TripleID-Q: RDF Query Processing Framework using GPU

Chantana Chantrapornchai Chidchanok Choksuchat

I. INTRODUCTION

Linked data [1] utilize web resources to connect related data around the Internet. They contain common data such as DBpedia [2], biomedical data [3], geographical features data [4], etc. These linked data are presented in Resource Description Framework (RDF) [5] which is a standard and common framework to share and reuse data across the Internet. RDF data contain relationships, each of which is in a triple statement: subject, predicate, and object. subject denotes the resource, predicate shows the property of the subject and object is the value of the property. Each of these, subject, predicate, and object, is usually an Internationalized Resource Identifier (IRI), which is a very long string. RDF data contain millions of triple statements which result in a significant data size. Thus, it is time consuming to load and queries such as millions of triples.

With current parallel technology and architecture, it is possible to utilize multi-threading to perform such tasks to speedup the overall processing time. Current architecture has been advanced allowing it to process applications using multi-threading on many cores. Multi-threading can be in the form of high-level concurrency using Java Executor Service [6], or low-level CPU threads such as OpenMP or pthreads for a multi-core or many core computer. GPUs are one of such hardware platforms that contain many thousand cores. Due to its inexpensive cost, it becomes a cost-effective platform to gain high-speed processing, especially for imaging and graphic applications. Nowadays, a GPU has been used for general-purpose computing in many other application areas [8]. However, to use the GPU, applications must be designed properly to support the GPU architecture.

Though there are many open source tools for querying RDF data such as Redland [9], RDFLib [10], RDFsh [11], HDT [12] etc., which are easy to use, some of them are implemented in scripting languages which usually consume lots of time to load data, to create the internal representation as well as to query the model when the data become very large. Some are the libraries interfacing with C or Java with a complex data structure, making it difficult to port to utilize GPU to speedup the processing. Free community version can process limited number of triples (around 20 millions) [13]. The well-known open source one such Virtuoso [14] can support larger number of triples but do not support the use of GPUs. Blazegraph [15] is a high-performance graph database supporting Semantic Web (RDF) and SPARQL query on CPUs and GPUs with Java language but it also is not offered as an open-source or community-edition products on GPUs’ version.

To utilize a GPU for query processing, we have to consider two main aspects: the GPU architecture and the nature of the RDF query processing. For the first issue, a contemporary GPU have thousand cores supporting many concurrent threads. All these threads share the GPU memories. The GPU memory size is limited and the data must be transferred to the GPU memory before these threads can start computing.

To process an RDF query, all RDF data must be entirely loaded and stored in certain data structure. The aforementioned RDF libraries use graphs and heap storages to store RDF data. Some framework creates indexes for fast processing such as Header Dictionary Triple (HDT) which extracts common terms and creates dictionary as well as index triples by subject [12]. This format compresses the original RDF data very well. However, the implementation of these above data structure mostly are based on a list iterator, or recursive pointer. They contains deep pointers which are complex to load data GPU memory and let the threads to work on.

To process queries using GPU threads, data must be transformed into a proper form. The format should be compact so that all million triples can reside in a GPU memory. Also, the data structure should allow threads to look for proper relations of high-level concurrency using Java Executor Service [6], or low-level CPU threads such as OpenMP or pthreads for a multi-core or many core computer. GPUs are one of such hardware platforms that contain many thousand cores. Due to its inexpensive cost, it becomes a cost-effective platform to gain high-speed processing, especially for imaging and graphic applications. Nowadays, a GPU has been used for general-purpose computing in many other application areas [8]. However, to use the GPU, applications must be designed properly to support the GPU architecture.

Though there are many open source tools for querying RDF data such as Redland [9], RDFLib [10], RDFsh [11], HDT [12] etc., which are easy to use, some of them are implemented in scripting languages which usually consume lots of time to load data, to create the internal representation as well as to query the model when the data become very large. Some are the libraries interfacing with C or Java with a complex data structure, making it difficult to port to utilize GPU to speedup the processing. Free community version can process limited number of triples (around 20 millions) [13]. The well-known open source one such Virtuoso [14] can support larger number of triples but do not support the use of GPUs. Blazegraph [15] is a high-performance graph database supporting Semantic Web (RDF) and SPARQL query on CPUs and GPUs with Java language but it also is not offered as an open-source or community-edition products on GPUs’ version.

To utilize a GPU for query processing, we have to consider two main aspects: the GPU architecture and the nature of the RDF query processing. For the first issue, a contemporary GPU have thousand cores supporting many concurrent threads. All these threads share the GPU memories. The GPU memory size is limited and the data must be transferred to the GPU memory before these threads can start computing.

To process an RDF query, all RDF data must be entirely loaded and stored in certain data structure. The aforementioned RDF libraries use graphs and heap storages to store RDF data. Some framework creates indexes for fast processing such as Header Dictionary Triple (HDT) which extracts common terms and creates dictionary as well as index triples by subject [12]. This format compresses the original RDF data very well. However, the implementation of these above data structure mostly are based on a list iterator, or recursive pointer. They contains deep pointers which are complex to load data GPU memory and let the threads to work on.

To process queries using GPU threads, data must be transformed into a proper form. The format should be compact so that all million triples can reside in a GPU memory. Also, the data structure should allow threads to look for proper relations...
with a high degree of parallelism. Our research goal is to speedup large RDF query processing using a GPU. In order to achieve this goal, the following subproblems are investigated.

- How to design the compact representation for RDF data that is suitable for the GPU memory layout.
- Decide the information that needs to be inside the GPU memory for processing.
- How to utilize the GPU threads for concurrent processing.
- How to integrate the tasks performed by a GPU and CPU to obtain final query results.

To address the above issues, we propose a simplified format, *TripleID*, which is a transformed representation to encode the RDF data into unique IDs. The conversion to *TripleID* can be done in linear time. Such a file is small and can be easily loaded to the GPU memory. The data are kept in GPU memory as long as needed.

We adapt the search algorithm to utilize GPU threads to look for specific data according to a user query. The found data are returned to the GPU host and then mapped back to the corresponding name. The CPU side manages how to store, and select the returned data properly. It sends new data to the GPU for the next search. A CPU and GPU interact with each other depending on query operations such as union, intersection, or join. To lookup *TripleID*, the GPU threads are invoked. There is no need to transfer data to the GPU memory again. Some data returned from the GPU may be removed due to redundancy and may be merged with previous returned results. CUDA Merge-Join and Thrust libraries are used to speed up or more can be obtained compared to querying using the traditional RDF store.

Then, all IDs are loaded to the GPU memory. The converted *TripleID* files are 2-4 times smaller than the original NT files and the conversion time to *TripleID* is 3 times faster than the original NT file loading time. The framework can process a simple *RDF* query faster than traditional RDF library. Especially for the *RDF* store loading time. The framework starting from taking RDF data in the triple form and the conversion time to *TripleID* is 3 times faster than the original NT files.

A. Resource Description Framework (RDF)

Resource Description Framework (RDF) is a common format used to describe data in a relation form. It is represented in a triple form, (subject, predicate, object) where each term is usually a Internationalized Resource Identifier (IRI) which can be linked to another web resource [18].

