Ontology-Driven Data Partitioning and Recovery for Flexible Query Answering

Lena Wiese

Institute of Computer Science
University of Göttingen
Goldschmidtstraße 7
37077 Göttingen
Germany
lena.wiese@uni-goettingen.de

Abstract. Flexible Query Answering helps users find relevant information to their queries even if no exactly matching answers can be found in a database system. However, relaxing query conditions at runtime is inherently slow and does not scale as the data set grows. In this paper we propose a method to partition the data by using an ontology that semantically guides the query relaxation. Moreover, if several different partitioning strategies are applied in parallel, a lookup table is maintained in order to recover the ontology-driven partitioning in case of data loss or server failure. We tested performance of the partitioning and recovery strategy with a distributed SAP HANA database.

Keywords: Query Relaxation, Anti-Instantiation, Recovery, Distributed Database

1 Introduction

When storing large-scale data sets in distributed database systems, these data sets are usually *partitioned* into smaller subsets and these subsets are distributed over several database servers. When answering queries in such a distributed database system, it might be necessary to contact several servers to collect matching data records. It is hence worthwhile to improve *data locality* of the partitioning approach such that the amount of servers involved in answering a single query is reduced. In this paper we improve data locality for a partitioning that can be used in an *intelligent query answering* system. These intelligent query answering mechanisms are increasingly important to find relevant answers to user queries. Flexible (or cooperative) query answering systems help users of a database system find answers related to his original query in case the original query cannot be answered exactly. *Semantic* techniques rely on taxonomies (or ontologies) to replace some values in a query by others that are closely related according the taxonomy. This can be achieved by techniques of *query relaxation* – and in particular *query generalization*: the user query is rewritten into a weaker, more general version to allow for related answers.

In this paper we make the following contributions:

- we introduce ontology-driven data partitioning that is based on a semantic clustering of values in a column,
- we apply different partitioning strategies (ontology-driven vs. round robin) to several columns of a database table with the aim to show improved data locality for flexible query answering in a distributed database,
- we describe a recovery procedure based on a replicated lookup table,
- we present performance tests of the query answering procedure as well as the recovery procedure based on the Medical Subject Headings (MeSH) in a distributed SAP HANA database that shows that ontology-driven partitioning leads to lower execution times for flexible query answering with less servers involved while still allowing for a fast recovery.

1.1 Organization of the article

Section 2 introduces the main notions used in this article and gives an illustrative example. Section 3 describes ontology-driven query answering, Section 4 shows query answering with derived partitions, Section 5 analyzes the update behavior, Section 6 discusses deletions and Section 7 presents a recovery procedure. Related work is presented in Section 8 and Section 9 concludes this article with suggestions for future work.

2 Background and example

2.1 Query generalization

Query generalization has long been studied in flexible query answering [13]. Query generalization at runtime has been implemented in the CoopQA system [8] by applying three generalization operators to a conjunctive query. *Anti-Instantiation* (AI) is one query generalization operator that replaces a constant (or a variable occurring at least twice) in a query with a new variable y. In this paper we focus on replacements of constants because this allows for finding answers that are semantically close to the replaced constant.

As the query language we focus on conjunctive queries expressed as logical formulas. 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 quantifier.

A query formula Q is a conjunction of literals with some variables X occurring freely (that is, not bound by variables); that is, $Q(X) = L_{i_1} \wedge \ldots \wedge L_{i_n}$. The Anti-Instantiation (AI) operator chooses a constant a in a query $Q(X)$, replaces one occurrence of a by a new variable y and returns the query $Q^{AI}(X, y)$ as the relaxed query. The relaxed query Q^{AI} is a deductive generalization of Q.

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

The 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. We can find this informative answer by relaxing the query condition *Cough* and instead allowing other related values (like *Asthma*) in the answers. An example generalization with AI is $Q^{AI}(x_1, x_2, x_3, y) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, y) \wedge \mathit{Info}(x_1, x_2, x_3)$ by introducing the new variable y. It results in an non-empty (and hence informative) answer: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{Asthma}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{‘}\mathit{New\ Str\ 3}, \mathit{Newtown}')$. Another answer obtained is the fact that Mary suffers from a broken leg as: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{brokenLeg}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{‘}\mathit{New\ Str\ 3}, \mathit{Newtown}')$.

