EvoGen: a Generator for Synthetic Versioned RDF

Marios Meimaris
[1]University of Thessaly,
[2]ATHENA Research Center,
Greece
m.meimaris@imis.athena-innovation.gr

ABSTRACT

Synthetic data are widely used for evaluation, testing, and experimentation. However, there is a lack of systems, tools and datasets that can be used for benchmarking in the context of evolution. In the case of RDF, generation of synthetic data that change through time must take into account evolving paradigms and characteristics that make sense, rather than arbitrary insertions and deletions of triples. In this paper, we discuss requirements for generation of synthetic evolving datasets by abstracting several characteristics of the process, and present EvoGen, a tool for evolving dataset generation that is based on the widely used Lehigh University Benchmark (LUBM) generator.

Categories and Subject Descriptors

H.2.8 [**Information Systems Applications**]: Database Management—*Database Applications*

Keywords

RDF, Data Management, Benchmarks, Synthetic Data

1. INTRODUCTION

The Resource Description Framework[1] (RDF) is a W3C recommendation for modelling and publishing data in Linked Data and Semantic Web contexts. Due to the wide adoption of RDF, as well as the dynamicity of data published in the Data Web, the need for handling evolution in such datasets becomes increasingly relevant. The importance of evolution management in the context of the Data Web has been stressed out in the literature as a means of addressing issues such as provenance tracking, longitudinal querying, semantic change detection and representation, dealing with broken URIs and so on [6, 1]. Evolution in RDF data, in its simplest form, is the act of inserting and deleting triples over time. However, in real settings, evolution takes place in different schemes and settings, depending on the context. For instance, versioning can take place on either the resource level, or the dataset level, and at the same time, changes can be detected and represented as simple (e.g. triple additions/deletions), or more complex (e.g. schema changes, groupings of triple additions that result in higher level changes). Moreover, other requirements that are often present in evolving RDF have to do with metadata, such as provenance and temporal annotations. There exist several tools for storage and archiving that come with varying functionalities, however, there is a lack of support when it comes to benchmarking them. Two main aspects must be considered towards this goal. First, the existence of real and synthetic datasets of varying size and complexity that represent several cases of evolution is an important element of such a benchmark. Then, the performance evaluation of these tools require the existence of appropriate query workloads and representative evolving operations. In this paper, we intend to address the former, i.e. generation of synthetic datasets in evolving contexts.

Contributions. The contributions of this paper can be summarized as follows:

- we discuss the requirements and characteristics of the process of creating synthetic versioned RDF data,
- we describe *EvoGen*, a prototype implementation for configurable synthetic dataset generation in evolving contexts that extends the Lehigh University Benchmark (LUBM) generator with support for the defined characteristics.

This paper is outlined as follows. Section 1 introduces the subject. Section 2 provides an overview of related work. Section 3 discusses conceptual aspects of different evolution paradigms in RDF. Section 4 describes the implemented system, and Section 5 concludes the paper.

2. RELATED WORK

The Lehigh University Benchmark (LUBM) [7] includes an implementation for RDF synthetic data generation. LUBM was originally aimed at providing datasets for benchmarking reasoners and systems with reasoning/inferencing capabilities for OWL and DAML ontologies. In fact, the generator creates both explicit and implicit relationships between the data. Nevertheless, it has been used extensively in the evaluation of SPARQL engines and RDF stores in general as well [19, 9, 10, 2, 4, 8, 15]. LUBM provides a set of 14 benchmark queries consisting of 1-6 conjunctive triple patterns, however, these have been appended to include queries with

[1]http://www.w3.org/RDF/

more complicated patterns in several other works, especially when it comes to evaluating SPARQL engines that need more complicated queries (e.g. in [9]). SP^{2}Bench [17] is a benchmark RDF dataset generator for evaluating SPARQL engines. It is targeted at query efficiency rather than reasoning and has been widely used in the literature [18, 8, 11]. There exist several other works in benchmarking RDF systems and tools, such as FedBench [16] and the Berlin SPARQL Benchmark [3], however not all of them provide synthetic data generation capabilities, even less so in the case of evolving data. An extensive comparison of RDF benchmarks is done in [5]. Fernandez et al. [6] provide metrics for benchmarking archiving systems in RDF and Linked Data settings.

Our approach aims at filling the gap of synthetic data generation in evolving contexts. For this purpose, we have chosen to extend LUBM in order to provide capabilities of versioned data, because it is a widely adopted and established benchmark, and can be used in storage, querying, as well as reasoning scenarios.

