

ePub^{WU} Institutional Repository

Javier David Fernandez Garcia and Sabrina Kirrane and Axel Polleres and Simon Steyskal

HDT crypt: Compression and Encryption of RDF Datasets

Article (Accepted for Publication)
(Refereed)

Original Citation:

Fernandez Garcia, Javier David and Kirrane, Sabrina and Polleres, Axel and Steyskal, Simon (2018) HDT crypt: Compression and Encryption of RDF Datasets. *Semantic Web Journal*. ISSN 1570-0844

This version is available at: <http://epub.wu.ac.at/6489/>

Available in ePub^{WU}: September 2018

ePub^{WU}, the institutional repository of the WU Vienna University of Economics and Business, is provided by the University Library and the IT-Services. The aim is to enable open access to the scholarly output of the WU.

This document is the version accepted for publication and — in case of peer review — incorporates referee comments. There are minor differences between this and the publisher version which could however affect a citation.

HDT_{crypt}: Compression and Encryption of RDF Datasets

Javier D. Fernández^{a,b,*}, Sabrina Kirrane^a, Axel Polleres^{a,b} and Simon Steyskal^{a,c}

^a *Institute for Information Business, Vienna University of Economics and Business, Austria*
E-mail: {javier.fernandez,sabrina.kirrane,axel.polleres,simon.steyskal}@wu.ac.at

^b *Complexity Science Hub Vienna, Austria*

^c *CT RDA BAM CON-AT, Siemens AG, Austria*
E-mail: simon.steyskal@siemens.com

Abstract. The publication and interchange of RDF datasets online has experienced significant growth in recent years, promoted by different but complementary efforts, such as Linked Open Data, the Web of Things and RDF stream processing systems. However, the current Linked Data infrastructure does not cater for the storage and exchange of sensitive or private data. On the one hand, data publishers need means to limit access to confidential data (e.g. health, financial, personal, or other sensitive data). On the other hand, the infrastructure needs to compress RDF graphs in a manner that minimises the amount of data that is both stored and transferred over the wire. In this paper, we demonstrate how HDT – a compressed serialization format for RDF – can be extended to cater for supporting encryption. We propose a number of different graph partitioning strategies and discuss the benefits and tradeoffs of each approach.

Keywords: RDF, HDT, compression, encryption, linked data protection

1. Introduction

In recent years, we have seen an increase in the amount of structured data published online using the Resource Description Framework (RDF), in a manner that not only lends itself to data integration but also supports data exchange. Although Linked Data publishers focus on exposing and linking *open* data, there are scenarios where individuals and organisations need to store and share sensitive or private data. Additionally, there are number of regulations concerning the financial, medical, personal, or otherwise sensitive data that require companies to employ strong data protection mechanisms, such as encryption and anonymisation. In order to ensure confidentially it is necessary to encrypt the data not only when it is *in transit* but also when it is *at rest*. In such scenarios, where multiple users have different access rights to different parts of

the data, users should only be able to access the data they are allowed to access.

When it comes to Linked Data protection, to date research has focused on the encryption of partial RDF graphs using eXtensible Markup Language (XML) encryption techniques [19–21] or proposing strategies for querying encrypted RDF data [30]. One of the primary challenges of existing encryption strategies is that they result in a verbose serialization that prevents their use at scale. RDF compression is an emerging research area that focuses on reducing the space requirements of traditional RDF serializations. One approach to efficient data exchange is a (binary) RDF serialization format known as HDT (Header Dictionary Triples) [16] that can be used to compress large datasets in a manner than can be queried without prior decompression [37]. Together encryption and compression mechanisms could be used to cater for the compact storage and efficient exchange of confidential data.

In this paper, we combine “compression+encryption” functionality for RDF datasets, thus allowing

*Corresponding author. E-mail: javier.fernandez@wu.ac.at.

service providers to store and share confidential data while reducing storage and bandwidth usage. In particular, we propose HDT_{crypt} , an extension of HDT to represent encrypted datasets for multiple users with different access rights (i.e. users can only access particular subgraphs of the RDF dataset). To do so, we assume a service provider defines the different “access restricted” subgraphs of a dataset, and we investigate different partitioning strategies to better capture and represent the redundancy (i.e. repeated triples and terms) between them in HDT.

The contributions of our paper can be summarised as follows, we: (i) demonstrate how HDT compression can be extended to cater for encrypted RDF data; (ii) examine a number of alternative partitioning strategies that can be used to reduce the number of duplicates in encrypted HDT (referred to as HDT_{crypt}); and (iii) compare different partitioning strategies in terms of bandwidth and performance. Experiments show that each of our partitioning strategies is able to achieve space savings over the compression baseline (up to 31%), and are comparable in terms of query performance. We present different space/performance trade-offs and discuss how partitioning strategies are influenced both by the number of access restricted subgraphs and the distribution of triples across subgraphs.

The rest of the paper is structured as follows: In *Section 2* we discuss related work on RDF encryption and compression. *Section 3* provides the necessary background information on HDT and *Section 4* describes how compression can be combined with encryption. *Section 5* details the different partitioning strategies that can be used in conjunction with graph based encryption. In *Section 6* we evaluate using both real-world and synthetic RDF datasets and discusses the trade-off between space and performance. Finally, we conclude and highlight future work in *Section 7*.

2. Related Work

When it comes to encryption and RDF, the focus to date has been on proposing strategies for the partial encryption of RDF graphs [19–21] or the querying of encrypted data [30]. Giereth [20, 21] demonstrate how XML based encryption techniques can be used to encrypt confidential data in an RDF-graph, while all non-confidential data is left as plaintext. Gerbracht [19] built on this work by examining how encryption techniques can be used to encrypt RDF elements and RDF subgraphs, in a manner that reduces the storage over-

head. Kasten et al. [30] in turn discuss how data can be encrypted and queried according to SPARQL triple patterns. However this proposal suffers from scalability problems given that each triple is encrypted multiple times depending on whether or not access to the subject, predicate and/or object is restricted. A recent work by Fernández et al. [13] uses Predicate-based Encryption [31] to enable controlled access to encrypted RDF data, i.e., data providers can generate query keys based on (triple-)patterns, whereby one decryption key can decrypt all triples that match its associated triple pattern. In the database and cloud community, Searchable Symmetric Encryption (SSE) [10] has been extensively applied to store and search data in a secure manner. SSE techniques focus on the encryption of outsourced data such that an external user can encrypt their query and subsequently evaluate it against the encrypted data. The more recent Fully Homomorphic Encryption (FHE) [18] technique allows any general circuit/computation over encrypted data, however it is prohibitively slow for most operations [7, 42]. None of these works examine the interplay between encryption and compression, which is the focus of our present paper. In particular, we investigate different HDT compression strategies for RDF datasets, which are organised into different RDF graphs that need to be encrypted with different keys. However, our approach could be adapted to work with partially encrypted graphs.

Following the categorization in [39], an RDF compressor can be classified as either *syntactic* or *semantic*. *Syntactic* compressors try to detect redundancy at the serialisation level, whereas *semantic* compressors try to eliminate logical redundancies. HDT was designed as a binary serialisation format for RDF graphs, but its optimised encodings means that HDT also excels as a syntactic RDF compressor [16, 37]. In HDT RDF data is encoded into two main data-driven components: a *Dictionary* that maps all distinct terms in the dataset to unique identifiers (IDs) (reducing symbolic redundancy), and a triple component that encodes the inner RDF structure as a compact graph of IDs (reducing structural redundancy). This kind of redundancy is also addressed in k^2 -triples [1]. However, in the case of k^2 -triples the authors perform a predicate-based partition of the dataset into disjoint subsets of (subject, object) pairs. These subsets are highly compressed as (sparse) binary matrices that also allow for efficient data retrieval. RDF compression can also benefit from semantic redundancy. Theoretic foundations of exploiting logical redundancies with respect to rules

and grammars have been investigated by [41] and [35], respectively. In particular, the recent compressor gRe-Pair [35] reports the best compression ratios over the structure of RDF graphs (i.e. the graph after ID replacement), to the best of our knowledge.

Likewise, Joshi et al. [28] use rules to discard triples that can be inferred from others, and they only encode these “primitive triples”. In doing so they reduce the number of triples and consequently save space. The authors also propose a combination of semantic and syntactic compression, by integrating their approach with syntactic HDT compression techniques. Interestingly the results were similar to those obtained by simply using HDT. Recently, Wu *et al.* [39] have proposed SSP, a hybrid syntactic and semantic compressor. Their evaluation demonstrates that SSP+bzip2 is slightly better than HDT+bzip2. Other approaches, like HDT-FoQ [37] or WaterFowl [9] also enable compressed data to be retrieved without the need for decompression. Both techniques, based on HDT serialization, report competitive performance at the price of using more space than other compressors such as k^2 -triples or gRePair. We also use HDT compression, however specifically we examine the syntactic redundancy between RDF graphs that need to be encrypted separately, and propose and evaluate four alternative HDT compression strategies. The exploitation of semantic redundancies within HDT is out of scope and left for future work (for more details on semantic compression and HDT we refer the reader to the work by Hernández-Illera et al. [26]).

3. Preliminaries

Before we present our approach, we need to introduce some concepts and terminology from RDF and HDT. Thereafter, in Section 4, we propose a general mechanism to extend HDT with encryption, termed HDT_{crypt} .

As usual, an *RDF Graph* G is a finite set of triples from $I \cup B \times I \times I \cup B \cup L$, where I , B , L denote IRIs, blank nodes and RDF literals, respectively [24]. Figure 1 shows an example of an RDF graph representing two individuals $ex:Bob$ and $ex:Alice$, and the project $ex:pastProject$ of the latter. In this paper, we discuss different ways to compress and encrypt such datasets, using HDT a particular compression format for RDF graphs.

HDT [16] is a binary, compressed serialization format for optimized RDF storage and transmission,

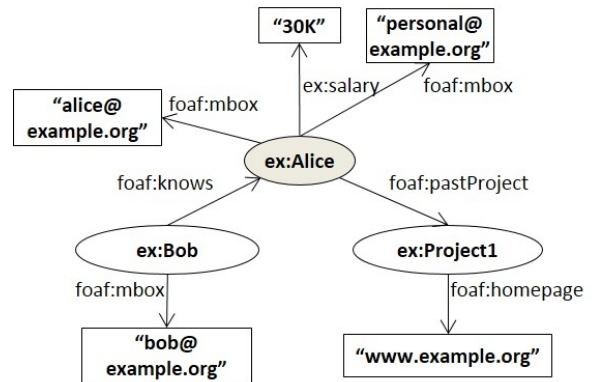


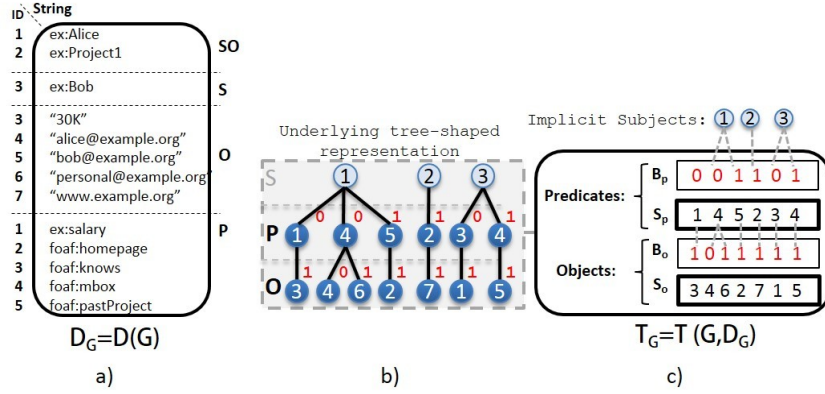
Fig. 1. Example of an RDF graph G .

which also allows certain lookups and queries over compressed data. It is therefore very suitable for the efficient exchange and querying of large datasets. HDT encodes an RDF graph G into three components: the *Header* component H holds metadata, including relevant information necessary for discovery and parsing; the *Dictionary* component D is a catalogue that encodes all RDF terms in G and maps each of them to a unique identifier; the *Triple* component T compactly encodes G 's graph structure as tuples of three IDs that are used to represent the directed labelled edges in an RDF graph.

Figure 2 shows the *Dictionary* component (a), the underlying graph structure (b) and the final *Triple* component (c) for the previous RDF graph G (Figure 1).

3.1. HDT Dictionary Component D

This component organises the terms in a graph G according to their positions in RDF triples, thus we also write $D(G)$ to denote the dictionary component constructed from graph G : the section SO manages terms occurring both as subject and object, and maps them to the ID-range $[1, |SO|]$, where $|SO|$ is the number of such terms acting as subject and object. Sections S and O comprise terms that only occur as subjects or objects, respectively. Both sections are mapped from $|SO|+1$, ranging up to $|SO|+|S|$ and $|SO|+|O|$, respectively. Finally, section P organises all predicate terms, which are mapped to the range $[1, |P|]$. It is worth noting that no ambiguity is possible once we know the role (i.e. the position in a triple, being subject, predicate or object) played by the corresponding ID. For further details, we refer to [38]. For convenience, we write $id(x, D)$ for the particular ID assigned to an RDF term x , whereas we refer to all IDs and RDF

Fig. 2. HDT Dictionary and Triples for our full graph G .

220 terms mapped in a dictionary component D as $ids(D)$
 and $terms(D)$, respectively. Note that, for simplicity,
 we omit the “role” parameter in these functions, which
 should be provided in case the terms in subjects (or
 objects) and predicates are not disjoint [38]. Also, it is
 225 worth mentioning that in the original HDT proposal,
 blank nodes are treated exactly as any other term [16],
 considering an optional skolemization of blank nodes
 as a pre-processing step.

