On Designing Archiving Policies for Evolving RDF Datasets on the Web

Kostas Stefanidis, Ioannis Chrysakis, and Giorgos Flouris

Institute of Computer Science, FORTH, Heraklion, Greece
{kstef,hrysakis,fgeo}@ics.forth.gr

Abstract. When dealing with dynamically evolving datasets, users are often interested in the state of affairs on previous versions of the dataset, and would like to execute queries on such previous versions, as well as queries that compare the state of affairs across different versions. This is especially true for datasets stored in the Web, where the interlinking aspect, combined with the lack of central control, do not allow synchronized evolution of interlinked datasets. To address this requirement the obvious solution is to store all previous versions, but this could quickly increase the space requirements; an alternative solution is to store adequate deltas between versions, which are generally smaller, but this would create the overhead of generating versions at query time. This paper studies the trade-offs involved in these approaches, in the context of archiving dynamic RDF datasets over the Web. Our main message is that a hybrid policy would work better than any of the above approaches, and describe our proposed methodology for establishing a cost model that would allow determining when each of the two standard methods (version-based or delta-based storage) should be used in the context of a hybrid policy.

1 Introduction

DBpedia, Freebase, and YAGO are, among many others, examples of large data repositories that are available to a wide spectrum of users through the Web. These data repositories store information about various entities, such as persons, movies, organizations, cities and countries, as well as their relationships. Typically, data in such datasets are represented using the RDF model [6], in which information is stored in triples of the form (*subject*, *predicate*, *object*), meaning that *subject* is related to *object* via *predicate*. By exploiting the advances of methods that automatically extract information from Web sites [16], such datasets became extremely large and grow continuously. For example, the Billion Triples Challenge dataset of 2012[1] contains about 1.44B triples, while DBPedia v3.9, Freebase and Yago alone feature 25M, 40M and 10M entities, and 2.46B, 1.9B and 120M triples, respectively [12].

Dynamicity is an indispensable part of the current Web, because datasets are constantly evolving for a number of reasons, such as the inclusion of new

[1] http://km.aifb.kit.edu/projects/btc-2012/

experimental evidence or observations, or the correction of erroneous conceptualizations [14]. The evolution of datasets poses several research problems, which are related to the identification, computation, storage and management of the evolving versions. In this paper, we are focusing on the *archiving problem*, i.e., the problem of efficiently storing the evolving versions of a dataset, with emphasis on datasets stored on the Web.

Datasets on the Web are interlinked; this is promoted by the recent hype of the Linked Open Data (LOD) cloud[2], which encourages the open publication of interrelated RDF datasets to encourage reusability and allow the exploitation of the added value generated by these links. Currently, the LOD cloud contains more that 60B triples and more than 500 million links between datasets.

The interlinking of evolving datasets causes problems in the Web context. The open and chaotic nature of the Web makes impossible to keep track of who uses (i.e., links to) a given dataset, or what are the effects of a given change to interrelated datasets; this is in contrast to closed settings, where every change in a dataset is automatically propagated to all related parties. As a result, access to previous versions should be allowed to guarantee that all related applications will be able to seamlessly continue operations and upgrade to the new version at their own pace, if at all. In addition, even in a non-interlinked setting, experiments or other data may refer to a particular state of the dataset and may be non-understandable when viewed under the new version. Finally, certain applications may require access to previous versions of the dataset to support *historical* or *cross-version* queries, e.g., to identify past states of the dataset, to understand the evolution/curation process, or to detect the source of errors in the current modeling.

Supporting the functionality of accessing and querying past versions of an evolving dataset is the main challenge behind archiving systems. The obvious solution to the problem is to store all versions, but this can quickly become infeasible, especially in settings where the dataset is too large and/or changes are too frequent, as is the case in the Web of data [12]. To avoid this, several works (e.g., [9, 5, 17]) have proposed the use of *deltas*, which essentially allow the on-the-fly generation of versions, given any version and the deltas that lead to it. Even though this approach generally reduces space requirements, it causes an overhead at query time, because versions need to be reconstructed before being queried.

Given that RDF is the de-facto standard language for publishing and exchanging structured information on the Web, the main objective of this paper is to present our work towards designing a number of archiving policies for evolving RDF datasets. To do this, we study the trade-offs between the above two, basic archiving policies. In fact, we are proposing to use a cost model that employs the time and space overheads that are imposed by them in order to support *hybrid policies*, where some of the versions and some of the deltas are stored. Such policies could enjoy the best of both worlds by identifying which versions and which deltas should be stored to achieve optimal performance in terms of

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

both space requirements and query efficiency. To our knowledge this is the first work studying this problem for RDF datasets on the Web.