An example of the RDF triple is shown as:

```
<http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#Air>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#AbioticEntity>
```

The above triple implies Air is a subclass of AbioticEntity based on RDFS vocabulary. The SPARQL [19]. A SPARQL's SELECT statement is similar to SQL SELECT statement. A given query can ask for subjects, predicates, and/or objects of the triples. The query in Listing 1 contains two subqueries, asking to “find all authors of The Journal of Supercomputing”, adapted from [20]. ?authors are variables whose values are the answers for the SELECT statement. dc is an abbreviation prefix of a standard vocabulary resource from Dublin Core [21].

```
Listing 1: find all authors of The Journal of Supercomputing

1 | PREFIX dc: <http://purl.org/dc/elements/1.1/>
2 | SELECT ?yr ?authors
3 | WHERE {
4 |  ?journal dc:title "The Journal of Supercomputing"^^
5 |      xsd:string .
6 |  ?journal dc:creator ?authors .
}
```

If one would like to infer a subclass (rdfs:subClassOf) between any two terms ?x, ?z. We can create two subqueries that are connected via a temporary variable, i.e., if ?x is a subclass of ?y, and ?y is a subclass of ?z, then ?x is a subclass of ?z [22] as shown in Listing 2.

```
Listing 2: Subclass transitivity

1 | SELECT ?x ?z
2 | WHERE {
3 |  ?x rdfs:subClassOf ?y .
4 |  ?y rdfs:subClassOf ?z .
}
```

To process the above query, using a traditional RDF tool, it is necessary to load all triples into the memory. The triples are stored in data structures such as graph models. Each subquery...
is then processed and the results from each subquery are kept for merging.

**B. Graphics Processing Unit (GPU) and Compute Unified Device Architecture (CUDA)**

A Graphics Processing Unit is originally used to process graphics objects for display. With the advanced hardware, they contain thousand cores which can be used to do any kind of general-purpose computations in parallel. Though they have a lower clock speed than the CPU, the thousand cores can process faster if they are utilized properly.

In general, a GPU, sometimes called device, resides in a computer, called host. To utilize the GPU, a proper programming framework is needed. Compute Unified Device Architecture (CUDA) is one of the commonly used framework supporting an NVIDIA GPU [8]. In CUDA, threads are organized as grids of thread blocks. Threads in a block are executed simultaneously.

CUDA cores are grouped into Streaming Multiprocessor (SM). One GPU card contains 4-26 SMs. A GPU has many types of memories such as local, shared, global memories, etc. Global memory can be accessed by all threads in all blocks while the shared memory can be accessed by only threads in the same block. Global memory has the largest sizes, varying from 2G to 24GB depending on the card models. Even though the access time is slower than that of shared memory, the shared memory usually has the size up to 112KB. For general-purpose computing, the global memory is commonly utilized since it is the largest and and it can be both read and written. In some cases, for small frequently accessed data, the shared memory may be used. The data from the global memory must be copied to shared memory before accessing them.

Under this architecture, the GPU memory transfer latency can be an obstacle to improve the program execution time. Algorithms that utilize the GPU must be designed in such a way that the required data needs to be kept inside the GPU memory as long as possible to reduce the transfer time, thus reducing the whole execution time.

In our case, all RDF data must be transferred to the GPU memory before the querying process can be done. Since RDF data is large, global memory is used to store all of them. Compacting them will be advantageous since more RDF data will be held. The search is performed by concurrent threads and the found triple positions are returned. Complex queries processing can also be done inside the GPU memory with proper data arrangement.

**C. Related work**

Since we are interested in processing large RDF data using a GPU or a parallel platform. Such a platform has lots of computing nodes/cores which can be advantageous for parallel processing. Also, the platform needs all data on the device’s memory for processing while it has limited memory size. Thus, we study the previous works in two aspects: 1) utilization of a GPU or any parallel platform for information processing 2) the advantage of compacting data for saving memory storage or splitting data for concurrent processing.

1) **RDF processing with parallel platforms:** With the advancement of parallel platforms with many computing cores and bigger memory, large information can be stored and processed inside the device. The information processed can be in various forms such as database, large text files, or RDF data etc. He et al. considered speeding up relational database using the GPU [23]. The authors focused on designing data-parallel primitives such as split, merge, map, gather-scatter, sort, and join, for memory optimization. The main problem in GPU programming is that the array in the GPU memory must be allocated before the GPU kernel is invoked. They developed the lock-free scheme for storing result outputs where two phases are used: the first phase was to examine the total size of the results for the GPU memory allocation, the next step was to perform the operation on the result array in the GPU. Breß et al. [24] proposed a workload optimization scheme, called probability outsourcing. They considered benchmarking of 4 database operations aggregate, select, sort, and join across GPU devices. The implementation is based on CUDA framework. Groppe, et.al. focused on distributed merge join processing for RDF triples [25]. They used partitioned B+ tree for indexed triples. The indices were built using a cluster of 7 computers. Another concurrent technology available was Java stream and multithreading where Corcoglioniti et al. [7] proposed a library tool for process RDF data supporting filter, aggregate, inference, and deduplication. The tool processes the data in a pipeline fashion.

Some researchers were interested in inferring knowledge from RDF data, called RDF Schema (RDFS) entailment. RDFS contains a standard set of rules for an RDF vocabulary which new relations can be inferred from. One of the motivated works to us was presented by Heino and Pan. The RDFS entailment was performed on a cluster of CPUs with one device, (and subdevices) [26]. Their algorithm was implemented using OpenCL while the RDF graph representation was used. The steps of the entailment were similar to [27] while there was a synchronization between steps. The key concept was to remove duplicate items before sending the results back to the CPU to save the data transfer time and to compact the transferred data. Liu et.al [28] studied the problem of reasoning for RDF reasoning using streaming RDF triples over time. These reasoning rules can be implemented using several subqueries. Makni [29]’s proposal focused on social media data stream which can be often changed.

Table I summarizes the previous work mentioned and compares them in the aspect of target tasks, representation and platform tested. The works in [23], [25] focus on relational database operations while the work in [24] targets at query plan optimization through various GPU devices. The work in [7], [26] targets the RDF processing where entailment problem was considered in [26] and the later work in [7] presents Java library for RDF processing. Most of these works used hash table for speeding up the query while some utilizes indexing scheme such as B+ tree. Our work in the last row, we consider the similar common operators with TripleID representation without spending time to generate indices. The compact representation allows GPU to process large number of triples as well as RDFS entailment.
TABLE I: Comparison of previous works in RDF processing schemes using parallel technology.

| Previous works        | Platforms             | Representation | Target tasks                        |
|-----------------------|-----------------------|----------------|--------------------------------------|
| He et al. [23]         | GPU/CUDA              | N/A            | Relational database operation: join, sort, gather-scatter, map |
| Heino and Pan [26]     | A cluster/ GPU/ OpenCL | std::vector   | RDFS entailment                      |
| Breß et.al. [24]       | GPU /CUDA             | N/A            | Optimization of workload of database operation aggregate, select, sort, and join |
| Groppe et.al. [25]     | Cluster of computers  | B+ tree        | Distributed merge join with indexing |
| Corcoglioniti et.al. [7]| Java/ Multithread     | HashMap        | RDF libraries for building RDF processing pipeline |
| Our work              | GPU/CUDA              | TripleID       | RDF query select, union, join, filter and RDFS entailment |

In this work, optimization of RDF storage utilizing both CPUs and GPUs was considered. The RDF data might be stored and processed on GPUs or CPUs depending on the speed up dynamic measurement. Reasoning algorithms that are suitable for GPU computing were selected. The approach consists of three steps: optimizing SPARQL aggregate and ordering using CUDA reduction, parallel constraint check by GPUs, and dynamic materialization by the GPU.

