Graph Pattern based RDF Data Compression

Jeff Z. Pan[1], José Manuel Gómez Pérez[2], Yuan Ren[1], Honghan Wu[3,1], Haofen Wang[4], and Man Zhu[5]

[1] Department of Computing Science, University of Aberdeen, United Kingdom
[2] iSOCO, Spain
[3] Nanjing University of Information & Technology, China
[4] East China University of Science & Technology
[5] School of Computer Science, Southeast University, China

Abstract. The growing volume of RDF documents and their inter-linking raise a challenge on the storage and transferring of such documents. One solution to this problem is to reduce the size of RDF documents via compression. Existing approaches either apply well-known generic compression technologies but seldom exploit the graph structure of RDF documents. Or, they focus on minimized compact serialisations leaving the graph nature inexplicit, which leads obstacles for further applying higher level compression techniques. In this paper we propose graph pattern based technologies, which on the one hand can reduce the numbers of triples in RDF documents and on the other hand can serialise RDF graph in a data pattern based way, which can deal with syntactic redundancies which are not eliminable to existing techniques. Evaluation on real world datasets shows that our approach can substantially reduce the size of RDF documents by complementing the abilities of existing approaches. Furthermore, the evaluation results on rule mining operations show the potentials of the proposed serialisation format in supporting efficient data access.

1 Introduction

The digital universe is booming, especially in terms of the amount of metadata and user-generated data available. Studies like IDC's Digital Universe[6] estimate that the size of the digital universe turned 1Zb (1 trillion Gb) for the first time in 2010, reached 1.8Zb just one year later in 2011 and will go beyond 35Zb in 2020. Some interesting figures include that 70% of such data is user-generated through several channels like social networks, mobile devices, wikis and other content publication approaches. Even more interestingly, 75% of such data results from data transformation, copying, and merging while metadata is the fastest growing data category. This is also the trend in semantic data, where datasets are increasingly being dynamically and automatically published, e.g. by semantic sensor networks [3], and consumed, e.g. by silico experiments in the form of scientific workflows [2]. In these domains a large number of distributed data sources are considered as opposed to classic data integration scenarios. This means that the amount of data available is growing at an exponential rate but also that data is not statically stored in their datasets. Combined with the growing size of the overall Linked

[6] http://www.emc.com/leadership/digital-universe

Open Data cloud, with more than 30 billion triples, and of its individual datasets, with some of its hubs e.g. DBPedia exceeding 1,2 billion triples, the need of effective RDF data compression techniques is clear.

This raises serious data management challenges. Semantic data need to be compact and comprehensible, saving storage and communication bandwidth, while preserving the data integrity. Several approaches can be applied to achieve lossless RDF document compression. They can be categorised into either application-dependent or application-independent approaches: Application-dependent approachs include Michael Meier's rule-based RDF graph minimisation [10] and Reinhard et. al.'s appraoch [12]. They are usually semi-automatic, requiring human input. Application-independent approaches are more generic. First of all, universal file compression techniques [4], such as bzip2 [7] and LZMA [8], can be applied on RDF document. Such approaches alter the file structure of RDF documents and can significantly reduce file size. Alternative RDF serialisations, such as HDT serialisation [5], lean graphs [8] and K2-triples [1] can be used to reduce file size. Such techniques preserve the structured nature of RDF documents. Another approach is based on logical compression, such as the rule-based RDF compression [9], which can be used to substantially reduce the number of triples in an RDF document. Ontology redundancy elimination [6] can also be regarded as logical RDF compression in which the RDF documents are intepreted with OWL (Web Ontology Language [9]) semantics.

Despite the compression results achieved by existing works, they make little or no use of the graph structure of RDF datasets. For example, universal compression techniques usually exploit the statistical redundancy in a document and the document is treated as a series of ordered characters. However an RDF document is essentially a graph in which the ordering in which nodes and edges are presented is irrelevant to the semantics of the data. Even the few approaches that leverage this kind of information are constrained to simple and fixed graph structures. This makes them less effective when reducing the size of compressed file. For example, logical compression [9] compresses re-occuring star-shaped graph structures of varying center nodes in an RDF document with single triples:

Example 1. In an RDF document, if it contains the following triples, where both m and n are large numbers:

$$< s_1, p_1, o_1 >, \ldots, < s_1, p_n, o_n >,$$
$$\ldots$$
$$< s_m, p_1, o_1 >, \ldots, < s_m, p_n, o_n >$$

then it can be compressed with the following triples:

$$< s_1, p_1, o_1 >, \ldots, < s_m, p_1, o_1 > \quad (1)$$

And a rule $<?s, p_1, o_1 > \rightarrow <?s, p_2, o_2 >, \ldots, <?s, p_n, o_n >$, where $?s$ is a variable, can be applied to recover all the removed triples.

