Efficient Semantic Search over Structured Web Data: A GPU Approach

Ha-Nguyen Tran, Erik Cambria and Hoang Giang Do

School of Computer Science and Engineering
Nanyang Technological University
{hntran,cambria,do0008ng}@ntu.edu.sg

Abstract. Semantic search is an advanced topic in information retrieval which has attracted increasing attention in recent years. The growing availability of structured semantic data offers opportunities for semantic search engines, which can support more expressive queries able to address complex information needs. However, due to the fact that many new concepts (mined from the Web or learned through crowd-sourcing) are continuously integrated into knowledge bases, those search engines face the challenging performance issue of scalability. In this paper, we present a parallel method, termed *gSparql*, which utilizes the massive computation power of general-purpose GPUs to accelerate the performance of query processing and inference. Our method is based on the backward-chaining approach which makes inferences at query time. Experimental results show that gSparql outperforms the state-of-the-art algorithm and efficiently answers structured queries on large datasets.

1 Introduction

In recent years, the Resource Description Framework (RDF) and the Web Ontology Language (OWL) are widely applied to model knowledge bases such as DBPedia [2], Yago [17], or SenticNet [6]. Searching and retrieving the complete set of information on such knowledge bases is a complicated and time-consuming task. The crucial issue is that the searching process requires a deep understanding of the semantic relations between concepts. In other words, semantic search engines generally need to integrate an inference layer which derives implicit relations from the explicit ones based on a set of rules.

With the immense volume of daily crawled data from the Internet sources, the sizes of many knowledge bases have exceeded millions of concepts and relations [7]. Real-time inference on such huge datasets with various user-defined rulesets is a non-trivial task which faces challenging issues in term of system performance. As a consequence, efficiently searching and retrieving information on large-scale semantic systems have attracted increasing interest from researchers recently. Query engines such as OWLIM [3], Sesame [4], and Jena [8] integrate an inference layer on top of the query layer to perform the reasoning process and retrieve the complete set of results. This approach is termed backward-chaining reasoning.

Those methods are also designed to support in-memory execution. Most of them, however, are facing the problems of scalability and execution time. As a result, it has until now been limited to either small datasets or weak logics. Distributed computing methods [21] have been introduced to deal with large graphs by utilizing parallelism, yet there remains the open problem of high communication costs between the participating machines.

Recently, Graphic Processing Units (GPUs) with massively parallel processing architectures have been successfully leveraged for query processing [14, 19, 20, 18]. GPUs have also been utilized to enhance the performance of forward-chaining reasoning [12, 15] which makes explicit all implicit facts in the pre-processing phase. The benefits of the forward-chaining inference scheme are 1) the time-consuming materialization is an off-line computation; 2) the inferred facts can be consumed as explicit ones without integrating the inference engine with the runtime query engine. However, the drawback of this approach is that we can only reason and query on the knowledge bases with a pre-processed ruleset. In addition, the amount of inferred facts could be very large in comparison with the original dataset.

To address the requirements of scalability and execution time for semantic search engines over large-scale knowledge bases with custom rules, in this paper we introduce a parallel method, termed *gSparql*, which utilizes the massive computation power of general-purpose GPUs. Our method accepts different rulesets and executes the reasoning process at query time when the inferred triples are determined by the set of triple patterns defined in the query. To answer SPARQL queries in parallel, we convert the execution plan into a series of primitives such as sort, merge, prefix scan, and compaction which can be efficiently done on GPU devices. We also present optimization techniques to improve the performance of reasoning and query processing. To highlight the efficiency of our solution, we perform an extensive evaluation of gSparql against the state-of-the-art semantic query engine Jena. Experiment results on LUBM [10] show that our solution outperforms the existing method on large datasets.

The rest of the paper is structured as follows: Section 2 provide formal definitions of our problem; Section 3 introduces the GPU-based approach to accelerate semantic search engines; Section 4 presents the GPU implementation of the inference engine; experiment results are shown in Section 5; finally, Section 6 concludes the paper.