2) Compressed data formats: Since the GPU memory size is limited and the copying time to and from the GPU memory can degrade overall performance, it is advisable to compact data before transferring. One of the pioneer efforts on transforming and compressing the RDF representation was by Atte et al. The representation was called BitMat, which stores relations in a bit matrix: one matrix is created for one predicate. Madduri and Wu presented a FastBit software tool using bitmaps compression. Kim et al. considered the binary Header-Dictionary-Triple (HDT) form and processed RDF queries using the GPUs. The bitmaps as well as dictionary in HDT were loaded to the GPU memory. The prefix sum was applied to compute predicate and object positions in bitmaps. They experimented on a simple set of queries. HDT is a popular compressed format. However, the conversion to this form takes a lot of time and memory. For a larger number of triples, HDT with Java interface was required to increase Java heap memory to handle more elements in the set and would take even more conversion time or C implementation should be applied. The paper, however, did not address the issue of speeding up the conversion process and data scaling. The bitmap itself is compact in a storage but when queried, bitmaps, dictionary information are needed. Such information must also be loaded into the GPU memory for searching and the conversion from such data structure to suit the GPU memory layout is required.

For a very large RDF file, Hexastore with MPI was used to support a cluster processing. Hexastore data can be split across the nodes in the cluster so that a concurrent query can be performed. Thus, a file splitting is another approach to handle concurrent searches. Simple file splitting scripts take a lot of time to run, hence using MapReduce to process large files is another possibility to run on a cluster which is recognized as batch processing. We may consider stream processing in the future with overlapping the memory transfer and the computation. Interesting Merge-Join operation in the GPU library introduced by Baxter is generic based on unified memory, and easy to use. Often, the number of merged results may be too large; thus, on a GPU host computer whose memory size equals 12G, it was possible to apply the libraries when the number of elements of each vector was around 5-6 thousands.

In this work, we consider processing the RDF data. We begin with considering the traditional search algorithm. The search algorithm can be customized in the framework. Although it is possible to use a fast search, the fast search usually needs preprocessing such as creating prefix/suffix tables or implicit state machines. The construction of the preprocessed table requires space and time overhead for different search strings. Our framework assumes simplicity by using thousand threads to do brute force matching. From the preliminary study and previous work, with thousand threads, the gained speedup with the optimized search scheme may not be significant considering preprocessing overhead.

The framework transforms the RDF data into the TripleID format which encodes IRI strings into IDs. The TripleID data are then transferred into the GPU memory. After that, concurrent threads search the required triples according to the given query. Indexing scheme is currently not considered. It is possible to create an index based on a tree structure such as HDT. The concurrent search scheme is also possible with indexing e.g. by subjects. Note that the GPU memory is also required to store index information for each tree level. With more indexing types, more memory space is needed.

In the following section, the framework is first presented and the algorithms for different query operations are described based on TripleID.

III. TRIPLEID-Q: PROCESSING FRAMEWORK

The challenges of this research are to process the big data set with the limited GPU memory and to simplify the representation properly for GPU computation. The design goal is as follows. 1) The format should be simple so as to minimize the conversion overhead. 2) It should not occupy too large space. 3) Since the GPU has a lot of threads to help search, we will not focus on the index construction, rather we intend to use the large number of threads to look for the data.
TripleID-Q framework contains components to perform input conversion, look for query answers, and return the results based on such a representation, presented in Figure 1. The RDF file (N3/N-Triple type) is transformed into four files as, Subject ID, Predicate ID, Object ID and TripleID files. The first three files are in the same format containing tuples in a form: (key, value), where key is an integer and value is a string. The TripleID file contains only triples in the form (SubjID, PredID, ObjID) and is a binary file assuming each 32-bit unique ID. When loading these ID files in to memory, zlib may be used to encode the values to save memory space for text. In theory, the size of IDs is max(lg n1, lg n2, lg n3) where n1 is the total unique term used for subjects, n2 for predicates, and n3 for objects respectively.

In Figure 1, Subject ID, Predicate ID and Object ID files are loaded into memory in Step (1). We use hash tables to store the tuples, (key, value) pairs. The given query is transformed into a triple form (? P ?) (2), where P is the predicate ID. For example, to search ABCPress publishes, which journals, in Step 2, ABCPress and publishes are transformed into SubjID, PredID ObjID, which are 1, 2, respectively. The query becomes 1, 2, ?. In Step 3, the Triple ID file is split into chunks and the chunk is loaded into GPU memory. Then, GPU threads concurrently look for 1, 2, in the GPU memory. The found triples are marked and returned. In Step 5, the TripleID 1, 2, 1 is mapped back to the values using the hash tables.

The framework is described as shown in Algorithm 1. A TripleID file is read by chunks. It is assumed that the keys to search are in array key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively, where value 0 is reserved to represent a free variable “?” and subject IDs are implicit, i.e., in an increasing sequence of 1, 2, 3 ..., N, where N is a total number of distinct subjects. In the second and third levels, SeqY and SeqZ are lists of PredID and ObjID. BitmapY and BitmapZ are markings of starting positions for predicates and objects respectively. Thus, all of the four arrays must be transferred to the GPU memory, and the concurrent search must be done through BitmapY, BitmapZ, SeqY, and SeqZ. Also, only the thread numbers that are related to indices plays the role of matching. GPUsearch depends on the selected search algorithm. This work implements a brute-force matching which finds the matches between given key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively.

A TripleID chunk is stored as dataArray in the GPU main memory. Thread i compares dataArray[i], dataArray[i+1], dataArray[i+2] to key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID, respectively. In Step 5, the TripleID file contains only triples in the form (SubjID, PredID, ObjID), and is a binary file. When loading these ID files, zlib may be used to encode the values to save memory space for text. In theory, the size of IDs is max(lg n1, lg n2, lg n3) where n1 is the total unique term used for subjects, n2 for predicates, and n3 for objects respectively.

In Figure 1, Subject ID, Predicate ID and Object ID files are loaded into memory in Step (1). We use hash tables to store the tuples, (key, value) pairs. The given query is transformed into a triple form (? P ?) (2), where P is the predicate ID. For example, to search ABCPress publishes, which journals, in Step 2, ABCPress and publishes are transformed into SubjID, PredID ObjID, which are 1, 2, respectively. The query becomes 1, 2, ?. In Step 3, the Triple ID file is split into chunks and the chunk is loaded into GPU memory. Then, GPU threads concurrently look for 1, 2, in the GPU memory. The found triples are marked and returned. In Step 5, the TripleID 1, 2, 1 is mapped back to the values using the hash tables.

The framework is described as shown in Algorithm 1. A TripleID file is read by chunks. It is assumed that the keys to search are in array key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively, where value 0 is reserved to represent a free variable “?” and subject IDs are implicit, i.e., in an increasing sequence of 1, 2, 3 ..., N, where N is a total number of distinct subjects. In the second and third levels, SeqY and SeqZ are lists of PredID and ObjID. BitmapY and BitmapZ are markings of starting positions for predicates and objects respectively. Thus, all of the four arrays must be transferred to the GPU memory, and the concurrent search must be done through BitmapY, BitmapZ, SeqY, and SeqZ. Also, only the thread numbers that are related to indices plays the role of matching. GPUsearch depends on the selected search algorithm. This work implements a brute-force matching which finds the matches between given key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively.

A TripleID chunk is stored as dataArray in the GPU main memory. Thread i compares dataArray[i], dataArray[i+1], dataArray[i+2] to key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively. In Step 5, the TripleID file contains only triples in the form (SubjID, PredID, ObjID), and is a binary file. When loading these ID files, zlib may be used to encode the values to save memory space for text. In theory, the size of IDs is max(lg n1, lg n2, lg n3) where n1 is the total unique term used for subjects, n2 for predicates, and n3 for objects respectively.

In Figure 1, Subject ID, Predicate ID and Object ID files are loaded into memory in Step (1). We use hash tables to store the tuples, (key, value) pairs. The given query is transformed into a triple form (? P ?) (2), where P is the predicate ID. For example, to search ABCPress publishes, which journals, in Step 2, ABCPress and publishes are transformed into SubjID, PredID ObjID, which are 1, 2, respectively. The query becomes 1, 2, ?. In Step 3, the Triple ID file is split into chunks and the chunk is loaded into GPU memory. Then, GPU threads concurrently look for 1, 2, in the GPU memory. The found triples are marked and returned. In Step 5, the TripleID 1, 2, 1 is mapped back to the values using the hash tables.

