Exchange and Consumption of Huge RDF Data

Miguel A. Martínez-Prieto[1,2], Mario Arias[1,3], and Javier D. Fernández[1,2]

[1] Department of Computer Science, Universidad de Valladolid (Spain)
[2] Department of Computer Science, Universidad de Chile (Chile)
[3] Digital Enterprise Research Institute, National University of Ireland Galway
{migumar2,jfergar}@infor.uva.es,mario.arias@deri.org

Abstract. Huge RDF datasets are currently exchanged on textual RDF formats, hence consumers need to post-process them using RDF stores for local consumption, such as indexing and SPARQL query. This results in a painful task requiring a great effort in terms of time and computational resources. A first approach to lightweight data exchange is a compact (binary) RDF serialization format called `HDT`. In this paper, we show how to enhance the exchanged HDT with additional structures to support some basic forms of SPARQL query resolution without the need of "unpacking" the data. Experiments show that i) with an exchanging efficiency that outperforms universal compression, ii) post-processing now becomes a fast process which iii) provides competitive query performance at consumption.

1 Introduction

The amount and size of published RDF datasets has dramatically increased in the emerging Web of Data. Publication efforts, such as Linked Open Data[4] have "democratized" the creation of such structured data on the Web and the connection between different data sources [7]. Several research areas have emerged alongside this; RDF indexing and querying, reasoning, integration, ontology matching, visualization, etc. A common Publication-Exchange-Consumption workflow (Figure 1) is involved in almost every application in the Web of Data.

Publication. After RDF data generation, publication refers to the process of making RDF data publicly available for diverse purposes and users. Besides RDF publication with dereferenceable URIs, data providers tend to expose their data as a file to download (RDF dump), or via a SPARQL endpoint, a service which interprets the SPARQL query language [2].

Exchange. Once the consumer has discovered the published information, the exchange process starts. Datasets are serialized in traditional plain formats (*e.g.* RDF/XML [5], N3 [4] or Turtle [3]), and universal compressors (e.g. gzip) are commonly applied to reduce their size.

Consumption. The consumer has to post-process the information in several ways. Firstly, a decompression process must be performed. Then, the serialized RDF must be parsed and indexed, obtaining a data structure more suitable for tasks such as browsing and querying.

[4] `http://linkeddata.org/`

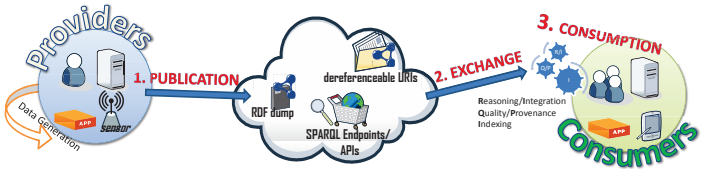


Fig. 1. Publication-Exchange-Consumption workflow in the Web of Data.

The scalability issues of this workflow arise in the following **running example**. Let us suppose that you publish a huge RDF dataset like Geonames (112 million triples about geographical entities). Plain data take up 12.07 GB (in Ntriples[5]), and compression should be applied. For instance, its gzip-compressed dump takes 0.69 GB. Thus, compression is necessary for efficient exchange (in terms of time) when managing huge RDF. However, after decompression, data remain in a plain format and an intensive post-processing is required[6]. Even when data is shared through a SPARQL endpoint, some queries can return large amounts of triples, hence the results must be compressed too.

Nowadays, the potential of huge RDF is seriously underexploited due to the large space they take up, the powerful resources required to process them, and the large consumption time. Similar problems arise when managing RDF in mobile devices; although the amount of information could be potentially smaller, these devices have more restrictive requirements for transmission costs/latency, and for post-processing due to their inherent memory and CPU constraints [14]. A first approach to lighten this workflow is a binary RDF serialization format, called `HDT` (*Header-Dictionary-Triples*) [11], recently accepted as a **W3C Member Submission** [6]. This proposal highlights the need to move forward plain RDF syntaxes to a data-centric view. `HDT` modularizes the data and uses the skewed structure of RDF graphs [9] to achieve compression. In practical terms, `HDT`-based representations take up to 15 times less space than traditional RDF formats [11].

Whereas publication and exchange were partially addressed in `HDT`, the consumption is underexploited; `HDT` provides basic retrieval capabilities which can be used for limited resolution of SPARQL triple patterns. This paper revisits these capabilities for speeding up consumption within the workflow above. We propose i) to publish and exchange RDF serialized in `HDT`, and then ii) to perform a lightweight post-process (at consumption) enhancing the `HDT` representation with additional structures providing a full-index for RDF retrieval. The resulting enhanced `HDT` representation (referred to as `HDT-FoQ`: *HDT Focused on Querying*) enables the exchanged RDF to be directly queryable with SPARQL, speeding up the workflow in several correlated dimensions:

- RDF datasets are exchanged in compact `HDT`, reducing transmission costs.
- `HDT-FoQ` is built on top of `HDT`, requiring little post-processing. It excels in consumption latency (the time awaited until the dataset can be consumed).
- `HDT-FoQ` provides efficient in-memory resolution of triple patterns and joins.

[5] `http://www.w3.org/TR/rdf-testcases/#ntriples`

[6] *Post-processing* is the computation needed at consumption (parsing+indexing) before any query can be issued.

Our experimental results report figures on each of the achievements above. In particular, our HDT-driven approach completes the workflow $10-15$ times faster than traditional solutions, outperforming them in the three processes. Query performance evaluation shows that i) the resultant indexed HDT-FoQ achieves the best overall performance for triple patterns resolution, and ii) an ad-hoc join implementation on top of HDT-FoQ reports competitive results with respect to optimized solutions within the state-of-the-art.

This paper is structured as follows. Section 2 reviews the state-of-the-art and sets HDT foundations. In Section 3, HDT is revisited for basic consumption, and Section 4 describes how HDT-FoQ enhances it to achieve efficient SPARQL resolution. Section 5 provides experimental results about the impact of HDT in the current scenario. Finally, Section 6 concludes and devises future work.

