Relational Schema Optimization for RDF-based Knowledge Graphs

George Papastefanatos[a], Marios Meimaris[a], Panos Vassiliadis[b]

[a] *Athena Research Center, Athens, Hellas*
[b] *University of Ioannina, Ioannina, Hellas*

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. We have implemented our algorithm on top of a relational backbone, where each merged CS is stored in a relational table, and therefore, CS merging results in a smaller number of required tables to host the source triples of a data set. Moreover, we perform an extensive experimental study to evaluate the performance and impact of merging to the storage and querying of RDF datasets, indicating significant improvements. We also conduct a sensitivity analysis to identify the stability and any possible weaknesses of our algorithm, and report on our results.

1. Introduction

The Resource Description Framework[1] (RDF) and SPARQL Protocol and RDF Query Language[2] (SPARQL) are W3C recommendations for representing and querying data on the semantic web and are widely used for the management of knowledge graphs. 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

Email addresses: gpapas@athenarc.gr (George Papastefanatos), m.meimaris@athenarc.gr (Marios Meimaris), pvassil@cs.uoi.gr (Panos Vassiliadis)

[1] https://www.w3.org/RDF/

[2] https://www.w3.org/TR/sparql11-overview/

found to be very efficient in simple workloads, but not efficient enough when it comes to more complex query workloads [1][2][3][4][5][6][7].

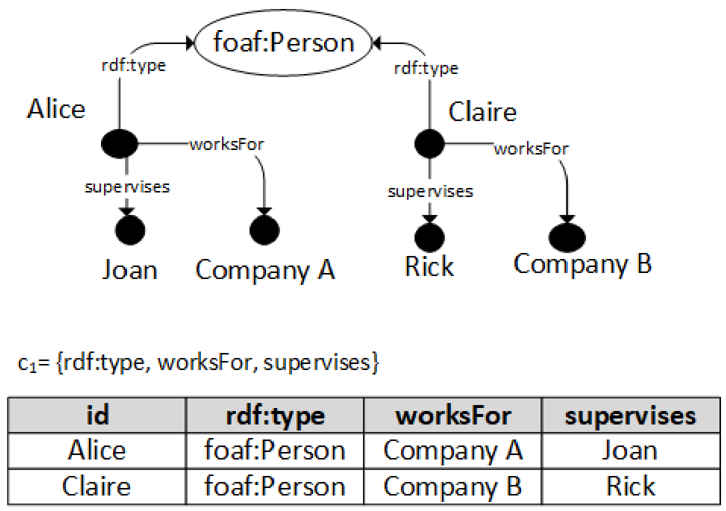
In response to this limitation, recent works have shown that extraction and exploitation of the implicit schema of the data can be beneficial in both storage and SPARQL query performance [3][4]. In order to organize on disk, index and query triples efficiently, these efforts heavily rely on two structural 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 [8]:

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

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 [3, 2, 4, 8, 9]. 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.

However, a mapping from RDF to the relational space is not always straightforward, 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, frequently 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), imposing large overheads in the extraction, storage and disk-based retrieval[1][3]. 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



c_1= {rdf:type, worksFor, supervises}

id	rdf:type	worksFor	supervises
Alice	foaf:Person	Company A	Joan
Claire	foaf:Person	Company B	Rick

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

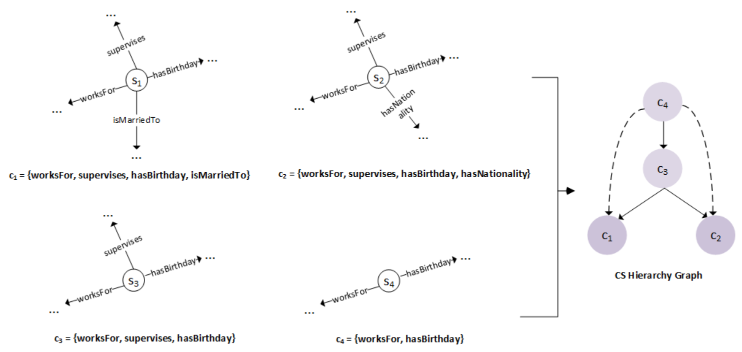
Table 1: RDF datasets along with their number of CSs and links between CSs.

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

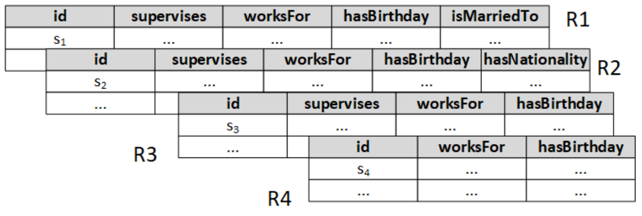
different CS would result in a large numbers of relational tables with few tuples in each of them, as shown in Figure 2(b), that require a very large number of joins to answer queries; (ii) on the other hand, creating a single, "universal" table, accommodating the CS's of all the nodes of the graph would 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.

The problem is both open and hard to solve. This is due to the following reasons.

A design-driven solution is not applicable. One could possibly 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


supervises
hasBirthday
worksFor
isMarriedTo
s_1
c_1 = {worksFor, supervises, hasBirthday, isMarriedTo}
s_2
hasNationality
c_2 = {worksFor, supervises, hasBirthday, hasNationality}
s_3
c_3 = {worksFor, supervises, hasBirthday}
s_4
c_4 = {worksFor, hasBirthday}
c_4
c_3
c_1
c_2
CS Hierarchy Graph

(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).



R1

id	supervises	worksFor	hasBirthday	isMarriedTo
s_1	...	...	...	...

R2

id	supervises	worksFor	hasBirthday	hasNationality
s_2	...	...	...	...
...				

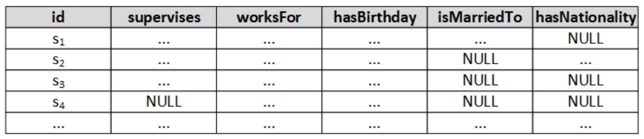
R3

id	supervises	worksFor	hasBirthday
s_3	...	...	...
...			

R4

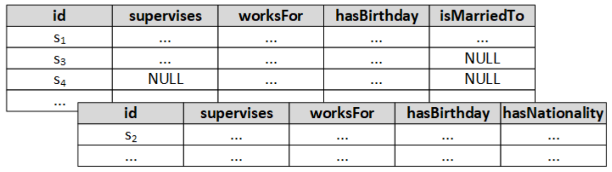
id	worksFor	hasBirthday
s_4	...	...
...	...	...

(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 are more instances belonging to these CSs, not shown in the figure.

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.

State of the art is not addressing the problem. Second, 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.

The problem requires finding a sweet spot. At the same time, the problem is hard to solve: as we show in Section 5, the complexity is exponential, and therefore, brute force methods are not adequate. The problem is driven by the opposing forces of space consumption and query efficiency. Depending on the underlying relational storage scheme, the translation of a SPARQL query to SQL requires completely different joins and self-joins, with very different table sizes being involved. We elaborate on this via an illustrative example in Section 4. Practically, we need a balance in the following two factors that push the solution to different edges of a spectrum:

- One possible edge of the spectrum is to define one table for each CS. In this case, the penalty to pay is that the number of unions of joins needed to answer queries that will be posed to the dataset reaches its maximum. Thus, we need to push towards merging CSs into the same relational storage (which is actually the driving force for our solution).
- The other end of the spectrum is that we define a single "universal" table for all the data set, with the union of all properties as its attributes. In this case we pay the price of a largely empty table, with a large number of NULL values and unnecessary space consumed. Thus, we need to constrain the tendency to merge CS's with a stopping force that prevents the creation of very few, ultra-big, NULL-filled merger tables.
- The point is to find a sweet spot between the two extremes that compromises the different costs.

Contribution. *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.*

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. We also conduct a sensitivity analysis that allows us to evaluate how sensitive the proposed algorithm is to the density threshold that it employs, as well as to small fluctuations of the relational schema.

In short, our contributions are as follows:

- We formulate the problem as a lattice reduction problem and introduce a novel CS merging algorithm that takes advantage of CS hierarchies;
- We present our implementation, *raxonDB*, an RDF engine built on top of a relational backbone that takes advantage of this merging for both storing and query processing (which we detail in length);
- We perform an experimental evaluation that indicates significant performance improvements for various parameter configurations.
- We perform a sensitivity analysis on the robustness of our proposed algorithm.

An earlier, condensed version of this paper has appeared in [10].

Roadmap. In Section 2, we present the background and related work for this paper. In Sections 3 and 4, we provide preliminary definitions and delineate the problem, and in Section 5 we present algorithms towards its solution. In Section 6, we discuss how query processing takes place in our RDF engine, whose architecture is presented in Section 7. In Section 8, we discuss the experimental evaluation of the paper. In Section 9 we perform a sensitivity analysis of our method and report on our results. We conclude the paper in Section 10, with a summary of our findings and a discussion of future work.

2. Related Work

The recent survey [5] presents a most complete list of RDF systems; it categorizes them based on the underlying storage technologies, the querying capabilities as well as the nature of processing (centralized vs distributed approaches). Due to the tabular structure that tends to implicitly underlay RDF data, some of these 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 [11] and Hexastore [12] build tables on all six permutations of SPO. Built on a relational backbone, Virtuoso [13] 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 [14] and DB2RDF [15], 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 [16]. Also, it raises performance issues when handling complex queries with many joins, as the amounts of intermediate results increase [17].

Vertical partitioning segments data in two-column tables. Each table corresponds to a property, and each row to a subject node [16]. This provides great performance for queries with bound objects, but suffers when the table sizes have large variations in size [18]. TripleBit [19] 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 [20], H2RDF [21] and S2RDF [22]. 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 previous works [3],[23], we defined *Extended Characteristic Sets (ECSs)* as typed links betwen CSs, and we showed how ECSs can be used to index triples and greatly improve query performance. In [1], 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 [2] 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 [4], the authors use CSs and ECSs in order to assist cost estimation for federated queries, while in [9], [24], the authors use CSs in order to provide better triple reordering plans. Then, [25] targets the problem of query containment in SPARQL and proposes a graph index for accelerating containment checking. These approaches can be combined with our work to provide fast results to specific classes of queries over an RDBMS. 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.

We would like also to highlight that the core research question of this paper is not to contrast relational with triple-store technology, but assuming that the relational storage is given, how we can make it more efficient for managing RDF data. RDBMS are super-mature software products, building their ecosystem of tools, robustness, scalability and availability upon a mature technology of almost half a century. Clearly, triple stores, as a domain-specific solution, come with all the benefits that the so-called "dinosaurs" don't have: they are specifically adapted to the problem at hand and can apply optimizations to their code that the general-purpose RDBMSs cannot. However, assuming that the general-purpose RDBMSs will be completely obliterated or not used for storing triples is a rather extreme scenario in our opinion. This is not only due to the tools, expertise and stability properties that they carry, but also due to the simple social fact that developers might stick to technology they already master. Finally, stressing and exploring the limits of relational technology in the service of storing triples and answering questions is also a valid research topic.