2 Semantic Search on Web Data

RDF[1] is a W3C recommendation that is used for representing information about Web resources. Resources can be anything, including documents, people, physical objects, and abstract concepts. The RDF data model enables the encoding, exchange, and reuse of structured data. It also provides the means for publishing both human-readable and machine-processable vocabularies.

[1] https://www.w3.org/TR/rdf-primer/

RDF data is represented as a set of triples $< S, P, O >$, as in Table 1, where each triple $< s, p, o >$ consists of three components, namely *subject*, *predicate*, and *object*. Each component of the RDF triple can be represented in either as a universal resource identifier (URI) or in literal form.

Subject (s)	**Predicate (p)**	**Object (o)**
x:Alice	y:isSisterOf	x:Andy
x:Bob	y:isSiblingOf	x:Andy
x:Bob	y:hasParent	x:Brad
x:Alice	y:liveIn	x:London
x:Bob	y:liveIn	x:Paris
x:Andy	y:liveIn	x:England
x:Alice	rdf:type	x:Female
x:Alice	y:age	23
x:Andy	y:age	17
x:Bob	y:age	25
x:London	y:isPartOf	x:England
y:hasParent	rdfs:domain	x:Person
x:Female	rdfs:subClassOf	x:Person
x:isSiblingOf	rdf:type	owl:SymProp

Fig. 1. RDF triples

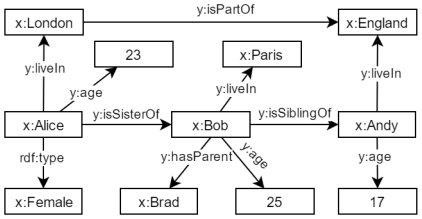

Fig. 2. RDF knowledge graph

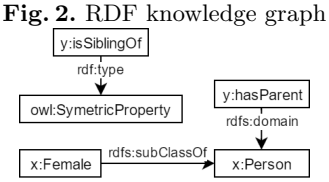

Fig. 3. RDF schema graph

RDF data is also represented as a directed labeled graph. The nodes of such a graph represent the subjects and objects, while the labeled edges are the predicates. Such a representation is believed to be cognitive inspired and it is adopted in many areas of artificial intelligence [16, 5, 23]. We give the formal definition of an RDF graph as follow:

Definition 1. *An RDF graph is a finite set of triples (subject, predicate, object) from the set $T = U \times U \times (U \cup L)$, where U and L are disjoint, U is the set of URIs, and L the set of literals.*

For example, Figure 2 illustrates the RDF graph based on the RDF triples in Figure 1. RDF graphs are further classified into two sub-types, namely *RDF knowedge graph* and *RDF schema graph*. The set of nodes in an RDF knowledge graph includes entities, concepts, and literals, as can be seen in Figure 2. On the other hand, the RDF schema graph describes the relationships between types/predicates. Each edge connects two types or predicates (Figure 3).

Similar to a RDF graph, a SPARQL query[2] also contains a set of triple patterns. The subject, predicate and object of a triple pattern, however, could be a variable, whose bindings are to be found in the RDF data.

Definition 2. *A SPARQL triple pattern is any element of the set $T = (U \cup V) \times (U \cup V) \times (U \cup L \cup V)$, where V is the variable set.*

[2] https://www.w3.org/TR/rdf-sparql-query/

A SPARQL triple pattern can also be recursively defined as follows:

1) If P_1 and P_2 are SPARQL triple patterns, then expressions with the forms of P_1 . P_2, P_1 *OPTIMAL* P_2, and P_1 *UNION* P_2 are also SPARQL triple patterns.

2) If P is a SPARQL triple pattern and C is a supported condition, then P *FILTER* C is also a SPARQL triple pattern.

