RDFStats – An Extensible RDF Statistics Generator and Library

Andreas Langegger  
Institute for Application-oriented Knowledge Processing  
Johannes Kepler University Linz  
Altenberger Straße 69, 4040 Linz, Austria  
Email: al@jku.at

Wolfram Wöß  
Institute for Application-oriented Knowledge Processing  
Johannes Kepler University Linz  
Altenberger Straße 69, 4040 Linz, Austria  
Email: wolfram.woess@jku.at

Abstract—In this paper RDFStats is introduced, which is a generator for statistics of RDF sources like SPARQL endpoints and RDF documents. RDFStats does not only provide a statistics generator, but also a powerful API for persisting and accessing statistics including several estimation functions that also support SPARQL filter-like expressions.

For many Semantic Web applications like the Semantic Web Integrator and Query Engine (SemWIQ), which is currently developed at the University of Linz, detailed statistics about the contents of RDF data sources are very important. RDFStats has been primarily designed and implemented for the SemWIQ federator and optimizer, but it can also be used for other applications like linked data browsers, aggregators, or visualization tools. It is based on the popular Semantic Web framework Jena developed by HP Labs Bristol and can be easily extended and integrated into other applications.

I. INTRODUCTION

Primarily, there are two main reasons why the generation of statistics for RDF sources is a very important aspect when building Semantic Web applications: firstly, supporting the user when navigating and secondly, supporting applications when processing large and possibly distributed RDF graphs. The main focus of RDFStats accounts for the latter, however, it has been designed in a generic way to serve as a basis for other projects. As a sub-project of the Semantic Web Integrator and Query Engine (SemWIQ) [2] developed at the University of Linz, its purpose is the collection of statistics for datasets behind SPARQL endpoints. However, the generator can also be used to collect statistics from local or remote RDF documents. The software is based on the popular Java-based Semantic Web development framework Jena, which has been developed by HP Labs Bristol and can be easily extended and integrated into other applications.

The main features of RDFStats include:
- generation of different statistical items like instance counts (instances per class) and histograms (per class, property, value type)
- two generators for RDF documents (local files or Web resources) and SPARQL endpoints
- RDFStatsModel API for accessing statistics and various statistical estimation functions – it is based on Jena’s RDFDataset Model class and is able to manage statistics for multiple RDF sources, each represented by an RDFDataset
- An estimation function is available which is able to parse filter expressions and take these restrictions into account (e.g. how many triples are there for instances of rev:Review and property values for dc:date $\geq "2009-01-01"$).
- RDFStatsUpdatableModel API, which provides operations for adding and updating statistics (can be used in multi-threaded applications)
- RDFDataset API for accessing meta data about an RDF source
- The Histogram API provides typical histogram operations like estimating absolute and relative quantities and calculating cumulative sums.
- Different histogram types for the most important XML data types like boolean, integer, float, dates, strings, etc. For strings, there exist two different histogram types: the OrderedStringHistogram compresses strings into prefix-bins keeping the string’s order and thus supporting range estimations, and the nominal-scaled SimpleStringHistogram, which is also used for resource URIs. For all other value types, a nominal-scaled generic single-bin histogram is used as long as not a more specialized builder has been registered.
- The extensible histogram builder API allows to register further histogram builders and histogram types for custom data types.
- A base64 codec is used to encode and decode histogram data. A custom histogram type can read or write arbitrary additional data to the input and output streams.
- The optional RDF-based configuration uses the Jena Assembler mechanism to transparently persist statistic models. Thus, it is a matter of changing some lines in the configuration file to decide whether to persist the statistics into files, a relational database, a Jena TDB instance, etc.

In the following section related work will be discussed. In

1A cooperation is planned with DERI Galway to use RDFStats for the automatic generation of voID descriptions [1].
Section III the basic concepts behind the statistics collection process and histogram generators are described and in Section IV it is shown how the acquired statistics are applied. The last section concludes the paper.

II. RELATED WORK

Depending on an application’s requirements, there are different approaches of collecting statistics for RDF sources. For instance, for the Semantic Web search engine SWOOGLE [3], several features are extracted like ontology-ratio (ratio between t-box and a-box statements), term reference relations and OWL import relations between different RDF documents, as well as several other inter-RDF source relations. By contrast, the purpose of RDFStats is targeted towards single sources. It provides more in-depth statistics about specific RDF sources which are, for instance, used by the SemWIQ federator and query optimizer.