The earlier version of this paper [10] is extended here via a detailed discussion of the problem formulation, the detailed exposition of our query processing method for raxonDB, the presentation of the system architecture as well as a detailed sensitivity analysis.

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 [3, 1, 8]. *Characteristic Sets* (CS)[8] 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 $p_1 \ldots p_n$. 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.

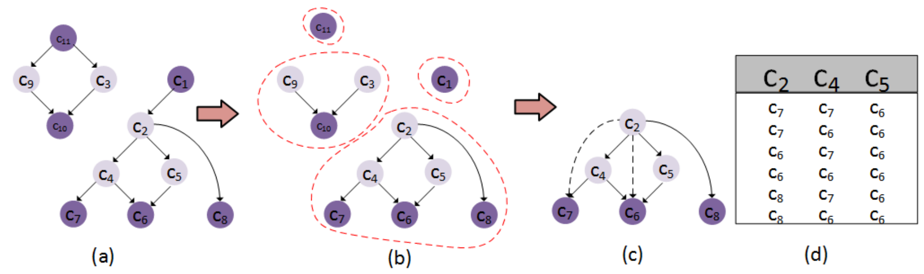
Definition 1. ***Property Sets and Property Tables***. In its more general form, a CS is a property set P_i, i.e., a set of RDF predicates and the set of all property sets is denoted with P. A *Property Table* (also: *CS Table*) T_i for a given property set P_i is a relational table comprising an identifier *sid* for the subject node s and $|P_i|$ columns, where $P_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,n}\}$ 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 can hold values for the predicate columns in $I \cup B \cup L \cup NULL$, i.e., each cell in the relational table can either have a standard value of an RDF triple object, or $NULL$ when the subject node identified in *sid* does not have a property in any of its triples. Intuitively, a table can be used as storage for a CS if it has one column for each of the properties of the CS, as well as an extra column for a primary key *sid*. Then, all subjects of the CS can be stored into the table (and as we will see right next, all subjects of CSs that have a subset of the properties of the CS under discussion – therefore the need for NULL in the above discussion). 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 $c_i \succ c_j$, when c_i is a subset of c_j, or $c_i \subset c_j$.

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. 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 $c_i \succ c_j$, 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 $c_1 \succ c_2$ and there exists no other c_i such that $c_1 \succ c_i \succ c_2$. The directed edge stems from the parent node and arrives at the child node.



C2	C4	C5
C7	C7	C6
C7	C6	C6
C6	C7	C6
C6	C6	C6
C8	C7	C6
C8	C6	C6

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.

In the case of the aforementioned $c_1 = \{type, name\}$ and $c_2 = \{type, name, marriedTo\}$, a directed edge starts from the ancestor node c_1 towards the descendant node c_2. Hierarchies in our context have no relationship with hierarchies in other domains, like e.g., OLAP. If the reader prefers, she can think of them in terms of type hierarchies, where children types extend parent types with extra attributes. Practically, as we will also see next, a descendant CS, if translated to a relational table in terms of storage, it can also host all its ancestral CS's in its relational table, with NULL values for the columns that the ancestors lack. CS subsumption relationships can also be seen in Figure 3(a) as directed edges between nodes. 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. 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 [26]), although in our case, the construction is ad-hoc, depending on the available CS's, and serves a different purpose.

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).

Definition 4. ***CS Ancestral Sub-graphs***. Given an inferred hierarchy

$H_c = (V, E)$, a CS c_{base}, a set of CSs $c_1, \ldots, c_k$, and a sub-graph $H^{anc}_{c_{base}} = (V^{'}, E^{'})$ with $V^{'} \subseteq V$ and $E^{'} \subseteq E$, we say that $H^{anc}_{c_{base}}$ *is an ancestral sub-graph over* c_{base} when $\forall i \in [1..k]$, it holds that $c_i \succ c_{base}$ and $(c_i, c_{base}) \in E^{'}$.

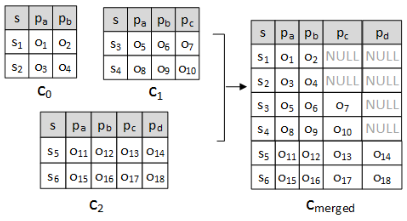
Intuitively, *any* set of ancestors of a node c_{base} forms an ancestral sub-graph. More than one ancestral sub-graphs can be defined over c_{base}, as any subset of its parents is considered an ancestral sub-graph over c_{base}. For instance, in Figure 3(c), nodes c_7, c_4, c_2 form an ancestral sub-graph 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 NULL 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 $V^{'} = \{c_1, c_2, \ldots, c_k\}$ 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 NULL 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, $c_0 \succ c_1 \succ c_2$. 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 6. ***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



s	p_a	p_b
s_1	o_1	o_2
s_2	o_3	o_4

c_0

s	p_a	p_b	p_c
s_3	o_5	o_6	o_7
s_4	o_8	o_9	o_{10}

c_1

s	p_a	p_b	p_c	p_d
s_5	o_{11}	o_{12}	o_{13}	o_{14}
s_6	o_{15}	o_{16}	o_{17}	o_{18}

c_2

s	p_a	p_b	p_c	p_d
s_1	o_1	o_2	NULL	NULL
s_2	o_3	o_4	NULL	NULL
s_3	o_5	o_6	o_7	NULL
s_4	o_8	o_9	o_{10}	NULL
s_5	o_{11}	o_{12}	o_{13}	o_{14}
s_6	o_{15}	o_{16}	o_{17}	o_{18}

c_{merged}

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

unambiguously. Also, the total number of relational tables will be equal to the number of ancestral sub-graphs in the merge graph.

4. Problem Formulation

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 unions of 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).

Illustrative Example. Before formalizing the problem formulation, we believe it is better to illustrate the problems that pop-up via an illustrative example. 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 '1992-02-24'.
        ?z isMarriedTo ?w.
        ?w hasNationality 'GR' }
```

Assuming a design with a "universal" relation (Figure 2c), the query evaluation over the relational schema (see Section 5 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 relational algebra). At the same time, there is a price of NULL values, too, as (Figure 2c) shows. Assuming a universal relation R_U, the SQL translation of the above SPARQL query would be requiring the self-join of this very large table to itself more than once:

```
SELECT x.id, x.worksFor, z.id, w.id
```

```
FROM R_U AS x, R_U AS z, R_U AS w
WHERE  x.supervises = z.id AND
       z.isMarriedTo = w.id AND
       z.hasBirthday = '1992-02-24' AND
       w.hasNationality = 'GR' AND
       x.worksFor IS NOT NULL
```

Note that the join between $?x$ and $?y$ on $worksFor$ is excluded from the produced joins, as $?y$ is practically a completely free variable; at the same time, what is required is that $?x$ does have a value for property $worksFor$.

At the other end of the spectrum, in the multiple CS tables case (Figure 2b), each one of the three join query conditions would require a two-step translation: (a) detect which tables are candidate for each variable, and use their virtual union as the virtual table to instantiate the variable, and (b) factor out the unions and create the Cartesian product of the candidates' joins. Here, for example, variable $?x$ can be populated only by tables having both properties $worksFor$, *supervises* (i.e., $R1$ and $R2$); variable $?y$ by any table; variable $?z$ only by tables having both properties *isMarriedTo*, *hasBirthday* (i.e., only $R1$), and variable $?w$ by any table having the property *hasNationality* (i.e., $R2$). Then, the "join structure" of the SPARQL query (omitting selections and projections) is:

$$?x \bowtie_{supervises} ?z \bowtie_{isMarried} ?w$$

which is translated to the following expression in relational algebra:

$$(R1 \cup R2\ ) \bowtie_{supervises} R1 \bowtie_{isMarried} R2$$

By factoring out the unions, ultimately, the query is translated to

$$(R1 \bowtie_{supervises} R1 \bowtie_{isMarried} R2) \cup (R2 \bowtie_{supervises} R1 \bowtie_{isMarried} R2)$$

To give an idea of how easy it is to explode the number of produced subqueries, the reader might want to consider the following: Had we introduced a single triple pattern on $?y$, it would produce an extra join in the join path and the need to join all the above subqueries with the union of the candidates to populate $?y$.

Thus, despite its space efficiency, the latter case imposes performance overheads due to the large number of unions of joins that 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.

Problem formulation. Assume a data set D which, at the *logical level*, is a collection of subject nodes, $D^L = \{s_1, \dots s_n\}$. Naturally, many subject nodes share the same CS. At the *physical level*, each subject node is stored in a relational table, so, we assume an underlying relational database $D^P = \{\ r_1 \dots r_r\ \}$, such that for each $s \in D^L \ \exists\ r \in D^P$ suitable to contain all the data of s. We define the total function *storage*: $D^L \rightarrow D^P$ (in other words, for each subject node there exists exactly one table where all its data are going to be hosted). By abuse of notation we say that $storage(D^L) = D^P$.

Similarly to subjects, we can define the same relationships for CSs. A CS is a *representative* for a subject, if for each property of a subject, the CS includes it. A CS is a *strict representative* of a subject if it contains exactly the properties of this subject and none other. CSs form hierarchies, which we formally address as an inferred graph $H_c = (V, E)$, in the sense of Definition 3.

A relation r is a legal placeholder for the data of a subject node when its schema includes at least (but not obligatorily exactly) an *id* column for the node id and a set of columns, one per property of the CS of the subject node. We formalize this relationship by introducing the function *legalHost*: $D^L \times D^P \rightarrow Boolean$, i.e., $legalHost(s, r)$ is true when r is a legal placeholder for s. A physical-level relational database D^P is a legal placeholder for the data of a data set D with a logical-level set of subject nodes DL when for each subject node s there exists a table r, s.t., $legalHost(s, r)$. By abuse of notation we say that $legalHost(D^L, D^P)$ is true.

The inverse of function *storage* is not necessarily a function, and it's not necessary that it is total. The idea here is that a relational table might store data from multiple nodes, with different CSs. For example, we have already seen that in the case of subsumption, we can store all the properties of subsumed nodes into the same relation. Thus, we have several alternatives for the storage function, i.e., several alternative sets of tables D^P for the storage of the same RDF dataset D.

Finally, the problem is reduced to the following: 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 fast way to merge CS nodes via a merging process of low complexity, while at the same time, minimizing an objective cost function $cost()$ that penalizes the production of large merger tables.

5. 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 $|r_1| >> |r_2|$, 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[1].[3] 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 $c_i \succ c_j$:

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

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.

As an example, consider the tables T_1=*(sid, rdfType, worksFor)* with $r_1 = 1K$ records and T_2=*(sid, rdfType, worksFor, supervises, memberOf)* with $r_2 = 9K$ records; merging T_1 to T_2 will produce $r_{null}(T_1, T_2) = 0.2$. r_{null} takes positive values and increases proportionally to the difference in the number of columns $|c_1| - |c_2|$ between the two merged tables and the number of rows $|r_1|$ of the small table. Thus, merging a small table with a bigger one, which have slight differences in their schemas, results in small values for r_{null}.

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:

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

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. In the previous example, we merge a third table T_3=*(sid, worksFor)* with $r_3 = 2K$ records with T_2. Then $r^{g}_{null}(g)|_{T_2}$ =0.66.

Thus, we define a cost function that works on the graph level, as follows:

$$cost(g) = \sum_{i=1}^{n} r^{g}_{null}(g_i)|_{c_{di}} \tag{3}$$

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

In our running example, instead of T_1, we choose T_4=*(sid, rdfType, worksFor, supervises)* with 12K records to merge with T_1 and T_3. In that case the cost

[3]We employ the term $\setminus$ for set difference in accordance to a long tradition of set theory – see `https://mathworld.wolfram.com/SetDifference.html`

will become $r^g_{null}(g)|_{T_4}$ =0.33, i.e., it is lower than that of T_2 acting as a dense node even though the size of T_4 is bigger.

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 $d : N \to R$, 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, Algorithm 1 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.

It first identifies all connected components of the inferred hierarchy (Line 2) using a standard DFS traversal (not explained in the algorithm). Then, it iterates each component (Line 3) and generates all possible sub-graphs (Line 6). For each sub-graph, the cost is calculated and if it is smaller than the current minimum (Line 7), the minimum cost and best sub-graph are updated (Lines 8-9). Finally, the best sub-graph is added to the final list (Line 11), which is the output of the algorithm.

To generate the sub-graphs (method generateNextSubgraph), 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.

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 ones. An assignment is only possible if there exists a parent-child relationship between a non-dense 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 in 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 $O(d^k)$.

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  $connectedComponents \leftarrow findConnectedComponents(L_c)$;
3  for each $connectedComponent$ do
4      init $min \leftarrow MAX\_VALUE$;
5      init $bestSubgraph$ ;
6      while $next \leftarrow connectedComponent.generateNextSubgraph()$ do
7          if $cost(next) < min$ then
8              $min \leftarrow cost(next)$;
9              $bestSubgraph \leftarrow next$;
10     end
11     $finalList.add(bestSubgraph)$;
12 end
13 return $finalList$;
```