3.2. HDT Triple Component T

230 This component encodes the *structure* of the RDF
 graph after ID substitution, taking into consideration
 a particular dictionary D , thus, we write $T(G, D)$ to
 denote a triple component that was constructed from
 the triples in G using the IDs in dictionary D . More
 235 concretely, RDF triples are encoded as groups of three
 IDs: (id_s, id_p, id_o) , where id_s , id_p , and id_o are the
 IDs of the corresponding subject, predicate, and object
 terms in the dictionary. T organises all triples into a
 240 forest of trees, one per different subject: the subject is
 the root; the middle level comprises the ordered list of
 predicates reachable from the corresponding subject;
 and the leaves list the object IDs related to each (sub-
 ject, predicate) pair. This *underlying representation* (il-
 245 lustrated in Figure 2b) is effectively encoded follow-
 ing the *BitmapTriples* approach [16]. It comprises *two*
sequences: S_p and S_o , concatenating all predicate IDs
 in the middle level and all object IDs in the leaves, re-
 spectively; and *two bitsequences*: B_p and B_o , which are
 250 aligned with S_p and S_o respectively, using a 1-bit to
 mark the end of each list (Figure 2c). In practice, each
 ID in S_p and S_o is encoded with a fixed-length encod-
 ing, using $\log(n)$ bits, where n is the maximum ID in

the sequence [16]. Again, we use $ids(T)$ to refer to all
 IDs used in a triple component T .

3.3. HDT Header Component H

260 The HDT Header includes (i) the machine-readable
 metadata that is necessary to process an HDT file (*for-
 mat metadata*); and (ii) additional human-readable in-
 formation to describe the dataset (usually in the form
 of VoID¹ descriptions). The format metadata is mainly
 265 focused on characterising the dictionary and triple for-
 mats. In general, an HDT file of a graph G consists
 of a single header H , dictionary D and triples T ,
 $HDT(G) = (H, D, T)$. Nonetheless, the HDT specifi-
 270 cation [15] is flexible and allows for several dictionar-
 ies or triple components to be specified in H as soon
 as the interpretation of their relationship is provided in
 the header. It was envisaged that this would be used to
 split huge RDF graphs into several chunks or streams,
 where a sequential order of the components is assumed
 by default [15]. In the following section we exploit and
 expand this feature to encode a partition of the graph
 G with several dictionaries and triples.

4. HDT_{crypt}: Extending HDT for Encryption

275 We introduce HDT_{crypt} , an extension of HDT that
 involves encryption of RDF graphs. We first define
 the notion of access-restricted RDF datasets and the
 implications for HDT (Section 4.1). Then, we show
 an extension of the HDT header component to cope
 280 with access-restricted RDF datasets (Section 4.2),

¹<http://www.w3.org/TR/void/>

which leads to the final HDT_{crypt} encoding. Finally, as HDT_{crypt} can manage several HDT Dictionary components, we describe the required operations to integrate different Dictionary components within an HDT collection (Section 4.3). These operations will be the basis to represent the shared components between access-restricted datasets efficiently, addressed in Section 5.

4.1. Representing access-restricted RDF datasets

We consider hereinafter that users wishing to publish *access-restricted RDF datasets* divide their complete graph of RDF triples G into (named) graphs, that are accessible to other users, i.e. we assume that access rights are already materialised per user group in the form of a set (cover) of separate, possibly overlapping, RDF graphs, each of which are accessible to different sets of users.

Borrowing terminology from [25], an *access restricted RDF dataset* (or just “dataset” in the following) is a set $DS = \{G, (g_1, G_1), \dots, (g_n, G_n)\}$ consisting of a (non-named) default graph G and named graphs s.t. $g_i \in I$ are graph names, where in our setting we require that $\{G_1, \dots, G_n\}$ is a cover² of G . We further call DS a *partition* of G if $G_i \cap G_j = \emptyset$ for any $i \neq j$; $1 \leq i, j \leq n$. Note that from any dataset DS , a *canonical partition* DS' can be trivially constructed (but may be exponential in size) consisting of all non-empty (at most $2^n - 1$) subsets G'_S of triples $t \in G$ corresponding to an index set $S \in 2^{1, \dots, n}$ such that $G'_S = \{t \mid t \in \bigcap_{i \in S} G_i \wedge \neg \exists S' : (S' \supset S \wedge t \in \bigcap_{j \in S'} G_j)\}$.

Figure 3 shows an example of such a dataset composed of three access-restricted subgraphs (or just “subgraphs” in the following) G_1, G_2, G_3 for the previous full graph G (Figure 2a). Intuitively, this corresponds to a scenario with three access rights: users who can access general information about projects in an organisation (graph G_1); users who have access to public email accounts and relations between members in the organisation (graph G_2); and finally, users who can view personal information of members, such as the salary and personal email accounts (graph G_3). As can be seen, the triple (ex:Alice foaf:mbox "alice@example.org") is repeated in subgraphs G_2 and G_3 , showing a redundancy which can produce significant overheads in realistic scenarios with large-scale datasets and highly overlapping graphs. Canonical partitioning groups these triples into disjoint sets so

that no repetitions are present. In our example in Figure 3, the set $G'_{\{2,3\}}$, which can simply be written as G'_{23} , holds this single triple, (ex:Alice foaf:mbox "alice@example.org"), hence this triple is not present in G'_2 and G'_3 . In this simple scenario, G'_1 is equivalent to G_1 as it does not share triples with other graphs.

Thus, we consider hereinafter an *HDT collection* corresponding to a dataset DS denoted by $HDT(DS) = (H, \mathcal{D}, \mathcal{T})$ as a single H , plus sets $\mathcal{D} = \{D_1, \dots, D_n\}$, $\mathcal{T} = \{T_1, \dots, T_m\}$ of dictionary and triple components, respectively, such that the union of triple components encodes a cover of G , i.e. the overall graph of all triples in the dataset DS . We do not assume that there is a one-to-one correspondence between individual triple components in \mathcal{T} and graphs in DS ; different options of mapping subgraphs to HDT components will be discussed in Section 5 below. The relation between the dictionaries and the triple components (in other words, which dictionaries are used to codify which triple components) is also flexible and must be specified through metadata properties. In our case, we assume $H = \{R, M\}$ to contain a relation $R \subseteq \mathcal{D} \times \mathcal{T}$, which we call the *dictionary-triples map* with the implicit meaning that dictionary components encode terms used in the corresponding triple components, and M is comprised of additional header metadata (as mentioned above, the header contains a variety of further (meta-)information in standard HDT [15], which we skip for the considerations herein). It is worth noting that we do not prescribe that either \mathcal{D} or \mathcal{T} do not overlap. However, it is clear that one should find an unambiguous correspondence to decode the terms under $ids(T)$.

Thus, we define the following admissibility condition for R . An HDT collection is called *admissible* if:

$$\begin{aligned} & - \forall D_i, D_j \in \mathcal{D} : (D_i, T), (D_j, T) \in R \wedge i \neq j \implies \\ & \quad terms(D_i) \cap terms(D_j) = \emptyset \\ & - \forall T \in \mathcal{T} : i \in ids(T) \implies \exists (D, T) \in R \wedge i \in \\ & \quad ids(D) \end{aligned}$$

For any admissible HDT collection HDT we define the *T-restricted collection* HDT^T as the collection obtained from removing: (i) all triple components $T' \neq T$ from HDT ; (ii) the corresponding D' such that (D', T') is in R and (D', T) is not in R ; and (iii) the relations (D', T') from R . Thus allowing an HDT collection to be filtered by erasing all dictionary and triple components that are not required for T .

²In the set-theoretic sense.

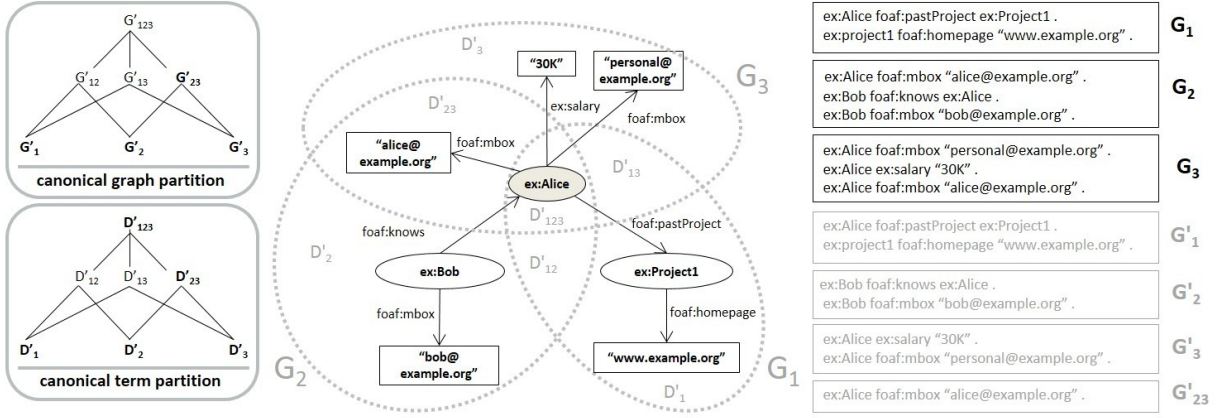


Fig. 3. An access-restricted RDF dataset such that G comprises three separate access-restricted subgraphs G_1, G_2, G_3 ; the graph's canonical partition is comprised of four non-empty subgraphs $G'_1, G'_2, G'_3, G'_{123}$, whereas the *terms* in these graphs can be partitioned into five non-empty subsets corresponding to the dictionaries $D'_1, D'_2, D'_3, D'_{12}, D'_{13}$.

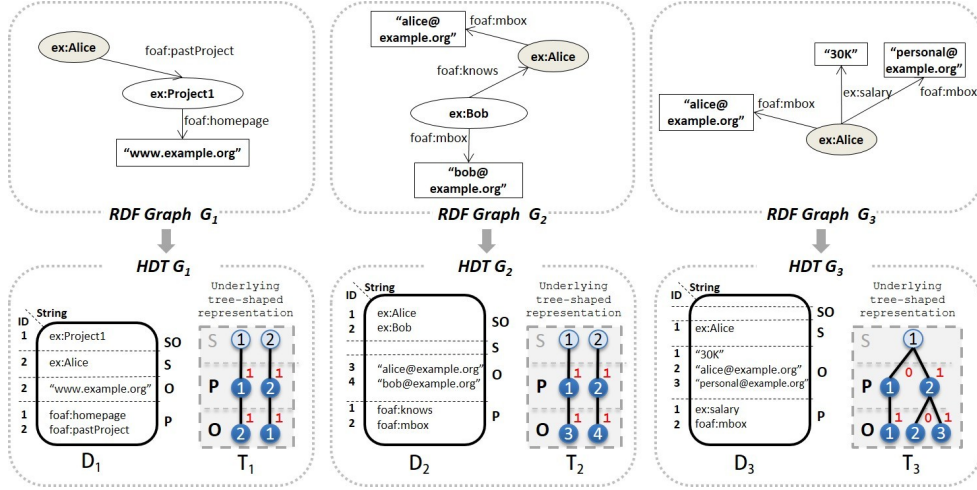


Fig. 4. $HDT_{crypt-A}$, create and encrypt one HDT per partition.

4.2. HDT_{crypt} encoding

375 We now introduce the final encoding of the HDT_{crypt}
 extension. HDT_{crypt} uses AES (Advanced Encryption
 Standard) [11] to encrypt the D and triple components
 of an HDT collection and extends the header H with
 380 a keymap $kmap : \mathcal{D}_{crypt} \cup \mathcal{T}_{crypt} \mapsto I$ that maps *en-*
*crypt*ed components to identifiers (IRIs), which denote
 AES keys that can be used to decrypt these compo-
 385 nents.

Thus, $HDT_{crypt} = (H, \mathcal{D}_{crypt}, \mathcal{T}_{crypt})$ where $H =$
 $\{R, kmap, M\}$, $R \subseteq \mathcal{D}_{crypt} \times \mathcal{T}_{crypt}$, and the components
 385 in \mathcal{D}_{crypt} and \mathcal{T}_{crypt} are encrypted with keys identified
 in $kmap$.

The operations to *encrypt* and *decrypt* the dic-
 tionary and triples are described as follows. First,
 the operation *encrypt* takes one or more dictionary
 and triples and encrypts the components with a given
 key. Formally, we write $encrypt(c, key_{crypt}) = c_{crypt}$,
 where $c \in \mathcal{D} \cup \mathcal{T}$ to denote the component $c_{crypt} \in$
 $\mathcal{D}_{crypt} \cup \mathcal{T}_{crypt}$ obtained by encrypting c with the key
 key_{crypt} . While, we add an identifier of the components
 to the header metadata. In other words, $id(c_{crypt}) \mapsto$
 $IRI(key_{crypt})$ is added to the $kmap$, where id denotes
 the ID of the component in \mathcal{D}_{crypt} and \mathcal{T}_{crypt} and IRI a
 unique identifier for the symmetric key.

For the decryption, it is assumed that an authorized
 user u has partial knowledge about these keys, i.e. they
 400

have access to a partial function $key_u : I_u \mapsto K$ that maps a finite set of “user-owned” key IDs $I_u \subseteq I$ to the set of AES (symmetric) keys K . The decryption simply takes the given compressed component(s) and performs the decryption with the given symmetric key. Formally, we write $decrypt(c_{crypt}, key_{crypt}) = c$, where $c_{crypt} \in \mathcal{D}_{crypt} \cup \mathcal{T}_{crypt}$ to denote the component $c \in \mathcal{D} \cup \mathcal{T}$ obtained from decrypting c_{crypt} with the key $key_{crypt} = key(kmap(c_{crypt}))$. Further we write $decrypt(HDT_{crypt}, I_u)$ to denote the non-encrypted HDT collection consisting of all decrypted dictionary and triple components of HDT_{crypt} which can be decrypted with the keys in $\{key_u(i) \mid i \in I_u\}$. In other words, the T -restriction of HDT_{crypt} is defined analogously to the above-said.