In a SPARQL query, the *SELECT* keyword is used to identify the variables which appear in the result set. For example, one wants to list all people whose parent is Brad and whose ages are greater than 20. The SPARQL query for this question is illustrated below:

```
SELECT ?a ?b
FROM {
   ?a rdf:type x:Person.
   ?a y:hasParent x:Brad.
   ?a y:age ?b
   FILTER (?b > 20)
}
```

This query returns an empty result set because we cannot find any matches in the RDF data triples in Table 1. However, if we consider the semantic relations by using rules $R = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$, which are given below, the result set of the search query is (?a, ?b) = {(x:Alice, 23), (x:Bob, 25)}.

R_1 : (?x y:isSisterOf ?y) → (?x y:isSiblingOf ?y)

R_2 : (?x y:isSiblingOf ?y) (?y y:isSiblingOf ?z) → (?x y:isSiblingOf ?z)

R_3 : (?x rdf:type ?y) (?y rdfs:subClassOf ?z) → (?x rdf:type ?z)

R_4 : (?x y:isSiblingOf ?y) (?y y:hasParent ?z) → (?x y:hasParent ?z)

R_5 : (?x ?p ?y) (?p rdfs:subPropertyOf ?q) → (?x ?q ?y)

R_6 : (?x ?p ?y) (?p rdf:type owl:SymmetricProperty) → (?y ?p ?x)

R_7 : (?x ?p ?y) (?p rdfs:domain ?z) → (?x rdf:type ?z)

These results can be explained as follows: based on rules R_1, R_2, and R_6, we can infer a triple (x:Alice y:isSiblingOf x:Bob); then, we obtain a triple (x:Alice y:hasParent x:Brad) by applying R_4 to that triple; finally, R_7 generates two other triples relevant to the query, i.e., (x:Alice rdf:type x:Person) and (x:Bob rdf:type x:Person).

3 GPU-accelerated Semantic Search

In this section, we first present our schema to index the RDF data. Then, we introduce an overview of our rule-based reasoning and query processing system, termed *gSparql*. Our method is based on backward-chaining reasoning and is accelerated by the massive parallel computing power of GPUs.

3.1 RDF Index

In gSparql, the triplestore layout is based on the *property tables* approach in which all triples with the same predicate name are stored in the same table [1]. In the traditional method, a property table consists of two columns $< s, o >$ and is sorted by *subject*. In order to support efficient merging and joining operations during reasoning and query processing, we maintain another $< o, s >$ column table, which is sorted by *object*, for each predicate name. Our method uses a dictionary to encode URI and literal RDF terms into numeric values (Figure 4). This encoding is commonly used in large-scale triplestores such as RDF-3X [13] and Hexastore [22] to reduce tuple space and faster comparison. The numeric values are stored in their native formats.

URI/Literal	Numeric ID
y:liveIn	0
y:age	1
rdfs:subClassOf	2
...	...
x:Alice	10
x:Andy	11
x:Bob	12
x:England	13
x:London	14
x:Paris	15
...	...

Fig. 4. URI/Literal dictionary

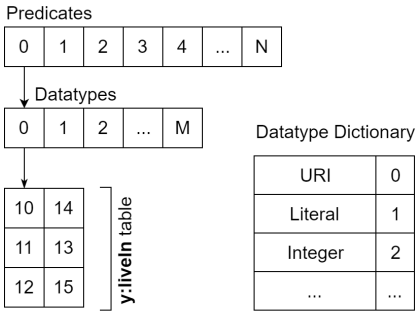

Fig. 5. RDF index

In practice, the objects of the same predicate name might have different datatypes. The objects in the predicate related to *Born* in Wikipedia, for example, consist of Literal values (e.g., "179-176 BC"), Integer values (e.g., 1678), and Datetime values (e.g., 06 June 1986). For each predicate name, we further divide the column tables $< s, o >$ and $< o, p >$ into smaller ones based on the object datatypes. Figure 5 illustrates the triplestore layout used in our method.

The advantages of our RDF data index are: 1) gSparql is able to immediately make comparisons between numeric data in the FILTER clauses without further requesting actual values from the dictionary; 2) Our method can directly return the results of unary functions on RDF terms such as $isIRI$, $isLiteral$, or $isNumeric$; 3) We only need to execute joining operations on columns with the same datatype. Thus, unnecessary joins can be pruned out; 4) The vertical partitioning approach enables GPU kernels to retrieve the table content in a coalesced memory access fashion which significantly improves the performance of GPU-based systems.

