Replication and Versioning of Partial RDF Graphs

Bernhard Schandl

University of Vienna
Department of Distributed and Multimedia Systems
bernhard.schandl@univie.ac.at

Abstract. The sizes of datasets available as RDF (e.g., as part of the Linked Data cloud) are increasing continuously. For instance, the recent DBpedia version consists of nearly 500 millions triples. A common strategy to avoid problems that arise e.g., from limited network connectivity or lack of bandwidth is to replicate data locally, therefore making them accessible for applications without depending on a network connection. For mobile devices with limited capabilities, however, the replication and synchronization of billions of triples is not feasible. To overcome this problem, we propose an approach to replicate parts of an RDF graph to a client. Applications may then apply changes to this partial replica while being offline; these changes are written back to the original data source upon reconnection. Our approach does not require any kind of additional logic (e.g., change logging) or data structures on the client side, and hence is suitable to be applied on devices with limited computing power and storage capacity.

1 Introduction

The RDF data model has been designed to facilitate the publication of semantically meaningful data about resources on the Web. It is innately intended to be used in a decentralized, distributed, and uncontrolled manner. Because of these requirements, RDF has been designed so that data from different sources can be easily merged and integrated, its vocabulary is extensible, and it employs an open world semantics, which essentially means that data consumers (applications or end users) can never be assured that they are aware of all relevant information.

According to current estimates[1], the Linked Data cloud consists of more than 13 billions RDF triples, which are distributed across hundreds of sources. They can be accessed using a variety of means, ranging from directly de-referencing HTTP URIs, over issuing selective queries via SPARQL, to downloading data dumps and deploying them in local triple stores. It is a matter of the concrete application which method is the appropriate one, since they expose different characteristics w.r.t. efficiency and performance.

[1] Linked Data Set Statistics: http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics

Especially in the mobile domain it is very unlikely that an application will access all of them online, since mobile network connectivity is not always available for reasonable prices. To overcome this problem, and to decrease response times, data from remote sources can be replicated locally, and applications can operate on these local copies. However, it is not practical to duplicate several billions RDF triples to a mobile device with limited computing power and memory capacity, and often this is not required at all for a specific application; e.g., because the application is only able to operate on resources of certain types.

The replication of data from external sources, which can be selected based on the specific application context, is relatively straightforward for read-only data. However, we expect the Semantic Web to evolve into a read-and-write system; this claim is supported by the currently ongoing efforts towards a standardization of update functionality for the SPARQL query language [1], or the specification of write-back mechanisms for non-RDF data sources[2].

When applications modify partially replicated data, the problem of synchronization upon reconnection becomes apparent. The computation of a *diff* (i.e., sets of added and removed triples) involving partial RDF graphs is not straightforward, even if we leave aside the problems that blank nodes impose on comparing RDF graphs [2, 3]. RDF is set-based and does not provide opaque identifiers for single triples, therefore they can only be identified by explicitly and fully naming their subject, predicate, and object. Hence it is difficult to compare incomplete RDF graphs (which is required for merging changes from different sources) without additional information.

One can overcome this problem by instantiating services that monitor and track changes to RDF graphs and therefore make added and removed triples explicit; e.g., by marking triples as deleted instead of physically removing them. However this approach requires hooking into existing software infrastructure used by applications (in particular, triple stores) and is therefore not applicable in many scenarios. In this paper, we present an approach how parts of RDF graphs can be replicated to clients for local modification, whereas the replicas are enriched with triples that describe which parts of the (full) base graph are missing. For this, binary strings are added to the partial replica; these strings are mapped to an ordering of the triples in the base graph. Upon reconnection, this additional information can be used to identify which triples have been added and removed without the need for additional tracking infrastructure on the client side.

In the following, we outline the requirements and the general design of our approach (Section 2). We introduce the concept of triple bitmaps, with which partial graphs are enriched in order to allow the computation of diffs (Section 3), and we explain how changes to partial graphs can be merged with the original data (Section 4). We present some implementation details (Section 5) and address several known limitations of our approach, as well as general problems in the context of replicating and merging RDF graphs (Section 6). We conclude with a discussion of related work (Section 7).

