A Replication Scheme for Multiple Fragmentations with Overlapping Fragments

Lena Wiese, Tim Waage and Ferdinand Bollwein

Institute of Computer Science
University of Göttingen
Goldschmidtstraße 7
37077 Göttingen, Germany
Email: {wiese | tim.waage}@cs.uni-goettingen.de
ferdinand.bollwein@stud.uni-goettingen.de

In this article, we introduce a replication procedure in a distributed database system that supports several fragmentations of the same data table. One application that requires multiple fragmentations is flexible (similarity-based) query answering. The major feature of our replication procedure is that replication and recovery respect the overlaps of fragments stemming from different fragmentations. In this paper we extend the data replication problem (DRP) by not only considering hard constraints to ensure a fixed replication factor but also adding soft constraints that express desired data locality of fragments. We furthermore analyze the case that there are more fragmentations (leading to the situation that some replication conditions are optional); and we study the influences of data updates (insertions and deletions) on the data distribution.

Keywords: Bin Packing Problem with Conflicts (BPPC), Data Replication Problem (DRP), Distributed Database, Fragmentation, Integer Linear Programming (ILP),

Received March 2016; revised May 2016

1. INTRODUCTION

When storing large-scale data sets in distributed database systems, these data sets are usually *fragmented* (that is, partitioned) into smaller subsets and these subsets are distributed over several database servers. Moreover, to achieve better availability and failure tolerance, copies of the data sets (the so-called *replicas*) are created and stored in a distributed fashion so that different replicas of the same data set reside on distinct servers. Two major challenges with data fragmentation and replication are to enable efficient query answering while retrieving data from several servers and to handle changes in the data set while maintaining data in a consistent state. Usually only a single optimal fragmentation of a data table is obtained in related approaches. Our approach is aimed at having several fragmentations of the same table and then finding a replication of fragments that takes overlaps of fragments into account and due to this reduces the amount of occupied servers.

In addition to technical requirements of data distribution, *intelligent query answering* mechanisms are increasingly important to find relevant answers to user queries. Flexible (or cooperative) query answering systems help a user 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 to 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 also allow related answers. However, the relaxation procedure is extremely time-consuming and it is hence highly impractical to relax queries at query processing time. In order to offer query relaxation with only a modest overhead it is worthwhile to preprocess the data into semantically coherent clusters based on the notion of similarity in a given taxonomy or ontology. Our approach clusters the data according to several so-called relaxation attributes in the base table. This approach has the advantage that semantically similar values for an individual attribute can be obtained by only retrieving data from a single fragment without the need to consult the ontology and substitute values at runtime – this hence improves the performance of flexible query answering and allows for an improved parallel

processing of queries. However, allowing multiple relaxations leads to several different fragmentations of the same data table. Hence, query relaxation is a good application to study the support for multiple fragmentations.

In this paper we extend the work in [1, 2, 3] by making the following additional contributions:

- we add novel *soft* data locality constraints based on affinity between fragments and formalise them in a data replication problem as an integer linear program;
- we extend the original m-copy replication scheme by allowing more fragmentations ($r > m$) than the replication factor m and making all excess replication constraints optional;
- we devise a heuristic process to handle insertions and deletions of data, as well as merging and splitting of fragments;
- we analyze the runtime performance of the proposed procedure in a 10-node SAP HANA cluster.

1.1. Related Work

We survey related work referring to fragmentation and data distribution as well as to optimization and bin packing. Lastly, we survey the topic of ontology-based flexible query answering which forms the basic use case of our fragmentation and replication procedure.

Fragmentation and Data Replication. There is a long history of *fragmentation* approaches for the relational data model and they are widely covered in standard textbooks like [4]. Most approaches consider workload-aware fragmentations that optimize distribution of data for a given workload of queries. In addition, data locality with respect to data dependencies and common query patterns heavily affects a system's performance. A wide range of cost metrics needs to be taken into account, e.g. the number of processes required, CPU time, job latency, memory utilization, disk and network I/O. Moreover, the problem of data *replication* is a major issue in distributed database systems that are prone to failures. Various algorithms have been proposed for data partitioning and replication in distributed database systems with the goal of minimizing those costs. A good overview of general research challenges can be found in [5, 6].

We survey some fragmentation and replication approaches in more detail.

Several approaches apply vertical (column-wise) fragmentation and consider attribute affinity in a given workload of transactions as an optimization measure. A recent comparative evaluation of vertical fragmentation approaches is provided in [7]; as opposed to these approaches, we aim at an application of horizontal (row-wise) fragmentation for large data sets and we apply the notion of affinity to horizontal fragments.

Some of these approaches consider replication. This is particularly important for in-memory columnar stores like [8].

In a recent article, [9] combine frequent pattern mining and optimization steps for finding optimal vertical fragments. They use the apriori algorithm to identify those attributes that are often accessed together in transactions. This extends the traditional approach of an affinity matrix because several attributes can be considered at the same time. In a next step they extend this approach using a branch-and-bound algorithm that optimizes the combination of attributes in a common vertical fragment further.

In terms of horizontal fragmentation, usually a single optimal fragmentation is identified; in contrast to this our approach tolerates multiple fragmentations in parallel and adapts the replication procedure to the overlaps in the fragmentations. However, all of the following horizontal fragmentation approaches can be combined with our replication approach.

[10] address Multiple Query Optimization (MQO) with a high level of operation sharing between queries. The authors apply a divide and conquer approach to find a horizontal partitioning in a data warehouse that helps trading off speed and optimality of the solution.

In [11] the authors introduce a workload aware horizontal partitioning strategy that analyses query executions to identify and group sets of data items that are accessed together frequently. It is based on a graph structure with nodes representing (sets of) tuples and edges indicating that the connected tuples are often used together within a single transaction. They also consider replication of fragments. As opposed to our approach, the number of fragments is a fixed parameter of the system. That approach was later implemented in practice in the "Relational Cloud" database system [12]. More recently in a competitor system, [13] reduce the amount of inter-partition dependencies and implement an advanced transaction routing.

Horizontal fragmentation can also improve data processing in in-memory stores as shown in [14]; their system features replicated indexes and stored procedure routing.

[15] use an internal data structure for evaluating several partitioning configuration. It is integrated with parallel query optimizers of so called "Massively Parallel Processors" (MPPs). A similar approach is chosen by [16] where data partitioning is based on hashes, ranges or indices in combination with a query optimizer.

As one of the few approaches addressing a combined horizontal and vertical fragmentation, [17] propose a hybrid partitioning method that also considers partitioned indexes as well as materialized views.

Several database systems provide automatic fragmentation like IBMs DB2 Database Advisor [18], Vertica's DBDesigner [19] or Oracle's partitioning by reference [20]. Some fragmentation approaches have been verified using database systems like Teradata [21], H-Store

[14] or Postgres [22].

A major challenge is still to be able to adapt a given fragmentation to changing data sets. One approach that addresses this issue is [23] for the particular case of data sets growing continuously to large sizes. Alternatively, the incremental repartitioning technique of [24] can adapt an existing fragmentation to changes in the data set while supporting workload-aware replication with fine-grained quorums. In a further approach, [22] rely on repartitioning transactions to manage data set modifications. The approach of [25] is based on access counters for data fragments. When reaching a certain threshold the fragment is transferred towards the node that accesses it the most (but not necessarily directly to it).

Providing data replication in conjunction with data fragmentation is necessary in practical distributed systems. [26] propose two heuristics considering redundancy and thus data replication. On the one hand they presented a fast greedy algorithm that performs well when the data is not very dynamic. On the other hand they presented a genetic algorithm based heuristic that delivered a better solution quality for the price of longer runtimes. A greedy algorithm is also the foundation of the work of [27]. Here, data distribution is coordinated using a master node that frequently receives access patterns. It then starts to allocate fragments in decreasing order of their size. Multiple replicas can be considered. However the algorithm requires concrete knowledge of the network topology (including link costs), a knowledge that distributed database systems are usually not aware of. [28] presents a simulation framework for testing several dynamic replication strategies. They also use it to test six strategies and evaluate them in terms of bandwidth consumption and overhead, while limiting their studies to read-only scenarios.

None of these approaches considers multiple fragmentations in parallel; in contrast to these approaches, we take advantage of common subfragments between different fragmentations such that one fragmentation can be recovered from others. Moreover, they disregard semantical similarity of values inside a fragment as is needed for our application of query relaxation.

Optimization and Bin Packing. Bin packing is one of the classical NP-complete problems. As BPP is a special case of the more general BPPC, these properties carry over to BPPC as well. Some variants of classical bin packing have been surveyed in [29]. A more recent survey of treating BPPC as a variant of vertex coloring can be found in [30]. BPPC has been shown to be APX-hard (it is not approximable with a ratio less than 1.5; see [31]). One of the primary sources of BPPC is [32]. A recent branch-and-price approach for BPPC is analyzed in [33].

However, as the number of fragments we consider in our overlap-DRP is comparably low, these complexity theoretic considerations usually do not affect the practical implementation and any off-the-shelf ILP solver will find an optimal solution.

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 [34]; they also consider transfer cost of replicas between servers. Virtual machine placement is a very recent topic in cloud computing [35, 36]. However, these specifications do not address the problem of overlapping resources as we need for the query relaxation approach in this article.

Flexible Query Answering. A database system may not always be able to answer queries in a satisfactory manner. In particular, if a database answer is empty, the corresponding query is said to be a "failing query" (see for example [37]). The reasons for this can be manifold; for instance, in a selection query, selection conditions may be too strict to yield any result. Some authors (for example [38]) differentiate between "misconceptions" and "false presuppositions" as causes for failure. *Cooperative database systems* search for answers that – although not exactly matching the user's original query – are *informative answers* for the user: they provide data that are "closely related" to the user's intention; or they fix misconceptions in a query and return answers to the modified query. Current search engines, web shops or expert systems use similar cooperative techniques to provide users with data that might be close to their interest.