3.2 Overview of gSparql

In this section, we present an overview of our GPU-based semantic search engine over structured Web data in RDF/OWL format, termed *gSparql*. Our method

integrates an inference layer to make explicit semantic relations between concepts at query time. As can be seen in Figure 6, the input of our system is a SPARQL query and the output is a result set which contains a collection of relevant tuples in which the selected variables are bound by RDF terms.

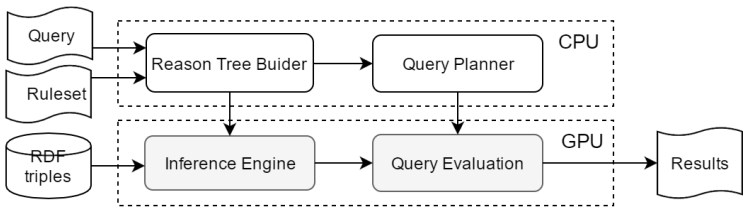

Fig. 6. An overview of gSparql

The engine module first parses the input query Q into a set of single triple patterns. For each triple pattern p, the Reason Tree Builder module generates a reasoning tree which is in fact a rooted DAG (directed acyclic graph) with the root of p based on the predefined ruleset. Figure 7 shows a part of the reasoning tree of the *(?a y:hasParent x:Brad)* triple pattern. A reasoning tree comprises two types of nodes, namely pattern nodes and rule nodes. A pattern node is established by connecting rule nodes using *or* operations. A rule node, in contrast, is created by applying *and* operations between pattern nodes. General speaking, the parent of a rule node R_i is the consequent of the rule and its child nodes are the antecedents of R_i. The module builds those trees by recursively applying rules to pattern nodes in DFS fashions. The reasoning tree construction terminates when no more rules can be further applied. The Inference engine, then, searches for all RDF triples which can match p using the reasoning tree. The matching process takes the majority of the querying time due to time-consuming operations such as joining, filtering, and duplication elimination. Thus, our method takes advantage of massively parallel computing ability of GPUs to accelerate such operations. In particular, we employ an efficient parallel scheme which combines GPU-friendly primitives such as sort, merge, prefix scan.

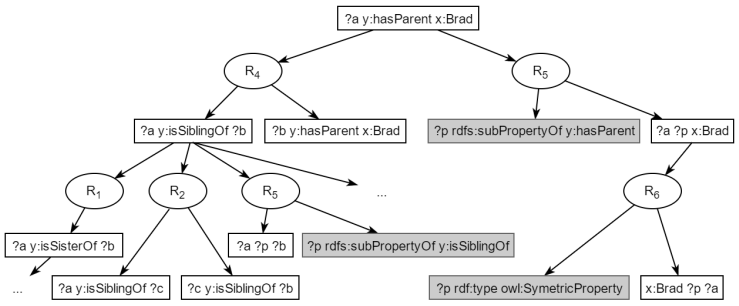

Fig. 7. Reasoning tree of (?a y:hasParent x:Brad)

The Query Evaluation module then executes joining the resulting triples of all single triple patterns in the search query. Similar to the Inference Engine, the matching process takes the majority of the querying time and thus is performed on the GPU. A data buffer in the device memory is utilized to temporarily maintain the required data and intermediate results during execution. After performing the query evaluation phase, the final results are transferred back to the main memory.

4 Inference Engine

In comparison to materialization-based method, the reasoning and query processing system based on backward-chaining is usually required to perform more computation at the query time. The real-time inference is considered as the bottleneck of this approach which decreases the overall response time. Thus, the objective of gSparql is to enhance the performance of the backward-chaining reasoning process.

