Sampling Semantic Data Stream: Resolving Overload and Limited Storage Issues

Naman Jain[1], Manuel Pozo[2], Raja Chiky[2], and Zakia Kazi-Aoul[2]

[1] VIT University, Vellore, TN 632014, India
namanjain2009@vit.ac.in
[2] ISEP - LISITE, Paris 75006, France
{manuel-jesus.pozo-ocana, raja.chiky, zakia.kazi}@isep.fr

Abstract. The Semantic Web technologies are being increasingly used for exploiting relations between data. In addition, new tendencies of real-time systems, such as social networks, sensors, cameras or weather information, are continuously generating data. This implies that data and links between them are becoming extremely vast.
Such huge quantity of data needs to be analyzed, processed, as well as stored if necessary. In this paper, we propose sampling operators that allow us to drop RDF Triples from the incoming data. Thereby, helping us to reduce the load on existing engines like CQELS, C-SPARQL, which are able to deal with big and linked data. Hence, the processing efforts, time as well as required storage space will be reduced remarkably.
We have proposed *Uniform Random Sampling*, *Reservoir Sampling* and *Chain Sampling* operators which may be implemented depending on the application.

Keywords: Big Data, Linked data-stream, Processing time, Sampling

1 Introduction

The semantic web handles many systems, such us Twitter, Facebook or Google, which generate increasing volumes of semantic data everyday. The problem of "too much (streaming) data but not enough (tools to gain and derive) knowledge" was tackled by [7]. They envisioned a Semantic Sensor Web (SSW), in which sensor data are annotated with semantic metadata to increase interoperability and provide contextual information essential for situational knowledge. CQELS[6], SPARKWAVE[5], C-SPARQL[3] etc. are existing technologies to exploit these semantic and streaming (continuous and infinite) data, and are based on recommended standard RDF, as the format of representation.

CQELS[6] is a native approach in an RDF environment based on 'white-boxes'. It provides its own processing model and its own operators to deal with streams, for example, window operators or query semantic operators. C-SPARQL[3] on the other hand, uses a 'black-box' approach which delegates the processing to other engines such as stream/event processing engines and SPARQL query processors by translating to their provided languages.

T. Herawan et al. (eds.), *Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013)*, Lecture Notes in Electrical Engineering 285, DOI: 10.1007/978-981-4585-18-7_5,

Although almost all the engines are based on the SPARQL Language, there are only a few systems which are able to process big quantity of data on the fly. Moreover, these engines do not feature any tool that would allow them to reduce the processing efforts and improve the processing time. For many applications, we must obtain compact summaries of the stream. These summaries could allow accurate answering of queries with estimates, which approximate the true answers over the original stream [4].

Thus, we propose the implementation of such sampling operators that could be used in conjunction with other existing real-time engines. These sampling operators will allow us to deal with the requested population by applying heuristic methods. *Uniform Random Sampling*, *Reservoir Sampling* and *Chain Sampling* were implemented on the data streams and were then compared with the error percentage, storage requirements and the load suffered by the engine. Thus, these sampling methods will help reduce processing time and the required memory space.

2 Extension to Existing Systems

We propose to extend existing semantic data stream querying engines by creating an external abstraction of the sampling operator. The extension acts as follows:

First, we recognize the different operators of the language and split the initial query according to them. This will allow us to use the same operator at different levels of the query: in the input, for sampling static data or streaming data and in the output to sample the result of the query, or both simultaneously.

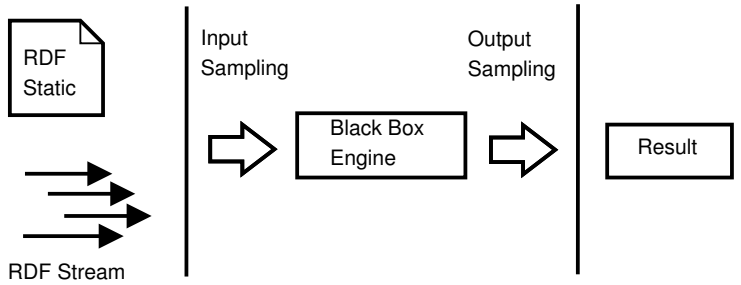


Fig. 1. Block Diagram

After splitting the query, we make syntax correction. If we identify any errors in our operator, we stop the execution of the query. Hence, if everything is correct, we create a key-value map where we store all the resources and all the requested sampling actions. Thus, we can apply the sampling operator independently by creating a thread for each of them.

