Efficient Social Network Data Query Processing on MapReduce

Liu Liu
UMass Amherst
liuliu@ecs.umass.edu

Jiangtao Yin
UMass Amherst
jyin@ecs.umass.edu

Lixin Gao
UMass Amherst
lgao@ecs.umass.edu

ABSTRACT

Social network data analysis becomes increasingly important for business intelligence and online social services. Lots of social network data is presented by Resource Description Framework (RDF). Accordingly, *SPARQL*, an RDF query language, becomes popular for social network data analysis. As the sizes of social networks expand rapidly, a SPARQL query usually involves a large quantity of data, and thus parallelizing its execution is desirable. MapReduce is a well-known and popular big data analysis tool. However, the state-of-the-art translation from SPARQL queries to MapReduce jobs is not efficient because it mainly follows a two layer rule which needs to transform the SPARQL triple pattern to the standard SQL join. In this paper, we propose two primitives to enable efficient translation from SPARQL queries to MapReduce jobs. We use multiple-join-with-filter to substitute traditional SQL multiple join when feasible, and merge different stages in the query workflow. The evaluation on social network data benchmarks shows that the translation based on these two primitives can achieve up to 2x speedup in query running time comparing to the traditional two layer scheme.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Design, Experimentation, Performance

Keywords

MapReduce; RDF; SPARQL; query processing

1. INTRODUCTION

As social networks become more popular, social network data analysis is increasingly important. In order to improve the integration and reuse of the data, many social network providers start to use the semantic web data model, Resource Description Framework (RDF), to present the data. Many organizations have published social network data in the RDF format. The Friend of a Friend (FOAF) [6] project aims to create a Web of machine-readable pages for describing people, where the links represent relationships between people and the things they do. Facebook's Open Graph [4] allows applications to connect with real world actions and objects. Freebase [3] is an online collection of structured data across different domains such as music, movie, people, and so on. DBpedia [2] extracts structured content from Wikipedia and makes it available on the Web, so as to allow users to find the information spread across different Wikipedia articles.

When being represented in the RDF format, social network data consists of a set of triples. Figure 1 shows a few triples which are used to describe the city of Berlin in DBpedia. Publishing the social network data in the RDF format facilitates meaningful and easy-structured queries. For example, DBpedia makes it possible to ask a query like "list all computer scientists who live in Berlin". These queries are usually written in the W3C-endorsed SPARQL language [18].

```
Berlin  dbpedia-owl:country     dbpedia:Germany
Berlin  dbpedia-owl:leader      dbpedia:Klaus_Wowereit
Berlin  dbpedia-owl:populationTotal  3499879
```

Figure 1: Sample triples from DBpedia.

The size of social network data is huge and is increasing rapidly. For example, DBpedia has 1 billion triples in Version 3.7 but 1.89 billion triples in Version 3.8. Answering a SPARQL query from billions of triples in reasonable time is challenging. Distributed frameworks such as MapReduce [7] allow parallel processing on massive data using a large cluster of commodity machines. MapReduce provides a simple and flexible programming model and hides the distributed execution of computation from users. Therefore, it is a promising tool for processing SPARQL queries on social network data. However, existing methods of translating SPARQL queries into MapReduce workflow are not efficient. They rely on translating the SPARQL query into a series of SQL joins first, and then map the SQL join flow directly to MapReduce jobs. This two layer mapping is actually unnecessary and inefficient.

In this paper, we propose two primitives to enable efficient translation from SPARQL queries to MapReduce jobs. Our first primitive is multiple-join-with-filter, which can substitute SQL multiple join when feasible. The second primitive

HotPlanet'13, August 16, 2013, Hong Kong, China.

is to merge the *SELECTION* stage with the *JOIN* stage, which can make better use of the flexibility of the MapReduce model. By employing these two primitives on SPARQL queries, we can achieve better query performance with less jobs and intermediate tables.

The prototype of our primitives is implemented on Hadoop [1]. We evaluated various queries on two social network benchmarks with different data sizes. The results show that our primitives can achieve up to 2x speedup compared to the traditional two layer mapping method.