Some more closely related work includes the statistics generated by Jena TDB (tuple database), which is a persistent native RDF store. The TDB optimizer [4] uses these statistics to improve the estimated number of matching triples for single triple patterns during the execution of SPARQL queries. Based on this information, triple patterns are re-ordered dynamically during the evaluating of basic graph patterns. TDB statistics include frequency counts for predicates (e.g. (foaf:knows 12)) and more detailed rules which are made up of triple patterns and a count estimation for the approximate number of matches the triple will yield. Compared to RDFStats, the statistics are for plain RDF graphs and therefore more generic. The TDB statistics generator is not supposed to be used for other applications.

In relational database systems histograms are commonly used by query optimizers. Although newer approaches like wavelets have been introduced, histograms are still the favorite approximation method of all commercial database systems [5]. A histogram on an attribute A is constructed by partitioning the source distribution of A into n mutually disjoint subsets called bins. The frequencies and values in each bin can then be approximated in some common fashion. Since histograms for database optimizers have been proposed [6], much research has been undertaken and many different approaches and algorithms have been published [5].

The vocabulary used by RDFStats to describe statistics is based on SCOVO (Statistical Core Vocabulary [7]). Another vocabulary, called Void (Vocabulary of Interlinked Datasets [1]) will be used in future to extend RDFStats datasets with void descriptions.

III. STATISTICS GENERATION

Collecting statistics in order to facilitate the management and to improve the performance when querying large datasets is a usual task when implementing information systems. Although a great amount of work has been done in the database field, actually implementing such a system seams more trivial that it actually is. It involves the automatic generation and dimensioning of histograms for a range of different data types, but also the implementation of feasible estimation algorithms when accessing the collected statistics (see Section IV).

There are many options of generating statistics from RDF graphs. The most generic approach is to record how often RDF properties are used (e.g. Jena TDB statistics). More details can be collected by generating histograms for each distribution of RDF property values. However, in case of RDFStats, also classes are taken into account. This decision was mainly taken because SemWIQ also uses type information for query federation. However, in a future version RDFStats may be extended to create histograms also for all non-typed resources in addition to the class-specific histograms. The generation process is based on SPARQL queries, which requires (depending on data) approximately 30% less amount of data to be transferred as if processing all triples in a naïve way. It can be best described using the flowchart in Figure 1.

The process starts with the initialization of a new or already existing RDFStats dataset, which describes the statistics for a specific RDF source, e.g. a SPARQL endpoint. Then the generator fetches the set of classes used in the RDF source using the SPARQL query pattern DISTINCT { [] a ?c}4. For each class, an InstanceCount item (sub-class of

4Although not widely known, the WHERE keyword is optional as defined in the SPARQL specification.
SCOVO item) is created which stores the number of instances and optionally also the set of all occurring instance URIs\(^3\). It first tries to execute `COUNT ( * ) WHERE \{ [ ] a c \}`, where \(c\) is the class obtained before. If `COUNT` results in an exception (because it is not part of the official standard, however, most implementations support this as an extension), or if instance URIs should be fetched, the generator executes `DISTINCT ?u \{ ?u a c \}` and scrolls through all solutions. Fetching instance URIs for large datasets can be very costly. Therefore, RDFStats should be executed directly at the endpoint whenever possible. If the endpoint does not support the `COUNT` extension, even counting instances will be very costly. However, `COUNT` and other aggregate functions are very likely to be supported by the second version of SPARQL.

Finally, the generator creates histograms for each combination of class \(c\), property \(p\) (used in the extension of \(c\)), and type range \(t\) of property values. Because a property value can be a (plain or typed) literal, blank node, or URI, it is necessary, to generate different histograms for each of the occurring types. For the most important XSD types, there are already corresponding histogram generators as well as histogram implementations available. For custom datatypes, additional histogram and generator implementations may be registered at the RDFStatsGeneratorFactory.

A. Histogram Generation Process

RDFStats separates the code used for building histograms including encoding/decoding and accessing actual histogram data into two different sets of classes implementing the interfaces Histogram and HistogramBuilder. The APIs of both interfaces are depicted in Figure 2. The type hierarchy is only shown for the histogram types, but for each histogram type there must be a corresponding builder which can be registered at the HistogramBuilderFactory. This way, RDFStats can be easily extended with additional histogram types and corresponding builders. The idea behind the code separation is basically that the builder collects usually unordered values and finally creates the histogram. The histogram data can then be stored into an RDFStatsModel. To generate histograms for SPARQL endpoints or RDF documents the RDFStatsGeneratorFactory is used. The complete API will not be discussed in detail, but the most important artifacts are named here for reference.

