VEDAS: An Efficient GPU Alternative for Store and Query of Large RDF Data Sets

Pisit Makpaisit
Kasetsart University - Bangkhen Campus: Kasetsart University
chantana chantrapornchai (fengcnc@ku.ac.th)
Kasetsart University - Bangkhen Campus: Kasetsart University https://orcid.org/0000-0002-8699-5736

Research

Keywords: Query Processing, Parallel Processing, Graphic Processing Units, Resource Description Framework, SPARQL

Posted Date: June 9th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-587426/v1

RESEARCH

VEDAS: An Efficient GPU Alternative for Store and Query of Large RDF Data Sets

Pisit Makpaisit and Chantana Chantrapornchai*

*Correspondence:
fengcnc@ku.ac.th
Department of Computer Engineering, Kasetsart University, Bangkok, Thailand
Full list of author information is available at the end of the article

Abstract

Resource Description Framework (RDF) is commonly used as a standard for data interchange on the web. The collection of RDF data sets can form a large graph which consume time to query. It is known that modern Graphic Processing Units (GPUs) can be employed to execute parallel programs in order to speedup the running time. In this paper, we propose a novel RDF data representation along with the query processing algorithm that is suitable for GPU processing. Since the main challenges of GPU architecture are the limited memory sizes, the memory transfer latency, and the vast number of GPU cores. Our system is designed to strengthen the use of GPU cores and reduce the effect of memory transfer. We propose a representation consists of indices and column-based RDF ID data that can save GPU memory requirement. The indices and pre-upload filtering technique are then applied to reduce the data transfer between host and GPU memory. We add the index swapping process to facilitate the sort and join the data with the given variable and add the pre-upload step to reduce the size of results' storage, and the data transfer time. The experimental results show that our representation is about 35% smaller that the traditional NT format and 40% less compared to that of gStore. The query processing time can be speedup ranging from 1.95 to 397.03 when compared with RDF3X and gStore processing time with WatDiv testsuite. It achieves speedup 578.57 and 62.97 for LUBM benchmark when compared to RDF-3X and gStore. The analysis shows the query cases which can gain benefits from our approach.

Keywords: Query Processing; Parallel Processing; Graphic Processing Units; Resource Description Framework; SPARQL

Introduction

The Resource Description Framework (RDF) was proposed by W3C as a data exchange standard in semantic web. As it contains subject-predicate-object relations (triples), where each term can be IRIs (International Resource Identifier), a collection of them, called *triple stores* or *RDF dumps*, can represent a large linked data useful for querying and inference. Today, it is widely used in many areas such as describing taxonomy of animals [1], earth environmental thesaurus [2], influence tracker [3], US. patent description [4], or even Wikipedia [5] etc. Because it can represent a large connectivity, the RDF dump file can contain significant number of triples, which require efficient methods to store and query data.

In 2008, World Wide Web Consortium released the standard query language for RDF called Simple Protocol and RDF Query Language (SPARQL). It is a query language similar to an SQL in a traditional database with the supports of basic

query operations such as filtering, join, projection, sorting, etc. but it is capable of querying across RDF data in the network where endpoints are applicable.

To efficiently retrieve query results from the large RDF data, we need an RDF processing platform that can perform a SPARQL query in an acceptable time. The high performance and cost-efficient hardware accelerator like GPU is the one of the platform solutions. Nevertheless, we have to face a few design challenges for building applications for GPU:

1 The GPU has limited resources while RDF dumps are a large text file.
2 The transfer latency between the host and GPU can degrade the speedup gained. Normally, GPU processing requires all data kept in the GPU memory.
3 The number of threads in GPUs are large. Utilizing them at the same time can increase the processing speedup.

In this work, we propose a framework which is based on the TripleID representation[6]. To make it fit inside the GPU memory, we compress the representation by transforming them into a column format with column indices which can save a lot of memory since the RDF data is usually sparse. Next, the algorithm for primitive SAPRQL query operations such as selecting and join on our new representation are proposed. We also propose the pre-upload filtering technique that can reduce the data transfer between the host and GPU memory. In the experiments, we compare our column-based representation with the compressed exhaustive indices representation in RDF-3X and graph representation, gStore, in the aspect of size and query processing time. The results are promising due to the decrease of query time and storage size.

In short, VEDAS has the following benefits.

1 The representation is based on TripleID which can save the storage size upto 65% compared to N-triple format. The representation considers proper indices to allow fast tuple querying.
2 It provides a support for basic operations in querying which considers the GPU resource properly, e.g., the limited GPU memory and transfer overhead, and the use of massively parallel threads.
3 It works well in the case of the query that contains lots of join operations. For example, in the experiments, the query type C yields a speedup upto 284 compared to gStore and 13.09 compared to RDF-3X. These join operations lead to the large thread workload that significantly hides the transfer time.

The structure of this paper is as following. Section Background and Related work explains the related work and the inspiration of our work. Section VEDAS Framework and Operations presents our representation and the proposed processing algorithms based on the GPU. The experiments, comparison results, and analysis are described in Section Experiments. Then, Section discusses the extension to cover complex query types. Finally, Section Conclusion and Future Work concludes the work and discusses the future implementation.

Background and Related work

In this section, we first present the preliminary knowledge on Resource Description Framework and SPARQL. The background on GPU processing is also included. Next, we highlight the related work in RDF representation and query processing.

Resource Description Framework (RDF)

There are many representations for Resource Description Framework (RDF) data such as N-Triples, N3, N-Quad, RDF/XML, Turtle etc. The simplest and most popular representation is N-Triples where each statement (or each line) contains a triple of the form ⟨*subject, predicate, object*⟩ where *predicate* expresses the relation between the *subject* and the *object*. Each term, *subject*, *predicate*, and *object* can be any IRI string [7].

Such example of representation in N-Triples is in Figure 1. The triple implies that `Air` is a subclass of `AbioticEntity` which is based on RDFS vocabulary. `<http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#Air>` is a subject, `<http://www.w3.org/2000/01/rdf-schema#subClassOf>` is a predicate, and `<http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#AbioticEntity>` is an object. These terms are IRIs and are obtained from *biomedical ontology* [8]. The above N-Triples can be converted into RDF/XML as in Figure 2:

```
<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> .
```

Figure 1: N-Triples example.

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdf:Description rdf:about="http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#Air">
    <rdfs:subClassOf rdf:resource="http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#
    AbioticEntity"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 2: RDF/XML example.

Another interpretation of the RDF data is a directed labeled multigraph. *Subject* and *object* are vertices in a graph and *predicate* are edges that connect its corresponding *subject* and *object* as shown in Figure 3. The graph is shown Figure 4. The example implies that Bob is male and he knows Alice. Alice's birthday is May 20, 1997. Alice is the founder of YumYum restaurant that has fried rice in the menu.

Subject	Predicate	Object
Bob	knows	Alice
Bob	gender	Male
Alice	birthDate	"1997-05-20"
Alice	founder	YumYum
YumYum	isA	Restaurant
YumYum	hasMenu	"Fried Rice"
Bob	likes	Alice

Figure 3: Triple Example.

SPARQL is a query language that is commonly used for RDF data [9]. A SPARQL's SELECT statement is analogous to the SQL SELECT statement. Based

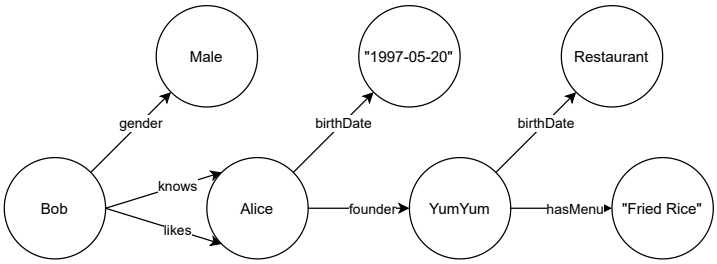


Figure 4: Triple data in graph visualization.

on N-triples, the query can select subjects, predicates, and/or objects of the triples. Like a normal SQL, a query can contain subqueries. In Listing 1, there are two subqueries: 1) find the journals with the title *The Journal of Supercomputing* and 2) find all authors from the above journals. In the query, `?authors` are variables whose values are the final answers for the SELECT statement. `dc` is an abbreviation prefix of `<http://purl.org/dc/elements/1.1/>` which is a standard vocabulary resource from Dublin Core [10]. For more complex queries, SPARQL also has modifiers for example, LIMIT, FILTER, and UNION. Our work will first demonstrate the use on the basic query type which is based on SELECT and WHERE. We also give the guideline for the implementation to other important query modifiers later on.