2. TRANSLATION FROM SPARQL TO MAPREDUCE

In this section, we first discuss the existing methodology used to translate a SPARQL query to MapReduce workflow based on SQL join operations. Then we illustrate the limitations of the current solutions.

2.1 Current Translation Methods

A SPARQL query consists of elementary search conditions (i.e., triple patterns or triplets). Each pattern is a triplet where one or more of its components are variables. SPARQL query processing is executed mainly using relational operators such as join, since search conditions in the query usually share some unknown variables. A series of join operations can be used iteratively on different shared variables to select the intersection from variable sets represented by triple patterns. In order to obtain operands for the join operations, the most common way is to parse the RDF data once and generate base tables, each corresponding to one triple pattern in the SPARQL query. This logic stage, in which base tables are generated as the operands for the later *JOIN* stage, is commonly known as *SELECTION*.

```
(?User  hasInterest  "Keepon Pro")     //Triple Pattern ID = 1
(?Friend  locatedAt  "Hong Kong")      //2
(?Photo  laterThan  "1/1/2013")        //3
(?User  friendOf  ?Friend)             //4
(?Photo  taggedBy  ?User)              //5
(?Photo  taggedBy  ?Friend)            //6
```

Figure 2: Targeted advertisement.

The SELECTION stage scans the data and emits the qualified shared variables to form the base tables. Each base table corresponds to one triple pattern. Take the query shown in Figure 2 as an example. This query can be used in online social networks such as Facebook for targeted advertisement. It aims to find users who are from a specific area and are close friends to the users who are interested in a certain product. If both the user and his/her friend are tagged in the same photos recently posted, they are considered as close friends. The SELECTION stage generates 6 base tables which correspond to hasInterest(*?User*), locatedAt(*?Friend*), laterThan(*?Photo*), friendOf(*?User?Friend*), taggedBy(*?User?Photo*) and taggedBy(*?Friend?Photo*). These base tables are then stored as the intermediate files, and will be joined in the following JOIN stage. Current methodologies directly map this logic SELECTION to a MapReduce job. The map phase output key is the triple pattern ID, and the output value is the matched shared variables. The reduce phase collects shared variables with the same triple pattern ID to form base tables and writes them into a file system.

Although all the current methods share the same SELECTION stage, the later JOIN stage varies. The naive pairwise join method [13] joins the base tables in a pairwise manner and translates each logic join into one MapReduce job. This method does not fully exploit the feature of multiple shared variables, which can be joined using multiple join concepts in SQL. The later works [15, 19] adopt the multiple join mechanism. Especially, [19] adopts multiple join as the first join operation when possible, but due to the rules of the SQL join, later joins are all pairwise. This kind of translation represents the typical two layer mapping method. The MapReduce join workflow of the targeted advertisement query using this method is illustrated in Figure 3, which needs 4 jobs in total.

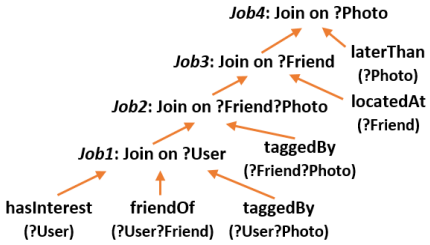


Figure 3: Two layer mapping.

2.2 Limitations of Current Methods

Current two layer mapping is not efficient since it results in unnecessary MapReduce jobs. This is mainly because the traditional SQL join operator's semantic does not fit into the join use cases in SPARQL query processing well. The number of base tables covered in each join is therefore limited, which leads to more MapReduce jobs.

Besides the JOIN stage translation, the SELECTION stage translation is also not efficient. Current methods directly map the SELECTION into one MapReduce job, and use the generated base tables as the later stage's input. In fact, it is more efficient to take advantage of the flexible MapReduce model to merge the SELECTION into JOIN so as to eliminate an individual job for the SELECTION.

