Journal of Visual Languages and Computing 31 (2015) 222–229

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Graph databases methodology and tool supporting index/store versioning

Pierfrancesco Bellini, Ivan Bruno, Paolo Nesi*, Nadia Rauch

DISIT Lab, Department of Information Engineering, University of Florence, Italy

ARTICLE INFO

Available online 6 November 2015

Keywords:
RDF knowledge base versioning
Graph stores versioning
RDF store management
Knowledge base life cycle

ABSTRACT

Graph databases are taking place in many different applications: smart city, smart cloud, smart education, etc. In most cases, the applications imply the creation of ontologies and the integration of a large set of knowledge to build a knowledge base as an RDF KB store, with ontologies, static data, historical data and real time data. Most of the RDF stores are endowed with inferential engines that materialize some knowledge as triples during indexing or querying. In these cases, deleting concepts may imply the removal and change of many triples, especially if the triples are those modeling the ontological part of the knowledge base, or are referred by many other concepts. For these solutions, the graph database versioning feature is not provided at level of the RDF stores tool, and it is quite complex and time consuming to be addressed as black box approach. In most cases the indexing is a time consuming process, and the rebuilding of the KB may imply manually edited long scripts that are error prone. Therefore, in order to solve these kinds of problems, this paper proposes a lifecycle methodology and a tool supporting versioning of indexes for RDF KB store. The solution proposed has been developed on the basis of a number of knowledge oriented projects as Sii-Mobility (smart city), RESOLUTE (smart city risk assessment), ICARO (smart cloud). Results are reported in terms of time saving and reliability.

1. Introduction

Large graph databases are getting a strong push in their diffusion for setting up new kind of big data services for smart cities, digital libraries, competence modeling, health care, smart education, etc. This fact is mainly due to their capability in modeling knowledge and thus on creating Knowledge-Based, KB, systems [1]. Graph databases may be implemented as RDF stores (Resource Description Framework) [2], to create interactive services in which reasoning and deductions can be elaborated including inference engines on top of the store. An RDF store is grounded on the concept of triple that puts in relationship two entities. A vocabulary defines the common characteristics of things belonging to classes and their relations. A vocabulary, also called ontology, is defined by using RDFS (RDF Schema, RDF Vocabulary Description Language) or the OWL extension (Ontology Web Language). Recently RDF store have been also addressed as noSQL stores for big data [3]. A large set of ontologies and related data sets are now accessible, see for example the large number of LOD (linked open data) accessible and related each other via URI [4,5]. RDF stores may be made accessible via an entry point to pose semantic queries formalized for example in SPARQL [6] (SPARQL Protocol and RDF Query Language, recursive definition). Non trivial RDF stores based solutions are typically produced by exploiting multiple ontologies, loading data triples and testing/validating the obtained results. This means that they are built by using

* Corresponding author.
E-mail addresses: pierfrancesco.bellini@unifi.it (P. Bellini), ivan.bruno@unifi.it (I. Bruno), paolo.nesi@unifi.it (P. Nesi), nadia.rauch@unifi.it) (N. Rauch).

http://dx.doi.org/10.1016/j.jvlc.2015.10.018
1045-926X/

some ontology building methodology [7,8], integrated with a knowledge base development life cycles.

The RDF store may grow over time adding new triples, and may have the capacity to learn if endowed with an inferential reasoner/engine, i.e., producing new knowledge as new triples. Thus, the inferential engine associated with the RDF store materializes new triples during reasoning (for example at the time of indexing or querying). These facts are the main motivations to low performances in indexing, and critical performances in deleting triples of RDF stores as graph databases since they are involved in removing the materialized triples. These features impact on store performances, and thus, in literature, many benchmarks for the evaluation of RDF stores are present. When RDF stores are used as a support for a KB, some of the changes in the RDF store can be destructive for the graph model, such as changes in the triples modeling the ontology on which millions of instances are related. In order to keep the performance acceptable, the RDF store has to be rebuilt from scratch or from some partial version to save time in releasing the new version. Thus, the life-cycle may present multiple cycles in which the RDF store is built incrementally via progressive refinements mediating among (i) reusing ontological models, (ii) increasing the capability of making deductions and reasoning on the knowledge base, (iii) maintaining acceptable query performance and rendering performances, (iv) simplifying the design of the front-end services, (v) satisfying the arrival of additional data and models and/or corrections, etc. A commonly agreed lifecycle model to build KBs is not available yet and many researchers have tried to embed KB development steps into some conventional software life-cycle models [9]. In general, development of KB systems is a multistep process and proceeds iteratively, using an evolutionary prototyping strategy. A number of lifecycle models have been proposed specifically for KB systems [10]. In the lifecycle model, a change in the ontology may generate the review and regeneration of a wide amount of RDF triples. The problem of ontology versioning as addressed in [11,12] can be easily applied if the ontology is not used as a basis for creating a large RDF KB store. Moreover, in [13], the versioning of RDF KB has been addressed similarly to the CVS solutions by using commands as: *commit, update, branch, merge*, and *diff*. The differences are computed at semantic level on files of triples. At database level, the key performance aspects of an RDF KB store version management are the storage space and the time to create a new version [14]. Therefore,