5.1. 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.

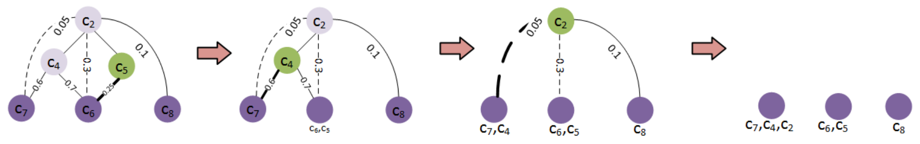


Figure 5: An example of greedy merging. Dense nodes are coloured in deep purple. At each step, the non-dense node under examination is coloured with green, while the edge that minimizes r_{null} can be seen in bold.

The main idea behind Algorithm 2 is to iterate the non-dense nodes, and for each of them, to 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. In the beginning, the algorithm initiates a hash table, $mergeMap$, with an empty list for each dense node d (Lines 1-4). Then, the algorithm iterates all non-dense nodes, denoted via k (Line 5), and for each dense child d_k of k, it calculates the cost r_{null} of merging k with this dense child (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. Figure 5 shows an example of greedy merging.

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 $optimalMerge$.

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

6. Query Processing over Relational CSs

In this section, we present the algorithms used to process SPARQL queries, via their translation to SQL queries, over the schema that has been produced by the greedy algorithm of the previous section. We refer to Table 2 as an example of a SPARQL query and the relational tables over which it is evaluated.

Processing a SPARQL query on top of the relational database of CSs entails (i) parsing the SPARQL syntax, (ii) retrieving the CSs that are candidates to

```
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 $d \in D$ do
3      $mergeMap.put(d, newList())$;
4  end
5  for each $k \in K$ do
6      $min \leftarrow MAX\_VALUE$;
7      init $bestDense$;
8      for each $d_k \in p.get(k)$ do
9          $cost \leftarrow r_{null}(k, d_k)$;
10         if $cost < min$ then
11             $min \leftarrow cost$;
12             $bestDense \leftarrow d_k$;
13     end
14     $mergeMap.get(bestDense).add(k)$;
15 end
16 return $mergeMap$;
```

Table 2: A sample database over which the SPARQL is evaluated

Tables	Columns
c_1	rdfType, worksFor, supervises
c_2	worksFor, supervises, isFriendOf
c_3	rdfType, hasBirthday, isMarriedTo
c_4	rdfType, hasNationality
c_5	supervises, hasNationality

```
SELECT ?x ?y ?z ?w
WHERE { ?x worksFor ?y.
?x supervises ?z.
?z hasBirthday '1992-02-24'.
?z isMarriedTo ?w.
?w hasNationality 'GR' }
```

instantiate the query variables and the joins between them, (iii) formulating the SQL syntax of the query by looking up the dictionary and mapping CS to tables in the database, and finally, (iv) executing the query for producing the results.

Typically, a SPARQL query like the one just mentioned, is translated to an SQL query that contains UNIONS of subqueries, with each subquery including as many joins as the joins between SPARQL variables. Several times, the same table participates in different subqueries, and in fact, possibly in different roles.

Main idea and terminology. In fact, the idea behind this translation is as follows:

1. For each *subject variable* of the SPARQL query (i.e, for every variable s that participates in a statement $s\ p\ o$ within the SPARQL query), we have to create a list of candidate CS's, and consequently, find their host tables that would possess all the properties needed for the particular query, in a list of candidate data providers of the variable. To facilitate the identi-

fication of candidate CSs, we create a set of required properties for each subject variable. Assuming s is the subject variable under investigation, the union of all properties p_i of any statement within the query of the form $s\ p_i\ o_i$, represents the set of all properties that a CS *must* have in order to provide valid instantiations for the variable. For every subject variable s, we call the set of these properties *subject variable property set*, or *variable CS* for short.

2. For each join of two variables of the query, we have to create the Cartesian product of the candidate data providers of the two variables; then, iterate the creation of the respective Cartesian product for all the joins of the SPARQL query variables.

3. Each member of the Cartesian product, is a candidate subquery with joins between them and the result is produced as their union. In the rest of our deliberations, we will use the term *subquery* to refer to these subquery members of this Cartesian Product. Practically speaking, the produced SQL query is a UNION query of several subqueries, with each subquery containing the appropriate join between candidate CSs, one for each variable of the join.

Algorithmic steps. In Algorithm 3, the algorithm that produces the SQL translation of an incoming SPARQL query is presented. The algorithm first retrieves the list of Properties P_D, the list of characteristics sets CS_D and the joins ECS_D between the CSs from the data dictionary (*Lines 2*). It then parses and validates the SPARQL syntax and extracts the triple patterns T_Q contained in the `WHERE` clause (*Line 3*). We consider simple SPARQL queries that contain triple patterns of the following 3 forms, covering the fundamental types of data in RDF:

```
{?var2 predicate ?var2}
{?var2 predicate URI}
{?var2 predicate literal}
```

i.e., we do not consider variable in the predicate part or any `OPTIONAL` and `FILTER` expressions.

In *Lines 4,6*, the algorithm creates the collections that are used for building the SQL expression of the query. It first creates the list of variables CSs for the variables of the query. To speed up the process, instead of using a bag of variable CS's for a query, we use a map (without duplicates, that is) such that identical work is avoided in the subsequent steps. The extraction of a CS involves identifying the common subsets of properties in T_Q, and mapping them to the list P_D. The resulting collection CS_Q contains the different *variable CS* extracted from the query. Then, the algorithm extracts triples that contain projected variables $Project_Q$ as well as restrictions $Restr_Q$ and assigns them to CSs, in order to construct the `SELECT` and `WHERE` clauses, respectively. A restriction is a triple whose object is bound to a URI or a literal – practically, selections in terms of relational algebra.

Algorithm 3: *SQL Query Building Algorithm*

```
   Data:  An SPARQL query $Q$
   IDs of properties in the db dictionary $P_D$
   The list of CSs in the data $CS_D$
   The list of ECS in the data $ECS_D$
   Result: A string $sql$ with the syntax of the SQL query
1  $sql \leftarrow \emptyset$ ;
   // load dictionaries from database
2  $(P_D, CS_D, ECS_D) \leftarrow dbLookup()$;
   // extract the triple patterns t of $Q$
3  $T_Q[t] \leftarrow parse(Q)$;
   // extract variable CSs from $Q$
4  $CS_Q(v, p[]) \leftarrow extractCS(T_Q, P_D)$;
   // extract triples with projected variables or restrictions
      and map them to variable CSs
5  $Project_Q(v, t[]) \leftarrow extractProjections(T_Q, CS_Q)$;
6  $Restr_Q(v, t[]) \leftarrow extractRestrictions(T_Q, CS_Q)$;
   // extract pairs of variables with joins in $q$
7  $ECS_Q(v_i, v_j) \leftarrow extractECS(T_Q, CS_Q)$ ;
   // map variable CS to CS in the data
8  $CSMap(v, cs[]) \leftarrow \text{MapCS}(CS_Q, ECS_Q, CS_D, ECS_D)$;
   // create lists of CSs that will form join subqueries
9  $subqueries(cs[]) \leftarrow \text{getSubqueries}(CSMap)$;
   // create SQL syntax
10 $sql \leftarrow createSQL(CSMap, Restr_Q, CS_Q, subqueries, ECS_Q, P_D)$;
11 return $sql$;
```

The next step in the query processing method (Line 7) is to identify pairs of variable CSs in the query, which are called Extended Characteristic Sets ECS_Q [3]. ECSs are pairs of CSs for which at least one subject-object relationship exists between their variables – practically, the equivalent of relational joins. In our reference example, the SPARQL query contains three CSs, $v_x = \{worksFor, supervises\}$, $v_z = \{hasBirthday, isMarriedTo\}$, $v_w = \{hasNationality\}$, two ECSs (v_x,v_z) and (v_z,v_w), and finally two restrictions, $?z$ `hasBirthday '1992-02-24'` and $?w$ `hasNationality 'GR'`. The step aims at the construction of *chain patterns*, i.e., linked lists of CSs in the query that form subsequent subject-object joins. We use the list ECS_Q to identify links between ECSs; a link between two ECSs exists when the CS acting as the pair value of one ESC is the CS acting as the pair key of another ECS, i.e., (v_i, v_j) $\rightarrow$ (v_j, v_k). These links are used to create CS chains, which represent the way CS tables in the database will be synthesized in joins. In our query example, the only chain derived is ($v_x \rightarrow v_z \rightarrow v_w$), due to the triples $?x$ `supervises` $?z$ and $?z$ `isMarriedTo` $?w$.

