

Hierarchical Property Set Merging for SPARQL Query Optimization

Marios Meimaris
Athena Research Center
m.meimaris@athenarc.gr

George Papastefanatos
Athena Research Center
gpapas@athenarc.gr

Panos Vassiliadis
University of Ioannina
pvassil@cs.uoi.gr

ABSTRACT

Characteristic sets (CS) organize RDF triples based on the set of properties associated with their subject nodes. This concept was recently used in indexing techniques, as it can capture the implicit schema of RDF data. While most CS-based approaches yield significant improvements in space and query performance, they fail to perform well when answering complex query workloads in the presence of schema heterogeneity, i.e., when the number of CSs becomes very large, resulting in a highly partitioned data organization. In this paper, we address this problem by introducing a novel technique, for merging CSs based on their hierarchical structure. Our method employs a lattice to capture the hierarchical relationships between CSs, identifies dense CSs and merges dense CSs with their ancestors, thus reducing the size of the CSs as well as the links between them. We implemented our algorithm on top of a relational backbone, where each merged CS is stored in a relational table, and we performed an extensive experimental study to evaluate the performance and impact of merging to the storage and querying of RDF datasets, indicating significant improvements.

1 INTRODUCTION

The Resource Description Framework¹ (RDF) and SPARQL Protocol and RDF Query Language² (SPARQL) are W3C recommendations for representing and querying data on the semantic web. The semantic web has been established as a source of diverse datasets from a multitude of domains, such as life sciences, statistics, finance, open science and health. Wide adoption has led to increasingly larger dataset sizes, and at the same time complex analytical queries have started to emerge, reflecting the ever increasing recognition of value of analytical processing in the modern data-centric economy. In light of this, RDF data management methods are calling for improvements in the performance of RDF storage and querying engines, as has been discussed in recent works, where state-of-the-art RDF engines are found to be very efficient in simple workloads, but not efficient enough when it comes to more complex query workloads [17][16][9][12].

In response to this limitation, recent works have shown that extraction and exploitation of the implicit schema of the data can be beneficial in both storage and SPARQL query performance [9][12]. In order to organize on disk, index and query triples efficiently, these efforts heavily rely on two structural components of an RDF dataset, namely (i) the notion of *characteristic sets* (CS), i.e., different property sets that characterize subject nodes, and (ii) the join links between CSs. Formally, given an RDF dataset D ,

and a subject node s , the characteristic set $cs(s)$ of s is defined as follows [13]:

$$cs(s) = \{p \mid \exists o : (s, p, o) \in D\}$$

In plain words, a CS is the set of attributes (or RDF properties) of a given subject node. Abstracting from RDF triples to their CSs, an RDF graph can be represented on the structural level by a graph containing CSs as nodes, and links between CSs as edges, where a link between two CSs exists whenever a link between two subject nodes exists in the original RDF graph. Due to the ability to represent all of the properties of a subject node with a single set, rather than multiple triples, CSs have been thoroughly used as a means to optimize query planning, storage, indexing and query processing [5, 9, 12, 13, 16]. In their most general form, they are used as the basis for mapping RDF to a relational structure, where each CS forms a relational table. An illustration of this mapping can be seen in Figure 1. There are two entities, *Alice* and *Claire*, each represented in an RDF graph as a node. The properties of these two nodes are (a) their type, (b) the company for which they work and (c) their supervisor. The set of these tree properties, $\{rdf:type, worksFor, supervises\}$, forms the characteristic set for these two nodes. The direct representation as a relation is depicted at the bottom of Fig. 1, with the characteristic set becoming the attributes of the relation.

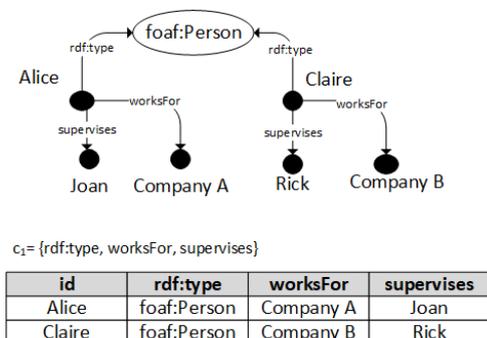


Figure 1: A simple RDF graph consisting of a single characteristic set, c_1 , and its resulting table.

However, a mapping from RDF to the relational space is not always helpful, as the structural looseness that is allowed in the RDF world can translate to multiple, heterogeneous CSs that represent skewed distributions of triples. For example, instead of the homogeneity of the graph in Figure 1, where all of the nodes share the structure, i.e., the same CS, consider the case of Figure 2(a) where nodes are described by four different CSs. In fact, it is frequent, that there exist many different CSs within the same dataset, representing varying internal structure for the nodes of the source graph. This schema heterogeneity in loosely-structured datasets is indeed frequently found in the real world (e.g., Geonames contains 851 CSs and 12136 CS links),

¹<https://www.w3.org/RDF/>

²<https://www.w3.org/TR/sparql11-overview/>

Table 1: RDF datasets along with their number of CSs and links between CSs.

Dataset	# Tables (CSs)	# of CS joins
Reactome	112	346
Geonames	851	12136
LUBM 2000	14	68
WatDiv 100	5667	802
Wordnet	779	7250
EFO	520	2515
DBLP	95	733

imposing large overheads in the extraction, storage and disk-based retrieval[17][9]. For reference, some well established RDF datasets along with their associated numbers of CSs (first column) are shown in Table 1.

In these cases, we end up with a lot of CSs, each of which may actually represent very few nodes. There are two antagonizing approaches in creating a relational schema from a set of CSs. (i) Creating a relational table for each different CS would result in a large numbers of relational tables with few tuples in each of them, as shown in Figure 2(b) that require a very large number of joins to answer queries; (ii) on the other hand, creating a single, "universal" table, accommodating the CS's of all the nodes of the graph would create a very wide relation, as in Figure 2(c), which is practically overly empty (i.e., too many NULL values) and space inefficient, due to the fact that different tuples would have very different cells populated.

There are two factors that affect or constrain the design choices we have. First, one could wonder "why not splitting the universal relation in the traditional manner?". Unfortunately, the traditional decomposition via functional dependencies is neither available, or enforceable to the source data. It is very possible that the incoming data to be stored are not necessarily accompanied by metadata descriptions that prescribe functional dependencies. Even if a DBA would design them, it is also possible that the data violate them. Thus, in the context of this paper, we proceed with an automated design that assumes that no functional dependencies are available and all schema dependencies are online detected by the incoming data. Second, the desired analysis and the expected workload we consider in this paper play a significant role in the determination of the appropriate structure. Most SPARQL queries involve complex graph patterns, i.e., query conditions, which must be evaluated on the different CSs[9–11]; applying such patterns to a universal relation will result in multiple self-joins whereas applying patterns to a fine-grained relational schema, would again impose multiple joins (and multiple unions of the intermediate results for merging the query output) across a high number of CS relational tables. Consider the evaluation of the following SPARQL query on the nodes of Figure 2(a),

```
SELECT ?x ?y ?z ?w
WHERE {
  ?x worksFor ?y.
  ?x supervises ?z.
  ?z hasBirthday '2011-02-24'.
  ?z isMarriedTo ?w.
  ?w hasNationality 'GR' }
```