2 State-of-the-art

Huge RDF datasets are currently serialized in verbose formats (RDF/XML [5], N3 [4] or Turtle [3]), originally designed for a document-centric Web. Although they compact some constructions, they are still dominated by a human-readable view which adds an unnecessary overhead to the final dataset representation. It increases transmission costs and delays final data consumption within the Publication-Exchange-Consumption workflow.

Besides serialization, the overall performance of the workflow is determined by the efficiency of the external tools used for post-processing and consuming huge RDF. Post-processing transforms RDF into any binary representation which can be efficiently managed for specific consumption purposes. Although it is performed once, the amount of resources required for it may be prohibitive for many potential consumers; it is specially significative for mobile devices comprising a limited computational configuration.

Finally, the consumption performance is determined by the mechanisms used for access and retrieval RDF data. These are implemented around the SPARQL [2] foundations and their efficiency depends on the performance yielded by RDF indexing techniques. Relational-based solutions such as Virtuoso [10] are widely accepted and used to support many applications consuming RDF. On the other hand, some stores build indexes for all possible combinations of elements in RDF (SPO, SOP, PSO, POS, OPS, OSP), allowing i) all triple patterns to be directly resolved in the corresponding index, and ii) the first join step within a BGP to be resolved through fast merge-join. Hexastore [18] performs a memory-based implementation which, in practice, is limited by the space required to represent and manage the index replication. RDF-3X [17] performs multi-indexing on a disk-resident solution which compresses the indexes within B^+-trees. Thus, RDF-3X enables the management of larger datasets at the expense of overloading querying processes with expensive I/O transferences.

Speeding up consumption within this workflow is influenced by two factors: i) the RDF serialization format, as it should be compact for exchanging and friendly for consumption, and ii) efficient RDF retrieval. Scalability issues underlying to these processes justify the need for a binary RDF format like HDT [6].

2.1 Binary RDF Representation (HDT)

HDT is a binary serialization format which organizes RDF data in three logical components. The **Header** includes logical and physical metadata describing the RDF dataset and serves as an entry point to its information. The **Dictionary** provides a catalog of the terms used in the dataset and maps them to unique integer IDs. It enables terms to be replaced by their corresponding IDs and allows high levels of compression to be achieved. The **Triples** component represents the pure structure of the underlying graph after the ID replacement.

Publication and exchange processes are partially addressed by HDT. Although it is a machine-oriented format, the Header gathers human-friendly textual metadata such as the provenance, size, quality, or physical organization (subparts and their location). Thus, it is a mechanism to discover and filter published datasets. In turn, the Dictionary and Triples partition mainly aims at efficient exchange; it reduces the inherent redundancy to an RDF graph by isolating terms and structure. This division has proved effective in RDF stores [17].

3 Revisiting HDT for Basic Consumption

HDT allows different implementations for the dictionary and the triples. Besides achieving *compression*, some implementations can be optimized to support native data retrieval. The original HDT proposal [11] gains insights into this issue through a triples implementation called Bitmap Triples (BT). This section firstly gives basic notions of succinct data structures, and then revisits BT emphasizing how these structures can allow basic consumption.

3.1 Succinct Data Structures

Succinct data structures [16] represent data using as little space as possible and provide direct access. These savings allow them to be managed in faster levels of the memory hierarchy, achieving competitive performance. They provide three primitives ($\mathcal{S}$ is a sequence of length n from an alphabet Σ):

- $\texttt{rank}_a(\mathcal{S}, i)$ counts the occurrences of $a \in \Sigma$ in $\mathcal{S}[1, i]$.
- $\texttt{select}_a(\mathcal{S}, i)$ locates the position for the i-th occurrence of $a \in \Sigma$ in $\mathcal{S}$.
- $\texttt{access}(\mathcal{S}, i)$ returns the symbol stored in $\mathcal{S}[i]$.

In this paper, we make use of succinct data structures for representing sequences of symbols. We distinguish between binary sequences (*bitsequences*) and general sequences. i) **Bitsequences** are a special case drawn from $\Sigma = \{0, 1\}$. They can be represented using $n + o(n)$ bits of space while answering the three previous operations in constant time. We use the implementation of González, *et al.* [12] which takes, in practice, 37.5% extra space on top of the original bitsequence size. ii) **General sequences** are represented using *wavelet trees* [13]. A wavelet tree represents a general sequence as a balanced tree of height $h = \log \sigma$, comprising a total of h bitsequences of n bits. It uses $n \log \sigma + o(n) \log \sigma$ bits and answers `rank`, `select` and `access` in proportional time to its height h.

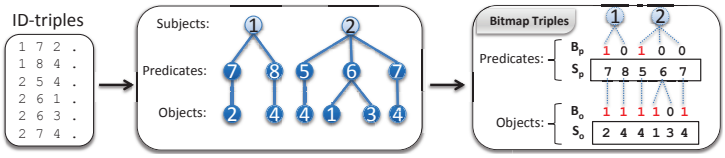


Fig. 2. Description of Bitmap Triples.

3.2 Bitmap Triples for SP-O Indexing

HDT describes *Bitmap Triples* (BT) as a specific triples encoding which represents the RDF graph through its adjacency matrix. In practice, BT slices the matrix by subject and encodes the *predicate-object* lists for each subject in the dataset. Let us suppose that the triples below comprise all occurrences of subject s:

$$\{(s, p_1, o_{11}), \cdots, (s, p_1, o_{1n_1}), (s, p_2, o_{21}), \cdots (s, p_2, o_{2n_2}), \cdots (s, p_k, o_{kn_k})\}$$

These triples are reorganized into *predicate-object* adjacency lists as follows:

$$s \rightarrow [(p_1, (o_{11}, \cdots, o_{1n_1}), (p_2, (o_{21}, \cdots, o_{2n_2})), \cdots (p_k, (o_{k1}, \cdots, o_{kn_k})].$$

Each list represents a predicate, p_j, related to s and contains all objects reachable from s through this predicate.