3. CHARACTERISTICS OF EVOLUTION

Generation processes for evolving datasets have to meet several functional, as well as non-functional requirements regarding the data creation. These include parameterization and configurability, as well as scalability and efficiency. The generation of evolving datasets must be abstracted from the generation of synthetic data in general, in order to identify, and consequently quantify, the inherent characteristics that are specific to evolving contexts.

In order to define and quantify input parameters, we go on to outline high-level, dataset-independent characteristics that will enable parameterization in the generation process. We consider D as a diachronic RDF dataset as defined in [13, 12], and a set of $n+1$ distinct temporal instantiations of D at time points $t_i \dots t_{i+n}$. A diachronic dataset D is a time-agnostic representation that is used to refer to D statically through time, without referencing its particular versions. In the scope of this work, we regard versions to be discrete snapshots of a dataset through time. Given these, we introduce the notions of *shift*, *monotonicity* and *strictness* to capture the type, volume, and structural aspects of changes that D undergoes between t_i and t_{i+n}. These are defined as follows.

1. Shift: the *shift* of an evolving dataset captures the modification of its size through different versions. In essence, it represents the direction towards which its size leans through the passing of time. The shift of a dataset is given with respect to a time period $[t_i, t_{i+n}]$, and depending on its value, it can lead to *incremental*, *decremental* or *unchanged* data. For example, when comparing two versions of an RDF dataset D, at times t_i and t_j, the shift of D will be incremental if the dataset size increased (i.e. more insertions than deletions), decremental if it decreased (i.e. more deletions than insertions) and unchanged if it remained the same (i.e. equal number of insertions and deletions). In order to quantify the shift h of D between t_i and t_{i+n}, we regard it as a function $h(D)|_{t_i}^{t_{i+n}}$ of low-level changes (triple insertions/deletions) in D between t_i and t_{i+n} with respect to t_i:

$$h(D)|_{t_i}^{t_{i+n}} = \frac{|D_{i+n}| - |D_i|}{|D_i|} \tag{1}$$

When defining h in a given time period, the changes are distributed across all versions that exist within that period.

2. Monotonicity: the *monotonicity* of a dataset captures whether or not a dataset with an incremental or decremental shift changes monotonically in a given time period $[t_i, t_j]$. A shift is monotonic when only additions or deletions of triples occur in all consecutive versions of a dataset between t_i and t_j. For example, the evolution of a dataset between t_i and t_j with incremental (decremental) shift is monotonic iff between t_i and t_j only triple additions (deletions) take place. More formally, an RDF dataset D is monotonically incremental if it has an incremental shift, and the following holds for any arbitrary time points t_k and t_l:

$$\nexists t_k, t_l : |D_l| > |D_k|, t_i \leq t_k < t_l \leq t_j \tag{2}$$

and decremental when

$$\nexists t_k, t_l : |D_l| < |D_k|, t_i \leq t_k < t_l \leq t_j \tag{3}$$

Monotonicity is an important characteristic that can be used for supporting use cases where datasets are only changing in one direction. In real-world scenarios, this is especially useful for creating datasets that are only increasing (e.g. log files, publication databases etc.).

3. Strictness: *strictness* is a boolean property that a dataset exhibits when it follows predictable schema patterns through time. Because of the schema looseness typically associated with RDF, an abstraction of schema in RDF datasets is needed in order to define strictness. We recall the notion of Characteristic Sets [14] as the needed abstraction. A characteristic set of a subject node s is essentially the collection of all distinct properties p that appear in triples with s as subject. Given an RDF dataset D, and a subject s, the Characteristic Set $S_c(s)$ of s is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\}$$

and the set of all S_c for a dataset D at time t_i is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\}$$

A dataset D is strict in a given time period $[t_i, t_j]$ if the set of all Characteristic Sets $S_c(D)$ remains the same within this time period:

$$\nexists t_k, t_l : S_c(D_{t_k}) \neq S_c(D_{t_l}), t_i \leq t_k < t_l \leq t_j \tag{4}$$

Whether the actual subject and object nodes change is not relevant unless it causes a change in $S_C(D)$, as we are essentially interested in *structural* changes, rather than instance-level changes. In essence, (4) holds when there are no inherent structural changes between versions of D within a given time period.