Assuming a design with a "universal" relation (Figure 2c), the query evaluation over the relational schema (see Section 4 for the details on the SPARQL-to-SQL query evaluation) would require one self-join for each one of the *worksFor*, *supervises* and

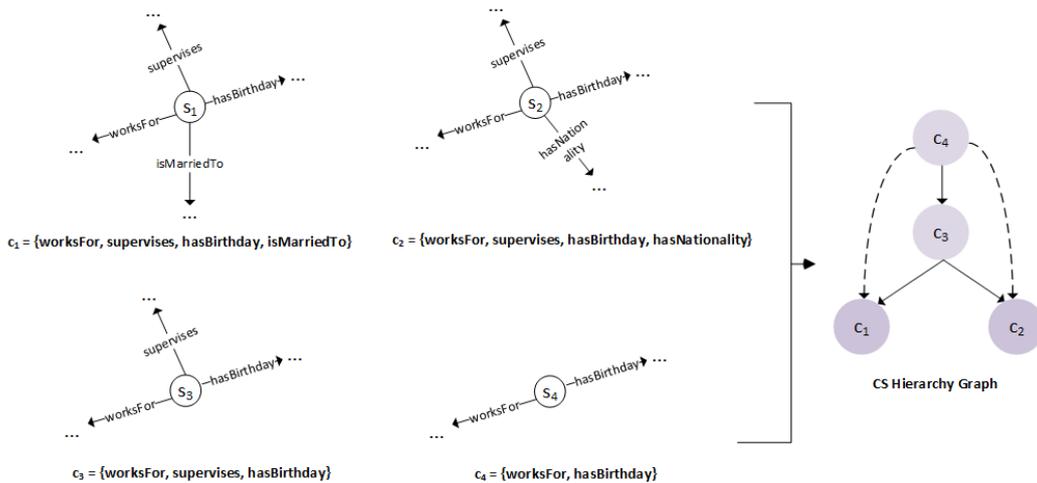
isMarriedTo query conditions (*hasBirthday* and *hasNationality* conditions are executed as select operations in SQL). At the same time, there is a price of NULL values, too, as (Figure 2c) shows. At the other end of the spectrum, in the multiple CS tables case (Figure 2b), each one of the three query conditions would require one self join and additionally three joins between each CS table and all other CS tables in the database – i.e., 4 joins per table. Thus, despite its space efficiency, the latter case imposes performance overheads due to the large number of joins the query must perform to fetch data from multiple tables. This number can become significantly large in real-world datasets, as shown in the second column of Table 1.

In this paper, we tackle the problem of mapping heterogeneous RDF datasets to a relational schema with the aim to facilitate the processing of complex analytical SPARQL queries, by automating the decision of which tables will be created in order to host the incoming data, such that there are no overly empty tables and extremely large numbers of joins.

Interestingly, the problem is both open and hard to solve. The current approaches to address the problem do not avoid the problems that we have mentioned before and include approaches to store data generically as triples, as property tables (practically one table per CS) or as attribute value pairs as vertical partitions (see Section 2 for a detailed discussion). All these solutions are found in the extremes of the dilemma between empty space and query performance without achieving a "sweet" compromise between the two forces. Therefore, the state of practice and the state of the art provide room for improvement, and in fact, to the best of our knowledge, this is the first effort to address the problem. At the same time, the problem is hard to solve: as we show in Section 4, the complexity is exponential, and therefore, brute force methods are not adequate.

In our approach, we introduce an algorithm to automate the design process. We start by considering that each CS is a single table and by exploiting their hierarchical relationships, we merge related CSs into the same table. Moreover, we define a density-based cost function that help us stop the merging process for CSs that contain a large numbers of triples. In this way, we achieve merging of CS based on the structural similarity as well as the number of triples they contain. We follow a relational implementation approach by storing all triples corresponding to a set of merged CSs into a separate relational table and by executing queries through a SPARQL-to-SQL transformation. Although alternative storage technologies can be considered (key-value, graph stores,etc), we have selected well-established technologies and database systems for the implementation of our approach, in order to take advantage of existing relational data indexing and query processing techniques that have been proven to scale efficiently in large datasets and complex workloads. To this end, we present a novel system, named *raxonDB*, that exploits these hierarchies in order to merge together hierarchically related CSs and decrease the number of CSs and the links between them, resulting in a more compact schema with better data distribution. *raxonDB*, built on top of PostgreSQL, provides significant performance improvements in both storage and query performance of RDF data with respect to the state of the art techniques presented in Section 2. In short, our contributions are as follows:

- We introduce a novel CS merging algorithm that takes advantage of CS hierarchies,
- we implement *raxonDB*, an RDF engine built on top of a relational backbone that takes advantage of this merging for both storing and query processing,



(a) Four subject nodes, s_1, s_2, s_3, s_4 with their properties and their CSs (left) and the resulting CS hierarchy graph (right).

id	supervises	worksFor	hasBirthday	isMarriedTo
s_1

id	supervises	worksFor	hasBirthday	hasNationality
s_2

id	supervises	worksFor	hasBirthday
s_3

id	worksFor	hasBirthday
s_4

(b) Edge case where each CS becomes a relational table. No NULL values exist in any of the tables.

id	supervises	worksFor	hasBirthday	isMarriedTo	hasNationality
s_1	NULL
s_2	NULL	...
s_3	NULL	NULL
s_4	NULL	NULL	NULL
...

(c) Edge case where all CSs become one universal table. NULL values exist in this table.

id	supervises	worksFor	hasBirthday	isMarriedTo
s_1
s_3	NULL
s_4	NULL	NULL
...

id	supervises	worksFor	hasBirthday	hasNationality
s_2
...

(d) Merging c_1, c_3, c_4 together and leaving c_2 unmerged.

Figure 2: Example of four CSs coming from different source datasets, and their respective CS hierarchy graph. Examples of the two edge cases (all tables vs one table), as well as the merging case can be seen. In the figure, the CSs can be seen as derived from data instances (nodes s_1, s_2, s_3, s_4). It can generally be assumed that there are more instances belonging to these CSs, not shown in the figure.

- we perform an experimental evaluation that indicates significant performance improvements for various parameter configurations.

Roadmap. In Section 2, we present the background and related work for this paper. In Section 3, we provide preliminary definition and delineate the problem, and in Section 4 we present algorithms towards its solution. In Section 5, we discuss the experimental evaluation of the paper. We conclude the paper in Section 6, with a summary of our findings and future work.

2 RELATED WORK

Due to the tabular structure that tends to implicitly underlay RDF data, recent works for RDF data management systems have been implemented in relational backbones. They generally follow three storage schemes, namely (a) triples tables, (b) property tables, and, (c) vertical partitioning. A *triples table* has three columns, representing the subject, predicate and object (SPO) of an RDF triple. This technique replicates data in different orderings in order to facilitate sort-merge joins. RDF-3X [14] and Hexastore [21] build tables on all six permutations of SPO. Built on a relational backbone, Virtuoso [4] uses a 4-column table for quads, and a combination of full and partial indexes. These methods work well for queries with small numbers of joins, however, they degrade with increasing sizes, unbound variables and joins.