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
5    "The Journal of Supercomputing"^^ xsd:string .
6 ?journal dc:creator ?authors . }
```

In the big data era, RDF data is popular since it is a kind of NoSQL which has the information linkage and with the trend of data governance, such a standardized form is encouraged. The RDF data size is rapidly growing. Examples of large data sets include GeoSpatials (1.888M Triples), U.S. Census data (1 billion triples), World Bank Linked data (160 million triples), DBpedia (247 million triples), etc. [11] One of the challenges in this domain is to retrieve and process them efficiently. SPARQL is a de facto standard for querying RDF data. The syntax is similar to an SQL in the relational database. For example, "SELECT ?x ?y WHERE { ?x founder ?y . ?y isA Restaurant . }" is a query that lists all pair of person's name who is a restaurant founder and restaurant name. In Figure 3, the result for ?x is Alice and for ?y is YumYum. The SPARQL is a sub-graph matching in RDF graph. For more complex query, SPARQL has a modifier to describe the query for example, LIMIT for limit the number of result, FILTER for filtering results with boolean conditions and UNION modifiers for combining the results. Our work will first focus the basic query that contains SELECT and WHERE but we will offer the guideline for applying the implementation to some important modifier in Section Extension to Other Operations.

Graphics Processing Unit (GPU)

In the past, the GPU has been used to accelerate the the graphic applications such as gaming applications. Currently, many other applications utilize them to improve the performance due to its large number of parallel processing units.

The GPU is a Single Instruction Multiple Data (SIMD) architecture which can process multiple data simultaneously with its thousands cores running on the same instruction. Its architecture groups multiple processing units into multiple Streaming Multiprocessors (SMs). The GPU has a thousands of threads executing on these SMs, and are controlled by the scheduler.

The GPU sits inside a computer (called host) which also has CPU and main memory. It also has its own memory space that is separated from the host memory. Based on the latest GPU technology, it can have a maximum of 32 GB memory per card, which is small compared to the size of host memory which can be enlarged to hundreds or thousands of gigabytes. In addition, the GPU has a hierarchical memory layout such as registers, local memory, shared memory and global memory. The global memory has the largest size where the register is the fastest memory. Each thread has its own register and local memory. A group of threads (called a thread block) can access the same shared memory and are executed in the same SM. The global memory is accessible from all threads.

To use the GPU for computation, the data must be transferred from the host memory to GPU global memory. Transferring latency is one of the overhead incurred in the GPU processing etc. Once the data resides in the GPU memory, the GPU can start its execution. Thus, to maximize the application performance on GPUs, the following is the common considerations.

1. Reduce the transfer data size between CPU and GPU.
2. Hide the memory transfer latency by overlapping processing time and memory transfer time.
3. Maximize the parallelism between all the threads.
4. Optimize the GPU memory usage such as using shared memory to share data among of threads instead of global memory, enabling the locality, adjusting thread memory access pattern to reduce the global memory transfer etc.

In our work, we are interested to utilize the GPU to improve the query performance for RDF data. Due to the above constraints on GPU, we develop the RDF compact representation and introduce the query processing framework that is suitable for GPU processing. The framework contains three basic operations, pre-upload filter, index swapping, parallel merge-join which optimize the transfer time and enable the GPU parallelism. The framework will be scaled up to support multiple GPUs and a cluster in the near future.

Related Works

There are various works on RDF stores and query processing. We highlight the two subareas which are most related to us: RDF representation and parallel query processing.

RDF Representation

The RDF data store can be categorized into 3 classes: relational, graph, and matrix representations. The relational approach has been around for a long time [12]. It

treats the RDF data like a row in a table in a relational database. Using this approach has a benefit which allows the user to manipulate the data just like in relational database [13, 14, 15]. In [16], the SQL query was designed to run on a distributed system. Because RDF data is normally large, indexing the triples is important to make it efficient for querying [17]. However, this approach needs the high computation power when handling a high number of related data. The join operation is the bottleneck of the system.

Another natural approach is to use the graph representation. The graph representation shows the relationships among data. gStore [18] is one of the example that stores the RDF data in this representation. Thus, SPARQL query is represented as a graph. The sub-graph matching algorithm is used to find the result of query in such a representation. gStore has VS*-tree that contains the indices of the data, making the matching process faster [19]. Even though graph approach is more natural to handle the relation, it has the scaling problem for the limited shared memory system [20, 21]. Moreover, the irregular access pattern makes it difficult to effectively implement on GPU to utilize many threads and GPU memory.

The matrix representation is an alternative approach that is easy to compress the data and create indices. Yuan et al. proposed TripleBit that stores RDF data in bit matrix [22]. MAGiQ stored RDF in sparse matrix and proposed matrix algebra for query data [23]. The SPARQL query is converted into an equivalent matrix algebra and existing matrix algebra library (MATLAB and GraphBLAS) was utilized to process. gSMat also stores RDF as a sparse matrix and translates the join operation to the sparse matrix multiplication [24]. Because matrix multiplication is one of GPU basic operators, this work implements the join operator on both CPU and GPU (with CUDA).

Table 1 shows the summary of each representation. All of representations need indices to rapidly access the triple. For compressing the data, most of all works replace the RDF terms with unique id or bits.

Table 1: RDF Representation in the literature.

Representation	Detail	Advantage	Disadvantage
Relational	Store a list of triples with indices e.g. RDF-3X, TriAD	Simple and many techniques from relational database can be applied	Intensive computation for join the related data
Graph	Store subjects and objects as nodes and predicate as edges e.g. gStore	Natural way to retrieve link data	Scaling limitation on in-memory database. Hard to parallelize for a SIMD architecture
Matrix	Use a sparse square matrix to represent the RDF e.g. TripleBit, gSMat, MAGiQ	Reuse the existing matrix kernel on various platforms	Fixed size, complicate to represent multigraph

Besides the data representation, query processing and the join operator are also important. MapSQ handles the SPARQL query by using MapReduce framework in joining [25]. SMJoin utilizes the multi-way join algorithm to reduce the network cost and processing time [26] .

Distributed SPARQL Query

Some work considered to use a distributed approach to process the SPARQL query. Feng et al. [27] classified the distributed RDF system into 3 classes i.e. 1) the

one based on the existing general distributed computing framework like Hadoop, Spark etc. [28, 29] 2) the one based on the partitioning method [16, 30] and 3) the federated system that integrate multiple systems into virtual one. The classification is based on the storage types: partition, graph, and DBMS and two query executive strategies: partition and DBMS. The paper indicates that architecture, storage, and query are key factors of SPARQL query performance. For example, TriAD is based on partition, gStoreD is based on graph and, S2RDF is based on DBMS. The partitioning approch (TriAD) seems to outperform the others.

Peng et al. evaluated a SPARQL query using a distributed scheme [30]. In this work, the authors used the partial evaluation and the assembly framework. The authors modeled RDF data as a graph as well as the query. They proposed an algorithm to find a local partial match as partial answers in each fragment from the RDF graph. WatDiv, LUBM and BTC were used as benchmarks for measuring performance. The experiments also compared various cases: a large number of triples, varying intermediate results, each stage performance, partitioning strategy, in-memory operations etc.

In TriAD, the authors proposed asynchronous shared-nothing message passing architecture for processing SPARQL query [16]. The approach partitions the RDF graph and distributed the portions. METIS were used for graph partitioning. The SPARQL query was also transformed into a graph and the bindings between free variables and RDF entity are created. The query were executed in a distributed fashion with a global plan. The benchmarks, LUBM, BTC and WSDTS, were used for testing.

From the literature, for the distributed system, the aspects that are highly impact to the query efficiency are the architecture and the query type. Both affects the intermediate join subquery result size, the joining plan of subquery operations, the data partitioning strategy, and algorithms for matching etc.

SPARQL Optimization on GPU

The GPU can process the result matching from the subquery upon the join operation. For processing a subquery, the data for such query must be resided in the GPU memory. Transferring the data to GPU memory usually incurs significant overhead. The performance of the SPARQL query on the GPU highly depends on the representation that affects the total transfer size and the join algorithm. The join algorithm that is suitable for the GPU and the good query planner is the key to increase the query processing performance.

MapSQ [25] uses MapReduce technique on the GPU to increase the processing speed. The authors split the answer processing into 2 steps: 1) finding the subquery results by using gStore, 2) joining the results of subqueries by using the proposed MapReduce-based join algorithm that is implemented bsed on the GPU. They used LUBM benchmark to measure the performance and compared the results to gStore and gStoreD. The speedup gained was ranging from 1.15 to 2.05. SRSPG [28] is similiar to MapSQ but it implements a parallel join algorithm on Apache Spark which was executed on the GPU.

For the matrix-based approach, e.g. MAGiQ [23], it leverages MATLAB-GPU and SuiteSparse package for execution on the GPU. gSMat [24] also uses the sparse

matrix based representation and implements its own GPU join algorithm called SM-based join. The gSMat gained the speedup from RDF-3X and gStore ranging 1.87 to 16.13 times against various query types for the WatDiv 500M benchmark. With the sparse matrix library, the query engine has benefits from the new library version optimization. An ordinary matrix does not support multi-graph which is the nature of RDF data. It is also complicated to handle the advanced forms of SPARQL.

TripleID-Q [6] relies on the relational row based format to represent the RDF data and converted the triple into integer IDs to compact the data. The sub-result triples are simultaneously marked by GPU threads. The results are joined with the merge-join approach. However, the work did not consider the query planner and optimization. Table 2 compares the previous works in RDF processing using GPUs.

Table 2: Comparison of query processing for GPU-based approaches.

Works	Architecture/ Methods	Operations considered	Optimization	Data sets
MapSQ [25]	Distributed (MapReduce)	Select/Join	MapReduce-based join algorithm	LUBM
MAGiQ [23]	MATLAB-GPU and SuiteSparse	Select/Join	GraphBLAS library	LUBM-10240
gSMat [24]	Sparse matrix-based	Select/Join	Predicate statistic query plan generated algorithm and sparse matrix-based join algorithm	WatDiv, YAGO, DBpedia
TripleID-Q [6]	Relational model	Select/Join	GPU parallel search for triple pattern	SP^2Bench, BTC2009

VEDAS Framework and Operations

Due to the constraints in the GPU architecture, the main design goals are to minimize memory usage and speed up the query processing time. Figure 5 shows the components of our framework which includes three parts. 1) data storage and representation, 2) data loader and 3) query processor.

Data storage is where the converted RDF data is kept in the host side. It contains the proposed representation that is designed for GPU processing. The storage refers to the disk storage where the N-Triple data are first kept.

Data loader contains two subcomponents which are *parser* and *indexer*. The parser performs the syntax parsing of N-Triple data and the indexer makes the transformation from N-Triple data into *dictionary* and *index*. Such *Triple-ID* with the indices format facilitates the searching process.

From a SPARQL query, the query processing is done in *query processor*. It contains the parser which parses the SPARQL query and outputs an internal format of query operations. Next, the query planner will find the optimal order of query operations. At last, the query executor utilizes the query plan obtained from the query planner and applies the plan accordingly.

Data Representation

N-Triple format contains a string datatype as a basic element (such as IRI). Importing such a large number of triples directly to the GPU is not appropriate since it occupies lots of memory and induces large GPU-CPU memory transfer. Our representation converts the string data into 4 bytes integer (called *id*). This step

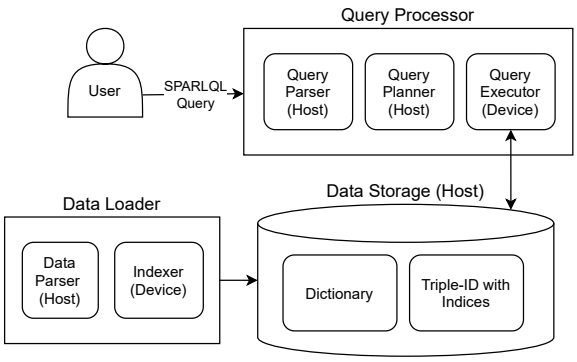


Figure 5: Elements of VEDAS.

particularly uses a hash function for encoding. In our case, we represent 1 triple with 12 bytes memory (4 bytes for a subject, 4 bytes for a predicate and another 4 bytes for an object). The mapping between a string *IRI* and the unique integer is saved in a dictionary on the host memory.

Each N-Triple statement contains three terms: ⟨ *subject (S)*, *predicate (P)*, and *object(O)* ⟩. Each S, P, and O is converted into a unique id, recorded in a dictionary. Thus, the triple statement becomes triple-ID, $tp = \langle id_1, id_2, id_3 \rangle$, where id_1 is the id of associated *subject* (S), id_2 is the id of associated *predicate*(P) and id_3 is the id of associated *object*(O). In general, the intermediate results after performing more than one subquery in a sequence may contain the different number of ids. We denote t as a tuple of (m, id)s, where m is the number of such id.

From a given triple-ID, for a fast access, the indexer constructs *permutations* for all possible indices, i.e., *POS*, *PSO*, *OPS*, *OSP*, *SPO*, *SOP*, i.e., $D_{POS}, D_{PSO}, D_{OPS}, D_{OSP}, D_{SPO}$ and D_{SOP}. For example, D_{POS} is the associated triple-ID that is sorted in order of predicates, objects, and subjects respectively.

Figure 6a shows the dictionary used in converting the terms to the triple-IDs. Figure 6b shows the triple data of example in Figure 3 after converting with a hashing function to the triple-ID. Figure 7a shows the triples after sorted by SOP to create the column-based representation in Figure 7b.

Term	*id*	**Term**	*id*
Alice	1	knows	8
Bob	2	gender	9
YumYum	3	birthDate	10
Male	4	founder	11
”1997-05-20”	5	isA	12
Restaurant	6	hasMenu	13
”Fried Rice”	7	likes	14

(a) Dictionary

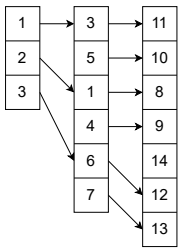
Subject	**Predicate**	**Object**
2	8	1
2	9	4
1	10	5
1	11	3
3	12	6
3	13	7
2	14	1

(b) Converted triple-ID

Figure 6: Dictionary and converted Triple-ID example.

Subject	Predicate	Object
1	3	11
1	5	10
2	1	8
2	4	9
2	4	14
3	6	12
3	7	13

(a) After sorting with SOP index



(b) Indexed column-based representation

Figure 7: Triple-id representation and example index.

Data Loader

The data loader is responsible for converting triple data in N-Triples format to the triple-ID format. Dictionary and completed indices are also constructed.

For the implementation, Redland raptor 2.2 library was used to parse N-Triples files. After that integer *id* for each term is assigned and all triple statements are converted into the triple-ID format. To create the permutation index, we employ *Thrust* library [31] to sort all triples on GPUs in various ways such as sort by subject/object, subject/predicate, predicate/object etc.

Since we need to sort the large set of data 6 times, the large data transfer of triple data to GPU memory incurs. However, this process is performed only once for each dataset and the transformed data are saved for future use.

Query Parser

Simple SPARQL queries may contain only one subquery. In Listing 2, there is only one subquery: Who knows Alice. The free variable is *?who* which is the subject while the bounded variables are `knows` and `Alice`.

Listing 2: Simple query