4.1 Optimized Backward-Chaining

The main routine of Inference Engine execution path is illustrated in Algorithm 1. The input of the algorithm is a single triple pattern p in the SPARQL query. To find the results of p, we first identify the set of rules R_p which can be applied to generate p (Line 2). For each rule $r \in R_p$, we recursively find the matches of triple patterns in the left hand side (LHS) of r, then apply the rule to generate the matches of the right hand side (RHS) pattern. The results of all rules are then merged to obtain the inferred triples of p (Lines 3-14). The inferred results are finally combined with the matches of p in the triplestore to produce the final solutions (Lines 16-17).

The real-time backward-chaining inference is executed by matching and joining RDF triples based on the reasoning trees using bottom-up approaches. To match a rule node, we first search for the matches of its child nodes in the RDF triplestore. Then, joining operations are applied to return the rule node's results. This is the common procedure to make inferences on *general rules*. The matches of a pattern node are obtained by merging the results of its child rule nodes. In rules R_2 and R_4 of the example ruleset, the triple patterns related to column tables *y:isSiblingOf* and *y:hasParent* appear on both sides of the rules respectively. In these cases, new triples are potentially generated when we continuously apply the same rules to the derived triples. We call such rules *recursive rules*.

Optimizations: In order to reduce the number of rule nodes in the reasoning tree and enhance the performance of real-time inference, we apply the following optimization techniques:

- *Pre-compute the RDF schema graph:* Based on the observation that the triple pattern related to the RDF schema graph such as *(?a rdfs:subPropertyOf y:hasParent)* is more generic than *(?a ?p x:Brad)*, since the latter pattern depends on the input query while the former does not. Such schema triple

Algorithm 1: bwc-reasoning: Backward-Chaining reasoning

Input: triple pattern p, ruleset R, triplestore D
Output: set of triples T

```
1  R_p := find_rule(p, R);
2  T := {};
3  foreach rule r ∈ R_p do
4      LP := get_lhs(r, p);
5      LP_Result := {};
6      foreach pattern i ∈ LP do
7          if is_computed(i) then
8              if can_terminate(i) then
9                  break;
10             else
11                 LP_Result[i] = get_result(i);
12         else
13             LP_Result[i] := bwc-reasoning(i, R, D);
14     T_r := make_inference(r, LP_Result);
15     T := merge(T, T_r);
16 T_p = match_pattern(p, D);
17 T := merge(T, T_p);
18 return T
```

patterns, however, appear frequently in the reasoning trees. In order to reduce the execution time spending on searching the matches of those schema triple patterns, we perform the materialization process on the RDF schema graph in the pre-processing step [21]. As a result, our system only needs to perform reasoning on the non-schema triple patterns (Line 13).

- *Memorize intermediate results:* An optimization technique to reduce the processing time is memorizing the reasoning results of triple patterns which appear many times in all reasoning trees. In other words, we only perform the backward-chaining reasoning on the subtree of such triple patterns once, then we maintain results for the next calls. The two techniques are applied to the *is_compute* function at Line 7.
- *Prune out unnecessary tree branches:* The pre-calculation of the triple patterns allows us to prune out the reasoning branches which cannot contribute to the final solutions. For example, assume that we have $B \cap C \rightarrow A$ and $D \cap E \rightarrow C$. This implies that $B \cap (D \cap E) \rightarrow A$, or $(B \cap D) \cap E \rightarrow C$ in a different expression. If $B \cap D = \emptyset$, the conclusion A cannot be derived from D and E. By applying this property, we can avoid the unnecessary expensive join operation between D and E [21]. In the reasoning tree described in Figure 7, we can consider the triple pattern *(?p rdf:type owl:SymmetricProperty)* as D and the triple pattern *(?p rdfs:subPropertyOf y:hasParent)* as D. In this case, Rule R_6 will fire only if some of the subjects of the triples that are matched to

(?p rdfs:subPropertyOf y:hasParent) is also the subject of triples part of *(?p rdf:type owl:SymmetricProperty)*.Since *(?p rdfs:subPropertyOf y:hasParent)* and *(?p rdf:type owl:SymmetricProperty)* are schema triple patterns which are computed in the pre-processing step, we are able to check the condition $B \cap D = \emptyset$ very fast and efficiently using the GPU.