Property Tables places data in tables with columns corresponding to properties of the dataset, where each table identifies a specific resource type. Each row identifies a subject node and holds the value of each property. This technique has been implemented experimentally in Jena [22] and DB2RDF [3], and shows promising results when resource types and their properties are well-defined. However, this causes extra space overhead for null values in cases of sparse properties [1]. Also, it raises performance issues when handling complex queries with many joins, as the amounts of intermediate results increase [8].

Vertical partitioning segments data in two-column tables. Each table corresponds to a property, and each row to a subject node [1]. This provides great performance for queries with bound objects, but suffers when the table sizes have large variations in size [20]. TripleBit [23] broadly falls under vertical partitioning. In TripleBit, the data is vertically partitioned in chunks per predicate. While this reduces replication, it suffers from similar problems as property tables. It does not consider the inherent schema of the triples in order to speed up the evaluation of complex query patterns.

In distributed settings, a growing body of literature exists, with systems such as Sempala [18], H2RDF [15] and S2RDF [19]. However, these systems are based on the parallelization of centralized indexing and query evaluation schemes.

Due to the high heterogeneity in the schema during the integration and analysis of multiple RDF datasets, latest state of the art approaches rely on implicit schema detection in order to index/store triples based on their schema. In our previous work [9], we defined *Extended Characteristic Sets (ECSs)* as typed links between CSs, and we showed how ECSs can be used to index triples and greatly improve query performance. In [17], the authors identify and merge CSs, similar to our approach, into what they call an *emergent schema*. However, their main focus is to extract a human-readable schema with appropriate relation labelling and they do not use hierarchical information of CSs, rather they use semantics to drive the merging process. In [16] it is shown how this *emergent schema* approach can assist query

performance, however, the approach is limited by the constraints of human-readable schema discovery. In our work, query performance, indexing and storage optimization are the main aims of the merging process, and thus we are not concerned about providing human-readable schema information or any form of schema exploration. In [12], the authors use CSs and ECSs in order to assist cost estimation for federated queries, while in [5], the authors use CSs in order to provide better triple reordering plans. To the best of our knowledge, this is the first work to exploit hierarchical CS relations in order to merge CSs and improve query performance.

3 PRELIMINARIES

RDF triples consist of a subject s , a predicate p and an object o . An RDF dataset is represented as a directed labelled graph where subjects and objects are nodes, and predicates are labelled edges. Formally, let I, B, L be infinite, pairwise disjoint sets of URIs, blank nodes and literals, respectively. Then, $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$ is a triple. RDF does not enforce structural rules in the representation of triples; within the same dataset there can be largely diverse sets of predicates emitting from nodes of the same type [9, 13, 17]. *Characteristic Sets (CS)*[13] capture this diversity by representing implied node types based on the set of properties they emit. Formally, given a dataset D , and a subject node s , the characteristic set $cs(s)$ of s is $cs(s) = \{p \mid \exists o : (s, p, o) \in D\}$, and the set of all CSs is denoted with C . In what follows, we present basic definitions for CSs and their more generalized form of *Property Sets*. Intuitively, a property set is a set of RDF predicates $p_1 \dots p_n$. Therefore, a CS is also a property set. As we are interested in creating property sets by merging CSs, we use this more general form (i.e., property set instead of CS) in order to denote sets of attributes that are not necessarily CSs in the original dataset, but are used after merging several CSs as the basis for a relational table.

Although each real world measurement can have its very own characteristic set, for all practical purposes, the information stored in RDF triples typically refers to recurring concepts and their features. Thus, some co-occurrences of features are common, e.g., to triples coming from the same source and representing "records" pertaining to the same "concept" (e.g., triples for events, pertaining to the concept *Event* come with the same set of features like datetime and description. All event instances from the same source are likely (albeit not obliged) to carry overlapping subsets of these features. At the same time, *despite the commonalities, there are differences too, typically produced by the heterogeneity of data sources and the lack of structural restrictions in RDF data.* For example, an RDF dataset containing information about people, is likely to associate a *name* property for each instance of a *Person* class, while a *isFatherOf* property would not be associated with all instances for obvious reasons. Thus, there is a possibility to define subset relationships between the features of similar concepts. To this end, we introduce the notion of subsumption and the notion of hierarchy of subsumed property sets.

Definition 1. (Property Sets and Property Tables). In its more general form, a CS is a property set P_i , i.e., a set of RDF predicates and the set of all property sets is denoted with P . A *Property Table* (also: *CS Table*) T_i for a given property set P_i is a relational table comprising an identifier *sid* for the subject node s and $|P_i|$ columns, where $P_i = \{p_{i,1}, p_{i,2}, \dots, p_{i,n}\}$ are the predicates emitting from s . T_i contains the set r_i of tuples that instantiate the values of the properties for *sid*, i.e., $T_i = (sid \cup P_i, r_i)$. A tuple

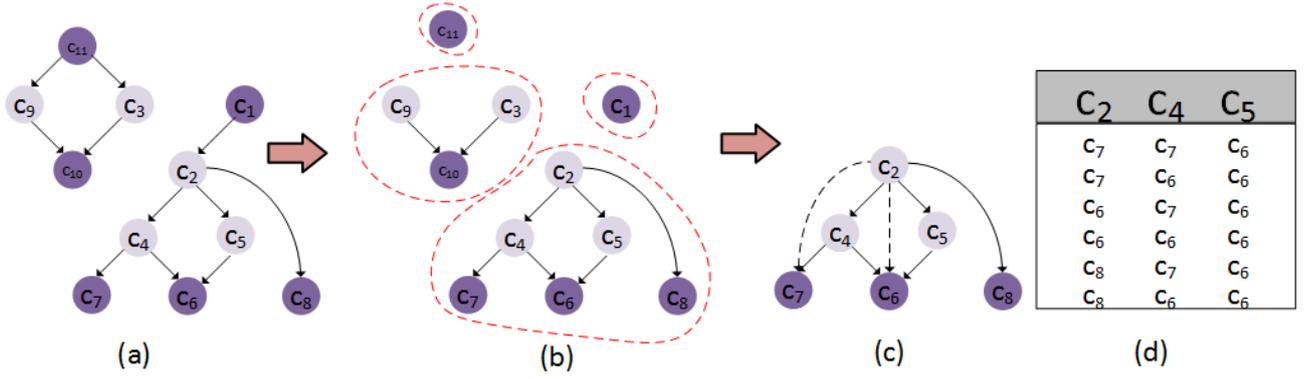


Figure 3: (a) A CS hierarchy graph with dense nodes colored in deep purple, (b) the connected components derived by cutting off descendants from dense nodes, (c) a connected component with dashed lines representing inferred hierarchical relationships, (d) all possible assignments of dense nodes to non-dense nodes.

can hold values for the predicate columns in $I \cup B \cup L \cup N U L L$, i.e., each cell in the relational table can either have a standard value of an RDF triple object, or $N U L L$ when the subject node identified in sid does not have a property in any of its triples. In Figure 2(a), four subject nodes, s_1, s_2, s_3, s_4 , are shown. These have four different CSs based on their property sets, namely c_1, c_2, c_3, c_4 .