[2] pushback: `http://esw.w3.org/topic/PushBackDataToLegacySources`

2 Approach and Model

2.1 Requirements

In this section, we introduce the requirements that have led to the specification of our partial graph replication algorithm. As stated before, our general objective is to allow Semantic Web-based applications (which we denote as *clients* in the following) to replicate parts of base RDF graphs (which are stored in a remote *repository*), to make changes to these local replicas, and to synchronize changes back to the base graphs upon reconnection. In particular, the framework aims to fulfill the following requirements:

- *No additional client-side processing.* Our approach should be applicable to environments with limited processing and storage capacity, like mobile devices and handhelds. Moreover, the approach should be compatible with any RDF storage system on the client's side and should therefore not require modifications or hooks. It should also not rely on mechanisms that track changes on the client side, e.g., by marking deleted triples.
- *No additional data structures on the client side.* For the same reasons (namely, avoiding to interfere with client-side RDF storage and processing infrastructure), and to reduce client-side workload, it is desirable that the approach does not require additional data structures beyond the RDF model on the client side. For the server side, we do not impose this requirement since here usually more resources are available, and server environments are more easy to control and maintain than distributed client systems.
- *Support for light-weight clients.* Subsuming the previous two requirements, our approach should be designed in a manner that allows it to be applied in mobile environments.
- *Stateless graph repositories.* The repository (which holds the base graphs to be replicated to clients) should not require to maintain status information (e.g., information about partial graphs) about clients in order to keep a separation of concerns between the repository and the clients.
- *Flexibility w.r.t. subgraph selection.* The approach should not assume a certain graph structure, or a specific ratio of the sizes of the base graph and the partially replicated graph; it should be able to handle any such ratio with acceptable performance and scalability.

2.2 Workflow

In the design of our approach we follow the naming convention of the popular *Subversion* system[3] (SVN) for file versioning. In SVN, a server (called *repository*) hosts a set of versioned files, which are arranged in a hierarchical directory structure. A client can transfer these files, or a subset thereof, to a local *working copy*; this step is called *checkout.* This working copy is enriched with metadata containing information about its base revision, which are stored in additional

[3] Subversion: `http://subversion.tigris.org`

hidden files. Then, modifications are applied to the working copy. The client can *update* its working copy with recent changes from the repository at any time. During this step, modifications from other clients (which have been applied to the repository) are *merged* with the client's working copy. When concurrent changes are merged, *conflicts* can occur, which have to be resolved on the client side. The client's modifications are then transferred to the repository in the course of the *commit* action.

For the specification of our partial graph versioning system, we follow the naming convention of SVN. While SVN treats files as atomic units of versioning, we apply versioning to *(named) RDF graphs*. However, as described before our approach additionally enables clients to checkout *partial RDF graphs*, which has significant impact on the workflow of versioning:

- First of all, our approach should be applicable to devices with limited computing power and memory, hence we operate with partial graphs on the client side. However, the same restriction applies to updates that are transmitted from the repository to the client: as these updates can as well become very large we must ensure that clients are not overburdened with calculation of diffs and merging operations.
- As we will see later, the lack of client-side infrastructure (as defined in the requirements) leads to the problem that the client is not aware of its partial RDF graph's characteristics: for instance, after a partial checkout the client cannot decide which fraction of the original graph was retrieved. Therefore, partial graphs cannot be merged on the client.
- The detection of conflicting modifications (which can occur on the syntactical or the semantic layer) requires domain knowledge and often involves relatively expensive reasoning. The need for partial reasoning has already been recognized [4], and there exist proposals how this can be accomplished on mobile devices (e.g., [5]). However, complete underlying data is required for this task, which in our scenario is the case only on the repository.

Because of these reasons we have specified a modified replication and versioning workflow for partial RDF graphs, as depicted in Figure 1. The most significant change is the shift of merge operations from the client to the repository; this is due to the fact that only the repository has complete information about graphs and modifications, while clients have only partial information.

A second significant change is that clients cannot receive updates from the repository, because this would again require capabilities for merging and conflict detection on the client side. Instead, clients must always commit their changes to the repository and can then checkout a new partial graph. Consequently, the lifetime of a working copy is limited to one checkout-commit cycle.

2.3 Partial Graphs

The key concept in our approach is the enrichment of *partial graphs* with *triple bitmaps*. A partial graph contains a subset of the triples of a given RDF graph

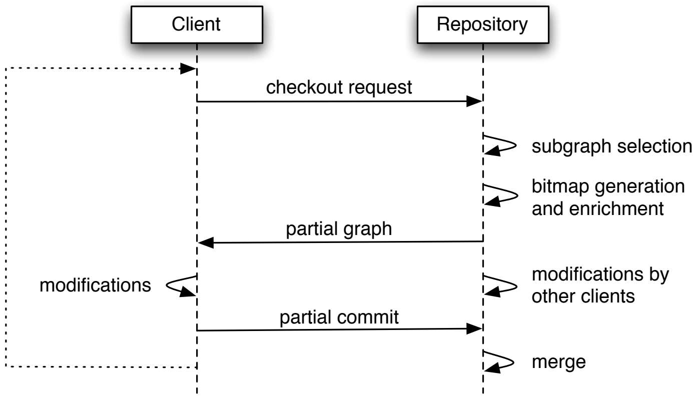

Fig. 1: Partial Graph Versioning Workflow

(the *base graph*). Triple bitmaps encode which triples from the base graph are missing in the partial graph. Partial graphs are enriched with the necessary triple bitmaps and are transferred to the client, which applies changes to it. When the modified partial graph is committed, the repository is able to analyze the submitted triple bitmaps and infer which triples from the base graph were missing in the original partial graph, and consequently can infer which modifications were applied to the partial graph. In the following we formally describe our approach.