4.3. Integration operations

Finally, we define two different ways of integrating dictionaries $D_1, \dots, D_k \in \mathcal{D}$ within an HDT collection: D -union and D -merge. In the former, we replace dictionaries with a new dictionary that includes the union of all terms. In the latter, we establish one of the dictionaries as the dictionary baseline and rename the IDs of the other dictionaries.

4.3.1. D -union

The D -union is only defined for $D_1, \dots, D_k \subseteq \mathcal{D}$ if the following condition holds on R : $\forall (D_i, T) \in R : (\neg \exists D_j \notin D_1, \dots, D_k \text{ such that } (D_j, T) \in R)$. In other words, we can perform a D -union if all T -components depending on dictionaries in the set D_1, \dots, D_k only depend on these dictionaries. Then, we can define a trivial D -union of HDT wrt. D_1, \dots, D_k , written $HDT_{D_1 \cup \dots \cup D_k}$, as follows:

- replace $\{D_1, \dots, D_k\}$ dictionaries with a single dictionary $D_{1\dots k} = D_1 \cup \dots \cup D_k$, such that $\forall x \in terms(D_1) \cup \dots \cup terms(D_k)$
 - * $x \in terms(D_{1\dots k})$
 - * $id(x, D_{1\dots k})$ is obtained by sequentially numbering the terms in $terms(D_1) \cup \dots \cup terms(D_k)$ upon an (arbitrary) total order, e.g., lexicographically ordering the terms (as it is done in HDT dictionaries by default).
- replace all $(D_i, T) \in R, i \in \{1, \dots, k\}$, with new $(D_{1\dots k}, T')$ relations, where T' is obtained from T by replacing the original IDs from D_i with their corresponding new IDs in $D_{1\dots k}$.

4.3.2. D -merge

In the more general case where the condition for D -unions does not hold on $D_1, \dots, D_k \subseteq \mathcal{D}$, we can define another operation, D -merge, written $HDT_{D_1 \triangleright \dots \triangleright D_k}$. We start with the binary case, where only two dictionaries D_1 and D_2 are involved; $HDT_{D_1 \triangleright D_2}$ is obtain as follows:

- replace D_1 and D_2 with a single $D_{12} = D_1 \triangleright D_2$,³ such that
 - * $\forall x \in terms(D_1) : id(x, D_{12}) = id(x, D_1)$
 - * $\forall x \in terms(D_2) \setminus terms(D_1) : id(x, D_{12}) = id(x, D_2) + max(ids(D_1))$
- replace all $(D_1, T_1) \in R$ with (D_{12}, T_1)
- replace all $(D_2, T_2) \in R$ with (D_{12}, T'_2) , where T'_2 is obtained from T_2 by analogous ID changes.

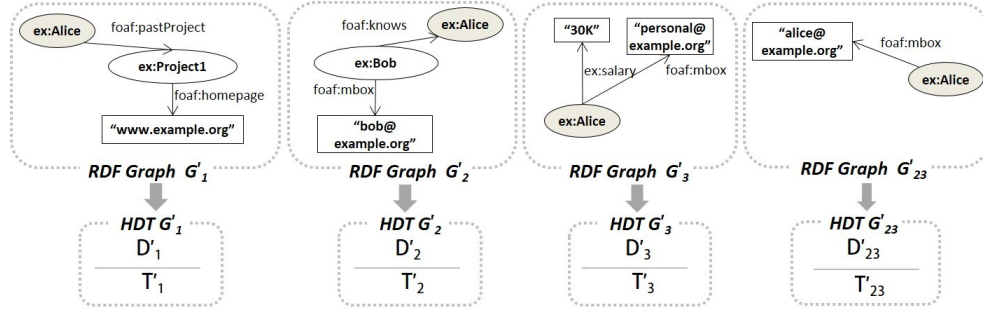
D -merge can then be trivially generalized to a sequence of dictionaries assuming left-associativity of \triangleright operator. That is, $HDT_{D_1 \triangleright D_2 \triangleright \dots \triangleright D_k} = HDT_{((D_1 \triangleright D_2) \triangleright \dots) \triangleright D_k}$.

For convenience, we extend the notation of $T(G, D)$ from Section 3.2 to D -unions and D -merges: let (D_1, \dots, D_k) be a sequence of dictionaries and G an RDF graph such that $terms(G) = \bigcup_{D_i \in (D_1, \dots, D_k)} terms(D_i)$. Then we will write $T(G, (D_1 \cup \dots \cup D_k))$ and $T(G, (D_1 \triangleright \dots \triangleright D_k))$ for the triples part generated from G according to the combined dictionary $((D_1 \cup D_2) \cup \dots) \cup D_k$ and $((D_1 \triangleright D_2) \triangleright \dots) \triangleright D_k$ respectively. Finally, we note that for any admissible HDT collection, both D -union and D -merge preserve admissibility.

5. Efficient Partitioning HDT_{crypt}

After having introduced the general idea of HDT_{crypt} and the two different ways of integrating dictionaries within an HDT collection, we now discuss four alternatives strategies that can be used for distributing a dataset DS across dictionary and triple components in an HDT_{crypt} collection. These alternatives, referred to as $HDT_{crypt-A}$, $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$, provide different space/performance tradeoffs that will be evaluated in Section 6. We note that HDT behaves differently than the normal RDF merge regarding blank nodes in different “partitions” as, by default, HDT does not rename the blank nodes to avoid shared labels [27]: the original blank nodes are skolemized to constants (unique per RDF graph) and preserved across partitions, so that we do not need to consider blank node (re-)naming separately.

³We use the directed operator \triangleright instead of \cup here, since this operation is not commutative.

Fig. 5. $HDT_{crypt-B}$, extracting non-overlapping triples.

5.1. $HDT_{crypt-A}$: A Dictionary and Triples per Named Graph in DS

The baseline approach is straightforward, we construct separate HDT components $D_i = D(G_i)$ and $T_i = T(G_i, D_i)$ per graph G_i in the dataset, see Figure 4, thereafter each of these components is encrypted with a respective, separate key, identified by a unique IRI $id_i \in I$, i.e., $kmap(D_i) = kmap(T_i) = id_i$ and $R = \{(D_i, T_i) \mid G_i \in DS\}$. For re-obtaining graph G_i a user must only have access to the key corresponding to id_i , and can thereby decrypt D_i and T_i and extract the restricted collection HDT^{T_i} , which corresponds to G_i . Obviously, this approach encodes a lot of overlaps in both dictionary and triples parts: that is, for our running example from Figure 4, the IRI for `ex:alice` is encoded in each individual D component and the overlapping triples in graphs G_2 and G_3 appear in both T_2 and T_3 respectively (cf., Figure 4).

5.2. $HDT_{crypt-B}$: Extracting non-overlapping Triples in DS'

In order to avoid the overlaps in the triple components, a more efficient approach could be to split the graphs in the dataset DS according to their canonical partition DS' and again construct separate (D, T) -pairs for each subset $G'_S \in DS'$, see Figure 5. That is, we create $D'_S = D(G'_S)$ and $T'_S = T(G'_S, D'_S)$ per graph $G'_S \in DS'$, where $S \in 2^{1, \dots, i}$ denotes the index set corresponding to a (non-empty) subset of DS' . R in turn contains pairs (D'_S, T'_S) and $kmap$ entries for keys identified by I'_S per G'_S used for the encryption/decryption of the relevant D'_S and T'_S . The difference for decryption now is that any user who is allowed access to G_i must have all keys corresponding to any I'_S such that $i \in S$ in order to re-obtain the original graph G_i .

First, the user will decrypt all the components for which they have keys, i.e. obtaining a non-encrypted

collection HDT' consisting of components $D' = \{D'_1, \dots, D'_k\}$, $T' = \{T'_1, \dots, T'_k\}$ consisting of the components corresponding to a partition of G_i . Then, for decompressing the original graph G_i , we create separate T'_S -restricted HDTs, which are decompressed separately, with G_S being the union of the resulting subgraphs.

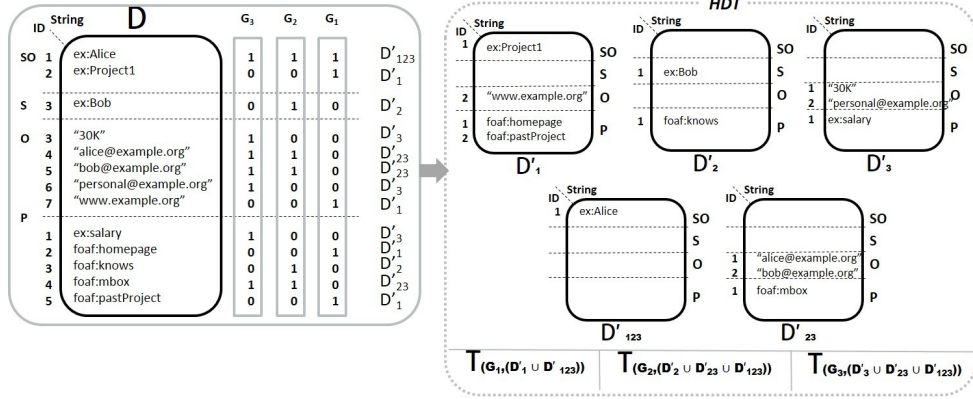
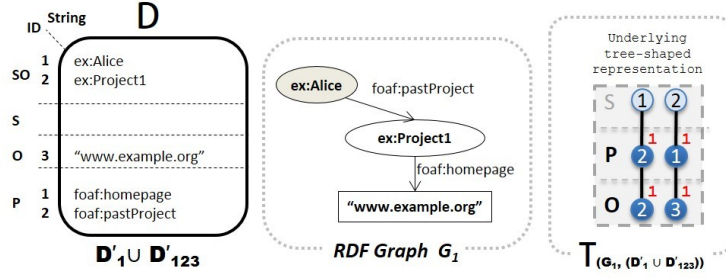
5.3. $HDT_{crypt-C}$: Extracting non-overlapping Dictionaries in DS'

Note that in the previous approach, we have duplicates in the dictionary components. An alternative strategy would be to create a *canonical partition* of *terms* instead of triples, and create separate dictionaries $D'_S \in D'$ for each non-empty term-subset,⁴ respectively. Figure 6 shows the canonical partition of terms in our running example: as can be seen, the original dictionary is split into five non-empty terms-subsets corresponding to the dictionaries D'_{123} (terms shared in all three graphs), D'_{23} (terms shared in graphs G_2 and G_3 that are not in D'_{123}) and D'_1, D'_2, D'_3 (terms in either G_1, G_2 or G_3 resp. and are not shared between graphs). This partition can be computed efficiently, thanks to the HDT dictionary D of the full graph G , which we assume to be available⁵. To do so, we keep⁶ an auxiliary bitsequence per graph G_i (see Figure 6, top left), each of size $terms(D)$. Then, we iterate through triples in each graph G_i and, for each term, we search its ID in D , marking such position with a 1-bit in the bitsequence of G_i . Finally, the dictionaries of the subsets can be created by inspecting the combinations of 1-bits

⁴Again, here $S \in 2^{1, \dots, n}$ represents an index set.

⁵All HDT_{crypt} strategies are evaluated from an existing full graph G . Our evaluation in Section 6 also reports the time to create the HDT representation of the full graph G

⁶This auxiliary structure is maintained just at compression time and it is not shipped with the encrypted information.

Fig. 6. $HDT_{crypt-C}$, extracting non-overlapping dictionaries.Fig. 7. Union of dictionaries (in $HDT_{crypt-C}$) to codify the non-overlapping dictionaries of a partition.

in the bitsequences: terms in $D'_{xy\dots z}$ will be those with a 1-bit in the bitsequences of graphs $xy\dots z$ and 0-bits in other graphs. For instance, in Figure 6, D'_{123} is constituted only by `ex:alice`, because it is the only term with three 1-bits in the bitsequences of G_1 , G_2 and G_3 . In contrast, `ex:Project1` will be part of D'_1 as it has a 1-bit only in the bitsequence of G_1 .

The number of triple components in this approach are as in $HDT_{crypt-A}$, one per graph G_i . However, they are constructed slightly differently as, in this case, we have a canonical partition of terms and one user will just receive the dictionaries corresponding to subsets that correspond to terms in the graph G_i that they have been granted access to. In other words, the IDs used in each T_i should unambiguously correspond to terms, but these terms may be distributed across several dictionaries.⁷ Thus, we encode triples with a D -union (see Section 4.3) of the D'_S such that $i \in S$. That is, for each

G_i we construct $T_i = T(G_i, (\bigcup_{i \in S} D'_S))$, and add the respective pairs (D'_S, T_i) in R .

Figure 7 illustrates this merge of dictionaries for the graph G_1 and the respective construction of $T(G_1, (D'_1 \cup D'_{123}))$. The decompression process after decryption is the exact opposite. For decompressing the graph G_i , the decrypted dictionaries $\bigcup_{i \in S} D'_S$ are used to create a D -union D_i which can be used to decompress the triples T_i in one go. Finally, as a performance improvement at compression time, note that, although the canonical partition of terms has to be built to be shipped in the compressed output, we can actually skip the creation of the D -union dictionaries to encode the IDs in the triples. To do so, we make use of the bitsequences to get the final IDs that are used in the triples: One should note that the ID of a term in a D -union of a graph G_i is the number of previous 1-bits in the bitsequence of G_i (for each SO , S , O , and P section). For instance, in our example in Figure 7, `ex:Project1` is encoded with the ID=2. Instead of creating D_1 , we can see that in the bitsequence of G_1 (see Figure 6, top right) we have two 1-bits in the predicate section up to

⁷Given the partition definition, it is nonetheless true that a term appears in one and only one term-subset.

the position where `ex:Project1` is stored in the original dictionary, hence its $ID=2$.