As can be seen from the example query Q^{AI}, query relaxation by anti-instantiation can go too far and lead to *overgeneralization*: while the first example answer (with the value asthma) is a valuable informative answer, the second one (containing broken leg) might be too far away from the user's query interest. Here we need semantic guidance to identify the set of relevant answers that are close enough to the original query.

2.2 Ontology-Driven Partitioning

In previous work [21], a clustering procedure was applied to partition the original tables into partitions based on single *relaxation attribute* chosen for anti-instantiation; whereas concomitant research [22] handles intelligent replication for *multiple* relaxation attributes. Partitioning is achieved by grouping (that is, *clustering*) the values of the respective table column (corresponding to the relaxation attribute) and then splitting the table into partitions according to the clusters found. The clustering relies on a similarity metrics which is derived from paths ("proximity") between any two terms in an ontology or taxonomy.

We assume that each of the clusterings (and hence the corresponding partitioning) is *complete*: every value in the column is assigned to one cluster and hence every tuple is assigned to one partition. We also assume that each clustering and each partitioning are also *non-redundant*: every value is assigned to exactly one cluster and every tuple belongs to exactly one partition (for one of the relaxation attributes); in other words, the partitions inside one partitioning do not overlap.

More formally, we apply the clustering approach described in [21] (or any other method to semantically split the attribute domain into subsets) on the relaxation attribute, so that each cluster inside one clustering is represented by a *head* term (also called prototype) and each term in a cluster has a similarity *sim* to the cluster head above a certain threshold α. We then obtain a clustering-based partitioning for the original table F into partitions as specified in the following definition.

Definition 1 (Clustering-based partitioning). *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 partitions $\{F_1, \ldots, F_n\}$ (defined over the same attributes as F) is a* clustering-based *partitioning if*

- *Horizontal partitioning: for every partition 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 $c_i \cap c_j = \emptyset$)*

For example, clusters on the `Diagnosis` column can be made by differentiating between fractures on the one hand and respiratory diseases on the other hand. These clusters then lead to two partitions of the table `Ill` that can be assigned to two different servers:

Server 1:

Respiratory	PatientID	Diagnosis
	8457	Cough
	2784	Flu
	2784	Asthma
	8765	Asthma

Server 2:

Fracture	PatientID	Diagnosis
	2784	brokenLeg
	1055	brokenArm

Server 1 can then be used to answer queries related to respiratory diseases while Server 2 can process queries related to fractures. The example query $Q(x_1, x_2, x_3) = \mathit{Ill}(x_1, \mathit{Flu}) \wedge \mathit{Ill}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ will then be rewritten into $Q^{Resp}(x_1, x_2, x_3, y) = \mathit{Respiratory}(x_1, y) \wedge \mathit{Respiratory}(x_1, \mathit{Cough}) \wedge \mathit{Info}(x_1, x_2, x_3)$ and redirected to Server 1 where only the partition Respiratory is used to answer the query. In this way only the informative answer containing asthma is returned – while the one containing broken leg will not be generated. The most important advantage of the ontology-driven partitioning is that for answering the relaxed query only a single database server is contacted; with any other partitioning strategy that distributes data among the servers (that is, based on ranges or hash values or in round robin fashion), the relevant data might be distributed across several servers. If several tables are partitioned, *derived partitioning* can be used to store the matching tuples of the other tables (joinable by patient ID like the Info table) and hence improve data locality during processing of join queries.

2.3 Derived Partitioning

When having several tables that can be joined in a query, *data locality* is important for performance: Data that are often accessed together should be stored on the same server in order to avoid excessive network traffic and delays. If one table is chosen as the primary clustering table (like `Ill` in our example), partitioning of related tables (like `Treat` and `Info` in our example) can be derived from the primary partitioning. They are obtained by computing a semijoin between the primary table and the secondary table. Each derived partition should then be assigned to the same database server on which the primary partition with the matching join attribute values resides. Note that while the primary partitioning is usually non-redundant (each tuple of the original table is contained in exactly one partition), that might not be the case for derived partitioning: one tuple of a joinable tables might be contained in several derived partitions.