3. TWO MAPREDUCE PRIMITIVES FOR SPARQL QUERIES

In this section, we describe our mechanism to optimize the MapReduce processing for SPARQL queries. It mainly consists of two primitives: (1) using multiple-join-with-filter to substitute SQL join when feasible; (2) merging the SELECTION stage's job into the JOIN stage's job by adopting a Select-Join job.

3.1 Multiple-join-with-filter Primitive

We have observed the different join use cases between SPARQL and SQL query processing. In SQL, each table represents one type of real-world objects. Schema is well designed for each table to minimize the data redundancy. In contrast, base tables are generated by triple patterns in SPARQL, which may lead to duplicated fields among different base tables caused by different predicates between the same subject and object. Take one simple example: two people *?A* and *?B* represented by subject and object within triplets may have different relationships represented by predicates such as relative, friend, colleague and so on. Therefore, finding pairs of people with these relationships leads to

base tables corresponding to different relationships but all having the same fields *?A?B*.

The join operations can be performed efficiently using the MapReduce programming model. Suppose we have a SPARQL query with 4 base tables: *?X?Y*, *?X?Y*, *?X?Z* and *?X?Z*, where the duplicated fields are caused by different predicates as illustrated above. Current methods based on SQL will join *?X?Y* and *?X?Z* separately first, and then join their results on field *?X*. Each join operation corresponds to one MapReduce job. In fact, there exists a more efficient operation under this circumstance which better leverages the flexibility of the MapReduce programming model. We can join all base tables on *?X*, and filter out entries with different values of duplicated *?Y* and *?Z* fields simultaneously, which needs only one job. We name this kind of operation *multiple-join-with-filter*. By applying this primitive, we can construct query workflow with less MapReduce jobs.

3.1.1 Semantics of Multiple-join-with-filter

Different from the SQL multiple join, which has only one join key, the multiple-join-with-filter primitive allows to define another "filter key". The filter key is defined on fields (other than the join key) among the subset of the tables participating in the multiple join. It is responsible for further filtering the entries calculated by traditional multiple join. Note that the filter key can be empty. In that case, we just use the SQL multiple join. This operation's syntax is described as follows:

Multiple Join [tables] *on* [Join Key]
Filter *on* [Filter Key]
Dump Result *to* [MT]

Consider the example appeared in the previous targeted advertisement query (Figure 2). After joining on *?User*, we need to join *?User?Friend?Photo*, *?Friend* and *?Friend?Photo* on *?Friend*. Multiple-join-with-filter can be adopted in the following way:

Multiple Join [*?Friend*, *?Friend?Photo*, *?User?Friend?Photo*] *on* [*?Friend*]
Filter *on* [*?Photo*]
Dump Result *to* [*?User?Photo*]

In this example, multiple-join-with-filter uses *?Friend* as the join key. At the same time, all the resulted entries with different values of *?Photo* from *?User?Friend?Photo* and *?Friend?Photo* are filtered out. Multiple-join-with-filter uses *?Photo* as the filter key. Similar with the SQL multiple join, multiple-join-with-filter can also be implemented using one MapReduce job. The new query execution plan for targeted advertising is shown in Figure 4. Compared with the two layer mapping, it has less jobs (3 vs. 4). This is because the multiple-join-with-filter job substitutes both joins on *?Friend?Photo* and *?Friend* along the query path with one MapReduce job.

3.1.2 MapReduce Model of Multiple-join-with-filter

When SQL joins are implemented on MapReduce, one SQL join needs one MapReduce job, which is widely used in different database related applications. The most common and flexible join model in MapReduce is called reduce-side join. In the reduce-side join, map phase output key is the join key, and the value consists of other fields in the same entry. The value fields are tagged with the source table ID.

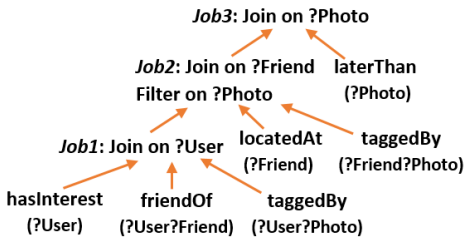


Figure 4: One layer Mapping.

