ABSTRACT

We motivate and describe techniques that allow to detect an “emergent” relational schema from RDF data. We show that on a wide variety of datasets, the found structure explains well over 90% of the RDF triples. Further, we also describe technical solutions to the semantic challenge to give short names that humans find logical to these emergent tables, columns and relationships between tables. Our techniques can be exploited in many ways, e.g., to improve the efficiency of SPARQL systems, or to use existing SQL-based applications on top of any RDF dataset using a RDBMS.

Categories and Subject Descriptors
H.3 [Information Storage and Retrieval]: Miscellaneous

Keywords
RDF, Structure recognition, Relational schema

1. INTRODUCTION

RDF is the data model for the Semantic Web and Linked Open Data. It represents data as collection of <subject, property, object> triples. By providing flexibility for users to represent and evolve data without the need for a prior schema – sometimes called the “schema last” approach – and identifying properties and (references to) subjects uniformly using URIs, RDF has been gaining ground as the standard for global data exchange and interoperability, recently through the popularization of micro-formats such as RDFa, which are increasingly used embedded in web pages. This creates a need for database technologies that can query large amounts of RDF efficiently with SPARQL or SQL.

SQL-speaking relational database systems (RDBMS’s) require to declare a schema upfront (“schema first”) and can only store and query data that conforms to this schema. RDF systems typically rely on a “triple-store” architecture, which store all data in a single table containing 0 (subject, property, object) columns. SQL systems tend to be more efficient than triple stores, because the latter need query plans with many self-joins – one per SPARQL triple pattern. Not only are these extra joins expensive, but because the complexity of query optimization is exponential in the amount of joins, SPARQL query optimization is much more complex than SQL query optimization. As a result, large SPARQL queries often execute with a suboptimal plan, to much performance detriment. RDBMS’s can further store data efficiently e.g. using advanced techniques such as column-wise compression, table partitioning, materialized views and multi-dimensional data clustering. These techniques require insight in the (tabular) structure of the dataset and have so far not been applicable to RDF stores.

Semantic Web technology has its roots in Artificial Intelligence and knowledge representation, and we think it is seldom realized that its notion of “schema” in the term “schema last” differs from the corresponding “schema” notion in “schema first” for relational technology. Semantic Web schemas – ontologies and vocabularies – are intended to allow diverse organizations to consistently denote certain concepts in a variety of contexts. In contrast, relational schemas describe the structure of one database (=dataset), designed without regard for reuse in other databases.

Our work shows that actual RDF datasets exhibit (i) a very partial use of ontology classes and (ii) subjects share triples with properties from classes defined in multiple ontologies. To illustrate, (i) in the crawled WebDataCommons there is information on less than a third of the ontology class properties in the actual triples, and (ii) we find in DBpedia that each subject combines information from more than eight different ontology classes on average. As such, when analyzing the actual structure of RDF datasets by observing which combinations of properties typically occur together with a common subject (called “Characteristic Sets” of properties [14]), any single ontology class tends to be a poor descriptor. Knowledge of the actual structure of a dataset is essential for RDBMS’s to be able to store and query data efficiently. Our work allows RDF stores to automatically discover this actual structure, which we call the emergent relational schema. The emergent relational schema enables to internally store RDF data more like relational tables, allowing SPARQL query execution to use less self-joins, which also reduces the complexity of query opti-

1With “triple-store” we mean RDF or graph stores that use any data structure, be it a graph edge-list, B-tree, hash map, etc. that stores individual triples (or quads), or graph edges without exploiting their connection structure.
We discover all occurring CS’s and detect something akin to a UML class diagram by analyzing them.

2. EMERGING A RELATIONAL SCHEMA

We assign class, attribute and relationship labels (human-understandable names) to the recognized CS’s using multiple methods.

3. CS Merging. We merge CS’s that are semantically or structurally similar to each other, with the purpose of making the schema more compact. We re-run Steps 2 and 3 iteratively in order to automatically tune the similarity threshold parameter \( \tau_{\text{sim}} \) to the nature of the dataset.

4. Schema Filtering. We filter low frequency CS’s, but make sure to conserve highly referenced CS’s (akin to relational “dimension tables”). As reference relationships can be indirect (via via) we use a PageRank-like algorithm to count how often referenced each CS is. We also filter out CS properties that are too sparsely populated.

5. Instance Filtering. We filter out instances (rows) to increase literal type homogeneity, and filter out individual triples to eliminate erroneously multi-valued attributes, and to improve foreign-key cardinality homogeneity.

The “class diagram” where each merged-CS that survived filtering is a class, is represented as a relational schema consisting of tables and foreign key relationships. Each class becomes one table, and its properties its columns, but relationships and multi-valued attributes lead to additional tables. Properties for which multiple literal types occur frequently, are represented by multiple table columns. The <10% triples that do not fit this schema remain stored in a separate SPO table. We now discuss the five steps in detail.

2.1 Basic CS Discovery

Given an RDF dataset \( R \), the Characteristic Set of a subject \( s \) is defined as \( \text{cs}(s) = \{ p | \exists o : (s, p, o) \in R \} \) [14].