In the example, the entire partitioning for clusters on the `Diagnosis` column 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

3 Ontology-driven Query Answering

To enable ontology-driven query answering, when a user sends a query to the database, the term (that is, constant) that can be anti-instantiated has to be extracted, the matching cluster has to be identified and then the user query has to be rewritten to return answers covering the entire cluster.

3.1 Metadata and Test Dataset

In order to manage the partitioning, several metadata tables are maintained:

- A **root** table stores an ID for each cluster (column *clusterid*) as well as the cluster head (column *head*) and the name of the server that hosts the cluster (column *serverid*).
- A **lookup** table stores for each cluster ID (column *clusterid*) the tuple IDs (column *tupleid*) of those tuples that constitute the clustered partition.
- A **similarities** table stores for each head term (column *head*) and each other term (column *term*) that occurs in the active domain of the corresponding relaxation attribute a similarity value between 0 and 1 (column *sim*).

Our prototype implementation – the OntQA-Replica system – runs on a distributed SAP HANA installation which is an in-memory database system and hence shows a fast execution without disk accesses. All runtime measurements shown below are taken as the median of several (at least 5) runs per experiment.

The example data set consists of a table (called *ill*) that resembles a medical health record and is based on the set of Medical Subject Headings (MeSH [20]). The table contains as columns an artificial, sequential *tupleid*, a random *patientid*, and a *disease* chosen from the MeSH data set as well as the *concept* identifier of the MeSH entry. We varied the table sizes during our test runs. The smallest table consists of 56,341 rows (one row for each MeSH term), a medium-sized table of 1,802,912 rows and the largest of 14,423,296 rows (obtained by duplicating the original data set 5 times and 8 times, respectively). A clustering is executed on the MeSH data based on the concept identifier (which orders the MeSH terms in a tree); in other words, entries from the same subconcept belong to the same cluster. One partitioning (the clustered partitioning) was obtained from this clustering and consists of 117 partitions which are each stored in a smaller table called *ill-i* where i is the cluster ID. To allow for a comparison and a test of the recovery strategy, another partitioning of the table was done using round robin resulting in a table called *ill-rr*; this distributes the data among the database servers in chunks of equal size without considering their semantic relationship; these partitions have an extra column called *clusterid*.

3.2 Identifying matching Clusters

Flexible Query Answering intends to return those terms belonging to the same cluster as the query term as informative answers. Before being able to return the related terms, we hence have to identify the matching cluster: that is, the ID of the cluster the head of which has the *highest* similarity to the query term. We do this by consulting the similarities table and the root table. The similarities are derived by using the Unified Medical Language System as an ontology (or more precisely is-a hierarchy) and the similarity interface on top of it [12]. The relaxation term t is extracted from the query and then the top-1 entry of the similarities table is obtained when ordering the similarities in descending order:

```
SELECT TOP 1 root.clusterid FROM root, similarities
 WHERE similarities.term='t' AND similarities.head = root.head
 ORDER BY similarities.sim DESC
```

The query was tested on similarities tables of sizes 56341 entries, 14423296 entries and 72116480 entries. The runtime measurements in Figure 1 show a decent performance of at most 125 ms impact even for the largest table size.

3.3 Query Rewriting Strategies

After having obtained the ID of the matching cluster, the original query has to be rewritten in order to consider all the related terms as valid answers. We tested and compared three query rewriting procedures:

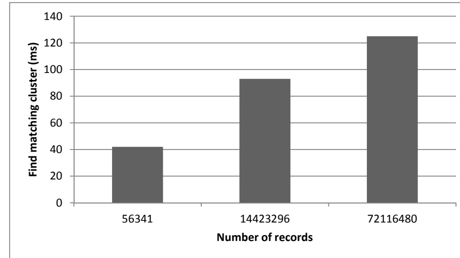

Fig. 1. Identify matching cluster

- lookup table: the first rewriting approach uses the lookup table to retrieve the tuple IDs of the corresponding rows and executes a JOIN on table *ill*.
- extra clusterid column: the next approach relies on the round robin table and retrieves all relevant tuples based on a selection predicate on the clusterid column.
- clustered partitioning: the last rewriting approach replaces the occurrences of the *ill* table by the corresponding *ill-i* table for clusterid *i*.