```
1 SELECT ?who
2 WHERE {
3 ?who  knows  Alice . }
```

Some query can consist of more than one subquery, and there can be more one free variables such as:

Listing 3: Query with multiple subqueries

```
1 SELECT ?x ?y
2 WHERE {
3 ?x founder ?y .
4 ?y isA Restaurant . }
```

In Listing 3, there are two subqueries: x is the founder of y and y is a restaurant. The free variables are *?x* and *?y* which are the subject and object of the first subquery and the subject of the second subquery respectively. The bounded variables are `founder`, `isA`, and `Restaurant` which are predicate, predicate, and object respectively.

Each subquery has a set of triple-IDs that are matching results. The first subquery extracts the set of subject/object pairs that has predicate **knows**. The second one will return the set of subjects that are the founder of **YumYum**. The relational join is used to combine the results, resulting in only rows of the first and second result that has the same *?y*. For a SPARQL query that consists more than 2 subqueries, the optimization may consider the order of joins to reduce the intermediate results as inputs to the next join operation. We will discuss how to order them in the query planner section.

In our notation, a SPARQL query Q consists of l free variables and k subqueries, $Q = (SV, SQ)$ where $SV = \{?x_1, ?x_2, ..., ?x_l\}$ and $SQ = \{sq_1, sq_2, ..., sq_k\}$. Real-life SPARQL query can contain any number of free variables and subqueries. In our case, we assume our subquery $sq_i = \langle e_1, e_2, e_3 \rangle$ consists of 3 elements e_1, e_2 and e_3 where e_1, e_2 and e_3 are free variables or *id*.

The subquery returns an intermediate result $R = (V, T)$ where V is a list of free variables $\langle ?x_1, ?x_2, ..., ?x_m \rangle$ and T is a list of t that is sorted by variable $?x_1$. To construct the intermediate result R from subquery, there are many ways to select the index to be used. To use the different index, the order of free variables is changed correspondingly. The proper index should be selected to reduce the overall processing time.

The query parser converts the given SPARQL query to an internal format. For implementation, the open source Redland's Rasqal [32] is used to parse SPARQL queries. For query $Q = (SV, SQ)$, we will store the free variables in SV and all subqueries SQ to use in the next query planner and executor. For each subquery sq_i, the bounded variable will be converted to *id*. For example, the query in Listing 3 will have $SV = \{?x, ?y\}$ and $SQ = \{\langle ?x, 11, ?y \rangle, \langle ?y, 12, 6 \rangle\}$

Query Planner

The query planner takes the query in a triple-ID form obtained after parsing. It analyzes and creates a sequence of operations for execution. There are 3 basic operators used in VEDAS framework.

1. **Upload**: The operator uploads intermediate result R_i from subquery sq_j to GPU memory. It also indicates the index of $?x$ used in subquery sq_j.
2. **Join**: The process that combines the results R_i and R_j to another result R_k. The total column number of R_k maybe greater than the total column number of R_i and R_j.
3. **Index swap**: We use the sort-merge join as the only one join method, requiring that the first variable of both V_1 and V_2 must be the same. Index swap is an operator that swaps the order of V_i for preparing for the next join operation.

Our assumption is that the operators in all subqueries are processed in a sequential fashion and the join operator is a binary join; not a multi-way join. The results of each operator form an intermediate result R_i if the operator is processed at step i. All operators have a cost. The upload operator cost is the transfer time from the host memory to the GPU memory. Joining and index swapping are operators processed on the GPU. They are also not as fast as simple processing task. For a given query Q, the query planner component creates the order of the above 3 operators to

construct the final query result. The process order of operators is directly impact to the performance of querying. If the order is well arranged, the number of index swap operators can be decreased which can increase the performance. However, sometimes we can increase the number of index swap operators on small intermediate results to decrease the number of join operators for a large data set. However, the query planning problem is known to be NP-hard.

In this paper, we assume to use a manual static scheduler to mange the order of operations. The strategy is to interleave the upload and join operations if possible. The order of upload and join operations is determined by the triple pattern based on the SPARQL query. This approach constructs the good enough left-deep plan for evaluation the framework.

Query Executor

After obtaining the order of operators from the query planner, the query executor executes the sequence of operators and stores the intermediate results in the GPU memory. The final results are transferred from the GPU memory back to the host after finishing all operators. Query executor contains several subcomponents according to the basic operators supported in the previous section.

Upload Operator

The upload function $U(D, sq)$ copies of intermediate results of subquery sq to GPU memory. Let D denote the set of all indices of the dataset: so $D = \{D_{POS}, D_{PSO}, D_{OPS}, D_{OSP}, D_{SPO}, D_{SOP}\}$. We use F_{sq} for the set of free variables of subquery sq and B_{sq} is set of bounded variables of sq. The upload function U returns $R = (V, T)$ where $V = \langle ?x_1, ?x_2, ..., ?x_m \rangle$ is the tuple of free variables in sq and T is the list of tuples returned for query sq. In reality, $|F_{sq}|$ is only 1 or 2. The upload algorithm is shown in Algorithm 1.

Algorithm 1 Upload Function U.

Input: D, sq
Output: $R = (V, T)$
1: $Index \leftarrow$ Select index to use from D ▷ Choose by query planner
2: $LowerOffset, UpperOffset \leftarrow$ Calculate data offset of sq from B_{sq}
3: Tighten $LowerOffset$ and $UpperOffset$ with the variable bound
4: $T \leftarrow$ triple-IDs matching with sq from range $LowerOffset$ and $UpperOffset$
5: Upload T to GPU memory
6: **return** (F_{sq}, T)

First, bounded variables in B_{sq} are used to identify indices to be used. For example if $sq = \langle ?z, 4, 5 \rangle$, the datasets that are indexed by predicate/object (D_{POS}) and object/predicate (D_{OPS}) can be used. Because we store the triple in the column-oriented fashion, we can upload only the related columns instead of all columns. The columns to be uploaded is only the column of free variables that are matched from the index. The resulting consecutive rows are selected based on the range *LowerOffset* to *UpperOffset* (Lines 2-4). In Line 3, *LowerOffset* and *UpperOffset* can be compact based on the information from other triples or after joining the results. The filter technique to filter these tuples will be described in Pre-Upload Filtering.

?x	?y
1	3

(a) Result for sq_1

?y
3

(b) Result for sq_2

Figure 8: Subquery result example.

For the example in Listing 3, $SQ = \{\langle ?x, 11, ?y\rangle, \langle ?y, 12, 6\rangle\}$. Assume the returned tuples of sq_1 and sq_2 are $\langle 1, 11, 3\rangle$, $\langle 3, 12, 6\rangle$ respectively. The results are $\langle 1, 3\rangle$ and $\langle 3\rangle$. Figure 8a and Figure 8b show the result of sq_1 and sq_2 in the triple-ID format.

For sq_1, if we use the index D_{PSO}, the data contains 2 columns and sorted by the subject (or column of $?x$). If D_{POS} is used, the data also contains 2 columns but are sorted by the object (or $?y$). In this case, D_{POS} is selected because the results sorted by $?y$ can immediately be joined with results from sq_2 that are also sorted by $?y$. The subquery sq_2 can use both D_{OPS} and D_{OSP} and obtain the same results.