The rest of this paper is organized as follows. Section 2 presents a classification of the queries we are interested in, and Section 3 defines deltas between versions of datasets. Section 4 describes the basic archiving policies. Section 5 introduces the hybrid archiving policies, while Section 6 discusses a number of extensions. Section 7 focuses on different implementation strategies by considering different aspects of the archiving problem. Section 8 presents related work, and finally, Section 9 concludes the paper.

2 Query Types

We consider 3 types of queries, namely *modern*, *historical* and *cross-version*. These differ on the version(s) that they require access to, in order to be answered. *Modern queries* are queries referring to the current version of a dataset. For instance, in a social network scenario, one may want to pose a query about the (current) average number of friends of a certain group of subscribers. *Historical queries* are queries posed on a (single) past version of the dataset. In the example above, one could be interested to know the average number of friends for the same group at a given past version. Finally, *cross-version queries* are posed on several versions of the dataset, thereby retrieving information residing in multiple versions. For example, one may be interested in assessing how the number of friends of a certain person evolved over the different versions of the dataset. Such queries are important for performing different types of analytics across versions or for monitoring and understanding the evolution process.

From a different perspective, and motivated by [4] that focuses on storing and querying large graphs, we distinguish between *global* and *targeted* queries. Abstractly, to answer a global query, we need the entire version of a dataset. In contrast, targeted queries require accessing only parts of the version. For example, the average number of friends, at a given version, of all subscribers of a social network is a global query, while the average number of friends for a specific group of subscribers is a targeted query.

Finally, we distinguish queries as *version-centered* and *delta-centered* queries. Version-centered queries require the versions themselves for the computation of results. On the other hand, delta-centered queries manage to work only with deltas. For instance, retrieving the difference in the number of friends of a group of subscribers in a social network between two versions, represents a delta-centered query. We generalize the notion of delta-centered queries, so as to include queries that need for evaluation, along with deltas, stored versions, while dictating no versions reconstructions.

3 Changes and Deltas

The option of storing deltas, rather than versions, was proposed as an alternative archiving strategy to reduce storage space requirements. To implement this, one

has to develop a *language of changes*, which is a formal specification of the changes that the system understands and detects, as well as *change detection and application algorithms* that allow the computation of deltas, and their subsequent on-demand application upon versions [9].

One way to do this, is to use *low-level deltas*, which amount to identifying the RDF triples added and deleted to get from one version to the other (e.g., [15, 17]). Many approaches however, employ more complex languages, resulting to *high-level deltas*, which produce more concise deltas that are closer to human perception and understanding, and capture changes from a semantical (rather than syntactical) perspective (e.g., [9, 8]).

Languages are coupled with appropriate change detection algorithms, which, given two versions V_i, V_j, produce a delta δ that describes the changes that lead from V_i to V_j; obviously, said delta is expressed using the language of changes that the corresponding tool understands and detects. Moreover, languages are coupled with change application algorithms, which take a version, say V_i, and a delta, δ, in the input and return the result of applying δ on V_i. Note that both algorithms should abide by the semantics of the language of changes that they implement. In addition, the change detection and application algorithms should be *compatible*, in the sense that the output of change detection between V_i and V_j, when applied upon V_i, should return V_j (this is called *consistent detection and application semantics* in [9]).

Also it often makes sense to store V_j, rather than V_i. This gives more flexibility to the archiving system, as it allows the storage of intermediate versions (or the current one), rather than necessarily the first one only. To support this feature, the change detection algorithm should be able to compute the *inverse delta*, denoted by δ^{-1}, either from the original input (V_i, V_j) or from the delta itself; moreover, the change application algorithm should be able to use it to correctly produce V_i, when V_j and δ^{-1} are available. This property is called *reversibility* [9].

In this work, we are not restricting ourselves to any particular language and change detection/application algorithm. We only have two requirements: first, that such algorithms exist and they are *compatible*, and, second, that such algorithms satisfy the *reversibility* property.

4 Basic Archiving Policies

In this section, we elaborate on the pros and cons of the two main archiving approaches, namely *full materialization* and *materialization using deltas* (a high level representation is depicted in Fig. 1). In addition, we present a third basic approach that materializes exhaustively both versions and deltas.