- *Remove redundant triple patterns:* This technique is based on the consistency property of a semantic knowledge base and the search query. The triple pattern *(?a rdf:type x:Person)* can be derived from R_7: *(?a ?p ?b) (?p rdfs:domain x:Person)* $\rightarrow$ *(?a rdf:type x:Person)*. In the sample SPARQL query, we need to perform the joining operation between the results of *(?a y:hasParent x:Brad)* and *(?a rdf:type x:Person)*. Due to the pre-computation of the schema triple *(?p rdfs:domain x:Person)*, we can understand that the results contain *(y:hasParent rdfs:domain x:Person)*. Therefore, the result set of *(?a rdf:type x:Person)* is the superset of the matched solution of *(?a y:hasParent x:Brad)*. As a consequence, the joining operation between the result sets of the two single triple patterns is redundant. Thus, the time-consuming reasoning process for the triple pattern *(?a rdf:type x:Person)* can be ignored.

4.2 GPU Implementation

This subsection discusses about the GPU implementation of the inference engine. Initially, we give brief descriptions of some important GPU primitives which significantly outperform the CPU-based counterparts. Then, we discuss how to map these primitives to different groups of inference rules.

Prefix scan: A prefix scan (in short, *scan*) employs a binary operator to the input array of size N and generates an output of the same size. An important example of prefix scan is prefix sum which is commonly used in database operations. In gSparql, we apply the GPU implementation from [11].

Sort: Our system employs Bitonic Sort algorithm for sorting relations in parallel. The bitonic sort merges bitonic sequences, which are in monotonic ascending or descending orders, in multiple stages. We adapt the standard bitonic sort algorithm provided by NVIDIA library.

Merge: Merging two sorted triples is a useful primitive and is a basic building block for many applications. To perform the operation on GPUs, we apply an efficient GPU Merge Path algorithm [9].

Sort-Merge Join: Following the same procedure as the traditional sort-merge joins, we execute sorting algorithms on two relations and after that merge the two sorted relations. Due to the fact that our triplestore layout is based on vertical partitioning approach, the sort-merge join is well-suited for reasoning and query processing execution.

Next, we discuss how to apply these operations on various groups of rules. Our considered inference rules can be divided into some major groups, namely copy rules, subject/object join rules, and predicate join rules.

Copy rules: For this group of rules, the joining operations are not required. We simply copy the whole column table into a new one. The rule R_1 is an example of the rule group. At implementation level, we do not perform actual copy operations.

Subject/Object join rules: Performing this rule group requires joining two predicate tables in the positions of subject or object (e.g, rules R_2, R_3, R_4). Since our triplestore maintains both sorted predicate tables $< s, o >$ and $< o, s >$, these join rules are straightforwardly executed by the standard sort-merge join.

Predicate join rules: This kind of rules joins two triple patterns in which one join attribute is located at the predicate position of a triple pattern and the other attribute is in the subject or object position of the remain triple pattern (e.g, rules R_5, R_6, R_7). We reduce the joining operation of the rules to *scan* and *merge*. First, we scan the triplestore to collect the predicate names of the join attribute in the first triple pattern. Then, we merge column tables of the obtained predicate names.

Recursive Rules: The main routine of making inferences on recursive rules is illustrated in Algorithm 2. The general idea of the algorithm is to recursively apply the rule R to derived triples until no new triple is found.