The framework is described as shown in Algorithm 1. A TripleID file is read by chunks. It is assumed that the keys to search are in array key.subj, key.pred, key.obj corresponding to Subject ID, Predicate ID, and Object ID respectively, where value 0 is reserved to represent a free variable “?” and subject IDs are implicit, i.e., in an increasing sequence of 1, 2, 3 ..., N, where N is a total number of distinct subjects. In the second and third levels, SeqY and SeqZ are lists of PredID and ObjID. BitmapY and BitmapZ are markings of starting positions for predicates and objects respectively. Thus, all of the four arrays must be transferred to the GPU memory, and the concurrent search must be done through BitmapY, BitmapZ, SeqY, and SeqZ. Also, only the thread numbers that are related to indices performs the search. Compared to this work, we generally consider storing TripleIDs without any index and use lots of threads to directly search through them. The preprocessing requires only for the data conversion, but the indexing process is not required.

Based on Algorithm 1, it is easy to handle multi-GPUs and a cluster of GPUs whenever more host memory is available in Line 3 of Algorithm 1 we can read each chunk for each GPU and in Line 6, the search kernel is called for each GPU. The results are aggregated from all GPUs and may be exchanged between GPU memory. CUDA-aware can be setup to combine MPI_Send and cudaMemcpy together in

Algorithm 1: Parallel Search for TripleID

Input: dataArray, key
Output: positionArray
1 Allocate device memory for dataArray, key, positionArray.
2 while not EOF do
3 Read a TripleID chunk in dataArray.
4 Copy dataArray, positionArray (initialized to false) and copy key to the GPU memory
5 Call GPUsearch with dataArray, key, and positionArray
6 Copy positionArray back to the host. Map positionArray to corresponding triples found.
7 end
8 Free all the memory.
one command and chunks are distributed to each node.

IV. HANDLING MULTIPLE QUERY OPERATIONS

Previous section demonstrates a mechanism to handle a single query for triples where the query for subject, predicate, object and any two combinations are possible. In this section, we explain the handling of union and join operations of subqueries (also called triple patterns).

A. Union Operation

One query contains many subqueries, for instance, the query from [40]:

Listing 3: Query with union 1

1  SELECT * WHERE {  
2    {<http://dbpedia.org/resource/Cabezamesada>  
3      rdfs:comment ?var0 . }  
4  UNION{<http://dbpedia.org/resource/Cabezamesada>  
5      foaf:depiction ?var1 .}  
6  UNION{<http://dbpedia.org/resource/Cabezamesada>  
7      foaf:homepage ?var2 .}}

The above query consists of three subqueries of the same triple pattern S P ?. In this example, each triple can be a result of only one triple pattern.

However, for the query as following, there are two variables in each subquery, where a triple may be the answer of two subqueries. For example, the triples that are answers of the first subquery may also be the answer of the second subquery, ?var2 foaf:depiction ?var3. That is ?var2 may be <http://dbpedia.org/resource/Cabezamesada>, foaf:depiction may be ?var0 and ?var1 may be ?var3.

Listing 4: Query with union 2

1  SELECT * WHERE {  
2    {<http://dbpedia.org/resource/Cabezamesada>  
3      rdfs:comment ?var0 . }  
4  UNION{<http://dbpedia.org/resource/Cabezamesada>  
5      foaf:depiction ?var1 .}  
6  UNION{<http://dbpedia.org/resource/Cabezamesada>  
7      foaf:homepage ?var2 .}}

B. Join Operation

Results from several subqueries can be joined regarding the relation between each subquery. Example in Listing 5 [40] contains 5 subqueries, each of which is the pattern according to Table II:

Listing 5: Query with 5 subqueries

1  PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns>  
2  PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl>  
3  SELECT ?X, ?Y1, ?Y2, ?Y3  
4  WHERE {  
5    ?X rdf:type ub:Professor .  
6    ?X ub:worksFor <http://www.Depart0.University0.edu> .  
7    ?X ub:name ?Y1 .  
8    ?X ub:emailAddress ?Y2 .  
9    ?X ub:telephone ?Y3 .  
10  }

Table II: Subquery pattern of Listing 5

| Sub-query | Triple pattern |
|-----------|---------------|
| q0: ?X rdf:type ub:Professor | (? P O) |
| q1: ?X ub:worksFor <http://www.Depart0.University0.edu> | (? P O) |
| q2: ?X ub:name ?Y1 | (? P ?) |
| q3: ?X ub:emailAddress ?Y2 | (? P ?) |
| q4: ?X ub:telephone ?Y3 | (? P ?) |

The relational join implementation is based on the design pattern of CUDA library by Modern GPU or
Example related triple patterns

In (2). Merge-join in Mgpu is called with key vectors. The results obtained are indexed pairs that display the positions of both keys that are joined. The positions of both keys are used to extract the corresponding value vector elements from the vectors in (3). Similarly for the cases of OO, PP, OP, OS, PS, PO, SP, SO, the proper terms of the triples for each result from any subquery \( q_i \), \( R_q \) and results from subquery \( q_j \), \( R_q \) are copied as keys for merging. The positions for pairs of keys that match are returned as a vector of an element index pair as depicted in Figure 5. Then, the result vectors after joining are used for the next join in the next relation in \( REL \).

Results: \( R_{q_0} \rightarrow R_{q_k} \)

Fig. 4: Vectors of triple results.

C. Other Operations

To handle other operations such as FILTER, the query results are first obtained, then the IDs of terms must be converted back to string values. A regular expression may be used to filter ID names of the matched TripleIDs.

An extra structure is needed to only keep variables in each subquery. The variables for each subquery are used to find relationships \( REL \) are discussed in Subsection IV-E. From a SELECT statement, the selected variables must be returned. To handle DISTINCT, a hash table is used to store the results of a variable. Various GPU hash table versions are suggested in the literature [41], [42]. In the future, finding a good ID assignment of subjects, predicates, and objects in such a way to preserve the ordering and to filter out part of triples that are not relevant to the subquery is an interesting problem. The total remaining triples will reduce the size of the GPU memory used in performing operations such as join, union, etc.
Figure 6 presents an overall process when mixing these operations, i.e., SELECT, DISTINCT, JOIN. After the query is split into subqueries $q_0, ..., q_k$, each subquery is searched against the TripleIDs by GPU threads. The resulting triples are marked as the answer of a subquery and the marked triples are extracted to store in the vectors corresponding to the subquery. The filter is used during this step. The join operation starts from the left result $R_{q_0}$ to the right one $R_{q_k}$. Note that before joining, each result vector must be sorted. After joining all results, the final results are merged to keep only distinct values. When considering query optimization, join ordering can be changed.

V. EXPERIMENTS

The experiments demonstrate the efficiency of the framework in the following aspects. First, the conversion time to TripleID format is compared to the conversion to other formats and the size of TripleID file is compared to the original file type such as RDF and N-Triple file, and other formats such as HDT and RDF store. Next, the search time to the these files is measured in various aspects: the number of subqueries, the number of input triples, and different operations.

The tested machine had the following specification: Intel(R) Core(TM) i7 − 5820K CPU @ 3.30GHz, 6 cores, and 16 GB RAM with NVIDIA Tesla K40. The card contained 15 Multiprocessors, 192 CUDA cores per MP (totally 2,880 CUDA cores) with maximum clock rate 745 MHz (0.75 GHz). Memory bus width was 384-bit. Total amount of global memory was 12GB. The targeted thread block size and grid size equal to 1024 and 480 respectively, which yield the best performance on our machine. Other tests that explore the other block size and grid size are demonstrated in [35].

