Implementing Triple-Stores using NoSQL Databases

Eleni Stefani
Department of Informatics
Faculty of Natural Sciences
University of Tirana
Tirana, Albania
eleni.stefani@fshnstudent.info

Klesti Hoxha
Department of Informatics
Faculty of Natural Sciences
University of Tirana
Tirana, Albania
klesti.hoxha@fshn.edu.al

Abstract

Knowledge bases empower various information-retrieval systems nowadays. The usual implementation of them is through RDF based triple-stores. The available toolkits that enable this are usually less mature in comparison with well established document-oriented NoSQL databases. In this work we report on an alternative implementation of triple-stores using NoSQL databases. In comparison with similar solutions in this regard, we decided to not use RDF at all, therefore no data or query mapping was needed. We propose the implementation of a vocabolary using a separate document collection. This would also facilitate the dynamic enlargement of it in automated fact-extraction scenarios. Our results show that using a document-oriented NoSQL database for storing and retrieving triples offers a considerable performance. The involved preprocessing needed because of the limitations of a non RDF based solution, did not affect this. The achieved performance was also higher in comparison with doing the same knowledge retrieval operations using a purposely built linked data toolkit.

1 Introduction

Triple-stores [Rus11] are the first choice implementation of linked data knowledge bases [BHBL11]. The usual representation of a triple-store is through a RDF based solution. It supports the definition of ontologies and relations between entities using a standardized approach. There are several open source tools that back up this solution, but unfortunately none of them have an industry acceptance comparable with other traditional ways of storing data (i.e. RDBMS, NoSQL).

Big players of the information retrieval landscape have already reported success stories by using a knowledge base for enriching their information output [Don16], [Pau17]. However these are proprietary implementations that have not been open sourced so far. Considering the increase in usability perception when incorporating information from a knowledge base in search results [ALC15], many have created or planned adding a knowledge base (usually tripe-store powered) in their actual information retrieval system setup.

The lack of fully mature solutions in this regard and the complexity of them, have hindered the ubiquity of knowledge bases in comparison with traditional data stores. Furthermore, for simple scenarios that just try to gain advantage of linked data, having to deal with a constraining RDF schema might be unnecessary.

On the other hand, NoSQL databases are very well accepted at the time of this writing. They offer an incomparable performance in extensive data creation scenarios, are very scalable, and the existing solutions for implementing them allow for quick deployment in traditional servers or in the cloud. Furthermore, there is an already trained crowd of developers with hands on experience with this data store category.

In this work we report on a prototype implementation of a triple-store using a document-oriented NoSQL database. It does not use RDF, just simple subject-predicate-object triples stored in a MongoDB[1] database. Our goal is to provide preliminary insights of using already established NoSQL databases for storing graph oriented data (a typical setting for linked data contexts). We aimed on experimenting about incorporating triple structured data in information systems without having to rely on a heavyweight RDF manipulation framework.

[1] https://www.mongodb.com/download-center/community

We provide a sample vocabulary based on DBPedia [ABK+07] predicates. It was stored in a separate document collection. Our experiments were performed using a subset of DBPedia knowledge. We evaluate our approach in terms of performance comparing it also with Apache Jena[2], an open source Linked Data framework that supports RDF based implementations of triple-stores.

In the rest of this paper after giving a short overview of various triple-store implementation approaches in section 2, we give detailed insights of the developed prototype in section 3. The paper is concluded with detailed data about the evaluation of our prototype.

2 Triple-Store Implementation Approaches

There are several approaches for implementing triple-stores. The most frequent ones are purpose-built implementation frameworks or graph databases.

2.1 Purpose-built frameworks

Also named as native triple stores, purpose-built frameworks are technologies developed for storage and retrieval of RDF data.

Apache Jena

Jena is a Java based framework for dealing with semantic web/linked data scenarios. It provides a Java library that allows the manipulation of RDF graphs. It supports RDF, RDFS, RDFa, OWL for storing triples (according to published W3C recommendations) and SPARQL Query Language for retrieving information from graphs. The allowed data serialization are RDF/XML, Turtle and Notation 3. Apache Jena also includes a SPARQL server, Apache Jena Fuseki which can be run as a stand alone server. It offers access to the same Jena features using a HTTP interface.

Sesame

Sesame is another alternative for implementing triple-stores using RDF data [BKVH02]. It needs a repository for data storage, but this repository is not included in Sesame architecture, that's why Sesame is database-independent. It can be combined with a variety of DBMSs. In its architecture it contains a layer, named SAIL (Storage and Inference Layer) for managing communication with the database in use. Sesame can only accept queries written in SeRQL (a RDF query language) and converts them in queries suitable to run on the underlying repository.