Definition 2. (CS Subsumption). Given two CSs, c_i and c_j , then c_i subsumes c_j , or $c_i > c_j$, when c_i is a subset of c_j , or $c_i \subset c_j$. This subsumption forms a parent-child relationship. For example, consider c_1, c_2 that describe human beings, with $c_1 = \{type, name\}$ and $c_2 = \{type, name, marriedTo\}$. It can be seen that $c_1 \subset c_2$ and therefore c_1 subsumes c_2 . CS subsumption relationships can be seen in Figure 3(a) as directed edges between nodes. In the example of Figure 2(a), the four CSs exhibit strong subset-superset relationships between their properties. For instance, c_1 and c_2 have property sets that are supersets of both c_3 and c_4 , while c_3 also subsumes c_4 . The set of all parent-child relationships defines a CS hierarchy.

Definition 3. (CS Hierarchy and Inferred Hierarchy). CS subsumption creates a partial hierarchical ordering such that when $c_i > c_j$, then c_i is a parent of c_j . Formally, a CS hierarchy is a graph lattice $H = (V, E)$ where $V \in C$ and $E \in (V \times V)$. A directed edge between two CS nodes c_1, c_2 exists in H , when $c_1 > c_2$ and there exists no other c_i such that $c_1 > c_i > c_2$. The directed edge stems from the parent node and arrives at the child node. An example CS hierarchy can be seen in Figure 3(a). Given a hierarchy H , we denote the *hierarchical closure* of H with H_c , so that H_c extends H to contain *inferred* edges between hierarchically related nodes that are not consecutive, e.g. a node and its grandchildren. Intuitively, for every set of features c that describes some concept in the dataset, we introduce a node in our graph. Every strict superset of features of c is a descendant of c , and every strict subset of features of c is a superset of c . An example inferred hierarchy can be seen on the right of Figure 2(a), with the inferred relationships in dashed lines as well as in Figure 3(c) for a sub-graph of the graph in Figure 3(a). In the remainder of this paper, we refer to H_c as the *inferred hierarchy* of H . The lattice resembles the traditional OLAP cube lattice (see [7]), although in our case, the construction is ad-hoc, depending on the available CS's, and serves a different purpose.

Definition 4. (CS Ancestral Sub-graphs). Given an inferred hierarchy $H_c = (V, E)$, a CS c_{base} , a set of CSs c_1, \dots, c_k , and a sub-graph $H_{c_{base}}^{anc} = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$, we

say that $H_{c_{base}}^{anc}$ is an *ancestral sub-graph* over c_{base} when $\forall i \in [1..k]$, it holds that $c_i > c_{base}$ and $(c_i, c_{base}) \in E'$. Intuitively, any set of ancestors of a node c_{base} forms an ancestral sub-graph. More than one ancestral sub-graphs can be defined over c_{base} , as any subset of its parents is considered an ancestral sub-graph over c_{base} . For instance, in Figure 3(c), nodes c_7, c_4, c_2 form an ancestral sub-graph over c_7 . Similarly, nodes c_6, c_4, c_2 and c_6, c_5, c_2 form ancestral sub-graphs over c_6 .

Having defined the hierarchy of characteristic sets and the respective graph, we are now ready to provide the foundation for the core of the problem of this paper. Basically, the goal is to find a way to store data in a way that balances two antagonizing goals: (a) the number of $N U L L$ values and the unnecessary space increase, vs., (b) the resulting decrease in query processing time that would result from the fragmentation of stored data in multiple nodes and the need to join them. To address this issue, we can attempt to *merge* different nodes into one, paying the price of empty space to avoid the price of joins. There exist two edge cases here, namely (i) assign a table to each CS in the incoming data, resulting in as many tables as CSs, and (ii) assign one universal tables for all CSs. This table would contain all of the properties found in the input dataset. The two edge cases for the running example of Fig. 2 can be seen in Fig. 2(b,c).

Definition 5. (Hierarchical CS Merge). Given an ancestral sub-graph $a = (V', E')$, where $V' = \{c_1, c_2, \dots, c_k\}$ as defined above, and the set of property tables $T(V')$, then $hier_merge(a, T(V'))$ is a hierarchical merge of a that results in a single table $T_a = (c_{base}, r_a)$. As c_{base} is the most specialized CS in a , the columns of T_a are exactly the properties in c_{base} , while $r_a = \bigcup_{i=1}^k r'_i$ is the

union of the records of all CSs in V' , where r'_i is the projection of r_i on the properties in c_{base} . Consequently, r'_i contains $N U L L$ values for all the non-shared properties of c_{base} and c_i . In essence, this is an *edge contraction* operator that merges all tables of the nodes of an ancestral sub-graph into one and keeps the signature (i.e., the properties) of the base CS c_{base} . For instance, assume that $V' = \{c_0 = (P_0, r_0), c_1 = (P_1, r_1), c_2 = (P_2, r_2)\}$ is the set of vertices of an ancestral sub-graph with three CSs, with $P_0 = \{p_a, p_b\}$, $P_1 = \{p_a, p_b, p_c\}$ and $P_2 = \{p_a, p_b, p_c, p_d\}$. Thus, $c_0 > c_1 > c_2$. The output of the merging process for our running example can be seen in Figure 2(d). Hierarchical merging can be seen in Figure 4.

Definition 5. (Merge Graph). Given an inferred CS hierarchy $H_c = (V, E)$, a merge graph is a graph $H' = (V', E')$ that consists of a set of n ancestral sub-graphs, and has the following properties: (i) H' contains all nodes in H such that $V' \equiv V$, i.e., it covers all CSs in the input dataset, (ii) H' contains a subset of the edges in H such that $E' \subset E$, (iii) each node is contained in exactly one ancestral sub-graph a_i , (iv) all ancestral sub-graphs are pair-wise disconnected, i.e., there exist no edges between the nodes of different ancestral sub-graphs. Thus, each ancestral sub-graph can be contracted into one node unambiguously, using the *hier_merge* operator. Also, the total number of relational tables will be equal to the number of ancestral sub-graphs in the merge graph.

The primary focus of this work is to improve the efficiency of storage and querying in relational RDF engines by exploiting the implicit schema of the data in the form of CSs. The two extreme approaches – i.e., (a) a separate table for each CS, on the one end of the spectrum, or, (b) merging all the data into a universal table with a schema expressed as the union of the individual features of the different CS's, at the other end of the spectrum, have the following drawbacks, respectively: (i) multiple joins of separate tables at query processing, if many tables are used, and (ii) bad utilization of disk space if nodes are excessively merged into a very small number of tables (due to excessive empty space with NULLs).