Full Materialization. The most obvious solution to evaluate queries of any type involves maintaining all the different versions of a dataset. Under this archiving policy, every time a new version of a dataset is available, such version is stored in the archive. The advantages of this approach is that the archiving task comes

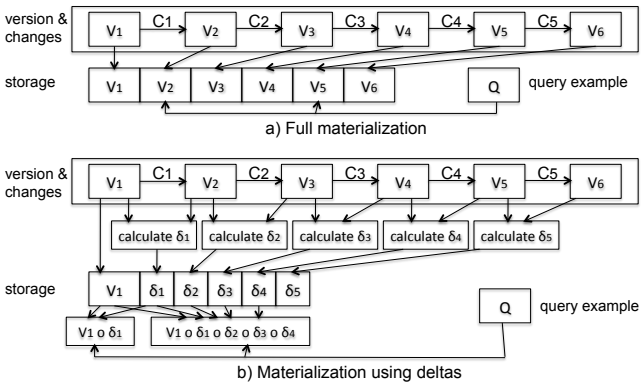

Fig. 1: Visualizing the a) *full materialization* and b) *materialization using deltas* archiving policies.

with no processing cost, as the version is stored as-is. Moreover, since all versions are materialized, the full materialization policy allows efficient query processing (in general, for all types of queries). The main drawback of this policy is that the space overhead may become enormous, especially in cases where the stored versions are large and/or too many versions need to be stored (e.g., due to the fact that the dataset evolves too often). Under this policy, even the slightest change in the dataset between two versions would force the full replication and storage of the new version in the archive, resulting in large space requirements.

Materialization Using Deltas. A feasible alternative to the full materialization policy is the storage of deltas. Under this policy, only one of the versions of the dataset needs to be fully materialized (e.g., the first or the last one); to allow access to the other versions, deltas that describe the evolution of the versions from the materialized version should be computed and stored. Clearly, in this case, space requirements are small. However, the evaluation of queries would require the on-the-fly reconstruction of one or more of the non-materialized versions, which introduces an overhead at query time. Furthermore, storing the deltas introduces an overhead at storage time (to compute the deltas).

Materialization Using Versions and Deltas. From another extreme, the materialization using versions and deltas policy maintains both all different versions of a dataset, as well as all deltas between any two consecutive versions. That is, for each new published version of the dataset, we store in the archive the version along with its delta from the previous version. The main advantage of this approach is that allows for efficient query processing, even for delta-centered queries. However, computing deltas, at storage time, introduces an overhead. The obvious drawback of the policy is its vast space requirements.

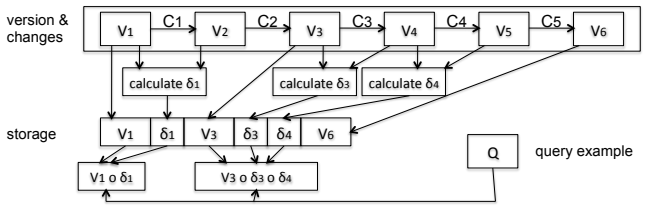

Fig. 2: Visualizing the *hybrid* archiving policy.

5 Hybrid Archiving Policies

The main motivation behind our paper is that a *hybrid solution*, where some of the versions are stored under the *full materialization* policy, while others are stored using the *materialization using deltas* policy, would work best. Thus, a hybrid policy would materialize only *some* of the versions (which could include the first, last and/or intermediate versions), as well as deltas for reconstructing the non-materialized versions (Fig. 2). The objective of a hybrid policy is to strike a balance between query performance and storage space; applied correctly, it would allow us to enjoy the best of both worlds, by leading to modest space requirements, while not introducing a large overhead at storage and query time, according to the specific needs and peculiarities of the corresponding application domain.

The challenge in this direction is to determine what to materialize at each step, i.e., a version or a delta, by employing an appropriate cost model. Each of the two policies would introduce a specific time and space overhead, which need to be quantified and compared via the cost model. To make this more specific, let's assume that an existing version V_{i-1} evolves into V_i, and we need to decide whether to store V_i itself, or the appropriate delta δ_i.

5.1 Storing a version

At storage time, the policy of full materialization (fm) causes a time overhead to store V_i, namely $t^{fm}_{store}(V_i) = |V_i| \cdot g$, where $|V_i|$ is the size of V_i (#triples), and g is the cost of storing a triple in the disk[3]. At query time, given a specific query q_j, there is a time overhead for executing q_j over V_i, that is, $t^{fm}_{query}(V_i, q_j) = c(V_i, q_j)$, where c reflects the complexity of executing q_j over V_i and depends on $|V_i|$ and the size of the result $res(q_j)$ of q_j. Assuming that p_{V_i} is an estimation on the number of the set of upcoming queries Q that refer to V_i, the time overhead for executing all Q queries is expressed as:

$$t^{fm}_{query}(V_i, Q) = p_{V_i} \cdot avg_Q(t^{fm}_{query}(V_i, q_j))$$

[3] In general, there may be other factors affecting the storage time (e.g., caching effects, storage method, disk technology, etc.), which may cause small deviations from this time estimate; however this paper aims to provide an approximation of the various costs, and minor deviations are acceptable.