In the reduce phase, all entries with the same join key are aggregated on the same reducer. Then cartesian product will be conducted for each key according to the value's tag, which is the source table ID.

Similar with the traditional SQL multiple join, multiple-join-with-filter can also be implemented within one MapReduce job. The difference is that the filter key should also be tagged in the map phase, so as to allow the reducer to identify it. When performing the cartesian product, the reduce function will identify the filter key through the tag. If all filter keys from different tables are the same, then the current entry is qualified to be concatenated and output to the file system.

3.2 Select-Join Primitive

In the traditional two layer mapping, the SELECTION stage is translated to one independent MapReduce job individually. However, this direct mapping from the SELECTION stage to one job is not efficient. In fact, the flexibility of the MapReduce programming model allows to generate base tables and conduct join simultaneously. As a result, we no longer need an individual job for the SELECTION stage. This new hybrid stage in which the SELECTION stage is integrated into the JOIN stage is referred to as *Select-Join* primitive.

To merge SELECTION into JOIN in one MapReduce job, (key, value) pairs for them should be differentiated separately. For this reason, we built two groups of map and reduce routines in the Select-Join primitive. One group called *Select group* is used to generate base tables. The Select group behaves as the original SELECTION stage. The other group called *Join group* is responsible for the join operation. The map phase will scan the RDF data to match each qualified triplet of which the corresponding base table attends the first join operation. These triplets will then be handled by the Join group. In the Join group, multiple-join-with-filter is used if feasible. The remaining triplets, which are not processed in the first join operation, are written into the file system as base tables by the Select group. The syntax of Select-Join is described as follows:

Select *on* [triplets A]
Multiple Join [triplets B] *on* [Join Key]
Filter *on* [Filter Key]
Dump Result *to* [ST]

In the Select group, we further adopt another optimization which is referred to as *Skip Write*. Compared with the previous work, such as [15], in which base tables have to be shuffled to reduce side, we totally eliminate the shuffle step for base table generation. Instead, all base tables are written directly into the file system from the map side, bypassing the reduce side.

Now we illustrate how Select-Join works by showing two concrete examples, with single and multiple shared variables. The first example represents a SPARQL query with a single shared variable, shown in Figure 5.

```
(?Post  type  Post)                          //ID = 1
(?Post  laterThan  "1/1/2013")               //2
(?Post  tag  ?Tag)                           //3
GROUP BY  ?Tag
ORDER BY DESC  (count (?Post))
```

Figure 5: Finding hot topics.

This query figures out recent hot topics in a social network, such as Facebook. Since it has one shared variable *?Post*, we only need one join on *?Post* to obtain the query result. It can be achieved with one MapReduce job by merging the SELECTION stage into the JOIN stage. All the triple patterns will participate in the Join group. The Select group is not employed in this case, since no base tables need to be generated. Actually, all SPARQL queries with one shared variable can be done within one MapReduce job, which totally eliminates the intermediate data. This kind of SPARQL queries is very common in daily social applications, and is usually referred to as *Star Pattern*.

```
(?User  createdAt  "2009")                  //ID = 1
(?User  creatorOf  ?Status)                  //2
(?Status  laterThan  "1/1/2013")             //3
```

Figure 6: Finding zombie accounts.

Next, we check the SPARQL query shown in Figure 6, which has multiple shared variables. This query finds the zombie accounts defined by the administrator. Here, "status" means any action, such as post, comment, or tag certain accounts have done. Notice here we have more than one shared variables. It cannot be handled using only one MapReduce job. In this example, we assume to join *?User* first. The Select group in the Select-Join will output one base table corresponding to the triple pattern with ID 3. The Join group, on the other hand, joins on *?User* through triple patterns with ID 1 and 2, and then writes the intermediate tables into the file system.

3.3 Workflow Generation

In this subsection, we illustrate our mapping algorithm, which translates SPARQL queries into MapReduce workflow, based on the two proposed primitives.

