst Man Cybernet 19(1):17–30
20. Caviedes JE, Cimino JJ (2004) Towards the development of a conceptual distance metric for the umls. J Biomed Inform 37(2):77–85
21. Al-Mubaid H, Nguyen HA (2006) New ontology-based semantic similarity measure for the biomedical domain. IEEE, Washington, DC. pp 623–628
22. Jiang JJ, Conrath DW (1997) Semantic similarity based on corpus statistics and lexical taxonomy. CoRR cmp-lg/9709008
23. Lin D (1998) An information-theoretic definition of similarity. In: Shavlik JW (ed). Proceedings of the fifteenth international conference on machine learning. Morgan Kaufmann, San Francisco. pp 296–304
24. lp_solve. http://lpsolve.sourceforge.net/
25. U.S. National Library of Medicine: Medical Subject Headings. http://www.nlm.nih.gov/mesh/
26. Chu WW, Yang H, Chiang K, Minock M, Chow G, Larson C (1996) CoBase: a scalable and extensible cooperative information system. JIIS 6(2/3):223–259
27. Shin MK, Huh S-Y, Lee W (2007) Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy. Expert Syst Appl 32(2):469–484
28. Halder R, Cortesi A (2011) Cooperative query answering by abstract interpretation. In: SOFSEM2011. LNCS, vol. 6543. Springer, Berlin/Heidelberg. pp 284–296
29. Muslea I (2004) Machine learning for online query relaxation. In: Knowledge Discovery and Data Mining (KDD). ACM, New York. pp 246–255

30. Hill J, Torson J, Guo B, Chen Z (2010) Toward ontology-guided knowledge-driven XML query relaxation. In: Computational Intelligence, Modelling and Simulation (CIMSiM). IEEE, Washington, DC. pp 448–453
31. Quamar A, Kumar KA, Deshpande A (2013) Sword: scalable workload-aware data placement for transactional workloads. In: Guerrini G, Paton NW (eds). Joint 2013 EDBT/ICDT conferences. ACM, New York. pp 430–441
32. Barber F, Salido MA (2014) Robustness, stability, recoverability, and reliability in constraint satisfaction problems. Knowl Inf Syst 41(2):1–16
33. Liebchen C, Lübbecke M, Möhring R, Stiller S (2009) The concept of recoverable robustness, linear programming recovery, and railway applications. In: Robust and online large-scale optimization. Springer, Berlin/Heidelberg. pp 1–27
34. Büsing C, Koster AM, Kutschka M (2011) Recoverable robust knapsacks: the discrete scenario case. Optimization Lett 5(3):379–392

doi:10.1186/s13677-014-0018-0
Cite this article as: Wiese: **Clustering-based fragmentation and data replication for flexible query answering in distributed databases.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:18.