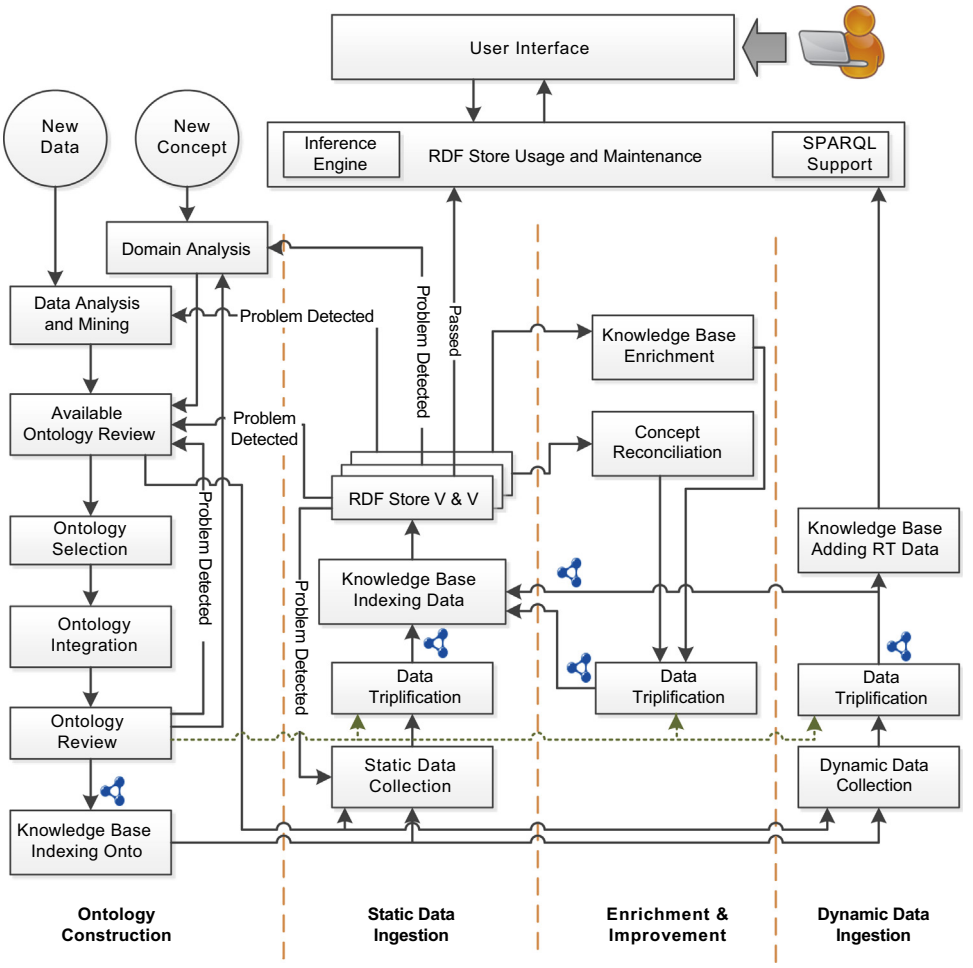

Fig. 1. RDF KB life cycle model.

possible approaches could be to store: (a) each version as an independent triples store [11–13]; (b) the deltas in terms of triples between two consecutive versions and implementing a computational and time consuming chain of processes to maintain and apply deltas [15].

In this paper, a versioning system for RDF KB proposes to integrate both (a) and (b) solutions. It manages versioning of RDF stores by: (i) keeping trace of the set of triples to build each version, (ii) storing each version and related set of triples, (iii) providing an automated tool for keeping trace of triple files, descriptions for store building and stores, (iv) allowing the versioning of the RDF KB store, (v) reducing the critical manual error prone operations. This approach allows to make indexing versioning for RDF stores that materialize triples at indexing (as OWLIM [http://www.ontotext.com/]) or at querying (as Virtuoso [http://virtuoso.openlinksw.com/]) without influencing the RDF store reconstruction. The resulting time for returning to a previous version and to reconstruction of a new one is satisfactory and viable, since some of the RDF stores are very time consuming in indexing, while other do not allow the deletion of triples. Therefore, the paper presents an RDF KB methodology life-cycle suitable for big data graph databases, and a versioning tool for RDF KB stores that has been developed and tested for SESAME OWLIM and Virtuoso; and thus it can be simply extended to other RDF stores. The solutions have been developed for Km4City project [16], and adopted for other RDF KB oriented projects as Sii-Mobility Smart City national project and RESOLUTE H2020 European Commission Project. They are large KB oriented projects in the Smart City, smart cloud, smart railway domains, developed at the DISIT Lab of the University of Florence http://www.disit.org/6568.

