HadoopSPARQL: A Hadoop-Based Engine for Multiple SPARQL Query Answering

Chang Liu[1](✉), Jun Qu[1], Guilin Qi[2], Haofen Wang[1], and Yong Yu[1]

[1] Shanghai Jiaotong University, Shanghai, China
{liuchang,qujun51319,whfcarter,whfcarter,yyu}@apex.sjtu.edu.cn
[2] Southeast University, Nanjing, China
gqi@seu.edu.cn

1 Introduction

An increasing amount of data represented using Resource Description Framework (RDF) have appeared on the Semantic Web. By September 2011, datasets from Linked Open Data [4] had grown to 31 billion RDF triples, interlinked by around 504 million RDF links. As a consequence, it is extremely challenging to deal with the scalability issue of handling such large amount of semantic data.

SPARQL [7] is a standard query language for RDF datasets. There has been a lot of work [2,5] to handle SPARQL queries. However, most of them only treat SPARQL as a transaction-based query language, and consider low latency query answering time as an important design requirement. Furthermore, the query engine processes one query at a time and concentrates on single query optimizations. Nevertheless, users can also use SPARQL in very different scenarios. For example, two users may submit queries to a dataset about publications at the same time. The first user wants to get a list containing all authors who publish at least one proceeding and at least one article while the second user wants to get a list containing all authors who publish at least one article but not necessarily publish a proceeding. Then, Query 1 and Query 2 in Fig. 1 are submitted at the same time.

In this scenario, the system is highly desired to process these queries in parallel. [3] discussed a multi-query optimization algorithm for SPARQL. However, their method is based on revision from a set of basic graph queries into a new set of queries which may include OPTIONAL restriction. Introducing OPTIONAL restriction into queries, however, potentially increases the computational complexity. We find that the essential optimization opportunity for multiple queries lies in identifying common subqueries which may cause duplicate calculations. The engine should avoid such redundant calculations. At last, since the datasets are growing larger and larger, the scalability problem becomes more and more severe. MapReduce [1] has proved as a scalable framework to handle high latency and highly parallel tasks over large-size datasets. A natural idea is to leverage MapReduce techniques to overcome the above challenges. To this end, we built a system, called HadoopSPARQL, based on Hadoop[1].

[1] http://hadoop.apache.org/.

E. Simperl et al. (Eds.): ESWC 2012 Satellite Events, LNCS 7540, pp. 474–479, 2015.
DOI: 10.1007/978-3-662-46641-4_48

Table 1. Sample SPARQL queries

Query 1

```
SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
?article dc:creator ?person.
?inproc rdf:type bench:Inproceedings.
?inproc dc:creator ?person.
?person foaf:name ?name. }
```

Query 2

```
SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
?article dc:creator ?person.
?person foaf:name ?name }
```

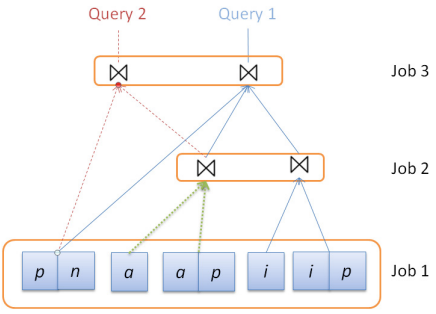

Fig. 1. An execution plan for Query 1 and Query 2

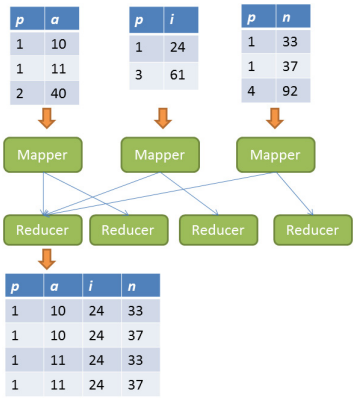

Fig. 2. Execution of a join operator

The major feature of HadoopSPARQL is that it allows the users to submit multiple queries at the same time. To handle multiple queries, we propose an algorithm to detect the common subqueries. To leverage the MapReduce framework, we use multi-way join operator instead of the traditional two-way join operator. Therefore we propose our new optimization algorithm to calculate the best join order. Furthermore, HadoopSPARQL provides a Web interface to allow accessing the underlying system using Web browsers.