Assume the user sends a query

```
SELECT mesh,concept,patientid,tupleid
 FROM ill WHERE mesh ='cough'.
```

and 101 is the ID of the cluster containing cough. In the first strategy (lookup table) the rewritten query is

```
SELECT mesh,concept,patientid,tupleid  FROM ill JOIN lookup
 ON (lookup.tupleid = ill.tupleid AND lookup.clusterid=101).
```

In the second strategy (extra clusterid column) the rewritten query is

```
SELECT mesh,concept,patientid,tupleid
 FROM ill-rr WHERE clusterid=101
```

In the third strategy (clustered partitioning), the rewritten query is

```
SELECT mesh,concept,patientid,tupleid FROM ill-101
```

In the small *ill* table with 56341 entries, 90 related answers are obtained, in the medium-sized *ill* table with 1802912 entries, 2880 related answers are obtained and in the large *ill* table with 14423296 entries, 23040 related answers are obtained. The runtime measurements in Figure 2 in particular show that the lookup table approach does not scale with increasing data set size.

4 Query Answering with Derived Partitions

While the evaluation of a selection query on a single table shows a similar performance for all rewriting strategies, the evaluations of queries on two tables

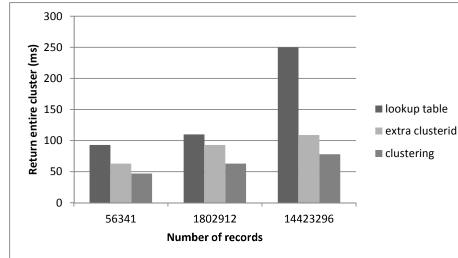

Fig. 2. Return entire cluster as related answers

using a distributed JOIN show a performance impact for the first two strategies when the secondary table is large. We tested a JOIN on the patient ID with a secondary table called *info* having a column *address*. The original query is

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

We devised two test runs: test run one uses a small secondary table (each patient ID occurs only once) and test run two uses a large secondary table (each patient ID occurs 256 times). For the first rewriting strategy (lookup table) the secondary table is a non-partitioned table. For the second strategy, the secondary table is distributed in round robin fashion, too. For the last rewriting strategy, the secondary table is partitioned into a derived partitioning: whenever a patient ID occurs in some partition in the *ill-i* table, then the corresponding tuples in the secondary table are stored in a partition *info-i* on the same server as the primary partition.

In the first strategy (lookup table) the rewritten query is

```
SELECT a.mesh,a.concept,a.patientid,a.tupleid,b.address
 FROM ill AS a,info AS b,lookup WHERE lookup.tupleid=a.tupleid
 AND lookup.clusterid=101 AND b.patientid= a.patientid.
```

In the second strategy (extra clusterid column) the rewritten query is

```
SELECT a.mesh,a.concept,a.patientid,a.tupleid,b.address
 FROM ill-rr AS a,info-rr AS b
 WHERE a.clusterid=101 AND b.patientid=a.patientid.
```

In the third strategy (clustered partitioning), the rewritten query is

```
SELECT a.mesh,a.concept,a.patientid,a.tupleid,b.address
 FROM ill-101 AS a JOIN info-101 AS b
 ON (a.patientid=b.patientid).
```

As Figure 3 shows, a small secondary table does not make much of a difference when executing the join operation (one matching tuple in the secondary table for each tuple in the primary table). However, for the larger secondary table (256