[2]https://jena.apache.org/

4Store

4Store is a RDF DBMS which stores RDF data as quads, adding an additional property for storing the graph name. 4store uses a custom data structure for storing the quads data and it also uses its own tool for querying them, 4s-query [CMEF+13].

2.2 Graph Databases

Because of their structure, graph databases offer a natural option for storing triples since the standard representation of triples is also a graph. In this section we describe some examples of this category of databases.

AllegroGraph

AllegroGraph[3] enables linked data applications through a graph database and also an application framework. It offers similar features as the above described tools: storing and retrieving triple data. Data retrieval can be done using SPARQL or Prolog. It supports data serializations like N-Quads, N-Triples, RDF/XML, TriG, TriX, and Turtle formats and it can be used with various programming languages. Similarly to a relational database, it supports ACID transactions.

Virtuoso

Virtuoso Universal Server[4] is another alternative for implementing triple-stores. It can access RDF data stored in a RDBMS repository which may be part of Virtuoso itself, or an external one [EM09]. The usual database schema is relatively simple, RDF data are stored as quads in a table with four columns. A quad includes the triple and the graph name, *subject S, predicate P, object O and graph G*. Regarding the query language, Virtuoso uses a combination of SPARQL and SQL. It translates SPARQL queries to SQL ones according to the database schema.

2.3 Research Prototypes

Other approaches of implementing triples stores include research prototypes which try to exploit technologies that weren't initially developed for this purpose.

Dominik Tomaszuk [Tom10] has also experimented on implementing RDF triples in JSON or BSON documents in MongoDB. The document structure that he suggests consists of storing subject, predicate and object data as document fields. Consequently, in each document can be stored only one triple. About knowledge retrieval the author analyzes some algorithms for interpreting SPARQL queries.

[3]https://franz.com/agraph/
[4]https://virtuoso.openlinksw.com/

Franck Michel [MFZM16] in its work represents xR2RML as a mapping language for querying MongoDB with SPARQL.

Another approach tends to store RDF triples into CouchDB using JSON documents [CMEF+13]. In this approach, each document can store only one JSON object (even if technically can be more than one) where the key represents the subject of the triple. The value of the object consists of two JSON arrays, one for storing predicates and the other for storing objects of triples. The relation of predicates and objects is done according to their indexes on array. Because of this structure, a document can store more than one triple only if all triples share the same subject. To add new triples, already existing documents can be modified or new documents can be created. For running queries, the proposed system accepts SPARQL queries which are then converted to queries CouchDB can process.

3 Our Approach

3.1 Data Structure Schema

In this section is described the schema that is used for storing triples into JSON documents and how we dealt with unique identification of entities. We did not use a RDF based specification for this approach, so our sources won't be described by URIs. We wanted to experiment with creating simple knowledge graphs without the overhead involved by using traditional linked data tools (RDF, SQARQL). To cope with unique identification of entities we suggest the use of two type of documents:

- **knowledge documents** for storing knowledge,
- **entities documents** for identifying entities

Each of them needs to be stored in a separate MongoDB collection, consequently two collections are needed: KNOWLEDGE collection and ENTITIES collection.

An entity document defines only one entity and includes two fields, *_id and name*. A document of this type would look like this:

```
{
    "_id": entity ID,
    "name": entity name
}
```

Knowledge documents are responsible for storing triples. The set of all knowledge documents stored in database represents the knowledge graph.

We also run some preprocessing when inserting triples in our triple-store. Let T(S,P,O) be a triple where S is *subject*, P is *predicate* and O is *object*. Then in the respective knowledge document will be stored:

- Subject which will be replaced by its id on entity document. By default subject is an entity.
- Predicate will be stored as it is.
- Object which will be replaced by its id defined in entity documents, if is an entity, or will be stored as it is if is not.

Data Structure Schema of Knowledge Documents

Let T(S,P,O) be a triple where S is *subject*, P is *predicate* and O is *object*. Then a knowledge document structure will be as follows:

```
{
    "_id": document id,
    "subject": S(T),
    "P(T)": O(T)
}
```

A document can store information about only one subject. Two fields are required in each document: *_id*, an auto generated number attached to every document and *subject*, the actual subject of the triple. The third pair has as key the predicate of the triple in question and as value the object of the triple. In the same way we can continue adding knowledge about a certain subject. If S(T) is a set with triples then a knowledge document is created as:

```
{
    "_id": document id,
    "subject": S(T_1),
    "P(T_1)": O(T_1),
    "P(T_n)": O(T_n)
}
```

3.2 Interaction with database

In this work, we did not to use SPARQL. SPARQL is suitable only for querying RDF data which also have been excluded from this approach. Under these conditions, for database interaction we have utilized the MongoDB Query Language. The developed system offers the possibility of adding and retrieving information through triples. This can be done through an exposed RESTful API.