600 5.4. $HDT_{crypt-D}$: Extracting non-overlapping 605 Dictionaries and Triples in DS'

In $HDT_{crypt-D}$, we combine the methods of both $HDT_{crypt-B}$ and $HDT_{crypt-C}$. That is, we first create a *canonical partition of terms* as in $HDT_{crypt-C}$, and a *canonical partition of triples* DS' as in $HDT_{crypt-B}$. Then, we codify the IDs in the subsets of DS' with the IDs from the dictionaries. Note, however, that in this case there is – potentially – an n:m between the resulting dictionary and triple components. In other words, triples in T'_S can include terms that are not only in D'_S as they may be distributed across several term-subsets. For instance, in our running example, T'_1 in $HDT_{crypt-B}$ includes `ex:Alice` (see Figure 5) which is stored in D'_{123} in $HDT_{crypt-C}$ (see Figure 6). One alternative could be to create a D -union of each graph G'_S and codify triples in T'_S with the corresponding IDs. However, it is trivial to see that this would lead to an exponential number of D -union dictionaries, one per T'_S component. In addition, we would need to physically recreate all these dictionaries at compression time, and also at decompression time in order to decompress each single graph G'_S . Thus, we perform a D -merge approach (see the definition in Section 4.3), which fits perfectly with n:m relations. This is illustrated in Figure 8. As can be seen, triples in each G'_S of the canonical partition are encoded with an appropriate D -merge of term-subsets. A practical example is shown in Figure 9, representing the encoding of graph G'_3 . As defined in D -merge, IDs are assigned in order, that is for a merge $D'_1 \triangleright \dots \triangleright D'_k$, the IDs in D'_k are shifted $\max(ids(D'_1)) + \dots + \max(ids(D'_{k-1}))$. For instance, in our example, the predicate `ex:salary` will be encoded in G'_3 with the $ID=2$, because its local ID in D'_3 is 1, and it has to be shifted $\max(ids(D'_{123})) + \max(ids(D'_{23})) = 1$, hence its final $ID=1+\max(ids(D'_{123}))+\max(ids(D'_{23})) = 2$. Note that here we restrict the dictionaries D' per section (SO , S , O and P). Given the special numbering of IDs in HDT, where S and O IDs follow from SO as explained in Section 3.1. This is illustrated in our example, e.g. the object “30K” with local $ID=1$ in D'_3 is mapped in the D -merge dictionary with 4, as it sums up all the previous objects and subjects IDs in D'_{123} and D'_{23} .

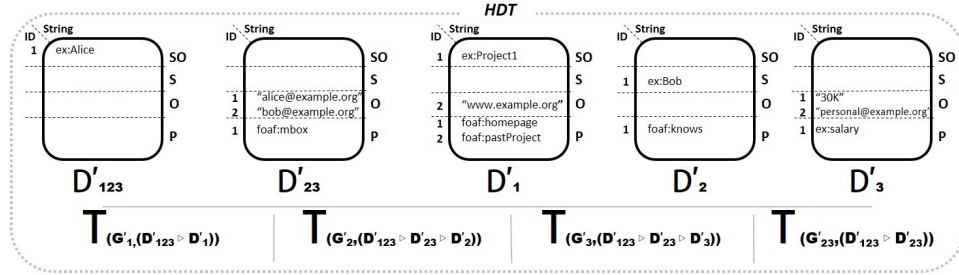
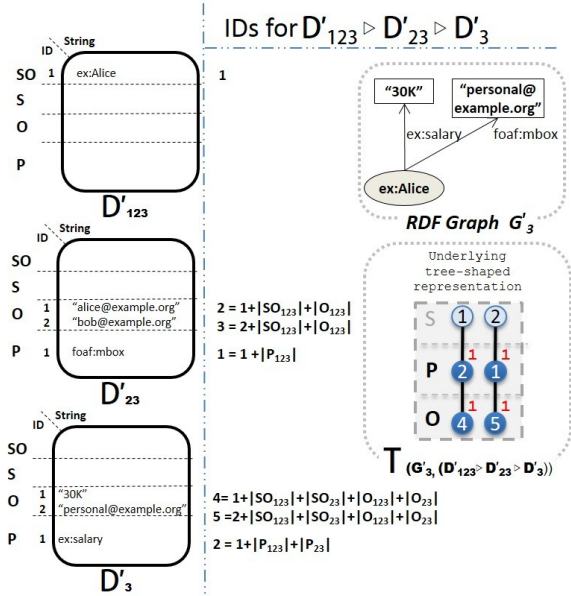
645 It is worth mentioning that no ambiguity is present in the order of the D -merge as it is implicitly given

by the partition DS' as per the canonical term partition. Thus, the decompression follows the opposite process: for each graph T'_S in the partition of the graph G_i , the user processes each ID and, depending of the value, they get the associated term in an appropriate term subset. For instance, if the user is accessing the predicate $ID=2$ in our example, one can easily see that $2 > |P_{123}| + |P_{23}|$, so dictionary D'_3 has to be used⁸. The local ID to look at is then $2 - |P_{123}| - |P_{23}| = 1$, hence the predicate $ID=1$ in D'_3 is inspected and then `foaf:pastProject` is retrieved. Finally, note that not all terms in a D -merge are necessarily used when encoding a particular T'_S . For instance, in our example in Figure 9, the object “`bob@example.org`” with $ID=2$ in D'_{23} (and $ID=3$ in the D -merge) is not used in T'_3 . However, this ID is “blocked”: it cannot be used by a different object in T'_3 as this ID is taken into account when encoding the present objects (“30K” and “`personal@example.org`”), once we sum the $\max(ids(D'_{23}))$ as explained above. The same consequence applies to subjects, so that subject IDs are not necessarily correlative in T'_S . This constitutes a problem for the HDT Bitmap Triples encoding (presented in Section 3.2), given that it represents subjects implicitly assuming that they are correlative. Thus, $HDT_{crypt-D}$ has to explicitly state the ID of each subject, which constitutes a space overhead and a drawback of this approach, despite the fact that duplicate terms and triples are avoided. Technically, instead of a forest of trees, triples are codified as tuples of three IDs, using an existing HDT triples representation called *Plain Triples* [15].

6. Evaluation

This section evaluates the performance of HDT_{crypt} by comparing each of the aforementioned partitioning strategies with respect to the performance of the algorithms and the size of the compressed encrypted dataset. We first describe our experimental setup in detail. Then, we present our evaluation results in terms of three distinct yet related tasks: (i) performance of compression and encryption algorithms and size of resulting datasets; (ii) performance of decryption and decompression algorithms; and (iii) performance of

⁸We abuse notation to denote the cardinality of a set, e.g. $|P_{123}|$, as the maximum id represented in such dictionary set.

Fig. 8. $HDT_{crypt-D}$, extracting non-overlapping dictionaries and triples.Fig. 9. Merge of dictionaries (in $HDT_{crypt-D}$) to codify the non-overlapping dictionaries and triples of a partition.

triple pattern queries⁹ over the compressed datasets, which constitute the basis for SPARQL's graph pattern matching [25].

Finally, we provide a summary and discussion of the results in Section 6.5. Additional experiments can be found in Appendix A.

6.1. Experimental Setup

The proof-of-concept HDT_{crypt} prototype¹⁰ uses the existing HDT-C++ library¹¹ for compression and de-

compression, and standard Java libraries for AES encryption/decryption¹².

The evaluation is performed on three different datasets, described in Table 1.

First, we selected DBpedia, the well-known RDF knowledge base extracted from Wikipedia, which was chosen due to the volume and variety of the data and large number of dictionary terms therein. We used two different versions, DBpedia 3.8¹³ and the latest version 2016-10¹⁴, which is double the size of the previous one. Hereinafter, we will use the term *DBpedia* to refer to both versions, as the results are comparable. Then, we chose a realistic scenario using the configuration used in SAFE [32], a query federation engine with access control. The SAFE dataset includes public statistical data (referred to as external) and anonymised clinical data (internal).

Additionally, in order to test the scalability of the various partitioning strategies we use the Lehigh University Benchmark (LUBM) [23] data generator to obtain synthetic datasets of incremental sizes from 1,000 universities (LUBM1K, including 0.13 billion triples) to 4,000 universities (LUBM4K, 0.53 billion triples). Table 1 shows the original dataset sizes in plain N-Triples (NT). In addition, we provide details of the size of the datasets compressed with *gzip*, HDT and HDT+*gz* (*gzip* compression over the HDT file). This shows that our HDT compression ratios are in line with the original proposal [16]. Finally, the last column of the table shows the time (in minutes) to compute the HDT representation of each dataset. In turn, the HDT creation time for LUBM grows linearly with the number of triples. This result is also in accordance with

⁹Matching RDF triples in which each component may be a variable

¹⁰Source code and all experiment data are available at the HDT_{crypt} homepage: <https://aic.ai.wu.ac.at/ComCrypt/HDTcrypt/>

¹¹<https://github.com/rdfhdt/hdt-cpp>

¹²<http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>

¹³<http://wiki.dbpedia.org/data-set-38>

¹⁴<http://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10>

Table 1
Statistical dataset description

DATASET	TRIPLES	S _O	S	O	P	Size (GB)				HDT creation time (m)
						NT	NT+gz	HDT	HDT+gz	
DBpedia 3.8	0.43BN	22.0M	2.8M	86.9M	58.3K	61.6	4.9	6.4	2.7	96
DBpedia 2016-10	0.84BN	44.5M	55.9M	225.6M	63.8K	122.0	9.6	12.1	5.0	249
SAFE	0.07BN	171.5K	7.4M	3.6M	346	12.4	0.3	0.6	0.07	10
LUBM1K	0.13BN	5.0M	16.7M	11.2M	18	18.0	0.6	0.7	0.2	18
LUBM2K	0.27BN	10.0M	33.5M	22.3M	18	36.2	1.3	1.5	0.5	36
LUBM3K	0.40BN	14.9M	50.2M	33.5M	18	54.4	1.9	2.3	0.8	57
LUBM4K	0.53BN	19.9M	67.0M	44.7M	18	72.7	2.5	3.1	1.0	78

Table 2
% Duplicates and size of subgraphs.

SUBGRAPHS	DATASET	DUP %	Size of subgraphs (GB)											
			G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{11}	G_{12}
6	DBpedia 3.8	11.62%	11.6	11.7	11.5	11.7	11.6	11.5						
	DBpedia 2016-10	11.62%	23.2	23.2	23.0	23.1	23.0	22.5						
9	DBpedia 3.8	22.32%	8.9	8.9	8.9	8.8	8.9	8.8	8.8	8.9	8.7			
	DBpedia 2016-10	22.32%	17.6	17.5	17.5	17.4	17.5	17.5	17.5	17.1	17.4			
12	DBpedia 3.8	32.54%	7.6	7.6	7.7	7.6	7.6	7.6	7.7	7.6	7.6	7.6	7.6	7.5
	DBpedia 2016-10	32.54%	15.2	15.0	15.2	15.1	15.1	15.1	15.1	15.1	15.1	15.1	14.7	15.2
8	SAFE	0.00%	7.0	3.2	1.9	0.1	0.1	0.1	0.01	0.01				
6	LUBM1K	37.89%	14	5.2	5	4.5	1.6	0.6						
	LUBM2K	37.89%	27	10.7	10.7	8.9	3.1	1.3						
	LUBM3K	37.89%	39.5	16.1	15.2	13.4	4.6	1.9						
	LUBM4K	37.89%	52.8	21.4	20.3	17.9	6.1	2.4						
9	LUBM1K	38.26%	14	5.2	4.5	3.1	1.6	1.3	0.9	0.6	0.2			
	LUBM2K	38.26%	27	10.7	8.9	6.2	3.1	2.3	1.9	1.3	0.4			
	LUBM3K	38.26%	39.5	16.1	13.4	9.2	4.6	3.4	2.8	1.9	0.6			
	LUBM4K	38.26%	52.8	21.4	17.9	13.0	6.1	4.6	3.7	2.4	0.8			
12	LUBM1K	38.10%	8.8	5.1	4.5	4.3	1.5	1.3	1.1	1.1	0.9	0.7	0.6	0.2
	LUBM2K	38.10%	17.7	10.1	8.9	8.6	3.1	2.5	2.3	2.1	1.9	1.5	1.3	0.4
	LUBM3K	38.10%	26.6	15.2	13.4	12.9	4.6	3.8	3.4	3.2	2.8	2.2	1.9	0.6
	LUBM4K	38.10%	35.6	20.3	17.9	17.2	6.1	5.1	4.6	4.2	3.7	3.0	2.4	0.8

the HDT technique, which reports linear scalability regarding the input size and the terms in the dictionary (cf. [16]).

For the LUBM dataset we group data based on the rdf:type of resources and use these groupings to generate three different subgraph datasets (the size of each subgraph is shown in Table 2):

- 12 subgraphs, composed of *UnderGraduateStudent* (G_1), *Courses* (G_2), *Publication* (G_3), *GraduateStudent* (G_4), *Department* (G_5), *ResearchAssistant* (G_6), *AssociateProfessor* (G_7), *TeachingAssistant* (G_8), *FullProfessor* (G_9), *AssistantProfessor* (G_{10}), *University* (G_{11}) and *Lecturer* (G_{12}).
- 9 subgraphs, composed of the union of *UnderGraduateStudent* and *GraduateStudent* (G_1), *Courses* (G_2), *Publication* (G_3), the union of *AssistantProfessor*, *ResearchAssistant*, and *TeachingAssistant* (G_4), *Department* (G_5), *AssociatePro-*

fessor (G_6), *FullProfessor* (G_7), *University* (G_8) and *Lecturer* (G_9).