Algorithm 2: Reasoning procedure for recursive rules

Input: set of triples T, rule R
Output: set of triples T

```
1 NewT := T;
2 while NewT not empty do
3 |   InferT := apply_rule(NewT, R);
4 |   NewT := T \ InferT;
5 |   T := T ∪ NewT;
6 return T
```

The algorithm often generates a large number of duplicated triples in each iteration. The first type of data duplication is witnessed in the inferred triples (i.e., *InferT* set) which are produced by applying rules on the input triple sets (Line 3). To remove such duplicated triples, we implement the sort-based approach [12] on the GPU. First, the derived triples are sorted by using the GPU Sort primitive. Duplicated triples are then pruned out by applying the compaction algorithm. In this method, we identify the valid triples which are different from their next ones. We then employ the Prefix Scan operation to calculate the output locations of these triples. Finally, we write the output triples to the *InferT* array in parallel. The second type of triple duplication is observed when the derived triples in *InferT* have already existed in T (Lines 4-5). Since *InferT* and T are two sorted triple sets, we resolve the duplication problem by modifying the GPU-based Merge operation. In this modification, we detect the new triples and maintain them in the *NewT* array. After that, we merge the remaining triples in *InferT* to the T set.

The approach, however, needs to request all existing triples of the related column table to identify triple duplications and new derived triples from the inferred set. Unlike in-memory systems which are able to maintain all data in the main memory, the storage capability of a typical GPU device is very limited. For the property table whose size cannot fit into the global memory, we must frequently transfer data between GPU and CPU memory during execution. This might become the bottleneck which significantly reduces the overall reasoning performance. To achieve the high performance in such cases, we implement a GPU-based Bloom Filter algorithm to resolve the problem of triple duplication.

5 Performance Evaluation

We evaluate the performance of gSparql in comparison with the state-of-the-art reasoning and query answering system based on backward-chaining inference, named Jena [8]. We perform the experiments using a set of 14 queries taken from LUBM [10] (as shown in the Appendix section). These queries involve properties associated with the LUBM university-world ontology, with none of the custom properties/rules whose support is actually our end goal.

The runtime of the CPU-based algorithms is measured using an Intel Xeon E5-1650 v2 3.50GHz CPU with 16GB of memory. Our GPU algorithms are tested using the CUDA Toolkit 6.0 running on NVIDIA Tesla K20c GPU with 5 GB global memory and 48 KB shared memory per Stream Multiprocessor. For each of those tests, we execute 100 different queries and record the average elapsed time. Figure 8 and Table 5 compare our gSparql system with the state-of-the-art Jena using backward-chaining reasoner. We perform our experiments on two different LUBM settings. The first dataset consists of 1.3 million triples generated from 10 universities (Figure 8). The second one has 13.5 million triples (i.e., 100 universities) (Table 5). The whole datasets are maintained in the main memory. As for the ruleset, we utilize the standard RDFS ruleset which is supported by Jena.

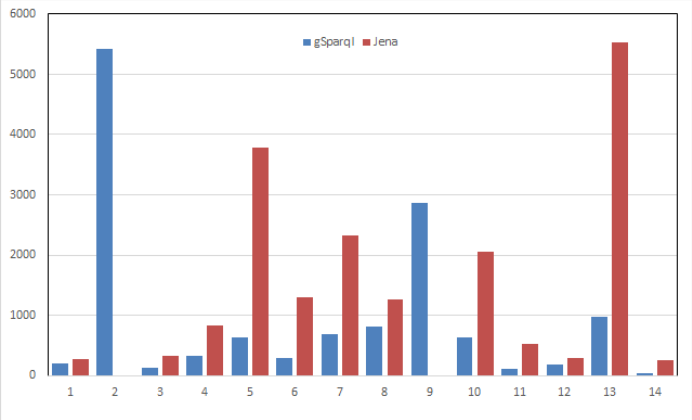

Fig. 8. Comparison with Jena on LUBM10

	gSparql (ms)	Jena (ms)
Q1	245	285
Q2	26657	>15m
Q3	178	336
Q4	1765	5454
Q5	3315	50571
Q6	679	9691
Q7	4572	21712
Q8	5830	9533
Q9	17945	>15m
Q10	2144	19735
Q11	154	1360
Q12	211	294
Q13	4899	67630
Q14	54	267