We first identify the basic set of CS’s by analyzing all triples stored in an RDF table in SPO order. Such table is produced by a standard bulk load employed by triple stores. While loading the triples into this representation, the URIs get encoded in a dictionary, such that columns \( S, P, O \) are not URI strings, but integers called object identifiers (OIDs) pointing into this dictionary. This is a standard technique. These integer OIDs form a dense domain starting at 0.

We now make a single pass through the SPO table and fill a hash map where the key is the set of OIDs of properties that co-occur for each subject. Note that due to the SPO ordering these are easily found as the \( P \)’s of consecutive SPO triples with equal \( S \). The key of the hash map is the offset in the SPO table where the CS first occurs. Its hash is computed by XOR-ing the hashes of all \( P \)’s (which are OIDs). The insert-order in the hash table (starting at 1) provides us with a dense numeric OID for each CS. Further, we remember in an array indexed by \( S \) which stores such CS-OIDs, to which CS each subject belongs (this array is part of the URI dictionary). Note that not all URIs in the dictionary may occur as a subject in the SPO table, for which case this array is initialized with zeros. After making the single pass over the SPO table, we will have all occurring basic CS’s in the hash map, and we also keep an occurrence count there.

Further we make a second pass over the SPO table, where we look at type information. For each triple with a literal object, we maintain a histogram of literal-type occurrences per property in a second hash map with key \( [P, \text{type}] \) and a count value. For each triple that is a non-literal, on the other hand, we look up to which CS its subject belongs (arcCS) and to which its object (ditCS) – this can be done efficiently using the array mentioned before. If there is a
(s1, offers, offer1)
(s1, region, region1)
(s2, offers, offer2)
(s2, offers, offer3)
(s2, region, region1)
...
<Example RDF triples>
<Basic explored CS’s>

Figure 2: Example of basic CS’s and their relationships
dstCS, we maintain another histogram stored in a third hash map with as key \{srcCS,P,dstCS\} and a count value. This histogram records how often basic-CS’s refer to each other and over which property (relationship statistics).

These algorithms are all simple and obviously linear in average-case complexity, therefore we omit a listing or further analysis. Figure 2 shows an example of the found basic CS and their relationships after the exploration process.

Diversity of the basic CS’s. Table 1 shows statistics on the basic CS’s and their properties for the synthetic RDF benchmark datasets LUBM\(^3\), SP2B\(^3\), and BSBM\(^4\), the originally relational datasets converted to RDF MusicBrainz\(^5\), EuroStat\(^6\), and DBLP\(^7\). PubMed\(^8\) and the native RDF datasets WebDataCommons\(^9\) ("WebData.") and DBpedia\(^10\). The number of basic CS’s can vary significantly regardless the number of input triples. If one would naively propose to store RDF data using a separate relational table for each CS and their relationships after the exploration process.

Data coverage by basic CS’s. Figure 1 shows the frequencies and the cumulative number of triples covered by the basic CS’s sorted by their frequencies, for one of each kind of dataset (synthetic, relational, native). In this figure, the number of CS’s needed for covering a large portion of the triples (e.g., 90\%) can be significantly different between the datasets. We show for reference in Table 1, that 90\% of the synthetic benchmark datasets can be covered by using a small number of CS’s (e.g., 7 for SP2Bench). Many Linked Open Data datasets originate from existing sources whose data is kept in relational databases. We see that in such datasets a few CS’s can cover almost all data. However, for complex datasets originally created as RDF (native), in order to cover 90\% of the triples, many CS’s are needed, in case of DBpedia more than 85,000.

2.2 CS Labeling
When presenting humans with a relational schema, short labels should be used as aliases for machine-readable and unique URIs for naming tables, columns and relationships between tables. For assigning labels to CS’s, we exploit semantic information (ontologies) as well as structural information. Because not all ontologies follow the same structure, we now see that a complex RDF dataset like DBpedia would lead to an unacceptable number of small tables. As we can also see that while most of the datasets have a single literal type for each CS property, DBpedia and WebDataCommons have many properties with more than one literal type in its object values (i.e., multi-type properties), so native datasets appear to be both complex and “dirty”.