2 Query Engine

The query engine consists of an optimizer to handle multiple SPARQL queries simultaneously and a Hadoop-based evaluator. In this section, we explain the key problems and design choices of the optimizer and the evaluator.

2.1 Operators

There are two kinds of operators, *data-loading operator* and join operator. Both of these two operators will produce a binding set as the result. A data-loading operator always corresponds to a triple pattern in the query. To evaluate a data-loading operator, we only need to load the data from HDFS files. Thus a data-loading operator has no input operator.

A join operator takes $k \geq 2$ inputs and produces a binding set. Formally speaking, a join operator is defined by a set of binding sets B_1, B_2, ..., B_k ($k \geq 2$) and a set of *key variables* K so that $K = S(B_1) \cap S(B_2) \cap ... \cap S(B_k)$ and $(S(B_i) - K) \cap (S(B_j) - K) = \emptyset$ for every $1 \leq i < j \leq k$. Here we use $S(B_i)$ to denote the schema of the binding set B_i. We can see that the definition of our join operator differs from the traditional multiple join operator which is widely used in RDBMS in the following aspect: We do not allow the schemas of any two input binding sets to share the same variable except the keys. For example, we do not allow to join three binding sets with schemas $\{x, y, a\}$, $\{x, y, b\}$ and $\{x, c\}$ on key x. This restriction will bring us two benefits: 1) the optimizer can leverage this information to accelerate the enumeration of candidate plans; and 2) the evaluator can efficiently calculate this kind of join.

2.2 Optimizer

The optimizer will translate a batch of SPARQL queries into one execution plan. An execution plan is a Directed Acyclic Graph (DAG) where each node in the graph represents an operator. Those nodes with 0 indegree are data-loading operators and those internal operators are join operators. The major task of an optimizer is to find the optimal execution plan. Since our optimizer is designed for multiple queries, it is composed of two parts: the first part detects the common sub-queries, and the second part employs a cost-based algorithm to generate the optimal execution plan.

The main task of the first part is to detect those duplicate sub-queries. Considering the example queries in Table 1, the following triple patterns appear in both of the two queries: $\langle$?`article`, `rdf` : `type`, `bench` : `Article`$\rangle$, $\langle$`?article, dc:creator, ?person`$\rangle$, and $\langle$?`person`, `foaf` : `name`, ?`name`$\rangle$.

Since the results of join operators will be stored in HDFS, we can reuse these results. If the two queries listed in Table 1 are evaluated together, we only need to calculate the result of the above sub-query once. By exploiting such duplications, a lot of redundant operations can be saved so that the performance can be improved.

Given a set of operators, one important problem is how to find the optimal execution plan, e.g. the best join orders. Here we employ a cost-based optimization algorithm to achieve this goal. We generate all potential execution plans, and estimate the cost of each of them. We choose the execution plan with minimal cost as our optimal plan and submit it to the query evaluator. However, there are always too many potential execution plans so that enumeration of all plans wastes a large amount of time. Several pruning techniques are applied to the optimizer to achieve an acceptable performance. In our use case study, the optimal plan of each query can be found within one second.

Figure 1 illustrates the execution plan for the example queries in Table 1. We use `a`, `i`, `n` and `p` to represent the variable `?article`, `?inproc`, `?name` and `?person` respectively. The bottom layer contains five data-loading operators corresponding to the five triple patterns. For example, a data-loading operator with schema `p, n` corresponds to a triple pattern $\langle$?`person`, `foaf` : `name`, ?`name`$\rangle$.

All nodes in the middle layer and top layer are join operators. We use solid and dashed lines to illustrate the query execution path of Query 1 and Query 2 respectively. We use dotted lines to illustrate the common sub-query. Notice the common sub-query contains only two triple patterns $\langle ?\texttt{article}, \texttt{rdf:type}, \texttt{bench:Article}\rangle$ and $\langle ?\texttt{article}, \texttt{dc:creator}, ?\texttt{person}\rangle$ instead of the three ones listed above. The reason is that in such a way, the engine only executes three Hadoop jobs instead of four, such that the execution time is reduced.

