Taxonomy-based Fragmentation for Anti-Instantiation in Distributed Databases

Lena Wiese
Institute of Computer Science
University of Göttingen
Goldschmidtstrasse 7
37077 Göttingen
Germany
Email: lena.wiese@uni-goettingen.de

***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 taxonomy-based fragmentation for the generalization operator Anti-Instantiation with which related information can be found in distributed data.**

I. Introduction

When data are stored in a cloud infrastructure, a distributed database system 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). For relational data, the theory of fragmentation has a long history (see for example [1]) and several procedure have been analyzed for splitting tabular data into fragments and subsequently assigning fragments to servers.

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.

In this paper we make the following contributions:

- We study how the values of a single relaxation attribute (that is, table column) can induce a horizontal fragmentation of the table based on a taxonomy on these values.
- We show how this taxonomy-based fragmentation can be used for flexible query answering by a query generalization operator called Anti-Instantiation.
- We compute the selectivity of fragments to enable load balancing on distributed database servers.

The paper is organized as follows. Sections II and III provide background on data fragmentation and query generalization (in particular anti-instantiation). Section IV presents the main contribution on taxonomy-based fragmentation; whereas Section V talks about how to decompose a query to be distributed among the servers. Section VI concludes the paper.

II. Data Fragmentation

We consider the case of data stored in relational tables.

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 a primary horizontal fragmentation based on a taxonomy of an attribute for which values should be generalized to allow for flexible query answering. For other tables (those that can be joined with the primary table) a derived fragmentation will be computed.

III. 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: The example query $Q(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ 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 [2] we apply a notion of generalization based on a model operator $\models$.

Definition 1 (Deductive generalization [2]): 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 [3] applies three generalization operators to a conjunctive query (which – among others – can already be found in the seminal paper of Michalski [4]): **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 $Q(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ an example generalization with AI is $Q^{AI}(x_1, x_2, x_3, y) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, y) \wedge \mathit{Info}(x_1, x_2, x_3)$. It results in an non-empty (and hence informative) answer: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{Asthma}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{`}\mathit{New\ Str\ 3}, \mathit{Newtown}\text{`})$. Another answer obtained is the fact that Mary suffers from a broken leg: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{brokenLeg}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{`}\mathit{New\ Str\ 3}, \mathit{Newtown}\text{`})$.

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

- turning constants into variables: $P(a)$ is converted to $P(x)$ (see [4])

- breaking joins: $P(x) \wedge S(x)$ is converted to $P(x) \wedge S(y)$ (introduced in [2])
- 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 listing Operator 1.

Operator 1 Anti-instantiation (AI)

Input: Query $Q(X) = L_1 \wedge \ldots \wedge L_n$ of length n
Output: Generalized query $Q^{gen}(X, Y)$ 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 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.

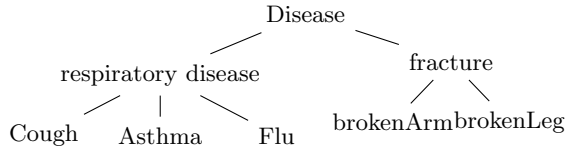
IV. Taxonomy-based Fragmentation

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.

For simplicity of the fragmentation process, we consider only a single attribute on which anti-instantiation should be applied. We call this attribute the *relaxation attribute*; in our example the relaxation attribute is the attribute *Diagnosis* in table *Ill*. 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. 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 strings $\{Cough, Flu, Asthma, brokenArm, brokenLeg\}$.

A *taxonomy* over an attribute is a tree structure where the leaves are the constants from the domain of the attribute. The inner nodes are more abstract descriptions of the leave nodes in the subtree below them. That is, each inner node represents the set of leave nodes in its subtree. For our purposes is suffices to have a taxonomy over the active domain of the relaxation attribute plus the constants that may occur in a query for the relaxation attribute.

Example 4: A simple taxonomy for the active domain of the *Diagnosis* attribute could be the following:



The inner node *respiratory disease* is the common ancestor of the leave nodes *cough*, *asthma* and *flu* and hence represents this set of constants.

Usually taxonomies are constructed manually by a domain expert. However, automatic construction of taxonomies has already been discussed in [5] and is still a hot topic of research (see for example [6]). Here we assume that a good taxonomy is provided which the fragmentation process can rely on.