A. DATA SETS

Two data sets are considered: Billion Triples Challenge Data Sets and SP$^2$Bench Data Sets. The first data set was obtained from Billion Triples Challenge [43]. The downloaded contents encoded in N-Quads format [44] were split into chunks of 10 million ($10^7$) statements, called chunk 01,02,..,07, each of which has a size of 350 MB. These splits were combined to obtain the files with various sizes as shown in [45] A whole crawled data available as “BTC-small” has size equal to 2.172 GB. These files were converted into an N-Triple format [17] format The conversion program (command-line tool), rdf-convert-0.4 (http://sourceforge.net/projects/rdfconvert/), was used.

TABLE IV: Data set characteristics (BTC).

| data set | # sub | #pred | #obj | #triples |
|----------|-------|-------|------|----------|
| 01       | 314,285 | 3,438 | 583,323 | 1,868,637 |
| 0103     | 778,772 | 5,849 | 1,383,945 | 8,160,648 |
| 0203     | 504,082 | 4,477 | 990,414  | 3,291,997  |
| 0207     | 366,654 | 3,563 | 688,019  | 2,017,469  |
| 012347   | 1,113,824 | 7,542 | 1,674,407 | 7,083,790 |
| BTC-small| 1,383,542 | 8,205 | 2,260,819 | 9,627,877 |

For SP$^2$Bench [20], the data sets were generated with different number of triples up to 100 million triples. SP$^2$Bench produces the data sets in an N3 format [45]. These files contain various numbers of subjects, predicates and objects as shown in Table V.

TABLE V: Data set characteristics (SP$^2$Bench).

| data set (triples) | # sub | #pred | #obj | #triples |
|--------------------|-------|-------|------|----------|
| 5M                 | 896,359 | 76     | 2,400,922 | 5,000,120 |
| 10M                | 1,712,642 | 77    | 4,662,411  | 10,000,091 |
| 20M                | 3,404,855 | 153   | 9,379,289  | 20,000,429 |
| 50M                | 8,639,994 | 306   | 24,058,862 | 50,000,100 |
| 100M               | 17,652,609 | 613   | 48,965,319 | 100,000,144 |

B. TOOLS’ DESCRIPTION

Our following experiments show the various tested tools. The gathered tools focus on RDF querying with free, open source development: Redland, Mentok, Stardog, Virtuoso, and HDT. They have various implementations. HDT has both C and Java implementation and interfacing. In the experiments, C implementation is used for Redland, Mentok and HDT. Implementation for HDT has an indexed supported for SPO. Virtuoso is the largest one with an open source support for large RDF data while Stardog community edition can support around 20 million triples while larger RDF data is supported with the enterprise version and free for trial for 30 days.

C. PREPROCESSING TIME

We measure the preprocessing of using different formats. The conversion to TripleID time is investigated and compared to the conversion time to HDT from the original NT format. Then we measure the loading time, the case of using these RDF stores, which reads and parses RDF files (and construct an internal graph model in some cases).

TABLE VI: Loading time in seconds using Redland, Mentok, and TripleID representations on BTC.

| data set | Redland | Mentok | TripleID |
|----------|---------|--------|----------|
| 01       | 14.89   | 114.87 | 0.52     |
| 0103     | 46.84   | 261.58 | 1.33     |
| 0203     | 31.66   | 166.84 | 0.92     |
| 0207     | 16.464  | 106.23 | 0.66     |
| 012347   | 68.64   | 369.90 | 1.95     |
| btc-2009-small | 83.77 | N/A    | 2.4      |

Table VI presents the loading time for each tool for the data set in Table IV Redland library consumes more time
to load the RDF file and construct the graph model. Note that the query time of Redland is about 1/2 or 1/3 of the model loading time. It is found that Mentok’s loading time was much more that of Redland while the query processing could obtain benefits from multiple MPI nodes. From this observation, when the number of triples becomes very large, the straight-forward program which reads RDF triples and creates a simple representation will save this preprocessing overhead.

**TABLE VII: Loading time in seconds for SP²Bench using Stardog, HDT, TripleID.**

| data set | Stardog (s) | HDT (s) | TripleID (s) |
|----------|-------------|---------|---------------|
| 5M       | 40.98       | 0       | 1.86          |
| 10M      | 873.98      | 0       | 4.1           |
| 20M      | 3,820.71    | 0.01    | 8.66          |
| 50M      | 424.54      | 0.03    | 19.65         |
| 100M     | 1,171.36    | 0.05    | 42.56         |

**TABLE VIII: Comparison for conversion time (HDT and TripleID) in seconds for BTC**

| data set | HDT (s) | TripleID (s) | Speedup HDT/TripleID |
|----------|---------|--------------|----------------------|
| 01       | 19      | 3.25         | 5.85                 |
| 0103     | 51      | 8.57         | 5.95                 |
| 0203     | 34      | 6.15         | 5.53                 |
| 0207     | 22      | 3.06         | 7.19                 |
| 012347   | 71      | 10.37        | 6.84                 |
| btc-2009 | 94      | 28.86        | 3.26                 |

**TABLE IX: Comparison for conversion time (HDT and TripleID) in seconds for SP²**

| data set | HDT (s) | TripleID (s) | Speedup HDT/TripleID |
|----------|---------|--------------|----------------------|
| 5M       | 56      | 14.37        | 3.90                 |
| 10M      | 62      | 31.04        | 2.00                 |
| 20M      | 231     | 62.08        | 3.72                 |
| 50M      | 360     | 148.44       | 2.43                 |
| 100M     | 1256    | 298.1        | 4.21                 |

Table VII compares the loading time of SP²Bench in Table VIII. SP²Bench generates larger number of triples. We compare the loading time of RDF data using the large triple store, Stardog [13]. To support large number of triples, the setting of Stardog was – JVM memory is 8G and Off heap memory is 64G. Stardog prefers the triples to be in the Turtle format or called in short, TTL [46]. Thus, RDF data were converted into TTL format. The reported time under “Stardog” column is the time used to load TTL files into Stardog data store. Under “HDT” column, the time for loading HDT data is shown. The time to load the HDT data set is very small compared to others since the HDT file is already small.

Table VIII and Table IX display the conversion time to the TripleID format compared to the conversion time to the HDT format for BTC and SP²Bench respectively. HDT with C implementation (rdf2hdt) is used for comparison. Conversion time to the HDT format is about 5 times longer than that of TripleID files for BTC and about 2-3 times longer for SP².

Table VIII and Table IX display the conversion time to the TripleID format compared to the conversion time to the HDT format for BTC and SP²Bench respectively. HDT with C implementation (rdf2hdt) is used for comparison. Conversion time to the HDT format is about 5 times longer than that of TripleID files for BTC and about 2-3 times longer for SP².

D. Compaction

Figure 7 and Figure 8 compare the file sizes after the conversion to TripleID for BTC and SP²Bench respectively.

As in the previous section, after transforming to TripleID format, the four files are generated. The file size in Column “TripleID” is the summation of TripleID file size plus the subject, predicate, object ID files’ size. The sizes are compared against the original RDF, NT and HDT files. TripleID size compared to NT size is around 3-4 times smaller. However, TripleID size is 2 times larger than that of HDT format since we do not eliminate redundancy (due to shared subject and object elements) and we do not perform the compression while as noted in the above subsection, the TripleID conversion time is about 3 times faster than the HDT conversion time.