Join Operator

The results from subqueries are usually joined together. Let $R_i \bowtie R_j$ denote the join of intermediate results from operators i and j respectively. Let $R_k = R_i \bowtie R_j$ be the join results. R_k composes of (V_k, T_k). Hence, Equations 1-2 show the new size of V_k and T_k

$$|V_k| = |V_i| + |V_j| - |V_i \cap V_j| \tag{1}$$

$$|T_k| \leq |T_i||T_j| \tag{2}$$

Let $\pi_i(T)$ be the data i^{th} column from T. For example $R = (V, T)$ with $|V| = s$, we denote the first column data of T by $\pi_1(T)$ and denote $\pi_s(T)$ for the last column. Algorithm 2 presents the join of R_i and R_j, where R_i has r free variables and R_j has s free variables.

Line 1 combines all the variables. In Line 2, the system applies the inner join to the first column of T_i and T_j. The first column is always sorted. The modern GPU's sort-merge join [33] will be used for inner join in this step. The join process will also get the index of rows that the first column matched to another intermediate results. After joining, the number of rows of the results is $|T_k|$. We allocate the memory on the GPU with size $|T_k| \times |V_k|$. Lines 3-4 collect the rows in T_i and T_j that correspond to the row index of the inner join processed by Line 2. The data will be merged into new data tuple T_k. Line 5 updates the variables storing the bound used in the pre-upload filtering phase.

Figure 9 shows an example of the join operation. In Figure 9a, R_i has two free variables $?x, ?y$ and in Figure 9b, R_j, has three free variables $?x, ?z, ?w$. The resulting join R_k has free variables $?x, ?y, ?z$ and $?w$ whose triple results are as in Figure 9c.

Algorithm 2 Join Algorithm.

Input: R_i with r variables, R_j with s variables
Output: R_k with t variables

1: $joinVarCount \leftarrow |V_i \cap V_j|$
2: $V_k \leftarrow \langle V_{i1}, V_{i2}, ..., V_{ir}, V_{jjoinVarCount+1}, ..., V_{js} \rangle$
3: Inner join $\pi_{1-joinVarCount}(T_i)$ and $\pi_{1-joinVarCount}(T_j)$ and obtain ($value$, $matched\_index$) from inner join.
4: Copy $\pi_{joinVarCount-r}(T_i)$ columns of the $matched\_index$ row to T_k.
5: Copy $\pi_{joinVarCount-s}(T_j)$ columns of the $matched\_index$ row to T_k.
6: Update the variable storing the bound.
7: **return** $R_k = (V_k, T_k)$

$?x$	$?y$
1	5
2	3
2	7
4	5
6	2

(a) R_i

$?x$	$?z$	$?w$
2	10	8
6	9	11
7	9	17

(b) R_j

$?x$	$?y$	$?z$	$?w$
2	3	10	8
2	7	10	8
6	2	9	11

(c) $R_k = R_i \bowtie R_j$

Figure 9: Join example.

Pre-Upload Filtering

To reduce the number of tuples to upload to the GPU memory, we bound the number of results before uploading. This is done by this preliminary filtering, called *Pre-Upload Filtering* phase. Suppose we have 2 intermediate results R_i and R_j, let $?w$ be first free variable in T_i. Suppose id of $?w$ is ranged from 1052 to 2654 for R_i and for R_j, it contains $?w$ with range 1548 to 3654. We keep the bound of minimum and maximum id as $(1052, 2654)$ for R_i and $(1548, 3654)$ for R_j. For each tuple that contains id, $t_i \in T_i$ and $t_j \in T_j$, t_i or t_j will not be considered in the resulting set, e.g., the variable id whose value is less than 1548 or greater than 2654, will be discarded.

To keep the bound for each variable, we construct the dictionary for each variable that keeps the boundary if each variable id (minimum and maximum). We denote $B(R_i)$ is a pair of minimum id and maximum id of $\pi_1(T_i)$.

Some variables may occur more than one time in query Q. Before uploading and after each join process, we update the bound $B(R_k)$. This pre-upload filter can reduce the data required to transfer to the GPU memory.

Index Swap Operator

In some case, for a given R_i and R_j, the first variable of both may not be the same, which prevents the join operation. The index swap operation is the operation for swap the variables from some other column to be in the first one and sort them afterwards. The purpose of this operator is to make it possible to join.

Let $S(R_i, ?x)$ be an index swap function that swaps variable $?x$ to be the first position in V_i list and sort tuple T_i with $?x$ column.

Figure 10 is an example of swapping variable $?z$ in the tuples. Figure 10b shows resulting tuples after sorting by $?z$ based on Figure 10a.

The cost of index swapping is the cost to sort $|T|$ tuples with $|V|$ elements on the GPU. The index swap in early stage may be time consuming while perform-

Algorithm 3 Index Swap Algorithm.

Input: Intermediate result R, Variable to swap $?x$
Output: Intermediate result R' with the first variable, $?x$

1: Swap first column and $?x$ of T.
2: Swap first variable and $?x$ in V.
3: Sort tuple in T with the new first column.
4: **return** $R' = (V, T)$

$?x$	$?y$	$?z$	$?w$
7	6	5	12
7	8	1	4
11	4	5	8
15	9	12	3
13	2	7	2
13	7	5	5
19	6	1	7

(a) Before swapping $?z$

$?z$	$?x$	$?y$	$?w$
1	7	8	4
1	19	6	7
5	7	6	12
5	11	4	8
5	13	7	5
7	13	2	2
12	15	9	3

(b) After swapping $?z$

Figure 10: Swap index by variable $?z$.

ing it in the later stage may be faster due to few number of columns and tuples. Note that in thrust library, the parallel sort complexity is $O(Nlog(N)/p)$. Hence, $O(|T|log(|T|)/p)$ for p threads.

The star-shaped query is frequently found in SPARQL queries. It is a pattern of queries that has one node with high degrees. A SPARQL query also can contain many star-shaped patterns in a query. This shape pattern prevents us to use the index swap because it uses one variable to join many tuples. On the other hand, linear-shaped query forces us to use the index swap operators many times.

Exploratory Subquery

For query $Q = \{sq_1, sq_2, ..., sq_k\}$ that has at least one subquery $sq_i = \langle e_1, e_2, e_3 \rangle$ where e_1, e_2 and e_3 are free variables. This subquery is called an *exploration subquery*. Such exploratory subquery requires to upload all triples in the data set since all three are free variables. However, with the help of pre-upload filter for given id, we bound the id values to reduce the number of tuples which is to reduce the data transfer to the GPU memory and the index swapping time.

Figure 11 summarizes the overall activity of VEDAS framework. We describe the work done in both host and GPU sides. The SPARQL query is first parsed into SV and SQ. Next, the query planner is constructed. For each subquery sq_i, the system will find the proper index and use pre-upload filter to throw away the out of range tuples before uploading the remainder to the GPU. The upload and join operators can be interleaved. This scheme will help to tighten the bound of free variables before uploading. Upon the completion of all upload and join tasks, the final result will be downloaded back to host memory. The final result contains only ids from related tuples, therefore dictionary mapping and decoding steps are necessary to transform them back to the original forms.

Example

Consider the RDF data in Figure 3. Figure 4 is based on the data D. At the initialization, VEDAS creates a dictionary used to hash to transform the terms to

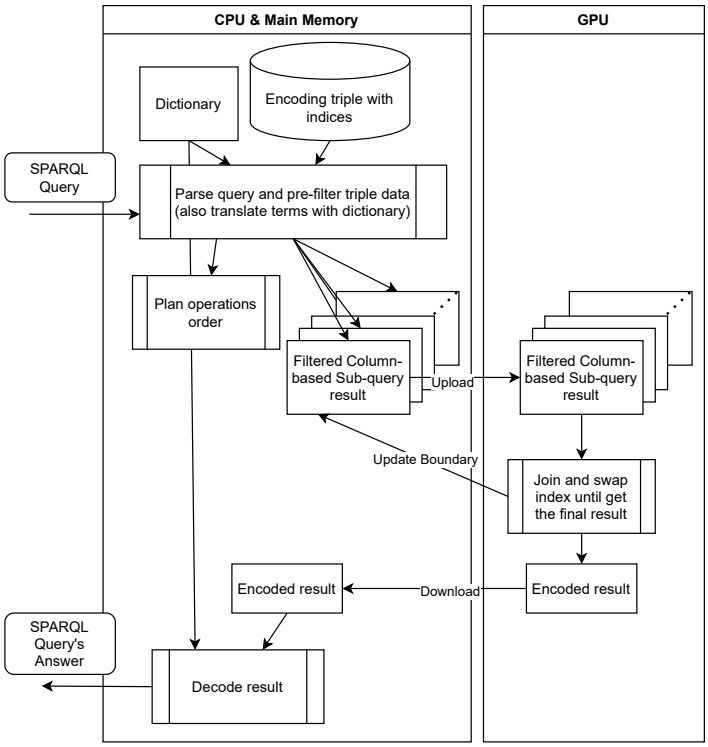


Figure 11: VEDAS's activity diagram to process the SPARQL query.

*id*s. In Figure 6b, the converted triple-ID along with the dictionary (Figure 6a) are shown. The indexer performs the sorting on the converted triple-ID to create 6 permuted indexed data. This step will get the indexed column-based triple-ID data with 6 indices i.e. D_{POS}, D_{PSO}, D_{OPS}, D_{OSP}, D_{SPO}, and D_{SOP}. The example of sorted triple-ID with SOP index and column-based data shown in Figure 7. All components and data in this initialized step are processed on the host side.

Listing 4: Example query