Table 1: Statistics on basic CS’s.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>#triples(^*)</th>
<th>#CS’s</th>
<th>#CS’s to cover 90%</th>
<th>Avg. #prop.</th>
<th>#multi-type properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM</td>
<td>100M</td>
<td>17</td>
<td>7</td>
<td>5.71</td>
<td>0</td>
</tr>
<tr>
<td>BSBM</td>
<td>100M</td>
<td>49</td>
<td>14</td>
<td>12.61</td>
<td>0</td>
</tr>
<tr>
<td>SP2Bench</td>
<td>100M</td>
<td>554</td>
<td>7</td>
<td>9.8</td>
<td>0</td>
</tr>
<tr>
<td>synthetic</td>
<td>data created by benchmark data generator</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MusicBrainz</td>
<td>170M</td>
<td>27</td>
<td>10</td>
<td>4.7</td>
<td>0</td>
</tr>
<tr>
<td>EuroStat</td>
<td>70K</td>
<td>44</td>
<td>8</td>
<td>7.77</td>
<td>0</td>
</tr>
<tr>
<td>DBLP</td>
<td>50M</td>
<td>249</td>
<td>8</td>
<td>13.70</td>
<td>0</td>
</tr>
<tr>
<td>PubMed</td>
<td>1.82B</td>
<td>3340</td>
<td>35</td>
<td>19.27</td>
<td>0</td>
</tr>
<tr>
<td>relational</td>
<td>RDF data from a relational database dump</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WebData.</td>
<td>90M</td>
<td>13354</td>
<td>930</td>
<td>7.94</td>
<td>551</td>
</tr>
<tr>
<td>DBpedia</td>
<td>404M</td>
<td>439629</td>
<td>85922</td>
<td>24.36</td>
<td>1507</td>
</tr>
<tr>
<td>native</td>
<td>real data originating as RDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^*\): Number of triples after removing all duplicates
## Ontologies in DBpedia dataset

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Number of Classes Used</th>
<th>Percentage of Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia-owl</td>
<td>1.94</td>
<td>37%</td>
</tr>
<tr>
<td>BSBM</td>
<td>3.96</td>
<td>3%</td>
</tr>
<tr>
<td>SP2Bench</td>
<td>4.94</td>
<td>4%</td>
</tr>
<tr>
<td>MusicBrainz</td>
<td>3.93</td>
<td>1%</td>
</tr>
<tr>
<td>EuroStat</td>
<td>3.14</td>
<td>84%</td>
</tr>
<tr>
<td>DBLP</td>
<td>6.58</td>
<td>8%</td>
</tr>
<tr>
<td>PubMed</td>
<td>4.94</td>
<td>-</td>
</tr>
<tr>
<td>WebData</td>
<td>2.27</td>
<td>33%</td>
</tr>
<tr>
<td>DBpedia</td>
<td>8.35</td>
<td>5%</td>
</tr>
</tbody>
</table>

### Discriminative Properties

- **Type properties.** Certain specific properties explicitly specify the concept a subject belongs to. The most common RDF property with this role is `rdfs:type`, where the 0 of triples with this property may be the URI identifying an ontology class. Recall that our first step is to find a good UML-like class diagram for the RDF dataset, where a CS roughly corresponds to an UML class, and specifically here we are trying to find a human-friendly short name (label) for each CS. Even though we stated above that any individual ontology class is a poor descriptor of the structure of the data. Our emergent relational schema, aims to provide a better description.

### Type properties

- **Discriminative Properties.** Even if no type property is present in the CS, we can still try to match a CS to an ontology class. We compare the property set of the CS with the property sets of ontology classes using the TF-IDF similarity score [18]. This method relies on identifying “discriminative” properties, that appear in few ontology classes only, and whose occurrence in triple data thus gives a strong hint for the membership of a specific class. An example is shown in Figure 5. In this example, as `cs4` and the class `PriceSpecification` of the GoodRelations ontology share discriminative properties like `gor:hasUnitOfMeasurement` and `gor:valueAddedTaxIncluded`. `PriceSpecification` can be used as the label of `cs4`. Detailed computation of the TF-IDF-based similarity score between a CS and an ontology class can be found in [17]. An ontology class is considered to be matching with a CS if its similarity score exceeds the sim-

---

**Figure 3:** Partial & mixed ontology class usage in CS's (table-left), Ontologies used in native RDF datasets (graphs-right)

**Figure 4:** Choosing a CS label from explicit RDF type annotations that refer to ontology classes in a hierarchy.

**Figure 5:** Example CS vs. Ontology class annotations to be used for finding the CS class label. For the rest, we count (i) how many subjects in the CS have it, and (ii) how many subjects in the whole dataset have it. Similar to TF-IDF [18], dividing (i) by (ii) provides a reasonable ranking to choose an appropriate ontology class. Finally, if the ontology class has label information (and this information is typically available), we then use it as the label for the CS.

We should, however, in this ranking also take into account the class hierarchy information that an ontology provides. Thus, we account for missing superclass annotations by inferring them for the purpose of this ranking. In Figure 4, if a triple in some CS has `rdf:type Company`, but not `Organization` or `Thing` explicitly, we still include these annotations in the ranking calculation.

In this example, “RadioStation” is chosen as its coverage of the subjects in the CS is above $\tau_{nf}$ ($97 > 5$) and its ranking score ($97/0.2=485$) is the highest.

---

[purl.org/goodrelations/]
The ontology class correspondence of a CS, if found, is also used to find labels for properties of the CS (both for relationships and literal properties).

**Relationships between CS’s.** If the previous approaches do not apply, we can look at which other CS’s refer to a CS, and then use the URI of the referring property to derive a label. For example, a CS that is referred as Address indicates that this CS represents instances of an Address class. We use the most frequent relationship to provide a CS label. For instance, in WebDataCommons 93532 instances refer to a CS via property address and only 3 via property locatedAt. Thus, Address is chosen as the label.