Problem Formulation. Given an inferred CS hierarchy $H_c = (V, E)$, the problem is to find a merge graph $H' = (V, E')$ in the form of a set of disconnected ancestral sub-graphs, that provides a good way to merge CS nodes. In other words, the problem is to find the best set of ancestral sub-graphs from an inferred hierarchy H_c that minimize an objective cost function $cost(x)$.

4 HIERARCHICAL CS MERGING

What makes the problem hard, is the complexity of finding a sweet spot in the Pareto equilibrium between conciseness of the relational schema (few tables) and internal density (with respect to the empty space they contain).

Schema conciseness. To start with the arguments in favor of reducing the number of nodes via merging, we can see that, by reducing the number of CS, the number of joins between them is also reduced. Furthermore, merging together CSs leads to a less skewed distribution of data to relational tables. Ultimately, this results in a drastically decreased disk-based I/O cost, as less tables are fetched, positively affecting query processing as well.

Density. On the contrary, merging tables results in the introduction of NULL values for the non-shared columns, which can degrade performance. Specifically, merging CSs with different attribute sets can result in large numbers of NULL values in the resulting table. Given a parent CS table $T_1 = (sid \cup c_1, r_1)$ and a child CS table $T_2 = (sid \cup c_2, r_2)$ with $|c_1| < |c_2|$ and $|r_1| \gg |r_2|$, the resulting $|c_2 \setminus c_1| \times |r_1|$ NULL cells will be significantly large compared to the total number of $r_1 + r_2$ records, thus potentially causing poor storage and querying performance[17]. For this reason, CS merging must be performed in a way that will minimize the presence of NULL values. The following function captures the NULL-value effect of the merge of two CS tables $T_i = (sid \cup c_i, r_i)$, $T_j = (sid \cup c_j, r_j)$ with $c_i > c_j$:

$$r_{null}(T_i, T_j) = \frac{|c_j \setminus c_i| \times |r_i|}{(|r_j| + |r_i|)} \quad (1)$$

r_{null} represents the ratio of null values to the cardinality of the merged table. The numerator of the fraction represents the total number of cell values that will be null, as the product of the number of non-shared properties and the cardinality of the parent CS. The denominator represents the resulting cardinality of the table.

In order to assess an ancestral sub-graph, we use a generalized version of r_{null} that captures the NULL value effect on the whole sub-graph:

$$r_{null}^g(g)|_{T_d} = \frac{\sum_{i=1}^{|g|} |c_d \setminus c_i| \times |r_i|}{|r_d| + \sum_{i=1}^{|g|} (|r_i|)} \quad (2)$$

Here, $T_d = (c_d, r_d)$ is the root of sub-graph g . However, merging a parent to a child changes the structure of the input graph, as the cardinality of the merged child is increased. Thus, we define a cost function that works on the graph level, as follows:

$$cost(g) = \sum_{i=1}^n r_{null}^g(g_i)|_{c_{d_i}} \quad (3)$$

where n is the number of dense nodes, c_{d_i} is a dense node and g_i is the ancestral sub-graph with c_{d_i} as the base node.

Thus, choosing dense CSs as bases is a seeding process that aims to minimize this NULL value effect by making sure that a large fraction of the input records will not be extended with NULL values. This is true because a CS base and its resulting merged table will have exactly the same properties (columns) and thus introduction of NULL values will not be needed for the records of the CS base.

The problem of selecting ancestral sub-graphs for the merge is computationally hard, as mentioned earlier. For this reason, we rely on heuristics in order to seed the process and provide an initial set of ancestral sub-graph *bases* for the final merged tables. The CS bases will be the only relational tables in the output, with the remaining tables merged into them. For this, we identify *dense* CS nodes in the hierarchy (i.e, with large cardinalities) and use these nodes as the bases of the ancestral sub-graphs. While node density can be defined in many different ways, in the context of this work we define a c_i to be dense, if the cardinality of its relational table is larger than a linear function of the maximum cardinality of CSs in D , i.e., a function $d : N \rightarrow R$, with $d(T_i) = m \times |r_{max}|$. Here, $m \in [0, 1]$ is called the *density factor*, and r_{max} is the cardinality of the largest CS table in D . By definition, if $m = 0$, no CSs will be merged (i.e., each CS will be stored in each own table), while if $m = 1$, no tables will be created, as no CS has a cardinality larger than that of the largest CS. With a given m , the problem is reduced to finding the optimal ancestral sub-graph for each given dense node.

Given this cost model and a predefined set of dense nodes, our algorithm will find the optimal sub-graph for each dense node. An inferred hierarchy graph can be converted to a set of connected components that are derived by removing the outgoing edges from dense nodes, since we are not interested in merging children to parents, but only parents to children. An example of this can be seen in Figure 3(b). For each component, we can compute $cost(g)$ as the sum of the costs of these components. The main idea is to identify all connected components in the CS graph, iterate through them, enumerate all sub-graphs within the components that start from the given set of dense nodes, and select the optimal partitioning for each component.

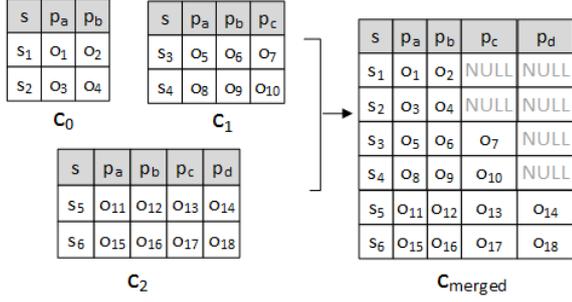


Figure 4: Merging the tables of c_0 , c_1 and c_2 .

The algorithm can be seen in Algorithm 1. The algorithm works by first identifying all connected components of the inferred hierarchy (Line 2). Identifying connected components is trivially done using standard DFS traversal, and is not shown in the Algorithm. Then, we iterate each component (Line 3), and for each component, we generate all possible sub-graphs. Then, we calculate the cost of each sub-graph (Line 7) and if it is smaller than the current minimum, the minimum cost and best sub-graph are updated (Lines 8-9). Finally, we add the best sub-graph to the final list (Line 11) and move to the next component.

To generate the sub-graphs, we do not need to do an exhaustive generation of 2^n combinations, but we can rely on the observation that each non-dense node must be merged to exactly one dense node. Therefore, sub-graph generation is reduced to finding all possible assignments of dense nodes to the non-dense nodes. An example of this can be seen in Figure 3. In the figure, nodes c_2, c_4, c_5 are non-dense, while nodes c_6, c_7, c_8 are dense. All possible and meaningful sub-graphs are enumerated in the table at the right of the figure, where we assign a dense node to each of the non-dense nodes. An assignment is only possible if there exists a parent-child relationship between a non-dense node and a dense node, even if it is an inferred one (e.g. c_2 is an inferred parent of c_7). Hence, the problem of sub-graph generation becomes one of generating combinations from different lists, by selecting one element from each list. The number of lists is equal to the number of non-dense nodes, and the elements of each list are the dense nodes that are related to the non-dense node.