3.2.1 Adding Knowledge

One of the first things needed for managing knowledge is the specification of a vocabulary. We have defined the vocabulary as the set of all valid predicates that can be added to our knowledge base, specifying also the valid data type for each vocabulary entry. There are three valid data types:

1. Single value,

2. Array of values,

3. Map, or in MongoDB language inner document. In this work we support only one level of inner documents.

For example, if we specify a vocabulary entry

- name | single value

and we try to add two triples

- John name John
- John name Johny

our developed prototype will store the last valid triple, **John name Johny**. Table 1 presents some vocabulary entries used in our prototype mostly for storing triples about locations.

Table 1: Vocabulary

Predicate	Data type
Name Anthem Capital Area code Birth place Death place Foundation place	Single value
Official Languages Cities Twin Countries Citizenship of Headquarter of Residence of	Array of values
Ethnics Leaders	Map(Ethnic, Percentage) Map(Name, Organization)

Other than the document schema, there is also the need for specifying a strategy of managing documents when new knowledge is added (through an API request in our case). Generally there are two options, modifying current documents in database in order to add new triples, or creating new documents. The one followed in this approach is to create new documents for each API request that adds new data. If the request contains an array of triples, the system groups them by subject and then for each group (subject) adds a new knowledge document in the database.

3.2.2 Querying Knowledge

As mentioned above, for retrieving knowledge the system uses MongoDB Query Language. An external user only needs to create a JSON filter according to MongoDB specification and pass it to the system through RESTful calls. Again, preprocessing might be needed. It consists of replacing entity names found in filter with their corresponding ids as defined in entities documents.

There are two particular cases of querying knowledge:

1. the requested entity is known,

2. the requested entity or entities are not known, but have to be in some relation (expressed through JSON filters)

In the first case, users can send an HTTP GET request with the name of entity/subject. For example, if knowledge about Albania is requested, user sends *http://server/Albania* request and the system responds with all available triples about this subject. This is the only case where there is no need of creating a JSON filter.

In the second case, users can send an HTTP GET request that contains a JSON filter. For example, *http://server/{"type":"country","capital":"Tirana"}* which requests an entity of type country and its capital is Tirana. Other supported logical operators (other than **and**) are **or**, **in**. It is also possible to use the following comparison operators: **equal**, **less than**, and **greater than**.

Figure 1 shows the workflow diagram of querying knowledge using filters. After finding the requested entities, the system gets all available knowledge stored about them and performs a merge process for each entity.

Merge is the process of combining documents that share a common subject into one. The final document will have a field containing the subject and all the pairs of predicate/object found in the merged documents. The second process that is performed before returning a response is serialization. Serialization refers to replacing entity ids with actual names as defined in entities documents. Figure 2 shows the workflow diagram of processing knowledge before returning a response.

4 Evaluation

In this section we describe the evaluation that we have performed for the developed prototype. We have performed our experiments with a set of 3.000 triples extracted from DBPedia. The system was tested for both use cases: adding and retrieving knowledge. In addition, we have also executed some tests of querying triples in Apache Jena in order to make a comparison with our approach in terms of performance. For this purpose the same dataset has been fed as RDF to

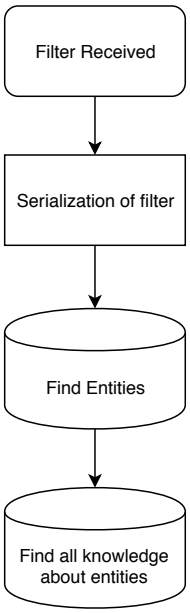

Figure 1: Filter Processing

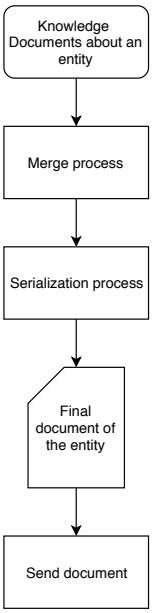

Figure 2: Knowledge Processing

the Jena framework. Because of this we cannot compare insert operations with our implemented insertion strategy.

Adding Knowledge

In our first experiment (Table 2) we test the performance of adding all triples with a single HTTP POST request. We measure the execution time which also includes all needed preprocessing before storing the knowledge: grouping triples by subject, creating entities documents, and finally adding the knowledge documents. Our triples contain knowledge about 15 entities (15 unique subjects), hence in our database after insertion will be 15 knowledge documents since all triples will be added from one request.

In the second experiment (also Table 2) we send an HTTP request for each triple that needs to be stored. Because each request contains a single triple, there will be 3.000 knowledge documents in the database.

It can be noticed that the first insertion strategy performs better. Using a single document that contains all knowledge related triples performs faster.