This transformation is illustrated in Figure 2; the ID-based triples representation (labeled as *ID-triples*) is firstly presented, and its reorganization in adjacency lists is shown on its right. As can be seen, adjacency lists draw tree-shaped structures containing the subject ID in the root, the predicate IDs in the middle level, and the object IDs in the leaves (note that each tree has as many leaves as occurrences of the subject in the dataset). For instance, the right tree represents the second adjacency list in the dataset, thus it is associated to the subject `2` (rooting the tree). In the middle level, the tree stores (in a sorted way) the three IDs representing the predicates related to the current subject: `5`,`6`, and `7`. The leaves comprise, in a sorted fashion, all objects related to the subject `2` through a given predicate: e.g. objects `1` and `3` are reached through the path `2`,`6`; which means that the triples (`2`,`6`,`1`) and (`2`,`6`,`3`) are in the dataset.

BT implements a compact mechanism for modeling an RDF graph as a forest containing as many trees as different subjects are used in the dataset. This assumption allows subjects to be implicitly represented by considering that the $i-th$ tree draws the adjacency list related to the $i-th$ subject. Moreover, two integer sequences: $\mathcal{S}_p$ and $\mathcal{S}_o$, are used for storing, respectively, the predicate and the object IDs within the adjacency lists. Two additional bitsequences: $\mathcal{B}_p$ and $\mathcal{B}_o$ (storing list cardinalities) are used for delimitation purposes. This is illustrated on the right side of Figure 2. As can be seen, $\mathcal{S}_p$ stores the five predicate IDs involved in the adjacency lists: $\{7, 8, 5, 6, 7\}$ and $\mathcal{B}_p$ contains five bits: $\{10100\}$, which are interpreted as follows. The *first* 1-bit (in $\mathcal{B}_p[1]$) means that the list for the *first subject* begins at $\mathcal{S}_p[1]$, and the *second* 1-bit (in $\mathcal{B}_p[3]$) means that the list for the *second subject* begins at $\mathcal{S}_p[3]$. The cardinality of the list is obtained by subtracting the positions, hence the adjacency lists for the first and the second subject contain respectively $3-1=2$ and $6-3=3$ predicates. The information stored in $\mathcal{S}_o = \{2, 4, 4, 1, 3, 4\}$ and $\mathcal{B}_o = \{111101\}$ is similarly interpreted, but note that adjacency lists, at this level, are related to subject-predicate pairs.

Table 1. Triple pattern resolution on BT (operations marked with $*$ are performed as many times as predicates to be included in the list obtained from `findPred`).

Triple Pattern	Operations
(S,P,O)	`findPred(S), filterPred(`$\mathcal{P}_s$`,P), findObj(pos), filterObj(`$\mathcal{O}_x$`,O).`
$(S,P,?O)$	`findPred(S), filterPred(`$\mathcal{P}_s$`,P), findObj(pos).`
$(S,?P,O)$	`findPred(S), {findObj(pos), filterObj(`$\mathcal{O}_x$`,O)}`*.
$(S,?P,?O)$	`findPred(S), findObj(pos)`*.

BT gives a practical representation of the graph structure which allows triples to be sequentially listed. However, direct accessing to the triples in the i-th list would require a sequential search until the i-th 1-bit is found in the bitsequence. Direct access (in constant time) to any adjacency list could be easily achieved with a little spatial $o(n)$ overhead on top of the original bitsequence sizes. It ensures constant time resolution for `rank`, `select`, and `access`, and allows efficient primitive operations to be implemented on the adjacency lists:

- `findPred(i)`: returns the list of predicates related to the subject `i` ($\mathcal{P}_i$), and the position pos in which this list begins in $\mathcal{S}_p$. This position is obtained as $pos = select_1(\mathcal{B}_p, i)$, and $\mathcal{P}_i$ is retrieved from $\mathcal{S}_p[pos, select_1(\mathcal{B}_p, i+1) - 1]$.
- `filterPred(`$\mathcal{P}_i$`,j)`: performs a binary search on $\mathcal{P}_i$ and returns the position of the predicate j in $\mathcal{P}_i$, or 0 if it is not in the list.
- `findObj(pos)`: returns the list of objects ($\mathcal{O}_x$) related to the subject-predicate pair represented in $\mathcal{S}_p[pos]$. It positions the pair: $x = rank_1(\mathcal{B}_o, pos)$, and then extracts $\mathcal{O}_x$ from $\mathcal{S}_o[select_1(\mathcal{B}_o, x), select_1(\mathcal{B}_o, x+1) - 1]$.
- `filterObj(`$\mathcal{O}_j$`,k)`: performs a binary search on $\mathcal{O}_j$ and returns the position of the object k in $\mathcal{O}_j$, or 0 if it is not in the list.

Table 1 summarizes how these primitives can be used to resolve some triple patterns in SPARQL: `(S,P,O)`, `(S,P,?O)`, `(S,?P,O)`, and `(S,?P,?O)`. Let us suppose that we perform the pattern `(2,6,?)` over the triples in Figure 2. BT firstly retrieves (by `findPred(2)`) the list of predicates related to the subject 2: $\mathcal{P}_2 = \{5, 6, 7\}$, and its initial position in $\mathcal{S}_p$: $pos_{ini} = 3$. Then, `filterPred(`$\mathcal{P}_2$`,6)` returns $pos_{off} = 2$ as the position in which `6` is in $\mathcal{P}_2$. This allows us to obtain the position in which the pair `(2,6)` is represented in $\mathcal{S}_p$: $pos = pos_{ini} + pos_{off} = 3 + 2 = 5$, due to $\mathcal{P}_2$ starts in $\mathcal{S}_p[3]$, and 6 is the second element in this list. Finally, `findObj(5)` is executed for retrieving the final result comprising the list of objects $\mathcal{O}_5 = \{1, 3\}$ related to the pair `(2,6)`.