[7] http://www.bzip.org/
[8] http://www.7-zip.org/
[9] http://www.w3.org/TR/owl2-overview/

This approach works well when the document contains many different nodes sharing many same "neighbours". But it is not applicable when such graph structures are not observed, e.g. when $n = 1$. In fact, its results, the triples in (1) is an example of such a scenario. This is because the logical compression presented by Joshi et al. [9] is constrainted to graph patterns with only 1 variable. By extending to more generic graph structures, improvement of compression rate can be easily achieved. In fact, triples in (1) can be compressed by exploiting a graph pattern with 2 variables:

Example 2. Without lose of generality, we assume m is an even number. We can further compress triples in (1) with the following ones

$$< s_1, p_x, s_2 >, \ldots, < s_{m-1}, p_x, s_m >$$

where p_x is a fresh predicate introduced for this graph pattern. And we can use a rule $<?s, p_x, ?o > \rightarrow <?s, p_1, o_1 >, <?o, p_1, o_1 >$ to decompress the triples.

Apparently, the number of triples is halved and such a further compression is guaranteed applicable and better on any results obtained by Joshi et al.'s approach. This shows that logical compression with lower compression rate can be achieved when more syntactic and semantic information of RDF datasets are better exploited.

In addition to semantic redundancies, the ways in which RDF graphs are serialised as sequences of bytes can also introduce another type of redundancies, i.e. the syntactic one. Existing approaches including textual serialisation syntaxes, e.g. RDF/XML, and binary ones, e.g. HDT [5] only deal with syntactic redundancies in concrete graph structures (defined as intra-structural redundancy in section 3.2). Without the knowledge of graph patterns in RDF graphs, they are not able to make use of the common graph structure (defined as inter-structural redundancy in section 3.2) shared by many instances of one graph pattern.

In this work, we aim at exploiting the graph structure of RDF datasets as a valuable source of information in order to increase the data compression gain in both the semantic level and syntactic level. Main contributions of the paper include:

1. we develop application-independent graph pattern-based logical compression and serialisation technologies for RDF documents;
2. we implement a framework that combines different compression techologies, including logical compression and serialisation;
3. we show that implementations of our approach can complement the compression abilities of existing solutions in semantic ans syntactic levels. The potentials of efficient data access are also revealed in the evaluation.

The rest of the paper is organised as follows. Section 2 will introduce basic notions, such as RDF graphs and graph patterns. Section 3 presents techniques of graph pattern based approaches for removing both semantic and syntactic redundancies. Sections 4 and 5 present our implementation and evaluations respectively.

2 Preliminaries

Resource Description Framework (RDF) [7] is the most widely used data interchange format on the Semantic Web. It makes web content machine readable by introducing

annotations. Given a set of URI reference $\mathcal{R}$, a set of literals $\mathcal{L}$ and a set of blank nodes $\mathcal{B}$, an RDF statement is a triple <s, p, o> on $(\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{L} \cup \mathcal{B})$, where s, p, o are the subject, predicate and object of the triple, respectively. An RDF document is a set of triples.

With these notions, to facilitate RDF data compression, we define a graph as follows:

Definition 1. ***(Graph)*** *A labeled, directed multiple graph (graph for short) $G = \langle N, E, M, L \rangle$ is a four-tuple, where N is a set of URI references, blank nodes, variables and literals, E is the set of edges, $M : E \to N \times N$ maps an edge to an ordered pair of nodes, L is the labelling function that for each edge $e \in E$, its label $L(e)$ is a URI reference.*

It is apparent that every RDF document can be converted to a graph whose nodes are not variables, and vice versa. In the following we use the notions RDF document, RDF triple set and RDF graph interchangably. Given a set T of RDF triples, we use $G(T)$ to denote the graph of the triple set. Given a RDF graph G, we use $T(G)$ to denote the set of triples that G represents.

Definition 2. ***(Graph Operations)*** *A graph $\langle N_1, E_1, M_1, L_1 \rangle$ is a sub-graph of another graph $\langle N_2, E_2, M_2, L_2 \rangle$ IFF $N_1 \subseteq N_2$, $E_1 \subseteq E_2$, $\forall e \in E_1$, $M_1(e) = M_2(e)$ and $L_1(e) = L_2(e)$.*

Two graphs $\langle N_1, E_1, M_1, L_1 \rangle$ and $\langle N_2, E_2, M_2, L_2 \rangle$ have a union IFF $\forall e \in E_1 \cap E_2$, $M_1(e) = M_2(e)$, $L_1(e) = L_2(e)$. Their union is a graph $\langle N, E, M, L \rangle$ such that $N = N_1 \cup N_2$, $E = E_1 \cup E_2$, $\forall e \in E_1 \cup E_2$, $M(e) = M_1(e)$ or $M(e) = M_2(e)$ and $L(e) = L_1(e)$ or $L(e) = L_2(e)$.

Definition 3. ***(Graph Pattern)*** *A graph pattern is a graph in which some nodes represent variables.*

The none-variable nodes in a graph pattern are called constants of the graph pattern.

In this paper we are not concerned with the direction of triples in a graph pattern. For conciseness, we use <$?x, p, o$> to represent a triple with $?x$ as either the subject, or the object.

Definition 4. ***(Instance)*** *A substitution $\S = (v_1 \to v_2)$ replaces a vector of variables v_1 with a vector of URI references/literals/blank nodes v_2.*

A graph G is an instance of a graph pattern G', denoted by $G : G'$, IFF there exists a sustitution $\S$ such that $G'_{\S} = G$. Given an RDF graph $\mathcal{D}$, we use $I_{\mathcal{D}}(G')$ to denote the set of all sub-graphs of $\mathcal{D}$ that are instances of G'. Obviously, $G \in I_{\mathcal{D}}(G)$ since an empty substitution $(\emptyset \to \emptyset)$ exists. And $I_{\mathcal{D}}(G) = \{G\}$ when G contains no variable.