The paper is organized as follows. Section 2 presents the RDF knowledge base life-cycle model and methodology for development. In Section 3, the RDF KB indexing flow and requirements for the RDF Indexing Manager tool are presented. Section 4 describes the RDF Index Manager tool, detailing the architecture, and the XML formal model for index descriptors. In Section 5, experimental results are reported providing data related to real cases, in terms of time and managed complexity. Conclusions are drawn in Section 6.

2. A knowledge base life-cycle

Building a RDF KB is a challenging practice that needs a well-defined methodology and lifecycle to keep under control the entire development process. RDF KBs are mainly developed thanks to a cycle approach that allows checking and validating the advances made, and if needed, to make adjustments when a problem is identified. As stated above, the lifecycle proposed in this paper has been derived from the DISIT Lab experience cumulated while developing a number of big data RDF KBs.

The proposed methodology and lifecycle for RDF KB is reported in Fig. 1. The life-cycle presents 4 vertical pillars and one horizontal block that represents the RDF Store usage and Maintenance. The life-cycle spans from the ontology creation to the RDF Store usage on the front-end where also real time data are added.

The pillars refer to the

Ontology construction, from domain analysis the setup of the RDF Store containing triples of the selected ontologies and possible additional triples to complete the domain model (Knowledge Base Ontology, KBO). The combined ontology is reviewed and possible problems may lead to more or less deep redefinition of the process.

Static data ingestion: this phase is related to the loading of the data instances of the ontological classes and attributes. Despite their name, static data may change rarely over time, for example, the position of bus stops may be considered static data even if they change seasonally. They come from several sources (static, statistical, historical, etc.), and have to be converted in triples according to the KBO coming from the previous phase. Then, they are finally indexed by using several sets of triples, maybe thousands. The indexing produces a KB including the former KBO, plus many data instances; thus, allowing performing the RDF Store V&V, Verification and Validation. The V&V allows to identify some problems, that may constrain the experts to (i) wrong data or incomplete data to need a review of the data mapping to the ontology (restart from the first step of this phase of data collection), (ii) missing ontology aspects and classes, thus leading to the review of the ontology built (returning to Ontology Review), (iii) problems in data collected that may be wrongly mapped to ontology classes (returning to Data Analysis and Mining), (iv) mistakes in data mapping that may lead to revise the whole Domain Analysis, and successive steps. If this phase is passed, the RDF Store passes to the phase of RDF Store Usage and Maintenance. Additional static data sets may be added to the KBO if the ontological model supports them without deletion, otherwise a review is needed.

Enrichment and improvement, E&I: this phase allows detecting and solving problems that may be present in the produced RDF Store. E&I processes may take advantage from the access to the partially integrated KB, exploiting for examples solutions of Link Discovering [17,18], and/or making specific semantic queries. Additional processes of E&I may be added to the RDF Store if the model supports them without performing some delete otherwise a model review is needed.

Dynamic data ingestion: when the RDF store is in use, collected data from real time information (for example, bus delay with respect to the arrival time, weather forecast, likes on the user profile, status of sensors, status of cloud processes, etc.) can be added to the RDF Store and saved into the repository of the historical triples. Additional dynamic data sources may be added to the RDF Store if the model supports them without performing some delete otherwise a model review is needed. Please note that dynamic data should not need to validate and verify process since the data to be added in real time are new instances of data already mapped and integrated as historical data. They may need monitoring to be sure that the data quality received is conformant with the planned one.

3. RDF indexing flow and requirements

As described in the previous section, there are several reasons for which into the RDF KB life cycle the process may lead to (i) revise the ontology (and thus to revise the data mapping and triplification invalidating the indexing and the materialization of triples), (ii) revise the data ingestion including a new data mapping, quality improvement, reconciliation, enrichment and triplification. As stated in the previous section, the life-cycle model foresees two steps where the Knowledge Base Indexing has to be performed: KBIO for obtaining the first RDF store with the ontology only, and KBID adding to KBIO the triples of static data. On the other hand, as pointed out in the introduction, in most of the RDF store models, the versioning is not provided as an internal feature. This is due to the fact that it cannot be easily performed at index level and stored triples for their complexity in removing them, due to the triples materialization by inference. According to the proposed RDF KB life cycle, the modeling of a chain of connected versions of *indexes/RDF Stores,* with incremental complexity may be very useful to keep under control the evolving index with the aim of saving time by exploiting intermediate versions in generating the RDF Store/index for the successive deployment. For example, in the case of Smart City, the layered versions of the index may include the ontology, static and dynamic data, historical data, etc.