4 Focusing on Querying (*HDT-FoQ*)

`HDT` was originally intended for publication and exchange, but its triples component provides enough information for efficient RDF retrieval. The bitsequences delimiting adjacency lists provide an `SP-O` index which allows some patterns to be efficiently resolved (row `BT` in Table 2). It enables `HDT` to be exploited as a basis for an indexed representation (called `HDT-FoQ`: `HDT` *Focused on Querying*) which allows exchanged RDF to be directly consumed using `SPARQL`.

This section presents how `HDT` is enhanced from an innovative perspective focused on querying. Three main issues must be addressed to obtain an efficient configuration for SPARQL resolution: i) The **dictionary** is serialized in

Table 2. Indexes and Triple Pattern resolution through incremental proposals.

	Index Order			Triple Patterns						
	SP-O	PS-O	OP-S	SPO	SP?	S?O	S??	?PO	?P?	??O
BT	√	-	-	SP-O	SP-O	SP-O	SP-O	-	-	-
BT+$\mathcal{W}_P$	√	√	-	SP-O	SP-O	SP-O	SP-O	PS-O	PS-O	-
HDT-FoQ	√	√	√	SP-O	SP-O	SP-O	SP-O	OP-S	PS-O	OP-S

a compressed way which allows it to be included as part of the original HDT representation. It must also be directly consumable to provide efficient operations for querying the mapping between each term and the corresponding ID. ii) The original **triples** component representation is enhanced to provide efficient RDF retrieval covering all possible triple patterns in SPARQL. iii) Efficient **join algorithms** are implemented to perform Basic Graph Patterns (BGPs) [2].

4.1 Functional Dictionary Serialization

The dictionary component contributes greatly to the HDT compactness because it enables triples to be modeled through three-ID groups. However, RDF dictionaries can suffer from scalability drawbacks because they take more space than ID-triples representations [15]. An advanced serialization addresses this drawback while providing basic operations for consumption, *i.e.*, operations from *term to ID* (`locate`), and from *ID to term* (`extract`). HDT-FoQ relies on an HDT representation including such an advanced dictionary. It is organized as follows:

- **Common subjects and objects** (SO) maps to the range [1, |SO|] all terms playing *subject* and *object* roles.
- **Subjects** (S) maps to [|SO|+1, |SO|+|S|] all remaining terms playing as *subject*.
- **Objects** (O) maps to [|SO|+1, |SO|+|O|] all remaining terms playing as *object*.
- **Predicates** (P) maps terms playing as predicate to [1,|P|].

This four-subset partitioning fits the original HDT approach [11] and allows terms playing as subject and object to be represented only once. This dictionary organization is serialized through four independent streams which respectively concatenate, in lexicographic order, the terms within each subset. Each stream is finally encoded with *Plain Front-Coding* (PFC) [8]. It adapts differential Front-Coding compression [19] to the case of string dictionaries and it provides, at consumption, efficient `locate` and `extract` resolution in compressed space.

4.2 A Wavelet Tree-based Solution for PS-O Indexing

Bitmap Triples (BT) represents the triples component through adjacency lists prioritized by subject. This decision addresses fast querying for patterns providing the subject, but makes retrieval by any other dimension difficult.

We firstly focus on predicate-based retrieval on top of BT. This requires the efficient resolution of patterns providing the predicate while leaving the subject as variable: (?,P,O) and (?,P,?). In both cases, all occurrences of P must be quickly located, but BT scatters them along the sequence of predicates ($\mathcal{S}_p$) and its sequential scan arises as the trivial solution. Thus, the predicate-based retrieval demands indexed access to $\mathcal{S}_p$, which can be provided by representing the sequence with the wavelet tree structure.

Algorithm 1 `findSubj(i)`

1: $occs \leftarrow rank_i(\mathcal{W}_p, n)$;
2: **for** $(x = 1$ *to* $occs)$ **do**
3: $\quad pos[\,] \leftarrow select_i(\mathcal{W}_p, x)$;
4: $\quad \mathcal{S}[\,] \leftarrow rank_1(\mathcal{B}_p, pos[x])$;
5: **end for**
6: **return** $pos; \mathcal{S}$

Algorithm 2 `filterSubj(i,j)`

1: $pos_j \leftarrow select_1(\mathcal{B}_p, j) - 1$;
2: $pos_{j+1} \leftarrow select_1(\mathcal{B}_p, j+1) - 1$;
3: $occs \leftarrow rank_i(\mathcal{W}_p, pos_{j+1}) - rank_i(\mathcal{W}_p, pos_j)$;
4: **return** $occs$

This new wavelet-tree based representation of $\mathcal{S}_p$ is renamed $\mathcal{W}_p$. It adds an additional overhead of $o(n)\log(|P|)$ bits to the space used in the original $\mathcal{S}_p$, and allows each predicate occurrence to be located in time $O(\log|P|)$ through the `select` operation. This is an acceptable cost for our retrieval purposes because of the small number of predicates used, in practice, for RDF modeling. In the same way, the `access` operation also has a logarithmic cost, so any predicate within $\mathcal{W}_p$ is now retrieved in time $O(\log(|P|))$. Finally, note that `rank` allows the occurrences of a predicate to be counted up to a certain position of $\mathcal{W}_p$.