Then, the overall time overhead when storing a version V_i is defined, taking into consideration the estimated upcoming queries for the version, using an aggregation function $\mathcal{F}$:

$$t^{fm}(V_i, Q) = \mathcal{F}(t^{fm}_{store}(V_i), t^{fm}_{query}(V_i, Q))$$

The corresponding space overhead of storing V_i is denoted by $s^{fm}(V_i)$ and is equal to $|V_i|$.

5.2 Storing a delta

When storing the delta δ_i instead of V_i, in order to compute it, we need both versions V_{i-1} and V_i. Next, we distinguish between two cases, namely, V_{i-1} was stored during the previous step, or not.

V_{i-1} is stored

Assuming the scenario in which V_{i-1} is stored, there is a time overhead for computing and storing δ_i, at storage time, and for reconstructing V_i and executing a set of queries Q over it, at query time.

Specifically, we formulate the overhead to compute δ_i as $t^{\delta,V_{i-1}}_{compute}(V_i, V_{i-1}, \delta_i) = d_c(V_i, V_{i-1}, \delta_i)$, where d_c reflects a function expressing the difficulty of computing δ_i with respect to the size and form of V_{i-1}, V_i and δ_i. As when storing a version, the overhead to store δ_i is $t^{\delta,V_{i-1}}_{store}(\delta_i) = |\delta_i| \cdot g$, where $|\delta_i|$ is the size of δ_i (#triples).

At query time, the overhead for reconstructing the version V_i is given by:

$$t^{\delta,V_{i-1}}_{reconstruct}(V_{i-1}, \delta_i) = p_{V_i} \cdot d_a(V_{i-1}, \delta_i)$$

where d_a defines the difficulty of applying δ_i to V_{i-1} taking into account the size and form of V_{i-1} and δ_i. Similarly to t^{fm}_{query}, the overhead of executing a set of p_{V_i} queries Q over V_i is:

$$t^{\delta,V_{i-1}}_{query}(V_i, Q) = p_{V_i} \cdot avg_Q(t^{\delta,V_{i-1}}_{query}(V_i, q_j))$$

where $t^{\delta,V_{i-1}}_{query}(V_i, q_j)$ is the specific overhead for computing the results of q_j over V_i.

The overall time overhead in this policy is defined, using an aggregation function $\mathcal{G}$, as follows:

$$t^{\delta,V_{i-1}}(V_i, Q) = \mathcal{G}(t^{\delta,V_{i-1}}_{compute}(V_i, V_{i-1}, \delta_i), t^{\delta,V_{i-1}}_{store}(\delta_i), t^{\delta,V_{i-1}}_{reconstruct}(V_{i-1}, \delta_i), t^{\delta,V_{i-1}}_{query}(V_i, Q))$$

Finally, the space overhead for storing the delta, denoted by $s^{\delta,V_{i-1}}(\delta_i)$, equals to $|\delta_i|$.

V_{i-1} is not stored

When V_{i-1} is not stored, it must be somehow created to allow computing the delta; towards this aim, we propose two alternative policies. The former reconstructs sequentially all previous versions, starting from the latest stored one, in order to manage to reconstruct V_{i-1}. The latter just maintains temporarily the current version of a dataset (until the next "new version" appears), independently of the decision regarding the storage of the version or its delta.

Reconstruct V_{i-1}
Assume that V_j is the latest stored version. Then, we should first reconstruct V_{j+1}. Using V_{j+1}, we reconstruct V_{j+2}, and so forth. At the final step, we use V_{i-2} to reconstruct V_{i-1}. This way, the time overhead to reconstruct V_{i-1} is:

$$t^{\delta,\neg V_{i-1}}_{prev\_reconstruct}(V_{j+1}, V_{i-1}) = (i-1-j) \cdot avg_{j\ to\ (i-1)}(d_a(V_x, \delta_{x+1}))$$

while the overhead to store V_{j+1}, V_{j+2}, ..., V_{i-1} is:

$$t^{fm,\neg V_{i-1}}_{prev\_store}(V_{j+1}, V_{i-1}) = t^{fm}_{store}(V_{j+1}) + t^{fm}_{store}(V_{j+2}) + \ldots + t^{fm}_{store}(V_{i-1})$$

Having reconstructed V_{i-1}, the additional time overhead is defined as in the case in which V_{i-1} is stored. In overall, given an aggregation function $\mathcal{H}$, the time overhead of this policy is:

$$t^{\delta,\neg V_{i-1}}(V_i, Q) = \mathcal{H}(t^{\delta,\neg V_{i-1}}_{prev\_reconstruct}(V_{j+1}, V_{i-1}), t^{fm,\neg V_{i-1}}_{prev\_store}(V_{j+1}, V_{i-1}), t^{\delta,V_{i-1}}(V_i, Q))$$

Finally, the space overhead $s^{\delta,\neg V_{i-1}}(\delta_i)$ in this policy is $max\{|V_{j+1}|+|V_{j+2}|+\ldots+|V_{i-1}|\}+|\delta_i|$.