When the $\mathcal{D}$ is clear from context, we omit it in the notations.

An RDF graph can be considered as the union of instances of several graph patterns.

Definition 5. **(Rule)** *Let GP and GP' be two graph patterns, $GP \to GP'$ is a rule.*

Let $\mathcal{D}$ be an RDF document, results of applying $GP \to GP'$ on $\mathcal{D}$, denoted by $GP \to_{\mathcal{D}} GP'$, is another RDF document $\mathcal{D}' = \bigcup_{GP_{\S} \in I_{\mathcal{D}}(G)} GP'_{\S}$.

In other words, $\mathcal{D}'$ is the union of GP' instances with substitutions that are used by instances in $I_{\mathcal{D}}(GP)$.

With these notations, we invesitgated different graph pattern-based in for RDF and their relation to the compression problem.

3 Graph Pattern-based Approaches

In this section, we propose graph pattern-based approaches to deal with both semantic redundancies and syntactic redundancies in RDF data. They complement existing approaches by either generalising existing semantic compression techniques, i.e. rule based approaches, or extending serialisation approaches, e.g. RDF/XML or HDT, to deal with new type of syntactic redundancies.

3.1 Semantic Compression: Graph Pattern-based Logical Compression

As we mentioned in the previous section, an RDF graph can be expanded from a smaller graph with the help of rules, whose body and head are both RDF graph patterns. This essentially means that the instances of the bigger graph pattern can be replaced by smaller instances of the smaller graph pattern. Below is an example:

Example 3. In the DBpedia dataset, the following graph pattern has a large number of instances:

$$GP_1 :<?x, a, foaf : Person>, <?x, a, dbp : Person>$$

Such a graph pattern can be replaced by a smaller graph pattern. For example, we can use one type T to represent the two types in the above graph pattern, and replace the two triples with a single triple

$$GP_2 :<?x, a, T> .$$

In this example, GP_1 is compressed by GP_2. As a consequence, $I(GP_1)$ is compressed by $I(GP_2)$. This will reduce the number of triples in the original RDF document by 50%. Such a compression can be achieved by applying rule $GP_1 \rightarrow GP_2$ on the RDF document. Decompression is achieved by applying rule $GP_2 \rightarrow GP_1$ on the compressed data set.

During logical compression, the fact that GP_2 contains less triples than GP_1 is exploited to ensure that $I(GP_2)$ contains less triples than $I(GP_1)$. Such a reduction of triple number is the main focus in logical compression.

A Unified Model for Graph Pattern-based Logical Compression We generalise the above compression mechanism to support more variables with the following unified model:

Definition 6. (Graph Pattern-based Logical Compression of RDF) *Let $\mathcal{D}$ be a RDF document, its graph pattern-based logical compression consists of an RDF document $\mathcal{D}'$ and a rule set S, such that the following holds:*

$$\mathcal{D} = \bigcup_{GP_2 \to GP_1 \in S} (\mathcal{D}' \setminus I_{\mathcal{D}'}(GP_2)) \cup (GP_2 \to_{\mathcal{D}'} GP_1)$$

We call $\mathcal{D}$ the original RDF document, $\mathcal{D}'$ the compressed RDF document, S the decompression rule set. And for each $GP_2 \to GP_1 \in S$, we say that GP_1 is compressed by GP_2.

In this procedure, triples in $I_{\mathcal{D}}(GP_1)$ are replaced by triples in $I_{\mathcal{D}'}(GP_2)$. Note that all triples in $\mathcal{D}'$ that involve new resources introduced in GP_2 but not GP_1 are removed during decompression.

It's worth mentioning that variables in the decompression rules bind only to explicitly named entities in the RDF document. Hence the compression results $\mathcal{D}'$ can directly be used in RDF reasoning and SPARQL query answering, e.g. by reasoners supporting DL-safe rules [11].

In the above definition,for each GP compressed by GP', the original $|GP|$ triples in $\mathcal{D}$ are replaced by $|GP'|$ new triples in $\mathcal{D}'$. The extra cost of compression is the maintenance of the rule, which consists of $|GP| + |GP'|$ triples. To characterise the effect of compression we define the following notions:

Definition 7. (Compression Quantification) *For GP compressed with GP' using the graph pattern-based compression defined in Def. 6, let T_O, T_C and T_R be the total number of different triples in $I(GP)$, in $I(GP')$ and in R, respectively, then the compression gain is $T_O - T_C - T_R$, the compression ratio is $\frac{T_C + T_R}{T_O}$.*

Note that when several instances of GP share a triple, this triple will be compressed multiple times but the actual size of the document $\mathcal{D}$ will only be reduced by at most 1. Similarly, when several instances of GP' share a triple, this triple only needs to be included in the compression result $\mathcal{D}'$ once. With these considerations, the compression quantification of a single graph pattern can be characterised as follows:

Lemma 1. (Compression Ratio) *For a GP having n variables and N instances, assuming triples are shared redundantly by instances of GP for S times, then the compression quantification of compressing GP with GP' as defined in Def. 6 is as follows, where I is the redudant number of triples shared by instances of GP':*

$$T_O = N * |E| - S$$
$$T_C = N * m - I$$
$$T_R = m + |E|$$

The lemma is quite straightfoward, $N * |E|$ is the total number of triples in all the GP instances, S is the redundant number of shared triples, which should be removed when calculating T_O. Similarly I should be removed when calculating T_C.