The wavelet tree structure allows access by predicate to be supported on two new primitives traversing adjacency lists:

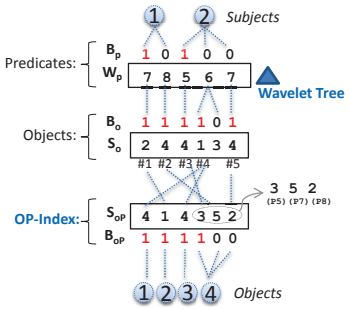
- `findSubj(i)`: returns the list of subjects related to the predicate `i` and the positions in which they occur in $\mathcal{W}_p$. This operation is described in Algorithm 1. It iterates over all occurrences of the predicate `i` and processes one of them for each step. It locates the occurrence position in $\mathcal{W}_p$ (line 3) and uses it for retrieving the subject (line 4) which is added to the result set.
- `filterSubj(i,j)`: checks whether the predicate `i` and the subject `j` are related. It is described in Algorithm 2. It delimits the predicate list for the $j-th$ subject, and counts the occurrences of the predicate i to pos_j (o_j) and pos_{j+1} (o_{j+1}). *Iff* $o_{j+1} > o_j$, the subject and the predicate are related.

Hence, the wavelet tree contributes with a `PS-O` index which allows two additional patterns to be efficiently resolved (row `BT+`$\mathcal{W}_p$ in Table 2). Both `(?S,P,?O)` and `(?S,P,O)` first perform `findSubj(P)` to retrieve the list of subjects related to the predicate `P`. Then, `(?S,P,?O)` executes `findObj` for each retrieved subject and obtains all objects related to it. In turn, `(?S,P,O)` performs a `filterObj` for each subject to test if it is related to the object given in the pattern.

Let us suppose that, having the triples in Figure 2, we ask for all subjects and objects related through the predicate `7`: `(?S,7,?O)`. `findSubj(7)` obtains the list of two subjects related to the predicate ($\mathcal{S} = \{1, 2\}$) and their positions in $\mathcal{W}_p$ (`pos={1,5}`). The subsequent `findObj(1)` and `findObj(5)` return the list of objects $\{2\}$ and $\{4\}$ respectively representing the triples `(1,7,2)` and `(2,7,4)`.

4.3 An Additional Adjacency List for `OP-S` Indexing

The wavelet-tree based enhancement leaves object-based access as the only non-efficient retrieval in our approach. As we illustrated in Figure 2, objects are represented as leaves of the tree drawn for each adjacency list, so the sequence $\mathcal{S}_o$ stores all object occurrences within the dataset (each one related to the corresponding predicate-subject pair). In this case, we require an additional index `OP-S` which allows adjacency lists to be traversed from the leaves.



Algorithm 3 `findPredSubj(i)`

1: $pos_i \leftarrow select_1(\mathcal{B}_{oP}, i)$;
2: $pos_{i+1} \leftarrow select_1(\mathcal{B}_{oP}, i+1) - 1$;
3: **for** $(x = pos_i \ to \ pos_{i+1})$ **do**
4: $\quad ptr \leftarrow select_1(\mathcal{B}_o, \mathcal{B}_{oP}[x])$;
5: $\quad \mathcal{P}[\,] \leftarrow access(\mathcal{W}_p, ptr)$;
6: $\quad \mathcal{S}[\,] \leftarrow rank_1(\mathcal{B}_p, ptr)$
7: **end for**
8: **return** $\mathcal{P}; \mathcal{S}$

Fig. 3. Final `HDT-FoQ` configuration.

The index `OP-S` is represented with an integer sequence: $\mathcal{S}_{oP}$, which stores, for each object, a sorted list of references to the predicate-subject pairs (sorted by predicate) related to it. It is worth noting that the $i-th$ predicate-subject pair is identified through the $i-th$ 1-bit in $\mathcal{B}_o$. A bitsequence $\mathcal{B}_{oP}$ is also used for representing cardinalities as in the upper levels. This is illustrated in Figure 3; for instance, the fourth list in $\mathcal{S}_{oP}$ (pointed to by the fourth 1-bit in $\mathcal{B}_{oP}$) stores the reference `{3,5,2}`: `3` points to the third 1-bit in $\mathcal{B}_o$ representing the relation between the object `4` and the predicate `5`; the reference `5` points to the fifth 1-bit (predicate `7`), and the third reference: `8` points to the predicate `8`.

This index `OP-S` enables efficient object-based retrieval trough two new primitives which traverse adjacency lists from the leaves:

- `findPredSubj(i)`: returns the list of predicate-subject pairs related to the object `i`. This operation is described in Algorithm 3. It firstly delimits the list for the object `i` and then iterates over each reference. Each step locates the position *ptr* which points to the position which represents the reference in $\mathcal{B}_o$ (line 4). This value is then used for retrieving the corresponding predicate from $\mathcal{W}_p$ (line 5), and the subject from $\mathcal{B}_p$ (line 6).
- `filterPredSubj(i,j)`: checks whether the object `i` and the predicate `j` are related, and narrows their occurrences in the list. It firstly delimits the list for the object `i` and then binary searches it to narrow the occurrences of j.

This enhancement contributes to our solution with the index `OP-S` and allows triple patterns `(?S,?P,O)` and `(?S,P,O)` to be efficiently resolved (see row **HDT-FoQ** in Table 2). The first one is resolved by performing `findPredSubj` for the object provided in the pattern. `(?S,P,O)` was resolved through the wavelet tree, but its resolution is now speeded up. In this case, `filterPredSubj(O,P)` narrows the references from `O` to `P`, and then retrieves the corresponding subject (as in line 6 of Algorithm 3).

The functionality of the index `OP-S` can be seen through the `(?S,?P,1)` pattern. It asks for all subject-predicate pairs related to the object `1` in the triples represented in Figure 2. The operation `findPredSubj(i)` firstly narrows the range of references from the object `1` to $\mathcal{S}_{oP}[1,1]$. It only comprises the value `4`

which points to the fourth 1-bit in $\mathtt{B}_p$. It represents the predicate $access(\mathcal{W}_p, 4) = 6$ and the subject $rank_1(\mathcal{B}_p, 4) = 2$, so this pattern matches the triple `(2,6,1)`.