Table 1. Comparison with Jena on LUBM100

As can be seen in the Figure 8 and Table 5, the response time of gSparql is faster than that of Jena up to several orders of magnitude. When the size of the LUBM dataset increases, the response time of the Jena system rises significantly. In contrast, the processing time of our method increases at a much slower rate. In our gSparql system, we take advantage of a large number of parallel threads, can efficiently handle operations such as joining, merging and sorting in large-scale, thus its performance remains stable. We take the Query 2 and 9 for examples, the Jena system cannot handle a large amount of data in the joining operations while our system still answers those questions efficiently. As can be seen in Query 13, by taking advantage of the fourth optimization technique, we can ignore the reasoning process on the triple pattern *(?X rdf:type ub:Person)*. As a result, gSparql can decrease the overall processing time.

6 Conclusion

In this paper, we introduce gSparql, a fast and scalable inference and querying method on mass-storage RDF data with custom rules to retrieve information on semantic knowledge bases. Our method focuses on dealing with backward-chaining reasoning, which makes inferences at query time when the inferred triples are determined by the set of triple patterns defined in the query.

To efficiently answer SPARQL queries in parallel, we first build reasoning trees for all triple patterns in the query and then execute those trees on GPUs in a bottom-up fashion. In particular, we convert the execution tree into a series of primitives such as sort, merge, prefix scan, and compaction which can be efficiently done on GPU devices. We also utilize a GPU-based Bloom Filter method and sorting algorithms to overcome the triple duplication. Extensive experimental evaluations show that our implementation scales in a linear way and outperforms current optimized CPU-based competitors.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. *Proceedings of the VLDB Endowment*, 1(1):411–422, 2007.
2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.
3. B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.
4. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.
5. E. Cambria and A. Hussain. *Sentic Computing: A Common-Sense-Based Framework for Concept-Level Sentiment Analysis*. Springer, Cham, Switzerland, 2015.
6. E. Cambria, S. Poria, R. Bajpai, and B. Schuller. SenticNet 4: A semantic resource for sentiment analysis based on conceptual primitives. In *the 26th International Conference on Computational Linguistics*, pages 2666–2677, 2016.
7. E. Cambria, H. Wang, and B. White. Guest editorial: Big social data analysis. *Knowledge-Based Systems*, 69:1–2, 2014.
8. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *World Wide Web conference*, pages 74–83. ACM, 2004.
9. O. Green, R. McColl, and D. A. Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340. ACM, 2012.
10. 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.
11. M. Harris, S. Sengupta, and J. D. Owens. Gpu gems 3, chapter parallel prefix sum (scan) with cuda. 2007.
12. N. Heino and J. Z. Pan. Rdfs reasoning on massively parallel hardware. In *International Semantic Web Conference*, pages 133–148. Springer, 2012.
13. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
14. J. Paul, J. He, and B. He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.
15. M. Peters, C. Brink, S. Sachweh, and A. Zündorf. Scaling parallel rule-based reasoning. In *European Semantic Web Conference*, pages 270–285. Springer, 2014.
16. D. Rajagopal, E. Cambria, D. Olsher, and K. Kwok. A graph-based approach to commonsense concept extraction and semantic similarity detection. In *WWW*, pages 565–570, Rio De Janeiro, 2013.
17. F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.
18. H.-N. Tran and E. Cambria. GpSense: A GPU-friendly method for common-sense subgraph matching in massively parallel architectures. In *CICLing*, Konya, 2016.
19. H.-N. Tran, E. Cambria, and A. Hussain. Towards gpu-based common-sense reasoning: Using fast subgraph matching. *Cognitive Computation*, 8(6):1074–1086, 2016.

20. H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*, pages 299–315. Springer, 2015.
21. J. Urbani, F. Van Harmelen, S. Schlobach, and H. Bal. Querypie: Backward reasoning for owl horst over very large knowledge bases. In *International Semantic Web Conference*, pages 730–745. Springer, 2011.
22. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
23. V. Zheng, S. Cavallari, H. Cai, K. Chang, and E. Cambria. From node embedding to community embedding. https://arxiv.org/abs/1610.09950, 2017.