The term "cooperative database system" was for example used in [39] for a system called "CoBase" that relies on several type abstraction hierarchies (TAH) to relax queries and hence to return a wider range of answers. In a similar manner, [40] employ abstraction of domains and define optimality of answers with respect to some user-defined relevancy constraints. The approach using fuzzy sets [37] analyzes cooperative query answering based on semantic proximity. With what they call "incremental relaxation" they apply generalization operators to single conjuncts in a conjunctive query; hence generalized queries form a lattice structure: queries with the same number of applications generalization operators are in the same level of the lattice. Ontology-based query relaxation has also been studied for non-relational data (like XML data in [41] or RDF data in [42]). With these graph structured data it might even be harder for a user to exactly express his query intent. In [43], for example, they introduce the two operators APPROX and RELAX in the query language SPARQL. As opposed to our work, the apply a rule-based cost function that counts the number of rule application – that is they calculate a distance between the original and the relaxed queries. In contrast, with our approach we can apply different similarity measures (where similarity is defined on terms) and flexibly adapt the

cluster sizes by changing the similarity threshold α.

Our approach cannot only be applied to the flexible query answering use case, but to all approaches where multiple partitioning candidates occur. However flexible query answering is an application where several fragmentations can be used to accommodate several user interests at the same time as explained below.

1.2. Organization of the article

Section 2 introduces the main notions used in this article and gives an illustrative example; related work is presented in Section 1.1. Section 3 describes replication for a single fragmentation. Section 4 defines the problem of data replication with overlapping fragments if there are less than m fragmentations (where m is the replication factor). Section 5 treats the more involved case that there are more than m fragmentations. Section 6 adds novel soft constraints to enforce data locality for more performance of query answering. Section 7 analyses the case that fragments are merged or split. Section 8 describes replication and recovery in a practical system. Section 9 concludes this article with suggestions for future work.

2. BACKGROUND AND EXAMPLE

We provide background on fragmentation and data distribution as well as on query generalization as a form of flexible query answering.

2.1. Fragmentation

We will in the following sections present a data replication scheme for horizontal fragmentation where fragments are sets of rows. As a special application for flexible query answering, we will later on define a *semantic* horizontal fragmentation and extend the conventional notion of horizontal fragmentation correctness. Conventional horizontal fragmentation correctness consists of three subproperties: *Completeness* requires that any row of the original data table is contained in one fragments. *Reconstructability* requires that the union of the fragments (that is, rows) yields the original database instance. *Non-redundancy* means that no row is contained in two fragments at the same time.

Definition 2.1 (Horizontal fragmentation correctness). *Let $I = \{t_1, \ldots, t_J\}$ be a data table (a set of tuples), and $F = \{f_1, \ldots, f_n\}$ be a fragmentation of I such that $f_i \subset I$. F is a* correct horizontal fragmentation*, iff:*

1. ***Completeness:*** *for every tuple t_j there is a fragment f_i such that $t_j \in f_i$.*
2. ***Reconstructability:*** $I = f_1 \cup \ldots \cup f_n$
3. ***Non-redundancy:*** *for any i, i' ($i \neq i'$) it holds that $f_i \cap f_{i'} = \emptyset$*

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 database tables shown in Table 1 and 2.

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

TABLE 1. Example ill table

`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

TABLE 2. Example info table

Derived fragmentation can be used to fragment secondary tables according to a primary table and then store the matching (joinable with a primary fragment) tuples of secondary fragments together with the primary fragment to improve data locality during query processing. 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), fragmentations of related tables (like `Info` in our example) can be derived from the primary fragmentation. They are obtained by computing a semijoin between the primary table and the secondary table. Each derived fragment should then be assigned to the same database server on which the primary fragment with the matching join attribute values resides. Note that while the primary fragmentation is usually non-redundant (each tuple of the original table is contained in exactly one fragment), that might not be the case for derived fragmentations: one tuple of a joinable tables might be contained in several derived fragments. In the example, the entire fragmentation on the `Diagnosis` column assigned to two servers then looks as in Figures 1 and 2.

2.2. Data distribution as a Bin Packing Problem

In a distributed database system data records have to be assigned to different servers. The **data distribution problem** – however not considering replication yet – is basically a Bin Packing Problem (BPP) in the following sense:

- K servers correspond to K bins
- bins have a maximum capacity W
- n data records correspond to n objects

Server 1:

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

Info_resp	*PatID*	*Name*	*Address*
	8457	Pete	Main Str 5, Newtown
	2784	Mary	New Str 3, Newtown
	8765	Lisa	Main Str 20, Oldtown

FIGURE 1. Derived fragmentation for respiratory diseases

Server 2:

Fracture	*PatientID*	*Diagnosis*
	2784	brokenLeg
	1055	brokenArm

Info_frac	*PatientID*	*Name*	*Address*
	2784	Mary	New Str 3, Newtown
	1055	Anne	High Str 2, Oldtown

FIGURE 2. Derived fragmentation for fractures diseases

- 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 W

This BPP can be written as an integer linear program (ILP) as follows.

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

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

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

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

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

We use x_{ik} as a binary variable that denotes whether fragment/object f_i is placed in server/bin k: if the variable is 1 this means that fragment f_i is assigned to server k. The variable y_k denotes whether server/bin k is used (that is, is non-empty); in other words, if y_k is 1, some fragment is assigned to it, if it is 0, then no fragment is assigned to server k. Moreover, each server/bin has a maximum capacity W and each fragment/object f_i has a weight w_i that denotes how much capacity the item consumes. As a simple example, W can express how many rows (tuples) a server can store and w_i is the row count of fragment f_i.

This representation can be interpreted as follows: The objective function (1) enforces a minimization of the number of used servers. Constraint (2) requires that each fragment is placed on exactly one server. Furthermore, constraint (3) requires that the capacity or each server is not exceeded. The last two constraints (4) and (5) ensure that the variables are binary.

A simple way to handle derived fragmentations (and hence multiple tables that can be connected by join operations) is to add the weights of the derived fragments (that should be stored on the same server as the primary fragment) to the weight w_i of each primary fragment. In this way when placing the primary fragment on one server, there will also be enough space to accommodate the derived fragments.

An extension of the basic BPP, the Bin Packing with Conflicts (BPPC) problem, considers a conflict graph $G = (V, E)$ where the node set corresponds to the set of objects. A binary edge $e = (i, j)$ exists whenever the two objects i and j must *not* be placed into the same bin. In the ILP representation, a further constraint (Equation 9) is added to avoid conflicts in the placements, because y_k can at most be 1, so that at least one of x_{ik} and x_{jk} must be 0.

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

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

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

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

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

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

Equation (9) effectively prohibits a placement of objects i and j on the same server k because y_k is at most 1 which requires at least one of x_{ik} or x_{jk} to be 0.

Several results were obtained regarding hardness and approximation of bin packing with conflicts. BPPC can basically be regarded as a combination of a vertex coloring and the basic BPP [30, 33].

For our application, we will use the BPPC representation to enforce simple m-copy replication of a fragmented table (for a single fragmentation) as well as an advanced replication scheme if a table is fragmented in r different ways (in multiple fragmentations) and a replication factor of m has to be ensured.

2.3. Query generalization

Query generalization has long been studied in flexible query answering and machine learning (see the seminal article [44]). Query generalization at runtime has been implemented in the CoopQA system [45, 46] by applying three generalization operators to a conjunctive query; while two of them (Dropping Condition and Goal Replacement) are purely syntactic operators, the third called *Anti-Instantiation* (AI) introduces a new variable and might be semantically restricted to avoid overgeneralization; this is what we do in this paper by obtaining fragmentations based on a clustering of the active domain of a relaxation attribute. More precisely, AI 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.

A query formula Q is a conjunction (denoted $\wedge$) of literals (consisting of a predicate and terms) with a set of variables X occurring freely; hence we write a query as $Q(X) = L_{i_1} \wedge \ldots \wedge L_{i_n}$.

As in [47] we apply a notion of generalization based on a model operator $\models$.

Definition 2.2 (Deductive generalization [47]). *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 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 (see [45]).

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: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{Asthma}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{'}\mathit{New\ Str\ 3}, \mathit{Newtown}\text{'})$. Another answer obtained is the fact that Mary suffers from a broken leg as: $\mathit{Ill}(2748, \mathit{Flu}) \wedge \mathit{Ill}(2748, \mathit{brokenLeg}) \wedge \mathit{Info}(2748, \mathit{Mary}, \text{'}\mathit{New\ Str\ 3}, \mathit{Newtown}\text{'})$ which is however an *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 which we will be achieved by the clustering-based fragmentation we propose.

Moreover, query generalization at runtime (as for example implemented in [46]) is highly inefficient. That is why our clustering-based fragmentation preprocesses data into fragments of closely related values (with respect to a relaxation attribute). From an efficiency point of view, this clustering-based fragmentation has two main advantages:

- it enables efficient query relaxation at runtime by returning all values in a matching fragment as relevant answers
- it reduces the amount of servers contacted during query answering in a distributed environment because only one server (containing the matching fragment) has to process the query while other servers can process other queries.

2.4. Relaxation Attributes

In previous work [3], a clustering procedure was applied to partition the original tables into fragments based on a *relaxation attribute* chosen for anti-instantiation. For this we used a notion of similarity between to constants; this similarity can be deduced with the help of an ontology or taxonomy in which the values are put into relation. Finding the fragments is hence achieved by grouping (that is, *clustering*) the values of the respective table column into clusters of closely related values and then splitting the table into fragments

according to the clusters found.

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 fragments of the table `Ill` that could be assigned to two different servers (see Figures 1 and 2).