**URI shortening.** If the above solutions cannot provide us a link to ontology information for providing attribute and relationship labels, we resort to a practical fall-back, using a link to ontology information for providing attribute locatedAt. For instance, in WebDataCommons 93532 instances covered by that ontology (Equation 1).

![Figure 6: CS’s with assigned labels](image)

Figure 6: CS’s with assigned labels

Figure 7 shows an example of merging CS's (the merging process is iterative). As shown in the figure, the number of NOT-NULL cells in table t3j is equal to the total number of NOT-NULL cells of the tables ti and tj, however, the number of NULL cells increases due to properties not in the intersection of the two CS's becoming padded with NULLs in the merged CS.

**Semantic merging.** We can merge two CS’s on semantic grounds when both CS class labels that we found were based on ontology information. Obviously, two CS’s whose labels were taken from the same ontology class URI represent the same concept, and thus can be merged. If the labels stem from different ontology classes we can examine the class hierarchy and identify the common concept/class shared by both CS’s, if any, and then justify whether these CS’s are similar based on the “generality” of the concept. Here the “generality” score of a concept is computed by the percentage of instances covered by it and its subclasses among all the instances covered by that ontology (Equation 1).

![Figure 7: Example of merging CS’s](image)

Figure 7: Example of merging CS’s

2.3 CS Merging

After basic exploration, there may be thousands of CS’s, in case of DBpedia even 100,000. This means the individual CS’s have only a few subjects (=rows, in relational terms) in them, so that storing them in a relational table would incur overheads (e.g. tables not filling a disk page, large database catalog, expensive metadata lookup). Further, many of these basic CS’s are very similar to each other (differing only in a few properties) and denote the same concept. When querying for that concept, one would have to formulate a UNION of many tables, which is cumbersome and also slows down queries. Finally, a relational schema with thousands of tables is just very hard to understand for humans. Therefore, the next step is to reduce the number of tables in the emergent relational schema by merging CS’s, using either semantic or structural information.

Rule 1. If an ontology class URI exists equal to the labels of both cs1 and cs2, then merge cs1 and cs2. (S1)

Rule 2. If there exists an ontology class O, being an ancestor of the labels of cs1 and cs2, and \( g_{score}(O) \) is less than \( \frac{1}{n_{max}} \), then merge cs1 and cs2. (S2)

In S2, \( \frac{1}{n_{max}} \) is used as the threshold for the generality score based on Urbd, the upper bound for the number of tables in the schema – which is one of the only three parameters of emergent relational schemas, see Table 2.

Figure 8 demonstrates the modifications to the explored CS’s of the example dataset and their relationships when sequentially applying merging rules S1 and S2. Here, since cs2
and cs9 both derived their label Offering from the Offering class of the GoodRelation ontology, according to S1, they are merged into a new CS (cs13). The references from/to cs9 and cs3 are also updated for cs13. Besides, since the labels of cs8 and cs4 have Broadcaster as their non-general common ancestor in the ontology hierarchy, they are merged into cs12 according to S2. The label of cs12 is assigned by using the name of the common ancestor ontology class. The full description about updating the label of a CS after merging can be found in [17].

Structural merging. While semantic merging is a relatively safe bet, it may not always be applicable or effective enough to reduce the amount of merged CS’s. Therefore, we also look at the structure of the CS’s and their relationships to see if these can be merged. The idea here is to identify CS’s that denote the same concept based on so-called “discriminative” properties, which are those with a high TF/IDF score (see the previous section). If the overlap between two CS’s contains enough “discriminative” properties, we can merge them.

Figure 9 shows an example where the overlapping properties of cs7 and cs10 indicate that these CS’s both originate from the “Location” entity. Here, the property rdfd:name is not discriminative since it appears in most of the CS’s. However, properties rdfd:street-address and rdfd:region give evidence that both represent a “Location”.

Equations 2 and 3 formally show the detailed computations for the TF/IDF score of each property in a cs and the cosine similarity score \( \text{sim}_{ij} \) between two cs’s (cs and csj), respectively. In these functions total\#CSs is the total number of CS’s, \( \#\text{containedCSs}(p) \) is the number of CS’s having property \( p \) in their property list.

\[
\text{tfidf}(p, cs) = \frac{1}{[D_p(cs)]} \times \log \frac{\text{total}\#CSs + 1}{\#\text{containedCSs}(p)} (2)
\]

\[
\text{sim}_{ij} = \frac{\sum_{p \in (cs_j)\cap(cs_j)} \text{tfidf}(p, cs_i) \times \text{tfidf}(p, cs_j)}{\sqrt{\sum_{p \in cs_i} \text{tfidf}(p_i, cs_i)^2} \times \sqrt{\sum_{p \in cs_j} \text{tfidf}(p_j, cs_j)^2}} (3)
\]

In summary, two CS’s are considered structurally similar if they are both referred from the same CS via the same property (rule S3) or their property sets have a high TF/IDF similarity score (rule S4). In Rule 3, \( \text{ref}(cs, p, cs_j) \) is the number of references from cs to csj via property \( p \), and \( \tau_{inf} \) is the infrequent threshold which is used to prevent non-frequent references from being considered in applying the rule S3. In Rule 4, \( \tau_{sim} \) is the similarity threshold above which we decide to merge two CS’s.