```
1 SELECT ?x ?y
2 WHERE {
3 ?x birthPlace India .
4 ?x isA Person .
5 ?x founder ?y .
6 ?y isA Restaurant . }
```

The data set D in the previous example is suitable for explanation but it is too small to use for the query example. From this point, we will assume that D is much larger than the previous example.

The user sends for SPARQL query Q as in Listing 4. The query Q has 4 subqueries $sq_1 = \langle ?x, 1522, 9483\rangle$ (Line 3 in Listing 4), $sq_2 = \langle ?x, 12, 6173\rangle$ (Line 4 in Listing 4), $sq_3 = \langle ?x, 11, ?y\rangle$ (Line 5 in Listing 4)and $sq_4 = \langle ?y, 12, 6\rangle$ (Line 6 in Listing 4). Suppose the planner arranges the operations as the following.

1 $R_1 = U(D, sq_1)$
2 $R_2 = U(D, sq_2)$

3 $R_3 = R_1 \bowtie R_2$
4 $R_4 = U(D, sq_3)$
5 $R_5 = R_3 \bowtie R_4$
6 $R_6 = S(R_5, ?y)$
7 $R_7 = U(D, sq_4)$
8 $R_8 = R_6 \bowtie R_7$

The execution plan tree is shown in Figure 12. Operations 1 and 2 use D_{POS} (or D_{OPS}) indices and upload them. The intermediate result R_1 and R_2 has only 1 free variable $?x$ ($V_1 = V_2 = ?x$). For the operation 4, we can upload only D_{PSO} and the result R_4 has two variables ($V_4 = ?x, ?y$). If $?x$ id of R_1, R_2 and R_4 is ranged from $(1, 1347)$, $(35, 1998)$ and $(48, 1595)$ respectively. When uploading the R_1 and R_2 to GPU memory, it can filter out $?x$ that is not in the range $(max(1, 35, 48), min(1347, 1998, 1595))$ or $(48, 1347)$ by using such bounding. Notice that these ranges can be known before uploading from the index data. Figure 13 shows the intermediate result for each operator, the result R'_i shows the intermediate result R_i if we do not apply pre-upload filter.

The operation 3 performs the join on the GPU. The result R_3 contains only 1 column. Suppose the join result data range is $(48, 934)$. This range is sent back to the host to update $?x$'s bound to $(48, 934)$. Therefore, operation 4 will use the new bound to determine the data transferred to the GPU memory. Even though the intermediate result R_3 and R_4 contains the different columns but it can be joined with the common variable $?x$.

The result R_5 has 2 columns $V_5 = \langle ?x, ?y \rangle$. To join with R_7 where $V_7 = \langle ?y \rangle$, $?y$ must be moved to first column by operation 7. After swapping the index with $?y$, we can join R_6 with R_7 and get the 2-column tuples leading to the join in operation 8.

The final result is transferred back to the host and it is decoded using the dictionary to obtain the terms back in the original form.

Figure 13 presents a numerical example of operations 1-8. Let Figures 13a and 13c be the intermediate results obtained from operations 1 and 2 that corresponds to sq_1 and sq_2. These results are not yet applied the pre-upload filter. Applying the pre-upload filter, in Figure 13b and Figure 13d, the bound $(48, 1347)$ is used. The tuples whose id of $?x$ valued less than 48 or greater than 1347 are eliminated and will not be uploaded to the GPU memory. The intermediate results of $R_1 \bowtie R_2$ are shown in Figure 13e. It consists of ids which are in both R_1 and R_2. Similarly, Figure 13f presents the uploaded tuples from the results of sq_3. After filtering with bound $(48, 934)$ with pre-upload filter, Figure 13f shows the reduced tuples. In this step, the bound is updated due to R_3. The intermediate result R_5 in Figure 13h is obtained from the join of R_3 and R_4. The output has 2 columns $?x$ and $?y$. Because R_5 cannot be joined with R_7 that is indexed by free variable $?y$, it is required to swap the index, resulting R_6 in Figure 13i. The bound is also updated to $(89, 900)$. Instead of upload R'_7 in Figure 13j, the tuples are filtered as in Figure 13k, for R_7. R_7 is then uploaded. The final result is shown in Figure 13l which is obtained from $R_6 \bowtie R_7$, i.e., selecting only matched $?y$ and adding the corresponding $?x$ in same tuple.

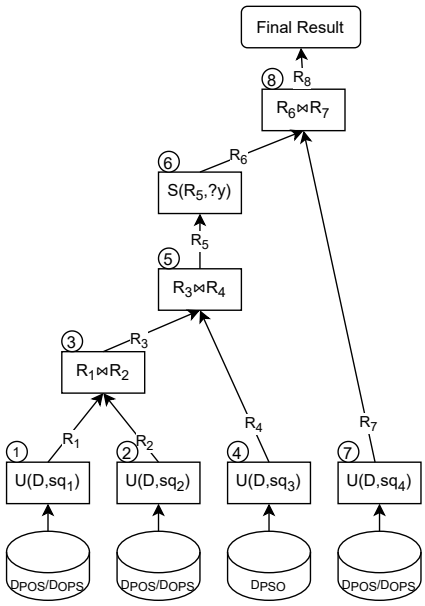


Figure 12: Execution plan for example query.

?x
1
39
42
45
48
50
65
78
730
934
1347

(a) R'_1

?x
48
50
65
78
730
934
1347

(b) R_1

?x
35
42
48
50
65
199
730
934
1322
1768
1998

(c) R'_2

?x
48
50
65
199
730
934
1322

(d) R_2

?x
48
50
65
730
934

(e) R_3

?x	?y
48	101
50	900
65	89
400	203
592	392
730	105
1029	223
1488	2093
1595	2123

(f) R'_4

?x	?y
48	101
50	900
65	89
400	203
592	392
730	105

(g) R_4

?x	?y
48	101
50	900
65	89
730	105

(h) R_5

?y	?x
89	65
101	48
105	730
900	50

(i) R_6

?y
50
101
800
900
934

(j) R'_7

?y
101
800
900

(k) R_7

?y	?x
101	48
900	50

(l) R_8

Figure 13: Intermediate result example of Listing 4.

Experiments

We compare VEDAS with gStore [18] and RDF-3X [34, 35] which are the state of art for open source RDF stores. The storage size and query processing time are measured. WatDiv [36] test suite and LUBM [37] are used as benchmarks. WatDiv is SPARQL benchmark that has different query structures and workload sizes. The generated queries have 4 categories: linear queries (L), star queries (S), snowflake-shaped queries (F) and complex queries (C).

The experiments are done on the system with the following specification: 64 CPU of Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 256 GB Memory. The system contains 4 NVIDIA Tesla V100s 32GB memory with CUDA 10.1. Our source code is implemented with C++ and NVIDIA thrust library [31]. Modern-GPU library (sort-merge join) [33] is used in the joining process. We use only 1 GPU in the experiments.

We also compare the two options of VEDAS including on-demand upload and pre-upload. The on-demand option is to upload the triple data to the GPU memory based on the filter and selection method described in Section VEDAS Framework and Operations indicated by the subquery. That is it uploads the triple-ID only when needed. The pre-upload option uploads all triple data (excludes indices) into the GPU memory. When a part of triple data is needed, the necessary rows will be moved to another place on the GPU memory. This results in the reduction in the upload time while increasing the storage used in the GPU memory. The performance of both options are compared while considering the pre-upload filtering algorithm and on GPU memory approach. At last, the analysis on the computing time and data transfer are performed. We demonstrate the query cases where the speedup can be gained. We also explain how the query planner can help improve the speedup.

Storage size

First, we compare the size of data files that each framework generated as a measurement. RDF-3X compresses all data into one file. gStore has many files for each dataset. For VEDAS, it has .vdd file for storing dictionary data and .vds file for storing triple data. We calculate the total size of all these files. Table 3 shows storage used in bytes for each framework for WatDiv dataset 100M, 300M and 500M N-Triples. VEDAS occupies the size less than RDF-3X and much less than gStore.

Table 3: Storage size (bytes).

	.nt (Raw Triple)	RDF-3X	gStore	VEDAS
100M	15 G	5.1 G	8.3 G	5.23 G
300M	45 G	17 G	27 G	15.67 G
500M	75 G	30 G	45 G	26.1 G

Query Time

WatDiv has 20 queries which are categorized to 4 patterns: C, F, S and L. For RDF-3X and gStore, the query time for each query excluding the loading data to memory time are measured. Table 4 presents the query time for each query. The summation of each query time in each category is shown in each query class time in Table 5 for each work and query class for each N-Triple sizes. Table 6 displays the

speedup of VEDAS query times (both on-demand and pre-upload cases) compared with that of RDF-3X and gStore.

NVIDIA nvprof is used to profile the proportion of computation time and upload time for the case of 300M data. The values are reported in Table 7. We also summarize the intermediate result size of each operation (upload, join and index swap) to compare with proportion of computation time and upload time.