Complexity Analysis. Assuming that a connected component g has k non-dense nodes and d dense nodes, and each non-dense node k_i is related to $e(k_i)$ dense nodes, then the number of sub-graphs that need to be enumerated are $\prod_{i=1}^k e(k_i)$. In the example of figure 3, the total number of sub-graphs is $e(c_2) \times e(c_4) \times e(c_5) = 3 \times 2 \times 1 = 6$. In the worst case all k nodes are parents of all d nodes. Then, the number of total sub-graphs is d^k , which makes the asymptotic time complexity of the algorithm $O(d^k)$.

4.1 Greedy Approximation

For very small d, k (e.g. $d, k < 4$), the asymptotic complexity of $O(d^k)$ is acceptable. However, in real-world cases, the number of connected components can be small, making d and k large. For this reason, we introduce a heuristic algorithm for approximating the problem, that does not require enumerating all possible combinations, relying instead on a greedy objective function that attempts to find the local minimum with respect to the defined cost model for each non-dense node.

Algorithm 1: *optimalMerge*

Data: An inferred hierarchy lattice L_C as a adjacency list, and a set of dense CSs D
Result: A set of optimal ancestral sub-graphs

```

1 init finalList;
2 connectedComponents ← findConnectedComponents(LC);
3 for each connectedComponent do
4   init min ← MAX_VALUE;
5   init bestSubgraph;
6   while next ← connectedComponent.generateNextSubgraph()
7     do
8       if cost(next) < min then
9         min ← cost(next);
10        bestSubgraph ← next;
11      end
12    finalList.add(bestSubgraph);
13 end
14 return finalList;
```

The main idea behind the algorithm is to iterate the non-dense nodes, and for each non-dense node, calculate r_{null} and find the dense node that minimizes this function for the given non-dense node. Then, the cardinalities will be recomputed and the next non-dense node will be examined. The algorithm can be seen in Algorithm 2. In the beginning, the algorithm initiates a hash table, *mergeMap*, with an empty list for each dense node (Lines 1-4). Then, the algorithm iterates all non-dense nodes (Line 5), and for each dense node, it calculates the cost r_{null} of merging it to each of its connected dense nodes (Lines 5-13), keeping the current minimum cost and dense node. In the end, the current non-dense node is added to the list of the dense node that minimizes r_{null} (Line 14). Notice that we do not need to split the hierarchy into connected components in order for *greedyMerge* to work.

Complexity Analysis. Given k non-dense nodes and d dense nodes, where each non-dense node k_i has $e(k_i)$ dense children, the *greedyMerge* algorithm needs $\sum_{i=1}^k e(k_i)$ iterations, because it requires iteration of all $e(k_i)$ nodes for each k_i . In the worst case, every k_i is related to all d dense nodes, requiring kd iterations. Assuming a constant cost for the computation of r_{null} , then the asymptotic complexity of the greedy algorithm is $O(kd)$, which is a significant performance when compared to the exponential complexity of *optimalMerge*.

Algorithm 2: *greedyMerge*

Data: A hash table p mapping non-dense CSs to their dense descendants, a set of dense CSs D , and a set of non-dense CSs K
Result: A hash table mapping dense CSs to sets of non-dense CSs to be merged

```

1 init mergeMap;
2 for each d ∈ D do
3   mergeMap.put(d, newList());
4 end
5 for each k ∈ K do
6   min ← MAX_VALUE;
7   init bestDense;
8   for each dk ∈ p.get(k) do
9     cost ← rnull(k, dk);
10    if cost < min then
11      min ← cost;
12      bestDense ← dk;
13    end
14    mergeMap.get(bestDense).add(k);
15 end
16 return mergeMap;
```

This process does not necessarily cover all CSs of the input dataset. For example, some CS nodes might not have any dense children. Given this, the percentage of the dataset that is covered by this process is called *dense CS coverage*. The remainder of the CSs are aggregated into one large table, T_{rest} , containing all of their predicates. If the total coverage of the merging process is large, then T_{rest} does not impose a heavy overhead in query

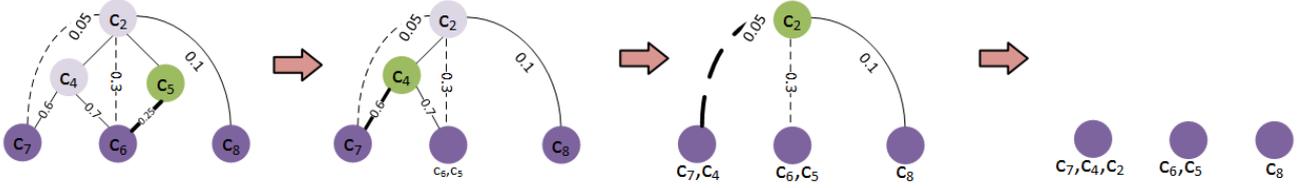


Figure 5: An example of greedy merging. Dense nodes are coloured in deep purple. At each step, the non-dense node under examination is coloured with green, while the edge that minimizes r_{null} can be seen in bold.

performance, as will be shown in the experiments. Finally, we load the data in the corresponding tables.

4.2 Implementation

We implemented *raxonDB* as a storage and querying engine that supports hierarchical CS merging, and can be deployed on top of standard RDBMS’s. Specifically, we used PostgreSQL 9.6, but *raxonDB* can be adapted for other relational databases as well.

CS Retrieval and Merging. The processes of retrieving and merging CSs take place during the loading stage of an incoming RDF dataset. CS retrieval is a trivial procedure that requires scanning the whole dataset and storing the unique sets of properties that are emitted from the subject nodes in the incoming triples, and is adopted from our previous work in [9] where it is described in detail. After retrieving the CSs, the main idea is to compute the inferred CS hierarchy and apply one of the described merging algorithms. Finally, each set of merged CSs is stored in a relational table. In each table, the first column represents the subject identifier, while the rest of the columns represent the union of the property sets of the merged CSs. For multi-valued properties, we use PostgreSQL’s array data type in order to avoid duplication of the rows.

Indexing. We deploy several indexes in *raxonDB*. First off, we index the subject id for each row. We also build foreign-key indexes on object-subject links between rows in different CSs, i.e., when a value of a property in one CS is the subject id of another CS. Next, we use standard B+trees for indexing single-valued property columns, while we use PostgreSQL’s GIN indexes, which apply to array datatypes for indexing multi-valued properties. This enables fast access to CS chain queries, i.e., queries that apply successive joins for object-subject relationships. Furthermore, we store these links on the schema level as well, i.e., we keep an index of CS pairs that are linked with at least one object-subject pair of records. These links are called Extended Characteristic Sets (ECSs) and are based on our previous work in [9]. With the ECS index, we can quickly filter out CSs that are guaranteed not to be related, i.e., no joins exist between them, even if they are individually matched in a chain of query CSs. Other metadata and indexes include the property sets of CSs, and which properties can contain multiple values in the same CS.