Rule 3. If cs and csj exist with \( \frac{\text{ref}(cs, p, cs_j)}{\text{freq}(cs)} \) and \( \frac{\text{ref}(cs, p, cs_j)}{\text{freq}(cs)} \) greater than \( \tau_{inf} \) then merge cs and csj. (S3)

Rule 4. If the similarity score \( \text{sim}_{ij} \) between cs and csj is greater than \( \tau_{sim} \) then merge cs and csj. (S4)

Figure 10 shows the updates to the CS’s and their relationships when we continue applying the rules S3 and S4. In this figure, cs5 and cs11 are merged according to the rule S3 as they are both referred by cs1 via the property offers. Besides, since cs7 and cs10 have high similarity score (as shown in Figure 10), they are merged into cs12.

We experimentally observed that the best order of applying the rules for merging CS’s is S1, S3, S2, S4. Further details can be found in [17].

2.4 Schema Filtering

Our goal is to represent a large portion of the input triples in a compact, human-friendly, relational schema. After CS merging, most of these merged classes cover a large amount of triples. However, it may happen that some classes still cover a limited number of RDF subjects, so if the merged CS covers < minj (e.g. 1000, see Table 2) subjects, it is removed from the schema; and we limit the UML class diagram to the merged Ubdm CS’s with highest frequency. Note that omitting CS’s with low frequency will only marginally reduce overall coverage.

Preserving Dimension Tables. However, for this removal of classes (merged CS’s) we make one exception, namely

At this stage, we also refer surviving merged CS’s as classes, similar to UML classes.
we conserve CS’s that – although small in terms of covered subjects – are referred to many times from other tables. The rationale is that such CS’s thanks to the large amount of incoming references represent important information of the dataset that should be part of the schema. This is similar to a dimension table in a relational data warehouse, which may be small itself, but is referred to by many millions of tuples in large fact tables over a foreign key. Thus, combining the information of basic CS detection and relationship detection, we preserve CS’s with a high frequency of incoming references. However, detecting dimension tables should not be handled just based on the number of direct relationship references. The relational analogy here are snowflake schemas, where a finer-grained dimension table like COUNTRY refers to an even smaller coarse-grained dimension table COUNTRY. To find the transitive relationships and their relative importance, we use the PageRank [16] algorithm on the graph formed by all CS’s (vertexes) and relationships (edges, regardless of direction). In each iteration, the score of a merged CS is computed based on the references from other merged CS’s and their scores computed in the previous iteration. Equation 4 shows the formula for each iteration:

\[ IR_k(cs_i) = \sum_{cs_j \to cs_k} IR_{k-1}(cs_j) \times \frac{ref(cs_j, cs_k)}{refSTo(cs_k)} \times \frac{freq(cs_j)}{freq(cs_k)} + refSTo(cs_k) \]

If \( IR_k(cs_i) \geq Ub_{hid} \) \( \rightarrow \) \( cs_i \) is a dimension CS.

The merged CS’s having a score higher than a threshold \( Ub_{hid} \) will be selected for inclusion in the schema. in which \( IR_k(cs_i) \) is the indirect-referenced score of \( cs_i \) after \( k \) iterations, \( ref(cs_j, cs_k) \) is the number of references from \( cs_j \) to \( cs_k \), \( freq(cs_j) \) is the frequency of \( cs_j \), and \( refSTo(cs_k) \) is the total number of direct references to \( cs_k \).

Specifically, the number of iterations \( k \) is set the same as the diameter of the CS graph. It is because, with that value, after \( k \) iterations, the \( IR_k \) score of each CS will get computed from all the CS’s. To compute the diameter of the graph, we implemented a fast and simple algorithm described by [3].

Minimizing the number of infrequent properties. A final step of schema filtering considers eliminating CS properties, which as column in a relational table would have many NULL values. If the property coverage ratio (see Equation 5) is less than the infrequent threshold \( \tau_{inf} \), that property is infrequent and it gets removed from the CS.

\[ coverageRatio(p, cs) = \frac{freq(p, cs)}{freq(cs)} \]

2.5 Instance Filtering

The output after labeling, merging, and schema filtering is a compact relational emergent schema. In the instance filtering phase, all RDF triples are visited again, and either stored in relational tables (typically > 90% of the triples, which we consider regular), or (the remainder) separately in a PSO table. Hence, our final result is a set of relational tables with foreign keys between them, and a single triple table in PSO format. In principle, the regular triples are those belonging to a merged CS (that survived schema filtering). However, not all such triples are considered regular in the end, as we perform three types of instance filtering, described next.