More formally, we apply a clustering heuristics on those attributes on which anti-instantiation should be applied. We call such an 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 occurring in a given table. For a given table instance I (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 I: $\pi_A(I)$. 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 $\{Cough, Flu, Asthma, brokenArm, brokenLeg\}$. Different relaxation attributes can be specified on a table resulting in clusterings of their active domains that lead to different fragmentations of the same table.

We assume a very general definition of a clustering as being a set of subsets (the *clusters*) of a larger set of values. In general, an arbitrary clustering procedure can be applied as surveyed in [48]. The clustering of the active domain of A induces a horizontal fragmentation of I into fragments $f_i \subseteq I$ 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 $\pi_A(I)$ is the active domain to be clustered, then the complete clustering $C = c_1, \dots, c_n$ covers the whole active domain and no value is lost: $c_1 \cup \dots \cup c_n = \pi_A(I)$. These requirements are summarized in the definition of a *clustering-based fragmentation* as follows.

Definition 2.3 (Clustering-based fragmentation). *Let A be a relaxation attribute; let I be a table instance (a set of tuples); let $C = \{c_1, \dots c_n\}$ be a complete clustering of the active domain $\pi_A(I)$ of A in I; let $head_i \in c_i$; then, a set of fragments $\{f_1, \dots, f_n\}$ (defined over the same attributes as I) is a* clustering-based *fragmentation if*

- *Horizontal fragmentation: for every fragment f_i, $f_i \subseteq I$*
- *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)*
- *Completeness: For every tuple t in I there is an f_i in which t is contained*
- *Reconstructability: $I = f_1 \cup \dots \cup f_n$*
- *Non-redundancy: for any $i \neq j$, $f_i \cap f_j = \emptyset$ (or in other words $c_i \cap c_j = \emptyset$)*

In our implementation, we 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. Based on this similarity specification, we derive a clustering of the active domain of each relaxation attribute A in a relation instance I. These similarity values can for example be calculated by using an ontology or taxonomy; we use a similarity measure (the path measure) to derive similarity values in the UMLS taxonomy in our experimental evaluation below.

We adapted the clustering procedure of [49] that does not require us to fix the number of fragments beforehand. Instead, for efficiency reasons (that is, to reduce the number of similarity calculations) we rely on a representative element called *head* (sometimes also called prototype or centroid) for each cluster. In order for this simplification to work properly, we assume that the similarity between any value inside one cluster and the cluster head should not be larger than a chosen threshold value α. More formally, we require the following additional threshold condition for our clustering: for the $head_i$ elements in the clusters c_i and a threshold value α that restricts the minimal similarity allowed inside a cluster, it holds that $head_i \in c_i$ and for any other value $a \in c_i$ (with $a \neq head_i$) it holds that $sim(a, head_i) \geq \alpha$.

Now, when executing a selection query with a selection condition $A = a$ on a relaxation attribute A, we identify the cluster the head of which is closest to the term a (that is, we identify c_i such that $sim(a, head_i)$ is maximal) and return the matching fragment f_i as the set of related answers. In our example, 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 $Q(x_1, x_2, x_3) = Ill(x_1, Flu) \wedge Ill(x_1, Cough) \wedge Info(x_1, x_2, x_3)$ will hence anti-instantiated to $Q^{AI}(x_1, x_2, x_3, y) = Ill(x_1, Flu) \wedge Ill(x_1, y) \wedge Info(x_1, x_2, x_3)$. Next, it will be rewritten as $Q^{Resp}(x_1, x_2, x_3, y) = Respiratory(x_1, Flu) \wedge Respiratory(x_1, y) \wedge Info(x_1, x_2, x_3)$ and redirected to Server 1 where *only* the fragment 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.

3. REPLICATION FOR A SINGLE FRAGMENTATION

First we consider only a *single* fragmentation (for example, as in [3] obtained for a single relaxation attribute). When doing m-way replication, m copies of the fragments obtained for the single fragmentation are replicated. We consider the following problem:

Definition 3.1 (Data replication problem for m copies (m-copy-DRP)). *Given a fragmentation $F = \{f_1, \dots, f_n\}$ and replication factor m, we obtain m copies $F^1, \dots, F^m$; for each fragment f_i (for $i = 1, \dots, n$), there must be m copies f_i^l (for $1 \leq l \leq m$)*

such that $f_i^1 \in F^1, \ldots, f_i^m \in F^m$ that are all assigned to different servers.

This corresponds to solving a BPPC instance where the conflict graph states that copies of the same fragment cannot be placed on the same server. More formally, for every i and every pair of fragment copies f_i^l and $f_i^{l'}$ there is an edge in the conflict graph.

Definition 3.2 (Conflict graph for m-copy-DRP). *The conflict graph $G^{mDRP} = (V, E)$ is defined by $V = F^1 \cup \ldots \cup F^m$ (one vertex for each fragment inside the m fragmentation copies) and $E = \{(f_i^l, f_i^{l'}) \mid i = 1, \ldots n, l = 1, \ldots m, l' < l\}$ (an undirected edge between two copies of the same fragment).*

The following ILP will find a fragment allocation to servers such that the number of used servers is minimized while respecting the m-copy replication.

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{12}$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik}^l = 1 \qquad i = 1, \ldots, n, \; l = 1, \ldots, m \tag{13}$$

$$\sum_{i=1}^{n} \sum_{l=1}^{m} w_i x_{ik}^l \leq W \cdot y_k \qquad k = 1, \ldots, K \tag{14}$$

$$x_{ik}^l + x_{ik}^{l'} \leq y_k \qquad k = 1, \ldots, K, \; i = 1, \ldots, n, \; l = 1, \ldots, m, \; 0 < l' < l \tag{15}$$

$$y_k \in \{0, 1\} \qquad k = 1, \ldots, K \tag{16}$$

$$x_{ik}^l \in \{0, 1\} \qquad k = 1, \ldots, K, \; i = 1, \ldots, n, \; l = 1, \ldots, m \tag{17}$$

The variable x_{ik}^l represents the placement of the lth copy of fragment f_i on server k. Equation (13) demands that each of the m copies of each fragment is assigned to one server. Equation (14) assigns to each copy of fragment f_i the capacity consumption w_i and ensures that the maximum capacity of each server is not exceeded. Similar to the basic BPPC described previously, Equation (15) ensures that copies of a fragment are placed on different servers because y_k can at most be 1, so that at least one of x_{ik}^l and $x_{ik}^{l'}$ must be 0.

4. OVERLAPS AND MULTIPLE FRAGMENTATIONS

We generalize the replication procedure to multiple fragmentations. This has the following advantages:

- The intelligent replication procedure reduces storage consumption and hence the amount of servers that are needed for replication.
- The system can handle several fragmentations that each cater different user information needs. For example, for the application of flexible query answering, the system can answer queries for several relaxation attributes.

Respiratory	PatientID	Diagnosis
	8457	Cough
	2784	Flu
	2784	Asthma
	8765	Asthma

Fracture	PatientID	Diagnosis
	2784	brokenLeg
	1055	brokenArm

TABLE 3. Fragmentation on Diagnosis attribute

IDlow	PatientID	Diagnosis
	2784	Flu
	2784	brokenLeg
	2784	Asthma
	1055	brokenArm

IDhigh	PatientID	Diagnosis
	8765	Asthma
	8457	Cough

TABLE 4. Fragmentation on PatientID attribute

More formally, we obtain r fragmentations $(F^1, \ldots, F^r)$ of the same table (for example, if r relaxation attributes are chosen and clustered); each fragmentation F^l ($1 \leq l \leq r$) contains fragments $f_1^l, \ldots, f_{n_l}^l$ where index n_l depends on the number of clusters found.

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 as before (see Table 3). And additionally, a different fragmentation on the patient ID can be obtained by dividing into rows with ID smaller than 5000 and those with ID larger than 5000 (see Table 4).

We assume that each of the fragmentations is *complete*: every tuple is assigned to one fragment: for any tuple j, if r relaxation attributes are chosen and clustered, then in any fragmentation F^l (for $1 \leq l \leq r$) there is a fragment f_s^l such that tuple $j \in f_s^l$.

We also assume that each clustering and each fragmentation are *non-redundant*: every value is assigned to exactly one cluster and every tuple belongs to exactly one fragment (for one clustering); in other words, the fragments inside one fragmentation do not overlap. However, fragments from two *different* fragmentations (for two different clusterings) may overlap. For example, both the *Respiratory* as well as the *IDhigh* fragments contain the tuple $\langle 8457, Cough \rangle$.

4.1. Data replication for overlapping fragments

As opposed to m-copy replication, we now analyze an intelligent data replication scheme with *multiple* fragmentations while at the same time minimizing the amount of data copies – and hence reducing overall storage consumption.

While in the standard BPP and BPPC representations usually disjoint fragments and exactly m copies are considered, we extend the basic BPPC as follows: With our intelligent replication procedure, less data copies (only m copies of each tuple) have to be replicated hence reducing the amount of storage needed for replication as opposed to conventional replication approaches that replicate m copies for each of the r fragmentations F^l (which would result in $r \cdot m$ copies of each tuple).

We argue that m copies of a tuple suffice with an advanced recovery procedure: that is, for every *tuple* j we require that it is stored at m *different* servers for backup purposes but these copies of j may be contained in different fragments: one fragmentation F^l can be recovered from fragments in any other fragmentation $F^{l'}$ (where $l \neq l'$). First we assume that there are **exactly** m relaxation attributes (that is, $r = m$). In case there are less than m relaxation attributes (that is, $r < m$), some of the existing fragmentations are duplicated to reach m fragmentations. The more involved case that there are more than m relaxation attributes (that is, $r > m$), is treated below in an upcoming section.