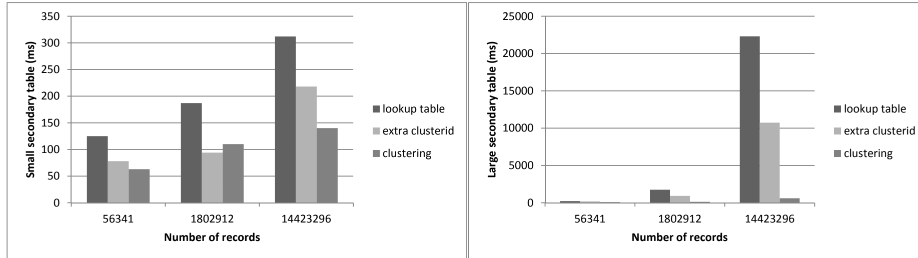

Fig. 3. Join on small and large secondary table

matching tuples in the secondary table for each tuple in the primary table), the impact of the lookup table access is huge in the case of the largest *ill* table.

5 Insertions

We tested the update behavior for all three rewriting strategies by inserting 117 new rows (one for each cluster). Any insertion requires identifying the matching cluster i again (see Section 3.2). Each insertion query looks like this for mesh term m, concept c, patientid 1 and tupleid 1:

`INSERT INTO ill VALUES ('m','c',1,1).`

In the first rewriting strategy, the lookup table has to be updated, too, so that two insertion queries are executed:

`INSERT INTO ill VALUES ('m','c',1,1).`
INSERT INTO lookup VALUES (i,1).

For the second rewriting strategy, the extra clusterid column contains the identified cluster i:

INSERT INTO ill-rr VALUES ('m','c',1,1,i).

For the third rewriting strategy, the matching clustered partition is updated:

INSERT INTO ill-i VALUES ('m','c',1,1).

As shown in Figure 4, we only tested insertions for the largest table. Here the lookup table approach has a huge runtime impact due to the maintenance of the lookup table entries.

6 Deletions

After the insertions we made a similar test by deleting the newly added tuples by issuing the query

`DELETE FROM ill WHERE mesh='m'.`

In the first rewriting strategy, the corresponding row in the lookup table has to

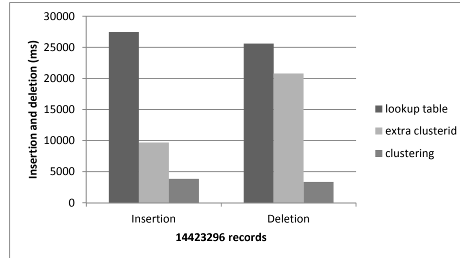

Fig. 4. Inserting one tuple into each cluster

be deleted, too, so that now first the corresponding tuple id of the to-be-deleted row has to be obtained and then two deletion queries are executed:

```
DELETE FROM lookup WHERE lookup.tupleid
 IN (SELECT ill.tupleid FROM ill WHERE mesh='m').
DELETE FROM ill WHERE mesh='m'
```

For the second rewriting strategy, no modification is necessary apart from replacing the table name and no clusterid is needed:

```
DELETE FROM ill-rr WHERE mesh='m'
```

For the third rewriting strategy, the matching clustered partition i is accessed which has to be identified first (as in Section 3.2):

```
DELETE FROM ill-i  WHERE mesh='m'
```

As shown in Figure 4, we only tested deletions for the largest table, where the time needed to identify the matching cluster is negligible. Even the round robin approach with extra clusterid does not perform well on this large data set.

7 Recovery

Lastly, we tested how long it takes to recover the clustered partitioning by either using the lookup table or the extra column ID. The recovery procedure was executed first on the original table and the lookup table by running for each cluster i:

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

for each cluster i. Then, the recovery procedure was executed on the round robin partitioned table with the extra clusterid column for each cluster i:

```
INSERT INTO c_i SELECT * FROM ill-rr WHERE clusterid=i
```

While for the smallest and the largest table the two approaches perform nearly identically, for the medium-sized table the extra cluster id approach offers some benefit.