Maximizing type homogeneity. Literal object values corresponding to each property in a CS can have several different types e.g., number, string, dateTime. The relational model can only store a single type in each column, so in case of type diversity, a relational system like MonetDB must use multiple columns for a single property. They contain the type-cast value of the literal, if possible, and NULL otherwise. The number of columns needed for representing the data from a \( cs_i \) hence is \( \sum_{p \in cs_i} \#o(Types(p)) \). This number can be large just due to a few triples having the wrong type (dirty data). To minimize the number of such columns, for each property, we filter out all the infrequent literal types (types that appear in \(< \tau_{inf} \) percent of all instances). All triples of class instances with infrequent types are moved to the PSO table.

Relationship Filtering. We further filter out infrequent or “dirty” relationships between classes. A relationship between \( cs_i \) and \( cs_j \) is infrequent if the number of references from \( cs_i \) to \( cs_j \) is much smaller than the frequency of \( cs_i \) (i.e., less than \( \tau_{inf} \) percent of the \( CS \)’s frequency). A relationship is considered dirty if the majority but not all the object values of the referring class (e.g., \( cs_i \)) refer to the instances of the referred class (\( cs_j \)). In the former case, we simply remove the relationship information between two classes. In the latter case, the triples in \( cs_i \) that do not refer to \( cs_j \) will be filtered out (placed in the separate PSO table).

We note that in the general case of \( n-m \) cardinality relationships, the relational model requires to create a separate mapping table that holds just the keys of both relations. However, in case one of the sides is 0...1, this is generally avoided by attaching a FK column to the table representing the other side. We try to optimize for this, by observing whether a multi-valued relationship is infrequent (\(< \tau_{inf} \)). If so, we remove the excess relationship to the separate PSO table, such that all remaining subjects in the class have maximally one relationship destination. Finally, if almost all instances of one class have exactly one match in the other class but a few (\(< \tau_{inf} \)) have none, we move all triples with that subject to the separate PSO table to preserve the exact \( n-1 \) cardinality (which keeps the FK column non-NULLable).

Multi-valued attributes. The same subject may have 0, 1 or even multiple triples with the same property, which in our schema leads to an attribute with cardinality > 1. While this is allowed in UML class diagrams, direct storage of such values is not possible in relational databases. Practitioners handle this by creating a separate table that contains the primary key (subject OID) and the value (which given literal type diversity may be multiple columns). The RDF bulk-loader of MonetDB does this, but only creates such separate storage if really necessary. That is, we analyze the mean number of object values (\( meanp \)) per property. If the \( meanp \) of a property \( p \) is not much greater than 1 (e.g., less than \((1+\tau_{inf}/100)\)), we consider \( p \) as a single-valued property and only keep the first value of that property in each tuple while moving all the triples with other object values of this property to the non-structured part of the RDF dataset. Otherwise, we will add a table for storing all the object values of each multi-valued property.

\[ meanp(p) = \sum p(k) \times k \]

where \( p(k) = \frac{\#times p has k object values}{freq(p)} \)

(6)
### 2.6 Parameter Tuning

An important question that we needed to address is how the various parameters guiding the recognition process should be set. Choosing improper parameters might result in a “bad” final schema with, e.g., small data coverage, lots of NULLs, etc. Further, since each input dataset can have different characteristics, it would be unfeasible to find a fixed parameter set that works optimally for all datasets.

The most dataset sensitive parameter we found to be the $\tau_{sim}$, used in labeling while matching ontologies using discriminative properties, as well as in the CS merging Rule 4 that determines up until which point merging should continue. It is a control on the strictness of finding equivalences between structures and ontologies, at 1 it is very strict while at 0 it is very lax. We evaluate the quality of the relational schema on two dimensions, namely (i) the number of tables (compactness of the schema) and (ii) its precision, which is the number of NOT-NULL cells, $fill(t)$, divided by the total number of cells, $cap(t)$, in all tables, as in Equation 7. There is a clear trade-off between having a compact schema and higher precision, depending on $\tau_{sim}$.

Our auto-tuned algorithm iteratively re-runs the labeling and merging steps with different values of $\tau_{sim}$. In each run, we measure the number of tables and the precision; we also compute a delta of these between successive values of $\tau_{sim}$. In Equation 8, $k$ is the total number of runs; $nT_i$ ($nT_{nom_i}$) and $pr\Delta_i$ ($prNom_i$) are the (normalized) number of tables and the schema precision of the $i^{th}$ run; $nT_{delta}$ and $pr\Delta_{nom}$ are the relative change in the normalized number of tables and the precision at the $i^{th}$ run, respectively. We use the lowest value of $\tau_{sim} > 0$ where $nT_{delta} > pr\Delta_{nom}$.

$$\tau_{sim} = \frac{1}{\sum_i \frac{fill(t)}{cap(t)}}$$

$$nT_{nom_i} = \frac{nT_i - nT_{ref}}{nT_i - nT_{ref}} \quad pr\Delta_{nom_i} = \frac{pr_{nom_i} - pr_{nom_1}}{pr_{nom_1} - pr_{nom_1}}$$

$$nT_{delta} = nT_{nom_i} - nT_{nom_{i-1}} \quad pr\Delta_{nom} = pr\Delta_{nom}$$

The left of Figure 11 shows normalized $nT_i$ and $pr\Delta_i$ for WebData Commons as a function of $\tau_{sim}$ in steps of 0.05, while the right side shows the deltas between steps. Auto-tuning chooses the cross-over point of the deltas ($\tau_{sim}=0.7$).