To better describe the process of RDF Index versioning, it is necessary to put in evidence the differences between the "*index*" and "*index descriptor*". An RDF KB store is in substance an "*index*", while content can be accessed via URI cited in the triples elements. The index is created by loading the triples into the RDF store, and as a result a binary index is built, maybe materializing additional triples according to the ontological model and the specific RDF store inferential engine adopted. The recipe to create the RDF Store index, that is the collection of atomic files containing triples (including triples of ontologies as well as those related to data sets: static, historical, dynamic), can be called as the "*index descriptor*", that may be used to generate a script for index generation. The script syntax can be different from an RDF Store to another, since their commands for loading and indexing can be different. This approach implies to have aside each pair "*index*" and "*index descriptor*" also the history of files containing triples with their versions, last update dates, and dependencies from other files. For example, see Fig. 2, where the reconciliation of triples connecting parking locations (File 1, ver 1.5) with respect to civic numbers depends on the ontology and on the parking area data sets. Thus leading to create a set of triples connected with dashed lines.

Definition. Let $F=\{f_1,f_2,f_3,\ldots\}$ be the set of triple files that are available for indexing and $DS=\{ds_1,ds_2,ds_3,\ldots\}$ is the set of datasets and ontologies that are available for ingestion. The function $ds: F\rightarrow DS$ associate the file to the dataset it belongs to, function $time: F\rightarrow\mathbb{N}$ associate each file with the time when it was created and function $dep: F\rightarrow\wp(F)$ associate each triples file with a set of files that it depends on (e.g. ontologies), $\wp(X)$ is the power set of set X. The *dep* function must not introduce a cyclic dependency among files. Moreover, a file should not depend on files created in the future:

$$\forall f\in F,\ \forall\ s\in dep(f).\ time(s)<time(f)$$

Example $\boldsymbol{DS}=\{km4c,otn,roads,services,busses\}$,

$$\boldsymbol{F}=\{kf_1,kf_2,of_1,rf_1,rf_2,sf_1,sf_2,bf_1,bf_1,bf_2,\},$$

$$\boldsymbol{ds}=\{(kf_1\rightarrow km4c),\ (kf_2\rightarrow km4c),\ (of_1\rightarrow otn),\ (rf_1\rightarrow roads),\ (rf_2\rightarrow roads),\ (sf_1\rightarrow services),\ (sf_2\rightarrow services),(bf_1\rightarrow busses),\ (bf_2\rightarrow busses)\}$$

$$\boldsymbol{time}=\{(kf_1\rightarrow 2),\ (kf_2\rightarrow 5),(of_1\rightarrow 1),\ (rf_1\rightarrow 3),\ (rf_2\rightarrow 8),\ (sf_1\rightarrow 2),\ (sf_2\rightarrow 8),\ (bf_1\rightarrow 3),(bf_2\rightarrow 8)\}$$

$$\boldsymbol{dep}=\{(kf_1\rightarrow\{of_1\}),(kf_2\rightarrow\{of_1\}),(rf_1\rightarrow\{kf_1\}),\ (rf_2\rightarrow\{kf_2\}),\ (sf_1\rightarrow\{kf_1\}),\ (sf_2\rightarrow\{kf_2\}),\ (bf_1\rightarrow\{kf_1\}),\ (bf_2\rightarrow\{kf_2\})\}$$

Definition A subset S of F is *indexable* iff

$$\forall f,f'\in S,\ f\neq f'\rightarrow ds(f)\neq ds(f')$$

Meaning that files need to be associated with different datasets. **Example** the set $\{kf_1,of_1,rf_1,sf_1\}$ is *indexable*