For multiple relaxation attributes, we hence consider the following data replication problem:

Definition 4.1 (Data replication problem with overlapping fragments (overlap-DRP)). *Given m fragmentations $F^l = \{f_1^l, \dots, f_{n_l}^l\}$ and replication factor m, for every tuple j there must be fragments $f_{i_l}^l$ (where $1 \leq l \leq m$ and $1 \leq i_l \leq n_l$) such that $j \in f_{i_1}^1 \cap \dots \cap f_{i_m}^m$ and these fragments are all assigned to different servers.*

We illustrate this with our example. Assume that 5 rows is the maximum capacity W of each server and assume a replication factor 2. In a conventional replication approach, overlaps in the fragments would be ignored. Hence, the conventional approach would replicate all fragments (*Respiratory*, *Fracture*, *IDhigh*, *IDlow*) to two servers each:

- First, assign the *Respiratory* fragment (with 4 rows) to one server $S1$ and a copy of it to another server $S2$.
- Now the *Fracture* fragment (with 2 rows) will not fit on any of the two servers; its two replicas will be stored on two new servers $S3$ and $S4$.
- For storing the *IDlow* fragment (with 4 rows), the conventional approach would need two more servers $S5$ and $S6$.
- The *IDhigh* fragment (with 2 rows) could then be mapped to servers $S3$ and $S4$.

Conventional replication would hence require at least six servers to achieve replication factor 2.

In contrast, our intelligent replication approach takes advantage of the overlapping fragments so that **three** servers suffice to fulfill the replication factor 2; that is, the amount of servers can be substantially reduced if a more intelligent replication and recovery scheme is used that respects the fact that several fragments overlap and that can handle fragments of differing size to optimally fill remaining server capacities. This allows for better self-configuration capacities of the distributed database system. First we observe how one fragment can be recovered from the other fragments:

- Fragment *Respiratory* can be recovered from fragments *IDlow* and *IDhigh* (because *Respiratory*= (*IDlow* ∩ *Respiratory*) ∪ (*IDhigh* ∩ *Respiratory*)).
- Fragment *Fracture* can be recovered from fragment *IDlow* (because *Fracture*= (*IDlow* ∩ *Fracture*)).
- Fragment *IDlow* can be recovered from fragments *Respiratory* and *Fracture* (because *IDlow*= (*IDlow* ∩ *Respiratory*) ∪ (*IDlow* ∩ *Fracture*)).
- Fragment *IDhigh* can be recovered from fragment *Respiratory* (because *IDhigh*= (*IDhigh* ∩ *Respiratory*)).

Hence, we can store fragment *Respiratory* on server $S1$, fragment *IDlow* on server $S2$, and fragments *Fracture* and *IDhigh* on server $S3$ and still have replication factor 2 for individual *tuples*.

We now show that our replication problem (with its extensions to overlapping fragments and counting replication based on tuples) can be expressed as an advanced BPPC problem. Let J be the amount of tuples in the input table, m be the number of fragmentations, K the total number of available servers and n be the overall number of fragments obtained in all fragmentations.

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{18}$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik} = 1 \qquad i = 1, \dots, n \tag{19}$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W \cdot y_k \qquad k = 1, \dots, K \tag{20}$$

$$z_{jk} \geq x_{ik} \qquad \text{for all } j : j \in f_i \tag{21}$$

$$z_{jk} \leq \sum_{(i:j \in f_i)} x_{ik} \qquad k = 1, \dots, K, j = 1, \dots, J \tag{22}$$

$$\sum_{k=1}^{K} z_{jk} \geq m \qquad j = 1, \dots, J \tag{23}$$

$$y_k \in \{0,1\} \qquad k = 1, \dots, K \tag{24}$$

$$x_{ik} \in \{0,1\} \qquad k = 1, \dots, K, i = 1, \dots, n \tag{25}$$

$$z_{jk} \in \{0,1\} \qquad k = 1, \dots, K, j = 1, \dots, J \tag{26}$$

In this ILP representation we keep the variables y_k

for the bins and x_{ik} for fragments – to simplify notation we assume that $i = 1, \ldots, n$ where $n = |F^1| + \ldots + |F^m| = n_1 + \ldots + n_m$: all fragments are numbered consecutively from 1 to n even when they come from different fragmentations. That is, $F^1 = \{f_1, \ldots, f_{n_1}\}$, $F^2 = \{f_{n_1+1}, \ldots, f_{n_1+n_2}\}$, and so on. We introduce K additional variables z_{jk} for each tuple j: $z_{jk} = 1$ if tuple j is placed on server k.

We maintain a mapping between fragments and tuples such that if fragment f_i is assigned to bin k, and tuple j is contained in f_i, then tuple j is also assigned to k (see Equation (21)); the other way round, if there is no fragment f_i containing j and being assigned to bin k, then tuple j neither is assigned to k (see Equation (22)); and we modify the conflict constraint to support the replication factor: we require that for each tuple j the amount of bins/servers used is at least m (see Equation (23)) to ensure the replication factor.

4.2. Reducing the amount of variables

The ILP representation in the previous section is highly inefficient and does not scale to large amounts of tuples: due to the excessive use of z-variables, for large J finding a solution will take prohibitively long. Indeed, in the given representation, we have K y-variables, $n \cdot K$ x-variables, and $J \cdot K$ z-variables where usually $J \gg n$. That is why we want to show now that it is possible to focus on the x-variables to achieve another ILP representation for overlap-DRP: for any tuple j such that j is contained in two fragments f_i and $f_{i'}$ (we assume that $i < i'$ to avoid isomorphic statements in the proof), it is sufficient to ensure that the two fragments are stored on two different servers. We analyze how many conflict conditions are necessary to ensure the replication factor per tuple.

Theorem 4.1. *If there hold $(m \cdot (m-1))/2$ equations of the form $x_{ik} + x_{i'k} = 1$ for any two fragments f_i and $f_{i'}$ such that $f_i \cap f_{i'} \neq \emptyset$ where $i < i'$, $i = 1, \ldots, n-1$, $i' = 2, \ldots, n$ and $k = 1, \ldots, K$, then it holds for any tuple j that $\sum_{k=1}^{K} z_{jk} \geq m$.*

Proof. Due to Equation (19), for every f_i there must be exactly one bin k such that $x_{ik} = 1$; If we make the $(m \cdot (m-1))/2$ pairs of overlapping fragments f_i and $f_{i'}$ mutually exclusive in the ILP representation, m bins are needed to accommodate all these fragments. Due to Equation (21), we assure that when $x_{ik} = 1$ then also $z_{jk} = 1$ for the given tuple j and any f_i such that $j \in f_i$. Hence $\sum_{k=1}^{K} z_{jk} \geq m$ (Equation 23) holds. □

Instead of considering all individual tuples j, we can now move on to considering only overlapping fragments (with non-empty intersections) and requiring the $(m \cdot (m-1))/2$ equations to hold for each pair of overlapping fragments.

We transform the previous ILP representation into the one that enforces a conflict condition for any two overlapping fragments. This coincides with the conventional BPPC representation, where the conflict graph is built over the set of fragments (as the vertex set) by drawing an edge between any two fragments that overlap.

Definition 4.2 (Conflict graph for overlap-DRP). *The conflict graph $G^{DRP} = (V, E)$ is defined by $V = F_1 \cup \ldots \cup F_m$ (one vertex for each fragment inside the m fragmentations) and $E = \{(f_i, f_{i'}) \mid f_i, f_{i'} \in V$ and $f_i \cap f_{i'} \neq \emptyset\}$ (an undirected edge between fragments that overlap).*

Continuing our example, we have a conflict graph over the fragments *Respiratory*, *Fracture*, *IDlow* and *IDhigh* with an edge between *Respiratory* and *IDlow*, and an edge between *Respiratory* and *IDhigh*, and an edge between *Fracture* and *IDhigh*. The ILP representation for overlap-DRP looks now as follows:

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{27}$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik} = 1 \qquad i = 1, \ldots, n \tag{28}$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W \cdot y_k \qquad k = 1, \ldots, K \tag{29}$$

$$x_{ik} + x_{i'k} \leq y_k \qquad k = 1, \ldots, K, f_i \cap f_{i'} \neq \emptyset \tag{30}$$

$$y_k \in \{0, 1\} \qquad k = 1, \ldots, K \tag{31}$$

$$x_{ik} \in \{0, 1\} \qquad k = 1, \ldots, K, i = 1, \ldots, n \tag{32}$$

The objective function (27) still requires that the number of used servers is minimized and we enforce the constraint (28) to assign each fragment f_i to one server k such that the capacity is not exceeded (29). The conflict constraints (30) represent edges of the conflict graph: whenever two fragments overlap, they should not be placed on the same server k such that $x_{ik} = 0$ or $x_{i'k} = 0$; in this way, the sum of x_{ik} and $x_{i'k}$ does not exceed y_k (which is at most 1).

5. OPTIONAL CONFLICTS

We now look at a special case of the replication procedure where the replication factor m is *smaller* than the number r of fragmentations of a table. In this case $(m < r)$, in order to ensure m-way replication for each tuple, only m out of the r fragments must be placed on different servers whereas the other $r - m$ can be placed on the already occupied servers (even when overlapping with some fragments on these servers) – hence reducing the overall amount of needed servers. It is required that for each tuple the replication factor is obeyed. Hence, with r fragmentations, there are r fragments that contain the tuple (still assuming that all fragments in single fragmentation are disjoint). Hence

f_1	tupleID	PatientID	Diagnosis
	1	2784	brokenLeg
	2	2784	Flu
	3	8765	Asthma
	4	8457	Cough

TABLE 5. Fragmentation on tupleID attribute

f'_1	tupleID	PatientID	Diagnosis
	1	2784	brokenLeg
	2	2784	Flu
f'_2	tupleID	PatientID	Diagnosis
	3	8765	Asthma
	4	8457	Cough

TABLE 6. Fragmentation on PatientID attribute

at least m of these fragments must be placed on different servers.