We also tried convert some large data set such as ‘012347’ and ‘btc-2009’ to Stardog format and we found that the size of Stardog format is around 1/2 of that of NT format. For the large case, SP²Bench, we compare against N3 and Stardog. N3 is smaller than NT size and our TripleID size is smaller than that of Stardog database.

E. Single Subquery Speedup

Table X displays processing time of each simple query containing one subquery.

```sql
SELECT distinct ?subject ?object
WHERE { ?subject owl:sameAs ?object}.
```

Column "Redland" shows the query time using Redland. The Redland library for this test was modified so that it can handle larger models. Using the traditional Redland library to search reaches the memory heap limit for allocation of a graph model storage whose size was larger than that of 01 case (1.8M triples), due to the growth of the internal model, represented by the hash table. Redland library reallocates the model whose size is double to the current one when the hash table density is more than 50%. The machine could not allocate large continuous heap memory area to store the model, which made the program stops running. We, then, modified

---

Footnote:

1The Stardog’s reported time for loading in total and in Triples per second. This 20M case has a slowest is around 4.4K triples/sec while for other cases, 5M is 113.5K triples/sec, 10M is 11.4K triples/sec, 50M is 103.5K triples/sec, and 100M is 83.1K triples/sec.
Redland source code to split into smaller submodels and to link the submodels as a list iterator. The splitting was done after parsing of the input RDF file by Rasqal parser.

Column "Mentok" shows query time using Mentok which is the reimplementation of Hexastore and the addition of MPI. This one demonstrates the use of distributed RDF models. Testing this library, we deployed Mentok on a cluster of 4 nodes with MPI, where each node was Intel(R) Xeon(R) CPU X3470 @ 2.93GHz. Column "HDT" displays the query time using HDT library (C implementation). These reported numbers are query time excluding loading time. Column "TripleID" is our search time. The speedup for each case (TripleID over Redland, TripleID over Mentok and TripleID over HDT) is displayed under column "Speedup". TABLE X: Time comparison in seconds between Redland, Mentok, HDT and TripleID for a simple query.

F. Multiple-Subquery Speedup

The performance of queries containing subqueries where each subquery contains union, join, or filter is measured. Particularly, the selected data sets from previous subsections are considered, with the cases of 5 million triples and 7 million triples, namely 0103 and 012347 from Table [V].

Three types of queries are considered with different focuses: Q1-Q5 only focus on union operations, Q6-Q8 focus on filter and union operations, and Q9-Q16 emphasize on join and filter operations. The join operation may be in the type of SS, OS, or two consecutive SSs or three consecutive SSs etc. Details of the queries are in Appendix ??.

TABLE XI: Query time in seconds using TripleID, HDT, Stardog on SP2Bench.

The speedup of querying using TripleID over Redland is significant which is about 48-390 times faster. Compared to the speedup of querying over Mentok for TripleID is about 4-10 times faster. We could not perform the test for BTC-small for Mentok since it used up the memory allowed in our cluster environment. HDT gives a close performance to our TripleID, and the addition of MPI displays the query time using HDT library (C implementation) [12]. These reported numbers are query time excluding loading time. Column "TripleID" is our search time. The speedup for each case is displayed under column "Speedup". TABLE XI: Time comparison in seconds between Redland, Mentok, HDT and TripleID for a simple query.

The speedup of querying using TripleID over Redland is significant which is about 48-390 times faster. Compared to the speedup of querying over Mentok for TripleID is about 4-10 times faster. We could not perform the test for BTC-small for Mentok since it used up the memory allowed in our cluster environment. HDT gives a close performance to our TripleID, and the addition of MPI displays the query time using HDT library (C implementation) [12]. These reported numbers are query time excluding loading time. Column "TripleID" is our search time. The speedup for each case is displayed under column "Speedup". TABLE XI: Time comparison in seconds between Redland, Mentok, HDT and TripleID for a simple query.

that the loading time was 776.96 seconds and the query time was 1.25 seconds. Thus, we could not perform the larger test using RDFlib since the process would use too much memory resource than allowed.

Table [XI] shows the total query time for our TripleID form in Column "total time" for SP2Bench. This benchmark contains more number of triples. We compare against Stardog 4.2.1, and HDT, with the query pattern "?PO", where P is rdf:type and O is foaf:Person. The total time obtained from querying 100 million triples is 2.28 seconds where the triple conversion time was 298.1 seconds.

TABLE XI: Query time in seconds using TripleID, HDT, Stardog on SP2Bench.

| Data set | Stardog | HDT | TripleID | Speedup |
|----------|---------|-----|---------|---------|
| 5M       | 0.25    | 1.21| 0.25    | 1.0     |
| 10M      | 0.65    | 0.77| 0.77    | 1.0     |
| 20M      | 1.07    | 1.14| 1.14    | 1.0     |
| 50M      | 1.66    | 1.82| 1.82    | 1.0     |
| 100M     | 2.28    | 2.55| 2.55    | 1.0     |

2Virtuoso 7.2.2 [14] which is a column store as well as isql from OpenLink Interactive SQL (Virtuoso), version 0.9849b were used. The default setting for Virtuoso was assumed.
ranges 16-108 times. In the case of Q16 (N/A), the query could not be executed using Redland because the query processing consumed all the memory resources and the execution was aborted. The rows with "-"] indicate that TripleID yields no speedup. For Virtuoso, the speedup varies from 3-1,131 times. For Q5 or Q6, the query pattern is "S??" where Virtuoso can perform very fast. Stardog performs queries much faster than Virtuoso for the queries that returns large number of results. Stardog also performs well when the the query pattern is "S??". It gives fast join results for queries Q9-Q12. The union operation takes longer time. HDT running time is "S??". It gives fast join results for queries Q9–Q12. The total execution time depends on the query operations and the number of results of the certain query. 

In some case, the number of final results does not reflect the total time since it also depends on the number of intermediate results before joining. The time in Column "join", indicated by 0*, which is closed to zero in Q9-Q12, implying that the number of intermediate results are small. When the join time such as in Q14 is detectable, the number of intermediate results is significant. In Q14, the first, second, and third subqueries return 22,626 results. In Q15, the first subquery returns 22,626 results and the second subquery returns 6,300 results. For the join with large intermediate results, the speedup will be more.

G. Entailment Queries

We apply our framework to process queries according to entailment rules. Table XIV presents rules used as a query benchmark [26] out of 13 D* rules [48] since the other rules involve only one subquery. These rules are transformed to the queries which contain subqueries. Hence, GPUSearch must be called twice.

Table XII: Comparison between Redland and TripleID performance for BTC-0103 dataset.

| Query | #Res | Redland | Virtuoso | Stardog | HDT | TripleID | Speedup |
|-------|------|---------|----------|---------|-----|----------|---------|
| Q1    | 20.01 | 43.29   | 7.54     | 7.82    | 2.38| 0.54     | 1.36    |
| Q2    | 78.06 | 0.34    | 2.00     | 0.98    | 1.15| 7.30     | 0.01    |
| Q3    | 870.89 | 0.34   | 2.00     | 0.98    | 1.15| 7.30     | 0.01    |
| Q4    | 897.67 | 0.34   | 2.00     | 0.98    | 1.15| 7.30     | 0.01    |
| Q5    | 24    | 43.21   | 5.54     | 0.04    | 0.14| 1.15     | 0.32    |
| Q6    | 18    | 43.05   | 5.55     | 0.04    | 0.14| 1.15     | 0.32    |
| Q7    | 22    | 43.23   | 1.45     | 0.04    | 0.14| 1.15     | 0.32    |
| Q8    | 20.37 | 43.27   | 19.49    | 4.87    | 1.49| 0.80     | 0.48    |
| Q9    | 1     | 43.04   | 17.67    | 2.14    | 0.14| 0.90     | 0.34    |
| Q10   | 0     | 43.08   | 6.17     | 16.49   | 0.36| 0.46     | 0.38    |
| Q11   | 98    | 43.95   | 5.61     | 14.70   | 0.36| 0.11     | 0.46    |
| Q12   | 1,529 | 43.08   | 6.17     | 16.49   | 0.36| 0.46     | 0.38    |
| Q13   | 30.42 | 43.16   | 8.23     | 22.97   | 2.62| 0.53     | 0.38    |
| Q14   | 144.84| 43.09   | 15.36    | 58.53   | 20.62| 0.98    | 0.68    |
| Q15   | 55.95 | 43.97   | 6.82     | 1.28    | 0.65| 0.43     | 0.42    |
| Q16   | 86.84 | 43.97   | 6.82     | 1.28    | 0.65| 0.43     | 0.42    |