Maintain temporarily V_{i-1}
The *reconstruct* V_{i-1} policy exhaustively reconstructs, from the latest stored version, all intermediate versions in order to catch V_{i-1}. Differently, we propose maintaining temporarily the current version. Using this heuristic, when the new version arrives, i.e., V_i, we can directly compare it with the previous one, i.e., V_{i-1}. Then, we drop the version that has been assigned as current, i.e., V_{i-1}, and maintain the new version, i.e., V_i. Abstractly speaking, the *maintain temporarily* V_{i-1} policy reduces both the time and space overheads, compared to the *reconstruct* V_{i-1} policy, since it only requires the maintenance of one additional version.

Following this approach, the time and space overheads are defined as follows:

$$t^{\delta,+V_{i-1}}(V_i, Q) = t^{\delta,V_{i-1}}(V_i, Q) + t^{fm}_{store}(V_{i-1})$$
$$s^{\delta,+V_{i-1}}(\delta_i) = s^{\delta,V_{i-1}}(\delta_i) + |V_{i-1}|$$

5.3 On selecting a hybrid archiving policy

To generalize, the decision on which policy is optimal, when a new version arrives, depends on the relative importance given to the time and space overheads for the application at hand. For example, a time-critical application might not care too much about the space overheads, whereas other applications could be more balanced in their requirements.

A simple way to model this, is by using the weighted summation of the time and space overheads of our hybrid policies, and select the best choice per case. In particular, assume a function $\mathcal{I}$ defined as:

$$\mathcal{I}^{\alpha} = w_t \cdot t^{\alpha} + w_s \cdot s^{\alpha}$$

where w_t, w_s are weights in [0,1], t^{α} corresponds to $t^{fm}(V_i, Q)$, $t^{\delta,V_{i-1}}(V_i, Q)$, $t^{\delta,\neg V_{i-1}}(V_i, Q)$ or $t^{\delta,+V_{i-1}}(V_i, Q)$, and s^{α} corresponds to $s^{fm}(V_i)$, $s^{\delta,V_{i-1}}(\delta_i)$,

$s^{\delta,\neg V_{i-1}}(\delta_i)$ or $s^{\delta,+V_{i-1}}(\delta_i)$ for the *storing a version*, *storing a* δ, *storing a* δ *& reconstruct* V_{i-1} or *storing a* δ *& maintain* V_{i-1} policy, respectively. Typically, the policy with the minimum cost, i.e., the minimum value for $\mathcal{I}^{\alpha}$, is the one that will be used. Although more sophisticated functions can be designed, this function is simple and intuitive, and allows directly comparing the policies costs.

6 Extending Hybrid Archiving Policies

Clearly, the process of version reconstruction appears to be costly. In this section, we study different ways for, whenever possible, i.e., for specific query types, either avoiding reconstructions, or reconstructing only parts of versions. This way, we manage to reduce the time overhead when processing queries.

6.1 Use only deltas

Given a delta-centered query, i.e., a query that can be evaluated directly on deltas without accessing any versions, when following the *use only deltas* policy, no version reconstruction is required. As an example, consider a query that targets at retrieving the difference in the number of friends of a specific group of subscribers in a social network between two versions. For computing the answer of this query, we count the number of triples in the delta, added or deleted, that involve at least one subscriber in the given group and reflect friendships between subscribers.