4.4 Joining Triple Patterns

`HDT-FoQ` is the result of post-processing `HDT` for RDF consumption. It is built, at consumption, on top of the exchanged `HDT`, which includes the functional dictionary and the Bitmap Triples. First, the wavelet tree is constructed using the implementation provided in the *libcds* library [1]. Then, the object structure in Bitmap Triples is scanned to build the `OP-S` index. The result is a compact RDF representation optimized to be managed and queried in main memory. As summarized in Table 2, `HDT-FoQ` only requires three indexes (`SP-O`, `PS-O` and `OP-S`) to perform efficient RDF retrieval, in contrast to the six combinations used in solutions within the state-of-the-art [18, 17].

`HDT-FoQ` efficiently performs triple patterns, setting the basis for SPARQL resolution. We rely on the fact that the SPARQL core is built around the concept of Basic Graph Pattern (BGP) and its semantics in order to build conjunctive expressions joining triple patterns through shared variables.

`Merge` and `Index` joins can be directly resolved on top of `HDT-FoQ`. `Merge join` is used when the results of both triple patterns are sorted by the join variable. It is worth noting that triple pattern results are given in the order provided by the index used (see Table 2). If the results of one triple pattern are not sorted by the join variable, `index join` is performed. It first retrieves all results for the join variable in one triple pattern and replaces them in the other one. Ideally, the first evaluation should be on the less expensive pattern, in terms of its expected number of results.

5 Experimental Evaluation

This section studies the Publication-Exchange-Consumption workflow on a real-world setup in which the three main agents are involved:

- The **data provider** is implemented on a powerful computational configuration. It simulates an efficient data provider within the Web of Data. We use an Intel Xeon E5645@2.4GHz, 96GB DDR3@1066Mhz.
- The **network** is regarded as an ideal communication channel for a fair comparison. It is considered free of errors and any other external interference. We assume a transmission speed of 2Mbyte/s.
- The **consumer** is designed on a conventional configuration because it plays the role of any agent consuming RDF within the Web of Data. It is implemented on an AMD-PhenomTM-II X4 955@3.2GHz, 8GB DDR2@800MHz.

We first analyze the impact of using `HDT` as a basis for publication, exchange and consumption within the studied workflow, and compare its performance with respect to those obtained for the methods currently used in each process. Then, we focus on studying the performance of `HDT-FoQ` as the querying infrastructure for SPARQL: we measure response times for triple pattern and join resolution.

Table 3. Description of the real-world datasets used in the experimentation.

Dataset	Plain Ntriples	Size (MB)	Available at
linkedMDB	6,148,121	850.31	http://queens.db.toronto.edu/~oktie/linkedmdb
dblp	73,226,756	11,164.41	http://dblp.l3s.de/dblp++.php
geonames	112,335,008	12,358.98	http://download.geonames.org/all-geonames-rdf.zip
dbpedia (en)	257,869,814	37,389.90	http://wiki.dbpedia.org/Downloads351

All experiments are carried out on a heterogeneous configuration of real-world datasets of different sizes and from different application domains (Table 3). We report "user" times in all experiments. The HDT prototype is developed in C++ and compiled using `g++-4.6.1 -O3 -m64`. Both the HDT library and a visual tool to generate/browse/query HDT files are publicly available[7].

5.1 Analyzing the Publication-Exchange-Consumption Workflow

The overall workflow analysis considers that the publication process is performed once, whereas exchange and preprocessing costs are paid each time that any consumer retrieves the published dataset. The publication policy affects the performance of exchange because it depends on the dataset size, but also the decompression time (as initial consumption step) which is directly related to the compressor used for publication. We use two Lempel-Ziv based compressors[8] for publication: the widely-used `gzip` and the `lzma` algorithm in the suite `p7zip`.

We assume that the publication process begins with the dataset already serialized. Thus, `gzip`/`lzma` based publication only considers the compression time, whereas processes based on `HDT` comprise the times required for generating the `HDT` representation and its subsequent compression.

Table 5 shows the time used for publication in the data provider: `gzip` is the faster choice and largely outperforms `lzma` and the HDT-based publication. However, size is the most important factor due to its influence on the subsequent processes (Table 4). `HDT+lzma` is the best choice. It achieves highly-compressed representations: for instance, it takes 2 and 3 times less space than `lzma` and `gzip` for `dbpedia`. This spatial improvement determines exchange and decompression (for consumption) times as shown in Tables 6 and 7.

On the one hand, the combination of `HDT` and `lzma` is the clear winner for exchange because of its high-compressibility. Its transmission costs are smaller than the other alternatives: it improves them between $10-20$ minutes for the largest dataset. On the other hand, `HDT+gzip` is the most efficient at decompression, but its improvement is not enough to make up for the time lost in exchange with respect to `HDT+lzma`. However, its performance is much better than the one achieved by universal compression over plain RDF. Thus, `HDT`-based publication and its subsequent compression (especially with `lzma`) arises as the most efficient choice for exchanging RDF within the Web of Data.

The next step focuses on making the exchanged datasets queryable for consumption. We implement the traditional process, which relies on the indexing of plain RDF through any RDF store. We choose three systems[9]: `Virtuoso` (re-

[7] `http://www.rdfhdt.org`

[8] `http://www.gzip.org/` (`gzip`), and `http://www.7-zip.org/` (`lzma`)

[9] Hexastore has been kindly provided by the authors. `http://www.openlinksw.com/` (Virtuoso), `http://ht tp://www.mpi-inf.mpg.de/~neumann/rdf3x/` (RF3X).

Table 4. Compressed sizes (MB).

Dataset	gzip	lzma	HDT+ gzip	HDT+ lzma
linkedMDB	38.86	19.21	15.77	**12.49**
dblp	468.47	328.17	229.23	**178.71**
geonames	701.98	348.93	305.30	**236.59**
dbpedia	3,872.29	2,653.79	1,660.73	**1,265.43**

Table 5. Publication times (seconds).

Dataset	gzip	lzma	HDT+ gzip	HDT+ lzma
linkedMDB	**11.36**	882.80	65.57	91.01
dblp	**162.91**	6,214.32	808.93	1,319.60
geonames	**196.90**	14,683.90	1,586.15	2,337.82
dbpedia	**956.71**	27,959.85	3,648.75	7,306.17

Table 6. Exchange times (seconds).