We now discuss how clusters of constants can be found in the taxonomy; these clusters later induce the horizontal fragmentation. A *cluster* of constants is a subset of the leaves which are semantically close; a cluster can be represented by the lowest common ancestor of all constants in the cluster. To define semantic closeness, we use the following notion of a distance between two leaves in the taxonomy.

Definition 2 (Path-based distance): Let l_1 and l_2 be two leaves in a taxonomy, let anc be their lowest common ancestor node, let $root$ be the root node of the taxonomy. Let pl denote the length of the shortest path between two nodes in the taxonomy (that is, the number of edges between the two nodes). Then the *path-based distance* between the two leaves is defined as the fraction of the shortest path between the two nodes (via their common ancestor) and the longest path between the two (via the root node):

$$dist(l_1, l_2) = \frac{pl(l_1, anc) + pl(l_2, anc)}{pl(l_1, root) + pl(l_2, root)} \quad (1)$$

$$= \frac{pl(l_1, l_2)}{pl(l_1, root) + pl(l_2, root)} \quad (2)$$

This distance measure ranges between 0 (for identical nodes) and 1 (for nodes whose shortest path is via the root node).

Example 5: The distance between *cough* and *asthma* is given by the length of the shortest path between the two via their common ancestor *respiratory disease* (which is 2) divided by the sum of the lengths of the paths to the root node (which is 2 for each path): $dist(cough, asthma) = \frac{2}{2+2} = \frac{1}{2}$. The distance between *cough* and *brokenLeg* is 1 (because their shortest path goes via the root node): $dist(cough, brokenLeg) = \frac{4}{2+2} = 1$.

Other distance measures or similarity measures have been studied (for example, [7], [8]) and could equally well be applied here.

For an inner node we can obtain the maximum of the differences between any two leave nodes below the inner node

and obtain an intra-cluster distance:

Definition 3 (Intra-cluster distance): Let v be an inner node and let l_1 and l_2 be two arbitrary leaves in the subtree starting at v. Then the *intra-cluster distance* for v is defined as the maximum distance between any two l_1 and l_2:

$$cdist(v) \quad = \quad max_{l_1,l_2}\, dist(l_1, l_2) \tag{3}$$

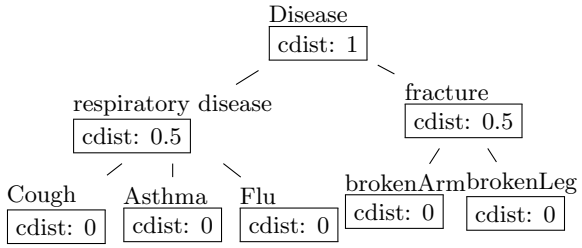
Now we can annotate each node in the taxonomy with its intra-cluster distance. Let us assume that the user specifies a threshold (denoted α) for the distances he is willing to tolerate when answering his query. The database system then identifies those inner nodes v – representing *disjoint* clusters – for which the intra-cluster distance does not exceed the threshold, that is $cdist(v) \leq \alpha$; these inner nodes represent the clusters. A taxonomy-based fragmentation is then one that is induced by these clusters:

Definition 4 (Taxonomy-based fragmentation): Let A be the relaxation attribute; let T be a tree-shaped taxonomy for A annotated with intra-cluster distances; for any node v in T, let $cluster(v)$ be the set of constants in the leave nodes of the subtree starting from v; let α be the threshold for the intra-cluster distance; let F be a table instance (a set of tuples); then, a set of fragments $\{F_1, \ldots, F_n\}$ (defined over the same attributes as F) is a *taxonomy-based* fragmentation if

- Vertical fragmentation: for every set of tuples F_i, $F_i \subseteq F$
- Clustering: for every F_i there is a node v_i in T such that $cluster(v_i) = \pi_A(F_i)$ (that is, the projection of F_i on A contains all values in cluster v_i)
- Threshold: for each such v_i, $cdist(v_i) \leq \alpha$
- 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,j, $F_i \cap F_j = \emptyset$ (or in other words $cluster(v_i) \cap cluster(v_j) = \emptyset$ due to the tree properties of T)

One could take the maximum clusters for which the threshold α is not exceeded; but also smaller disjoint clusters with lower intra-cluster distance are allowed.

Example 6: The taxonomy for the relaxation attribute *Diagnosis* annotated with the intra-cluster distances is as follows:



For a distance threshold $\alpha = 0.5$, we obtain two clusters: one for respiratory diseases and one for fractures. These two clusters induce two fragments on the relation *Ill*. These two fragments can be defined as materialized views with the following SQL statements.