### 3. EXPERIMENTAL EVALUATION

**Metrics.** We propose several metrics for evaluating the quality of the emergent schema. These metrics rely on the fact that a structure is considered to be good if it is compact (few and thin tables), precise (few NULLs) and has large coverage (few triples that have to be moved to separate PSO storage). Given an RDF dataset $R$ and its total number of triples $|R|$, the first performance metric, $C$, is the percentage of input triples covered by the schema:

$$C = \frac{\sum_i cov(t_i)}{|R|}$$

### 3.1 Experimental Results

**Labeling evaluation.** We presented the emergent schemas of the DBpedia and WebDataCommons datasets to 19 humans and asked them to rate the labels. On a 5-point Likert scale from 1 (bad) to 5 (excellent) label quality, the top 3 labels of each table were scored by at least 3 persons. As shown in Table 3, the top 3 label candidates received an average rating of 3.6 for WebDataCommons and 3.8 for the DBpedia dataset. The finally chosen labels (one among the top 3) got better scores (4.1 and 4.6, respectively). We therefore conclude that the ordering of label candidates created by our algorithms produces encouraging results, as the chosen labels get higher ratings than the other candidates. Furthermore, our evaluation shows that 78% (WebDataCommons) and 90% (DBpedia) of the labels are rated with 4 points or better, hence are considered “good” labels by the users. The emergent relational schemas for the nine datasets we tested are too large to include in this paper, Figure 12 shows EuroStat, one of the simpler schemas.\(^{13}\)

**Merging/Filtering performance.** Figure 13 and Table 4 show the performance of the proposed merging algorithms and the filtering techniques for detecting a compact relational emergent schema with high coverage. According to

\(^{13}\)See w.w.cui.nl/~boncz/emergent for the other datasets.
Figure 12: Final emergent schema for EuroStat – the lighter a column, the more NULLs (percentage in parentheses).

Figure 13: Schema quality $Q$ during merging & filtering

Figure 14: Building time & database size for single triple table (SPO) and reorganized relational tables (CS-based) (normalized by bulk-load time and database size for all six S,P,O table permutations (ALL)).

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Number of tables</th>
<th>Coverage – Metric C (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM</td>
<td>before merging</td>
<td>after merging</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>BSBM</td>
<td>49</td>
<td>8</td>
</tr>
<tr>
<td>SP2B</td>
<td>554</td>
<td>13</td>
</tr>
<tr>
<td>EuroStat</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>BSBM</td>
<td>44</td>
<td>10</td>
</tr>
<tr>
<td>DLBP</td>
<td>249</td>
<td>9</td>
</tr>
<tr>
<td>PubMed</td>
<td>3340</td>
<td>14</td>
</tr>
<tr>
<td>WebData</td>
<td>13554</td>
<td>3000</td>
</tr>
<tr>
<td>DBpedia</td>
<td>439629</td>
<td>542</td>
</tr>
</tbody>
</table>

Table 4: #tables and metric C after merging & filtering

bulking loading time for building a single SPO table as well as building all the six permutations of S, P, O (marked [ALL]). Thus, MonetDB integrates emergent schema detection into its RDF bulk-loading without recognizable delay.

**Compression.** Figure 14 shows that the database size stored using relational tables can be two times smaller than the database size of a single SPO triple table. The reason is that in the relational representation the S and P columns effectively get compressed away, only the O columns remain.

**Query processing.** As a proof that the recognized emergent schema can be easily integrated and boost the performance of existing RDF stores, we report on the effort at OpenLink to integrate emergent relational schema technology in one of the state-of-the-art RDF stores, Virtuoso [8]\(^4\) It was a few months work to integrate Characteristic Set based storage, query execution and query optimization in Virtuoso. We compare a classic Virtuoso RDF quad table (Virt-Quad) and this CS-based implementation (Virt-CS) on the BSBM benchmark at 10 billion triples scale.

The experimental results in Table 5 show that exploiting the emergent relational schema even in this very preliminary implementation already improves the performance of Virtuoso on a number of BSBM Explore queries by up to a factor of 5.8 (Q3, Hot run). Note that the Cold run is much slower comparing to the Hot run as most of the time goes in the statistics gathering, not in the execution. We see less gain from CS's in other queries, e.g., Q5, since the first condition on the BSBM products (on a range of numeric property) is selective, so the other columns of the CS (or self-joins to

\(^4\)https://github.com/v7fasttrack/virtuoso-opensource
RDF quads) are done on a small fraction of the subjects of the first range check. In Q3 more single-valued properties are accessed per subject, resulting in much more gain.