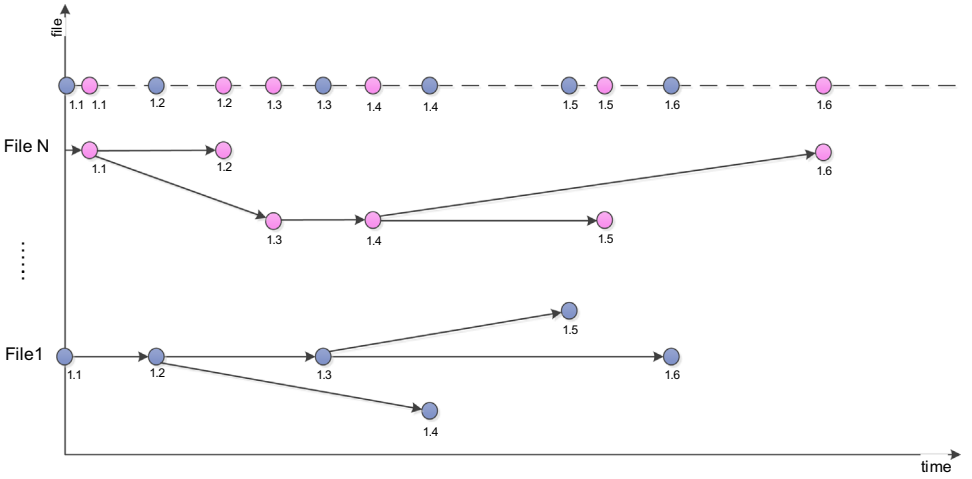

Fig. 2. Example of set of file versioning.

while $\{kf_1, rf_1, rf_2\}$ is not indexable because $ds(rf_1) = ds(rf_2) = roads$.

Definition The function $C: \wp(F) \rightarrow \wp(F)$ associates a subset of F with closure of the subset with respect to the *dep()* function. It can be computed using the recursive function:

$$C(S) = \begin{cases} S \cup C(dep(S)/S) & S \neq \varnothing \\ \varnothing & S = \varnothing \end{cases}$$

where

$dep(S) = \cup_{s \in S} dep(s)$

Example $C(\{rf_1, sf_2\}) = \{kf_1, kf_2, of_1, rf_1, sf_2\}$

Definition Let $I = \{i_1, i_2, i_3, \ldots\} \cup \{\varepsilon\}$ be the set of indexes produced and ε is the empty index. The function $from: I \rightarrow I$ associates an index with the index it was started from and the function $files: I \rightarrow \wp(F)$ associate an index with the set of files to be added to the index we are starting from. Consider that the "*from*" function must not introduce a cyclic dependency among indexes.

Example: $\boldsymbol{I} = \{i_1, i_2, i_3, i_4\}$

$\boldsymbol{from} = \{(i_1 \rightarrow \varepsilon), (i_2 \rightarrow i_1), (i_3 \rightarrow i_2), (i_4 \rightarrow i_2)\}$

$\boldsymbol{files} = \{(i_1 \rightarrow \{kf_1\}), (i_2 \rightarrow \{rf_1, bf_1\}), (i_3 \rightarrow \{sf_1\}), (i_4 \rightarrow \{sf_2\})\}$

Definition Function $\phi: I \rightarrow \wp(F)$ provides for each index the set of files that are indexed, it is defined recursively as:

$$\phi(i) = \begin{cases} \phi(from(i)) \cup files(i) & i \neq \varepsilon \\ \varnothing & i = \varepsilon \end{cases}$$

Example $\phi(i_1) = \{kf_1\}$, $\phi(i_2) = \{kf_1, rf_1, bf_1\}$, $\phi(i_3) = \{kf_1, rf_1, bf_1, sf_1\}$, $\phi(i_4) = \{kf_1, rf_1, bf_1, sf_2\}$

Definition An index $i \in I$ is *correct* if $C(\phi(i))$ is indexable meaning that in the closure of files in the index are not present different versions of files of the same dataset. **Example** the indexes i_1, i_2, i_3 are correct while i_4 is not correct because $C(\phi(i_4)) = \{kf_1, kf_2, of_1, rf_1, bf_1, sf_2\}$ is not indexable.

Since the RDF KB building is an evolving process, it is not possible to predict whether one has to keep a specific previously created version of the index or not. Any small change could be used to generate a new version, while the suggestion is to save versions every time a consolidated point is available similarly to virtual machine snapshots. Moreover, since the triples associated with each single data set are accessible, reconstruction of partial intermediate versions are also possible, saving time in generating triples. Furthermore, each times some ontologies change, most of triples must be generated again, and therefore, for the same dataset, more triples versions could exist.

3.1. Requirements for ***RDF Index Manager tool***

On the basis of the above presented model, the RDF KB indexing versioning activities described can be supported by means of an RDF Index Manager (RIM), that should allow

- keep tracing *RDF KB Store Versions, RKBSV,* in terms of *files of triples*, *index-description*, and RDF *Index*,
- maintaining a repository of *RKBSVs* where they could be stored and retrieved,
- selecting a *RKBSV* from the repository for modification, to examine changes and the history version, to be used as base for building a new version,
- managing the *index descriptor* as a list of files containing triples,
- generating a RDF KB index on the basis of an *RKBSV* independently from the RDF store kind automatically, and in particular for SESAME OWLIM and Virtuoso,
- monitoring the RDF KB index generation and the feeding state,
- suggest the closest version of the *RKBSV* with respect to the demanded new index in terms of files of triples, and
- avoiding manually managing the script file of indexing, since it is time consuming and an error prone process.