Table 2: Storing Knowledge

No.Triples	No.Requests	No.Docs	Time
3.000	1	1	2.3 sec
3.000	3.000	3.000	6.8 sec

Retrieving Knowledge

In order to evaluate the performance of knowledge retrieving operations in our developed prototype, we have also executed the same queries to the same knowledge set (RDF) loaded in Apache Jena. We performed this experiment using different numbers of triples. The execution time for our prototype includes all the required preprocessing steps. Other than this, we measured the performance of these queries using both triple insertion strategies described above. The experiments were performed using the same hardware in order to avoid biased results. We executed the following queries:

1. get all triples,
2. get triples where subject is Albania,
3. get triples where type is poet,
4. get triples where types is poet or political leader

Table 3 shows the results of these experiments. It can be noticed that our approach is more efficient in terms of performance. Also, when considering the response size for the same number of triples, it is bigger in Apache Jena. This is because of the RDF data overheads. In regards to the two data insertion approaches described above (that affect the number of documents in our database), results show that storing triples in fewer documents reduced the response time when retrieving knowledge.

Table 3: Retrieving Knowledge

No.	System	Triples	Docs	Resp.Size	Time
1	Ours	3.000	3.000	100KB	0.8 sec
1	Ours	3.000	15	100KB	0.2 sec
1	Jena	3.000	-	1.4MB	5 sec
2	Ours	1.000	1.000	33.3KB	0.6 sec
2	Ours	1.000	1	33.3KB	0.2 sec
2	Jena	1.000	-	76.6KB	1.8 sec
3	Ours	300	300	10KB	0.06 sec
3	Jena	300	-	70KB	1.4 sec
4	Ours	500	500	13KB	0.06 sec
4	Jena	500	-	120KB	1.8 sec

Updating Knowledge

The performance of our developed prototype is also tested in regards to updating documents. Table 4 shows the results. The dataset contains 3000 triples in total, but we experiment with a different number of documents that store them. We run the same update query for all setups. As we also noticed with our knowledge retrieval experiments, results show that storing triples in fewer documents gains a shorter response time when updating knowledge.

Table 4: Updating Knowledge Documents

Total Docs in Dataset	Affected Docs	Time
3.000	14	0.6 sec
15	1	0.04 sec
150	1	0.02 sec

5 Conclusion

In this work we propose the implementation of knowledge base triple-stores using already mature NoSQL solutions like MongoDB. Lacking of a unique identifying schema included in RDF, we propose the implementation of a vocabulary using a separate document collection. This involved the addition of some preprocessing steps when inserting or retrieving knowledge. In comparison with other related works that serialize knowledge data in a separate DBMS, we don't deal with mapping RDF stored data or SPARQL queries to the serialized dataset. This approach reduces a lot the complexity of the solution and focuses the effort on the actual triple-stored knowledge itself.

Our experiments show that by avoiding the overheads of the traditional way of storing triple structured data (RDF) and taking advantage of the performance oriented features of document-oriented databases, we can achieve a considerable performance when performing basic knowledge update operations (insert, update, retrieve). This was confirmed also when comparing the performance of our developed prototype with Apache Jena, a well established open source linked-data toolkit.

Based on our results, a better performance can be achieved when storing multiple triples per document. Increasing the number of documents considerably increased the response time.

Concluding, we showed that it is possible to implement a triple-store knowledge base using not purposely built toolkits. In various scenarios, a RDF based implementation can create unnecessary complexities that would increase the implementation time of a knowledge base. Furthermore, when dealing with previously unknown relations (fact extraction through text mining), a NoSQL based implementation facilitates a dynamic enlargement of the vocabulary.

References

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[ALC15] Ioannis Arapakis, Luis A Leiva, and B Barla Cambazoglu. Know your onions: understanding the user experience with the knowledge module in web search. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1695–1698. ACM, 2015.

[BHBL11] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[BKVH02] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[CMEF+13] Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan F Sequeda, and Marcin Wylot. Nosql databases for

rdf: an empirical evaluation. In *International Semantic Web Conference*, pages 310–325. Springer, 2013.

[Don16] Xin Luna Dong. How far are we from collecting the knowledge in the world. In *International Conference on Web Engineering*, 2016.

[EM09] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[MFZM16] Franck Michel, Catherine Faron-Zucker, and Johan Montagnat. A mapping-based method to query mongodb documents with sparql. In *International Conference on Database and Expert Systems Applications*, pages 52–67. Springer, 2016.

[Pau17] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[Rus11] Jack Rusher. Triple store, 2001. *Last Accessed: http://www. w3. org/2001/sw/Europe/events/20031113-storage/positions/rusher. html*, 2011.

[Tom10] Dominik Tomaszuk. Document-oriented triple store based on rdf/json. *Studies in Logic, Grammar and Rhetoric,(22 (35))*, page 130, 2010.