By collapsing multiple triple patterns into a single abstract CS table, query optimization gets a plan search space of the same order as for the equivalent SQL. For Q3, the compilation time drops from 40.2 msecs to 5.4 msecs when using the recognized schema. In many RDF applications, e.g. Open PHACTS, query optimization time dominates and can run into the tens of seconds. Due to the extreme search space resulting from triple patterns, there are often ad hoc restrictions on plans, e.g. no hash join or no joins on hash build sides. With CS, a more thorough search of the plan space becomes again practicable and we expect qualitatively better plans to result.

4. RELATED WORK

We note that previous work has already proposed building relational-resembling RDF stores [22, 7, 12, 21, 4]. However, these proposals either demand the presence of an all-explaining ontology (which then gets remapped to relational tables), or ask the database system administrator to create and maintain “property tables” explicitly. Our approach, in contrast, does not require any form of explicit schema ingestion. Second, since these approaches just use the structure internally in the SPARQL engine to make things faster, they do not address the challenge of making the schema understandable to humans (compactness, finding short aliases). For the latter, a related line of work is creating summaries of the graph structure to aid query formulation [6], yet these do not focus on making RDF database systems faster, and typically require a cluster to compute, whereas our approach is cheap and can piggyback on RDF bulk-loading. Related to the automatic structure exploration from data is work on ontology mining [13] which discovers ontologies from unstructured text on the web. In our approach, we recognize the emergent structure in RDF data (e.g., mixing of ontologies), and do not change the semantics, and focus on providing a relational view of it.

Frequent itemset mining, which has been studied in many data mining papers [1, 5, 9], is equivalent to the basic CS recognition, originally proposed by [14]. We use this technique but go beyond that by finding a schema graph with cross-CS relationships, and we employ a host of techniques to make this schema graph compact and human-friendly (finding labels).

A recent study on the structure refinement for the RDF data, [2] proposed an integer linear programming (ILP)-based algorithm which allows an RDF dataset being partitioned into a number of “sorts” where each sort satisfies a predefined structured-ness fitting threshold. This approach, relying mainly on the similarity and correlation between the properties of sorts, may merge subjects describing unrelated entities but having many common properties into a single sort (as also shown in their experiment with DrugCompany and Susan). While our solution only merges related CS’s together by exploiting the discriminating properties and the availability of the semantics/ontologies information. Besides, no relationship exploration as well as labeling for the sorts are considered in this work, and thus, no relational schema is recognized.

Consulting external resources for entity labeling is suggested by [20] in the context of table data reconstruction as well as by [19] a study on labeling hierarchical clusters. The former study shows that column names and table names usually cannot be found in table data itself. To reconstruct HTML table data they therefore rely on an external database with hyponym information. The latter study also mentions that documents often do not contain self-descriptive terms. To overcome this, they suggest using “anchor texts” as an additional resource in their document labeling task. Anchor texts are pieces of text on and next to hyperlinks to a specific document. The relational equivalent of anchor texts are names of foreign key relationships. In our case we rely on property names that refer to other tables for name suggestions, supplemented by ontology information when present.

5. CONCLUSIONS

In this paper, we introduced the notion of – and demonstrated practical techniques for – discovering an emergent relational schema in RDF datasets, that recovers a compact and precise relational schema with high coverage and useful labels as alias for all machine-readable URIs (which it preserves). The functional benefit of an emergent relational schema for RDF datasets is both in giving users better understanding of the structure of an RDF dataset, while also allowing the often > 90% of regular triples to be queried from existing SQL applications, which still dominate the IT industry. Our MonetDB RDF bulk loader enables this. We think that this also provides impetus to make SQL more semantic, e.g. stimulating usage of URIs in SQL metadata.

The emergent relational schema can also be used under the cover of an SPARQL engine as a new storage approach, where the 90% regular triples are stored in tabular structures and the rest in SPO format. We think that the knowledge of an emergent schema gives SPARQL engines just what they need to close the performance gap with SQL systems. This we demonstrated in Virtuoso, with gains both in compression, query execution and query optimization. The tabular structure opens up many opportunities to improve physical access patterns using (partial) clustered indexes, zone maps, table partitioning and even database cracking [11].

Looking ahead, the prospect of people supporting SQL applications on top of RDF data raises many new questions. Users will desire to tweak a found emergent schema by hand, e.g. by manually improving some labels. We propose making a found emergent schema explicit using a vocabulary, and researching techniques to control schema evolution to preserve schema stability while the emergent schema adapts over time to changes in the underlying RDF datasets.

---

Table 5: Query time (msecs) w/wo the recognized schema

<table>
<thead>
<tr>
<th></th>
<th>Q2</th>
<th>Q3</th>
<th>Q5</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virt-Quad</td>
<td>11567</td>
<td>7</td>
<td>4.2</td>
<td>4210</td>
<td>53</td>
</tr>
<tr>
<td>Virt-CS</td>
<td>2485</td>
<td>6</td>
<td>3.5</td>
<td>2965</td>
<td>9</td>
</tr>
</tbody>
</table>

(Cold: First query runtime after re-starting the server; Hot: Run the query 3 times and get the last runtime; Opt.: Query optimization time)
6. REFERENCES