Histograms can be generated for nominal and metric scales. The values of a metrically scaled source distribution are comparable. Examples are numbers, dates, but also short strings like names, which can be compared lexicographically. A nominal scale exists for boolean values but also for URIs, which are in fact strings, but have another intrinsic meaning and therefore should not be compared lexicographically. Similarly, longer texts should not be compared lexicographically. As can be seen in Figure 2, there are two histogram types for strings: the SimpleStringHistogram and the OrderedStringHistogram. The first one uses a histogram bin for all distinct values and counts the occurrences of equal strings. The second one is repeatedly cutting strings to common prefixes if the required amount of bins for distinct string prefixes exceeds a configurable value. While the first does not implement the ComparableDomainHistogram methods like `getCumulativeQuantity()`, etc. the second type does, and is therefore used for any plain and string-typed literal. The SimpleStringHistogram is used for URIs mainly. It has the big disadvantage, that it may become very large, because distinct strings are not compressed into equal histogram bins. In the case that the extension of an RDF class (its instances) has a large number of predicates with distinct URIs, the histogram will become very large. A possible solution and future work may be hashing. However, it is very difficult to find an appropriate hash function for any conceivable source distribution.

B. Order-preserving String Histogram

Automatically generating histograms for strings is not trivial and it is even more difficult to find a fast, memory-efficient, and all-time best information preserving approach for keeping the lexicographical order of the original string values. Especially for SemWIQ this information is very important in order to be able to optimize range queries like \{ \{ foaf:name \n FILTER (?n > "M") \} \}. The basic idea for an algorithm was to cut down strings to common prefixes. In a first version, to save memory, this compression was applied during the building phase each time the allocated number of bins exceeded a specified maximum value. However, now each histogram type in RDFStats does also store the number of distinct values occurring in the source distribution (at the extra cost of memory). Therefore, the compression algorithm printed as Algorithm 1 can only be applied at the end of collecting values when creating the histogram.

Generally, a string histogram can be thought as a set \(V\) of tuples \((str, card)\). Each of these tuples represents a bin with a quantity or cardinality \(card\) and a unique string label \(str\). During the building phase, the occurrences of equal strings

\(^3\)Instance URIs are, for example, collected for SemWIQ to be able to process federated `DESCRIBE` queries and queries with URI subjects.
Algorithm 1 builds an order-preserving string histogram

Require: \( V \), the set of collected distinct values with tuples \((str, card)\) where \( str \) is a string and \( card \) its cardinality.

\( W \), immediately gathers tuples \((str, card)\) like \( V \).

Require: \( max \), the maximum occurring string length in \( V \).

Require: \( size \), the preferred final histogram size.

if \( | V | \leq size \) then
  return \( V \)
end if

\( len \leftarrow max \)
repeat
  \( W \leftarrow \emptyset \)
  \( len \leftarrow (\text{int}) len/2 \)
  \( break \leftarrow false \)
  for all \((str, card) \in V\) do
    \( prefix \leftarrow \text{leftTrim}(str, len) \)
    if \( \text{containsStr}(W, prefix) \) then
      \( W \leftarrow W \setminus \{(prefix, card)\} \)
      \( card \leftarrow card + \text{getCard}(W, prefix) \)
    end if
  end for
  if \( |W| \leq size \lor len = 1 \) then
    \( W \leftarrow W \cup \{(prefix, card)\} \)
  else
    \( break \leftarrow true \)
    break for
  end if
until \( |V| \leq size \lor len = 1 \)

return \( W \)

\( str \) are summed up leading to the integer \( card \). In case of the \( OrderedStringHistogram \), this set \( V \) is finally compressed into a smaller set \( W \) also with tuples \((str, card)\) by Algorithm 1 as will be explained briefly.

The algorithm is only applied if \( |V| > size \), otherwise \( V \) is just returned. In this case, the histogram will keep the full information (full-length bin labels). Otherwise, the compression is applied repeatedly until \( |V| \leq size \) or the length of the prefix labels is finally just 1. This means, that at least one character will be kept as a label. In the worst case, the histogram will need as many bins as distinct first characters exist. During each iteration, the intermediate set \( W \) is filled with trimmed prefixes \( prefix \) from \( V \). The trim length \( len \) is initialized with the maximum length \( max \) found in the set of original strings, and divided by 2 each iteration. As a result of trimming, prefixes will fall into equal bins leading to less bins of \( W \) in total.

C. Persisting Statistics

Collected statistics can be stored into any Jena model using Jena’s assembler mechanism. RDFStats already provides an easy way for configuration based on RDF files which may include assembler configurations. In order to schedule RDFStats as a cron job, the configuration file approach is more convenient than using command line arguments. Additionally, RDFStats configurations may be integrated into other RDF-based configuration files. This way, RDFStats can be seamlessly integrated into other applications like SemWIQ for instance. Alternatively, RDFStats can also be used with command line arguments and without configuration files.