4. RDF Index Manager tool

The RDF Index Manager tool satisfy the above presented requirements, creates and manages *index descriptors*, and files of triples, and generates automatically the corresponding *indexes* independently from the RDF store type. The *index descriptor*, as mentioned before, is a list of ontologies and related data sets described with their triple files and version. The chosen approach with generation and update is to: (i) build the entire index (*build all*) by loading triples when ontologies and related data set change, (ii) extend the index when only new data sets and triples have to be added *(incremental building)*, (iii) make a physical copy (*clone*) of a consolidated RDF index when an index descriptor is built starting from an older consolidated descriptor. The big amount of triples to load in the index suggested exploiting the bulk data loading supported by many RDF stores.

The main functionalities provided by the tool are described as following: setup of a new index descriptor, to create an empty index descriptor; clone a previous index descriptor to create a new version that it is populated with the same data sets and triples version of the parent with some addition. A clone of the parent RDF index is made and used to build the new store loading the new additional data sets; copy a previous index with updated versions to create a new version populated with same data sets of parent and new versions of triples. This allows speeding up the creation of an update version of the index descriptor. A new RDF index will be created and loaded from scratch; edit the index descriptor to add a data set (ontology, static, historical and reconciliations), select triples version; update triples version of a data set; remove a data set; Import/Export the index descriptions as XML representations that could be used for backup/restore and share; RDF Index Generation by producing a scripted procedure (for Windows and Linux) according to the index descriptor and the selected RDF store kind. The procedure may be incremental or for reconstructing the index from scratch; monitoring the RDF Index Generation by controlling the store feeding as: the queue of data sets to be loaded, the data sets already in the store, time indicators (time spent, max time to upload a data set, etc..),

progression and output of building process; logging building data related to RDF store building for further access (i.e. statistical and verification analysis).

4.1. Architecture, RDF Index Generation and evolution

The RDF Index Manager is constituted by the following components. The *RDF Store Manager* manages different versions of RDF Stores exploiting the *Version Manager* which provides the triples files version for all the data sets. The *Index Manager API REST Interface* consists of a set of REST calls to be invoked by the indexing script during the RDF store building to keep trace of the indexing process status. The *Index Builder Manager* generates the scripts according to the RDF Store kind. The section contains a list of ontologies/files and each file is described by: an unique identifier corresponding to the name, the reference to the index, the version of triples to use, the operation to perform add, update, remove and commit, and an entity for setting if it was inherited by a cloning (Clone). The historical data differs from other sections for the presence of time interval that defines the triples to use (date and time for TripleStart and TripleEnd).

For the RDF Index generation the RDF Index Manager produces a script according to the index descriptor and the RDF store target. The script is structured in the following steps: (i) setup of script, (ii) initialization of RDF store, (iii) bulk uploading of triples into the store, (iv) RDF store finalization, (v) create possible additional indexes as textual indexes, geographical indexes that need additional database commands, and (vi) update index building status.

The RDF Index Manager has been realized as a PHP 5.5.x web application with MySQL support running under Ubuntu. The Fig. 3 shows the Building Monitor View when a batch script is running. This view provides different information panels: the output of script in real-time on top, the queue of data set to insert, the progress and the total time spent for the committed data set. Such information allows also evaluating the time necessary to build a repository using the two RDF Stores.

5. Experimental results

In Table 1, examples of results are reported. The data refer to the comparison of the usage of the RIM and versioning in building a Smart City RDF store. The RDF stores currently managed are Virtuoso 7.2 as open source RDF store and the commercial OWLIM SE ver. 4.3 and GraphDB 6.1.

The measures reported have been performed by means of an incremental building of the RDF Store for the three solutions. The building started with 12 files of triples including ontologies (first column), then each column of the table refers to the added triples/files (street graphs, smart city services, enrichment and reconciliations, historical data of real time data for 1 month). The time estimated for the cases of total indexing include: create, load, finalize; while those for incremental indexing include: clone, load, finalize. The three RDF store kinds have a different behavior. OWLIM and GraphDB create inferred triples at the indexing; this determines a higher number of triples with respect to Virtuoso, i.e., 73.4 wrt 46.2 million; and a higher indexing