4. IMPLEMENTATION

For our purposes, we implemented *EvoGen*[2], a generator of RDF datasets with configurable parameters according to the characteristics discussed in Section 3. The system is an extension of the existing Lehigh University Benchmark (LUBM) generator, which is written in Java and is a pure write-only solution that does not contain any abstractions and models for RDF datasets. While the original LUBM provides support for generation of OWL as well as DAML files, our implementation is only aimed at RDF/XML files, mainly because, unlike the original LUBM generator, EvoGen is not meant to be a benchmark for reasoning systems. The high-level architecture of EvoGen can be seen in Figure 1.

In this first version of EvoGen, the requirement for configurable *strictness* has not been implemented and is left as future work. Thus, the implemented functionality only concerns changes on the instance level, with a strict structure. In fact, the strictness of the structure is inherited from the schema used by LUBM in its original implementation.

Along with LUBM's original system, which requires parameters concerning the size of the dataset (in number of universities), EvoGen's generation process requires the following parameters:

1. number of versions: an integer denoting the total number of distinct versions (dataset materializations at different time points). The number of versions needs to be larger than 1 in order for EvoGen to generate datasets, else the original LUBM generator is invoked.
2. shift: this is the value $h(D)|_{t_i}^{t_j}$ defined in equation (1) for a time period $[t_i, t_j]$, i.e. a percentage of changes between versions D_i and D_j with respect to the size of D_i. In this version of *EvoGen*, only strictly incremental and decremental shifts between consecutive versions have been implemented. This means that every version will be shifted with respect to its previous version, instead of generating an aggregated shift between the first and the last version, which would require distributing the required changes along a number of versions and maintaining the $h(D)$ value only between the first and the last version in the given period. Instead, this is left as future work.
3. monotonicity: A boolean denoting the existence of monotonicity in the shift, or lack thereof.

The above parameters instantiate the generation process with the use of two basic components, namely the *Version Management* component and the *Change Creation* component. These are responsible for translating input parameters dynamically and mapping them to appropriate structures, holding information on the types of entities to create, the size and quantity of the created entities, as well as the decisions on which existing entities to evolve. These sit on top of an extended LUBM generator, which in essence is the core LUBM generator modified to accommodate serializations of different versions in the file system. The functionality is exposed through a Java API that can be invoked by importing the system's jar file and accessing its methods directly.

The Change Creation component is responsible for creating the elements to be added, or determining the elements

[2]Source code is available at: https://github.com/mmeimaris/EvoGen

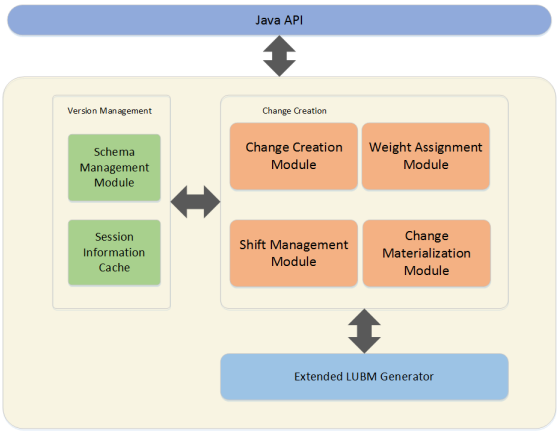

Figure 1: High-level architecture of *EvoGen*.

to be deleted, calculating the actual number of insertions/deletions for each class, based on dynamic weighting of the instances of each class with respect to the total dataset size, randomizing parts of the process with respect to actual created/removed instances and so on. This component is responsible for communicating with the *Extended LUBM Generator component*, which performs the actual serialization of dataset versions in the file system. The core LUBM implementation generates data based on a fixed schema, by iterating through each class and creating elements in each respective schema-imposed sub-structure. It performs some degree of randomization on URIs, mappings of elements to other elements (e.g. *Professor32* teaches at *University21*). In order to distribute the required changes so that the generation process outputs versions with the desired shift, we assign weights to each class-based sub-structure and dictate the cardinality of the instances of each class to the generator. This weighting is done in the *Weight Assignment Module*, which uses hard coded, normalized weights in the range of $0 \ldots 1$ for each class, based on ranges acquired by observing the output of the original, non-versioned LUBM datasets for varying sizes. By multiplying these weights with the desired shift value $h(D)|_{t_i}^{t_j}$, we can get an approximate number of instances that need to be created for each class.

The Version Management component holds and updates session information on each version during runtime, the schema of the dataset, the mapping of versions to descriptive metadata and files in the file system and so on.