Let us illustrate this case with a small example. Assume we have fragmentation F containing fragment f_1, fragmentation F' with fragments f'_1 and f'_2, as well as fragmentation F'' with fragments f''_1 and f''_2.

Assume that f_1 has an overlap with four other fragments f'_1, f'_2, f''_1 and f''_2: $f_1 \cap f'_1 \neq \emptyset$, $f_1 \cap f'_2 \neq \emptyset$, $f_1 \cap f''_1 \neq \emptyset$ and $f_1 \cap f''_2 \neq \emptyset$; we illustrate this case with a slightly modified example of our medical record (with an additional tupleID attribute) shown in Tables 5, 6 and 7. Hence, for f_1 we have in total four conflict conditions. It is not obvious which of these conditions should be satisfied to achieve a 2-way replication (for each tuple in f_2) while still minimizing the amount of used servers – thus, we elaborate the example a bit further: Assume that the maximum capacity for each server is $W = 6$ the size of f_1 is $w_1 = 4$, whereas the size of f'_1 is $w'_1 = 2$, the size of f'_2 is $w'_2 = 2$, the size of f''_1 is $w''_1 = 1$, and the size of f''_2 is $w''_2 = 3$.

We discuss some options how to distribute these fragments:

- Assume, we put f_1 on one server $S1$. When obeying **all** conflict conditions, we have to put f'_1 and f'_2 on a second server $S2$, and we have to put f''_1 and f''_2 on a third server $S3$. Hence we achieve 3-way replication for all tuples.
- Assume that we only require 2-way replication. Hence we can try to put some overlapping fragments on the same server as long as the 2-way replication is satisfied.
  If we put f_1 and f''_1 on one server $S1$, we can put f'_1 and f''_2 on a second server $S2$. We have to put f'_2 on a third server $S3$. Hence we achieve 2-way replication for all tuples but still need three servers.
- Indeed we can actually reduce the used servers to two while still achieving 2-way replication. We put f_1 and f'_1 on one server $S1$. We put f''_1, f'_2 and f''_2 on a second server $S2$.

f''_1	tupleID	PatientID	Diagnosis
	1	2784	brokenLeg
f''_2	tupleID	PatientID	Diagnosis
	2	8457	Cough
	3	2784	Flu
	4	8765	Asthma

TABLE 7. Fragmentation on Diagnosis attribute

What we see from the example is that it is however impossible to identify which conditions should be satisfied and which are optional only by looking at the individual pair-wise conflicts. The question we answer in the following is how to appropriately express optionality of conflict conditions in our data distribution ILP.

In general, we have r fragmentations of the form $F^l = \{f^l_1, \ldots, f^l_{n_l}\}$ and for every tuple j there must be fragments $f^l_{i_l}$ (where $1 \leq l \leq r$ and $1 \leq i_l \leq n_l$) such that $j \in f^1_{i_1} \cap \ldots \cap f^r_{i_r}$. First of all, we identify *common subfragments* between fragments in the given r fragmentations by computing an r-way intersection. That is we compute $f^1_{i_1} \cap \ldots \cap f^r_{i_r}$.

In our example, we have the following intersections

$$f_1 \cap f'_1 \cap f''_1 = \{t_1\}$$
$$f_1 \cap f'_1 \cap f''_2 = \{t_2\}$$
$$f_1 \cap f'_2 \cap f''_2 = \{t_3, t_4\}$$

Whenever this intersection is non-empty, we obtain pair-wise conflict conditions of the form $x_{i_l k} + x_{i_{l'} k} \leq y_k$ for $1 \leq l \leq r$ and $0 < l' < l$. Continuing the example, for the first intersection (containing tuple t_1), we obtain the pairwise conflicts

$$x_{1k} + x'_{1k} \leq 1$$
$$x_{1k} + x''_{1k} \leq 1$$
$$x'_{1k} + x''_{1k} \leq 1$$

As discussed before, only one of these conditions must be enforced to achieve 2-way replication. We express this "one-out-of-three" condition as follows. We transform the conflict conditions into conditions with new c variables (we need three new variables for each value of k):

$$x_{1k} + x'_{1k} \leq 1 + c_{1k}$$
$$x_{1k} + x''_{1k} \leq 1 + c_{2k}$$
$$x'_{1k} + x''_{1k} \leq 1 + c_{3k}$$

The meaning of the new variables is as follows: if the c-variables are 0, the conflict condition is satisfied (only one fragment is on server k or none of these fragments); if the c-variables are 1, the conflict condition is not satisfied (two fragments are on the same server). Hence we require the c-variables to be binary: $c_{ik} \in \{0, 1\}$. Next, in order to enforce m-way replication, we require that the sum of the c-variables is at most $r - m$ which

effectively means that $r-m$ conditions can be violated at most. In our case $r-m=3-2=1$, we obtain the condition $c_{1k}+c_{2k}+c_{3k}\leq 1$.

Now we generalize these conditions to arbitrary r and m values as follows.

$$\text{minimize} \sum_{k=1}^{K} y_k \tag{33}$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik}^{l} = 1 \qquad i=1,\ldots,n_l,\; l=1,\ldots,r \tag{34}$$

$$\sum_{i=1}^{n}\sum_{l=1}^{r} w_i^l x_{ik}^l \leq W\cdot y_k \qquad k=1,\ldots,K \tag{35}$$

$$x_{i_1k}^{l} + x_{i_2k}^{l'} \leq 1 + c_{ll'k}^{s} \qquad k=1,\ldots,K,\; l=1,\ldots,r,\; 0<l'<l,\; f_{i_1k}^{l}\cap g_s\neq\emptyset,\; f_{i_2k}^{l'}\cap g_s\neq\emptyset,\; s=1,\ldots,S \tag{36}$$

$$\sum_{\substack{l=1\\0<l'<l}}^{r} c_{ll'k}^{s} \leq r-m \qquad k=1,\ldots,K,\; s=1,\ldots,S \tag{37}$$

$$y_k\in\{0,1\} \qquad k=1,\ldots,K \tag{38}$$

$$x_{ik}^{l}\in\{0,1\} \qquad k=1,\ldots,K,\; i=1,\ldots,n_l,\; l=1,\ldots,r \tag{39}$$

$$c_{ll'k}^{s}\in\{0,1\} \qquad k=1,\ldots,K,\; i=1,\ldots,n_l,\; l=1,\ldots,r,\; 0<l'<l \tag{40}$$

Recall that we have r fragmentations of the form $F^l=\{f_1^l,\ldots,f_{n_l}^l\}$ where $l=1,\ldots,r$; we represent the placement of each such fragment on a certain server k by a variable x_{ik}^l. Now, fragments f_i^l all have different weights that are expressed as w_i^l. We denote by g_s the common *non-empty* subfragments between r fragments (as in the example above); we denote by S the overall number of such non-empty subfragments. Jointly Conditions (36) and (37) ensure the optionality of the placement for at most $r-m$ fragments for each common subfragment g_s.

6. DATA LOCALITY

Data locality is a feature to reduce latency of query answering by allocating some data fragments to the **same** server (because they are often accessed together in one query). An established notion for vertical table fragmentation is *attribute affinity* (two attributes are accessed together in the same query; see for example [4]). We transfer this notion to our application and define fragment affinity.

6.1. Affinity of Fragments

As a first step to derive data locality constraints for fragments, we define the notion of affinity of two fragments f_i and f_j. As already mentioned, this affinity notion stems from the definition of affinity between two attributes for vertical fragmentation. There, an affinity measure is derived from a typical workload – that is, a set of queries – and looking at occurrences of attributes in queries. For our definition we have to look at the query terms occurring in the query and identify the fragments in which this term semantically belongs.

Let Q be a query containing a selection condition $A=t$ on a relaxation attribute A for term t; let f_i be the fragment identified to be the fragment matching t according to the given similarity values; in other words, in the fragment f_i, the values in attribute A are the ones of the cluster in which t belongs in terms of maximal similarity to the cluster head (we still assume that there is a unique matching cluster for each term; otherwise we choose one out of the matching clusters at random). Then we can obtain a binary usage value for query Q and fragment f_i.

$$use(Q,f_i) = \begin{cases} 1 & \text{if } sim(t, head(f_i)) \text{ maximal,} \\ 0 & \text{otherwise} \end{cases}$$

We assume a given workload $\mathcal{Q}=\{Q_1,Q_2,Q_3,\ldots,Q_q\}$. We define affinity of two fragments based on their usage pattern in workload $\mathcal{Q}$

DEFINITION 6.1. ***Fragment Affinity.*** *Two fragments f_i and f_j are* affine *if there is a query $Q_k\in\mathcal{Q}$ such that $use(Q_k,f_i)=use(Q_k,f_j)=1$.*

In addition, some queries might be executed more often than others in the workload. We denote $acc(Q_k)$ the access frequency of *query* Q_k in the workload.

We can calculate the access frequency for each *fragment* f_i by summing over the individual usage values giving more weight to queries with higher access frequency:

$$acc(f_i)=\sum_{Q_k\in\mathcal{Q}} use(Q_k,f_i)\cdot acc(Q_k)$$

Based on this access frequency we can exclude some fragments from consideration for data locality: if the access frequency is below a threshold, we do not consider it in the upcoming ILP.

Next we have to derive an affinity measure for two fragments f_i and f_j because data locality constraints will be defined for pairs of fragments. The affinity value of a pair of fragments depends on the usage values and the access frequencies of the queries.