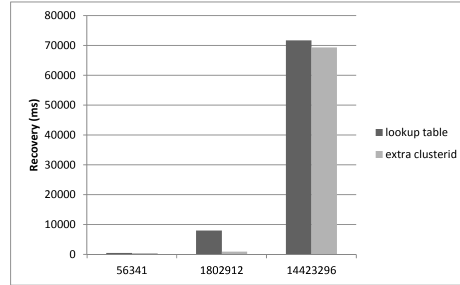

Fig. 5. Recovering the clustered partitioning

8 Related Work

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

8.1 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 [1] used a type abstraction hierarchy to generalize values; Shin et al [18] use some specific notion of metric distance in a knowledge abstraction hierarchy to identify semantically related answers; Halder and Cortesi [5] define a partial order between cooperative answers based on their abstract interpretation framework; Muslea [14] 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 [7] or RDF data in [3]). Another form of query relaxation requires user input in an interactive query relaxation process [9, 10] or analyse query relaxation based on a taxonomy with Bayesian Decision Theory [15].

All these approaches address query relaxation at runtime while answering the query. This is usually prohibitively expensive. In contrast, our approach precomputes the clustering and partitioning so that query answering does not incur a performance penalty. However user interaction might be needed during the clustering process in case some assigments of terms to clusters are ambiguous.

8.2 Data partitioning and Replication

There are some approaches for fine-grained partitioning and replication on object/tuple level; however none of these approaches support the flexible query an-

swering 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. 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. Our results in this paper also show that the fine-grained lookup table approach – when applied to flexible query answering – is inherently slow due to the large amounts of JOIN operations and so it does not scale well even if lookup tables are replicated on all servers.

[2] 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 partitioning that minimizes the number of cut hyperedges. In a second phase, they use a machine learning classifier to derive a range-based partitioning. Then they make an experimental comparison between the graph-based, the range-based, a hash-based partitioning on tuple keys and full replication. Lastly, they also compare three different kinds of lookup tables to map tuple identifier to the corresponding partition: indexes, bit arrays and Bloom filters. Similar to them, we apply lookup tables to locate the replicated data; however we apply this to larger partitions and not to individual tuples.

[16] also model the partitioning 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 [2] 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.

[19] assume three existing partitionings: 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 partitions are obtained, whereas we propose a ontology-driven partitioning approach here.

There is also related work on specifying resource management problems as optimization problems. An adaptive solution for data replication using a genetic algorithm is presented in [11]; 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; they also consider transfer cost of replicas between servers. Load shedding in complex event processing systems is treated in [6]. Virtual machine placement is a very recent topic in cloud computing [17, 4]. However, these specifications do not address the problem of overlapping resources as we need for the flexible query answering approach in this article.

9 Conclusion and future work

We presented an ontology-driven partitioning approach for the application of flexible query answering that finds related answers to a user query. We evaluated its performance on a distributed in-memory store. Due to the small size of the partitioned tables, the runtime performance is best for the clustered partitioning approach and the overhead of metadata management is negligible. It outperforms the lookup table approach that stores for each cluster the corresponding tuple IDs does not scale well as the data set size grows. In addition, the ontology-driven partitioning enables fine-grained load balancing and data locality: less servers have to be accessed when answering queries or updating tables. The idea of data locality can even be carried further by considering cluster affinity: if two clusters are accessed together frequently, their corresponding partitions can be placed on the same server. So far we did not address the dynamic adaptation of the clustering: whenever values are inserted or deleted, the clustering procedure on the entire data set might lead to different clusters. A particular problem that must be handled is the deletion of the head of a cluster: a new cluster head must be chosen before the current head can be deleted; in the simplest case, the term that is most similar to the previous head is chosen as the new head. Similarly, deletions and insertions lead to shrinking or growing partitions. Hence in some situation it might be useful to merge two smaller partitions that are semantically close to each other: we can merge two partitions when their heads are sufficiently similar to each other; or to repartition a larger partition into subpartitions based on a clustering of values of the relaxation attribute in the partition.

9.1 Acknowledgments

The author gratefully acknowledges that the infrastructure and SAP HANA installation for the test runs was provided by the Future SOC Lab of Hasso Plattner Institute (HPI), Potsdam.