This way for a delta-centered query, the corresponding overall time overheads for the *storing a* δ, *storing a* δ *& reconstruct* V_{i-1} and *storing a* δ *& maintain* V_{i-1} policies, reduce to:

$$t^{\delta,V_{i-1}}_{only\ \delta}(V_i,Q) = \mathcal{G}(t^{\delta,V_{i-1}}_{compute}(V_i,V_{i-1},\delta_i), t^{\delta,V_{i-1}}_{store}(\delta_i), t^{\delta,V_{i-1}}_{query}(\delta_i,Q))$$

$$t^{\delta,\neg V_{i-1}}_{only\ \delta}(V_i,Q) = \mathcal{H}(t^{\delta,\neg V_{i-1}}_{prev\_reconstruct}(V_{j+1},V_{i-1}), t^{fm,\neg V_{i-1}}_{prev\_store}(V_{j+1},V_{i-1}), t^{\delta,V_{i-1}}_{only\ \delta}(V_i,Q))$$

and

$$t^{\delta,+V_{i-1}}_{only\ \delta}(V_i,Q) = t^{\delta,V_{i-1}}_{only\ \delta}(V_i,Q) + t^{fm}_{store}(V_{i-1})$$

where $t^{\delta,V_{i-1}}_{query}(V_i,\delta_i)$ defines the cost of executing p_{V_i} queries over δ_i.

6.2 Use deltas and stored versions

Alternatively, one can assume a scenario in which both stored versions and deltas can be used for answering a query, instead of reconstructing versions referring to this query. For example, for computing the average number of friends of a specific user in a set of versions, we count first the number of friends in the stored version, and while accessing the deltas that correspond to the other versions of the query, we calculate the average requested value. As motivated by the example, this policy is applicable only to delta-centered queries.

The overall time overheads, denoted as $t^{\delta,V_{i-1}}_{\delta,V_i}(V_i,Q)$, $t^{\delta,\neg V_{i-1}}_{\delta,V_i}(V_i,Q)$ and $t^{\delta,+V_{i-1}}_{\delta,V_i}(V_i,Q)$ are defined as in Section 6.1. The only difference is on the cost $t^{\delta,V_{i-1}}_{query}(\delta_i,Q)$, which is replaced by $t^{\delta,V_{i-1}}_{query}(V_i,(\delta_i,\delta_{i-1},\ldots,\delta_x),Q)$, since one query in Q is evaluated over the version V_i, while the rest over the deltas δ_i, δ_{i-1}, …, δ_x.

6.3 Partial version reconstruction

However, there are cases in which version reconstruction cannot be avoided. To improve the efficiency of such scenarios, we propose the partial version reconstruction policy. Specifically, when queries access only parts of a dataset, like the targeted queries, we may construct only those parts required for the query execution.

Let's assume that an existing version V_{i-1} evolves into V_i. Let also V_i' be the needed part of V_i for query answering, and δ_i' be the part of δ_i that when applied to V_{i-1} constructs V_i'. Then, the overall time overheads for the *storing a δ*, *storing a δ & reconstruct V_{i-1}* and *storing a δ & maintain V_{i-1}* policies are defined as:

$$t^{\delta,V_{i-1}}_{partial}(V_i,Q)=\mathcal{G}(t^{\delta,V_{i-1}}_{compute}(V_i,V_{i-1},\delta_i),t^{\delta,V_{i-1}}_{store}(\delta_i),t^{\delta,V_{i-1}}_{reconstruct}(V_{i-1},\delta_i'),t^{\delta,V_{i-1}}_{query}(V_i',Q))$$

$$t^{\delta,\neg V_{i-1}}_{partial}(V_i,Q)=\mathcal{H}(t^{\delta,\neg V_{i-1}}_{prev\_reconstruct}(V_{j+1},V_{i-1}),t^{fm,\neg V_{i-1}}_{prev\_store}(V_{j+1},V_{i-1}),t^{\delta,V_{i-1}}_{partial}(V_i,Q))$$

$$t^{\delta,+V_{i-1}}_{partial}(V_i,Q)=t^{\delta,V_{i-1}}_{partial}(V_i,Q)+t^{fm}_{store}(V_{i-1})$$

where $t^{\delta,V_{i-1}}_{reconstruct}(V_{i-1},\delta_i')$ includes as well the cost for computing δ_i'.

7 Implementation Strategies

Above, we studied a number of policies for storing evolving datasets, each of which has different pros and cons and is suitable for a different usage scenario. In particular, full materialization policies are good when versions are not too large or not too many, or when the storage space required for storing the evolving datasets is not an issue compared to the access time (which is optimal in this policy). On the contrary, delta-based strategies are on the other end of the spectrum, optimizing storage space consumption but causing overheads at query time for most types of queries. The intermediate policies considered in this paper (hybrid policies, partial reconstruction of versions, temporary storage of the latest version) aim to strike a balance between these extremes. Table 1 presents a summary of the various policies, while next we focus on different implementation strategies by considering different aspects of the problem of what to store.

To decide which policy to follow, one needs to somehow "predict" the future behavior of the dataset, e.g., the number, type and difficulty of queries that will be posed upon the new or already existing versions. Even though an experienced curator may be able to do such a prediction with a reasonable accuracy (e.g., based on past queries or on the importance of the new version), predictions may

Table 1: Archiving policies.