Definition 6.2. ***Affinity Measure.*** *For two fragments f_i and f_j, we define their affinity value in a workload $\mathcal{Q}$ as*

$$aff(f_i, f_j) = \sum_{Q_k \in \mathcal{Q}} use(Q_k, f_i) \cdot use(Q_k, f_j) \cdot acc(Q_k)$$

Due to symmetry of the affinity measure, in the following we always assume – without loss of generality – that $i < j$. Moreover, we only present the case where $r = m$ (as in Section 5); a generalization to the case $r > m$ (as in Section 4) can be done analogously.

6.2. Locality constraints

We add data locality constraints to the integer linear program. These data locality requirements are *soft* constraints in the sense that they should only be satisfied as long as other constraints are not violated. Hence, as opposed to the replication constraints (which are treated as hard constraints which all have to be satisfied), not all of the data locality constraints must be satisfied but we want to satisfy as much of them as possible. We can express this by introducing new variables a_{ijk} and b_{ijk} (for each pair of affine fragments and for each server k) which we require to be binary: $a_{ijk} \in \{0, 1\}$ and $b_{ijk} \in \{0, 1\}$. More precisely, if f_i and f_j are affine fragments (to be put on the same server), we add constraints:

$$x_{ik} - x_{jk} \leq a_{ijk} \qquad (i, j) : aff(f_i, f_j) > 0$$
$$x_{jk} - x_{ik} \leq b_{ijk} \qquad (i, j) : aff(f_i, f_j) > 0$$

The a and b variables can both be 0 whenever the two fragments are both on the **same** server k (because $x_{ik} = x_{jk} = 1$) – or if neither of them is on k (because $x_{ik} = x_{jk} = 0$); otherwise one of them must be at least 1 (because either $x_{ik} = 1$ and $x_{jk} = 0$ or vice versa) leading to a higher penalty in the sum.

Lastly, we modify the objective function to also minimize the sum of a and b values

$$\text{minimize} \sum_{k=1}^{K} y_k + \sum_{k=1}^{K} (a_{ijk} + b_{ijk})$$

This means, we minimize the number of used bins/servers (denoted by y_k) and number of affine fragments f_i, f_j placed on different servers.

More generally, we add such a new summand for every pair of fragments f_i, f_j for which we obtained an affinity value $aff(f_i, f_j)$ larger than 0 – recall that we implicitly assume that $i < j$. This results in an objective function of the form:

$$\text{minimize} \sum_{k=1}^{K} y_k + \sum_{\substack{(i,j):\\ aff(f_i,f_j)>0}} \sum_{k=1}^{K} (a_{ijk} + b_{ijk})$$

However, not all of these summands should be given the same influence: the higher the affinity of two fragments f_i and f_j, the more we want to penalize a violation of their data locality constraint. Hence, we factor in the affinity value $aff(f_i, f_j)$ as a weight α_{ij} for each data locality constraint: $\alpha_{ij} = aff(f_i, f_j)$. A violation of the constraint incurs an extra cost of $2 \cdot \alpha_{ij} = 2 \cdot aff(f_i, f_j)$; this is due to the fact that – in case of a violation – there are two different servers k and k' such that $a_{ijk} = 1$ and $b_{ijk'} = 1$. This results in an objective function of the form:

$$\text{min.} \sum_{k=1}^{K} y_k + \sum_{\substack{(i,j):\\ \alpha_{ij}>0}} \alpha_{ij} \cdot \sum_{k=1}^{K} (a_{ijk} + b_{ijk})$$

Lastly, we might want to give the minimization of the number of used servers more weight than the data locality constraint. Hence we need a weight γ for the y_k variables that exceeds the affinity values; by defining γ to be higher than twice the sum of all affinity values, we achieve exactly that: we prefer to reduce the number of used servers at the cost of violating data locality constraints. More formally,

$$\gamma = 1 + 2 \cdot \sum_{i,j} aff(f_i, f_j)$$

Our final objective function is hence of the form:

$$\text{min. } \gamma \cdot \sum_{k=1}^{K} y_k + \sum_{\substack{(i,j):\\ \alpha_{ij}>0}} \alpha_{ij} \cdot \sum_{k=1}^{K} (a_{ijk} + b_{ijk})$$

The entire ILP with data locality constraints is the following.

$$\text{min. } \gamma \cdot \sum_{k=1}^{K} y_k + \sum_{\substack{(i,j):\\ \alpha_{ij}>0}} \alpha_{ij} \cdot \sum_{k=1}^{K} (a_{ijk} + b_{ijk}) \qquad (41)$$

$$\text{s.t.} \sum_{k=1}^{K} x_{ik} = 1 \qquad i = 1, \ldots, n \quad (42)$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W \cdot y_k \qquad k = 1, \ldots, K \quad (43)$$

$$x_{ik} + x_{i'k} \leq y_k \qquad k = 1, \ldots, K, f_i \cap f_{i'} \neq \emptyset \quad (44)$$

$$x_{ik} - x_{jk} \leq a_{ijk} \qquad (i, j) : aff(f_i, f_j) > 0 \quad (45)$$

$$x_{jk} - x_{ik} \leq b_{ijk} \qquad (i, j) : aff(f_i, f_j) > 0 \quad (46)$$

$$y_k \in \{0, 1\} \qquad k = 1, \ldots, K \quad (47)$$

$$x_{ik} \in \{0, 1\} \qquad k = 1, \ldots, K, i = 1, \ldots, n \quad (48)$$

$$a_{ijk} \in \{0, 1\} \qquad (i, j) : aff(f_i, f_j) > 0 \quad (49)$$

$$b_{ijk} \in \{0, 1\} \qquad (i, j) : aff(f_i, f_j) > 0 \quad (50)$$

Note that we need both Condition (45) and Condition (46) as we cannot express absolte values in an ILP.

7. HEAD REELECTION

The ontology-driven query answering process relies on comparisons with the head values of all fragments based

on [49]. That is why the deletion of a tuple containing a head value from a fragment or the modification of the underlying ontology raises the need to find a new head element for an existing fragment. Moreover, if one fragment grows too large (resulting in an overwhelming amount of related answers), it must be split into subfragments each with a new head element. The other way round, too small fragments might result into too few related answers; hence, *similar* fragments can be merged and a new head element has to be elected. We discuss heuristics for these three cases in the following subsections.

7.1. New head for existing cluster

We can simply choose an element in the existing cluster that is most similar to the old head. That is, for cluster c_i we compute all elements closest to $head_i$:

$$\{a \mid a \in c_i, sim(a, head_i) \text{ is maximal}\}$$

When there is more than one such element, in order to maximize similarity to all other elements in the cluster, we can choose the one for which the sum of similarities to all other elements is largest:

$$\{a \mid a \in c_i, \sum_{b \in c_i} sim(a, b) \text{ is maximal}\}$$

7.2. Cluster Splitting

Following the idea of [49], splitting a large cluster c_i into subclusters requires identifying the elements *farthest* away from the old head $head_i$ and defining them as the new heads for the subclusters. That is we compute the new heads as the set:

$$H = \{head_i^j \mid head_i^j \in c_i, sim(head_i^j, head_i) \text{ is minimal}\}$$

Next, we define the subclusters $c_i^1, c_i^2, \ldots, c_i^{|H|}$ by assigning to the those elements in c_i closest to the new head $head_i^j$:

$$c_i^j = \{head_i^j\} \cup \{a \mid a \in c_i; sim(a, head_i^j) \leq sim(c, head_i^{j'}); j \neq j'\}$$

7.3. Cluster Merging

Assume we a set C of small clusters. We group them together whenever the similarity of their heads is below a threshold α. That is, heuristically, we choose one $c_i \in C$ and compute the merged cluster c_{new} as

$$c_{new} = c_i \cup \bigcup c_j$$

for c_j such that $sim(headj, head_i) \leq \alpha$. We define the set of previous heads as

$$H' = \{head_i\} \cup \bigcup \{head_j\}$$

For finding a new head $head_{new}$ several heuristic options arise:

- in the simplest case, keep $head_i$ as $head_{new}$
- from $head_i$ and the heads $head_j$ from the merged clusters, choose as $head_{new}$ the one that has maximal similarity to the others:

$$head_{new} \in \{a \mid a \in H', \sum_{head_k \in H'} sim(a, head_k) \text{ is maximal}\}$$

- choose an arbitrary element from c_{new} that is most similar to the previous heads:

$$head_{new} \in \{a \mid a \in c_{new}, \sum_{head_k \in H'} sim(a, head_k) \text{ is maximal}\}$$

8. EXPERIMENTAL STUDY IN A LARGER CLUSTER

Our prototype implementation – the OntQA-Replica system – runs on a distributed SAP HANA installation with 10 database server nodes provided by the Future SOC Lab of Hasso Plattner Institute. This is an extension to previous work that only used a 3-node cluster [1, 2]; as a result we were able to extend the experiments to larger data sets. All runtime measurements 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 [50]). 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). We increased that table size by duplicating the original data up to 12 times, resulting in 230,772,736 rows.

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 fragmentation (the clustered fragmentation) was obtained from this clustering and consists of 117 fragments which are each stored in a smaller table called *ill-i* where i is the cluster ID. To allow for a comparison, another fragmentation 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 fragments have an extra column called *clusterid*.

In order to manage the fragmentation, 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 fragment.
- 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*) There are different metrics for calculating this similarity value. An overview is given in [3]. We used the path length scheme.

8.1. Identifying the matching cluster

All query rewriting strategies require the identification of the matching cluster previously. 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 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 *similarities* table has 6,591,897 rows (56341 rows of the basic data set times 117 cluster heads). The runtime measurements for this query show a decent performance of at most 24 ms. Note that the size of the *similarities* table is constant for all test runs, since the data set duplication does not create any new mesh terms.

8.2. 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:

- 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 fragmentation: the last rewriting approach replaces the occurrences of the *ill* table by the corresponding *ill-i* table for clusterid i.