We define an RDF graph $G \in \mathbb{G}$ as a set of tuples $(s, p, o) \in \mathbb{UB} \times \mathbb{U} \times \mathbb{UBL}$, called *triples* [6], where $\mathbb{U}$ is the set of all URIs, $\mathbb{B}$ the set of all blank nodes, and $\mathbb{L}$ the set of all literals. We denote with S_G the set of *subjects* in G, i.e., all elements that appear at the subject (s) position of any triple in G (according to the RDF specification, this can only be URIs and blank nodes); therefore $S_G \subset \mathbb{UB}$.

For *graph equivalence* between two graphs G and G' we reuse the definition from [7], which requires a bijection $\mathcal{M}$ that maps all elements (URIs, literals, and blank nodes) from G to corresponding elements in G', and vice versa [3]. While this mapping is straightforward for URIs and literals, it may be problematic for blank nodes, which must be *grounded* in order to ensure that two graphs can be compared (cf. Section 6). With “grounding” we denote the process of assigning unique identity to blank nodes.

A *partial RDF graph* $\overline{G}$ derived from its *base graph* G is an RDF graph that consists of two groups of triples: (*1*) the set of *base triples* $\overline{G}_b$, i.e., triples that are also contained in G; and (*2*) the set of *bitmap triples* $\overline{G}_t$ of the form $< s', p, o >$, where each s' is derived from an element $s \in S_G$, $p =$ `tbt:bitmap`[4], and o is a *triple bitmap literal*, related to s, G and $\overline{G}$, which we denote with $B_{(s,G,\overline{G})}$. Therefore, $\overline{G} = \overline{G}_b \cup \overline{G}_t$, where $\overline{G}_b \subseteq G$ and $\overline{G}_t = \bigcup_{s \in S_{\overline{G}_b}} B_{(s,G,\overline{G}_b)}$.

[4] `tbt:bitmap` is the abbreviated notation of a specific property that represents a resource’s triple bitmap.

3 Triple Bitmaps

A *triple bitmap* $B_{(s,G,\overline{G})}$ is a binary bitmap that represents the presence (0) or absence (1) of the triples of an RDF graph G within a partial RDF graph $\overline{G}$. A triple bitmap is determined by a tuple $(s, G, \overline{G})$, where s is an RDF language element that appears as the subject of at least one triple in both G and $\overline{G}$; i.e., $s \in S_G$ and $s \in S_{\overline{G}}$. The construction of such a triple bitmap is described in Algorithm 1. It uses a *monotonic sequential ordering* $O(G, s)$ over all triples in G for which s is on the subject position (i.e., S_G)[5]. This ordering must be recoverable for each revision of G, since $O(G, s)$ will change when triples are added to, or removed from G.

```
Input: RDF element $s$, graph $G$, partial graph $\overline{G}$
Output: Triple bitmap $B_{(s,G,\overline{G})}$
bitmap $\leftarrow$ empty bitmap with size $|O(G, s)|$ ;
forall triples $t_i$ in $O(G, s), 0 \leq i \leq |O(G, s)|$ do
    if $t_i \in \overline{G}$ then bitmap $[i] \leftarrow 0$ ;
    else bitmap $[i] \leftarrow 1$ ;
    i $\leftarrow$ i + 1 ;
end
```

Algorithm 1: Construction of a Triple Bitmap

We have chosen to build bitmaps based on distinct subjects, instead of distinct predicates or objects. This design decision is based on an analysis of typical RDF data found on the Web, which exhibit a certain ratio between the number of distinct resources and the number of triples in which each resource participates. For instance, in the 2009 Billion Triple Challenge dataset[6] only 1.3 millions out of more than 128 millions distinct subjects participate in more than 100 triples, and only ≈25,000 subjects appear in more than 1,000 triples (cf. Table 1). The vast majority of subjects occur in 2 to 10 statements. The distribution of distinct objects is entirely different: two thirds of all distinct objects appear in only one statement. Constructing bitmaps based on triple objects would therefore lead to a large number of bitmaps consisting of only one bit. The number of distinct predicates is far below the number of subjects or objects, which leads to a high number of statements per distinct predicate; in this case triple bitmaps would become very long.