With the above quantification, graph pattern-based RDF logical compression is a problem of finding appropriate graph patterns that yield best (highest) compression gain. To achieve that, we should look for graph patterns with the following criterias:

– larger N, i.e. more instances;
– larger $|E|$, i.e. more triples in the original pattern;
– larger I, i.e. more triples shared by instances of the compressed graph pattern;
– smaller m, i.e. less triples in the compressed pattern;
– smaller S, i.e. less shared triples among instances of the original graph pattern;

These criterias are not easy to satisfy at the same time as the values of these parameters are related to one another. Yet they can already help us to identify "good" or to eliminate "bad" patterns. For example, a compressed graph pattern should not contain circle (removing an edge to eliminate the circle will only reduce the value of m but not the others). More interestingly, we have the following observations:

Lemma 2. *Let GP be a graph pattern containing a constant triple t, whose subject, predicate and object are all constants, then compression gain of GP being compressed by any GP' is no higher than the compression gain of $GP \setminus \{t\}$ being compressed by $GP' \setminus \{t\}$.*

This lemma is quite obvious. Assuming compressing GP with GP' yields compression gain $T_O - T_C - T_R$, because t is shared by all instances of GP, it is maintained only once in T_O. If t is also in GP', it is similarly maintained in T_C only once. In this case, compressing $GP \setminus \{t\}$ with $GP' \setminus \{t\}$ will yield compression gain $(T_O-1)-(T_C-1)-(T_R-2)$; If t is not in GP', it is not included in T_C. Then the compression gain of compressing $GP \setminus \{t\}$ with GP' will be $(T_O-1)-T_C-(T_R-1)$. In both cases, compressing GP with GP' is not more benificial than compressing $GP\setminus\{t\}$ with $GP' \setminus \{t\}$. Although there is only minor, it implies that in graph-pattern based logical compression, we only need to focus on graph patterns with no such "constant triples".

Another observation from the quantification is as follows:

Lemma 3. *Let GP_1, GP_2 be two disconnected graph patterns, then the compression gain of compressing them into a single connected graph pattern is lower than compressing GP_1 and GP_2 separately.*

This lemma is also quite straight-forward. Compressing two disconnected graph patterns into a connected one will only require addition triples to connect previously disconnected instances. This implies that we should always compress each connected graph pattern separately.

A further observation is that the decompression rules themselves, particularly the head part, may also contain redundancies that can be exploited:

Lemma 4. *For N $(N \geq 2)$ graph patterns whose decompression rules share the same head triples $t_1, \ldots, t_M$ $(M \geq 2)$, the compression gain will be higher if we replace $t_1, \ldots, t_M$ with a single triple t' and further compress with a decompression rule $t' \rightarrow t_1, \ldots, t_M$.*

This lemma actually indicates that we can first compress all $t_1, \ldots, t_M$ with t', and then further compress the compressed graph pattern with the replaced rules. The reason is also straight-forward: by changing the rules, we do not change the original triples, nor the triples in the final compression results, but only replace the $M * N$ triples in the

```
<rdf:Description rdf:about="http://data.semanticweb.org/person/jeff-z-pan">
  <foaf:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Jeff Z. Pan</foaf:name>
</rdf:Description>
<rdf:Description rdf:about="http://data.semanticweb.org/person/jeff-z-pan">
  <foaf:creator rdf:resource="http://data.semanticweb.org/conference/iswc/2009/paper/research/423"/>
</rdf:Description>
<rdf:Description rdf:about="http://data.semanticweb.org/conference/iswc/2009/paper/research/423">
  <rdfs:label>Concept and Role Forgetting in ALC Ontologies</rdfs:label>
</rdf:Description>
```

RDF/XML F1

RDF/XML F2

```
<rdf:Description rdf:about="http://data.semanticweb.org/person/jeff-z-pan">
  <foaf:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Jeff Z. Pan</foaf:name>
  <foaf:creator>
    <rdf:Description rdf:about="http://data.semanticweb.org/conference/iswc/2009/paper/research/423">
      <rdfs:label>Concept and Role Forgetting in ALC Ontologies</rdfs:label>
    </rdf:Description>
  </foaf:creator>
</rdf:Description>
```

Fig. 1. Syntactic Redundancy

original compression rules with $N + 1 + M$ triples. This replacement will be beneficial when either M or N is large. In worst case, we only reduce the gain by 2, which is negiligible. This implies a practical simplification of compression: when a graph pattern contains multiple triples of form $<?x, p_i, o_i>$ associated to the same $?x$, we can first compress them into a single triple $<?x, p, o>$ before compressing the rest of the graph pattern.

It is worth mentioning that the logical compression approach can be applied on different syntactic forms of the RDF documents, as long as graph patterns and triples can be accessed. In fact, in our compression solution we also apply it on the compressed serialisation we will introduce in the next section because it has much smaller physical size than the origianl RDF document.