2.3 Evaluator

The evaluator will translate the execution plan (a DAG) into Hadoop jobs, and submit it to the Hadoop cluster for evaluation. The evaluator iteratively generates the Hadoop jobs. In the first round, all nodes in DAG with 0 indegree will be grouped into one job and removed from the DAG. Since all these nodes correspond to data-loading operators, the first Hadoop job will load the data from index files. Then in each iteration, all nodes with 0 indegree will be grouped into one job and removed. Since all internal nodes in the original DAG correspond to join operators, the second and later Hadoop jobs will perform joins.

For example, considering the execution plan given in Fig. 1, all the operators are grouped into three jobs which are illustrated by orange boxes. Job 1 performs all the data-loading operators, while Job 2 and Job 3 do the joins.

The implementation of data-loading operators is straightforward, thus we only discuss the implementation of join operators. A set of join operators is translated into a Hadoop job. Each join will be assigned a unique ID. If there is a join with ID id defined by input binding sets $B_1, ..., B_k$ and variable key set K, the mappers will scan all binding sets $B_1, ..., B_k$. When a mapper reads a binding $b_i \in B_i$, it will emit $(id, b_i(K))$ as map output key, and $(b_i(S(B_i) - K))$ as map output value. Therefore for each join id, every $b_i \in B_i$ with the same $b_i(K)$ will be grouped into the same reducer. Since we restrict that $(S(B_i)-K) \cap (S(B_j)-K) = \emptyset$ for every $1 \leq i < j \leq k$, the join result $B_1 \bowtie B_2 \bowtie ... \bowtie B_k$ equals to $\bigcup_{key} B_1^{key} \times ... \times B_k^{key}$ where $B_1^{key} = \{b : b \in B_1 \wedge b(K) = key\}$ and $\times$ represents the Cartesian product. Therefore, to calculate the join, we only need to calculate the Cartesian product. Figure 2 gives an example to illustrate this calculation.

3 Demonstration Scenario

The goal of the demonstration is to illustrate how to use HadoopSPARQL to execute SPARQL queries. HadoopSPARQL provides a Web interface which can be used to submit queries and view the results. The dataset and queries will be described in Sect. 3.1 while Sect. 3.2 will describe the functionalities of Web interface.

3.1 Dataset and Queries

The demonstration uses a synthesis dataset called SP^{2}Bench [6] which is designed to test the performance of a SPARQL query engine. SP^{2}Bench generates data

Fig. 3. HadoopSPARQL Query UI

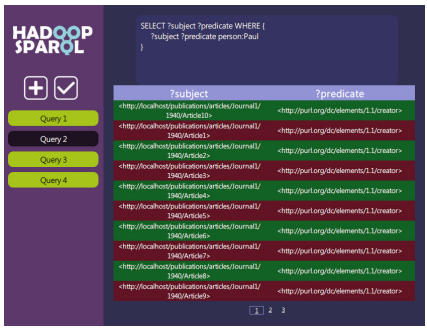

Fig. 4. HadoopSPARQL Result UI

based on DBLP database which includes authors, articles, conferences, journals and so on. One can use SP^{2}Bench to generate the dataset by specifying the data volume. SP^{2}Bench provides 12 testing queries which include the two queries in Table 1. Most of the benchmark queries are BGP queries and only two containing union and optional constructions. We use all queries as samples except the two unsupported by our system.

3.2 Web Interface

HadoopSPARQL provides a Web interface for users to submit queries through Web browser. The interface is written in JavaScript. Figure 3 shows the query UI. Users can add a new query by clicking the plus button, and remove a query by clicking the minus button. All queries are listed in the left of the screen. A submit button with tick symbol is used to submit these queries to the HadoopSPARQL system for evaluation. On the side bar, several sample queries from SP^{2}Bench are listed. Users can click on each of them to add it to their query set. Once the queries are submitted, the system will provide a link which can be used to view the results.

Figure 4 shows the result page. Queries submitted by the user are listed in the side bar. By clicking one of the queries, the content of the selected query is showed in the top while the results are listed below. The number of results usually exceeds one page's limitation, so there is a list of buttons allowing the user to view the whole result set.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of OSDI 2004, pp. 137–147 (2004)
2. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. In: Proceedings of VLDB 2011, pp. 1123–1134 (2011)
3. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for SPARQL. In: Proceedings of ICDE 2012 (2012)

4. LOD. http://linkeddata.org/
5. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
6. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: a SPARQL performance benchmark. CoRR (2008)
7. SPARQL. http://www.w3.org/TR/rdf-sparql-query/