Query Processing. Processing SPARQL queries on top of merged CSs entails (i) parsing the queries, (ii) retrieving the query CSs, (iii) identifying the joins between them, and (iv) mapping them to merged tables in the database. Steps (i)-(iii) are inherited from our previous work in [9]. For (iv), a query CS can match with more than one table in the database. For instance, consider a query containing a chain of three CSs, $q_1 \bowtie q_2 \bowtie q_3$, joined sequentially with object-subject joins. Each query CS q_i matches with all tables whose property sets are supersets of the property set of q_i . Thus, each join in the initial query creates a set of

permutations of table joins that need to be evaluated. For instance, assume that q_1 matches with c_1, c_2 , while q_2 matches with c_3 and q_3 matches with c_4, c_5 . Assume also that by looking up the ECS index, we derived that the links $[c_1, c_3]$, $[c_2, c_3]$, $[c_3, c_4]$ and $[c_3, c_5]$ exist in the index, i.e., they correspond to candidate joins in the data. Then, $[c_1, c_3, c_4]$, $[c_1, c_3, c_5]$, $[c_2, c_3, c_4]$ and $[c_2, c_3, c_5]$ are all valid table permutations that must be processed. Two main strategies can be employed here. The first is to join the UNIONS of the matching tables for each q_i , and the other is to process each permutation of tables separately and append the results. Given the filtering performed by the ECS indexing approach, where we can pre-filter CSs based on the relationships between them, the UNION would impose significant overhead and eliminate the advantage of ECS indexing. Therefore, we have implemented the second approach, that is, process a separate query for each permutation. Finally, due to the existence of NULL values in the merged tables, we must add explicit IS NOT NULL restrictions for all the properties that are contained in each matched CS and are not part of any other restriction or filter in the original query.

5 EXPERIMENTAL EVALUATION

We implemented *raxonDB* on top of PostgreSQL³. As the focus of this paper is to improve RDF storage and querying efficiency in relational settings, we rely on existing mechanisms within PostgreSQL for I/O operations, physical storage and query planning. In this set of experiments, we report results of implementing *hier_merge* with the greedy approximation algorithm, as experimenting with the optimal algorithm failed to finish the merging process even in datasets with small numbers of CSs.

Datasets. For this set of experiments, we used two synthetic datasets, namely *LUBM2000* ($\approx 300m$ triples), and *WatDiv* ($\approx 100m$ triples), as well as two real-world datasets, namely *Geonames* ($\approx 170m$ triples) and *Reactome* ($\approx 15m$ triples). *LUBM* [6] is a customizable generator of synthetic data that describes academic information about universities, departments, faculty, and so on. Similarly, *WatDiv*[2] is a customizable generator with more options for the production and distribution of triples to classes. *Reactome*⁴ is a biological dataset that describes biological pathways, and *Geonames*⁵ is a widely used ontology of geographical entities with varying properties and rich structure.

Loading. In order to assess the effect of hierarchical merging in the loading phase, we performed a series of experiments using all four datasets. For this experiment, we measure the size on disk, the loading time, the final number of merged tables, as well as the number of ECSs (joins between merged tables) and the percentage of triple coverage by CSs included in the merging process, for varying values of the density factor $m \in [0, 1]$. The

³The code and queries are available in <https://github.com/mmeimaris/raxonDB>

⁴<http://www.ebi.ac.uk/rd/services/reactome>

⁵<http://www.geonames.org/ontology/documentation.html>

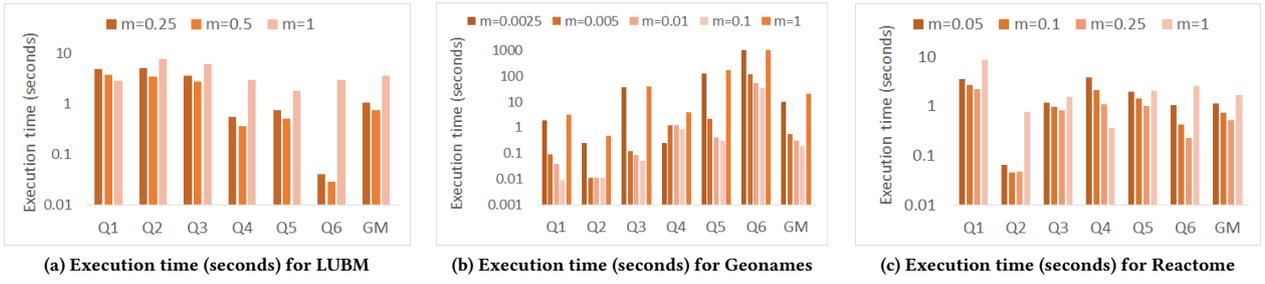


Figure 6: Query execution times in milliseconds

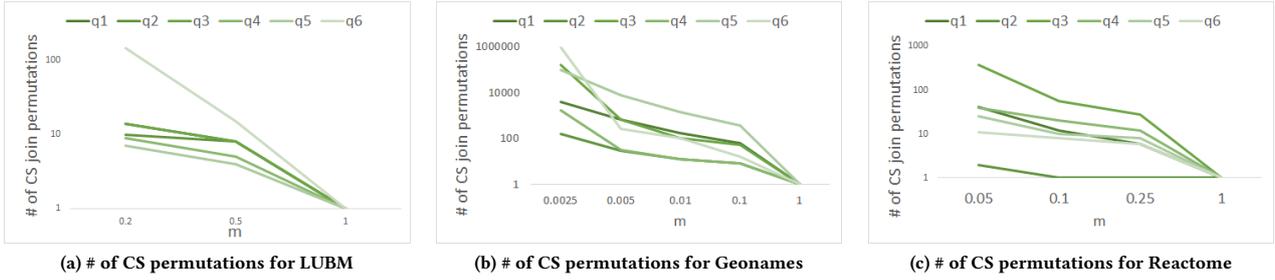


Figure 7: # of CS permutations for increasing m

Table 2: Loading experiments for all datasets

Dataset	Size (MB)	Time	# Tables (CSs)	# of ECSSs	Dense CS Coverage
Reactome Simple	781	3min	112	346	100%
Reactome (m=0.05)	675	4min	35	252	97%
Reactome (m=0.25)	865	4min	14	73	77%
Geonames Simple	4991	69min	851	12136	100%
Geonames (m=0.0025)	4999	70min	82	2455	97%
Geonames (m=0.05)	5093	91min	19	76	87%
Geonames (m=0.1)	5104	92min	6	28	83%
LUBM Simple	591	3min	14	68	100%
LUBM (m=0.25)	610	3min	6	21	90%
LUBM (m=0.5)	620	3min	3	6	58%
WatDiv Simple	4910	97min	5667	802	100%
WatDiv (m=0.01)	5094	75min	67	99	77%
WatDiv (m=0.1)	5250	75min	25	23	63%
WatDiv (m=0.5)	5250	77min	16	19	55%