The RDFStats vocabulary is depicted in Figure 3. A model may contain multiple RDFStats datasets. Each dataset has a \texttt{stats:sourceUrl} and a \texttt{stats:sourceType}, which has to be one of \texttt{stats:SPARQLendpoint} or \texttt{RDFDocument}. Depending on the source type, a different generator is used. During the process described before, several \texttt{stats:InstanceCount} and \texttt{stats:Histogram} items are generated with different dimensions. Besides having an \texttt{rdf:value}, each instance count has a class dimension and each histogram has a class, property, and range dimension. The \texttt{rdf:value} of a histogram is a base64 encoded string which is written and read by the corresponding histogram builders.

IV. USING AND ACCESSING STATISTICS

RDFStats statistics can be accessed by the APIs provided by the interfaces \texttt{RDFStatsModel} and \texttt{Histogram}.

A. The \texttt{RDFStatsModel} API

The model API depicted in Figure 4 provides many useful methods for retrieving RDFStats datasets, instance counts, and histograms. Most of the methods require a string \texttt{sourceUrl}, which constraints the obtained statistical items to a specific dataset identified by \texttt{stats:sourceUrl}. There are methods for getting the set of occurring classes and properties in the source graph, methods for obtaining instance counts, the known types for a specific instance URI, for asking whether a dataset describes an instance, and several other families of methods. The \texttt{hasHistogramFor()} methods can be used to ask for available histograms with different combinations of known class, property, or range URI. If all three parameters including the \texttt{sourceUrl} are known, the histogram can be obtained with \texttt{getHistogram()}. All these methods may be combined to fetch all histograms for a dataset. There are
also methods for fetching the total number of instances for a given class, or a set of classes and methods to obtain the URIs of instances. The triplesCount+ family of methods provides several exact and approximated calculations for triples available in an RDF source.

One of the triplesCount+ functions can take a SPARQL filter expression (ExprList) to provide logically combined multi-range estimations based on cumulative sums. The estimation is similar to the calculation of the mintern selectivity described in [8, Sect.5.3]. A valid filter expression is a simple or composite expression on a predicate variable including constants, unary operators (+, −, unary functions (isURI ()), datatype ()), binary arithmetic (+, −, /, *), logical (∧, ∨, ¬), and comparison operators (, ≠, ≤, ≥). Other n-ary functions are not supported. If the expression contains multiple variables, it is invalid because the variable is supposed to be bound by the potential value distribution represented by a single histogram only.

An RDFStats model can be easily created from an RDF file or another Jena model containing statistics by calling one of the RDFStatsModelFactory.create() methods. The RDFStatsUpdatableModel provides additional methods for manipulating models. In multi-threaded environments, an exclusive write lock can be obtained before calling the manipulation methods. These methods are also used by the generators.

B. The Histogram API

The histogram API provides typical methods for accessing histograms: getting the index of the bin a value falls into, getting a bin’s quantity, estimate the absolute and relative quantity for a value, etc. For histograms with a comparable domain, there are additional methods for getting the minimum and maximum as well as cumulative estimation functions. All the methods and also the available histogram implementations are shown in Figure 2.

V. CONCLUSION

Although RDFStats was implemented because statistics for remote SPARQL endpoints are essential for the SemWIQ federator and optimizer, it has been designed as a stand-alone and highly extensible framework for collecting RDF statistics. This decision was mainly taken, because the authors faced a real lack of efforts in this field when doing research towards statistics for RDF datasets.

RDFStats provides a solid framework for generating statistics from RDF sources like documents and SPARQL endpoints. It is highly extensible and it is expected that there will be additional generators, histogram types and corresponding builders, estimation functions and other tools (e.g. for visualization) developed in future. It has been released as an open source project under the Apache Software License 2.0 at http://semwiq.faw.uni-linz.ac.at/rdfstats.

For human users it is often rather difficult to get the big picture of the information contained in large datasets exposed by Semantic Web applications. Although linked data browsers can be attached to SPARQL endpoints to provide a navigational way of interaction, they usually do not provide a summary of contents. RDFStats could be integrated into user interfaces and other Semantic Web applications to provide this information but also to support tools to achieve a better performance when processing a large amount of data.

ACKNOWLEDGMENTS

RDFStats and SemWIQ are part of G-SDAM, funded by the Austrian BMWK (Federal Ministry for Education, Science and Culture), contract GZ BMWF-10.220/0002-II/10/2007.

REFERENCES