3.2 Syntactic Compression: Graph Pattern-based Serialisation

Logical compression reduces the number of triples in an RDF graph by eliminating semantic redundancies. Nevertheless, with the same set of triples, redundancies can arise depending on how triples are serialised in an RDF file. For example, one RDF graph can be represented as two different RDF/XML files of $F1$ and $F2$ in Figure 1. In $F1$ (cf. the bold and red texts in the upper part), URIs of *jeff-z-pan* and *iSWC09_423* appear twice; in $F2$ both of them appear only once (cf. the bold and blue texts in the lower part). While the two files convey the same meaning, $F2$ is more concise by using RDF/XML's abbreviation and striping syntaxes[10].

The above phenomenon emerging from RDF file *serialisation* can be characterised with the following equation, in which F is a file, $|F|$ is the file size in terms of bits, r_b is the average number of bits needed to encod a resource and N_c is the total number of resource occurrences.

$$|F| = N_c \times r_b \tag{2}$$

[10] `http://www.w3.org/TR/2002/WD-rdf-syntax-grammar-20020325/`

A serialisation F of an RDF graph g with resource occurrences of N_c contains syntactic redundancy if there is another serialisation $\bar{F}$ of g with resources occurrences $\bar{N_c}$, s.t. $\bar{N_c} < N_c$. Let n be the number of triples in an RDF graph. The worst case is $N_c = 3 \times n$, which means that the serialisation is to store triples one by one[11].

Most RDF serialisation approaches provide syntaxes to avoid the worst case e.g., the RDF abbreviation and striping syntax. Similar ideas are also adopted in other RDF serialisation syntaxes like Turtle[12] and Notation 3[13]. Beside the textual serialisations, Fernández and et. al. [5] introduced a binary serialisation approach which applies similar ideas by using bitmaps to record the resource occurrences.

Generally speaking, in the RDF graph, there are two types of syntactic redundancies. The first type is the intra-structure redundancies, which denotes the multiple occurrences of the same resources within the same structures (sub-graphs) of an RDF graph. For example, in Figure 1 *jeff-z-pan* has 2 occurrences in F_1. One of them is redundant and can be committed in F_2. The second type of syntactic redundancies is the inter-structure redundancies, which denotes the multiple occurrences of the same resources across different structures. Suppose there is another author *Jose* in ISWC09 dataset. He might be described using the same graph pattern of *jeff-z-pan*: $GP_{author} = \{<?x, foaf{:}name, ?n>, <?x, foaf{:}made, ?p>, <?x, rdf{:}type, foaf{:}Person>\}$. In such a graph pattern, resources such as $foaf{:}name$, $foaf{:}made$, $foaf{:}Person$, etc. do not have to be repeated for both *jeff-z-pan* and *Jose* in a serialisation. According to the above categorisation, existing work only deals with intra-structure redundancies leaving inter-structure ones untouched. We will show (cf. Sec. 3.2) how these inter-structure ones can be dealt with in our approach.

Grouping triples by graph patterns We introduce a graph pattern-based serialisation method which can remove both intra-structural redundancies and inter-structural redundancies.

$$I_{GP} \xrightarrow{serialised} GP + \big((r_{I_1,1}, ..., r_{I_1,k})...(r_{I_N,1}, ..., r_{I_N,k})\big) \tag{3}$$

As shown in formula 3, given the instances I_{GP} of a graph pattern GP, the serialisation method generates a sequence of bits which is composed of two components. The first component is the graph pattern its self. Such graph pattern is essentially a structure shared by its instances. The second component is a sequence of instances of GP and each instance is a list of resource IDs. By using this graph pattern-based serialisation, we can serialise an RDF graph G as a file F which takes the form as follows.

$$F \rightarrow I_{GP_1}, I_{GP_2}, \ldots, I_{GP_i}, \ldots \tag{4}$$

Given an graph pattern GP, the serialisation size of its instances can be calculated as follows:

$$|I_{GP}| = (|GP| + N \times |V|) \times b$$

[11] For the sake of simplicity, collection constructs like rdf:collection, rdf:list or rdf:bags are viewed as single resources in this calculation.

[12] `http://www.w3.org/TeamSubmission/turtle/`

[13] `http://www.w3.org/DesignIssues/Notation3`

In the above formula, $|GP|$ is the number of resources (edges and constant nodes) in GP, N is the number of instances of GP and V is the number of variable nodes in GP. The size of serialisation file F is calculated as follows:

$$|F| = (\sum_{GP_i \in \{GP\}} |I_{GP_i}|) + |Dictionary| \tag{5}$$

Compared to triple based serialisation (formula 2), the storage space saved by graph pattern based serialisation can be calculated as:

$$\sum_{GP_i \in \{GP\}} |GP_i| \times (N_{GP_i} - 1) \times b$$

4 Implementation

In this section, we discuss the implementation details. Firstly we overview our methods by a framework. Following which are the technical details of the graph pattern-based serialisation and the logical compression method respectively.