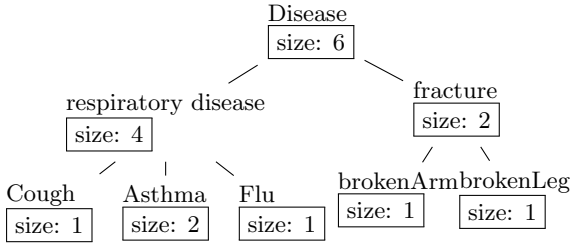
```
CREATE VIEW Respiratory AS
 SELECT * FROM Illness WHERE Diagnosis
 IN ('Cough', 'Flu', 'Asthma')

CREATE VIEW Fracture AS
 SELECT * FROM Illness WHERE Diagnosis
 IN  ('brokenLeg', 'brokenArm')
```

Given a taxonomy-based fragmentation, the database system can now start to allocate fragments to different servers in the distributed database system (or the cloud storage infrastructure). In order to make *load balancing* possible, we have to measure the size of each fragment. Here we choose the notion of selectivity of a fragment:

Definition 5 (Selectivity): The *selectivity* of a fragment F_i is the number of tuples contained in F_i (that is, its cardinality).

Again, other size measures can be chosen; for example, considering the exact storage size for each tuple. Based on the size annotations for each cluster, load balancing can now be executed to distribute the fragments among the servers in the network; the allocation should be such that each server stores data according to its storage capacities.

Example 7: The selectivity of each cluster can be annotated on the nodes in the taxonomy as follows:



Selectivity annotations for leaves in the taxonomy can be obtained by SQL statements similar to the following:

```
SELECT count(ID) FROM Illness
 WHERE Diagnosis = 'Cough'
```

Selectivity annotations for inner nodes in taxonomy can either be obtained by summing the selectivity annotations of the direct child nodes; or – if only the selectivity of the inner node has to be obtained without computing the one of the child nodes – with SQL statements similar to the following:

```
SELECT count(ID) FROM Illness
 WHERE Diagnosis
 IN ('Cough', 'Flu', 'Asthma')
```

Apart from 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 taxonomy-based fragmentation was computed). Each derived fragment should then be assigned to same database server on which the primary fragment with the matching join attribute values resides. The sizes of the derived fragments can also be computed and taken into account for the load balancing procedure.

Example 8: In our example, joins between the tables *Ill*, *Treat* and *Info* are possible based on the join attribute *Patien-*

tID. Based on each primary fragment of *Ill*, derived fragments can be obtained with SQL statements similar to the following (which computes a derived fragment on the *Info* table for the primary fragment containing respiratory diseases).

```
CREATE VIEW Info_resp AS
 SELECT * FROM Info WHERE PatientID
 IN (SELECT ID FROM Respiratory)
 AND Respiratory.PatientID=Info.PatientID
```

The entire fragmentation assigned to two servers then looks as follows:

Server 1:

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, Newtown
	2784	Mary	New Str 3, Newtown
	8765	Lisa	Main Str 20, Oldtown

Server 2:

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, Newtown
	1055	Anne	High Str 2, Oldtown

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.

V. Query Decomposition

At last, the flexible query answering can be executed on the obtained taxonomy-based fragmentation as follows:

1) The user sends a query to the database system with a constant for the relaxation attribute which should be anti-instantiated.
2) The database system identifies the cluster in the taxonomy in which the query constant lies.
3) The database system may split the query into subqueries and assign them to different servers.
4) The database system redirects the subquery which was anti-instantiated to the server that hosts the identified cluster.
5) The server can return the entire fragment for the subquery with the assertion that the distance threshold α is not exceed and hence the answers are relevant for the user.
6) The server may process other subqueries (for example joining the primary and the derived fragments).

Example 9: In the example query $Q(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ the constant *Cough* is anti-instantiated. The fragment matching the *Cough* constant is the one containing respiratory diseases. The anti-instantiated query $Q^{AI}(x_1, x_2, x_3, y) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, y) \wedge \mathit{Info}(x_1, x_2, x_3)$ is sent to the server hosting the *Respiratory* fragment. It executes the query on the primary *Respiratory* fragment and on the derived *Info_resp* fragment and returns the results to the user with the guarantee that the distances of the values in the relaxation variable are at most 0.5. That is, in this case only the first informative answer (see Example 3) with the disease asthma $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{Asthma}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{`}\mathit{New\ Str\ 3}, \mathit{Newtown}\text{`})$ is returned. In contrast, the answer for the disease brokenLeg is suppressed because it resides in the *Fracture* fragment.