- 6 subgraphs, composed of *UnderGraduateStudent* and *GraduateStudent* (G_1), the union of *AssistantProfessor*, *ResearchAssistant*, *TeachingAssistant*, *Lecturer*, *AssociateProfessor*, *FullProfessor* (G_2), *Courses* (G_3), *Publication* (G_4), *Department* (G_5) and *University* (G_6).

When triples represent relations between resources of different types all incoming/outgoing relations are replicated in both subgraphs.

For DBpedia (in the case of both versions), we generate 6, 9 and 12 subgraphs, each containing randomly selected triples amounting to 10% of the entire corpus (thus ensuring overlaps among subgraphs). Triples that do not appear in any subgraph are subsequently distributed evenly among the subgraphs.

In the case of SAFE, the dataset is already organised in 8 subgraphs, composed of 5 *external* graphs, includ-

ing statistical data from well-known organisation such as Eurostat and FAO, and 3 *internal* graphs including aggregated clinical data represented as RDF data cubes [32].

Given that the complexity of the partitioning is directly related to the number of duplicates across subgraphs, the size of each of the subgraphs and the overall duplicate ratio, as $\frac{(totalTriples - UniqTriples)}{totalTriples}$, is presented in column DUP % of Table 2. Note that the type-based selection of subgraphs in LUBM generates a skewed distribution of subgraph sizes but similar duplicate ratio (of approximately 38%) at increasing sizes (LUBM1K to LUBM4K). Thus, the comparison between techniques focuses on the effect of the 6/9/12 subgraphs and the efficiency at large scale. In contrast, the even distribution of DBpedia is reflected in the similar size of its subgraphs. Given that the number of duplicates increase with the number of subgraphs (12%, 22% and 33% for 6/9/12 respectively), the effect of duplicates is also evaluated. In SAFE, the already given 8 subgraphs contains few repeated triples (less than 0.01%). Note that the *internal* subgraphs corresponds to graphs G_4 , G_5 and G_7 in Table 2, i.e. the public *external* information corresponds to the biggest partitions.

In the following we show the performance results of each of the algorithms (compression and encryption, decryption and decompression, integration and querying). Experiments were performed in a –commodity server– (Intel Xeon E5-2650v2 @ 2.6 GHz, 16 cores, RAM 180 GB, Debian 7.9.). All of the reported (elapsed) times are the average of three independent executions in a cold cache scenario (caches are empty at the start of each process).

6.2. Compression and Encryption

Table 3 shows the compression and encryption times as well as corresponding resulting file sizes¹⁵ of the datasets for different partitioning strategies, whereas Table 4 shows the respective number of resulting dictionary and triple components.

The results show that $HDT_{crypt-C}$ is both the fastest and also produces the most compact representation (only marginally outperformed in space by $HDT_{crypt-D}$ in particular LUBM cases). $HDT_{crypt-C}$ is 37% faster than the baseline approach $HDT_{crypt-A}$ in DBpedia

(we refer to the average in both DBpedia versions hereafter), and 40% faster in LUBM. In SAFE, with few duplicates, $HDT_{crypt-C}$ is still 18% faster.

In contrast, $HDT_{crypt-B}$ is the slowest approach with a mean of 68% over the baseline, because it needs to create many dictionaries (e.g. 3904 in DBpedia 2016-10 as shown in Table 4) with overlapping terms. In turn, $HDT_{crypt-D}$ is highly influenced by the number of dictionary components, due to the additional complexity of creating the resp. triple components from the D -merge. Thus, $HDT_{crypt-D}$ is faster than the baseline in LUBM with 6 or 9 subgraphs, with few components as shown in Table 4, but it shows a worse performance in LUBM 12 subgraphs, as well as in all DBpedia and SAFE datasets.

Note that, as stated in Section 5, the creation of $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$ assumes that the HDT representation of the full graph G is already computed¹⁶. Otherwise, the HDT creation time (reported in Table 1) should be considered as a once-off overhead. In the worst case (i.e. the conversion is done for the sole purpose of encrypting a single dataset with a particular number of subgraphs), adding this time would make the $HDT_{crypt-C}$ perform similarly to the baseline in LUBM. In DBpedia, with a richer dictionary of terms, $HDT_{crypt-C}$ would be 35-50% slower than the baseline.

Additionally, when compared with the baseline approach $HDT_{crypt-A}$, $HDT_{crypt-C}$ achieves a mean of 33% space saving in DBpedia and 26% space saving in LUBM. In general, $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$ benefit from having an increasing number of overlapping dictionaries/triples, hence the DBpedia even distribution produces more space savings. For the same reason, an increasing number of subgraphs leads to more duplicates and space savings w.r.t the baseline, e.g. $HDT_{crypt-C}$ in LUBM achieves 24%, 26% and 27% savings with 6, 9 and 12 subgraphs respectively. It is worth mentioning that despite the fact that $HDT_{crypt-D}$ isolates the non-overlapping dictionaries and triples, there is an overhead in the representation as we do not use Bitmap Triples but Plain Triples (as stated in Section 5.4). This is more noticeable in DBpedia with long predicate and object lists. It is worth highlighting that, in SAFE, with almost no duplicates,

¹⁵Note that encryption produces negligible size overheads on the compressed files.

¹⁶In fact, HDT is becoming popular to store and serve large datasets by publishers and third parties, and a large portion of datasets in the Linked Open Data cloud is already available in HDT thanks to the project LOD Laundromat [3], crawling and serving the HDT conversion of datasets (<http://lodlaundromat.org/wardrobe/>).

Table 3
Performance of compression and encryption algorithms.

SUBGRAPHS	DATASET	Compression Time (minutes)				Encryption Time (seconds)				Size (GB)			
		<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>	<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>	<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>
6	DBpedia 3.8	102	197	62	121	98.17	108.84	80.85	91.07	9.64	9.33	7.43	8.59
	DBpedia 2016-10	306	565	197	378	144.54	143.06	109.95	123.91	18.91	18.38	14.04	16.25
9	DBpedia 3.8	117	225	72	142	124.65	140.40	98.24	125.94	11.64	10.91	7.92	8.76
	DBpedia 2016-10	267	520	175	315	182.61	172.50	113.18	128.33	23.12	21.78	15.00	16.58
12	DBpedia 3.8	131	214	81	152	156.52	245.16	187.90	221.89	13.87	12.49	8.49	8.91
	DBpedia 2016-10	300	485	202	326	228.92	201.26	128.29	138.70	27.79	25.11	16.14	16.88
8	SAFE	9	18	8	14	4.07	4.91	4.27	5.20	0.53	0.61	0.53	0.65
6	LUBM1K	34	41	21	33	12.25	11.21	9.85	10.94	1.40	1.08	1.05	1.05
	LUBM2K	78	94	47	73	21.88	18.74	17.95	18.24	2.86	2.19	2.15	2.16
	LUBM3K	125	143	72	112	32.24	26.82	25.72	26.00	4.35	3.31	3.28	3.30
	LUBM4K	169	191	97	151	54.56	33.83	33.17	33.90	5.65	4.45	4.41	4.45
9	LUBM1K	37	42	21	36	12.96	11.93	11.33	11.89	1.44	1.09	1.06	1.04
	LUBM2K	78	88	45	73	22.80	19.56	18.97	19.98	2.93	2.21	2.17	2.14
	LUBM3K	126	144	71	114	33.79	28.02	27.61	27.58	4.45	3.34	3.31	3.26
	LUBM4K	174	194	98	158	60.11	35.66	35.53	35.36	5.97	4.49	4.44	4.42
12	LUBM1K	36	44	23	39	12.78	13.48	12.21	12.98	1.45	1.11	1.06	1.05
	LUBM2K	75	94	49	82	23.32	21.62	20.23	21.38	2.96	2.25	2.17	2.15
	LUBM3K	116	142	73	126	33.92	29.03	28.35	29.50	4.50	3.41	3.31	3.26
	LUBM4K	158	190	99	175	60.80	37.85	38.20	37.36	6.03	4.56	4.44	4.44

Table 4
Number of dictionaries/triples in each approach.

SUBGRAPHS	DATASET	Dictionaries			Triples	
		<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i> <i>crypt-D</i>	<i>crypt-A</i> <i>crypt-C</i>	<i>crypt-B</i> <i>crypt-D</i>
6	DBpedia 3.8	6	63	63	6	63
9	DBpedia 3.8	9	510	511	9	510
12	DBpedia 3.8	12	3836	4095	12	3836
6	DBpedia 2016-10	6	63	63	6	63
9	DBpedia 2016-10	9	511	511	9	511
12	DBpedia 2016-10	12	3904	4095	12	3904
8	SAFE	8	32	48	8	32
6	LUBM	6	20	23	6	20
9	LUBM	9	39	64	9	39
12	LUBM	12	55	122	12	55

860 only $HDT_{crypt-C}$ is competitive in space with the base-
line, while $HDT_{crypt-B}$ and $HDT_{crypt-D}$ have to pay
a slight overhead for keeping the different structures,
which cannot leverage the minimal duplication across
subgraphs.

865 Encryption times are only a small portion of the
publication process, where $HDT_{crypt-C}$ is generally the
fastest approach except for DBpedia 3.8 with 12 sub-
graphs and SAFE, for which $HDT_{crypt-A}$ is the fastest,
and for LUBM3K/LUBM4K with 9 subgraphs as well
870 as LUBM4K with 12 subgraphs where $HDT_{crypt-D}$ is
marginally faster. Thus we can conclude that both the
885 number of files that need to be encrypted as well as
their respective file sizes influence the overall encryp-
tion time. Finally, it is worth noting that – as expected –
875 the performance time of the compression and encryp-

tion, as well as the result file sizes show linear growth
with increasing LUBM datasets.

6.3. Decryption and Decompression

According to our use case scenario we assume that
a user has been granted access to more than one named
graph, but not the whole dataset. For a fair compar-
ison, given the skewed size distribution of subgraphs in
LUBM (see Table 2), we set up a scenario where the
user has been granted access to half of the total sub-
graphs, including the smallest, average and largest sub-
graphs. This configuration corresponds to decrypting
and decompressing the subgraphs referred to as $M^6 =$
 $\{G_1, G_3, G_6\}$, $M^9 = \{G_1, G_2, G_5, G_8, G_9\}$ and $M^{12} =$
 $\{G_1, G_2, G_6, G_7, G_{11}, G_{12}\}$ in the case of 6, 9 and 12

Table 5

Performance of decryption and decompression algorithms for M^6 , M^9 and M^{12} , i.e., half of the 6/9/12 subgraphs including the smallest/average/largest subgraphs.

SUBGRAPHS	DATASET	Decryption Time (seconds)				Decompression Time (minutes)			
		<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>	<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>
M^6	DBpedia 3.8	61.56	79.08	64.92	79.80	22	18	14	18
	DBpedia 2016-10	108.64	125.36	108.69	127.51	51	46	39	53
M^9	DBpedia 3.8	88.64	148.52	111.84	129.31	26	22	17	25
	DBpedia 2016-10	146.93	200.97	151.41	171.56	49	45	36	50
M^{12}	DBpedia 3.8	93.10	220.46	195.11	242.85	22	22	17	26
	DBpedia 2016-10	160.88	256.05	179.65	206.75	37	34	27	37
M^6	LUBM1K	10.82	11.37	9.80	13.74	8	7	5	7
	LUBM2K	19.24	22.83	17.15	27.62	16	14	11	15
	LUBM3K	28.35	31.65	24.78	45.14	24	20	16	22
	LUBM4K	48.56	43.03	33.70	59.46	32	27	21	29
M^9	LUBM1K	12.84	13.36	11.86	17.52	8	10	6	8
	LUBM2K	22.77	24.47	20.63	33.15	17	21	12	16
	LUBM3K	32.94	37.32	30.30	48.95	26	32	18	23
	LUBM4K	48.00	52.35	51.36	70.12	34	41	24	32
M^{12}	LUBM1K	10.75	11.54	11.73	15.84	7	6	5	7
	LUBM2K	18.50	20.30	18.99	30.40	14	13	10	14
	LUBM3K	26.60	31.08	27.00	45.35	21	19	15	20
	LUBM4K	36.62	39.48	39.09	66.57	29	25	19	27

Table 6

Performance of decryption and decompression algorithms for M_L^8 and M_S^8 in the SAFE dataset.

SUBGRAPHS	DATASET	Decryption Time (seconds)				Decompression Time (seconds)			
		<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>	<i>crypt-A</i>	<i>crypt-B</i>	<i>crypt-C</i>	<i>crypt-D</i>
M_L^8	SAFE	3.98	4.45	4.01	4.70	182	169	118	174
M_S^8	SAFE	1.01	2.75	1.05	2.14	6	74	4	56

890 subgraphs respectively. As for the SAFE dataset, we
 consider a scenario where a subset of the *external* and
internal datasets are accessed. In particular, we also
 took half of the datasets, $M_L^8 = \{G_1, G_4, G_5\}$, in-
 cluding the largest *external* dataset G_1 , and $M_S^8 =$
 895 $\{G_4, G_5, G_6\}$, of smaller size.

Table 5 shows the time to decrypt and decompress
 each of the respective subgraphs in the case of DB-
 920 pedia and LUBM, while Table 6 shows the results for
 SAFE.