Figures 14, 15 and 16 compare the query time for each query class for all RDF-3X, gStore, and VEDAS. VEDAS obtains the speedup more especially on class C. This is because the intermediate result after the join operation is large; therefore, the GPU performs better in this case (see Table 7). Although almost of queries can have benefit from GPU, there are some queries that perform far better than the other systems e.g. C2, L2, L4 and L5. The query time and number of results for these queries are shown in Table 4 and Table 8.

For L2, L4, and L5, the computation of the queries take a lot of proportion compared to the upload time. The upload data are small; therefore, these queries are suitable for GPU. C2 query also has a high join-upload ratio (computation-transfer ratio), but not high as L2, L4 and L5. This query has a high computation work, i.e., it has 1M data for joining and 0.7M for sorting. Consequently, this will lead to the significant computation time on CPU.

For S1, it is the query that VEDAS can slowly process. The S1 query has the lowest computation-transfer ratio on GPU. This query has 17.5M rows for uploading and has only 183 rows for joining (in WatDiv300M). It performs a little bit slow when compared to the CPU approach whose cost for uploading data is lower.

In overall, the GPU approach is superior for the queries with many joins and high computation-transfer ratio queries. For queries with large upload data and that do not have lots of join, it will yield a small speedup. This leaves a room us to optimize the upload time and reduce intermediate result size using a better query planner in the future.

Table 4: Query time of RDF-3X, gStore and VEDAS.

	C			F					S							L				
Framework - Size	**C1**	**C2**	**C3**	**F1**	**F2**	**F3**	**F4**	**F5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**L1**	**L2**	**L3**	**L4**	**L5**
RDF-3X - 100M	44	383	59	8	28	63	43	56	12	45	2	3	3	7	4	25	12	5	6	19
gStore - 100M	338	724	26769	17	15	21	17	33	13	76	27	29	4	6	2	3	7	4	13	61
VEDAS - 100M	7	18	46	5	5	8	8	11	16	4	3	3	1	2	1	3	1	2	1	1
VEDAS (Preload) - 100M	5	12	46	4	4	6	7	7	11	3	2	2	1	2	1	2	1	2	1	1
RDF-3X - 300M	115	860	186	13	57	98	126	153	15	76	10	5	11	15	5	42	29	5	15	25
gStore - 300M	1373	2571	12100	26	16	220	34	523	11	203	62	62	6	10	5	5	47	3	25	181
VEDAS - 300M	14	41	134	10	10	19	17	28	38	9	5	5	2	4	1	6	3	5	1	2
VEDAS (Preload) - 300M	10	29	135	6	8	12	12	17	23	5	3	3	2	3	1	4	3	4	1	2
RDF-3X - 500M	130	4617	263	19	90	301	328	349	20	182	7	30	8	11	3	103	27	3	16	43
gStore - 500M	1599	3996	35720	31	26	263	47	570	8	312	79	94	8	17	2	4	11	5	42	171
VEDAS - 500M	24	66	227	16	17	29	27	43	56	13	7	8	4	11	1	11	4	10	1	4
VEDAS (Preload) - 500M	15	45	226	11	16	17	18	25	34	7	4	5	3	6	1	6	3	6	1	3

Table 5: Query time of RDF-3X, gStore and VEDAS in milliseconds.

	Query Time - 100M (ms.)				Query Time - 300M(ms.)				Query Time - 500M(ms.)			
Framework	**C**	**F**	**S**	**L**	**C**	**F**	**S**	**L**	**C**	**F**	**S**	**L**
RDF-3X	486.33	197.67	76	67	1161.66	446.67	122	115.67	5010.66	1088.32	260.67	191.67
gStore	27831.66	103.33	156.34	87.01	16043.67	819.99	359	261	41315	938.32	520	232.34
VEDAS (On-demand)	70.1	37.38	30.02	8.12	189.63	83.03	62.69	17.35	316.69	131.46	98.13	30.41
VEDAS (Pre-upload)	62.96	28.17	22.24	6.22	174	54.95	39.94	13.77	284.66	87.44	60.15	19.68

From the result, it is obvious that VEDAS with pre-upload gains more speedup that VEDAS on-demand. However, the on-demand case still has a close performance to the pre-upload case. It turns out that the on-demand approach is practical since it can save a lot of GPU memory with a little increment in processing time. When adding the pre-upload filter, it can improve a lot of performance.

Table 6: Speedup compared to RDF-3X and gStore.

Query Class	RDF-3X/ VEDAS (on-demand)	gStore/ VEDAS (on-demand)	RDF-3X/ VEDAS (Pre-upload)	gStore/ VEDAS (Pre-upload)
C (100M)	6.94	397.03	7.72	442.05
F (100M)	5.29	2.76	7.02	3.67
S (100M)	2.53	5.21	3.42	7.03
L (100M)	8.25	10.72	10.77	13.99
C (300M)	6.13	84.61	6.68	92.21
F (300M)	5.38	9.88	8.13	14.92
S (300M)	1.95	5.73	3.05	8.99
L (300M)	6.67	15.04	8.4	18.95
C (500M)	15.82	130.46	17.6	145.14
F (500M)	8.28	7.14	12.45	10.73
S (500M)	2.66	5.3	4.33	8.65
L (500M)	6.3	7.64	9.74	11.81

Table 7: Query processing time analysis for 300M data.

Query	Transfer	Compute	Upload	Join	Index Swap	Time (ms.)
C1	13.43%	86.55%	10.8M	1.5M	0.2M	14.12
C2	13.99%	85.99%	17.5M	1.0M	0.7M	41.39
C3	26.73%	73.26%	144.9M	2.2M	0	134.12
F1	12.37%	87.56%	4.0M	0.1M	5187	9.84
F2	6.39%	93.53%	3.8M	0.4M	37	9.66
F3	15.41%	84.56%	10.3M	0.1M	151	19.02
F4	10.61%	89.33%	6.9M	1.0M	1448	16.94
F5	24.10%	75.87%	15.6M	1.7M	42	27.57
S1	34.62%	65.38%	17.5M	183	0	37.78
S2	11.34%	88.66%	3.8M	0.8M	0	8.5
S3	12.35%	87.65%	1.9M	0.2M	0	4.52
S4	16.01%	83.99%	2.2M	4956	0	4.87
S5	5.36%	94.64%	0.7M	0.1M	0	2.45
S6	30.07%	69.94%	2.0M	92	0	4.19
S7	1.47%	98.53%	3	0	0	0.38
L1	31.00%	69.00%	3.2M	12	11	5.63
L2	5.10%	94.90%	0.7M	22863	21458	3.41
L3	33.70%	66.30%	3.1M	11	0	4.94
L4	1.75%	98.25%	84017	874	0	0.88
L5	3.27%	96.72%	0.7M	32075	30043	2.49

Table 8: Intermediate result sizes.

	C			F					S							L				
Triple size	C1	C2	C3	F1	F2	F3	F4	F5	S1	S2	S3	S4	S5	S6	S7	L1	L2	L3	L4	L5
100M	0	39	8004	33	9	11	156	53	5	2637	0	0	0	11	1	0	624	30	297	846
300M	0	195	24235	3	37	52	319	37	14	8049	0	1	0	46	0	1	1302	11	862	2032
500M	0	110	39984	0	68	98	501	42	0	13454	0	3	0	75	1	0	669	2	1468	957

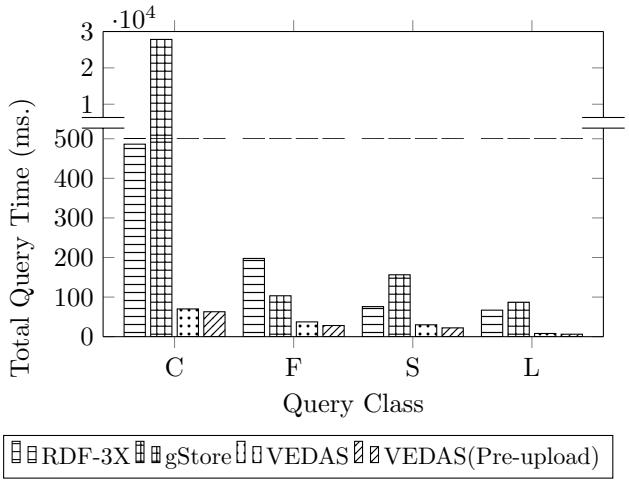


Figure 14: Query Time for 100M.

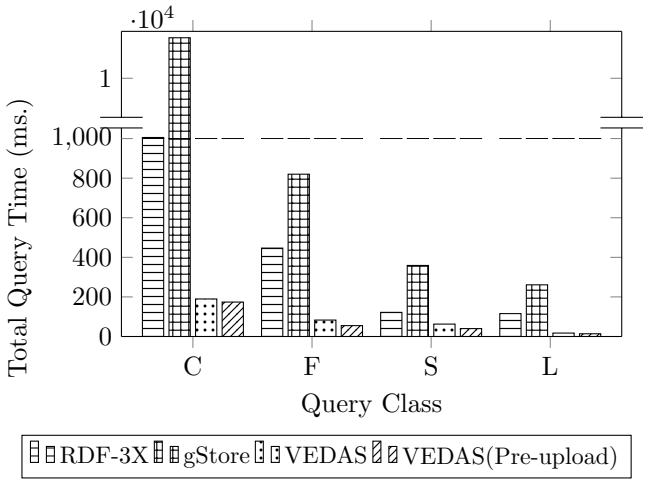


Figure 15: Query Time for 300M.

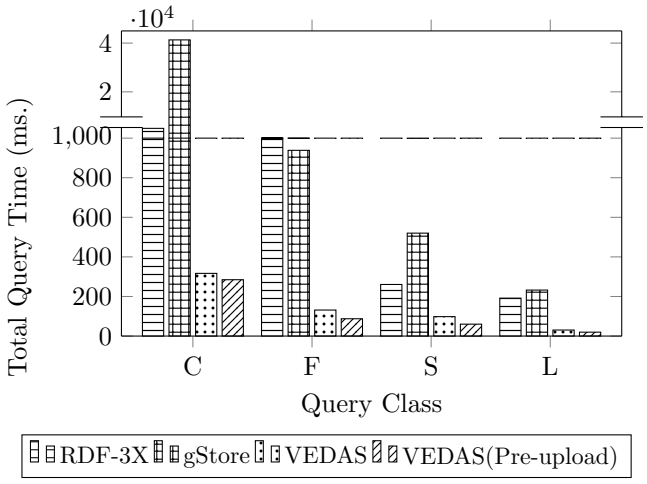


Figure 16: Query Time for 500M.

LUBM is synthetic benchmark like WatDiv. We generated 1024 universities dataset which is about 140 million triples. We consider 3 complex queries (L1, L2 and L7) and 1 simple query (L4) in [38] and [39]. L1 and L7 are queries containing cycles and they require the join with 2 variables. The queries have large intermediate result size but small final result size. L4 is a star query with high selectivity. Finally, L2 is 2 triples low selectivity query. The query times for each case and the number of result rows are shown in Table 9. The speedup of VEDAS compared to RDF-3X and gStore are shown in Table 10.

VEDAS runtime for query L1 is notably faster than that of RDF-3X and gStore. The reason is that L1 is high-computation query with large IR join. Therefore, it can use the advantage of GPU architecture. For L7 query, the join graph shape is same as L1. However, the number of data for joining are smaller and there are larger data for uploading. As a results, this case gains speedup less than L1. L2 query is the simple join with 2 triple data. The speedup is high since the join output is low

selectivity and requires high computation. For L4, it is high selectivity and requires to upload triples many times. This causes GPU processing gain less efficiently.

Table 9: Query time for LUBM benchmark of RDF-3X, gStore and VEDAS in milliseconds.

	Query time (ms.)			
Framework	**L1**	**L2**	**L4**	**L7**
RDF-3X	89,667.4	1,790.0	4.0	4207.0
gStore	9,760.2	987.4	1.4	952.4
VEDAS (On-demand)	154.9	22.7	1.9	128.7
VEDAS (Pre-upload)	137.6	16.4	1.2	102.9
Result size	2,528	1,106,277	10	45,222

Table 10: LUBM benchmark's speedup compared against RDF-3X and gStore.

Query Class	RDF-3X/ VEDAS (on-demand)	gStore/ VEDAS (on-demand)	RDF-3X/ VEDAS (Pre-upload)	gStore/ VEDAS (Pre-upload)
L1	578.57	62.97	651.55	70.92
L2	78.85	43.49	109.29	60.29
L4	2.01	0.70	3.31	1.16
L7	32.70	7.40	40.88	9.25

Table 11 shows the speedup of VEDAS and MAGiQ [23]. They are compared against RDF-3X. MAGiQ uses Matlab matrix library for GPU to process queries that is tuned and optimized for matrix processing. MAGiQ's experiment uses LUBM-10240 and difference CPU and GPU specification, therefore we cannot compare the speedup directly. However, it can be seen that our speedup are better or close to the MAGiQ result with the smaller dataset.