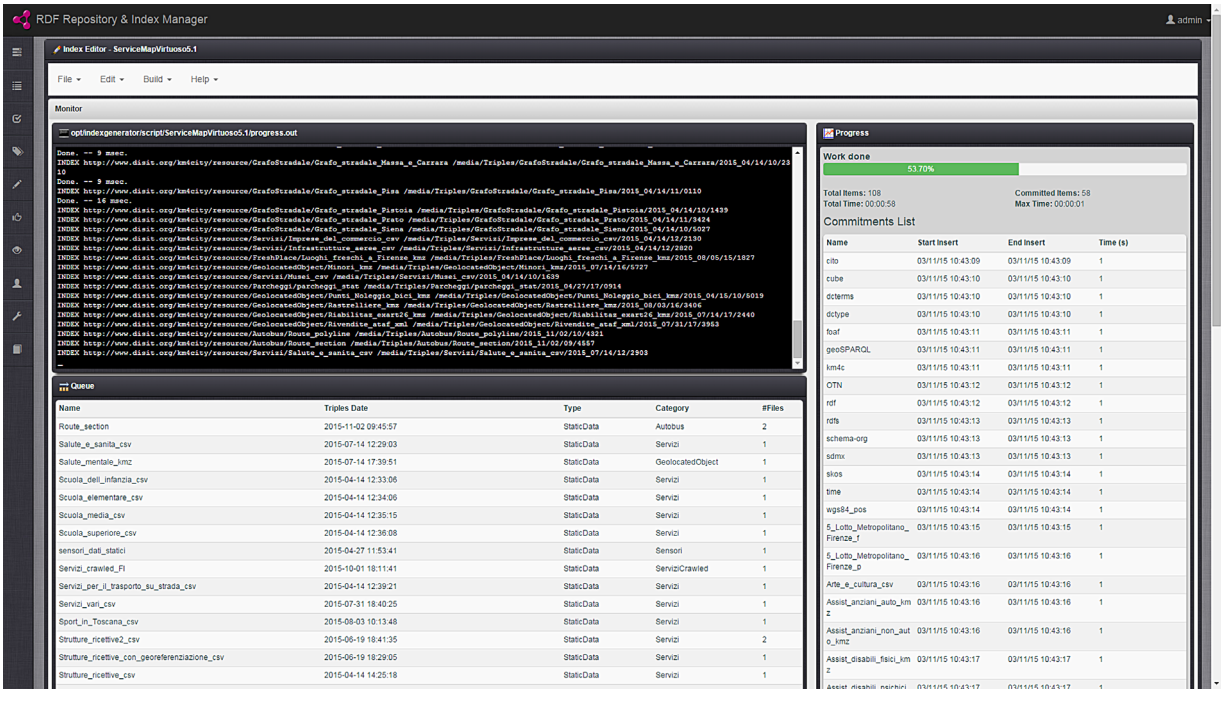

Fig. 3. RDF index building monitor.

Table 1
Saving time using Index Manager with respect to rebuilding. Data collected on Ubuntu 64 bit, 16 core x 2 GHz, 500 Gbyte HD.

	Ontologies	**+Street graphs**	**+Smart city services**	**+Enrich and reconciliations**	**+Historical data 1 month**
Indexing process					
Final number of triples	15,809	33,547,501	34,462,930	34,557,142	44,218,719
Final number of files	12	137	178	185	27,794
Added triples with respect to previous version	15,809	33,531,692	915,429	94,212	9,661,577
Added files with respect to previous version	12	125	41	7	27,609
OWLIM SE 4.3					
Indexing time without RIM (s)	18	6536	6198	7516	12,093
Indexing time with RIM (s)	11	6029	514	343	5745
% of saved time, RIM versioning	**38.9**	**7.8**	**91.7**	**95.4**	**52.5**
Final number of triples (including geo+inferred)	16,062	57,486,956	59,395,432	59,486,748	73,441,126
Disk space in Mbyte	310	8669	8936	9039	13,110
VIRTUOSO 7.2					
Indexing time without RIM (s)	146	806	964	1000	2487
Indexing time with RIM (s)	156	833	421	296	1932
% of saved time, RIM versioning	**−6.8**	**−3.3**	**56.3**	**70.4**	**22.3**
Final number of triples (including geo, no inferred)	21,628	35,452,613	36,301,322	36,420,445	46,232,510
Disk space in Mbyte	68	1450	1632	1631	2294
GraphDB 6.1					
Indexing time without RIM (s)	9	7818	7929	7671	12,915
Indexing time with RIM (s)	2	6791	454	214	4849
% of saved time, RIM versioning	**77.8**	**13.1**	**94.3**	**97.2**	**62.45**
Final number of triples (including geo+inferred)	15,809	57,486,415	59,394,891	59,487,551	73,441,929
Disk space in Mbyte	96	4276	4466	4643	5714

time. In both cases, the percentage of saved time, for non-small RDF stores, is very high, greater than the 22% up to the 97% of saved time. For small stores, Virtuoso can be indexed in shorter time, and thus it could be better to rebuild instead of cloning and versioning.