Our mapping algorithm processes variables in the triplets one by one. Each time, it joins the current shared variable that appears the most frequently. The algorithm launches one SQL join or one multiple-join-with-filter job for this shared variable. The SELECTION stage will always be merged into the first join. To do that, we build two map and reduce routines: one for selection and the other one for the first join. These two routines are integrated to generate the first MapReduce job. Whether the multiple-join-with-filter primitive is feasible depending on whether there is a valid filter key. This is done by counting the appearance of each shared variable except the join key among the participating triplets. If we find any shared variable appearing at more than one triplets being joined, we put them together to form the filter key.

The join result will be further joined with the remained base tables not covered yet following the same rule until all the base tables are processed. If it encounters the case where several shared variables have the same appearance times, the mapping algorithm will break the tie arbitrarily. Because finding optimal join sequence is not our focus in this paper, we leave this problem for further research. Once the final result can be generated, the mapping algorithm returns the final MapReduce workflow. The mapping algorithm is shown in Algorithm 1.

Algorithm 1: $WorkflowGeneration(Q)$

input : A SPARQL query Q (a set of triplets)
output: MapReduce workflow W

```
1  v = mostFrequentVariable(Q); /* join key                    */
2  tp1 = triplets in Q not sharing v; /* for Select            */
3  tp2 = triplets in Q sharing v; /* for Join                  */
4  fk = variables other than v appear at more than one
       triplets in tp2; /* filter key                          */
5  job = "Select on" + tp1 + "Multiple Join" + tp2 + "Join
   on" + v + "Filter on" + fk + "Dump Result to" + ST;
   /* ST is a set of tables                                   */
6  W.append(job);
7  while ST.size() > 1 do
       /* the last table corresponding to the final
          result                                               */
8      VA = correspondingVariables(ST);
9      v = mostFrequentVariable(VA);
10     tb = tables in ST sharing v;
11     fk = variables other than v appear at more than
       one tables in tb;
12     job = "Multiple Join" + tb + "Join on" + v +
       "Filter on" + fk + "Dump Result to" + MT;
13     W.append(job);
14     remove tb from ST;
15     add MT to ST;
16 return W;
```

4. EVALUATION

In this section, we demonstrate the performance of our proposed primitives. Our system is built on top of Hadoop. We evaluate both of the proposed primitives on social network benchmarks.

4.1 Experimental Setup

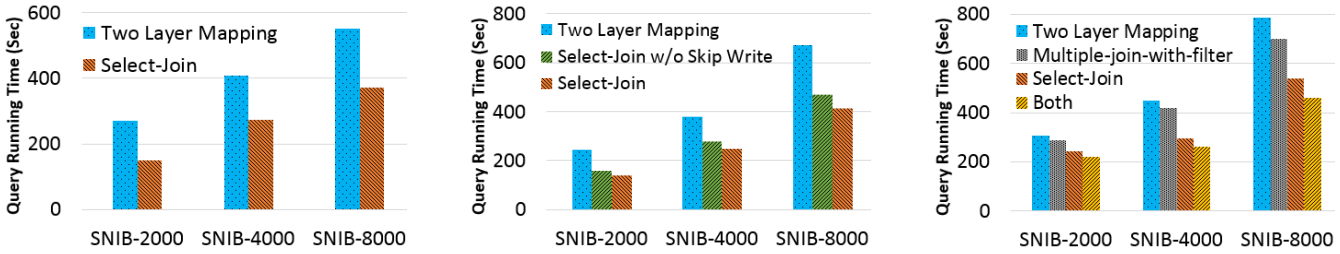
4.1.1 Cluster Setup

We deploy our system on a cluster with 4 physical nodes. Each node is equipped with a Xeon E5607 Quad Core 2.27GHz CPU, 4GB memory and 1Gb network card. All of the nodes are connected to one Gbps Ethernet switch.

The Hadoop framework is configured with 3 replicated blocks and a block size of 64MB. In the original SELECTION job, the number of reducers is set to the number of triple patterns in the query. In all other cases, the number of reducers is set to 8.