Function `MapCS` (*Line 8*) involves matching each variable CS to the underlying tables by using CSs definitions in the dictionary. Each variable CS v_i matches with all CSs produced from the data set and stored in relational tables whose property sets are supersets of the property set of v_i. A variable CS can match with more than one table in the database. In our case, $v_x \rightarrow \{c_1, c_2\}$, $v_z \rightarrow \{c_3\}$ and $v_w \rightarrow \{c_4, c_5\}$.

```
1  Function MapCS($CS_Q$, $ECS_Q$, $CS_D$, $ECS_D$):
2      $CSMap \leftarrow \emptyset$ ;
3      for each $(v_i, v_j) \in ECS_Q$ do
4          for each $(cs_i, cs_j) \in ECS_D$ do
5              if $(cs_j \subseteq v_j)\&(cs_i \subseteq v_j)$ then
6                  CSMap.add($v_i, cs_i$) ;
7                  CSMap.add($v_j, cs_j$) ;
8          end
9      end
       // do the same for CS that do not participate in Joins
10     for each $v \in CS_Q$ do
11         if !($v \in ECS_Q$) then
12             for each $cs \in CS_D$ do
13                 if $cs \subseteq v$ then
14                     CSMap.add($v, cs$) ;
15             end
16     end
17     return $CSMap$ ;
```

Due to the multiple matches between a variable CS and available CSs (i.e., tables) in the data, each join in the SPARQL query creates a set of SQL subqueries of table joins that need to be evaluated. For pruning the number of subqueries, the process first tries to match variable CSs with CSs of the data via the ECS_D index and then looks up remaining variable CSs (ones that do not participate in a Join condition) in the full list of CSs in the data dictionary. Assume, that by looking up the ECS_D, we derived that the links (c_1, c_3), (c_3, c_4) and (c_3, c_5) exist, i.e., they correspond to joins in the data. That means, that v_x *only* matches with c_1 (c_2 is not considered), while v_z and v_w match with c_3 and c_4, c_5, respectively. Then, (c_1, c_3, c_4), (c_1, c_3, c_5) are only valid subqueries that must be processed. The identification of the different subqueries is performed in function `getSubqueries()` (Line9). It iterates over the mappings in $CSMap$ and produces the cartesian product between the CSs matched for each variable. In our example, the chain $(v_x \rightarrow v_z \rightarrow v_w)$ will produce the following subqueries $c_1 \otimes \mathrm{c}_3 \otimes (\mathrm{c}_4, c_5)$.

Two main strategies can be employed here.

- The first is to join the `UNION`s of the matching tables for each v_i, i.e., $v_x \bowtie v_z \bowtie v_w \rightarrow c_1 \bowtie c_3 \bowtie (c_4\ UNION\ c_5)$.
- The second is to process each identified subquery separately and union the results, i.e., $(c_1 \bowtie c_3 \bowtie c_4)\ UNION\ (c_1 \bowtie c_3 \bowtie c_5)$.

Given the filtering performed by the ECS indexing approach, the first approach would impose significant overhead and eliminate the advantage of ECS indexing. Therefore, we have implemented the second approach, that is, form a separate query for each subquery.

```
1 Function getSubqueries($CSMap$):
      // different combinations of joins between CS in the data
2     $subqueries \leftarrow \emptyset$ ;
3     for each $v_i \in CSMap$ do
4         $cs_i[] \leftarrow CSMap(v_i)$;
5         subqueries.addAll(subqueries $\otimes cs_i[]$) ;
6     end
7     return $subqueries$ ;
```

For creating the `SELECT` clause, we parse the SELECT part and examine the projected variables: (i) if the variable is a subject variable then we include in the projection list the id column of the matched CS table (e.g., $c_1.id$); (ii) otherwise, if the variable is an object, we project the name of column, which corresponds to the predicate having as object this variable and belongs to the CS table of the subject (e.g., $c_1.worksFor$).

The final step is to construct the `WHERE` clause based on the $Restr_Q$ collection. Each restriction is translated to a filter condition; conditions involving multi valued properties use the `ANY` expressions. Also, due to the existence of `NULL` values in the merged tables, the process adds explicit `IS NOT NULL` conditions for all properties in the tables in the `FROM` clauses, which are contained in

```
Function CreateSQL($CSMap()$, $Restr_Q$, $CS_Q$, $subqueries$, $ECS_Q$, $P_D$):
    $where \leftarrow' Where'$ ;
    $select \leftarrow' Select'$ ;
    $from \leftarrow' From'$ ;
    // for each subquery create a separate FROM clause
    for each $\{cs_1, cs_2, ..., cs_n\} \in subqueries$ do
        $from \leftarrow cs_1 \bowtie cs_2 \bowtie ... \bowtie cs_n$ ;
        // create SELECT from projections in $q$
        for each $v \in CSMap$ do
            for each $t \in Project_Q(v)$ do
                if ($t.var$ is subject) then
                    $select \leftarrow' id'$ ;
                else
                    $select \leftarrow' t.p'$ ;
                end
            end
            // create WHERE clause from restrictions in $q$
            for each $p \in CS_Q(v)$ do
                if ($p \in Restr_Q(v).t[]$) then
                    $where \leftarrow' p\ =\ t.object'$ ;
                else
                    $where \leftarrow' p$ is not null$'$ ;
                end
            end
        end
        $sql \leftarrow select + from + where$ ;
        $sql \leftarrow sql + union$ ;
    end
    return $sql$ ;
```

the initial SPARQL query and are not part of a restriction. The query formulation is provided by the function `CreateSQL()`. The final SQL for the database and SPARQL presented in the beginning of this section is presented below.

```
SELECT x.id, x.worksFor, z.id, w.id
FROM c1 AS x, c3 AS z, c4 AS w
WHERE   x.supervises = z.id AND
        z.isMarriedTo = w.id AND
        z.hasBirthday = '1992-02-24' AND
        w.hasNationality = 'GR' AND
        x.worksFor IS NOT NULL
UNION
SELECT x.id, x.worksFor, z.id, w.id
FROM c1 AS x, c3 AS z, c5 AS w
WHERE   x.supervises = z.id AND
        z.isMarriedTo = w.id AND
        z.hasBirthday = '1992-02-24' AND
        w.hasNationality = 'GR' AND
        x.worksFor IS NOT NULL
```

Next we provide the cost analysis and a short discussion on the correctness of Algorithm 3.

Complexity analysis: Let t be the number of triple patterns in the `WHERE` clause of the SPARQL query, $v \leq t$ the number of variable CSs in the query, i.e., $|CS_Q|$, $a \leq t$ the number of CS pairs that participate in joins, i.e., $|ECS_Q|$, and r and j are the numbers of CSs and ECSs in the data, i.e. $|CS_D|$ and $|ECS_D|$, respectively. Lines 1-3 come at a minimum cost, as they initialize the dictionaries in memory (performed once on server startup) and parse the query syntax. Lines 4-7 perform one pass on t query's triple patterns for populating the lists CS_Q, $Project_Q$, $Restr_Q$ and a second pass for finding the ordered pairs in ECS_Q, i.e., $O(2 * t)$. For the cost of Line 8: `MapCS` function, a pairs of CSs will be compared to j pairs in the data; i.e., $a * j$ and any remaining CS will be matched against the r records. In the worst case, $a = t$ (all query triples involve CS joins), and the cost for assessing `MapCS` becomes $t*j$, i.e., every query triple is compared against the ECSs in the database. Next, the `getSubqueries` function (Line 9) produces a cross products of all CSs matched in the data. Let m be the average number of matching tables per variable (i.e., the average size of CSMap collection), then the cost for producing the subqueries is m^{a+1}. In the worst case, all v variable CSs are matched to all r CSs in the database and $a = t$, i.e., $O(r^{t+1})$. Finally, the cost of the SQL syntax formulation (Line 14) is determined by the $|subqueries|$, i.e, r^{a+1}. Also, for each variable CSs, a lookup is made in the $Project_Q$ and $Restr_Q$ lists (each takes $O(1)$ and the total is $2 * v$) to formulate the `SELECT` and `WHERE` clauses of the SQL query. Overall, the maximum cost is $O(2 * t + t * j + r^{t+1} + r^{t+1} * v)$, which is bound by $O(t * j + r^{t+1} * v)$. However, in practise $v, a < t$, and the cost is better approximated by $O(a * j + m^{a+1} * v)$.

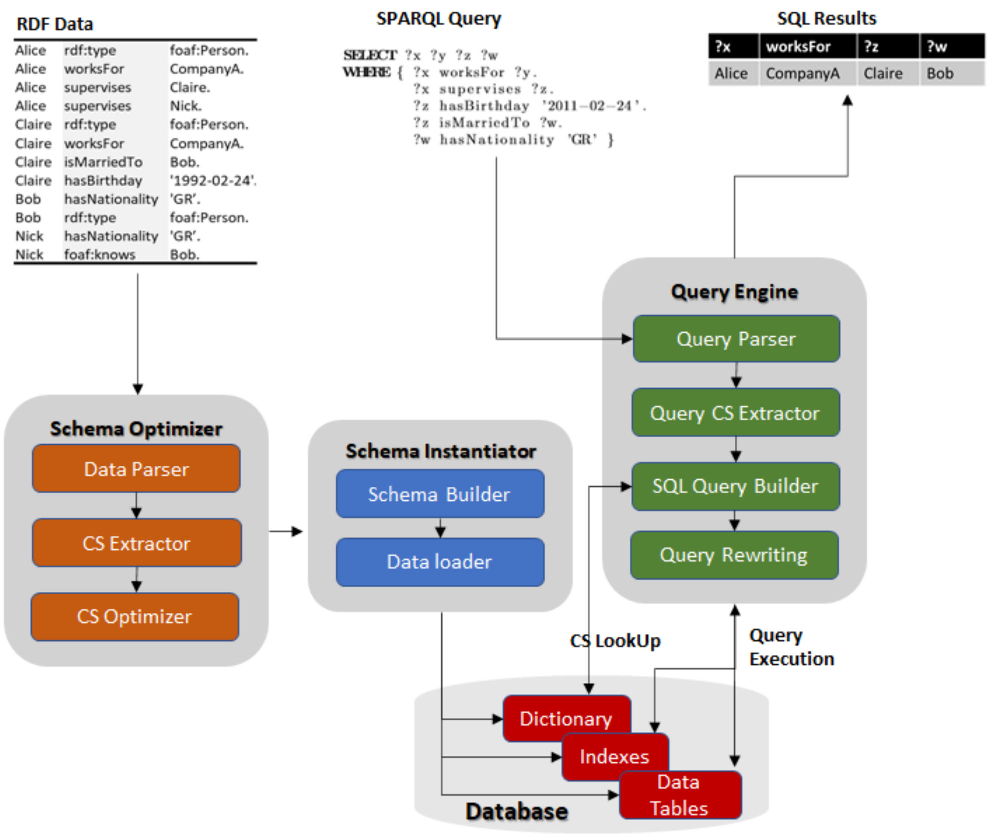
Correctness: *How do we guarantee that the sparql-to-sql rewriting of the above algorithm is correct, i.e., both queries return the same results when evaluated over an RDF graph or the relational equivalent database*? Recall that, our relational representation of an RDF graph maps CSs to tables, subject nodes to relational records (with ID the URI of the node), edges to relational columns and object nodes to single or composite values.