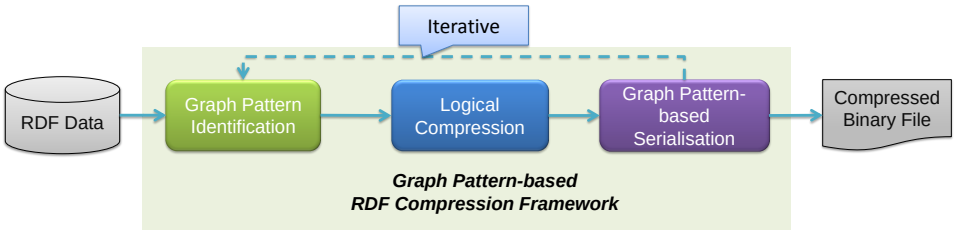

Fig. 2. RDF Data Compression Framework.

The framework (cf. Figure 2) is composed of 3 different steps, including graph pattern identification, logical compression and data serialisation, to exploit both semantic and syntactic redundancies discussed in previous sections.

We describe the main components in this framework as follows:

1. **Graph Pattern Identification** component implements an efficient and incremental graph pattern identification method. The identified graph patterns are not only utilised in semantic compression by the logical compression component, but also used to remove syntactic redundancies by the graph pattern-based serialisation components.
2. **Logical Compression** component implements the graph pattern-based logical compression techniques we presented in the Sec. 3.1. Given the identified graph patterns, it tries to reduce the size of the graph patterns so that the total entities encoded in the instances of graph patterns can be reduced.

3. **Graph Pattern Serialisation** component implements the graph pattern-based serialisation techniques we presented in Sec. 3.2. It produces a compact serialisation of the documents by grouping triple blocks with similar structures, called Entity Description Blocks (EDBs), into graph patterns, and then applying the serialisation techniques;

As shown as dashed line in Figure 2, an iterative compression method is implemented in our framework. After each iteration, the serialisation will result with graph pattern headed files (cf. formula 2 in section 3.2), which can be utilised to identify larger graph patterns efficiently. Larger graph patterns can be used to further reduce redundancies so that repetitive appearances of entities in different graph patterns can potentially be reduced. Below we explain the implementation details of the first two components. The graph pattern-based serialisation has been introduced in section 3.2.

4.2 Graph Pattern Identification

In this paper, we introduce an approach to find and generate graph patterns efficiently. The basic idea is based on an observation that most RDF dump files were generated by following some specific patterns. Such patterns lead to data patterns in the dumped file e.g. triples about an entity are often put together or putting the same relation triples together. Obviously, such data patterns are useful for the serialisation task, or at least good sources for finding more useful graph patterns.

Following this idea, we propose an incremental serialisation approach, which is composed of two stages. In the first stage, it utilises the graph patterns in the RDF file directly. Such existing graph patterns are called direct graph patterns. The second stage moves on from the results of the first stage by manipulating the direct graph patterns to get better patterns for compression. The second stage can be iterative by applying different graph pattern processing techniques.

Stage 1 Finding Direct Patterns This stage is composed of two steps of generating entity description block (EDB shortly) and grouping EDBs. When iterating the triples of an RDF file, we group the triples as an EDB, if these triples share the same subject and also form a continuous sequence in the RDF file.

$$(..., < s_1, p_1, o_1 >, < s_1, p_2, o_2 >, ..., < s_1, p_n, o_n >, ...)$$
$$\xrightarrow{grouped} (..., EDB(s_1), ...)$$

With this method, a sequence of triples is converted into a sequence of EDBs. In the second step, we group EDBs by their schema information which is called entity description pattern as follows.

$$EDP(EDB(s)) = (C, P),$$

where C is the types of s in this EDB and P is the properties of s in the EDB. Hence, basically the grouping operation is to put all EDBs with the same structure together so that we can apply techniques discussed in section 3.2 to store them. The EDP based serialisation approach is called Level 0 method, LV0 shortly.

Stage 2 Merge Graph Patterns In this stage, the main problem to be dealt with is how to $merge$ existing graph patterns from previous stage(s) to get a better pattern which can remove more redundancies. Hence, the key is how to define the $merge$ operator. Generally speaking, a bigger graph pattern will always be better because it can avoid storing the same resources multiple times in smaller graph patterns. One possible $merge$ operation can be merge the EDBs of the same entities together so that we do not have to store the same entity IDs in multiple places in a file. Another possible way is to utilise the linking nature of RDF graph i.e. merge EDBs by there relations. In this paper, we apply the second strategy in the evaluations, where we call it Level 1 method., LV1 shortly.

Later iterations The graph patterns identified stage 2 or later stages can be further enlarged by applying merge operation on the results of current stage. Obviously, more iterations require more processing time. Finding a trade-off between the compression gain and costs of the processing timing is critical in this iterative process. In this paper, we focus only on LV1, i.e. stage 2.

4.3 Logical Compression

We implement the logical compression approach with one variable, which is the basis for logical compression with multiple variables. As discussed in Sec. 4.2, the new graph pattern proposed in this paper eliminates 50% of triples by combining a pair of triples needed to be saved into one triple. This subsection discusses how this kind of compression rules are found.

The logical compression algorithm is shown in Algorithm 1. After direct patterns are constructed, subjects with same predicates will be grouped together into same direct patterns, making finding the instances of each graph pattern very easily. In order to facilitate the logical compression, we support $candidates(GP, threshold)$ as an atomic operation, which is calculated through fast indexing of the graph patterns, and will list the candidate graph patterns whose number of solutions is greater than threshold.