We start the abstraction of the operator by taking out all the sampling instances in the query. This will allow us to avoid the correction of the rest of operators and leave the task to the specific engine itself.

When we send this query to the engine, as shown in Figure 1, it will run all the threads in charge of each source. By adapting those threads, we apply the sampling method just before the data comes to the engine, thus discharging the processing tasks. In a similar way, we adapt the output of the engines, in order to apply sampling methods at the output and lighten storage space.

3 Sampling Methods

The query would contain the information of sampling type and the sampling percentage for each unique stream. We apply the appropriate sampling method for each stream in different threads. We used *Uniform Random Sampling*, *Reservoir Sampling* and *Chain Sampling* to compare their advantages and disadvantages. We may choose an appropriate sampling method depending on the application used for.

We have implemented our sampling operators with CQELS[6] and C-SPARQL [3] engine. Example below shows simple type of sampling at 50% i.e. taking one triple and dropping the next one, using CQELS as:

```
PREFIX lv: <http://deri.org/floorplan/>
SELECT ?person ?locName
FROM NAMED <C:/floorplan.rdf>
WHERE {
STREAM <C:/rfid.stream> [NOW] [SAMPLING %50]
{?person lv:detectedAt ?loc }
GRAPH <C:/floorplan.rdf> {?loc lv:name ?locName } }
[OUTSAMPLING %80]
```

We implemented *Output Sampling* using
Operator: [**OUTSAMPLING %** {Sampling Percentage}]

As shown in Figure 1, the output of the engine goes through this operator and is given as final result only if the operator permits. Normal sampling algorithm has been used for this i.e. if %80 is specified, then only 4 out of 5 results are termed as final results. If *Output Sampling* operator is not specified, then the percentage is assumed to be 100 and no result is left out after the processing is done by the engine.

The sampling operators can also be implemented on similar systems like Sparkwave[5], ETALIS[1] after changing few configurations. In this paper, we use CQELS engine for the experimentation and the query is about computing average of *8-hour daily maximum ozone concentrations* as measured by *Clean Air Status and Trends Network* (CASTNET) in United States of America. The RDF data contains 2,159,133 triples consisting of 308,380 number of entries.[3]

[3] `http://data-gov.tw.rpi.edu/wiki/Dataset_8/`

3.1 Uniform Random Sampling

In this sampling method, incoming triples are sent to the engine, only if uniform random generation of true/false, with a probability equal to sampling percentage divided by 100, returns true. This helps to keep number of samples in proportion to total incoming data.
Operator: [**UNISAMPLING** %{Sampling Percentage}]

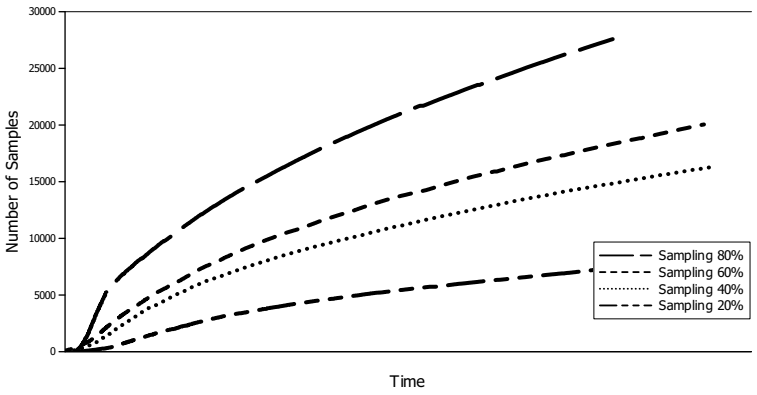


Fig. 2. Sample size variation/ Number of triples processed by engine in Uniform Random Sampling (During time span of 1 minute)

The query[4] is of the form

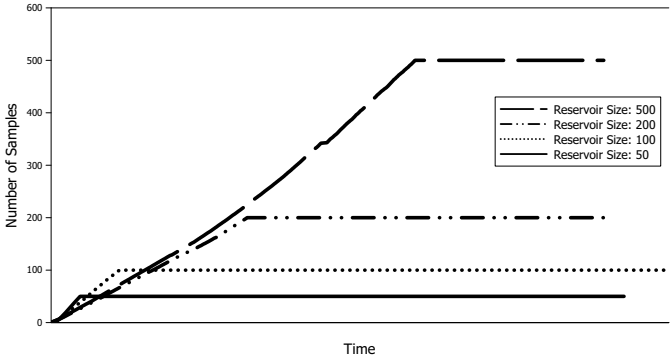
```
WHERE {
STREAM <C:/data-8.stream> [NOW][UNISAMPLING %60]
{?location vocab:ozone_8hr_daily_max ?value} }
```

Figure 2 shows sample size variation and number of triples processed by the engine. They both are same in this type of sampling as there is no removal of elements from the sample. This sampling method is less preferred as the sample size keeps on growing, which may lead to shortage of storage space, and even the outdated data element will be existing.