Dataset	gzip	lzma	HDT+ gzip	HDT+ lzma
linkedMDB	19.43	9.61	7.88	**6.25**
dblp	234.23	164.08	114.62	**89.35**
geonames	350.99	174.46	152.65	**118.29**
dbpedia	1,936.14	1,326.89	830.36	**632.71**

Table 7. Decompression times (seconds).

Dataset	gzip	lzma	HDT+ gzip	HDT+ lzma
linkedMDB	3.04	5.11	**0.33**	1.05
dblp	37.08	70.86	**4.63**	14.82
geonames	45.49	87.51	**8.81**	19.91
dbpedia	176.14	357.86	**46.51**	103.03

Table 8. Indexing times (seconds).

Dataset	Virtuoso	Hexastore	RDF3X	HDT-FoQ
linkedMDB	369.05	1,810.67	111.08	**1.91**
dblp	5,543.99	×	1,387.29	**16.79**
geonames	17,902.43	×	2,691.66	**43.98**
dbpedia	×	×	7,904.73	**124.44**

Table 9. Overall times (seconds).

Dataset	Comp.RDF+ Indexing	Comp.HDT+ HDT-FoQ
linkedMDB	125.80	**9.21**
dblp	1,622.23	**120.96**
geonames	2,953.63	**182.18**
dbpedia	9,589.48	**860.18**

lational solution), RDF3X (multi-indexing solution), and Hexastore (in-memory solution). We compare their performance against HDT-FoQ, which builds additional structures on the HDT-serialized datasets previously exchanged.

Table 8 compares these times. As can be seen, HDT-FoQ excels for all datasets: its time is between one and two orders of magnitude lower than that obtained for the other techniques. For instance, HDT-FoQ takes 43.98 seconds to index geonames, whereas RDF3X and Virtuoso use respectively 45 minutes and 5 hours. It demonstrates how HDT-FoQ leverages the binary HDT representation to make RDF quickly queryable through its retrieval functionality. Finally, it is worth noting that Virtuoso does not finish the indexing for dbpedia after more than 1 day, and Hexastore requires a more powerful computational configuration for indexing datasets larger than linkedMDB. This fact shows that we successfully achieve our goal of reducing the amount of computation required by the consumer to make queryable RDF obtained within the Web of Data.

Overall Performance. This section comprises an overall analysis of the processes above. Note that publication is decoupled from this analysis because it is performed only once, and its cost is attributed to the data provider. Thus, we comprise times for exchange and consumption. These times are shown in Table 9. It compares the time needed for a conventional implementation against that of the HDT driven approach. We choose the most efficient configurations in each case: i) Comp.RDF+Indexing comprises lzma compression over the plain RDF representation and indexing in RDF3X, and ii) Comp.HDT+HDT-FoQ compresses the obtained HDT with lzma and then obtains HDT-FoQ.

The workflow is completed between 10 and 15 times faster using the HDT driven approach. Thus, the consumer can start using the data in a shorter time, but also with a more limited computational configuration as reported above.

Table 10. Indexing sizes (MB).

Dataset	Virtuoso	Hexastore	RDF3X	HDT	HDT-FoQ
linkedMDB	518.01	6,976.07	377.34	48.70	**68.03**
dblp	3,982.01	×	3,252.27	695.73	**850.62**
geonames	9,216.02	×	6,678.42	1,028.85	**1,435.05**
dbpedia	×	×	15,802.03	4,433.28	**5,260.33**

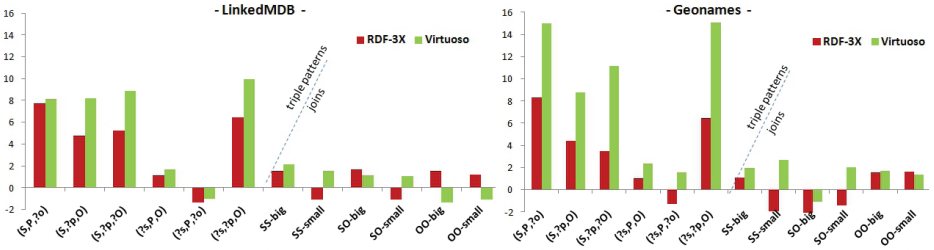


Fig. 4. Comparison on querying performance.

5.2 HDT-FoQ in Consumption: Performance for SPARQL Querying

This section complements the previous analysis focusing on the performance of HDT-FoQ as the basis for SPARQL querying. As explained, HDT-FoQ is the final result, in the consumer, when the workflow Publication-Exchange-Consumption is driven through our HDT-based approach, and it is used as an in-memory index for SPARQL querying. We firstly show the spatial needs of HDT-FoQ to be efficiently loaded in the consumer configuration and then study its performance for triple pattern and join query resolution. Our main aim is to show the HDT-FoQ efficiency for RDF retrieval and also for joining in order to demonstrate its capabilities for SPARQL resolution on top of that. We compare our results with respect to the indexing systems presented above.

Table 10 summarizes the sizes of the indexes built for each dataset within each studied solution. The columns HDT and HDT-FoQ, respectively, show the size of the original HDT representation (after decompression) and the resultant indexed one built on top of it. It is worth noting that the sizes reported for HDT-FoQ also include the overhead required for managing it in main memory. As can be seen, HDT-FoQ takes between 15% and 40% of extra space on top of HDT representations. These results place HDT-FoQ as the more compact index, largely doing better than the other solutions. Finally, we emphasize the comparison between Hexastore and HDT-FoQ because both are in-memory solutions. Whereas Hexastore requires $\approx$ 7 GB of main memory for managing just over 6 million triples (linkedMDB), our approach just uses 68.03 MB for fitting this dataset in memory. This achievement is analyzed from a complementary perspective: the consumer can manage just over 258 million triples (dbpedia) using HDT-FoQ (and 3GB of memory are still free in the system), whereas only 6 million triples can be managed using Hexastore (and only 1GB would remain free).