4.1.2 Benchmark and Dataset Generation

We adopt two benchmarks: Social Network Intelligence Benchmark (SNIB) [5] and Lehigh University Benchmark (LUBM) [9]. Each of them represents one type of social networks. SNIB simulates a small Facebook/Twitter style online social network. The data is extracted from public data sources such as DBpedia [2] and FOAF [6]. It basically has all core elements in a social network system, such as user,



(a) Find hot topics (b) Find zombie accounts (c) Targeted Advertising

Figure 7: Performance evaluation on SNIB dataset.

(a) LUBM query 4 (b) LUBM query 8 (c) LUBM query 9

Figure 8: Performance evaluation on LUBM dataset.

friendship, post, photo and tag. LUBM represents a network of university systems, in which all the entities in the university, such as student, professor and course are described in a format of triples. It is also attached with 14 SPARQL queries, which are intensively used to test SPARQL query engine performance.

SNIB dataset is generated using S3G2 [17], and 3 online social network datasets with 2000, 4000 and 8000 users are generated. LUBM dataset is generated using *LUBMft* [14]. We use its attached *UBAft* data generator with "-names" flag on to generate 3 datasets with real people names; each consists of 500, 1000 and 1500 universities separately. The *RDF2RDF* tool is then used to convert the OWL files into N-Triples (.nt) RDF serialization. All datasets are summarized in Table 1.

Table 1: Datasets Summary

Name	# of triples	Size
SNIB-2000	99,996,099	4.5GB
SNIB-4000	199,069,668	9.02GB
SNIB-8000	397,631,187	18.11GB
LUBM-500	85,821,591	7.14GB
LUBM-1000	171,782,401	14.32GB
LUBM-1500	257,501,329	21.6GB

4.2 Performance Evaluation

The SPARQL queries actually have some fixed basic graph patterns [11]. In order to show our performance more systematically, we categorize our test queries into 3 classes according to their number of MapReduce jobs in the query workflow. Each class actually corresponds to one common basic graph pattern, namely *star pattern*, *chain pattern* and *triangle pattern*. We deal with them incrementally to show our performance improvement, since each of them benefits differently from our primitives.

4.2.1 *Star Pattern*

All of the SPARQL queries with star pattern can benefit from the Select-Join primitive. Since there is only one shared variable in the SPARQL query, we can finish the query in one job. Figures 7a and 8a show the performance comparison for the previous finding hot topics query (Figure 5) and LUBM query 4.

The Select-Join primitive reduces the MapReduce job from 2 to 1 for the star pattern, and no intermediate data will be generated. We can observe that it outperforms the two layer mapping translation with 1.5x speedup on the finding hot topic query, and that it has even better improvement on LUBM query 4. This is because the selectivity of the original SELECTION stage on LUBM query 4 is relatively low. Therefore, the following join job has more data to read and shuffle. As a consequence, by eliminating intermediate files and the duplicated process, the primitive gains more performance improvement on it than on the hot topic query, which has relatively high selectivity.

4.2.2 *Chain Pattern*

The chain pattern is more complex than the star pattern. It has more than one shared variables, and thus cannot be completed by one MapReduce job. Because the Select-Join primitive needs to write base tables for the later JOIN stage in this case, the chain pattern also benefits from the Skip Write optimization built in the Select group. Figures 7b and 8b show the performance comparison for the finding zombie accounts query (Figure 6) and LUBM query 8.

The Skip Write optimization avoids the unnecessary shuffle when generating base tables. As a result, Select-Join (with Skip Write) costs less amount of time than the version without Skip Write. The time that Skip Write saves is relatively proportional to the size of base tables Select-Join needs to generate.

4.2.3 Triangle Pattern

The triangle pattern is the most complex pattern in our test queries. It benefits from both the multiple-join-with-filter primitive and the Select-Join primitive. We show the performance improvement for different primitives separately in Figures 7c and 8c for the previous targeted advertisement query (Figure 2) and LUBM query 9. Both primitives together achieve 2x speedup comparing to the traditional two layer mapping method.