In the above example, the generalized query was entirely sent to the appropriate server. However, usually the query has to be split into subqueries which have to be sent to distinct servers and later on be combined by the union operator.

Example 10: In the example query $Q(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{brokenLeg}) \wedge \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ the query has to be split into two subqueries $Q_1(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{brokenLeg}) \wedge \mathit{Info}(x_1, x_2, x_3)$ (which has to be answered by the *Fracture* fragment on Server 2) and $Q_2(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ (which has to be answered by the *Respiratory* fragment on Server 1).

To sum up, query decomposition for queries that cover more than one fragment is still a topic that requires further studies. Moreover, selection conditions on attributes other than the relaxation attribute lead to a search on all fragments, and hence the query has to be redirected to all servers that host such a fragment – however, this search can be improved with appropriate indexes.

Nevertheless, the proposed taxonomy-based fragmentation works well in the following cases:

- the query uses a selection condition over the relaxation attribute such that subqueries can be sent to the appropriate servers and can be anti-instantiated.
- the query computes a join between primary and derived fragment based based on the join attribute with which the derived fragmentation was obtained.

VI. Discussion and Conclusion

In this paper we proposed an Anti-instantiation approach for a distributed database system; with this approach, cloud

storage can be enhanced with an intelligent flexible query answering mechanism. The approach combines fragmentation based on a taxonomy with load balancing. 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 taxonomy) and can use an automatic load balancer that relies on the size information of each fragment. When receiving a user query, the database system can automatically decompose the query and redirect subqueries to the appropriate servers.

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 the their application for distributed storage of data: CoBase [5] used a type abstraction hierarchy to generalize values; Shin et al [8] use some specific notion of metric distance in a knowledge abstraction hierarchy to identify semantically related answers; Halder and Cortesi [9] define a partial order between cooperative answers based on their abstract interpretation framework; Muslea [10] 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 [11]).

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. It must be investigated how a more flexible choice of the relaxation attribute or the support for multiple relaxation attributes can be achieved.
- 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.
- To achieve a more adaptive load balancing, the interplay of distance and selectivity must be formalized; for example, a threshold β for the selectivity can be defined and fragments can be chosen such that none of the two thresholds is exceeded.
- A practical study on a large-scale distributed database system must be devised.
- Conditions for the non-redundancy of derived fragments must be established; for example, dependencies between data (like dependencies between diagnoses and treatments) can be considered.
- 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.
- The effect of updates on data in the fragments must be studied in detail.

REFERENCES

[1] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[2] T. Gaasterland, P. Godfrey, and J. Minker, "Relaxation as a platform for cooperative answering," *JIIS*, vol. 1, no. 3/4, pp. 293–321, 1992.

[3] K. Inoue and L. Wiese, "Generalizing conjunctive queries for informative answers," in *Flexible Query Answering Systems*. Springer, 2011, pp. 1–12.

[4] R. S. Michalski, "A theory and methodology of inductive learning," *Artificial Intelligence*, vol. 20, no. 2, pp. 111–161, 1983.

[5] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson, "CoBase: A scalable and extensible cooperative information system," *JIIS*, vol. 6, no. 2/3, pp. 223–259, 1996.

[6] M. B. Blaschko and A. Gretton, "Learning taxonomies by dependence maximization," in *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2008, pp. 153–160.

[7] P. Resnik, "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language," *Journal of Artificial Intelligence Research (JAIR)*, vol. 11, pp. 95–130, 1999.

[8] M. K. Shin, S.-Y. Huh, and W. Lee, "Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy," *Expert Systems with Applications*, vol. 32, no. 2, pp. 469–484, 2007.

[9] R. Halder and A. Cortesi, "Cooperative query answering by abstract interpretation," in *SOFSEM2011*, ser. LNCS, vol. 6543. Springer, 2011, pp. 284–296.

[10] I. Muslea, "Machine learning for online query relaxation," in *Knowledge discovery and data mining (KDD)*. ACM, 2004, pp. 246–255.

[11] J. Hill, J. Torson, B. Guo, and Z. Chen, "Toward ontology-guided knowledge-driven xml query relaxation," in *Computational Intelligence, Modelling and Simulation (CIMSiM)*, 2010, pp. 448–453.