Table 11: LUBM benchmark's speedup of VEDAS and MAGiQ compared against RDF-3X.

Query Class	RDF-3X/ VEDAS (on-demand)	RDF-3X/ VEDAS (Pre-upload)	RDF-3X/ MAGiQ
L1	578.57	651.55	429.84
L2	78.85	109.29	73
L7	32.70	40.88	37.89

Effect of Data Transfer

Table 12 shows the time percentage of each operator for each query for the 500M data size case. It shows that about 50% queries take at least 50% of time to upload the data. The slow queries are because such query uses too much upload time. Thus, to improve the system in the future, the upload process should be further optimized. The proper data can be selected to be uploaded and forced to reside in the GPU memory for reuse several times. Other approach is to use the pinned host memory or increase the page size to reduce the transfer time.

From the table, the join process time is also significant. There are some query types whose join operations take more than 60% of time, e.g., S5, and S7 which require insignificant upload time. For other cases, the join process takes time at the second place after the upload time. Also, the join process is usually after the filter process. The more the data can be filtered out, the less time used for the join operation since the time used for the join operation is determined by input relation sizes and the intermediate result size.

Because the data is large, GPU memory allocation and copy are also the time consuming process. It takes about 3% - 36% of query time. The memory usage optimization can be done by reusing the large pool of pre-allocated memory.

Table 12: Time percentage of each operation (500M datasize).

	C			F					S							L				
Time (Percentage)	**C1**	**C2**	**C3**	**F1**	**F2**	**F3**	**F4**	**F5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**L1**	**L2**	**L3**	**L4**	**L5**
Upload	59	54	89	57	14	63	45	59	69	37	30	45	0	76	0	71	25	79	0	13
Join	28	29	7	25	56	23	33	24	12	38	43	30	78	11	75	13	36	6	69	52
Index Swap	2	4	0	1	0	0	0	0	0	0	0	0	0	0	0	1	2	0	0	3
Download	0	0	0	0	1	0	1	0	0	5	0	1	0	0	4	0	2	0	9	2
Memory Allocate/Copy	11	14	3	18	30	14	21	16	19	20	27	24	23	13	21	15	36	15	22	29

Effect of Query Planner

In this section, we will show that VEDAS has a potential to enhance performance by improving the query planner component. As explained in Subsection Query Planner, our planner is the normal approach which considers from the leftmost subquery and join in the left-deep fashion. The bound of variables are updated after each join operation to reduce upload data. Since this plan order the join by the left-to-right triple order, it does not consider some existing order which can result in a small number of results and can filter a lot of data to upload.

Table 13 shows operation order for each plan type of query S5. U represents for upload operator, and the next 2 numbers are row and column sizes to upload. We use J for join operator and show the size of 2 intermediate results (IR) and output size. In the first column is the case where all matched triples are uploaded to the GPU memory and then they are joined as a result. The second column is the case where the left-deep join plan with left-to-right triple order. The third column is also left-deep join plan but it uploads the largest data first. These 3 plans query processing time are close to each other. The second plan that is used in VEDAS gets benefits from pre-upload filter a little bit. The last column is the plan where the upload of the 2 smallest data and then following with join results in 0 rows. Therefore, it is the fastest plan among 4 plans. This example shows that if we apply cardinality estimation to predict the intermediate result size for each join into query planner, we can select the best join order to process and reduce the data size of next related upload.

Table 13: Operation order for each query plan of S5 (300M).

Plan	Upload all then join	Interleave upload and join (order by triple pattern)	Interleaving upload and join (largest IR first)	Interleaving upload and join (smallest IR first)
Operation order	U 25k x 1 U 448k x 2 U 223k x 2 U 46k x 1 J 25k ⋈ 46k ⇒ 0 J 223k ⋈ 448k ⇒ 134k J 0 ⋈ 134k ⇒ 0	U 25k x 1 U 448k x 2 J 25k ⋈ 448k ⇒ 15k U 223k x 2 J 15k ⋈ 223k ⇒ 4k U 46k x 1 J 4k ⋈ 46k ⇒ 0	U 448k x 2 U 223k x 2 J 223k ⋈ 448k ⇒ 134k U 46k x 1 J 46k ⋈ 134k ⇒ 1k U 25k x 1 J 1k ⋈ 25k ⇒ 0	U 25k x 1 U 46k x 1 J 25k ⋈ 46k ⇒ 0 U 0 x 2 J 0 ⋈ 0 ⇒ 0 U 0 x 2 J 0 ⋈ 0 ⇒ 0
Query time (ms.)	4 ms.	3 ms.	4 ms.	1 ms.

Query C2 is one of the most complex query from WatDiv. It has 10 triple types from all subqueries and 6 variables to join; hence it will perform 10 upload operations and some index swap operations. This query can have numerous possible query plans and hard to find the optimal one. Table 14, shows the result of experiments with 3 query plans with operations summation and query times. The first plan (first column) separates the triple pattern into 2 groups, uses left-deep join for each group and join 2 groups at the final stage. The second column is like the previous one but separates into 3 groups which forms a star-shape. After processing each group, the subquery results are joined together. The last column plan is the complete left-deep

join plan that leads to many index swap operations. From the experiments, the third plan is the slowest one as expected. This is because it has a lot of index swap operations. The first plan uploads data more than the second one while the total query time is faster. The reason is the second one has more computation which dominates the query time. The query planner that considers the GPU operation cost can help select the efficient plan. The accurate CPU-GPU data transfer rate, latency and computation performance of each GPU model is required to improve the query plan.

Table 14: Operation order for each query plan of C2 (300M).

Plan	Separate into 2 subgroups and combine all later	Join each star-shape first and combine all later	Left-deep join
Operation order	U 3k x 1 U 4245k x 2 J 3k ⋈ 4245k ⇒ 434k U 294k x 1 S 434k x 2 J 294k ⋈ 434k ⇒ 38k U 1351k x 2 J 38k x 1351k ⇒ 38k U 88k x 1 U 86k x 1 J 86k ⋈ 88k ⇒ 4k U 4494k x 2 J 4k ⋈ 4494k ⇒ 11k U 4499k x 2 S 11k x 2 J 11k ⋈ 4499k ⇒ 11k U 4438k x 2 J 11k ⋈ 4438k ⇒ 68k U 223k x 1 S 68k x 3 J 68k ⋈ 223k ⇒ 3k S 3k x 3 J 3k ⋈ 38k ⇒ 201	U 88k x 1 U 86k x 1 J 86k ⋈ 88k ⇒ 4k U 4494k x 2 J 4k ⋈ 4494k ⇒ 11k U 2092k x 2 U 294k x 1 J 294k ⋈ 2092k ⇒ 379k U 1351k x 2 J 379k ⋈ 1351k ⇒ 379k U 3k x 1 J 3k ⋈ 379k ⇒ 38k U 4499k x 2 S 38k x 2 S 4499k x 2 J 38k ⋈ 4499k ⇒ 158k S 11k x 2 S 158k x 3 J 11k ⋈ 158k ⇒ 350 U 4438k x 2 U 223k x 1 J 223k ⋈ 4438k ⇒ 220k S 220k x 2 J 350 ⋈ 220k ⇒ 201	U 3k x 1 U 4245k x 2 J 3k ⋈ 4245k ⇒ 434k U 294k x 1 S 434k x 2 J 294k ⋈ 434k ⇒ 38k U 1351k x 2 J 38k ⋈ 1351k ⇒ 38k U 4438k x 2 J 38k ⋈ 4438k ⇒ 223k U 223k x 1 S 223k x 3 J 223k ⋈ 223k ⇒ 10k U 4499k x 2 S 10k x 3 S 4499k x 2 J 10k ⋈ 4499k ⇒ 70k U 4499k x 2 S 70k x 4 S 4499k x 2 J 70k ⋈ 4499k ⇒ 70k U 88k x 1 J 70k ⋈ 88k ⇒ 3k U 79k x 1 J 3k ⋈ 79k ⇒ 201
Summation	Total join 0.6M Total upload 19.7M Total index swap 0.5M	Total join 1.1M Total upload 17.5M Total index swap 4.9M	Total join 0.8M Total upload 19.7M Total index swap 9.7M
Query time (ms.)	37 ms.	39 ms.	45 ms.

Extension to Other Operations

Our current implementation focuses on the basic SPARQL namely `SELECT` (projection the selected variables) and `WHERE` (join). For advanced query forms, it can be implemented by the following guideline.

OPTIONAL

The `OPTIONAL` clause is equivalent to left join operator. The simplest scheme to handle this clause is to process all subqueries outside `OPTIONAL` clause first. Then the left join operation is applied.

For example, in Listing 5, the first and second patterns (`?person foaf:name ?name` and `?person foaf:age 40`) will be joined using the inner join. After that, the results will be joined with pattern in OPTIONAL clause (`?person foaf:homepage ?page`) with the left inner join.

Listing 5: OPTIONAL example query