[5] Such an ordering can be defined for all RDF graphs: either, the underlying storage mechanism already provides a *natural ordering*; if this it not the case a generic lexical sorting as proposed e.g., in [8] can be employed.

[6] 2009 Billion Triple Challenge: `http://vmlion25.deri.ie`

# of Participating Triples	Distinct Subjects		Distinct Predicates		Distinct Objects	
Total	128,079,322	(100.00%)	136,188	(100.00%)	279,710,101	(100.00%)
1	9,873,704	(7.71%)	23,222	(17.05%)	189,702,670	(67.82%)
2–10	99,168,416	(77.43%)	50,029	(36.74%)	82,011,684	(29.32%)
11–100	17,734,849	(13.85%)	38,812	(28.50%)	7,247,533	(2.59%)
100–1,000	1,276,612	(1.00%)	15,947	(11.71%)	704,735	(0.25%)
≥1,001	25,741	(0.02%)	8,178	(6.00%)	43,479	(0.02%)

Table 1: Billion Triple Challenge 2009 data set statistics: distribution of participating triples per distinct subject, predicate, and object

A similar distribution can be found in a highly important Linked Data source, the DBpedia 3.3 dataset[7]. Of 25,455 randomly selected subjects, none appears in only a single triple, while the vast majority (89.85%) appeared in 11 to 100 triples; 10.12% occur in 100 to 1,000 triples. Contrary, of 639,321 randomly selected objects, 634,198 appear in 10 or less triples, therefore the number of bitmap triples would be very high, which causes additional storage overhead.

3.1 Serialization of Triple Bitmaps

As we will see later, triple bitmaps are encoded as RDF literals in order to enrich a partial graph. For this purpose, we can encode a triple bitmap $b_{(s,G,\overline{G})}$ as a plain string consisting of a sequence of zero and one digits. Since this is quite verbose, and to facilitate interoperability with RDF-related standards, we apply *base64*-encoding [9] to the bitmap. This encoding is supported by a designated XML data type (`xsd:base64Binary`) and is recommended for representing arbitrary binary content in RDF [10]. To determine the precise length of the bitmap, it is padded with a 1 before the most significant bit; then the bitmap is padded with zeroes to fit the 6 bit pattern for base64-encoding. Figure 2 depicts the subsequent steps in the encoding process. In this example, four out of 21 statements from the original graph are not present in the partial graph; therefore the triple bitmap contains four 1s (positions 4, 8, 11, and 12).

Bit #				21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Original Bitmap				0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0
1-padded Bitmap			1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0
base64–padded Bitmap	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0
base64-encoded Bitmap	E						A						y						I					

Fig. 2: base64-Encoding of Triple Bitmaps

[7] DBpedia: `http://dbpedia.org`

3.2 Enriching Partial Graphs with Triple Bitmaps

We can now enrich the partial graph $\overline{G}$ with the set of bitmap triples $\overline{G}_t$, which contains one triple bitmap triple per distinct subject in $\overline{G}_b$. For the subject we do not directly use s but add a unique prefix[8], forming a new URI s'[9]:

$$\overline{G}_t = \{< s', \texttt{tbt:bitmap}, \texttt{"}B_{(s,G,\overline{G})}\texttt{"\^{}\^{}xsd:base64Binary} > \mid s \in S_{\overline{G}}\} \tag{1}$$

Such an enrichment technique is in line with the RDF formal semantics [11] and has been used in previous works, e.g., for non-deterministic labeling of RDF nodes in order to facilitate signing of RDF graphs [8] (the author calls this technique to "*make meaningless changes to an RDF graph*"). An example of the construction of a triple bitmap and the enrichment of a partial graph is depicted in Figure 3. Here, the base graph consists of four triples of which only two (#1 and #3) are included in the partial graph (non-replicated triples are depicted in grey). The resulting triple bitmap consisting of two zeroes and two ones (`"1010"`), which yields the string `"a"` after padding and base64-encoding.