Query performance is evaluated over linkedMDB and geonames. For each one, we design a testbed of randomly generated queries which covers the entire spectrum of triple patterns and joins. We consider 5000 random triple patterns of each type ((?S,P,?O) is limited by the number of different predicates). We split join

tests into Subject-Subject (SS), Object-Object (OO) and Subject-Object (SO) categories. For each one, we generate 15 queries with a high number of intermediate results (big subsets) and another 350 queries with fewer results (small subsets). The full testbed is available at http://dataweb.infor.uva.es/queries-eswc12.tgz

Querying times are obtained by running 5 independent executions of the testbed and average total user times. We compare HDT-FoQ against RDF-3X and Virtuoso (Hexastore could not run most queries because of the aforementioned limited free memory). We query these systems within a "warm" scenario: we run 5 previous executions before measuring time for these disk-based systems to have the required data available in main memory. Figure 4 shows the performance comparison as $\{time\_in\_compared\_system\}/\{time\_in\_FoQ\}$. For instance a value of 6 means that it performs 6 times faster than the compared system. For visualization purposes, we invert this ratio whenever we run slower, hence a value of -2 means that we perform 2 times slower.

It is worth noting that HDT-FoQ excels for almost every individual triple pattern. It speeds-up their resolution, only losing performance for (?S,P,?O), in which a logarithmic cost is paid for accessing predicates in the wavelet tree. The analysis of join performance shows that i) HDT-FoQ is the most efficient choice for most of the joins in medium-sized datasets such as linkedMDB, thanks to efficient triple pattern resolution, but ii) these stores leverage their optimized join implementations in larger datasets (geonames). Optimized join algorithms implemented on top of HDT-FoQ would allow it to compete fairly in this latter case by leveraging HDT-FoQ performance for triple pattern resolution.

6 Conclusions and Future Work

Inherent scalability drawbacks of huge RDF graphs discourage their consumption due to the space they take up, the powerful resources and the large time required to process them. In this paper, we focus on a novel direction for speeding up consumption. We firstly rely on an existing binary format, called HDT (Header-Dictionary-Triples), which provides efficient exchange. Then, we propose HDT-FoQ, a compact full-index created over HDT, at consumption. Thus, the exchanged RDF data become direct and easily queryable.

Our experiments show that huge RDF data are exchanged and post-processed (ready to be queried) $10-15$ times faster than traditional solutions. Then, the proposed in-memory system for consumption (HDT-FoQ) excels in triple pattern resolution, remains competitive in joins of middle-sized datasets and shows potential improvement for larger datasets.

These results open up interesting issues for future work. We should work on improving predicate-based retrieval because it reports the less-competitive performance. Our on-going work relies on the optimization of the predicate index by tuning the trade-off between access time and spatial needs. In addition, we plan to optimize our join algorithms with *Sideways Information Passing* mechanisms, leveraging efficient resolution of triple patterns. Finally, although the use of succinct data structures allows more data to be managed in the main memory, it could remain excessive for consumers with limited memory. Under this

scenario, we devise an evolution of HDT-FoQ to perform as an in-memory/on-disk system providing dynamic data management, *i.e.*, efficient insertion, updating and deletion of triples at consumption.

Acknowledgments

This research has been supported by MICINN (TIN2009-14009-C02-02) and Science Foundation Ireland under Grant No.~SFI/08/CE/I1380(Lion-II). The third author receives a grant from the JCyL and the ESF. We particularly wish to thank Claudio Gutierrez, for his continued motivation and selfless help, and the Database Lab (Univ. of A Coruña) for lending us the servers for our experiments.

References

1. *Compact Data Structures Library (libcds).* http://libcds.recoded.cl/.
2. *SPARQL Query Language for RDF.* W3C Recomm. 2008. http://www.w3.org/TR/rdf-sparql-query/.
3. *Turtle-Terse RDF Triple Language.* W3C Team Subm. 2008. http://www.w3.org/TeamSubmission/turtle/.
4. *Notation3.* W3C Design Issues. 1998. http://www.w3.org/DesignIssues/Notation3.
5. *RDF/XML Syntax.* W3C Recomm. 2004. http://www.w3.org/TR/REC-rdf-syntax/.
6. *Binary RDF Representation for Publication and Exchange (HDT).* W3C Member Subm. 2011. http://www.w3.org/Submission/2011/03/.
7. C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked Data On the Web (LDOW2008). In *Proc. of WWW*, pages 1265–1266, 2008.
8. N. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed String Dictionaries. In *Proc. of SEA*, pages 136–147, 2011.
9. L. Ding and T. Finin. Characterizing the Semantic Web on the Web. In *Proc. of ISWC*, pages 242–257, 2006.
10. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In *Proc. of CSSW*, pages 59–68, 2007.
11. J. Fernández, M. Martínez-Prieto, and C. Gutierrez. Compact Representation of Large RDF Datasets for Publishing and Exchange. In *ISWC*, pages 193–208, 2010.
12. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Proc. of WEA*, pages 27–38, 2005.
13. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of SODA*, pages 841–850, 2003.
14. D. Le-Phuoc, J. X. Parreira, V. Reynolds, and M. Hauswirth. RDF On the Go: An RDF Storage and Query Processor for Mobile Devices. In *Proc. of ISWC*, 2010. Available at http://iswc2010.semanticweb.org/pdf/503.pdf.
15. M. Martínez-Prieto, J. Fernández, and R. Cánovas. Compression of RDF Dictionaries. In *Proc of. SAC*, 2012. Available at: http://dataweb.infor.uva.es/sac2012.pdf.
16. G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM COMPUT SURV*, 39(1):art. 2, 2007.
17. T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
18. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proc. of the VLDB Endowment*, 1(1):1008–1019, 2008.
19. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.