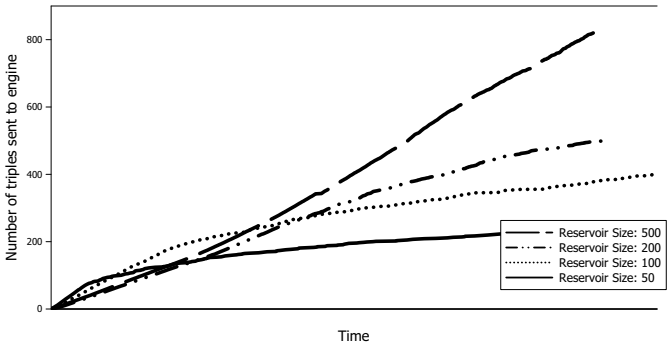
3.2 Reservoir Sampling

The problem of maintaining a sample of specified size 'k' is overcome by *Reservoir Sampling*[8]. In this, we add the triple 'i' to the sample with probability k/i and discarding a randomly chosen element from the reservoir (the sample) to make room for the new element.
Operator: [**RESSAMPLING** {Reservoir Size}]

[4] `PREFIX` & `SELECT` operators are omitted here to avoid repetition



(a) Sample Size variation after using Reservoir Sampling



(b) Load on engine after using Reservoir Sampling

Fig. 3. Plot of sample size variation and load on engine after using Reservoir Sampling (All plots are for the same time duration of 1 minute)

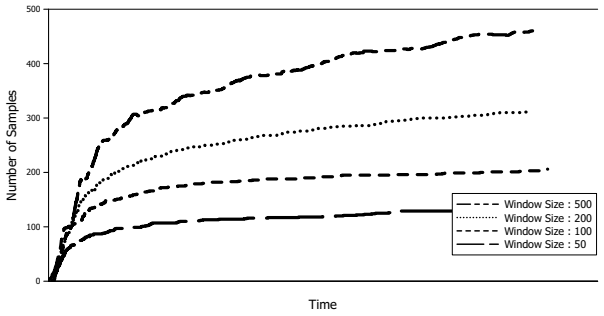
For instance, here reservoir size of 500 ensures that the number of samples at any point may not exceed 500. Initially, triples are added to the reservoir until it is full. Then incoming triples, with stream number 'i' are added with probability 500/i, after replacing a randomly chosen element from the reservoir. The query becomes:

```
WHERE {
STREAM <C:/data-8.stream> [NOW][RESSAMPLING 500]
{?location vocab:ozone_8hr_daily_max ?value} }
```

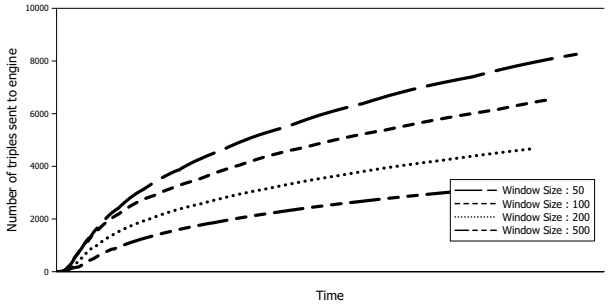
Figure 3a shows that the sample size becomes constant after the reservoir is full and Figure 3b shows number of triples processed by the engine over the time. This sampling method helps us to limit the maximum storage space of the samples, but is not favorable when recent data is more important, as the sample data may have 'expired'. A significant reduction in load on the engine can also be noted in Figure 3b when compared to Figure 2.

3.3 Chain Sampling

Babcock et al. [2] proposed the *Chain Sampling* method for sequence based windows. In this method, if the window size is 'n' we add each new element 'i' in the sample with probability min(i,n)/n. As each element is added to the sample we choose an index, in the range (i+1,i+n), of the element that replaces it when it expires. Once the element with that index arrives, we store it and choose the index which replaces it when it expires, thus forming a chain of replacements. STEP operator is used to specify the size of steps that the sequence based window takes.