900 Decryption times are almost negligible compared
 to the decompression time – similar to encryption vs.
 925 compression time. Again, the number of files is the
 dominating factor, hence $HDT_{crypt-A}$ is the fastest ap-
 proach regarding decryption.

905 Regarding decompression, (as per compression)
 $HDT_{crypt-C}$ is the fastest approach, achieving a mean
 of 30% time savings in DBpedia and LUBM w.r.t the
 930 baseline $HDT_{crypt-A}$. In DBpedia, given the even dis-
 tribution, having 6 subgraphs is always slightly faster
 910 than 9 and 12 subgraphs as the latter generates more
 duplicates. Regarding the number of graphs in LUBM,
 6 and 12 subgraphs behave similarly, while the decom-
 935 pression of 9 subgraphs is slightly slower. Nonethe-

less, we could verify that the difference between 9 and
 12 subgraphs is due to the slightly bigger total file size
 produced by M^9 in comparison to M^{12} . In turn, the
 difference between 9 and 6 subgraphs is a consequence
 of the larger number of generated dictionary/triples be-
 915 tween 9 and 6 subgraphs (as shown in Table 4). As per
 compression, there is a linear increase in performance
 times with increasing dataset sizes.

Finally, although the results for the SAFE dataset
 (shown in Table 6) follow a similar behaviour, it is
 worth mentioning that $HDT_{crypt-B}$ and $HDT_{crypt-D}$
 have to pay the price of loading additional structures
 (even in the presence of minimal duplication). Results
 show that, while this pays off in the case of the larger
 subset like M_L^8 , for a small subset like M_S^8 , $HDT_{crypt-A}$
 and $HDT_{crypt-C}$ are clearly faster than $HDT_{crypt-B}$ and
 935 $HDT_{crypt-D}$.

6.4. Querying Compressed Data

One of the main advantages of HDT compression
 is that it is possible to perform SPARQL triple pattern
 queries directly on the compressed data [37]. Whereas
 this also holds for approach $HDT_{crypt-A}$, as it already

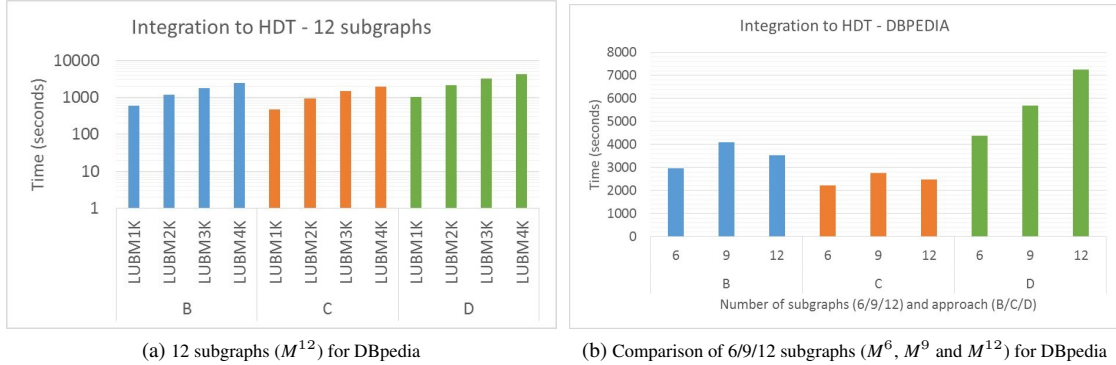


Fig. 10. Integration of the dictionary and triple components of M^6 , M^9 and M^{12} into one HDT per subgraph in DBpedia (average of the performance in both DBpedia versions).

consists of one file per subgraph, the other approaches presented, $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$, split a subgraph in different dictionary (D) and triple (T) components. For these latter approaches, query resolution can be done by two strategies:

1. *Querying an integrated HDT*: This strategy integrates all the dictionary and triple components of a subgraph into a new HDT (i.e. converting to the baseline $HDT_{crypt-A}$) which can be then queried.
2. *Local query on each dictionary and triple component*: In this case, the query is performed locally in each dictionary and triple component and the results are then integrated. Note that $HDT_{crypt-C}$ is not viable for this strategy as it would require to perform the D -union of all the dictionaries in order to search the triples IDs, which is then equivalent to integrating $HDT_{crypt-C}$ into a new HDT to be queried.

The following evaluation first inspects the performance overhead of the integration required by the former strategy. Then, we evaluate the query performance of the latter. For exemplary purposes, we present the average results of the DBpedia datasets, while the performance for LUBM and SAFE can be found in Appendix A.

Note that, although there are a number of strategies for querying *encrypted* data *directly* (see e.g., [4]), we consider these orthogonal and leave combining them with our partitioning for future work.

6.4.1. Integrating dictionary and triple components into a new HDT

Following our use case scenario, we assume that a user has decrypted half of the subgraphs, the i.e. M^6 ,

M^9 and M^{12} subgraphs. Figure 10 shows the time required by each strategy (i.e. $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$) to integrate their dictionary and triple components into one HDT per subgraph (e.g. G_1 , G_2 , G_6 , G_7 , G_{11} and G_{12} for M^{12}), similarly to the baseline $HDT_{crypt-A}$. This integration is performed as follows. First, all dictionary components are fed into a new dictionary, reorganizing the mapping between all terms and their corresponding IDs (as defined in Section 3.1). This first process is similar to the first step of the D -union (see Section 4.3.1). Then, we read the triple components and use the new dictionary to convert the triples to the new IDs, integrating all of them in a single new triple component per subgraph¹⁷.

We present the time to integrate the dictionary and triple components of M^{12} into the corresponding subgraphs (Figure 10 a), for DBpedia. Yet again we see that $HDT_{crypt-C}$ is the fastest approach, 29% and 56% faster than B and D in DBpedia. In general, all approaches show a linear increase over dataset sizes, as shown in Appendix A.

A comparison in terms of number of subgraphs is shown in Figure 10 b, reporting the times of merging M^6 , M^9 and M^{12} for DBpedia (the trends are similar for all datasets). As expected, given that the integration process yields to a partial decompression of the dictionary and triple components, the integration performance follows the same pattern as the decompression. That is, the even distribution of DBpedia results in a faster performance for 6 subgraphs, whereas the excessive duplicates of 12 penalises its performance.

¹⁷Note that this process slightly differs from the D -union as the latter only replaces the new IDs in each of the input triple component.

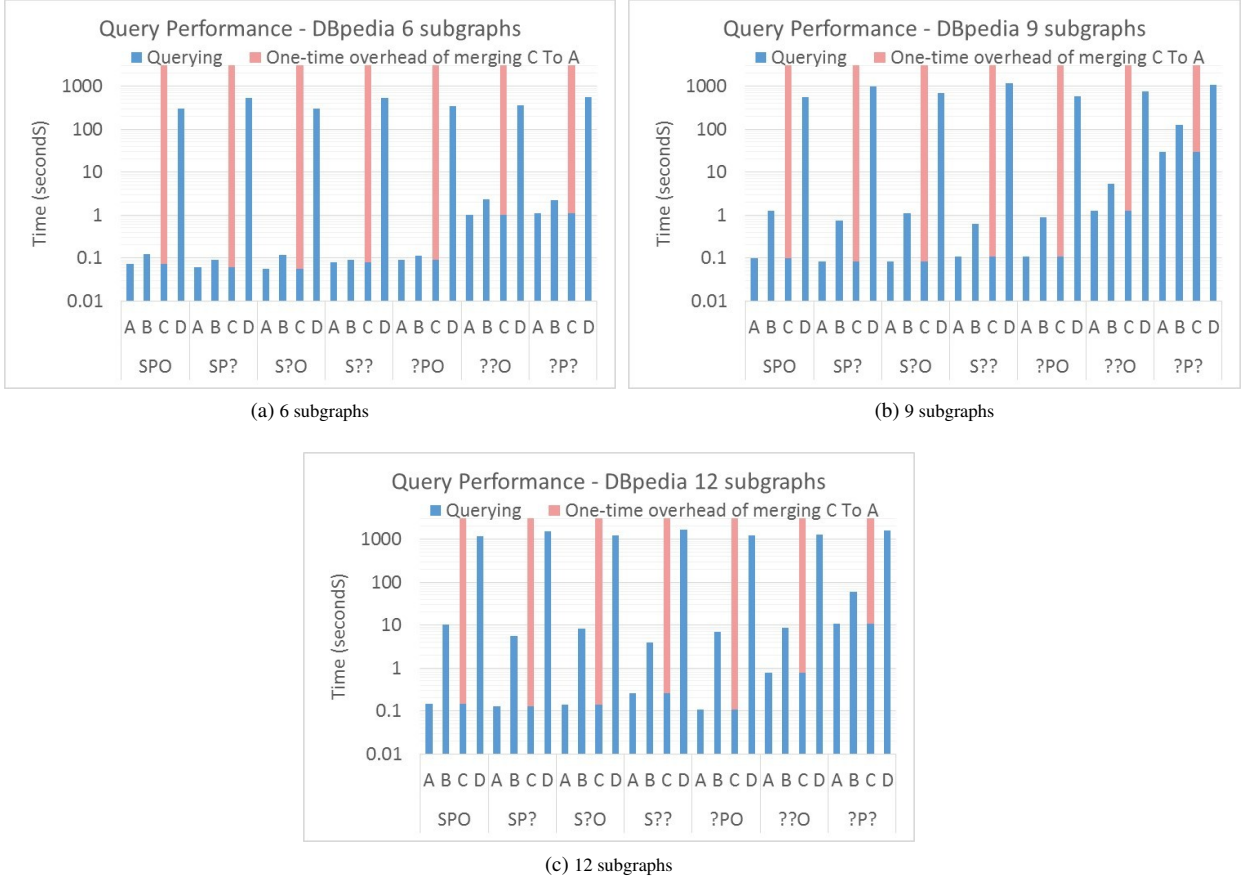


Fig. 11. Performance of Triple Patterns over DBpedia (average of the performance in both DBpedia versions).

1000 6.4.2. Query Performance

We evaluate the query performance of all partitioning strategies in our use case scenario. Thus, for each subgraph in M^6 , M^9 and M^{12} (and M_S^8 and M_L^8 in SAFE) we first generate 30 random queries for each triple pattern type¹⁸, assuring an even presence of different predicates. Figure 11 shows the average execution time of the selected queries for both DBpedia versions (the results for LUBM4K and SAFE are presented in Appendix A). Note that, as shown in the previous section, the integration into a new HDT results in a non-negligible time to perform the process. Thus, for $HDT_{crypt-A}$, $HDT_{crypt-B}$ and $HDT_{crypt-D}$ we follow the strategy where queries are executed locally in each dictionary and triple component. In contrast, query execution in $HDT_{crypt-C}$ would require the D -union of all the dictionaries to be created, which is then equivalent to integrating $HDT_{crypt-C}$ into a new

HDT to be queried. As such, the performance time for $HDT_{crypt-C}$ is presented as the sum of the time taken to create one integrated HDT (performed once), as previously explained in Section 6.4.1, and to subsequently query the integrated HDT (note again that this latter is equivalent to querying $HDT_{crypt-A}$).

Regarding the comparison between our strategies for partitioning, results show that $HDT_{crypt-A}$ and $HDT_{crypt-B}$ have the best performance for all patterns. This can be attributed to the fact that they benefit from efficient Bitmap Triples indexes, while $HDT_{crypt-D}$ must use Plain Triples (as stated in Section 5.4) that perform sequential scans to resolve queries. Note that $HDT_{crypt-D}$ is only competitive in the case of (?p?) queries (i.e. retrieving all subjects and objects related to a given predicate), given that most of the triples are returned and the total time is similar to a full sequential scan. In addition, Bitmap Triples indexes are less efficient for such query types [37]. As previously stated, $HDT_{crypt-C}$ behaves as $HDT_{crypt-A}$ but there is

¹⁸All queries are available at the HDTcrypt repository.

Table 7

Summary of performance of different $\text{HDT}_{\text{crypt}}$ strategies, where $\star \star \star$ stands for the best performance.

Strategy	Comp. & Encryp.			Decryp. & Decomp.	Querying	
	Time	Size	Preconditions	Time	Time	Preconditions
<i>crypt-A</i>	$\star \star$	\star	None	\star	$\star \star \star$	None
<i>crypt-B</i>	\star	$\star \star$	HDT of full graph G	$\star \star$	$\star \star$	None
<i>crypt-C</i>	$\star \star \star$	$\star \star \star$	HDT of full graph G	$\star \star \star$	$\star \star$	Once-off integration to a new HDT
<i>crypt-D</i>	\star	$\star \star \star$	HDT of full graph G	$\star \star$	\star	None

Table 8

Influence of the increasing number of subgraphs and duplicates in the performance of different $\text{HDT}_{\text{crypt}}$ strategies, where $+++$ stands for very positive and $---$ for very negative.

Strategy	Comp. & Encryp.		Decryp. & Decomp.	Querying
	Time	Size	Time	Time
<i>crypt-A</i>	—	—	—	—
<i>crypt-B</i>	—	$+++$	—	—
<i>crypt-C</i>	—	$+++$	—	—
<i>crypt-D</i>	$---$	$++$	—	$---$

a once-off overhead associated with merging all dictionary and triple components into one HDT (represented in red in Figure 14).