Regarding the `WHERE` clause of a SPARQL query, a triple pattern can be one of the forms ($?s$ p $?o$) or ($?s$ p URI | literal). A subject node is bound to a node on the rdf graph, which emits an edge with the type of the predicate p (i.e., properties) and references a node with the value of the object o. In our equivalent SQL query, i) each subject variable is mapped to (v_1, v_2, ..., v_n) tables whose schema contain *all* the properties referenced by the variable; the `UNION` of these tables contain the candidate entities of the query. We filter the recordset by ii) mapping all ($?s$ p URI | literal) triples (called restrictions in the algorithm) to the conjunction of ($v_i.p$ = URI | literal) condition expressions. Next, for each ($?s$ p $?o$) triple, we produce iii) a filter ($v_i.p$ `IS NOT NULL`) condition, for fetching those records with a not-null value for this property; iv) a join ($v_i.op = u_j.id$) condition between the `UNION` of v_i tables of the subject and the `UNION` of u_j tables of the object and v) the expressions $v_i.id$ or $v_i.p$ in the `SELECT` clause of the SQL query for each subject or object variable included in the `SELECT` clause of the SPARQL, respectively. For producing the final `FROM` clause, we rewrite the join of unions $(v_1 \cup v_2 \cup ...v_n) \bowtie (u_1 \cup u_2 \cup ...u_m)$ as unions of join subqueries; i.e., $(v_1 \bowtie u_1) \cup (v_1 \bowtie u_2) \cup ...(v_1 \bowtie u_m) \cup ...(v_n \bowtie u_m)$. Note that, the filter and join expressions formulated in the `WHERE` clauses refer to the tables via aliases and thus they are not affected from the rewriting within each subquery. The correctness of the above rewriting is based on the properties of relational algebra regarding the semantics of the `UNION` and `JOIN` operators and it can be generalized to union and joins between multiple tables.

7. System Implementation

We have implemented *raxonDB* as a storage and querying engine that supports the relational schema design for RDF data through hierarchical CS merging. The engine is built in Java and can be deployed on top of any RDBMS which offers support for non-scalar array types (SQL:1999). The architecture of *raxonDB* can be seen in Figure 6. The basic modules are the *Schema Optimizer*, which parses the input file, determines and optimizes the relational schema, the *Relational Schema Instantiator*, which constructs the schema and loads the data into the RDBMS; the *Database* which stores the data in relational tables, and maintains indexes and the dictionary; and finally the *Query Engine*, which translates a SPARQL into an SQL query and executes it over the database.

Schema Optimizer. The Schema Optimizer includes the *Data Parser*, the *CS Extractor*, and the *CS Optimizer*. The Data parser parses an input datafile which contains RDF triples in .nt format, validates the input, and represents all URI strings as integers, for reducing the memory footprint and computational overhead during processing of the data as well as the space needed for their storage. All triples are kept in memory in an array of integers. Next, the CS Extractor extracts the list of CSs; it first sorts the array on the subject column, and identifies the different sets of properties that are emitted from a subject


RDF Data

Alice	rdf:type	foaf:Person.
Alice	worksFor	CompanyA.
Alice	supervises	Claire.
Alice	supervises	Nick.
Claire	rdf:type	foaf:Person.
Claire	worksFor	CompanyA.
Claire	isMarriedTo	Bob.
Claire	hasBirthday	'1992-02-24'.
Bob	hasNationality	'GR'.
Bob	rdf:type	foaf:Person.
Nick	hasNationality	'GR'.
Nick	foaf:knows	Bob.

SPARQL Query

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

SQL Results

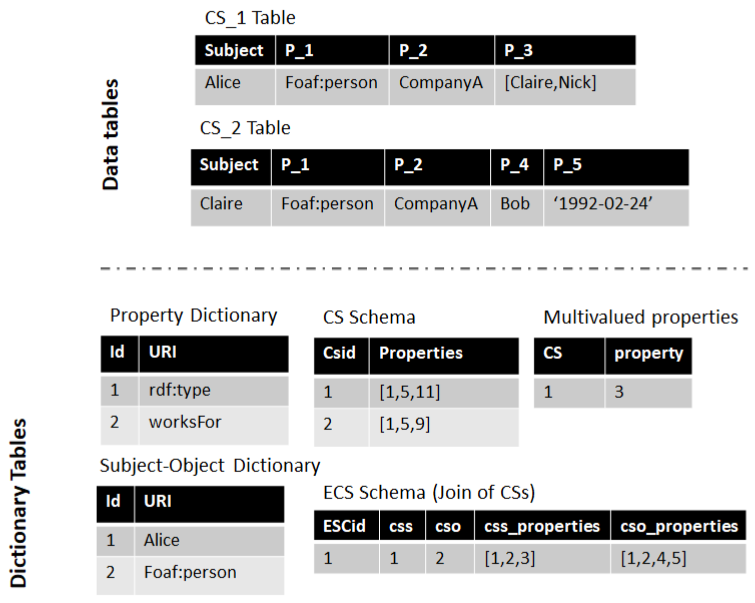
?x	worksFor	?z	?w
Alice	CompanyA	Claire	Bob

Figure 6: Architecture of *raxonDB*.

node. For each different combination of properties, a new CS is constructed and the respective triples are assigned to it. Operating on the list of CSs, the CS Optimizer computes the inferred CS hierarchy graph (i.e., the different connected components) and applies the algorithm of Section 5 for finding the best merging paths and constructing the final CSs.

Relational Schema Instantiator.The Relational Schema Instantiator builds the SQL `CREATE` commands for each CS and creates the tables in the database. In addition, it loads the data in the tables and builds the indexes and the dictionary tables.

Database Storage and Indexing. Each CS produced by the Data Loader is stored in a separate Data Table CS_i (i is an index given by the Data Loader during CS merging) with the following schema $CS_i = (s, p_1, p_2, ..., p_n)$. The first column contains the subject identifier (primary key), while all other columns correspond to the properties of the CS. For multi-valued properties, for which the same subject-property refers to multiple objects (e.g., *(Alice, supervises, Claire)* and *(Alice, supervises, Nick)* of input data in Fig. 6), we use an SQL


Data tables
Dictionary Tables

CS_1 Table

Subject	P_1	P_2	P_3
Alice	Foaf:person	CompanyA	[Claire,Nick]

CS_2 Table

Subject	P_1	P_2	P_4	P_5
Claire	Foaf:person	CompanyA	Bob	'1992-02-24'

Property Dictionary

Id	URI
1	rdf:type
2	worksFor

CS Schema

Csid	Properties
1	[1,5,11]
2	[1,5,9]

Multivalued properties

CS	property
1	3

Subject-Object Dictionary

Id	URI
1	Alice
2	Foaf:person

ECS Schema (Join of CSs)

ESCid	css	cso	css_properties	cso_properties
1	1	2	[1,2,3]	[1,2,4,5]

Figure 7: Example of tables constructed by *raxonDB*.

array data type (e.g., we store in the property column an array with all objects corresponding to the same subject, $[Claire, Nick]$), in order to avoid duplication of the rows. For space efficiency, all columns, including the subject column, store the integer representation instead of the actual URIs. The *Dictionary* tables keep all necessary metadata about the CSs, such that incoming SPARQL queries can be properly translated to the various data tables and values. First, the dictionary keeps the schema of all CSs, i.e., the property columns comprising each table. It also keeps the mappings of the integer representation of the subjects, properties and objects to the URIs. It, finally, contains the list of multi-valued properties and the CS tables they belong to, such that SQL-translated queries can employ array semantics in their syntax. The tables constructed by raxonDB for the example input dataset are shown in Fig. 7. Note that data tables actually store integer values instead of URIs (strings), which are resolved via the subject-object dictionary. For clarity, we have included the original URIs of the triples in the input file.

In addition to the data and dictionary tables, the database contains a set of indexes used for enabling fast query evaluation over the data. As mentioned, we build a primary key index on the subject column of each CS table. 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.

We also build a join table for holding the ECS, i.e, object-subject links between triples in different CSs. For example, if CS_1 contains the triple *(Al-*

ice, supervises, Claire) and CS_2 the triple *(Claire,isMarriedTo,Bob)*, then we store the reference of the $CS_2.subject$ column to the $CS_1.supervises$ column. This enables fast access to CS chain queries, i.e., queries that apply successive joins for object-subject relationships (e.g., the query pattern $?x, supervises, ?z.$ $?z, isMarriedTo, ?w$ of the SPARQL query in Fig 6).

We store these links on the schema level as a separate table, which contains all CS pairs that are linked with at least one object-subject pair of records. In the above example, the index keeps a record with the CS_1 and CS_2 as well as all the properties they contain. Their use for query optimization of chain queries has already been shown in Section 6. 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 query's CSs.

Query Engine. The query engine part implements the steps and algortihms described in Section 6.The *Query Parser* validates the SPARQL syntax and extracts the triple patterns contained in the `WHERE` clause. Next the *Query CS Extractor* identifies the common sets of properties emitted by variables acting as subjects in the query's triple patterns and maps them to variable CSs, denoted as $v_1, v_2, ..., v_n$. It also identifies joins between the extracted variable CSs, projected variables and finally maps restrictions to CSs.

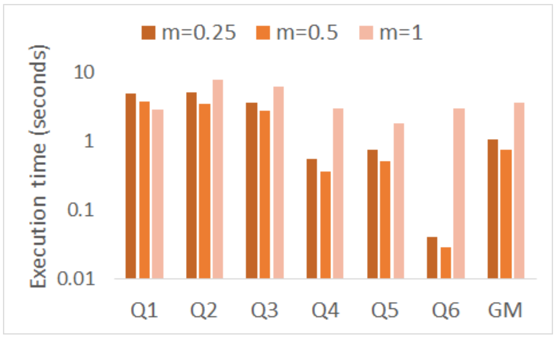
The *SQL Query Builder* tries to match the variable CSs to the underlying tables by using CSs definitions in the dictionary. Finally, a rewriting is performed by the *Query Rewriting* module, which also adjusts the SQL syntax applying any other necessary conditions expressed in the initial SPARQL query. Also, due to the existence of NULL values in the merged tables, the *Query Rewriting* module adds explicit IS NOT NULL conditions for all the properties that are contained in a matched CS and are not part of a restriction or filter in the original query.

8. Experimental Evaluation

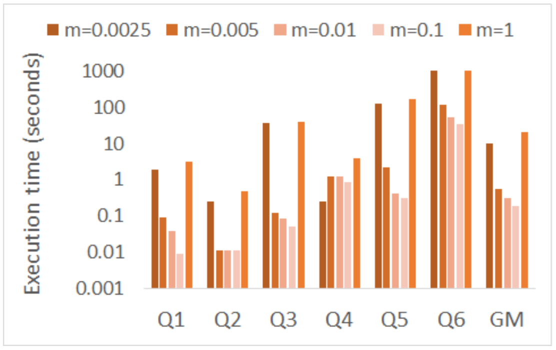
We implemented *raxonDB* on top of PostgreSQL[4]. As the focus of this paper is to improve RDF storage and querying efficiency in relational settings, we rely on existing mechanisms within PostgreSQL for I/O operations, physical storage and query planning. In this set of experiments, we report results of implementing *hier_merge* with the greedy approximation algorithm, which provides faster construction of the relational schemas for all datasets.All experiments were performed on a server with Intel i7 3820 3.6GHz, Debian v3.2.0 and allocated memory of 16GB.