Table XIII presents timing results for the larger BTC data set (012547) containing 7 million triples. The speedup trend is shown in the similar manner as in Table XII. More speedup is gained when compared to Redland, Virtuoso, Stardog and HDT, especially in Q1,Q2,Q3,Q4,Q14,Q16. The results imply that the total execution time depends on the query operations and the number of results of the certain query. In some case, the number of final results does not reflect the total time since it also depends on the number of intermediate results before joining. The time in Column "join", indicated by 0*, which is closed to zero in Q9-Q12, implying that the number of intermediate results are small. When the join time such as in Q14 is detectable, the number of intermediate results is significant. In Q14, the first, second, and third subqueries return 22,626 results. In Q15, the first subquery returns 22,626 results and the second subquery returns 6,300 results. For the join with large intermediate results, the speedup will be more.

G. Entailment Queries

We apply our framework to process queries according to entailment rules. Table XIV presents rules used as a query benchmark [26] out of 13 $D^*$ rules [48] since the other rules involve only one subquery. These rules are transformed to the queries which contain subqueries. Hence, GPUSearch must be called twice.
The execution time of queries in Appendix using TripleID, HDT, Stardog, Virtuoso, MySQL, HDT, and TripleID-C is reported in Table XV in column "TripleID", "HDT", "Stardog", "Virtuoso", "MySQL", and "TID/C" respectively. "TID/C" is TripleID implementation using only the GPU host. In column "#Res1", the number of the results of the first subquery, for example, in Rule (2), first query is the "If RDF Graph contains" part, as p rdfs:domain D. The second query is to search for all p’s that are previously found in all triples.

For Rule (5), the first query is p rdfs:subPropertyOf D. Column "#Dist1" is the number of distinct results from column "#Res1". For column "#Res2", the number of results is from the second search. Similarly, "#Dist2" is the number of distinct items from column "#Res2". At last, column "All" shows the total combined results from "#Res" and "#Res2".

In Table XV after eliminating redundant results from the first GPU search (Column "#Res1"), keysArray size is much smaller. Only distinct results are sent as inputs to the second GPU search. It is obvious that Virtuoso and Stardog can handle large databases very well. Comparing the speedup of our GPU version and CPU version, it is obvious that the speedup is up to 42 times. Our approach works well when there are a lot of intermediate results and final results, eg. R2,R3,R7 because of the simultaneous search from GPU threads. If there are very few results for a certain query, then the total execution time is dominated by memory transfer time as seen in R5, R9, and R11 cases, where Virtuoso or Stardog is faster.

H. Effects of Data Transfer Time

To observe the scaling aspect, when the number of results to transfer back increases. Let us consider the data set item ‘0103’ from BTC data set and take Q2 as an example. We double the data ‘0103’, 2 times, 4 times, 8 times, and 16 times, called 0103-2, 0103-4, 0103-8 and 0103-16, respectively. Figure 10a shows the “Query time” and “Data time” of Q2 of TripleID. The “Data time” shows the data transfer of the results back. For the case 0103-16, the query time is double from the case for the case 0103, while the data transfer time in this case is about 20% on average of the query time. However, the loading time of the data to GPU memory is double as expected in Figure 10b.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present a framework, TripleID-Q based on TripleID format for query processing. First, the conversion from standard RDF triple format to TripleID format is performed. The subject, predicate, and object ID files are generated and the TripleID file which contains rows of IDs of subjects, predicates, and objects is generated. The storage
TABLE XV: Execution time in seconds of queries according to entailment rules.

| Data   | Rule | #Res1 | #Dist1 | #Res2 | #Dist2 | All | TripleID | HDT | Stardog | Virtuoso | MySQL | TID/C |
|--------|------|-------|--------|-------|--------|-----|---------|-----|---------|----------|-------|-------|
| 012347 | R2   | 8.395 | 2.437  | 226.433 | 169    | 169.776 | **18.09** | 34.13 | 764.95  | 3073.12  | 4.0422 | 33.45  |
|       | R3   | 9.589 | 2.505  | 226.099 | 186    | 62.005  | 2.46    | 30.85 | 740.23  | 752.29   | 4.0471 | 35.41  |
|       | R5   | 6.545 | 450    | 0      | 0      | 0.28    | 0.53    | **0.19** | 0.23    | 4.17785  | 6.7    |
|       | R7   | 6.545 | 1,120  | 32.433 | 95     | 22.855  | **0.55** | 38.39 | 128.88  | 1.77676  | 6.18   | 20.87  |
|       | R9   | 10    | 4      | 1      | 1      | 1      | **0.19** | 7.03  | 200.72  | 0.25     | 3.36   | 0.03   |
|       | R11  | 26.785| 4.716  | 87     | 47     | 90     | 1.24    | 0.65  | 2.42    | 11.99    | 53.06  | 69.92  |
| 012346 | R2   | 10.383| 1,396  | 301.680 | 205    | 219.596 | 4.1068  | 493.57 | 309.12  | 14.48587 | 1410.93 | 68.08  |
|       | R3   | 11.438| 3.592  | 305.591| 210    | 89.372  | **3.53** | 45.61 | 455.42  | 913.18   | 8.12593 | 77.38  |
|       | R5   | 7.980 | 584    | 0      | 0      | 0      | 0.39    | 0.68  | 0.76    | **0.22** | 7.09816 | 11.58  |
|       | R7   | 7.980 | 1,496  | 57.884 | 100    | 40.622  | **1.00** | 53.39 | 442.34  | 1.90766  | 8.65   | 28.32  |
|       | R9   | 10    | 4      | 1      | 1      | 1      | 0.21    | 9.04  | 657.43  | **0.05**  | 4.55   | 0.26   |
|       | R11  | 36.561| 6.739  | 91     | 49     | 98     | 3.06    | **0.85** | 3.72    | 6.22     | 97.22  | 124.58 |

Fig. 10: Larger data for query Q2.

The experiments demonstrate various queries where the intermediate results are filtered, union and/or joined. While the complexity and the number of results have significant effects in computation time for traditional library, our approach can process the complex query, with large intermediate results in seconds due to the use of large number of simultaneous threads during searching and joining stages. Our approach can give speedup the queries varying from 17-108 times over the traditional RDF query tool. Compared with the above RDF stores, our algorithm can speedup the queries up to hundred times for many union and join operations. When compared with another compact representation, HDT, the speedup of our algorithm is up to 7 times. Consider the compactness of the representation. The total ID file size is about 2-4 times smaller than the original files. It is only 2 times larger than HDT file size and it is about half size of Stardog RDF store. On the other hand, TripleID representation is simple so that the conversion time to this format is faster than HDT’s conversion time about 3 times. The results show the trade-off between the compactness, conversion time and query time.

The application of our algorithm to entailment queries also imply the efficiency. We gain consistent speedup for these queries over using HDT presentation, Stardog, Virtuoso, MySQL.