In turn, it is also worth mentioning that $\text{HDT}_{\text{crypt-B}}$ query performance is closer to the baseline $\text{HDT}_{\text{crypt-A}}$ in the scenario with 6 subgraphs. This is mainly due to the larger number of dictionaries/triples to be queried in a scenario with a higher number of subgraphs (as shown in Table 4), which penalises the $\text{HDT}_{\text{crypt-B}}$ and $\text{HDT}_{\text{crypt-D}}$ methods. In this scenario, $\text{HDT}_{\text{crypt-A}}$ is the most efficient approach for query execution. The noticeable performance difference against the rest of the partitioning approaches suggests that the once-off merging that is required for $\text{HDT}_{\text{crypt-C}}$ can be amortised if the dataset is meant for intensive querying after decryption.

6.5. Discussion of the results

Overall, our empirical evaluation showed interesting results and allows us to draw conclusions on the applicability of each strategy. We summarize a ranking of results for each scenario in Table 7, and we outline the influence of the increasing number of subgraphs and duplicates in the data in Table 8, detailed as follows:

- $\text{HDT}_{\text{crypt-C}}$ is the most effective technique in terms of compression and decompression times, as well as compression sizes. In particular, it achieves additional 26-33% space saving over the –already compressed– baseline ($\text{HDT}_{\text{crypt-A}}$), and it is 37-40% faster to compress, and 30% faster to decompress. Note that the impact of these space and time savings are even more noticeable when dealing with big data.

As we noticed, if the original HDT of the full graph is not available beforehand, then the creation of $\text{HDT}_{\text{crypt-C}}$ can take more time than the baseline (it results in approx. the same time in LUBM and 35-50% slower in DBpedia, with a rich dictionary of terms), but it keeps the aforementioned noticeable space savings. In the extreme case of isolated subgraphs with few duplicates, as in SAFE, $\text{HDT}_{\text{crypt-C}}$ takes the same space as the baseline and is still 18% faster to encrypt.

- In contrast, $\text{HDT}_{\text{crypt-C}}$ does not allow the user to directly query the exchanged information. If such a scenario is required, this can be solved with a once-off conversion from $\text{HDT}_{\text{crypt-C}}$ to $\text{HDT}_{\text{crypt-A}}$. This conversion can be done for any strategy, but it is indeed faster for $\text{HDT}_{\text{crypt-C}}$.
- $\text{HDT}_{\text{crypt-B}}$ and $\text{HDT}_{\text{crypt-D}}$ also reduce the size of the baseline ($\text{HDT}_{\text{crypt-A}}$), and can be directly queried. Results show that $\text{HDT}_{\text{crypt-B}}$ and $\text{HDT}_{\text{crypt-D}}$ gain 6-24% and 24-26% space over the baseline respectively, at the cost of an extra 68% and 23% time for compression (performed only once by the data publisher). In turn, the decompression time outperforms the baseline by 7% and 9% for $\text{HDT}_{\text{crypt-B}}$ and $\text{HDT}_{\text{crypt-D}}$ respectively. In the extreme case of isolated subgraphs with few duplicates, as in SAFE, $\text{HDT}_{\text{crypt-B}}$ and $\text{HDT}_{\text{crypt-D}}$ suffer from a slight space overhead (15-23%) over the baseline, and non negligible additional decompressing times.
- The performance of directly querying several subgraphs in $\text{HDT}_{\text{crypt-B}}$ is close to the baseline $\text{HDT}_{\text{crypt-A}}$. Nonetheless, it is penalised at larger

number of partitions (such as 12 in our experiments) and larger number of duplicates (such as our even distribution in DBpedia). $HDT_{crypt-D}$ suffers from the additional problem of performing sequential scans, and is not competitive but for queries that retrieve large number of results.

– Encryption and decryption times are almost negligible compared to the compression/decompression counterparts.

– Compression sizes, compression and decompression times show linear growth with increasing dataset size.

– In general, an increasing number of subgraphs leads to more duplicates and more space savings of our novel $HDT_{crypt-B}$, $HDT_{crypt-C}$ and $HDT_{crypt-D}$ partitioning approaches over the baseline $HDT_{crypt-A}$. In turn, less data file sizes result in faster decompression of our novel approaches. In contrast, the compression time is penalised given that more components have to be generated. Our experiments also showed that the number of subgraphs does not have a strong influence on the query performance, but the skewed distribution of sizes and the large number of components (such as in DBpedia) can result in slight differences between scenarios.

7. Conclusions and Future Work

To date Linked Data publishers have focused on exposing and linking *open* data, however the Linked Data infrastructure could be extended to cater for the storage and exchange of confidential data. In this paper, we discussed how HDT compression can be extended to cater for RDF datasets which needs to be encrypted. Specifically, we proposed a number of different compression strategies that are compatible and demonstrated the need for careful integration when it comes to compressed encrypted RDF data. From our evaluation we can see that our proposal $HDT_{crypt-C}$ outperforms the other partitioning strategies both in terms of compression and decompression time, and it also produces the most compact representation, resulting in 26-31% space savings over the –already compressed– baseline. $HDT_{crypt-B}$ and $HDT_{crypt-D}$ also reduce the size of the baseline significantly. Whereas, when it comes to querying $HDT_{crypt-A}$ and $HDT_{crypt-B}$ outperform $HDT_{crypt-C}$, which incurs additional overhead as the dictionaries and triples need to be integrated in order to support querying. Additionally, we note that

compression, decompression and query performance is influenced both by the number of access restricted subgraphs and the distribution of triples across subgraphs, especially in $HDT_{crypt-D}$. In future work, we plan to extend our existing work to cater for querying over encrypted compressed data without the need for decryption. Our current work considers basic triple pattern resolution, while the HDT approach can be used as the basic engine to resolve full SPARQL queries. Our plan is to support this possibility on the compressed and encrypted data in future work.

Acknowledgements

Supported by the European Union’s Horizon 2020 research and innovation programme under grant 731601, Austrian Science Fund (FWF): M1720-G11, the Austrian Research Promotion Agency (FFG) under grant 845638 (SHAPE) and grant 861213 (CitySpin), and MINECO-AEI/FEDER-UE ETOME-RDFD3:TIN2015-69951-R. Axel Polleres was supported by the “Distinguished Visiting Austrian Chair” program as a visiting professor hosted at The Europe Center and the Center for Biomedical Research (BMIR) at Stanford University. Special thanks to Ratnesh Sahay for providing the SAFE dataset.

References

- [1] S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto and G. Navarro, Compressed Vertical Partitioning for Efficient RDF Management, *Knowl. Inf. Syst.* **44**(2) (2015), 439–474.
- [2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, Dbpedia: A nucleus for a web of open data, in: *Proc. of ISWC*, 2007.
- [3] W. Beek, L. Rietveld, H.R. Bazoobandi, J. Wielemaker and S. Schlobach, LOD laundromat: a uniform way of publishing other people’s dirty data, in: *Proc. of ISWC*, LNCS, Vol. 8796, Springer, 2014, pp. 213–228.
- [4] M. Bellare, A. Boldyreva and A. O’Neill, Deterministic and efficiently searchable encryption, in: *Proc. of CRYPTO*, Springer, 2007, pp. 535–552.
- [5] D. Brickley, R.V. Guha and (eds.), RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recomm., W3C, 2004. <http://www.w3.org/TR/rdf-schema/>.
- [6] N. Brisaboa, R. Cánovas, F. Claude, M.A. Martínez-Prieto and G. Navarro, Compressed String Dictionaries, in: *Proc. of SEA*, 2011, pp. 136–147.
- [7] M. Chase and E. Shen, Pattern Matching Encryption., *IACR Cryptology ePrint Archive* **2014/638** (2014).

- [8] L. Costabello, S. Villata, N. Delaforge and F. Gandon, Linked Data Access Goes Mobile: Context-Aware Authorization for Graph Stores, in: *5th WWW Workshop on Linked Data on the Web*, 2012.
- [9] O. Curé, G. Blin, D. Revuz and D.C. Faye, WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store, in: *Proc. of ESWC*, LNCS, Vol. 8465, 2014, pp. 302–316.
- [10] R. Curtmola, J. Garay, S. Kamara and R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: *Proc. of CSS*, ACM, 2006, pp. 79–88.
- [11] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, Springer, 2013. ISBN 3540425802.
- [12] J.D. Fernández, A. Polleres and J. Umbrich, Towards Efficient Archiving of Dynamic Linked Open Data, in: *Proc. of DIACHRON*, 2015.
- [13] J.D. Fernández, S. Kirrane, A. Polleres and S. Steyskal, Self-Enforcing Access Control for Encrypted RDF, in: *Proc. of ESWC*, 2017.
- [14] J.D. Fernández, Binary RDF for Scalable Publishing, Exchanging and Consumption in the Web of Data, PhD thesis, University of Valladolid, Spain, 2014.
- [15] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez and A. Polleres, *Binary RDF Representation for Publication and Exchange (HDT)*, W3C Member Submission, 2011. <https://www.w3.org/Submission/HDT/>.
- [16] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres and M. Arias, Binary RDF Representation for Publication and Exchange, *J. Web Semant.* **19** (2013), 22–41.
- [17] J.D. Fernández, M. Arias, M.A. Martínez-Prieto and C. Gutiérrez, Management of Big Semantic Data, in: *Big Data Computing*, Taylor and Francis/CRC, 2013, Chap. 4.
- [18] C. Gentry, Fully homomorphic encryption using ideal lattices., in: *Proc. of STOC*, Vol. 9, ACM, 2009, pp. 169–178.
- [19] S. Gerbracht, Possibilities to Encrypt an RDF-Graph, in: *Proc. of ICTTA*, IEEE, 2008, pp. 1–6.
- [20] M. Giereth, On Partial Encryption of RDF-Graphs., in: *Proc. of ISWC*, LNCS, Vol. 3729, Springer, 2005, pp. 308–322. ISBN 3-540-29754-5. <http://dblp.uni-trier.de/db/conf/semweb/iswc2005.html#Giereth05>.
- [21] M. Giereth, PRE4J - A Partial RDF Encryption API for Jena, *ACAD MED* **70**(3) (2006), 216–223. <http://jena.hpl.hp.com/juc2006/proceedings/giereth/paper.pdf>.
- [22] J.M. Giménez-García, J.D. Fernández and M.A. Martínez-Prieto, HDT-MR: A Scalable Solution for RDF Compression with HDT and MapReduce, in: *Proc. of ESWC*, 2015, pp. 253–268.
- [23] Y. Guo, Z. Pan and J. Heflin, LUBM: A Benchmark for OWL Knowledge Base Systems, *JWS* **3**(2) (2005), 158–182.
- [24] C. Gutiérrez, C. Hurtado, A.O. Mendelzon and J. Perez, Foundations of Semantic Web Databases, *JCSS* **77** (2011), 520–541.
- [25] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C Recomm., W3C, 2013. <http://www.w3.org/TR/sparql11-query/>.
- [26] A. Hernández-Illera, M.A. Martínez-Prieto and J.D. Fernández, Serializing RDF in compressed space, in: *Data Compression Conference (DCC)*, 2015, IEEE, 2015, pp. 363–372.
- [27] A. Hogan, M. Arenas, A. Mallea and A. Polleres, Everything You Always Wanted to Know About Blank Nodes, *Journal of Web Semantics (JWS)* (2014), 42–69.
- [28] A. Joshi, P. Hitzler and G. Dong, Logical Linked Data Compression, in: *Proc. of ESWC*, LNCS, Vol. 7882, Springer, 2013, pp. 170–184.
- [29] L. Kagal, T. Finin and A. Joshi, A Policy Based Approach to Security for the Semantic Web, in: *Proc. of ISWC*, 2003.
- [30] A. Kasten, A. Scherp, F. Armknecht and M. Krause, Towards search on encrypted graph data, in: *Proc. of PrivOn'14*, Vol. 1121, CEUR-WS.org, 2013, pp. 46–57.
- [31] J. Katz, A. Sahai and B. Waters, Predicate encryption supporting disjunctions, polynomial equations, and inner products, *J. Cryptology* (2013), 1–34.
- [32] Y. Khan, M. Saleem, M. Mehdi, A. Hogan, Q. Mehmood, D. Rebbholz-Schuhmann and R. Sahay, SAFE: SPARQL Federation over RDF Data Cubes with Access Control, *Journal of biomedical semantics* **8**(1) (2017), 5.
- [33] P. Kolari, L. Ding, G. Shashidhara, A. Joshi, T. Finin and L. Kagal, Enhancing Web Privacy Protection through Declarative Policies, in: *Proc. of POLICY*, 2005.
- [34] V. Kolovski, J. Hendler and B. Parsia, Analyzing Web Access Control Policies, in: *Proc. of WWW*, 2007.
- [35] S. Maneth and F. Peternek, Grammar-based graph compression, *Information Systems* **76** (2018), 19–45, ISSN 0306-4379.
- [36] F. Manola and R. Miller, *RDF Primer*, W3C Recomm., 2004, www.w3.org/TR/rdf-primer/.
- [37] M.A. Martínez-Prieto, M. Arias and J.D. Fernández, Exchange and Consumption of Huge RDF Data, in: *Proc. of ESWC*, LNCS, Vol. 7295, Springer, 2012, pp. 437–452.
- [38] M.A. Martínez-Prieto, J.D. Fernández and R. Cánovas, Querying RDF Dictionaries in Compressed Space, *SIGAPP Appl. Comput. Rev.* **12**(2) (2012), 64–77.
- [39] J.Z. Pan, J.M. Gómez-Pérez, Y. Ren, H. Wu and M. Zhu, SSP: Compressing RDF data by Summarisation, Serialisation and Predictive Encoding, Technical Report, K-Drive, 2014. http://www.kdrive-project.eu/wp-content/uploads/2014/06/WP3-TR2-2014_SSP.pdf.
- [40] S. Pearson and M.C. Mont, Sticky policies: an approach for managing privacy across multiple parties, *Computer* **44**(9) (2011), 60–68.
- [41] R. Pichler, A. Polleres, S. Skritek and S. Woltran, Complexity of redundancy detection on RDF graphs in the presence of rules, constraints, and queries, *SWJ* **4**(4) (2013).
- [42] R. Popa, N. Zeldovich and H. Balakrishnan, Cryptdb: A Practical Encrypted Relational dbms., Technical Report, MIT-CSAIL-TR-2011-005, 2011. <http://hdl.handle.net/1721.1/60876>.
- [43] E. Rissanen, Extensible access control markup language (xacml) version 3.0, OASIS Standard, available at <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html>, OASIS Committee Specification, 2013.
- [44] O. Sacco, A. Passant and S. Decker, An Access Control Framework for the Web of Data, in: *Proc. TrustCom*, 2011, pp. 456–463.
- [45] S. Steyskal and S. Kirrane, If you can't enforce it, contract it: Enforceability in Policy-Driven (Linked) Data Markets, in: *Proc. of the Posters and Demos Track SEMANTiCS2015*, 2015.
- [46] Q. Tang, On Using Encryption Techniques to Enhance Sticky Policies Enforcement, Technical Report, CTIT University of Twente, 2008. <http://eprints.eemcs.utwente.nl/14262/>.