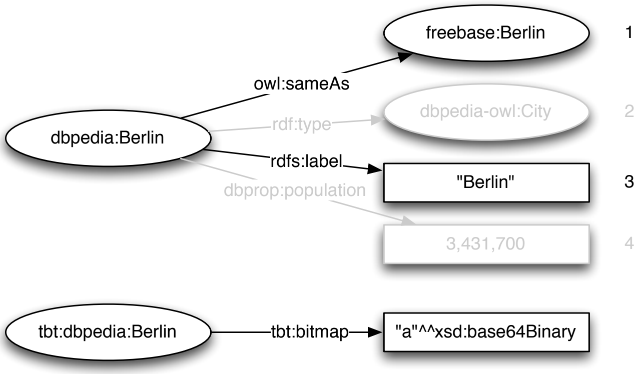

Fig. 3: Graph, partial graph, and bitmap triple

Our approach requires the transmission and storage of one triple bitmap per subject in $\overline{G}$. Bitmaps are encoded as triples, which consist of the subject, a (fixed) predicate, and the serialization of the triple bitmap, whereas each object requires one bit. According to the data presented in Table 1, more than 80% of

[8] Without loss of generality this prefix can be freely chosen, as long as it is guaranteed that it does not conflict with other subject URIs used in $\overline{G}$; this condition can however be ensured by various means, e.g., by introducing a special URI schema (as in our example), or by using HTTP URIs within a domain registered solely for this purpose.

[9] Blank nodes must be grounded beforehand so that they are uniquely identifiable and hence a prefixed URI s' can be determined.

all triple bitmaps are shorter than 16 bits and can thus be represented using two bytes. In the Virtuoso triple store, for instance, each triple requires a storage capacity of around 35 bytes [12], which means that the cost of transmitting and storing additional bits in a triple bitmap is much lower than storing additional triples. Therefore we decided to attach triple bitmaps to subjects rather than to objects.

During a partial checkout the repository sends the partial graph $\overline{G}$ (i.e., a subset of the triples of the base graph G plus one bitmap triple for each distinct subject in $\overline{G}$) to the client, where it is buffered in a local triple store. The revision number of the graph is maintained on the client side as part of the graph URI. Applications can now access and modify these data independent from the network connection.

4 Merging Partial Graphs

The triple bitmap for each subject s in the partial graph $\overline{G}$ allows us to determine whether triples have been added or, more importantly, removed from the partial graph without requiring to explicitly track these change operations. The only condition to be fulfilled by the client is that the triple bitmaps may not be modified or deleted; in this case it would become impossible to retrace which triples from the base graph were missing in the unmodified partial graph. However, since the set of bitmap triples $\overline{G}_t$ does not interfere with the rest of the contents of the partial graph $\overline{G}_b$, and an RDF graph entails all of its subgraphs [11] (in our case, $\overline{G} \models \overline{G}_b$), this condition can usually be maintained.

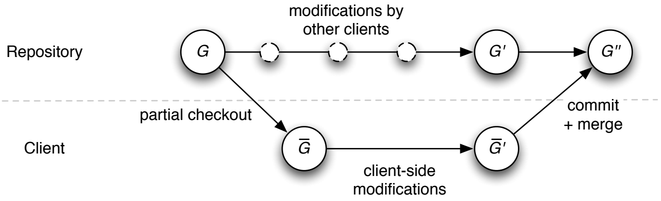

Fig. 4: Evolvement of modified partial graphs

It is unlikely that applications (which are not aware of the special semantics of the bitmap triples) modify or delete these triples "by accident". The typical update operations on RDF graphs as defined e.g., in [1] (i.e., insertion and deletion of triples) have no effects on other triples in the graph. The only exception are *resource deletions*: since in RDF resources can occur only within triples, to "remove a resource" means effectively to remove all triples in which the resource appears. To avoid this, our proposed method to enrich graphs with triple bitmaps is to add a prefix to the actual resource's URI (cf. Figure 3). Thus we create a separate resource for the bitmap triple and therefore reduce the probability of its accidental removal, while this separate resource is still implicitly connected to its base resource through their URI. For the merge operation this implies that if a committed partial graph $\overline{G}'$ contains a bitmap triple with subject s' but no triples with the corresponding subject s, the repository can apply one of two strategies:

1. *Partial resource removal*: the repository removes only triples that were present in the partial graph, according to the triple bitmap for s; or
2. *Complete resource removal*: the repository removes all triples where s occurs in the subject position, regardless of the contents of the triple bitmap.

These strategies represent two different assumptions on the client's intention that the repository can take: in the first case, it assumes that the client intended to remove a set of triples with subject s that is exactly the set of triples that have been checked out in the partial graph $\overline{G}$; in the second case the repository assumes that the client's intention was to remove all triples with subject s.

The decision which strategy to take can be supported by checking whether s is the object of other triples in $\overline{G}'$. If such triples exist, it is unlikely that the client's intention was to remove the resource s entirely since in this case the triples where s stands in the object position would have been removed, too. Therefore, in this case a partial resource removal should be chosen.