Datasets. For this set of experiments, we used two synthetic datasets, namely *LUBM2000* (≈300m triples), and WatDiv (≈100m triples), as well as two real-world datasets, namely *Geonames* (≈170m triples) and *Reactome* (≈15m triples). LUBM [27] is a customizable generator of synthetic data that describes academic information about universities, departments, faculty, and so

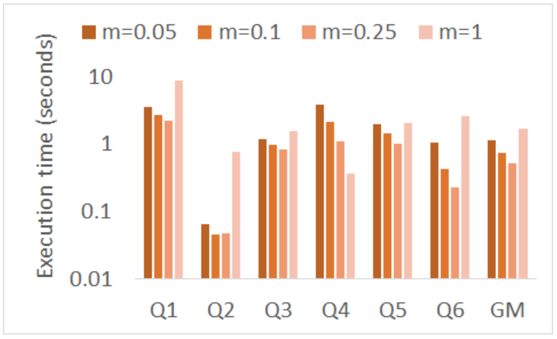
[4]The code and queries are available in https://github.com/gpapastefanatos/raxonDB



(a) Execution time (seconds) for LUBM



(b) Execution time (seconds) for Geonames

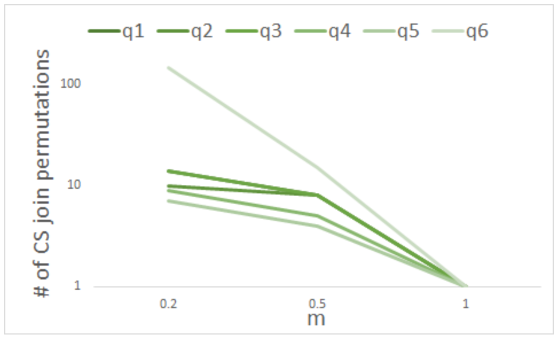


(c) Execution time (seconds) for Reactome

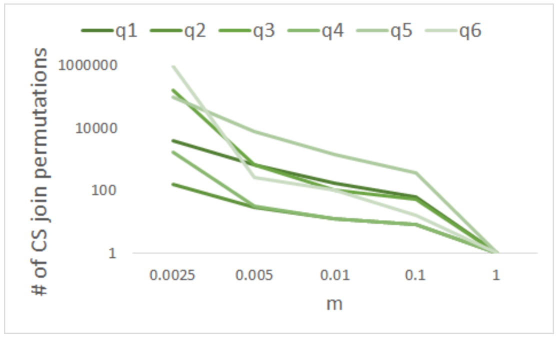
Figure 8: Query execution times in milliseconds

on. Similarly, WatDiv[28] is a customizable generator with more options for the production and distribution of triples to classes. *Reactome*[5] is a biological

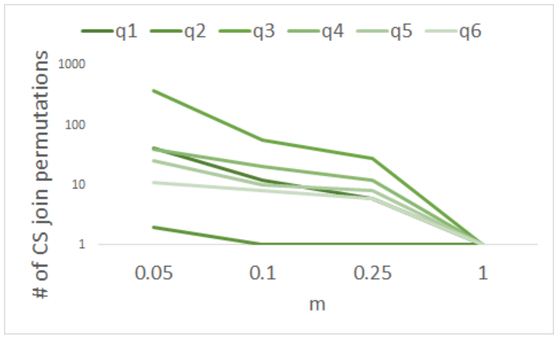
[5]http://www.ebi.ac.uk/rdf/services/reactome



(a) #subqueries for LUBM



(b) #subqueries for Geonames



(c) #subqueries for Reactome

Figure 9: Number of subqueries for increasing m

dataset that describes biological pathways, and *Geonames*[6] 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

[6]http://www.geonames.org/ontology/documentation.html

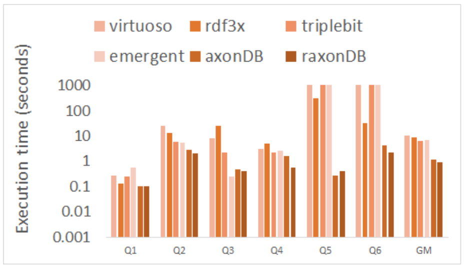
Table 3: Loading experiments for all datasets

Dataset	Size (MB)	Time (min)	# Tables (CSs)	# of ECSs	Dense CS % Coverage
Reactome Simple	781	3	112	346	100%
Reactome (m=0.05)	675	4	35	252	97%
Reactome (m=0.25)	865	4	14	73	77%
Geonames Simple	4991	69	851	12136	100%
Geonames (m=0.0025)	4999	70	82	2455	97%
Geonames (m=0.05)	5093	91	19	76	87%
Geonames (m=0.1)	5104	92	6	28	83%
LUBM Simple	591	3	13	68	100%
LUBM (m=0.25)	610	3	6	21	90%
LUBM (m=0.5)	620	3	3	6	58%
WatDiv Simple	4910	97	5667	802	100%
WatDiv (m=0.01)	5094	75	67	99	77%
WatDiv (m=0.1)	5250	75	25	23	63%
WatDiv (m=0.5)	5250	77	16	19	55%

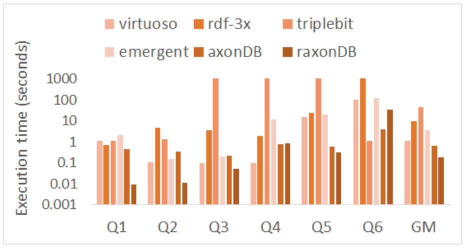
phase, we performed a series of experiments using all four datasets. For this experiment, we measure the size on disk, the loading time, the final number of merged tables, as well as the number of ECSs (joins between merged tables) and the percentage of triple coverage by CSs included in the merging process, for varying values of the density factor $m \in [0, 1]$. The results are summarized in Table 3. 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 in the data 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 [3][7]. We employ two metrics, namely *execution time* and *number of subqueries*. The results can be seen in Figures 8 and 9. 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

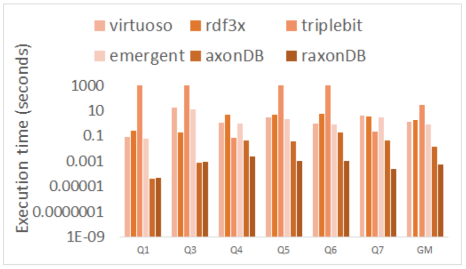
[7] Available also in https://github.com/gpapastefanatos/raxonDB



(a) Execution time (seconds) for LUBM2000



(b) Execution time (seconds) for Geonames



(c) Execution time (seconds) for Reactome

Figure 10: Query execution times in milliseconds for different RDF engines

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 fact that it is much more complicated than the other queries of the Lubm workload and by involving a very big table in two self-joins brings the system to its limits). In Section 9, a more detailed sensitivity analysis is presented regarding the impact of m on the database size and the performance of the query workload.

Competitors. 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. The density factor m was set at 0.6. We also compared the query performance with rdf-3x, Virtuoso 7.1, TripleBit and the emergent schema approach described in [2]. The results can be seen in Figure 10 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.

9. Sensitivity Analysis

The previous sections reveal a solution to the schema determination problem for RDF storage in relational engines. However, in the presence of data collection from diverse sources and with diverse properties per subject, is the greedy algorithm that we propose robust, in the sense that it maintains a reasonable performance even if the solution is not optimal?

To address this question, we have experimented with an answer to the problem of what would happen if we would add or remove tables from the solution produced from the greedy algorithm? Is there any major difference in performance?

9.1. Sensitivity Analysis for the Density Factor

The key parameter for the final number of tables is the density factor $m \in [0, 1]$. The selection of m has impact on the total number of tables and their extent, the nulls in columns as well as the query performance over the schema that will result from this selection. In this section, we discuss in details the impact of m through an example on the LUBM100 ($\approx$ 1.3M triples) and the workload of 6 queries presented in Figure 13. To include the aspect of data set size, too, we have worked with other variants of LUBM too, of different sizes, and the behavioral trends as identical; thus, we only report LUBM100 here.

Impact on the number of tables and their extent. Small numbers of m will create more tables, with $m = 0$ eliminating any merging effect and creating a separate table for each CS. Note that, the total number of records in a database for different values of m remains the same; changes in m only reorganize records to different tables. Small numbers of m build smaller tables on average. Large numbers of m merge different CSs into a few large tables with $m = 1$ resulting in at most 2 tables, one corresponding to the most dense node (attracting any parent CSs in its subgraph) and another corresponding to the aggregation of all other disconnected CS into a single table. Figure 11(a) and (b) show the impact of m on the number and the average size of tables. For $m = 0$, the database contains 13 tables, and as it decreases, the total number falls to 7, 5, 4, 3 and 2 for random different values of m. The average table size starts from $\approx$ 167K records and increases over $\approx$ 1M records for m close to 1.

Impact on the number of nulls. Subsequently, the selection of m determines the total number of nulls in the schema. A smaller value of m builds a

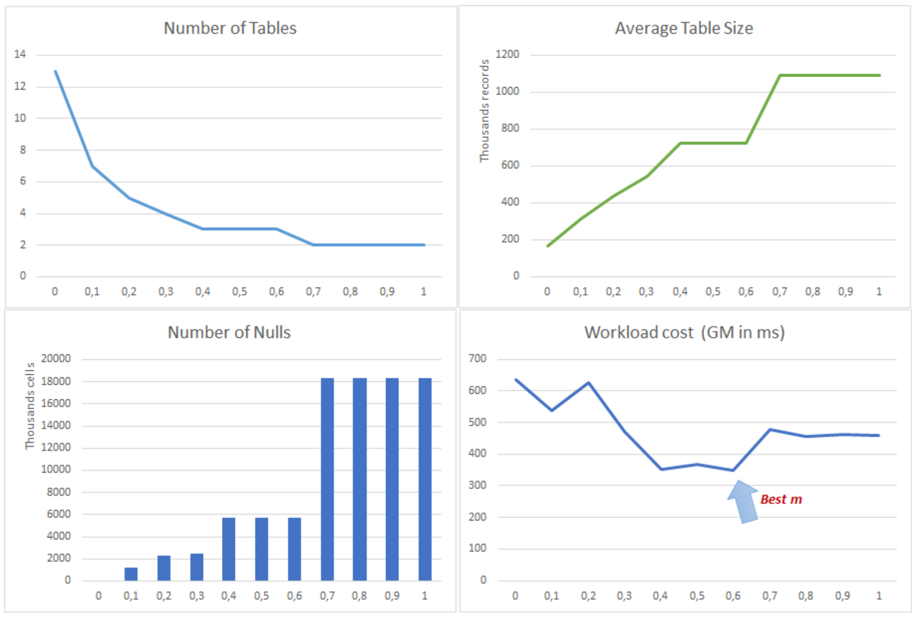


Figure 11: Impact of Density Factor on characteristics of LUBM100

more "compact" schema, with tables containing only a few columns with null values (null values drop to 0 for $m = 0$). On the contrary, the merging of a parent CS to its child results in filling with nulls all parent's cells for the columns that are defined in the child CS, and did not exist in the parent's CS. Thus, large values of m create tables with large number of null values (≈18M null values for $m > 0.7$).