8.3. Query Answering without Derived Fragments

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 fragmentation), the rewritten query is

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

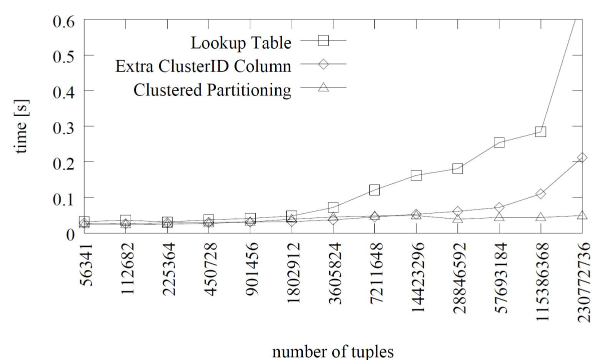

FIGURE 3. Time for executing queries without derived partitioning

The runtime measurements in Figure 3 in particular show that the lookup table approach does not scale with increasing data set size. The extra cluster-id column performs better, but does not scale either, when the data set becomes very large. The approach using clustered partitioning outperforms both by having nearly identical runtimes for all sizes of the test data set. Note, that after duplicating the data set 12 times it is 4096 times as large as the basic data set.

8.4. Query Answering with Derived Fragments

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
```

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

```
SELECT a.mesh,a.concept,a.patientid,
 a.tupleid,b.address,lookup.clusterid
```

```
  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 AS a,info AS b
  WHERE a.clusterid=101
  AND b.patientid=a.patientid.
```

In the third strategy (clustered fragmentation), 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).
```

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-fragmented table. For the second strategy, the secondary table is distributed in round robin fashion, too. For the last rewriting strategy, the secondary table is fragmented into a derived fragmentation: whenever a patient ID occurs in some fragment in the *ill-i* table, then the corresponding tuples in the secondary table are stored in a fragment *info-i* on the same server as the primary fragment.

Figure 4 presents the runtime measurements for queries with derived fragments with the small secondary table (one matching tuple in the secondary table for each tuple in the primary table). It can be observed that the necessary join operation causes all three approaches to perform significantly worse. The clustered partitioning strategy still shows the best performance with being roughly twice as fast as the other ones. While the lookup table approach performed worst without derived fragments, it performed better than the extra cluster-id column strategy when tested with derived fragments using small secondary tables.

However, as can be seen in Figure 5 both approaches are clearly outperformed by the clustered partitioning strategy when the secondary table is large (256 matching tuples in the secondary table for each tuple in the primary table). It delivers feasible performance up to 6-7 data set duplications, while the lookup table and extra cluster-id column approaches fail in doing so after only 2-3 data set duplications.

8.5. Insertions and Deletions

We tested the update behavior for all three rewriting strategies by inserting 117 new rows (one for each cluster). After the insertions we made a similar test

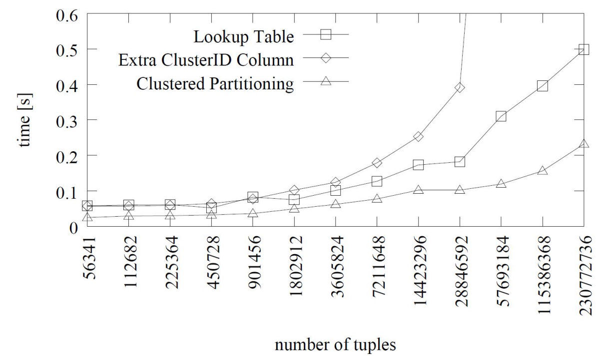

FIGURE 4. Time for executing queries with derived partitioning (small secondary tables)

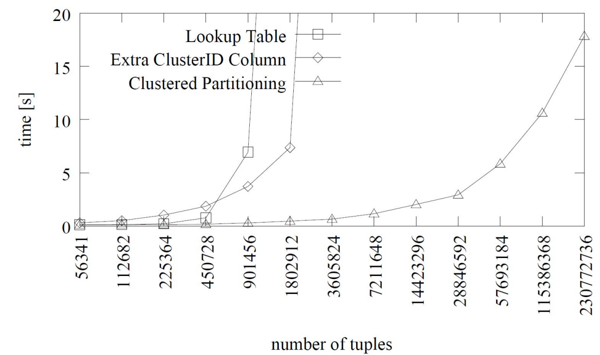

FIGURE 5. Time for executing queries with derived partitioning (large secondary tables)

by deleting the newly added tuples.

Any insertion requires identifying the matching cluster i as described previously. Then 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 fragment is updated:

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

As shown in Figure 6, the runtime for insertions appears to be constant for all approaches. Interestingly only the round robin approach performs worse by factor 2.5; this might be due to the fact that it takes longer to identify the fragment into which the insertion has to be written.

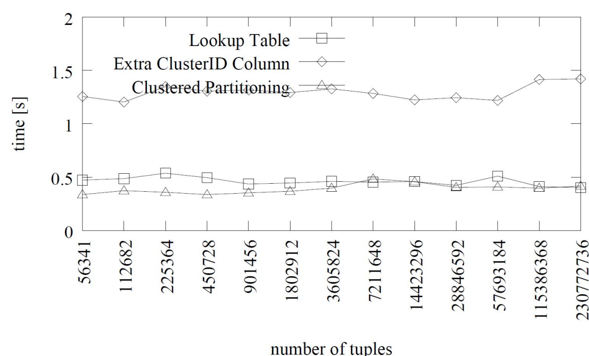

FIGURE 6. Insertion

Deletions require queries of the basic form

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

In the first rewriting strategy, the corresponding row in the lookup table has to 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 fragment i is accessed which has to be identified first:

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

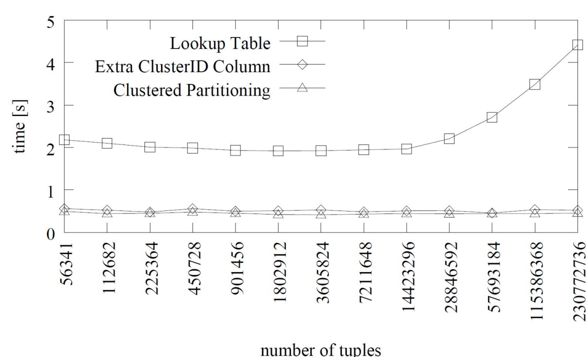

FIGURE 7. Deletion

Figure 7 presents the measurements for deletions. Here the runtimes for the extra cluster-id column and clustered partitioning approach is constant and on a similar level, while the lookup table strategy performs roughly 4 times worse due to its higher complexity. Starting from a certain data set size the deletion time of this approach even begins to grow significantly further.

8.6. Recovery

The recovery procedure recovers the clustered fragmentation. In particular, we show that one fragmentation (the clustered fragmentation) can be recovered from another one (the round-robin fragmentation). For the lookup table approach this is done using the following query on the original table and the lookup table by running for each cluster i:

```
INSERT INTO c_i SELECT mesh, concept,
 patientid, ill.tupleid FROM ill
 JOIN lookup
 ON (lookup.tupleid=ill.tupleid)
 WHERE lookup.clusterid=i
```

For the round robin fragmented table with the extra clusterid column the query for each cluster i is as follows:

```
INSERT INTO c_i
 SELECT mesh, concept, patientid, tupleid
 FROM ill-rr
 WHERE clusterid=i
```

In both cases this results in one c_i table per cluster.

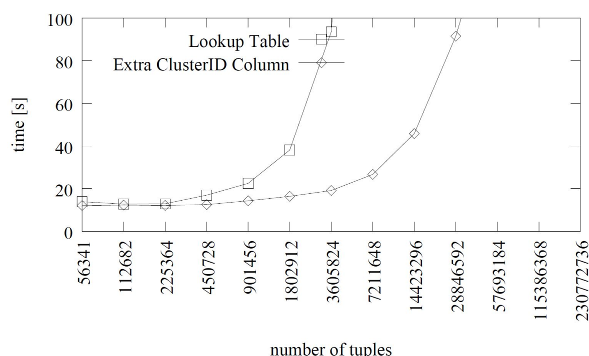

FIGURE 8. Recovery

As can be seen in Figure 8 both recovery procedures become unfeasible very quickly with the approach for the extra cluster-id column strategy being able to handle 2-3 data set duplications more in an acceptable timeframe.

8.7. Inserting a new head term

When inserting a new head term into the data set similarities to all other existing terms have to be computed and written in the *similarities* table. In addition the *root* table has to be updated. That means for our example data set 56341 similarity values must be calculated and inserted into the *similarities* table. Note that this number is constant for all data set sizes, since duplicating the data set does not create new terms. In our tests this took 250 seconds, mainly due to the similarity value computing.

8.8. Two Relaxation Attributes

Lastly we tested queries with two relaxation attributes. Note that requires finding the matching cluster twice.

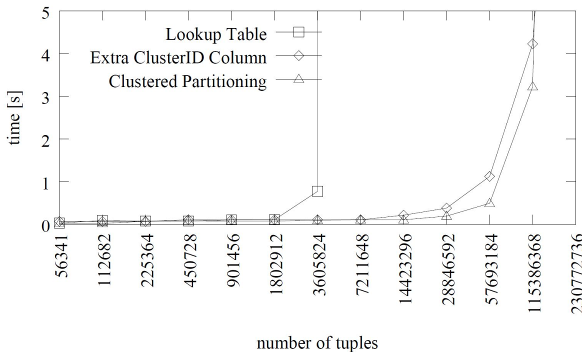

FIGURE 9. Time required for queries with 2 relaxation attributes

Figure 9 presets the results. It can be observed that queries with two relaxation attributes can be done in an acceptable runtime up to a certain data set size. After that point the runtime increases dramatically. While the lookup table approach starts to perform worse after 6 data set duplications, the extra cluster-id column strategy and the clustered partitioning strategy are feasible up to 10 data set duplications. This degradations might be due to the needed intermediate joins for which the system runs out of memory after a certain data set size.