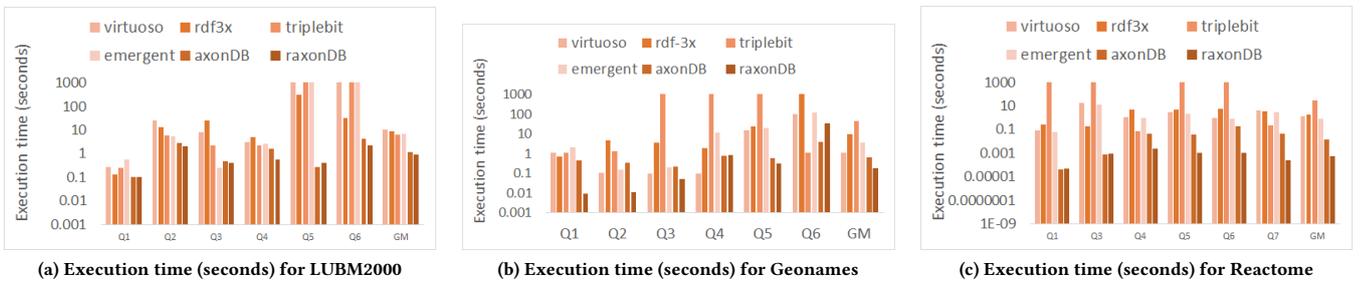


Figure 8: Query execution times in milliseconds for different RDF engines

results are summarized in Table 2. As can be seen, the number

of CS, and consequently tables, is greatly reduced with increasing values of m . As the number of CSs is reduced, the expected

number of joins between CSs is also reduced, which can be seen in the column that measures ECSs. Consequently, the number of tables can be decreased significantly without trading off large amounts of coverage by dense CSs, i.e. large tables with many null values. Loading time tends to be slightly greater as the number of CSs decreases, and thus the number of merges increases, the only exception being WatDiv, where loading time is actually decreased. This is a side-effect of the excessive number of tables (= 5667) in the simple case which imposes large overheads for the persistence of the tables on disk and the generation of indexes and statistics for each one.

Query Performance. In order to assess the effect of the density factor parameter m during query processing, we perform a series of experiments on LUBM, Reactome and Geonames. For the workload, we used the sets of queries from [9]⁶. We employ two metrics, namely *execution time* and *number of table permutations*. The results can be seen in Figures 6 and 7. As can be seen, hierarchical CS merging can help speed up query performance significantly as long as the dense coverage remains high. For example, in all datasets, query performance degrades dramatically when $m = 1$, in which case the merging process cannot find any dense CSs. In this case, all rows are added to one large table, which makes the database only contain one table with many NULL cells. These findings are consistent across all three datasets (Q_6 in LUBM exhibits a higher increase for $m = 1$ due to the dataset's characteristics; for lack of space we omit the explanation of this behaviour) and require further future work in order to identify the optimal value for m .

In order to assess the performance of *raxonDB* and establish that no overhead is imposed by the relational backbone, we performed a series of queries on LUBM2000, Geonames and Reactome, assuming the best merging of CSs is employed as captured by m with respect to our previous findings. We also compared the query performance with *rdf-3x*, *Virtuoso 7.1*, *TripleBit* and the emergent schema approach described in [16]. The results can be seen in Figure 8 and indicate that *raxonDB* provides equal or better performance from the original *axonDB* implementation, as well as the rest of the systems, including the emergent schema approach, which is the only direct competitor for merging CSs. Especially for queries with large intermediate results and low selectivity that correspond to a few CSs and ECSs (e.g. LUBM Q5 and Q6, Geonames Q5 and Q6) several of the other approaches fail to answer fast and in some cases time out.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we tackled the problem of automatically mapping heterogeneous RDF datasets to a relational schema by considering the implicit schema in RDF triples. We presented a method that extracts the Characteristics Sets, i.e., the set of properties describing the different classes of RDF instances in the data and exploits the hierarchical relationships between different CSs in order to merge and map them to relational tables. We have provided two algorithms, an optimal (exhaustive) one which selects ancestral sub-graphs of CS for merging in exponential time and greedy one, which via the use of heuristics improves the performance to polynomial time. We have implemented our methods on top of a standard RDBMS solution, i.e., PostgreSQL for extracting, indexing and query processing of SPARQL queries. Finally, we have experimented with two synthetic and two real-world datasets, all of them exhibiting high heterogeneity in their schemas, we

compared with various alternative RDF engines and the results for the performance of indexing and querying showed that our system outperforms for various types of workloads.

As future work, we will study computation of the optimal value for m , taking into consideration workload characteristics as well as a more refined cost model for the ancestral paths. Furthermore, we will study and compare our approach to a graph database setting, as well as experiment with a column-stored relational DB, in order to further scale the capabilities of *raxonDB*.

Acknowledgments. This work is partially funded by the projects *VisualFacts* (#1614 - 1st Call of the Hellenic Foundation for Research and Innovation Research Projects for the support of post-doctoral researchers), and "Moving from Big Data Management to Data Science" (MIS 5002437/3)- Action "Reinforcement of the Research and Innovation Infrastructure" (funded by Greece and the European Regional Development Fund).

REFERENCES

- [1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. 2007. Scalable semantic web data management using vertical partitioning. In *VLDB*.
- [2] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *ISWC*.
- [3] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *ACM SIGMOD*.
- [4] Orri Erling and Ivan Mikhailov. 2010. *Virtuoso: RDF support in a native RDBMS*. Springer.
- [5] A. Gubichev and T. Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*.
- [6] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.
- [7] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1997. Index Selection for OLAP. In *ICDE*, 208–219.
- [8] Maciej Janik and Krys Kochut. 2005. BRAHMS: a workbench RDF store and high performance memory system for semantic association discovery. In *ISWC*.
- [9] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. 2017. Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In *ICDE*.
- [10] Marios Meimaris, George Papastefanatos, Panos Vassiliadis, and Ioannis Anagnostopoulos. 2016. Efficient Computation of Containment and Complementarity in RDF Data Cubes. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. 281–292. <https://doi.org/10.5441/002/edbt.2016.27>
- [11] Marios Meimaris, George Papastefanatos, Panos Vassiliadis, and Ioannis Anagnostopoulos. 2018. Computational methods and optimizations for containment and complementarity in web data cubes. *Inf. Syst.* 75 (2018), 56–74. <https://doi.org/10.1016/j.is.2018.02.010>
- [12] G. Montoya, H. Skaf-Molli, and K. Hose. 2017. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC*.
- [13] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*.
- [14] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (2010), 91–113.
- [15] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. 2014. H 2 RDF+: an efficient data management system for big RDF graphs. In *ACM SIGMOD*.
- [16] MD Pham and P. Boncz. 2016. Exploiting emergent schemas to make RDF systems more efficient. In *ISWC*.
- [17] MD. Pham, L. Passing, O. Erling, and P. Boncz. 2015. Deriving an emergent relational schema from rdf data. In *WWW*.
- [18] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. 2014. Sempala: interactive SPARQL query processing on hadoop. In *ISWC*.
- [19] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF querying with SPARQL on spark. In *VLDB*.
- [20] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. 2008. Column-store support for RDF data management: not all swans are white. In *VLDB*.
- [21] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. In *VLDB*.
- [22] Kevin Wilkinson. 2006. Jena property table implementation.
- [23] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. In *VLDB*.

⁶Available also in <https://github.com/mmeimaris/raxonDB>