A Flexible N-Triples Loader for Hadoop

Victor Anthony Arrascue Ayala and Georg Lausen

University of Freiburg, Georges-Köhler Allee, Geb. 51, 79110 Freiburg, Germany
{arrascue,lausen}@informatik.uni-freiburg.de
http://dbis.informatik.uni-freiburg.de

Abstract. The wide adoption of the RDF data model demands efficient and scalable query processing strategies. For this purpose distributed programming paradigms such as Apache Spark on top of Hadoop are increasingly being used. Unfortunately, the Hadoop ecosystem lacks support for Semantic Web standards, e.g. reading an RDF serialization format, and thus, bringing in RDF data still requires a large amount of effort. We therefore present PRoST-loader, an application which, given a set of N-Triples documents, creates logical partitions according to three widely adopted strategies: Triple Table (TT), Wide Property Table (WPT) with a single row for each subject, and Vertical Partitioning (VP). Each strategy has its own advantages and limitations depending on the data characteristics and the task to be carried out. The loader thus leaves the strategy choice to the data engineer. The tool combines the flexibility of Spark, the deserialization capabilities of Hive, as well as the compression power of Apache Parquet at the storage layer. We managed to process Dbpedia (approx. 257M triples) in 3.5 min for TT, in approx 3.1 days for VP, and in 16.8 min for WPT with up to 1,114 columns in a cluster with moderate resources. In this paper we aim to present the strategies followed, but also to expose the community to this open-source tool, which facilitates the usage of Semantic Web data within the Hadoop ecosystem and which makes it possible to carry out tasks such as the evaluation of SPARQL queries in a scalable manner.

1 Introduction

RDF is a widely used data model to build and publish knowledge graphs, many of which are extremely large [4]. To process and query such massive graphs, centralized RDF processing models are not sufficient and therefore there is a shift towards distributed paradigms. Virtuoso, the de-facto industry standard triple-store, is capable of operating on a cluster of computers but only in its commercial form. Instead, the more general-purpose and open-source Hadoop ecosystem is designed to efficiently process massive amounts of data while being capable of scaling with commodity hardware. Technologies such as Spark and Hive allow for efficient and distributed querying of data, but the capabilities go well beyond this feature. Nowadays it is possible to train classification models, cluster data, conduct statistical analysis, and much more. It is hence no surprise that there is a growing interest in bringing Semantic Web data into that machinery to execute important tasks such as evaluating SPARQL queries. However, there is still

a lack of support for Semantic Web standards within that ecosystem. The first step towards providing any possible analysis with a sound basis is loading the data. This is the topic of the current poster presentation.

2 Background and Related Work

The most adopted logical partitioning strategies for RDF are Triple Table (TT), Wide Property Table (WPT), and Vertical Partitioning (VP) [1]. Each strategy has strengths and weaknesses for SPARQL query evaluation. A Triple Table is a single table for all triples with a schema (subject, predicate, object). This table is typically complemented with indexes for faster retrieval, but querying for long chains inevitably results in many self-joins. Systems such as Virtuoso[1] and RDF-3X [5] implement this scheme. A Wide Property Table has one row for each distinct subject and one column for each distinct predicate (properties). The values of this table are the objects for a given (subject, predicate) combination. Empty cells contain *null* values. WPTs can be very sparse, but in general they are good for star-shaped queries. DB2RDF [2] is based on a sophisticated variant of WPT. In Vertical Partitioning a table is created for each predicate which stores a (subject, object) tuple. This strategy is good for queries with bound predicates, whereas unbounded predicates cause all tables to be accessed. Systems such as S2RDF [6] implement and extend this scheme. Other systems leverage multiple partitioning schemes. This is the case for PRoST [3] which keeps a WPT and VP and can exploit both to evaluate a single query. PRoST-loader generates partitions following these three strategies, namely TT, WPT, and VP, thus leaving open the possibility of leveraging any of them. To the best of our knowledge no tools of this kind exist.

3 Loading Stages

Figure 1 illustrates the main stages of the loader. These will be described in detail with a focus on the main design decisions. As the use case we processed Dbpedia[2], one the most prominent published knowledge graphs.
(A) The Dbpedia files are located within a folder in HDFS. The program loads the data as an external Hive table. In contrast to internal tables, which have a strong bond with the corresponding HDFS, external tables use only a metadata description to access the data in its raw form. If an external table is deleted, only the metadata definition is deleted and the actual data remains intact. We then use Hive's SerDe library (acronym of Serializer/Deserializer) to parse the triple records. More specifically, we used a regular expression (RegexSerDe) to extract columns from the input files. The data is first deserialized from HDFS.
(B) The external table allows for fast access to the N-triples documents. Based on this we build the Triple Table, this time an internal Hive table, and populate

[1] https://virtuoso.openlinksw.com/
[2] https://wiki.dbpedia.org/.