Another situation in which bitmap triples may be accidentally removed are bulk deletions that affect multiple subject resources. Let us assume that a committed partial graph $\overline{G}'$ contains triples with a subject resource s without a corresponding triple bitmap statement. Let us further assume that s was already present in the base graph G, but $\overline{G}'$ contains statements with subject s that were not present in G. In this case, the repository cannot decide whether the original partial graph $\overline{G}$ contained statements that have been removed and are therefore not present in $\overline{G}'$. Again, one of two strategies can be applied:

1. *Optimistic resource removal*: the repository removes all triples with subject s that are not present in $\overline{G}'$; or
2. *Pessimistic resource removal*: the repository removes only those triples for which a triple bitmap is present in $\overline{G}'$, and leaves all other triples untouched, including triples with subject s that are not present in $\overline{G}$.

An extreme case of this class of operations is the removal of all triples in the graph; i.e., $\overline{G}'$ is empty. Such a situation cannot be handled using any kind of the heuristics mentioned before; in this case the repository has to decide whether to ignore the commit and leave G in its previous state, or to interpret the commit as total delete and remove all triples from G.

As an alternative to the a-priori selection of a certain removal strategy, the repository can reject the modification and return it to the client. This is a similar behavior as in SVN, where certain types of conflicting concurrent modifications must be resolved manually by the user.

The merge procedure (cf. Figure 4) considers three graphs as input: the modified partial graph $\overline{G}'$ as received from the client, the base graph G from which the original partial graph $\overline{G}$ was extracted, and the current revision of G, denoted as G'. Since the contents of the original partial graph (i.e., $\overline{G}$) can be computed using the triple bitmaps contained in $\overline{G}'$, we can apply a standard RDF diff algorithm [2] to detect and compare changes. Its result is a merged graph G'' which reflects all changes applied to G. In case of unresolved conflicts, the algorithm terminates without returning a merged graph, and the commit must be revised by the client (e.g., through an appropriate user interface). Conflicts can be detected by comparing either the two changesets $C_{G,\overline{G}'}$ and $C_{G,G'}$ (whereas a changeset C consists of one set of added triples C^a and one set of removed triples C^r [7, 13, 14]), or by comparing G' with the full graph that can be reconstructed from $\overline{G}'$. Algorithms for conflict detection on various levels of semantics have already been presented (e.g., [15]) and are out of the scope of this paper.

5 Implementation

We have implemented the presented approach as part of the MobiSem Replication Framework[10]. The prototype is based on the Jena Semantic Web library and provides the necessary methods to extract parts of an RDF graph, to compute and serialize triple bitmaps, and to reconstruct graphs based on partial commits. Modules for blank node grounding and conflict detection can be plugged into the framework since these tasks are highly depending on the schema that is used in the data; our implementation does not impose restrictions on these aspects.

6 Limitations and Discussion

Our proposed algorithm is able to cope with changes to partial RDF graphs, as long as the triple bitmaps that are attached to the partial graph remain intact. Even in situations where subjects without corresponding triple bitmap exist (or vice versa) the algorithm is able to terminate; however in this case assumptions on the client's intentions must be made. If this is not desirable, the repository can revoke changes which are not clearly resolvable.

As shown in Section 3, the computation, transmission, and storage of triple bitmaps is feasible under consideration of the structure of real RDF data sets. In extreme cases, however, this approach may become inefficient. First, if the base graph contains a large number of subject resources but only very few triples per subject, a large number of short triple bitmaps is added to the partial graph. On the other hand, if the base graph contains only few subjects while each subject is described by a large number of statements, triple bitmaps become very large and may become unmanageable. In these cases a modified variant of our approach can be applied where triple bitmaps are not attached to subjects, but to predicates or objects, depending on the structure of the data. This can be done without loss of functionality as long as the repository's interpretation of bitmap triples for each graph remains consistent over time.

[10] MobiSem Project: `http://www.mobisem.org`

A general problem in the context of RDF versioning and replication is the treatment of blank nodes. The comparison of two graphs requires blank nodes in the graphs to be matched, so that changes in the triples where these nodes occur can be detected. A number of solutions for blank node grounding have been discussed in the literature (including node renaming [7], usage of existing or artificial inverse-functional properties [16], explicit identity assertions like `owl:sameAs` and `owl:equivalentClass`, or feature vector comparison [17]). Our approach does not consider blank nodes differently from named nodes, therefore it requires an additional strategy to ground blank nodes as part of the calculation of diffs between the base graph and a committed partial graph.

A second problem in this context is the detection of conflicts that occur due to concurrent modifications to the same graph. Various approaches to tackle conflicts on the structural or semantic layer have been presented in the literature (e.g., [7, 15, 18]). However, this issue is out of the scope of our algorithm; instead we provide a hook for a conflict detection function as part of our merge algorithm.