5. PRELIMINARY EVALUATION

In order to preliminarily evaluate and validate the output of *EvoGen*, we perform a series of generation tasks for different combinations of numbers of universities and changes, and a fixed number of 10 versions, and we measure the achieved shift with respect to the required one. Specifically, we perform 10 runs of generations for three different values of h, namely $h(D) = 0.2$, $h(D) = 0.4$, and $h(D) = 0.6$ and we report the percentage difference between the mean of the achieved h and the required one. The results can be seen in Figure 2. With a small number of universities, the achieved shift differs significantly with respect to the required one, but as the number of universities, i.e. the dataset size, increases, the error decreases. Therefore, for a reasonably large number of dataset size, (e.g. > 5 universities), EvoGen performs as expected. Note, however,

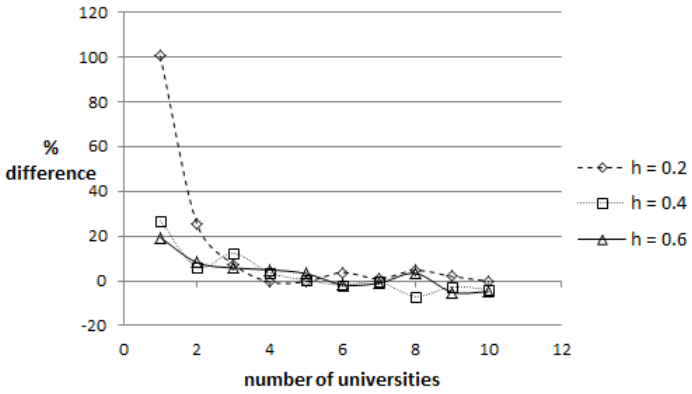

Figure 2: Average of achieved shift over 10 runs for 10 versions, for increasing number of universities.

that this preliminary evaluation does not take into account scalability and efficiency issues, which is left as future work.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discuss several characteristics of generating synthetic versioned RDF datasets, and we describe *EvoGen*, an implementation that addresses a subset of these characteristics. Furthermore, we perform a preliminary evaluation of the system in order to measure how well the desired shift is achieved.

As future work, we intend to fully address the discussed characteristics, enable the insertion and deletion of schema elements as well, and address issues of scalability and efficiency when creating very large datasets. Finally, we intend to design query workloads based on the created data, in order to further support benchmarking versioning and evolution management systems for RDF datasets.

Acknowledgements. This work is supported by the EU-funded ICT project "DIACHRON" (agreement no 601043).

7. REFERENCES

[1] S. Auer, T. Dalamagas, H. Parkinson, F. Bancilhon, G. Flouris, D. Sacharidis, P. Buneman, D. Kotzinos, Y. Stavrakas, V. Christophides, et al. Diachronic linked data: towards long-term preservation of structured interrelated information. In *Proceedings of the First International Workshop on Open Data*, pages 31–39. ACM, 2012.

[2] A. Bernstein, M. Stocker, and C. Kiefer. Sparql query optimization using selectivity estimation. In *Poster Proceedings of the 6th International Semantic Web Conference (ISWC)*, 2007.

[3] C. Bizer and A. Schultz. The berlin sparql benchmark, 2009.

[4] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

[5] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.

[6] J. D. Fernández, A. Polleres, and J. Umbrich. Towards efficient archiving of dynamic linked open data. In *Proceedings of the 1st DIACHRON workshop*, 2015.

[7] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[8] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 1–10. IEEE, 2010.

[9] E. G. Kalayci, T. E. Kalayci, and D. Birant. An ant colony optimisation approach for optimising sparql queries by reordering triple patterns. *Information Systems*, 50:51–68, 2015.

[10] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. Sparql query optimization on top of dhts. In *The Semantic Web–ISWC 2010*, pages 418–435. Springer, 2010.

[11] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM Transactions on Database Systems (TODS)*, 38(4):25, 2013.

[12] M. Meimaris, G. Papastefanatos, and C. Pateritsas. An archiving system for managing evolution in the data web. In *Proceedings of the 1st DIACHRON workshop*, 2015.

[13] M. Meimaris, G. Papastefanatos, S. Viglas, Y. Stavrakas, and C. Pateritsas. A query language for multi-version data web archives. *arXiv preprint arXiv:1504.01891*, 2015.

[14] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 984–994. IEEE, 2011.

[15] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 397–400. ACM, 2012.

[16] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *The Semantic Web–ISWC 2011*, pages 585–600. Springer, 2011.

[17] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp 2 bench: A sparql performance benchmark, icde. *Shanghai, China*, 2009.

[18] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM, 2010.

[19] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604. ACM, 2008.