Archiving policy		Short Description	Applicability
Basic policies (store all versions, one version and deltas between versions, or all versions and all deltas)	*full materialization*	maintain all the different versions of a dataset	all query types
	materialization using deltas	maintain only one version of a dataset and deltas describing the evolution from the stored version	all query types
	materialization using versions and deltas	maintain all versions of a dataset and all deltas between consecutive versions	all query types
Hybrid policies (given that an existing V_{i-1} evolves into V_i, decide whether to store V_i or δ_i)	*store a version*	store V_i	all query types
	store a delta	store δ_i (V_{i-1} is stored)	all query types
	store a delta & reconstruct V_{i-1}	store δ_i (V_{i-1} is not stored) V_{i-1} is reconstructed via a set of reconstructions from the latest stored version	all query types
	store a delta & maintain V_{i-1}	store δ_i (V_{i-1} is not stored) maintain V_{i-1}, until V_i arrives	all query types
Extensions (special cases for avoiding (full) reconstructions)	*use only deltas*	query evaluation on deltas, i.e., no version reconstruction is required	delta-centered queries
	use deltas & stored versions	query evaluation on deltas and stored versions, i.e., no version reconstruction is required	delta-centered queries
	partial version reconstruction	reconstruct only the parts needed for query evaluation	targeted queries

need to be revised. Based on this idea, we discriminate three different ways to take decisions regarding the policy to follow for the storage of each new version.

In general, when deciding if a new version will be materialized or not at the time of its publication, we care for achieving a feasible and efficient solution at local level. This notion of *locality* resembles a greedy approach that targets at decisions taking into account only the recent history of materializations, for example, by comparing the current version with only the previous one (possibly, by using materialized deltas). In other words, the locality implementation strategy takes a decision on the storage policy based on the current data/curator knowledge, and does not revise or reconsider such decision later.

Following a different implementation strategy, we may judge about what to materialize periodically. In particular, *periodicity* uses periods defined as sets, of specific size, of consecutive versions or versions published within specific time intervals. Then, decisions about which versions and deltas to store, within a period, are taken with respect to the versions published in the period. Periodicity succeeds in extending the local behavior of the previous approach. However, it

wrongly assumes that updates are uniformly distributed over periods, leading sometimes to maintaining versions with very few changes.

Alternatively, one may be interested in decisions that aim to store those versions that offer global potentials to the storage and query processing model. The *globality* implementation strategy works towards this direction. It resembles a more exhaustive approach targeting at storing decisions based on comparisons between the current version and all the previously materialized ones, i.e., taking into account the whole history of materializations. Note that globality could also lead to the re-evaluation of the decisions to keep/drop some of the previous versions, as new versions appear, which is not the case for the locality and periodicity strategies that do not reconsider past decisions. This way, globality typically comes at the cost of a huge number of comparisons between versions.

Clearly, a strategy that combines characteristics of the above approaches, in order to achieve efficient and effective storing schemes, is a challenge worth studying. In our current work, for bounding the number of the comparisons executed when the globality strategy, i.e., the strategy with the best quality, is employed, we focus on tuning the number of versions to be examined with respect to the estimated frequencies and types of the upcoming queries, and the frequencies and amounts of changes between consecutive versions.

8 Related Work

In the context of XML data, [7] presents a delta-based method for managing a sequence of versions of XML documents. Differences, representing deltas, between any two consecutive versions are computed, and only one version of the document is materialized. To achieve efficiency, [1] merges all versions of the XML data into one hierarchy; an element, associated with a timestamp, appears in multiple versions and is stored only once. Clearly, our work does not target at hierarchical models for archiving, but on graph-organized data on the Web.

[5] presents, in the context of social graphs, a solution for reconstructing only the part of a version that is required to evaluate a historical query, instead of the whole version, similar to our partial version reconstruction policy. However, the main focus of this work is different, since it considers time with respect to graph evolution. [3] considers as well the addition of time. Specifically, it enhances RDF triples with temporal information, thus yielding temporal RDF graphs, and presents semantics for these graphs, including a way for incorporating temporal aspects into standard RDF graphs by adding temporal labels. Our approach is different in that it does not consider temporal-aware query processing. [11] proposes a graph model for capturing relationships between evolving datasets and changes applied on them, where both versions and deltas are maintained. From a different perspective, [13] supports versioning by proposing the use of an index for all versions.