We find the compression graph patterns with the following procedure: We list all candidate graph patterns with enough instances using $candidates(GP, threshold)$. Each candidate graph pattern GP' is renamed as an object property $p_{GP'}$ in the compression. Suppose values of the variable in GP' are $a_1, a_2, \ldots, a_n$, we save the following triples in the compressed dataset:

$$\{< a_1, p_{GP'}, a_2 >, < a_3, p_{GP'}, a_4 >, \ldots, < a_{n-1}, p_{GP'}, a_n >\}$$

5 Evaluation

As introduced in the previous section, we implemented both RDF serialisation and logical compression based graph patterns. And we can support different incremental solutions. In this section, we evaluate their performance and compare against existing technologies.

Algorithm 1 Logical Compression Algorithm

```
1: procedure LOGICAL_COMPRESSION(GPs, threshold)                ▷ GPs are direct patterns
2:     for each direct pattern GP do
3:         NEWGPs ← candidates(GP, threshold)
4:         for each graph pattern GP' in NEWGPs do
5:             IGP' ← instantiations of variables in GP'
6:             rename GP' as p_GP'
7:             compressedSet ←< a_i, p_GP', a_{i+1} >        ▷ a_i ∈ IGP', a_{i+1} ∈ IGP'
8:     return compressedSet
```

Datasets The main strategy of our dataset selection is to use real world datasets with various size and from different domains. The idea is that LV0 of our incremental approach tries to use the direct graph patterns in the dumped RDF files. A heterogeneous datasets might reveal how our approach can work in different situations.

As aforementioned, LV0 utilises the data patterns in the dumped RDF file directly. One might be interested to such direct graph patterns. The first concern would be how many numbers of graph patterns one dataset could have. If there were too many graph patterns, the compression method might not work as expected. For example, in the worst case, each triple is a distinct pattern. In such case, the first level compression would degrade to be each triple based serialisation. The second concern might be the data distributions among such patterns. Our approach prefers leanly distributed data patterns. One of the main reason is that our method would be much more efficient when most of the data only reside in a small number of graph patterns.

Table 1. Dataset Statistics and Direct Graph Patterns

Dataset	Archive Hub	Jamendo	linkedMDB	DBLP2013
#Triples	431,088	1,047,950	6,148,121	94,252,254
Plain File Size	71.8M	143.9M	850.3M	14G
Compressed Size	2.5M	6M	22M	604M
$\#Direct$ GP	623	34	119	77
Top 5 GP	35%	78%	54%	72%

Table 1 gives the statistics of the four datasets used in our experiments. The datasets are sorted ascendantly by size from left to right. The last two rows show the statistics of direct graph patterns. The numbers of direct graph patterns in each dataset are displayed in the fifth row. In most datasets, the pattern numbers are quite small. In addition, the number does NOT increase with the dataset size. This is understandable because graph patterns are more related to the complexity of data schemas instead of the individual numbers. The last row list the ratio of entity numbers in top 5 largest graph patterns to the number of all entities in the dataset. As we can see, in the three large datasets, most data reside in the top 5 graph patterns. The exception is the Archive Hub dataset which also has the largest number of graph patterns. One reason is that the dataset is a gateway of collections in UK. This means that it might cover a large number of concepts.

A quick conclusion from the pattern analysis is that in most datasets the direct graph patterns might be good resources which can be utilised to remove redundancies in them.

Logical Compression Evaluation In section 3.1, we propose a general model of RDF logical compression. In this subsection, we focus on the evaluation of one particular type of graph patterns i.e. $<?x, p, o>$. This type of patterns, or in other word rules covers the intra-property and inter-property rules of [9]. As we pointed out in section 3.1, the compression techniques proposed in [9] can be further optimized by grouping two instances together and compress with 1 triple.

In table 2, we compare such optimised results with the results reported in [9]. As shown there, the optimized results outperforms existing work quite well. In addition to the instance grouping optimization, the results also benefits from frequent data values. As we mentioned, our logical compression, i.e. LV2 method is based on GP-LV0 result. In the first level serialisation, the data values are also assigned with an ID value based on their MD5 hashes. Hence, if some values are frequent, they will be treated similarly as frequent instances.

Table 2. The optimized results of one variable patterns

Data Set	# Total triples	Optimized Compression		RB Comp. ratio
		#Removed triples	Comp. ratio	
Archive Hub	431,088	187,887	**1.77**	1.41
Jamendo	1,047,950	436,101	**1.72**	1.22
LinkedMDB	6,148,121	2,679,593	**1.77**	1.33
DBLP	94,252,254	61,383,224	**2.86**	1.16

Graph pattern-based Serialisation Evaluation The proposed serialisation approach can deal with inter-structure syntactic redundancies which are not touched by most existing approaches. Table 3 gives an idea about the volume of such redundancies (removed by basic EDP patterns) in test datasets. The syntactic redundancies [14] removed by our approach can be quantified as $SRR_{syntac} = \sum_{EDP_i} |EDP_i| \times (f_{EDP_i} - 1)$, where f_{EDP_i} is the frequency or number of instances of EDP_i.