6. Conclusions

Graph databases are used in many different applications: smart city, smart cloud, smart education, etc., where large RDF KB store are created with ontologies, static data, historical data and real time data. Most of the RDF stores are endowed with inferential engines that materialize some knowledge as triples during indexing or querying. In these cases, the delete of concepts may imply the removal and change of many triples, especially if the triples are those modeling the ontological part of the knowledge base, or are referred by many other concepts. For these solutions, the graph database versioning feature is not provided at level of the RDF stores tool, and it is quite complex and time consuming to be addressed as black box approach. In most cases, the RDF store rebuilt by indexing is time consuming, and may imply manually edited long scripts that are error prone. In order to solve this kind of problem, in this paper, a lifecycle methodology and our RIM tool for RDF KB store versioning are proposed. The results have shown that saving time up to 95% are possible depending on the number of triples, files and cases to be indexed.

Acknowledgment

The authors would like to thank to the coworkers who have contributed to the experiments in the several projects, and in particular to Km4City: Giacomo Martelli, Mariano Di Claudio. Thanks also to Ontotext for providing a trial version of their tools.

References

[1] C. Grosan, A. Abraham, Intelligent Systems: A Modern Approach, Springer-Verlag, Berlin, 2011.
[2] G. Klyne, J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax-W3C Recommendation (2004).
[3] P. Bellini, M. Di P. Nesi N. Rauch, Tassonomy and review of big data solutions navigation, In: Rajendra Akerkar (Ed.). Big Data Computing, Western Norway Research Institute, Norway, Chapman and Hall/CRC Press 2013 ISBN 978-1-46-657837-1.
[4] T. Berners-Lee, Linked Data, ⟨http://www.w3.org/DesignIssues/LinkedData.html⟩, 2006.
[5] C. Bizer, A. Jentzsch, R. Cyganiak, State of the LOD cloud, ⟨http://lod-cloud.net/state/⟩ (retrieved 05.07.14).
[6] O. Hartig, C. Bizer, J.C. Freytag, Executing SPARQL queries over the web of linked data, in: Proceedings of the ISWC'09, Springer, 2009, 293–309.

[7] N.F. Noy, M.A. Musen., Ontology versioning in an ontology management framework, IEEE Intell. Syst. 19 (4) (2004) 6–13.
[8] M. Fernandez Lopez, Overview of methodologies for building ontologies, In: Proceedings of the IJCAI99 Workshop on Ontologies and Problem-Solving Methods: Lessons Learned and Future Trends, Stockholm, 1999.
[9] Feras A. Batarseh, Avelino J. Gonzalez, Incremental lifecycle validation of knowledge-based systems through CommonKADS, IEEE Trans. Syst. Man Cybern.: Syst. 43 (3) (2013) 643–654.
[10] L. Milette, Improving the Knowledge-Based Expert System Lifecycle, UNF Report, 2012.
[11] M. Klein, D. Fensel, A. Kiryakov, D. Ognyanov, Ontology versioning and change detection on the web, In: Proceedings of the 13th European Conference on on Knowledge Engineering and Knowledge Management (EKAW02), Springer, 2002, 197–212.
[12] Natalya F. Noy, Deborah L. McGuinness, Ontology development 101: A guide to creating your first ontology, Stanf. Med. Inform. (2001).
[13] M. Volkel, W. Winkler, Y. Sure, S.R. Kruk, M. Synak, SemVersion: a versioning system for RDF and ontologies, In: Proceedings of the 2nd European Semantic Web Conference, ESWC'05, Heraklion, Crete, May 29–June 1, 2005.
[14] Yannis Tzitzikas, Yannis Theoharis, Andreou Dimitris, On Storage Policies for Semantic Web Repositories That Support Versioning. The Semantic Web: Research and Applications, Springer, 2008, 705–719.
[15] D. Zeginis, Y. Tzitzikas, V. Christophides, On the foundations of computing deltas between RDF models, In: Proceedings of the 6th International Semantic Web Conference, ISWC/ASWC'07, Busan, Korea, 2007, 637–651.
[16] P. Bellini, M. Benigni, R. Billero, P. Nesi, N. Rauch, Km4City ontology bulding vs data harvesting and cleaning for smart-city services, Int. J. Vis. Lang. Comput. (2013).
[17] R. Isele, C. Bizer, Active learning of expressive linkage rules using genetic programming, Web Semantics: Sci. Serv. Agents World Wide Web 23 (2013) 2–15.
[18] A.C.N. Ngomo, S. Auer, Limes – a time-efficient approach for large-scale link discovery on the web of data, Integration 15 (2011) 3.