7 Related Work

The problems of versioning RDF graphs and tracking changes in the context of Semantic Web-based information systems have been acknowledged early. Kiryakov and Ognyanov [13] have introduced the basic foundations of versioning w.r.t. the semantics of the RDF model; based on their work models and ontologies for *RDF deltas* have been specified [2]. The characteristics of such deltas under the conditions of RDF Schema semantics have been analyzed [14]. Efficient storage structures [19] and aggregation algorithms [7] for versioned triple data were designed, and the semantics of graph merge operations have been studied in detail [20]. This research has lead to a number of concrete systems and frameworks, including SemVersion [16], the discontinued Graph Versioning System (GVS)[11], and the Talis Platform[12].

On a higher level, versioning can also be applied to knowledge bases under consideration of the semantics imposed by the underlying ontology language; an example of a complete framework for such high-level changes is presented by Plessers et al. [21]. Papavassiliou et al. [18] introduce the notion of higher-level changes to RDF graphs that are validated against preconditions. Similar to [7] they propose to aggregate atomic changes to higher-level composite changes and provide a reasoning-based algorithm to detect them.

The need for replication of RDF data has likewise been addressed in previous works: for instance, the Boca Semantic Web framework [22] provides support for graph replication and synchronization, both in real-time and in batch mode. This is accomplished on the named graph level, therefore it does not allow for the replication of arbitrary sub-graphs. Moreover, it requires that modifications are tracked on the client side, which requires special client-side software. In the

[11] Graph Versioning System: `http://gvs.hpl.hp.com`

[12] Talis Platform: `http://www.talis.com/platform/`

relational database world, algorithms for replication and synchronization are already well-established; through the usage of hybrid databases such mechanisms can also be applied to RDF data (as implemented, e.g., within the Virtuoso integration middleware [23]). Replication of RDF data has also been proposed on a peer-to-peer basis [24]. Our approach differs from these works in that it does not require any kind of special infrastructure on the client side; instead, partial graphs are (re)constructed by the repository only.

None of these approaches have explicitly considered the special conditions of management of subsets of RDF graphs in a distributed situation. There exist various approaches for the specification of graph subsets; e.g., Concise Bounded Descriptions [25], RDF Molecules [26], and Minimum Self-contained Graphs [27]; mostly they have been specified to overcome problems in conjunction with blank nodes. Some of these approaches have been applied in the context of versioning (e.g., GVS uses RDF Molecules as versioning subject), but none of them allows for unrestricted selective replication and synchronization of graph subsets. Therefore we consider previous works on modification tracking and versioning of RDF graphs as complementary to our work.

8 Summary and Conclusions

It is reasonable to replicate data sets from remote sources (e.g., the Linked Data Cloud) to mobile devices. This allows users to operate in offline mode, which is helpful in situations with limited connectivity or in situations where data transmission over a network is too expensive or too slow. However, under consideration of the limited resources available on mobile devices it is impractical to replicate large data sets. In this paper we have presented an approach that provides the possibility to replicate subsets of RDF graphs to clients, which can be processed and modified locally, and later written back to the base graph. Subsets of RDF graphs are enriched with triple bitmaps, which indicate the missing triples in the partial replica. Upon reconnection, these triple bitmaps can be utilized to determine which modifications have been made by the client. Hence, our algorithm does not require to set up special infrastructure on the client side, which increases its applicability for a wide range of situations.

The presented algorithm is based on the characteristic of RDF that triples can be added to a graph without interfering with the graph's original semantics. For the algorithm to work correctly it is important to ensure that the injected triple bitmaps are not accidentally modified or deleted. To decrease the probability of this case, triple bitmaps use a dedicated vocabulary and are not explicitly connected to the resources they describe. The algorithm is able to intercept certain error situations; however, in several cases a definite decision cannot be made. In such situations, the repository can either assume certain client intentions and perform corresponding actions, or refuse the modification; in this case the client must solve the ambiguity.

Our algorithm has been designed especially for mobile applications, where one cannot rely on a stable network connection, and because of limited resources the full replication of large data sets is not feasible. In the future we plan to extend the possible application fields, e.g., to collaborative ontology and data authoring, and to integrate the proposed method with further algorithms for conflict detection.

Acknowledgements This work has been funded by FIT-IT grant 815133 from the Austrian Federal Ministry of Transport, Innovation, and Technology. The author thanks Gunnar Aastrand Grimnes for providing statistics of the 2009 Billion Triples Challenge dataset.