The second row of Table 3 lists the SRR_{syntac} of four datasets. The third row shows the ratios of the redundancies over the whole datasets by $SRR_{syntac}/(3 \times \#Triples_G)$. The fourth row shows the syntactic redundancies our approach can further remove from the results of approaches only dealing with intra-structure redundancies like HDT, which are the inter-structure redundancies. It is interesting to see that in the first three datasets most syntactic redundancies are not dealt with by existing serialisation approaches. The situation in DBLP2013 is different which means that there are more intra-structure redundancies in it. The last row shows the improvements in compression ratio by removing inter-structural redundancies.

In general, these statistics show that compared to intra-structure redundancies the inter-structure ones constitute the major part of syntactic redundancies in all test datasets. This indicates that our graph pattern-based serialisation can improve compression ratio significantly.

[14] The syntactic redundancies are calculated by # (unnecessary) resource occurrences.

Table 3. Inter-structure redundancies removable

Dataset	Archive Hub	Jamendo	linkedMDB	DBLP2013
Total syntactic redundancies	370,389	999,353	5,939,980	79,399,947
Ratio over the original data	28.6%	31.8%	32.2%	28.1%
Inter-structure redundancies	355,917	855,893	5,583,975	46,208,641
Compression Ratio Improvement	38.49%	39.93%	44.65%	22.74%

Rule mining operation evaluation Mining rules from RDF graph directly might be expensive, when the dataset is large. One of the biggest advantage of our incremental approach is that the first serialisation results can provide efficient graph pattern manipulation operations. Firstly, its a compacted representation of the original RDF graph. This makes it more efficient in disk IO and RAM processing. Secondly, and more importantly, the GP-LV0 results are EDP based. From the EDP definition $EDP = (C, P)$, one can figure out that instance types are already treated as constant nodes. Such patterns can be used directly in mining compression rules. Finally, the graph pattern based serialisation makes it very convenient to get pattern based index which can be very useful for mining rules. Given these advantages, we propose a rule mining method based on GP-LV0 results. Figure 3 illustrates the mining time of our approach by comparing to RB [9] compression time. It can be figured out that we can get the compression rules in less than 10 seconds in 3 datasets. For DBLP dataset, we can get the results in about 3 minutes.

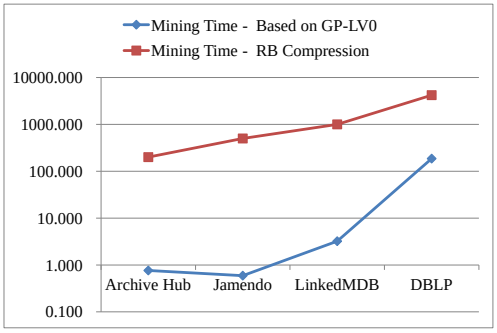

Fig. 3. Rule mining cost

6 Conclusion and Future Work

In this paper, we investigated the problem of application-independent, lossless RDF compression based on graph patterns. By considering the graph nature of RDF and its semantics, we focused on two types of redundancies, namely semantic redundancy and syntactic redundancy. We developed graph pattern-based logical compression and

novel serialisation technologies for RDF data. Evaluation results showed that our approach can complement existing technologies such as HDT and rule-based compression significantly in both semantic and syntactic levels on benchmark datasets. In addition, the evaluation of the rule mining task shows the potentials of the graph pattern-based serialisation in supporting efficient data accesses.

In the future, we will put special focuses on efficient data access over the proposed serialisation formats, e.g. extending the results with dedicated index structures to support SPARQL query answering. In addition, we will also further look into the redundancies of RDF data with special interests in the linked data environment, where the redundancies might be different when different data sources are linked together or different vocabularies are reused.

References

1. S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory rdf engines. arXiv preprint arXiv:1105.4004, 2011.
2. H. Chen, T. Yu, and J. Y. Chen. Semantic web meets integrative biology: a survey. Briefings in bioinformatics, 14(1):109–125, 2013.
3. M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. Web Semantics: Science, Services and Agents on the World Wide Web, 2012.
4. J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto. Rdf compression: basic approaches. In Proceedings of the 19th international conference on World wide web, pages 1091–1092. ACM, 2010.
5. J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez. Compact representation of large rdf data sets for publishing and exchange. In The Semantic Web–ISWC 2010, pages 193–208. Springer, 2010.
6. S. Grimm and J. Wissmann. Elimination of redundancy in ontologies. In The Semantic Web: Research and Applications, pages 260–274. Springer, 2011.
7. P. Hayes. RDF Semantics. Technical report, W3C, Feb 2004. W3C recommendation, http://www.w3.org/TR/rdf-mt/.
8. L. Iannone, I. Palmisano, and D. Redavid. Optimizing rdf storage removing redundancies: an algorithm. In Innovations in Applied Artificial Intelligence, pages 732–742. Springer, 2005.
9. A. K. Joshi, P. Hitzler, and G. Dong. Logical Linked Data Compression. In Proc. of ESWC2013, 2013.
10. M. Meier. Towards rule-based minimization of rdf graphs under constraints. In Web Reasoning and Rule Systems, pages 89–103. Springer, 2008.
11. B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. Web Semantics: Science, Services and Agents on the World Wide Web, 3(1):41–60, 2005.
12. R. Pichler, A. Polleres, S. Skritek, and S. Woltran. Redundancy elimination on rdf graphs in the presence of rules, constraints, and queries. In Web Reasoning and Rule Systems, pages 133–148. Springer, 2010.