(a) Sample Size variation after using Chain Sampling



(b) Load on engine after using Chain Sampling

Fig. 4. Plot of sample size variation and load on engine after using Chain Sampling (All plots are for the same time duration of 1 minute)

Operator: [**CHNSAMPLING**{WindowSize} **STEP**{StepSize}]
and the query is of the form

```
WHERE {
STREAM <C:/data-8.stream> [CHNSAMPLING 500 STEP 2]
{?location vocab:ozone_8hr_daily_max ?value} }
```

The above query shows that window size 'n' is 500 and window moves in steps of 2. Figure 4a shows that the sample size increases as the window size is increased but Figure 4b shows that number of triples being processed by the engine is lower for higher window sizes. We also note from Figure 3b and Figure 4b that load on the engine is much higher for *Chain Sampling*.

Chain Sampling ensures to keep sample elements which are present only in the window and eliminates the problem of 'expired element' in the sample. But, the act of storing replacements does not reduce the memory space requirements and load on the engine as effectively as *Uniform Random Sampling* and *Reservoir Sampling*.

Table 1. Variation of Sample size, Load on engine and Error due to different sampling methods

Method	Parameter	Sample Size	Triples Processed	Effective Sampling %	% Triples Processed	% of Error
Uniform	20	10034	10034	20.06	20.06	0.1819
Random	40	19993	19993	39.98	39.98	0.1540
Sampling	60	29998	29998	59.99	59.99	0.0793
[Sampling %]	80	40002	40002	80.00	80.00	0.0494
	500	500	2807	1.00	5.61	1.3532
Reservoir	1000	1000	4925	2.00	9.85	0.8041
Sampling	2000	2000	8445	4.00	16.89	0.6047
[Reservoir Size]	5000	5000	16490	10.00	32.98	0.3123
	10000	10229	42589	20.45	85.17	0.4410
Chain	20000	13183	36332	26.36	72.66	0.5774
Sampling	30000	14857	31260	29.71	62.52	2.5181
[Window Size]	40000	16116	27298	32.23	54.59	0.6559

4 Experimentation

The queries in Section 3 were first executed without any sampling, and then after incorporating different sampling operators. After computing 50,000 entries of *8-hour daily maximum ozone concentrations*, without any sampling operator, their average was found to be **49.9161**. Also, the average of results obtained by 20 iterations (after applying each sampling operator) is demonstrated in the Table 1. All results are for 50,000 entries. The % of error was also computed for all sampling operators with a specific parameter as shown in Table 1. It was done by finding average of relative difference, of experimental value from actual value (49.9161, in this case), for all 20 iterations of the same query.

We may note that in *Uniform Random Sampling* probability of error reduces as the sampling percentage is increased while load on the engine is in coalition with the sampling percentage specified. In *Reservoir Sampling*, we see that

the probability of error reduces with increase in reservoir size and a significant reduction in sample size as well as load on the engine is observed.

In *Chain Sampling* percentage of error depends on the elements in the current window and an improvement in sample size is observed in comparison to *Uniform Random Sampling* but load on the engine is comparatively high as sample replacements are also required to be processed by the engine.

5 Conclusion

The growing generated data from web applications is becoming a problem for the processing systems, and the relation between data is causing troubles when attempting to exploit data repositories. Therefore, In this paper we have proposed an extension of a real-time request system that allow us to reduce processing tasks and memory space requirements.

Different sampling methods suggested are useful for different applications. For example, if accuracy is required then we may go for *Uniform Random Sampling* with appropriate sampling percentage, and if memory size is fixed, we should go for *Reservoir Sampling*. *Chain Sampling* with suitable window size is favourable only when the samples need to be 'new'. In near future, we will build the sample according to the importance of incoming data elements, rather than randomly choosing them.

References

1. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in etalis. Semantic Web, 3(4): 397–407 (2012)
2. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 633–634. Society for Industrial and Applied Mathematics (2002)
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-sparql: Sparql for continuous querying. In: Proceedings of the 18th international conference on World wide web, pp. 1061–1062. ACM (2009)
4. Cohen, E., Cormode, G., Duffield, N.: Structure-aware sampling on data streams. In: Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, pp. 197–208. ACM (2011)
5. Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over rdf data streams. In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 58–68. ACM (2012)
6. Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: The Semantic Web–ISWC 2011, pp. 370–388. Springer (2011)
7. Sheth, A., Henson, C., Sahoo S. S.: Semantic sensor web. Internet Computing 12(4), pp. 78–83. IEEE (2008)
8. Vitter, J. S.: Random sampling with a reservoir. ACM Transactions on Mathematical Software (TOMS), 11(1):37–57 (1985)