5. RELATED WORK

Lots of works have been done in translating SQL into MapReduce workflow. Well-known ones include *Pig* [16] and *Hive* [20]. Based on Hive's query translation rule which maps each SQL sub-query to one MapReduce job, [12] introduces a job aggregation mechanism. It merges sub-queries sharing the same keys into one job to reduce the total number of jobs in the query workflow.

However, fewer works [10, 13, 15, 19, 21] focus on building efficient parallel SPARQL query execution engine. Moreover, none of them presents the primitives proposed in this paper. Those works can be categorized into two classes: using existing RDF storage solution, or building the engine from the ground up. In the first class, [15], [19] and [13] are all based on using SQL join to simulate the SPARQL processing. [13] is the earliest work to leverage MapReduce in semantic webs. It shows the feasibility of using the MapReduce model to conduct SPARQL queries. Since it adopts the naive pairwise join, the number of MapReduce jobs is unnecessarily more than needed. In [19], the authors discuss an indexing method using HBase [8] and a query plan determination scheme based on SQL. As the most relevant work, [15] provides a MapReduce mechanism to concurrently join shared variables in a greedy manner. This mechanism can significantly reduce the number of MapReduce jobs, but in fact, it suffers from huge intermediate tables.

In the second class, each of [10] and [21] builds a graph store and the corresponding query engine from the ground up. By making use of their optimized storage pattern, they can adapt the SPARQL query execution to the graph store so as to obtain better performance.

6. CONCLUSION

In this paper, we propose two primitives to optimize SPARQL queries on social network data using MapReduce. They consist of the multiple-join-with-filter primitive, which can replace the traditional SQL multiple join when feasible, and the Select-Join primitive. Based on these two primitives, we further present a query translation algorithm to translate SPARQL queries into MapReduce jobs. By adopting our primitives, we can efficiently reduce the number of MapReduce jobs and the amount of intermediate files. The evaluation results show that our primitives can achieve up to 2x speedup compared to the traditional two layer mapping method. In the future work, we plan to investigate the join order within our workflow translation to explore the feasibility of even better query performance.

Acknowledgments

The authors are grateful to the anonymous reviewers for their comments and suggestions. This work is partially supported by NSF grants CCF-1018114 and CNS-1217284.

7. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org/`.
[2] DBpedia. `http://dbpedia.org`.
[3] Freebase. `http://www.freebase.com/`.
[4] Open Graph. `https://developers.facebook.com/docs/opengraph/`.
[5] Social Network Intelligence Benchmark. `http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark`.
[6] The Friend of a Friend (FOAF) project. `http://www.foaf-project.org/`.
[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[8] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.
[9] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.
[10] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQLquerying of large RDF graphs. *VLDB Endowment*, 4(11), 2011.
[11] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE '12*, pages 666–677. IEEE, 2012.
[12] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *ICDCS '11*, pages 25–36. IEEE, 2011.
[13] P. Mika and G. Tummarello. Web semantics in the clouds. *Intelligent Systems, IEEE*, 23(5):82–87, 2008.
[14] E. Minack, W. Siberski, and W. Nejdl. Benchmarking fulltext search performance of RDF stores. *The Semantic Web: Research and Applications*, pages 81–95, 2009.
[15] J. Myung, J. Yeon, and S.-g. Lee. SPARQL basic graph pattern processing with iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, page 6. ACM, 2010.
[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110. ACM, 2008.
[17] M.-D. Pham, P. Boncz, and O. Erling. S3G2: A Scalable Structure-Correlated Social Graph Generator. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2013.
[18] E. Prud' Hommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.
[19] J. Sun and Q. Jin. Scalable RDF store based on HBase and MapReduce. In *ICACTE '10*, volume 1. IEEE, 2010.
[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB Endowment*, 2(2):1626–1629, 2009.
[21] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *VLDB Endowment*, 6(4), 2013.