9. CONCLUSION AND FUTURE WORK

We presented and analyzed a data replication problem for a flexible query answering system. It provides related answers by relaxing the original query and obtaining a set of semantically close answers. The proposed replication scheme allows for fast response times due to materializing the fragmentations. By solving an ILP representation of the data replication problem, we minimize the overall number of servers used for replication. In this paper the focus lies on supporting multiple relaxation attributes that lead to multiple fragmentations of the same table. A minimization of the number of servers is due to the fact that one fragmentation can be recovered from other fragmentations based on overlapping fragments. The experimental evaluation shows sensible performance results.

Future work has to mainly address dynamic changes in the replication scheme. Deletions and insertions of data lead to changing fragmentations sizes and hence an adaptation of the server allocations might become necessary (similar to [34]). The use of adaptive methods will be studied where (a large part of) a previous solution might be reused to obtain a new solution. As further fields of study, partial index maintenance for the fragments as well as the application of the approach to other data formats (like XML or RDF data [41, 42, 43]).

9.1. Acknowledgements

The authors gratefully acknowledge that the infrastructure and SAP HANA installation for the test runs was provided by the Future SOC Lab of Hasso Plattner Institute (HPI), Potsdam. Tim Waage has been funded by the German Research Foundation under grant number WI 4086/2-1.

REFERENCES

[1] Wiese, L. (2015) Horizontal fragmentation and replication for multiple relaxation attributes. *Data Science (30th British International Conference on Databases)*, pp. 157–169. Springer.

[2] Wiese, L. (2015) Ontology-driven data partitioning and recovery for flexible query answering. *Database and Expert Systems Applications*, pp. 177–191. Springer.

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

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

[5] Ke, Q., Prabhakaran, V., Xie, Y., Yu, Y., Wu, J., and Yang, J. (2011) Optimizing data partitioning for data-parallel computing. *13th Workshop on Hot Topics in Operating Systems, HotOS XIII*, pp. 13–13. USENIX Association.

[6] Stonebraker, M., Pavlo, A., Taft, R., and Brodie, M. L. (2014) Enterprise database applications and the cloud: A difficult road ahead. *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 1–6. IEEE.

[7] Jindal, A., Palatinus, E., Pavlov, V., and Dittrich, J. (2013) A comparison of knives for bread slicing. *Proceedings of the VLDB Endowment*, **6**, 361–372.

[8] Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., and Madden, S. (2010) Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, **4**, 105–116.

[9] Huang, Y.-F. and Lai, C.-J. (2016) Integrating frequent pattern clustering and branch-and-bound approaches for data partitioning. *Information Sciences*, **328**, 288–301.

[10] Bellatreche, L. and Kerkad, A. (2015) Query interaction based approach for horizontal data partitioning. *International Journal of Data Warehousing and Mining (IJDWM)*, **11**, 44–61.

[11] Curino, C., Zhang, Y., Jones, E. P. C., and Madden, S. (2010) Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, **3**, 48–57.

[12] Curino, C., Jones, E. P., Popa, R. A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., and Zeldovich, N. (2011) Relational cloud: A database-as-a-service for the cloud. *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 235–240. www.cidrdb.org.

[13] Turcu, A., Palmieri, R., Ravindran, B., and Hirve, S. (2016) Automated data partitioning for highly scalable and strongly consistent transactions. *IEEE Transactions on Parallel and Distributed Systems*, **27**, 106–118.

[14] Pavlo, A., Curino, C., and Zdonik, S. (2012) Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 61–72. ACM.

[15] Nehme, R. and Bruno, N. (2011) Automated partitioning design in parallel database systems. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1137–1148. ACM.

[16] Zhou, J., Bruno, N., and Lin, W. (2012) Advanced partitioning techniques for massively distributed computation. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 13–24. ACM.

[17] Agrawal, S., Narasayya, V., and Yang, B. (2004) Integrating vertical and horizontal partitioning into automated physical database design. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 359–370. ACM.

[18] Zilio, D. C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S. (2004) DB2 design advisor: integrated automatic physical database design. *Proceedings of the Thirtieth international conference on Very large data bases (Volume 30)*, pp. 1087–1097. VLDB Endowment.

[19] Varadarajan, R., Bharathan, V., Cary, A., Dave, J., and Bodagala, S. (2014) Dbdesigner: A customizable physical design tool for vertica analytic database. *30th International Conference on Data Engineering (ICDE)*, pp. 1084–1095. IEEE.

[20] Eadon, G., Chong, E. I., Shankar, S., Raghavan, A., Srinivasan, J., and Das, S. (2008) Supporting table partitioning by reference in oracle. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1111–1122. ACM.

[21] Bellatreche, L., Benkrid, S., Ghazal, A., Crolotte, A., and Cuzzocrea, A. (2011) Verification of partitioning and allocation techniques on teradata dbms. *Algorithms and Architectures for Parallel Processing*, pp. 158–169. Springer.

[22] Chen, K., Zhou, Y., and Cao, Y. (2015) Online data partitioning in distributed database systems. *18th International Conference on Extending Database Technology (EDBT)*, pp. 1–12. OpenProceedings.org.

[23] Liroz-Gistau, M., Akbarinia, R., Pacitti, E., Porto, F., and Valduriez, P. (2013) Dynamic workload-based partitioning algorithms for continuously growing databases. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XII*, pp. 105–128. Springer.

[24] Quamar, A., Kumar, K. A., and Deshpande, A. (2013) Sword: scalable workload-aware data placement for transactional workloads. In Guerrini, G. and Paton, N. W. (eds.), *Joint 2013 EDBT/ICDT Conferences*, New York, NY, USA, pp. 430–441. ACM.

[25] Gope, D. C. (2012) Dynamic data allocation methods in distributed database system. *American Academic & Scholarly Research Journal*, **4**, 1.

[26] Loukopoulos, T. and Ahmad, I. (2000) Static and adaptive data replication algorithms for fast information access in large distributed systems. *20th International Conference on Distributed Computing Systems*, pp. 385–392. IEEE.

[27] Kamali, S., Ghodsnia, P., and Daudjee, K. (2011) Dynamic data allocation with replication in distributed systems. *IEEE 30th International on Performance Computing and Communications Conference (IPCCC)*, pp. 1–8. IEEE.

[28] Ranganathan, K. and Foster, I. (2001) Identifying dynamic replication strategies for a high-performance data grid. *Grid Computing (GRID)*, pp. 75–86. Springer.

[29] Coffman Jr, E. G., Csirik, J., and Leung, J. Y.-T. (2007). Variants of classical one-dimensional bin packing.

[30] Malaguti, E. and Toth, P. (2010) A survey on vertex coloring problems. *International Transactions in Operational Research*, **17**, 1–34.

[31] Epstein, L. and Levin, A. (2006) On bin packing with conflicts. *SIAM Journal on Optimization*, **19**, 1270–1298.

[32] Jansen, K. and Öhring, S. (1997) Approximation algorithms for time constrained scheduling. *Information and Computation*, **132**, 85–108.

[33] Sadykov, R. and Vanderbeck, F. (2013) Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, **25**, 244–255.

[34] Loukopoulos, T. and Ahmad, I. (2004) Static and adaptive distributed data replication using genetic algorithms. *Journal of Parallel and Distributed Computing*, **64**, 1270–1285.

[35] Shi, W. and Hong, B. (2011) Towards profitable virtual machine placement in the data center. *Fourth IEEE International Conference on Utility and Cloud Computing (UCC)*, pp. 138–145. IEEE.

[36] Goudarzi, H. and Pedram, M. (2012) Energy-efficient virtual machine replication and placement in a cloud computing system. *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pp. 750–757. IEEE.

[37] Pivert, O., Jaudoin, H., Brando, C., and HadjAli, A. (2010) A method based on query caching and predicate substitution for the treatment of failing database queries. *ICCBR 2010*, LNCS, **6176**, pp. 436–450. Springer.

[38] Godfrey, P. (1997) Minimization in cooperative response to failing database queries. *IJCS*, **6**, 95–149.

[39] Chu, W. W., Yang, H., Chiang, K., Minock, M., Chow, G., and Larson, C. (1996) CoBase: A scalable and extensible cooperative information system. *JIIS*, **6**, 223–259.

[40] Halder, R. and Cortesi, A. (2011) Cooperative query answering by abstract interpretation. *SOFSEM2011*, LNCS, **6543**, pp. 284–296. Springer.

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

[42] Fokou, G., Jean, S., Hadjali, A., and Baron, M. (2015) Cooperative techniques for SPARQL query relaxation

in RDFs databases. *The Semantic Web. Latest Advances and New Domains*, pp. 237–252. Springer.

[43] Selmer, P., Poulovassilis, A., and Wood, P. T. (2015) Implementing flexible operators for regular path queries. *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference*, pp. 149–156. CEUR Workshop Proceedings.

[44] Michalski, R. S. (1983) A theory and methodology of inductive learning. *Artificial Intelligence*, **20**, 111–161.

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

[46] Bakhtyar, M., Dang, N., Inoue, K., and Wiese, L. (2014) Implementing inductive concept learning for cooperative query answering. *Data Analysis, Machine Learning and Knowledge Discovery*, pp. 127–134. Springer.

[47] Gaasterland, T., Godfrey, P., and Minker, J. (1992) Relaxation as a platform for cooperative answering. *JIIS*, **1**, 293–321.

[48] Berkhin, P. (2006) A survey of clustering data mining techniques. *Grouping multidimensional data*, pp. 25–71. Springer.

[49] Gonzalez, T. F. (1985) Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, **38**, 293–306.

[50] U.S. National Library of Medicine. Medical subject headings. http://www.nlm.nih.gov/mesh/.