References

1. Simon Schenk and Paul Gearon. *SPARQL 1.1 Update (W3C Working Draft 22 October 2009)*. World Wide Web Consortium, 2009. Available at `http://www.w3.org/TR/sparql11-update/`.
2. Tim Berners-Lee and Dan Connolly. *Delta: An Ontology for the Distribution of Differences Between RDF Graphs*. World Wide Web Consortium, 2006. Available at `http://www.w3.org/DesignIssues/Diff`, retrieved 15-Dec-2008.
3. Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax (W3C Recommendation 10 February 2004)*. World Wide Web Consortium, 2004.
4. Dieter Fensel and Frank van Harmelen. Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2):96, 94–95, 2007.
5. Luke Albert Steller, Shonali Krishnaswamy, and Mohamed Methat Gaber. A Weighted Approach to Partial Matching for Mobile Reasoning. In *Proceedings of the 8th International Semantic Web Conference (ISWC 2009)*, 2009.
6. Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. Minimal Deductive Systems for RDF. In *Proceedings of the 4th European Semantic Web Conference (ESWC 2007)*, 2007.
7. Sören Auer and Heinrich Herre. A Versioning and Evolution Framework for RDF Knowledge Bases. In *Ershov Memorial Conference*, volume 4378 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2006.
8. Jeremy J. Carroll. Signing RDF Graphs. In *Proceedings of the 2nd International Semantic Web Conference (ISWC 2003)*, pages 369–384, 2003.
9. S. Josefsson. *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*. Network Working Group, October 2006.
10. Johannes Koch and Carlos A. Velasco. *Representing Content in RDF 1.0 (W3C Working Draft 29 October 2009)*. World Wide Web Consortium, Available at `http://www.w3.org/TR/Content-in-RDF10/`, 2009.
11. Patrick Hayes. *RDF Semantics (W3C Recommendation 10 February 2004)*. World Wide Web Consortium, 2004.
12. Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In Sören Auer, Christian Bizer, Claudia Müller, and Anna V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.
13. Atanas Kiryakov and Damyan Ognyanov. Tracking Changes in RDF(S) Repositories. *Transformation for the Semantic Web KTSW 2002*, 2002.

14. Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On the Foundations of Computing Deltas between RDF Models. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 631–644, Berlin, Heidelberg, November 2007. Springer Verlag.
15. Yinglong Ma and Beihong Jin. An Combination Approach to Tackling Semantic Conflicts Based on RDF Model. In *Third International Conference on Semantics, Knowledge and Grid*, Los Alamitos, CA, USA, 2007. IEEE.
16. Max Völkel and Tudor Groza. SemVersion: An RDF-based Ontology Versioning System. In *Proceedings of the IADIS International Conference on WWW/Internet (ICWI2006)*, volume 1, pages 195–202. IADIS, October 2006.
17. Gunnar Aastrand Grimnes, Peter Edwards, and Alun D. Preece. Instance Based Clustering of Semantic Web Resources. In *Proceedings of the 5th European Semantic Web Conference (ESWC 2008)*, volume 5021 of *Lecture Notes in Computer Science*. Springer, 2008.
18. Vicky Papavassiliou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. On Detecting High-Level Changes in RDF/S KBs. In *Proceedings of 8th International Semantic Web Conference (ISWC 2009)*, 2009.
19. Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou. On Storage Policies for Semantic Web Repositories that Support Versioning. In *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, Heidelberg, June 2008. Springer Verlag.
20. Jeremy J. Carroll. Matching RDF Graphs. In *Proccedings of the 1st International Semantic Web Conference (ISWC 2002)*, pages 5–15, 2002.
21. Peter Plessers and Olga De Troyer. Ontology Change Detection Using a Version Log. In *Proceedings of the 4th International Semantic Web Conference (ISWC 2005)*, pages 578–592. Springer, 2005.
22. Lee Feigenbaum, Sean Martin, Matthew N. Roy, Benjamin Szekely, and Wing C. Yung. Boca: An Open-Source RDF Store for Building Semantic Web Applications. *Briefings in Bioinformatics*, 8(3):195–200, 2007.
23. OpenLink Software Inc. *Virtuoso Replication and Synchronization Services*, 2006. Available at http://virtuoso.openlinksw.com/Whitepapers/html/DMI_Replication_Services.htm.
24. Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *Proceedings of the 11th International Conference on World Wide Web*, pages 604–615, 2002.
25. Patrick Stickler. *CBD – Concise Bounded Description (W3C Member Submission 3 June 2005)*. World Wide Web Consortium, 2005. Available at http://www.w3.org/Submission/CBD.
26. Li Ding, Tim Finin, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. McGuinness. Tracking RDF Graph Provenance Using RDF Molecules. In *Proceedings of the 4th International Semantic Web Conference (ISWC 2005)*, November 2005.
27. Christian Morbidoni, Giovanni Tummarello, Orri Erling, and Reto Bachmann-Gmür. RDFSync: Efficient Remote Synchronization of RDF Models. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 533–546, Berlin, Heidelberg, November 2007. Springer Verlag.