Impact on workload performance. The performance of queries is primarily affected by the size of tables accessed by each query, especially by those that participate in join conditions. As briefly discussed in Section 8, varying the number of tables constructed by m for a specific schema has a result on *the total number of subqueries* that need to be combined via the appropriate `UNIONs` for answering an SQL query over this schema. Subqueries correspond to the different matches that a variable CS has to the underlying data CSs and each one is translated to a separate subquery concatenated via UNION clauses in the final SQL query and whose evaluations bring additional cost on the performance. Figure 11 (d) shows the geometric mean of the cost of the 6 queries for the different schema variants constructed based on m. For the schema created by small values of m ($0 \ldots 0.3$), most queries perform chains of joins and UNIONS to fetch the results. For values of m between 0.4 and 0.6, where the tables have been merged to 3, the cost is improved, due to the reduction in the number of subqueries needed. For this range of m the workload makes use of the optimal number of tables and achieves to best match its CSs to those of the underlying schema (see also Figure 9(a) for the decrease of the numbner of subqueries as m increase in LUBM). Finally, for values greater than 0.7, the performance deteriorates. Although the number of UNIONs remain similar in numbers, tables are now *oversized* and queries need costly self-joins with many `NOT NULL` filters to evaluate.

Concluding remarks. There are a few remarks to be made here.

- The most striking observation is the workload behavior of Figure 11: the two antagonizing forces are clearly demonstrated in this figure. Pushing towards too many tables (small m, making several CS appear as dense) increases the subqueries produced, and the result is that the system spends too much time storing the interim results of the subqueries, to facilitate their UNION. Pushing towards too few tables (large m, i.e., few CSs considered dense and attracting all others to their table) results in big tables, with many NULL values, that ultimately end up (self-)joining with each other and spending too much time examining tuples that are not useful.
- In the middle, there is a sweet spot of ranges for m, around 50% that should be targeted for a reasonable behavior of the system. *Overall, our sensitivity analysis demonstrates that the problem is valid and there is a not insignificant range of values for the robust tuning of the parameters of the proposed solution.*
- The number of NULL values and the table sizes are direct consequences of the implications produced by the upper left diagram of Figure 11 relating the value of m to the number of tables produced.

9.2. The separatist problem

A second research question that arises, has to do with the robustness of the proposed solution by the greedy algorithm. Practically, assuming that the greedy algorithm for a given m has already produced a solution, the question is whether small variations to this solution, by adding or removing tables would greatly impact the performance of the solution.

To assess this possibility, we conduct a workload-aware sensitivity analysis. We have constructed a workload of queries and tried to see the impact to performance, by trying to modify the final schema of the relational database. The sensitivity analysis method consists of starting from a medium value for the number of tables and adding a small number of tables to the solution, by carefully extracting subsets of the existing tables into new ones.

To facilitate this problem, however, we need to address the problem of *(a) how many tables to consider adding to the existing design, by extracting them from the merged tables of our greedy algorithm, and (b) which ones exactly*?

Assume we know a workload of frequent, expected queries in advance of the design. Equivalently, or possibly, more importantly, assume that once the design has been determined and deployed, we monitor which queries posed by the users are frequent. As already explained, the SPARQL queries are translated in SQL queries that contain UNIONS of subqueries, with each subquery including as many joins as the joins between SPARQL variables. Several times, the same table participates in different subqueries, and in fact, possibly in different roles. Thus, due to the possible vastness of the joins involved, a potential road to follow would be to minimize the size of the tables involved in these joins. That would mean avoiding storing tuples that would not participate in the derived joins and succinctly constraining tables to include only tuples that are useful for the answering of the queries of the query workload.

How could this be attained? The "separatist" problem assumes that we start (a) with a solution of the query-unaware algorithm, and (b) a set of queries that would be given as input to the problem, as previously discussed, and, we "extract" from the existing tables those ones who are the best candidates to be stored as separate tables that are to be used, "solo" given the query workload.

Our algorithm to address the problem is as follows:

1. For each query, for each variable of the query, say $?v$, specify all the properties that are used in the query. In other words, for every property, say p, for which an expression of the form $?v\ p\ o$, with o being either another variable, or a constant, add p in the property-list of ? v for the query under inspection. Thus, for each variable, of each query, we have a set of properties needed in a *property-list.*

2. For each variable of a query, specify which are the Characteristic Sets that contain its property list. This means they have all the needed property for being useful in evaluating the query. Thus, for each variable, we have a list of candidate data providers.

3. For each query, create the Cartesian product of candidate data providers of the query variables.

4. Count the frequency of each Characteristic Set over the union of the queries' Cartesian products, and put the CS's in a sorted list in decreasing frequency (i.e., the most popular first).

5. Find the place in the decreasing list of popularity with the steepest descent in their differential popularity (i.e., the fraction *(old.popularity - new.popularity)/old.popularity* and stop there (see for example in the Figure how the descent changes between the 4th-5th CS and the 6th-7th CS). In our implementation, in order to control this size, we also place min and max boundaries on the number of CSs involved.

6. The final output is the sorted list of the most popular CS's in the query workload, and a limit on how many of them will be allowed, under the constrain of min required and max allowed number of separatist tables: if the point of highest differential gain falls within [*min*, .., *max*] then, this is the number of separatist tables to extract; if it is lower than the min threshold, we select *min* tables, and if it is higher than the maximum tolerable, we select *max* tables (this is also convenient for controlling the experiment).

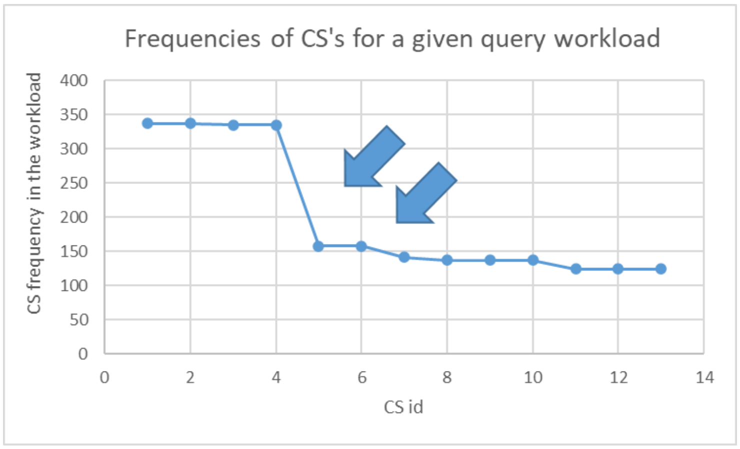


Figure 12: Distribution of frequencies for a given workload. Observe the points where there is a big difference in the derivative.

For these CS's, we intentionally isolate them as *separatists*, and assign a dedicated table that contains only them. Practically, this is facilitated by "extracting" these CS's from their merged path of the greedy algorithm into a new, "solo" table.

9.3. The implications of extra tables in the solution

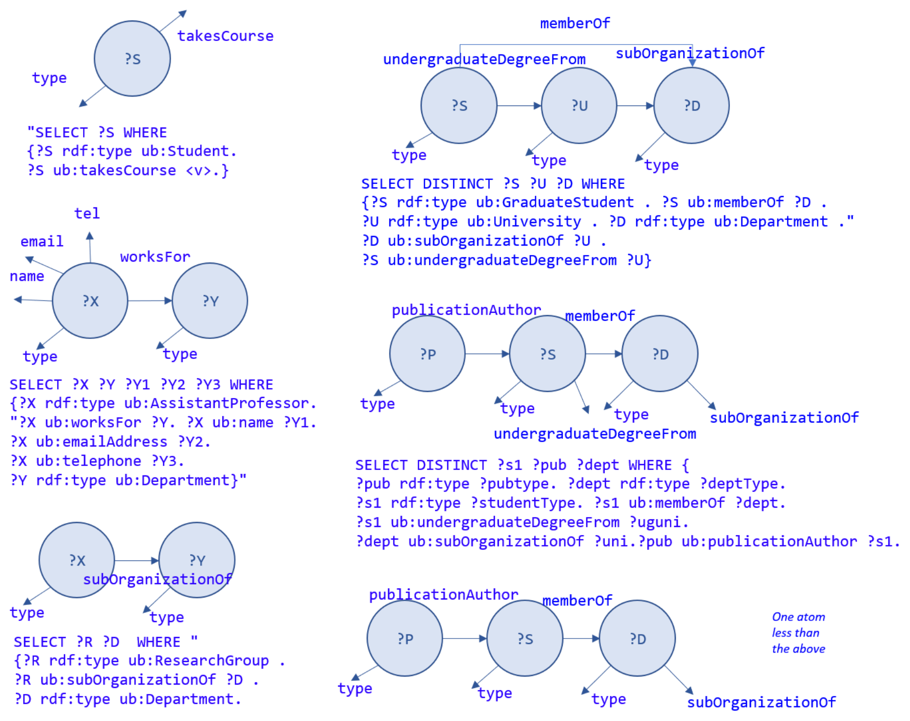
How does the set of solo, separatist tables affect the performance of the system given a workload of queries?

We have constructed a workload of queries and applied in three different versions of the LUBM dataset with a scale factor of 1, 10 and 100. We chose the LUBM data set because it allows us to control the size of data and see the impact of the data set size too, in the behavior of the system.

The workload includes queries with an increasing number of needed properties, and a variety of joins and join patterns (Figure 13). We have issued the same workload against (i) the Lubm data set at three scale factors (1, 10, 100) and (ii) alternative versions of the schema, that originally comes with 5 tables (for all scale factors) by allowing 1,2, and,3 extra popular tables to be separated. In all our results we start with a m value of 20%, such that the set of actual tables that store the data of the benchmark is a reasonably small (neither too high, or too low) number of 5 tables. Note that the best possible solution in terms of popularity includes the 4 top-most tables of Figure 12, two of which, however, are anyway isolated by the greedy algorithm. All experiments are conducted in a commodity i5 laptop with 8GB main memory.