Our framework relies on the hash data structure where three internal hashes for storing subjects, predicates, and objects are constructed during TripleID conversion. The available heap memory limits the total maximum subjects, predicates, and objects we can store. This makes the conversion process get killed when it consumes too much memory in the user space. This limitation is eliminated in the next version (demonstrated in the future version [49]) where the vector is used in placed of the hash table. Also, if the total triples sizes are too large for the available GPU memory, it can also be scaled out to use multiple GPUs to hold several portions of TripleID data similarly as in [50]. Streaming process is another solution to overcome this limit. The next implementation will consider streaming operations and external sorting for conversion and querying.

ACKNOWLEDGMENT

This work was supported in part by the following institutes and research programs: The Thailand Research Fund (TRF) through the Royal Golden Jubilee Ph.D. Program under Grant PHD/0005/2554, DAAD (German Academic Exchange Service) Scholarship project id: 57084841, NVIDIA Hardware grant, and the Faculty of Engineering at Kasetsart University Research funding contract no. 57/12/MATE.

REFERENCES

[1] L. D. Community, “Linked Data,” [http://linkeddata.org/](http://linkeddata.org/) retrieved: January 2017.
[2] J. Lehmann, R. Isle, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, “DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia,” *Semantic Web Journal*, 2014.
[3] M. Salvador, P. R. Alexander, M. A. Musen, and N. F. Noy, “BioPortal as a Dataset of Linked Biomedical Ontologies and Terminologies in RDF,” 2013.
[4] “Mentor,” [https://github.com/kasei/hexastore](https://github.com/kasei/hexastore) 2010.
[5] W3C, “Resource Description Framework,” [Online]. Available: http://www.w3.org/RDF/2004/02/tr/20040210/00,ch_typ.png, retrieved: May 2016.

[6] J. B. Cushing, J. French, and S. Bowers, Eds. Springer Berlin Heidelberg, 2012, vol. 7649, pp. 133–148.

[7] C. Choksuchat and C. Chantrapornchai, “Various GPU memory utilisation for large RDF search,” accepted for publication in The Semantic Web Conference, 2004.

[8] J. Groppe and S. Groppe, “Parallelizing join computations of SPARQL RDF Library,” in Proceedings of WWW-10. Hong Kong: Springer Berlin Heidelberg, 2001.

[9] S. Tramp, N. Arndt, and N. Heino, “tidfish,” [Online]. Available: https://github.com/seehi/tidfish.sh, retrieved: August 2016.

[10] D. Beckett, The design and implementation of the Redland librdf RDF API Library,” in Proceedings of WWDP10. ACM, 2008, pp. 1008–1019. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874389.2874395

[11] NVIDIA, “NVIDIA GP1 programming guide,” [Online]. Available: http://developer.nvidia.com/thrust, 2017.

[12] C. Choksuchat and C. Chantrapornchai, “Practical parallel string matching framework for RDF entailments with GPUs,” in Proceedings of International Conference on High Performance Computing in Asia-Paciﬁc Region (HPCAsia), 2018.

[13] J. D. Cathrin Weiss, Panagiotis Karras, M. A. Martnez-Prieto, and A. Bernstein, “Hexastore: Sextuple indexing for semantic web data management,” in Proceedings of PVldb. ACM, 2008, pp. 511–524. [Online]. Available: http://ceur-ws.org/Vol-1268/paper9.pdf

[14] NVIDIA, “Thrust,” https://developer.nvidia.com/thrust, 2017, retrieved: January 2017.

[15] S. Union, “Stardog4,” [Online]. Available: http://stardog.com/, retrieved: January 2017.

[16] NVIDIA, “NVIDIA GPU programming guide,” [Online]. Available: http://developer.nvidia.com/nvcc, 2015, retrieved: July 2015.

[17] J. D. C. Fernandez, M. A. Martinez-Prieto, C. Gutierrez, A. Polleres, and M. Arias, “Binary RDF representation for publication and exchange (HDT),” Web Semantics: Science, Services and Agents on the World Wide Web, vol. 19, no. 0, pp. 22–41, 2013. [Online]. Available: http://www.websemanticsjournal.org/index.php/ps/article/view/6528

[18] M. Schmidt, T. Hornung, M. Meier, C. Pinkel, and G. Lausen, “SP2Bench: A SPARQL performance benchmark,” in Semantic Web Information Management, R. de Virgilio, F. Giunchiglia, and L. Tanca, Eds. Springer Berlin Heidelberg, 2010, pp. 371–393.

[19] DCMI, “Dublin Core Metadata Element Set, Version 1.1,” [Online]. Available: http://dublincore.org/documents/dces/, 2016.

[20] Y. Kim, Y. Lee, and J. Lee, “An efficient approach to triple search and join of HDT processing using GPU,” in Proceedings of The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA), ser. IARIA, 2015, pp. 70–74.

[21] D. Beckett, “The design and implementation of the Redland librdf RDF API Library,” in Proceedings of SP2Bench. Hong Kong: Springer Berlin Heidelberg, 2001.

[22] C. Gutiérrez, A. Polleres, and M. A. Martinez-Prieto, “Parallelizing join computations of SPARQL RDF Library,” in Proceedings of WWW-14. New York, NY, USA: ACM, 2010, pp. 41–50.

[23] C. S. Kouzinooulos and K. G. Margaritis, “String matching on a multicore GPU using CUDA,” in Informatics, 2009. PCI ’09. 13th Panhellenic Conference on, Sept 2009, pp. 14–18.

[24] J.-L. Gailly and M. Adler, “A massively spiffy yet delicately unobtrusive compression library,” [Online]. Available: http://zlib.net/, 2013, retrieved: November 2015.

[25] J. D. Cathrin Weiss, Panagiotis Karras, M. A. Martnez-Prieto, and A. Bernstein, “Hexastore: Sextuple indexing for semantic web data management,” in Proceedings of PVldb. ACM, 2008, pp. 1008–1019. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874389.2874395

[26] F. Corcoglioniti, M. Rospocher, M. Amadori, and M. Mostarda, “Efficient multi-GPU collectives with NCCL,” https://developer.nvidia.com/cuda-developer-network, retrieved: May 2016.

[27] J. D. Cathrin Weiss, Panagiotis Karras, M. A. Martnez-Prieto, and A. Bernstein, “Hexastore: Sextuple indexing for semantic web data management,” in Proceedings of PVldb. ACM, 2008, pp. 1008–1019. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874389.2874395

[28] M. Atte, V. Chaoji, M. J. Zaki, and J. A. Hendler, “Matrix ‘Bit’ Loaded: A scalable lightweight join query processor for RDF data,” in Proceedings of the 19th International Conference on World Wide Web, ser. WWW ’10. New York, NY, USA: ACM, 2010, pp. 41–50.
Chantana Chantrapornchai obtained her Bachelor degree (Computer Science) from Thammasat University of Thailand in 1991. She graduated from Northeastern University at Boston, College of Computer Science, in 1993 and University of Notre Dame, Department of Computer Science and Engineering, in 1999, for her Master and Ph.D degrees respectively. Currently, she is an associated professor of Dept. of Computer Engineering, Faculty of Engineering, Kasetsart University, Thailand. Her research interests include: parallel computing, big data processing, semantic web, computer architecture, and fuzzy logic. She is also currently with HPCNC laboratory and a principle investigator of GPU Education program at Kasetsart University.

Chidchanok Choksuchat received the Ph.D in Computer and Information Science from Silpakorn University. She is currently a lecturer in Information and Communication Technology Programme, Faculty of Science, Prince of Songkla University, Thailand. Her research interests include the CUDA, Java Concurrency, Ontology Engineering, SPARQL Endpoint, Linked Open Data, Data Science Toolkit, R programming and Internet of Things.