References

1. Chu, W.W., Yang, H., Chiang, K., Minock, M., Chow, G., Larson, C.: CoBase: A scalable and extensible cooperative information system. JIIS 6(2/3), 223–259 (1996)
2. Curino, C., Zhang, Y., Jones, E.P.C., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1), 48–57 (2010)
3. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for sparql query relaxation in rdf databases. In: The Semantic Web. Latest Advances and New Domains, pp. 237–252. Springer (2015)
4. Goudarzi, H., Pedram, M.: Energy-efficient virtual machine replication and placement in a cloud computing system. In: IEEE 5th International Conference on Cloud Computing (CLOUD). pp. 750–757. IEEE (2012)

5. Halder, R., Cortesi, A.: Cooperative query answering by abstract interpretation. In: SOFSEM2011. LNCS, vol. 6543, pp. 284–296. Springer, Berlin/Heidelberg, Germany (2011)
6. He, Y., Barman, S., Naughton, J.F.: On load shedding in complex event processing. In: 17th International Conference on Database Theory (ICDT). pp. 213–224 (2014)
7. Hill, J., Torson, J., Guo, B., Chen, Z.: Toward ontology-guided knowledge-driven xml query relaxation. In: Computational Intelligence, Modelling and Simulation (CIMSiM). pp. 448–453 (2010)
8. Inoue, K., Wiese, L.: Generalizing conjunctive queries for informative answers. In: Flexible Query Answering Systems. pp. 1–12. Springer (2011)
9. Jannach, D.: Fast computation of query relaxations for knowledge-based recommenders. AI Communications 22(4), 235–248 (2009)
10. Kumaran, G., Allan, J.: Selective user interaction. In: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. pp. 923–926. ACM (2007)
11. Loukopoulos, T., Ahmad, I.: Static and adaptive distributed data replication using genetic algorithms. Journal of Parallel and Distributed Computing 64(11), 1270–1285 (2004)
12. McInnes, B.T., Pedersen, T., Pakhomov, S.V.S., Liu, Y., Melton-Meaux, G.: Umls::similarity: Measuring the relatedness and similarity of biomedical concepts. In: Vanderwende, L., III, H.D., Kirchhoff, K. (eds.) Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics. pp. 28–31. The Association for Computational Linguistics, Stroudsburg, PA, USA (2013)
13. Michalski, R.S.: A theory and methodology of inductive learning. Artificial Intelligence 20(2), 111–161 (1983)
14. Muslea, I.: Machine learning for online query relaxation. In: Knowledge discovery and data mining (KDD). pp. 246–255. ACM, New York, NY, USA (2004)
15. Pfuhl, M., Alpar, P.: Improving database retrieval on the web through query relaxation. In: Business Information Systems Workshops. pp. 17–27. Springer (2009)
16. Quamar, A., Kumar, K.A., Deshpande, A.: Sword: scalable workload-aware data placement for transactional workloads. In: Guerrini, G., Paton, N.W. (eds.) Joint 2013 EDBT/ICDT Conferences. pp. 430–441. ACM, New York, NY, USA (2013)
17. Shi, W., Hong, B.: Towards profitable virtual machine placement in the data center. In: Fourth IEEE International Conference on Utility and Cloud Computing (UCC). pp. 138–145. IEEE (2011)
18. Shin, M.K., Huh, S.Y., Lee, W.: Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy. Expert Systems with Applications 32(2), 469–484 (2007)
19. Tatarowicz, A., Curino, C., Jones, E.P.C., Madden, S.: Lookup tables: Fine-grained partitioning for distributed databases. In: Kementsietsidis, A., Salles, M.A.V. (eds.) IEEE 28th International Conference on Data Engineering (ICDE 2012). pp. 102–113. IEEE Computer Society, Washington, DC, USA (2012)
20. U.S. National Library of Medicine: Medical subject headings, http://www.nlm.nih.gov/mesh/
21. Wiese, L.: Clustering-based fragmentation and data replication for flexible query answering in distributed databases. Journal of Cloud Computing 3(1), 1–15 (2014)
22. Wiese, L.: Horizontal fragmentation and replication for multiple relaxation attributes. In: 30th British International Conference on Databases, BICOD. Lecture Notes in Computer Science, Springer (2015)