- [47] H. Van de Sompel, R. Sanderson, M.L. Nelson, L.L. Balakireva, H. Shankar and S. Ainsworth, An HTTP-based versioning mechanism for linked data, in: *Proc. of LDOW*, 2010.

Appendix A. Additional Performance Results

This appendix comprises the performance results for all datasets. See Section 6 for a description of the corpus and the complete discussion of results.

A.1. Integrating dictionary and triple components into a new HDT

1325 Figure 12 shows the time (in seconds) to integrate the dictionary and triples components of half of the partitions (M^6 , M^9 and M^{12} as explained in Section 6) of LUBM into a single HDT per subgraph. We present the time to integrate the dictionary and triple components of M^{12} into the corresponding subgraphs (Figure 12 a), and a comparison in terms of number of subgraphs (Figure 10 b). Figure 13 shows the integration of the SAFE dataset for the two scenarios, M_L^8 (left) and M_S^8 (right).

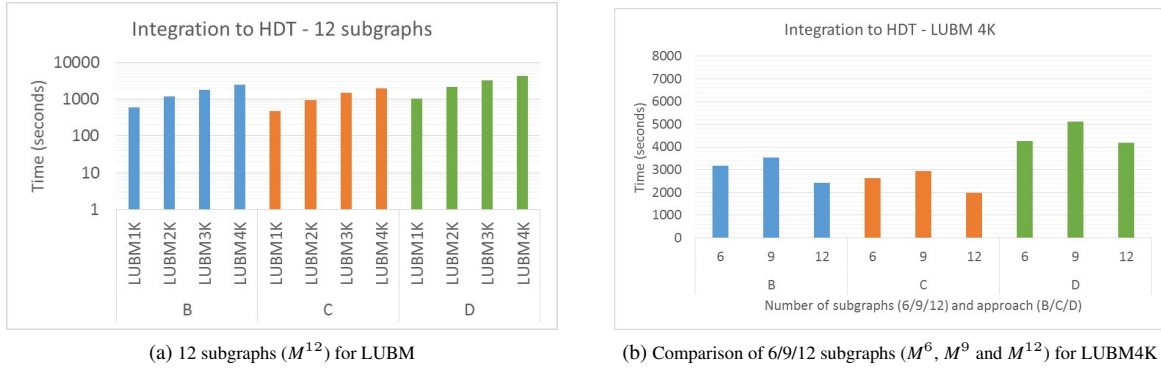


Fig. 12. Integration of the dictionary and triple components of M^6 , M^9 and M^{12} into one HDT per subgraph.

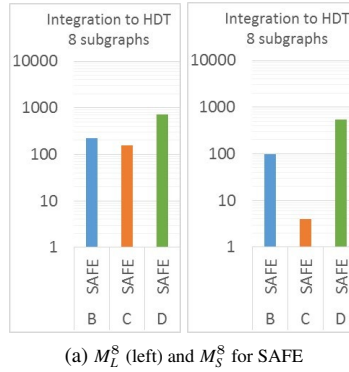


Fig. 13. Integration of the dictionary and triple components of M_S^8 and M_L^8 into one HDT per subgraph.

1330 A.2. Querying Compressed Data

1335 Figures 14 and 15 show the performance of the selected Triple Patterns over LUBM4K and SAFE, respectively. Results for smaller datasets of LUBM4K follow the same trends. As in the case of DBpedia, presented in Section 6, results show that $HDT_{crypt-A}$ and $HDT_{crypt-B}$ have the best performance for all patterns, outperforming its results when few triples are returned, such as (spo) and (sp?) queries. Note that, although $HDT_{crypt-B}$ has to query more dictionaries and triple components than $HDT_{crypt-A}$, the number of total components is very limited in LUBM (the

number of components is shown in Table 4) and each component is smaller in $HDT_{crypt-B}$ than in $HDT_{crypt-A}$. For instance, the resolution of a (sp?) pattern using $HDT_{crypt-A}$ for M^{12} in LUBM4K (see performance results in Figure 14 a) has to query 6 large triple components (one per subgraph), where duplicated triples can be present. In contrast, for $HDT_{crypt-B}$ we could verify that there are 37 triple components in M^{12} , but they are smaller and triples do not overlap. As for SAFE, note that the dataset is particularly small and has few overlapping triples, hence the techniques performance similarly, except for the aforementioned additional overheads in $HDT_{crypt-D}$.

1340

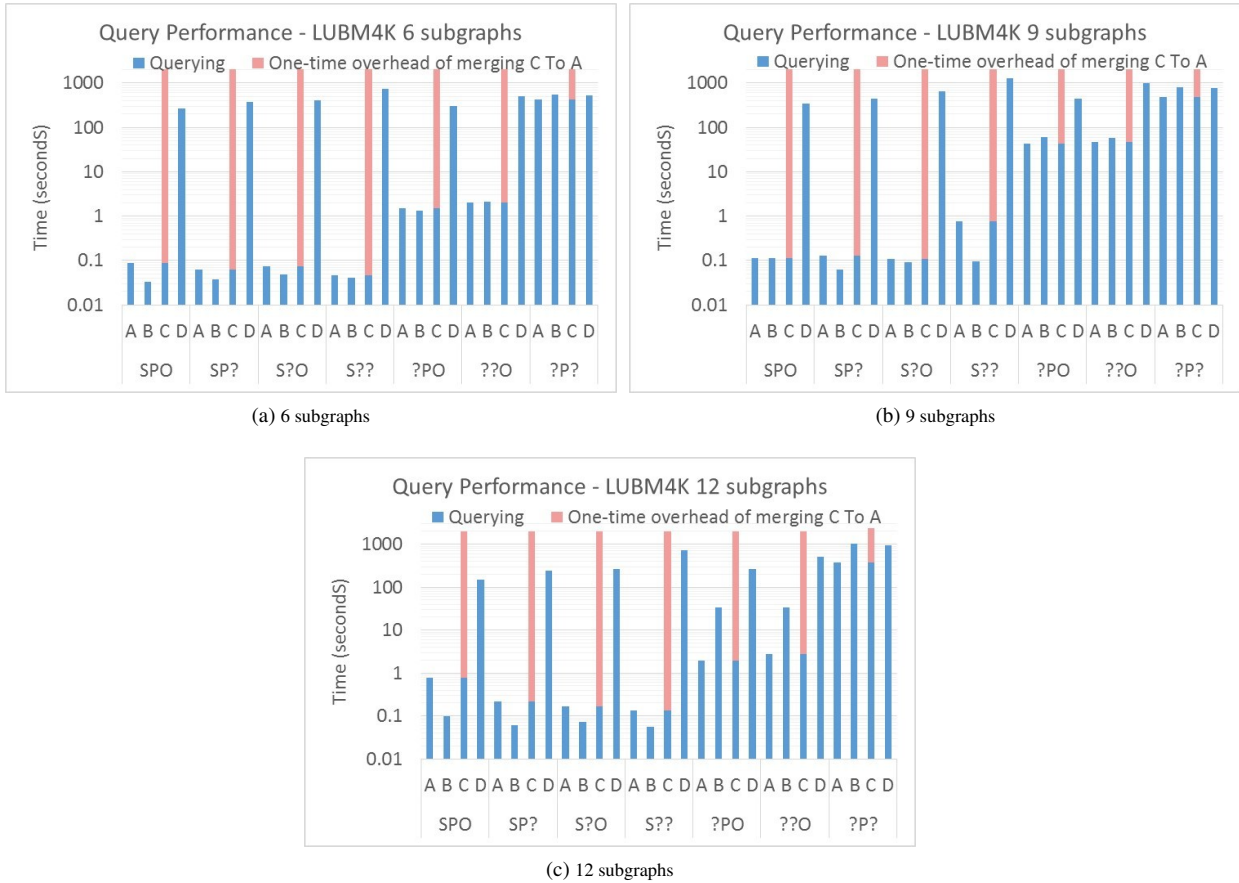


Fig. 14. Performance of Triple Patterns over LUBM4K.

Finally, Figure 16 presents the results of a particular scenario designed to evaluate the potential influence of the number of graphs in a fair manner. Note that, in the previous use case, the number of results could differ in each subgraph as M^6 , M^9 and M^{12} include different subgraphs (e.g. *ResearchAssistant* is included as G_6 in M^{12} but it is present neither in M^9 nor M^6). This fact hampers a fair comparison of the query performance, given that the number of results could differ. This situation is even worse in DBpedia, where each subgraph contains randomly selected triples. Thus, for this particular comparison, we select the *University* subgraph in LUBM, which is present in M^{12} (as G_{11} in Table 2), M^9 (as G_8) and M^6 (as G_6). We then generate 30 random triple pattern queries of each type (similarly to the previous scenario) and perform such queries on the aforementioned *University* subgraph. Figure 16 reports the total performance of all queries for LUBM4K (results are similar for smaller sizes). Note that $HDT_{crypt-A}$ reports the same time in all cases and they compress the same subgraph. In general, results are in line with the previous observations regarding the influence of subgraphs for decompression. That is, in general, 12 subgraphs is the fastest approach, whereas the larger size of the files and their duplication ratio place also a burden on the query performance of 9 subgraphs. Nonetheless, we can find a minor difference in $HDT_{crypt-D}$, where the

1345

1350

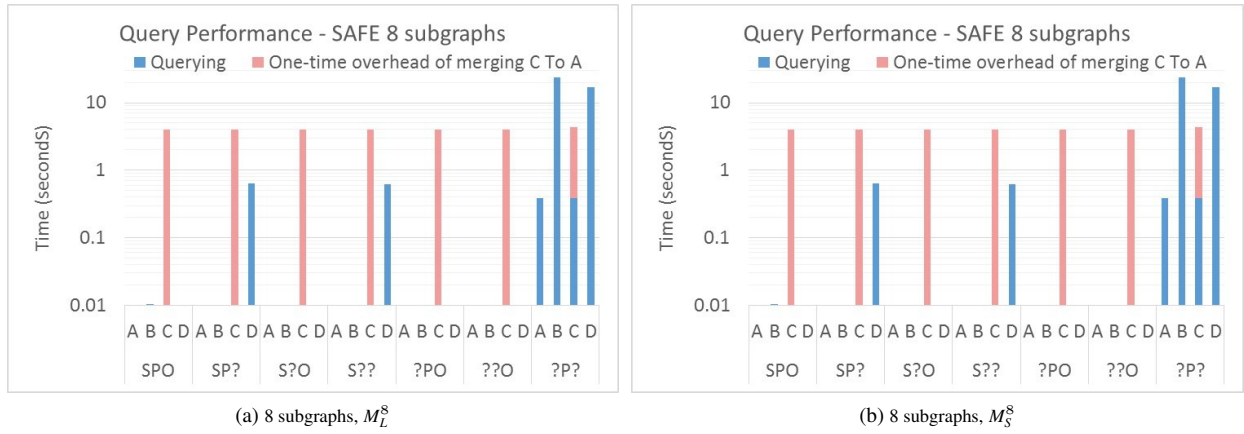
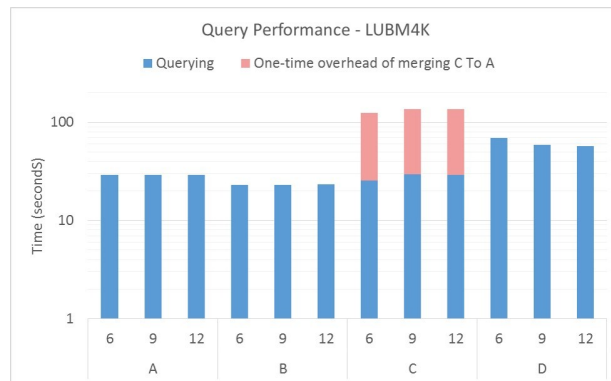


Fig. 15. Performance of Triple Patterns over SAFE.

1355 case of 6 subgraphs reports the worst performance. A closer look at the generated dictionary and triple components
 for the particular *University* subgraph allows us to conclude that this particular case produced a skewed distribution
 of sizes in 6 subgraphs. For example, the largest dictionary component takes 75MB, whereas it is only 27MB and
 12MB for 9 and 12 subgraphs respectively. Note that although this skewed distribution is also present in DBpedia,
 in practice, $HDT_{crypt-D}$ can be slower with 12 subgraphs than with 6 subgraphs, given that the much larger number
 1360 of dictionary and triple components in 12 subgraphs (due to the duplication ratio) are the predominant factor.

Fig. 16. Performance of all Triple Patterns over LUBM4K in the *University* subgraph.