Recently, several commercial approaches that support archiving come up. To do so, for example, Dropbox uses deltas between different versions [2], Googledrive[4]

[4] `drive.google.com/`

stores the entire versions, while, to our knowledge, none of them employs a hybrid approach. In general, it should be noted that adding timestamps to the triples in the datasets could partly solve the archiving problem, as versions could be generated on-the-fly using temporal information [1]. However, encoding temporal information in RDF triples often resorts to cumbersome approaches, such as the use of reification [3], which create overheads during querying. Furthermore, timestamps are generally absent in the context of Web data [10], which makes such a solution infeasible in practice. Studying this alternative is beyond the scope of this paper.

9 Conclusions

In this paper, we discuss the problem of archiving multiple versions of an evolving dataset. This problem is of growing importance and applies in many areas where data are dynamic and users need to perform queries not only on a single (possibly the current) dataset version, but also on a set of previous versions. The problem becomes more difficult in the context of the Web, where large, interconnected datasets of a dynamic nature appear.

Towards addressing the archiving requirements of data on the Web, we focus on designing different policies that, taking into consideration time and space overheads, aim to determine the versions of a dataset that should be materialized, as opposed to those that should be created on-the-fly at query time using deltas. We consider various parameters affecting such a decision, related to the type, frequency and complexity of queries, and size and frequency of changes.

Clearly, there are many directions for future work. As a first step, our plans include the evaluation of the cost model to experimentally verify whether the actual overheads appearing in practice are consistent with the theoretically expected ones. Moreover, it is our purpose to study methods for appropriately selecting the input parameters of our cost functions, algorithms for query evaluation and implementations in specific contexts. We peer a two-phase algorithm; in the first phase, the algorithm locates or reconstructs the version(s) required, in the second phase, for the query execution.

To further increase the efficiency of archiving, we envision building indexes on the materialized versions and deltas. In general, indexing could improve performance significantly, by enabling faster access of the data required for versions reconstructions and query executions. Specifically, we examine specific index structures suitable for specific query types and policies. For example, when assuming a *targeted query*, one option is to use an index for efficiently identifying only the part of a version or delta that the query targets at. Such an index appears to be of high importance when using, for example, the *partial version reconstruction* policy.

Interestingly, specific parts of versions or deltas that are not stored in the archive, but have been reconstructed to satisfy various query requirements, can be indexed in order to be re-used either as they are, or after being combined with each other. Our goal in this direction is to define a policy that exploits

previous reconstructions, as well as their combinations, to minimize the number of reconstructions needed for new queries. To do this, properties such as *applicability*, expressing whether a part of a version can be used for evaluating a specific query, and *combining ability*, expressing whether two parts of a version can be combined to produce a new one, need to be appropriately defined.

Acknowledgments

This work was partially supported by the European project DIACHRON (IP, FP7-ICT-2011.4.3, #601043).

References

1. P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *TODS*, 29:2–42, 2004.
2. I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: understanding personal cloud storage services. In *Internet Measurement Conference*, 2012.
3. C. Gutierrez, C. Hurtado, and A. Vaisman. Temporal RDF. In *ESWC*, 2005.
4. U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, 2011.
5. G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. In *WOSS*, 2012.
6. F. Manola, E. Miller, and B. McBride. RDF primer. www.w3.org/TR/rdf-primer, 2004.
7. A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *VLDB*, 2001.
8. N. Noy and M. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. In *AAAI*, 2002.
9. V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in RDF(S) KBs. *TODS*, 38(1), 2013.
10. A. Rula, M. Palmonari, A. Harth, S. Stadtmüller, and A. Maurino. On the diversity and availability of temporal information in Linked Open Data. In *ISWC*, 2012.
11. Y. Stavrakas and G. Papastefanatos. Supporting complex changes in evolving interrelated web databanks. In *OTM Conferences (1)*, 2010.
12. K. Stefanidis, V. Efthymiou, M. Herchel, and V. Christophides. Entity resolution in the Web of data. In *WWW*, 2014.
13. Y. Tzitzikas, Y. Theoharis, and D. Andreou. On storage policies for semantic Web repositories that support versioning. In *ESWC*, 2008.
14. J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. Towards dataset dynamics: Change frequency of Linked Open Data sources. In *LDOW*, 2010.
15. M. Volkel, W. Winkler, Y. Sure, S. Kruk, and M. Synak. SemVersion: A versioning system for RDF and ontologies. In *ESWC*, 2005.
16. G. Weikum and M. Theobald. From information to knowledge: harvesting entities and relationships from Web sources. In *PODS*, 2010.
17. D. Zeginis, Y. Tzitzikas, and V. Christophides. On computing deltas of RDF(S) knowledge bases. *TWEB*, 2011.