The results (see Figures 14, 15, 16) gave clear evidence on how things operate.[8] Remember that the alternatives to the greedy algorithm are targeted

[8]Unfortunately, the experimental environment originally used in Sec. 8 was no longer avail-



```
"SELECT ?S WHERE
{?S rdf:type ub:Student.
?S ub:takesCourse <v>.}
```

```
SELECT ?X ?Y ?Y1 ?Y2 ?Y3 WHERE
{?X rdf:type ub:AssistantProfessor.
"?X ub:worksFor ?Y. ?X ub:name ?Y1.
?X ub:emailAddress ?Y2.
?X ub:telephone ?Y3.
?Y rdf:type ub:Department}"
```

```
SELECT ?R ?D  WHERE "
{?R rdf:type ub:ResearchGroup .
?R ub:subOrganizationOf ?D .
?D rdf:type ub:Department.
```

```
SELECT DISTINCT ?S ?U ?D WHERE
{?S rdf:type ub:GraduateStudent . ?S ub:memberOf ?D .
?U rdf:type ub:University . ?D rdf:type ub:Department ."
?D ub:subOrganizationOf ?U .
?S ub:undergraduateDegreeFrom ?U}
```

```
SELECT DISTINCT ?s1 ?pub ?dept WHERE {
?pub rdf:type ?pubtype. ?dept rdf:type ?deptType.
?s1 rdf:type ?studentType. ?s1 ub:memberOf ?dept.
?s1 ub:undergraduateDegreeFrom ?uguni.
?dept ub:subOrganizationOf ?uni.?pub ub:publicationAuthor ?s1.
```

Figure 13: Workload of queries used.

selections of popular characteristic sets to be isolated from their ancestral properties. The greedy algorithm is fairly stable, with several exceptions. In most of the cases, the behavior is pretty close. This means that the result of the greedy algorithm is quite close to the behavior of the alternatives. In several cases, it is actually the fasted possibility. There exist some cases, however, where the separatist algorithm significantly improves the behavior of the system. This is the clear case of the most complicated query, Q2, that involves the 3 joins between the three variables, where the gains in table size produces gains of several orders of magnitude in terms of query time. At the same time, in most of the queries tested, as the number of tables involved increases, the execution time slightly increases too, as the number of unioned subqueries increases too, and there is an extra cost of saving the interim results to the disk.

As an overall assessment, we believe that based on our experimentation, *the separatist approach provides goods results, if the expected workload is complicated. If however (as typically expected) the queries are simple, the greedy*

able when we performed the experimental analysis in this section. Although the behavior is consistent, the reader is kindly instructed to avoid linking the exact numbers of the two different sections.

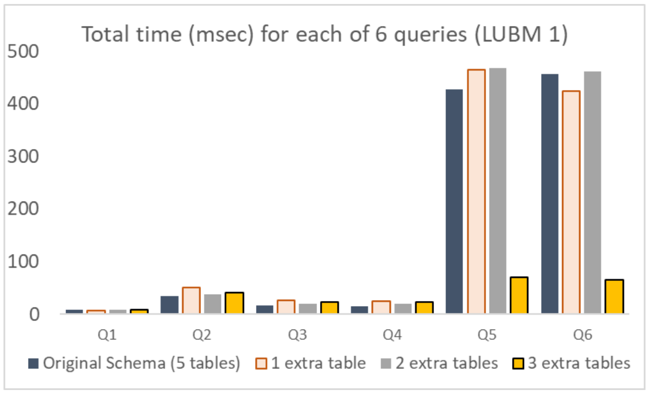


Figure 14: Execution of workload for Lubm 1

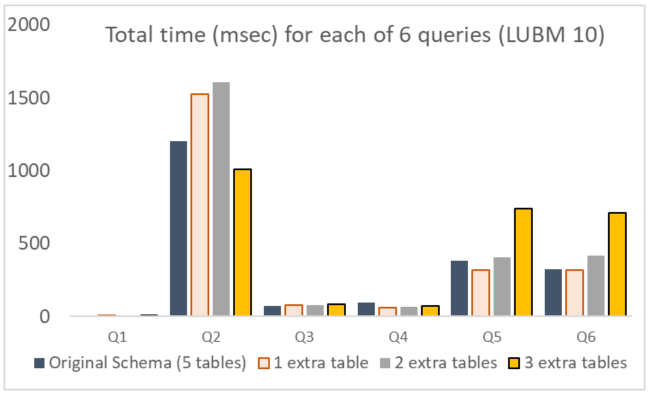


Figure 15: Execution of workload for Lubm 10

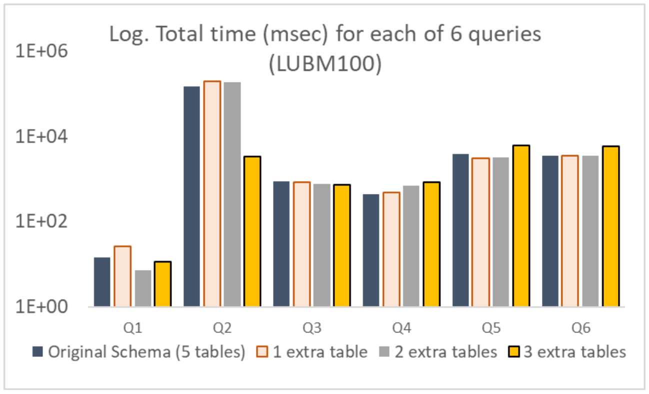


Figure 16: Execution of workload for Lubm 100

algorithm produces quite results that are fairly stable and close to the average behavior of the alternatives produces by the separatist algorithm.

10. Conclusions and Future Work

In this paper, we have tackled the problem of automatically mapping heterogeneous RDF datasets to a relational schema by considering the implicit schema in RDF triples. We have 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 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. Moreover, we have experimented with two synthetic and two real-world datasets, all of them exhibiting high heterogeneity in their schemata, 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. Finally, apart from presenting in detail the query processing method and the system architecture, we have also performed a sensitivity analysis that demonstrates that the algorithm's result is robust to small changes in the database schema and that the choice of m, although it can potentially affect the query performance significantly, comes with a fairly wide "sweet spot" of values near 0.5 that allow good query performance.

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 funded by the project *VisualFacts* (#1614 - 1st Call of the Hellenic Foundation for Research and Innovation Research Projects for the support of post-doctoral researchers).

References

[1] M. Pham, L. Passing, O. Erling, P. Boncz, Deriving an emergent relational schema from rdf data, in: Proceedings of the 24th International Conference on World Wide Web, 2015, p. 864–874. `doi:10.1145/2736277.2741121`.

[2] M.-D. Pham, P. Boncz, Exploiting emergent schemas to make rdf systems more efficient, in: P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, Y. Gil (Eds.), The Semantic Web – ISWC 2016, 2016, pp. 463–479. `doi:10.1007/978-3-319-46523-4_28`.

[3] M. Meimaris, G. Papastefanatos, N. Mamoulis, I. Anagnostopoulos, Extended characteristic sets: Graph indexing for sparql query optimization, in: IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 497–508. doi:10.1109/ICDE.2017.106.

[4] G. Montoya, H. Skaf-Molli, K. Hose, The odyssey approach for optimizing federated sparql queries, in: C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, J. Heflin (Eds.), The Semantic Web – ISWC 2017, 2017, pp. 471–489. doi:10.1007/978-3-319-68288-4_28.

[5] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, S. Sakr, Rdf data storage and query processing schemes: A survey, ACM Comput. Surv. 51 (4) (Sep. 2018). doi:10.1145/3177850.

[6] M. Meimaris, G. Papastefanatos, P. Vassiliadis, I. Anagnostopoulos, 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., 2016, pp. 281–292. doi:10.5441/002/edbt.2016.27.

[7] M. Meimaris, G. Papastefanatos, P. Vassiliadis, I. Anagnostopoulos, Computational methods and optimizations for containment and complementarity in web data cubes, Inf. Syst. 75 (2018) 56–74. doi:10.1016/j.is.2018.02.010.

[8] T. Neumann, G. Moerkotte, Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins, in: IEEE 27th International Conference on Data Engineering, 2011, pp. 984–994. doi:10.1109/ICDE.2011.5767868.

[9] A. Gubichev, T. Neumann, Exploiting the query structure for efficient join ordering in SPARQL queries, in: Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014, 2014, pp. 439–450. doi:10.5441/002/edbt.2014.40.

[10] M. Meimaris, G. Papastefanatos, P. Vassiliadis, Hierarchical property set merging for SPARQL query optimization, in: 22nd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT 2020 Joint Conference, 2020, pp. 36–45.

[11] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of RDF data, VLDB J. 19 (1) (2010) 91–113. doi:10.1007/s00778-009-0165-y.

[12] C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for semantic web data management, Proc. VLDB Endow. 1 (1) (2008) 1008–1019. doi:10.14778/1453856.1453965.

[13] O. Erling, I. Mikhailov, Virtuoso: RDF Support in a Native RDBMS, Springer Berlin Heidelberg, 2010, pp. 501–519. doi:10.1007/978-3-642-04329-1_21.

[14] K. Wilkinson, Jena property table implementation, in: 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems, Athens, Georgia, USA, 2006, pp. 35–46.

[15] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, B. Bhattacharjee, Building an efficient rdf store over a relational database, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, p. 121–132. doi:10.1145/2463676.2463718.

[16] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, p. 411–422.

[17] M. Janik, K. Kochut, Brahms: A workbench rdf store and high performance memory system for semantic association discovery, in: The Semantic Web – ISWC, 2005, pp. 431–445.

[18] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, S. Manegold, Column-store support for rdf data management: Not all swans are white, Proc. VLDB Endow. 1 (2) (2008) 1553–1563. doi:10.14778/1454159.1454227.

[19] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, Triplebit: A fast and compact system for large scale rdf data, Proc. VLDB Endow. 6 (7) (2013) 517–528. doi:10.14778/2536349.2536352.

[20] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, G. Lausen, Sempala: Interactive sparql query processing on hadoop, in: Proceedings of the 13th International Semantic Web Conference, 2014, p. 164–179. doi:10.1007/978-3-319-11964-9_11.

[21] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, N. Koziris, H_2RDF+: An efficient data management system for big RDF graphs, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, p. 909–912. doi:10.1145/2588555.2594535.

[22] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, G. Lausen, S2RDF: RDF Querying with SPARQL on Spark, Proc. VLDB Endow. 9 (10) (2016) 804–815. doi:10.14778/2977797.2977806.

[23] M. Meimaris, G. Papastefanatos, Double chain-star: an RDF indexing scheme for fast processing of SPARQL joins, in: 19th International Conference on Extending Database Technology (EDBT), 2016, pp. 668–669. doi:10.5441/002/edbt.2016.78.

[24] M. Meimaris, G. Papastefanatos, Distance-based triple reordering for sparql query optimization, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 1559–1562. `doi:10.1109/ICDE.2017.227`.

[25] T. Mailis, Y. Kotidis, V. Nikolopoulos, E. Kharlamov, I. Horrocks, Y. E. Ioannidis, An efficient index for RDF query containment, in: 2019 International Conference on Management of Data, SIGMOD Conference, ACM, 2019, pp. 1499–1516. `doi:10.1145/3299869.3319864`.

[26] H. Gupta, V. Harinarayan, A. Rajaraman, J. D. Ullman, Index selection for OLAP, in: Proceedings 13th International Conference on Data Engineering, 1997, pp. 208–219. `doi:10.1109/ICDE.1997.581755`.

[27] Y. Guo, Z. Pan, J. Heflin, Lubm: A benchmark for owl knowledge base systems, Web Semantics: Science, Services and Agents on the World Wide Web 3 (2) (2005) 158–182.

[28] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, Diversified stress testing of rdf data management systems, in: The Semantic Web – ISWC 2014, pp. 197–212.