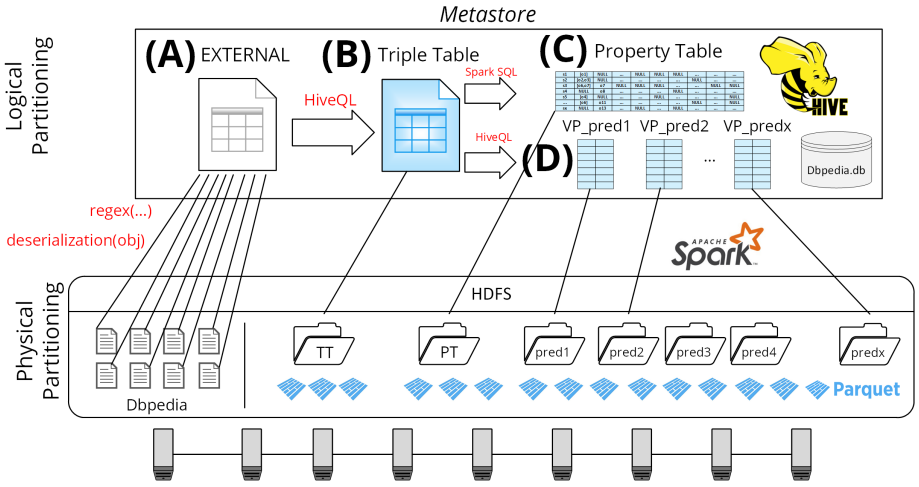

Fig. 1. PRoST-loader stages.

it by means of a HiveQL query. This is an opportunity to filter out different kinds of erroneous entries. As one example, a line in the NT document which cannot be matched by the three groups defined in the regular expression results in *null* values in the table. Moreover, when more than three elements are present in one line, multiple resources or literals might end up being mapped all together within the object column. We filter out these kinds of entries. Also, the predicate names are used later as column names in the Property Table. The maximum length of a column name in Hive is 128 characters. Therefore, predicates whose length exceeds this limit are discarded by filtering out rows in which they appear. The non-discarded rows are inserted into the Triple Table using the `INSERT OVERWRITE TABLE` clause, which at the end is stored in Parquet format.

(C) To build the Wide Property Table first we get a list of distinct predicates from the Triple Table. We need to distinguish between predicates with cardinality <1,1> and those with cardinality <1,N>. For the latter case the objects for a given subject will be stored in an array (Spark's *ArrayType*), which is later mapped to Parquet's logical types. Another control consists in making sure that all predicates are distinct, even in a case-insensitive scenario, because these are used as column names and Hive's table and column names are case-insensitive. Finally, the list of predicates and their cardinalities is used to compose a query and build the Property Table which is written to Parquet. Spark is used mainly for this task since we needed a complex user-defined aggregate function.

(D) Finally, using the list of predicates we build the VP tables, one at a time. Each table is created and populated using HiveQL and stored using Parquet.

4 Evaluation results

We performed our tests on a small cluster of 10 machines, 1 master and 9 workers, connected via Gigabit Ethernet connection. Each machine is equipped with 32GB of memory, 4TB of disk space and with a 6 Core (12 virtual cores) Intel Xeon E5-2420 processor. The cluster runs Cloudera CDH 5.10.0 with Spark 2.2 on Ubuntu 14.04. Yarn, the resource manager, in total uses 198 GB and 108 virtual cores. The most relevant settings of the cluster can be found in our github repository along with the code[3]. We used our loader to create the logical partitions for Dbpedia 3.5.1. This consists of 23 files (36.5 GB) loaded into an HDFS folder. The overall number of triples is 257,869,688. In the first stage 126 triples were removed which contained predicates with more than 128 characters. The number of distinct predicates is 39,554, of which 13,336 predicates have cardinality <1,N>. The loader was able to identify 1,328 pairs of predicates with colliding names in Hive, e.g. *dbpedia:nationalchampion*, and *dbpedia:nationalChampion*, and removed one from each pair. The Triple Table was written to HDFS in 3.55 minutes. The Vertical Partitioning finished in 3 days, 3 hours and 42 minutes. On average 8.7 tables were written per minute. For the Wide Property Table we had to remove the infobox-properties which reduced the number of properties to 1,114, since the cluster resources were not sufficient to run it on the 39K predicates. The limited WPT finished in 16.8 minutes.

5 Conclusions and Future Work

We believe there exists a deficit of tools for processing Semantic Web data in distributed frameworks such as Hadoop. Our open-source tool shows that choosing the right loading strategy is crucial to avoid wasting cluster resources and time. Hive and Parquet compression enable fast building of the logical partitions, while Spark's flexibility in defining distributed functions is essential to model data. Our tool currently uses the default physical partitioning from Hive and Spark. In the future we will explore other physical partitioning strategies.

References

1. Abadi, D.J., et al.: Scalable semantic web data management using vertical partitioning. In: Conference on Very Large Data Bases (2007)
2. Bornea, M.A., et al.: Building an efficient RDF store over a relational database. In: ACM SIGMOD Conference on Management of Data (2013)
3. Cossu, M., et al.: Prost: Distributed execution of SPARQL queries using mixed partitioning strategies. In: Conf. on Extending Database Technology, EDBT (2018)
4. Färber, M., et al.: Linked data quality of dbpedia, freebase, opencyc, wikidata, and YAGO. Semantic Web (2018)
5. Neumann, T., et al.: The RDF-3X engine for scalable management of RDF data. VLDB J. (2010)
6. Schätzle, A., et al.: S2RDF: RDF querying with SPARQL on spark. PVLDB (2016)

[3] https://github.com/tf-dbis-uni-freiburg/PRoST.