```
1 | SELECT ?name ?page
```

```
2 WHERE {
3     ?person foaf:name ?name .
4     ?person foaf:age 40 .
5     OPTIONAL { ?person foaf:homepage ?page }
6 }
```

UNION

The disjunction or union is the operator that combines 2 intermediate result sets. The straightforward approach is to use set union operation with performs parallel in $O(n)$.

FILTER

FILTER may be the most challenge clause. It can contain the complex expression and high-level function like `regex`, `substr`, `strlen`, `concat` etc. We can insert some information in the encoded integer of *id* in a triple ID used for comparing the data such as inserting some bits to specify the datatype and the rest of bits is ordered corresponding to the raw value order. This scheme makes it possible to compare the literal types and values. For simple filter expression like `FILTER(?year` $>$ `2018)`, the value 2018 is encoded into the same format as *id*, `year`, and compared with the year values in the data store. We can bound the range of the filter variable *id* before uploading to the GPU memory.

Listing 6 shows SPARQL query with FILTER clause. In the example, it has a logical operator `and` (&&) that can handle by adding more constraints to the variable bound. The query rewriting may be applied to handle `or` operator (||). For example, it may convert the FILTER with `or` operator into a UNION clause.

Listing 6: FILTER example query

```
1 SELECT ?name1 ?y
2 WHERE {
3     ?x foaf:name  ?name1 .
4     ?x foaf:age   ?y .
5     FILTER (?y > 25 && ?y < 50)
6 }
```

The high-level string function requires some mechanism to store the raw string or other string data structure to process with such function. We can construct the component to process the string function and obtain the resulting *id*s. After obtaining the results, the inner join can be used to join the triple results with the filtered *id*s to get the final results.

ORDER BY

One property of VEDAS is that it always maintains the ascending order of the first column. Assume that all *id* order is the same as the literal order (by using technique in Subsection FILTER. If the variable in `ORDER BY` matched the first column variable, the result list can be obtained immediately when using `ASC`. For `DESC`, the result list is sorted reversely. For other order patterns, GPU can directly perform the parallel sort before it returns them to the user. We can consider the `ORDER BY` variables in a query planner. If the planner arranges the operator which matches with the desired result order, the processing time can also be reduced.

Conclusion and Future Work

RDF query processing involves large triple data processing which can be time consuming. This work demonstrates to handle SPARQL query utilizing the thousands of threads in the GPU. The suitable data representation must be considered to compact the data and reduce the data transfer between GPU and CPU while utilizing the parallel threads effectively.

We introduce the compact representation to store the triple data used in both in host and GPU memory. The framework for querying the triple data with SPARQL processing utilizing the GPU is proposed. The triple data are converted into indexed column-based called *triple ID*. The triple data are stored in the host main memory and are uploaded the GPU when the query processing requires. The pre-upload filter is designed to reduce the data size, minimizing the transfer time. The uploaded data can be quickly accessed by indices. Index swapping operation is introduced to enable the GPU sorting and merge join. Then, the query plan for ordering the combination of upload, join and index swap can be created.

The experiments show that our approach gains the speedup ranging from 1.95 to 15.82 compared to RDF-3X and ranging from 2.76 to 397.03 compared to gStore. It is also shown that the on-demand upload and the pre-upload approaches yield the similar execution time. Thus, using on-demand upload may be a good choice. The timing results show the implication of using our approach to improve the query processing time based on the GPU. The analysis demonstrates the query types that can gain benefits from our framework.

There are many ways that can further improve the performance such as: 1) To overcome the GPU memory limitation and scale out the processing power, the extension to multi-GPU is an attractive solution. 2) Planing the operator order and parallelizing the operator tasks also increase the efficiency. 3) In the pre-upload filter process, we see that it can make the query faster if the eliminated triples is increased. The hashing function that can maximize the filtered tuples may be considered. 4) The new join algorithm for this new representation is another problem that is very interesting.

Nowadays, the multiprocessor architecture is popular and the new accelerator types are emerging. Our work here focuses on the single NVIDIA GPU only but the technique and representation also can be generalized to other accelerator types as well.

Declarations

Ethics approval and consent to participate

Not applicable

Consent for publication

The authors give the Publisher the permission to publish the Work.

Availability of data and materials

VEDAS system source code is available at the author Github https://github.com/Remixman/Vedas.

Competing interests

The authors declare that they have no competing interests.

Funding

This work was supported in part by The Thailand Research Fund (TRF) under the Royal Golden Jubilee Ph.D. Program under Grant no. PHD/0171/2560. We also would like to thank NVIDIA hardware grant and ARES system from Kasetsart University providing hardware support for running the experiments.

Authors' contributions

Pisit Makpaisit contributed to the design and implementation of the research, the analysis of the results and to the writing of the manuscript. Chantana Chantrapornchai contributed to the overall direction and planning, provided critical feedback and helped shape the research, analysis and manuscript. All authors discussed the results and commented on the manuscript.

Acknowledgements

Authors would like to thank you Office of Computing Service at Kasetsart University for the utilization of the AI clusters.

Author details

Department of Computer Engineering, Kasetsart University, Bangkok, Thailand.

References

1. National Inventory of Natural Heritage: TAXONOMIC REPOSITORY TAXREF. https://inpn.mnhn.fr/programme/referentiel-taxonomique-taxref?lg=en. Online; accessed 20 October 2020
2. IMATI - CNR: LusTRE: Linked Thesaurus fRamework for Environment. http://purl.oclc.org/net/DumpEarthRDF. Online; accessed 20 October 2020
3. Gerasimos Razis: Influence Tracker Dataset. https://old.datahub.io/dataset/influence-tracker-dataset. Online; accessed 20 October 2020
4. Research Group Agile Knowledge Engineering and Semantic Web (AKSW): USPTO Patent Data. https://old.datahub.io/dataset/linked-uspto-patent-data. Online; accessed 20 October 2020
5. Wikipedia: DBpedia. https://en.wikipedia.org/wiki/DBpedia. Online; accessed 20 October 2020
6. Chantrapornchai, C., Choksuchat, C.: TripleID-Q: RDF query processing framework using GPU. IEEE Transactions on Parallel and Distributed Systems **PP**, 1–1 (2018)
7. W3C: N-Triples. https://www.w3.org/2001/sw/RDFCore/ntriples/. Online; accessed 20 October 2020
8. Salvadores, M., Alexander, P.R., Musen, M.A., Noy, N.F.: BioPortal as a Dataset of Linked Biomedical Ontologies and Terminologies in RDF. IOS Press (2013)
9. Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF.W3C Recommendation. (2008). http://www.w3.org/TR/rdf-sparql-protocol/
10. DCMI: Dublin Core Metadata Element Set, Version 1.1. http://dublincore.org/documents/dces/ (2016)
11. W3C: DataSetRDFDumps. https://www.w3.org/wiki/DataSetRDFDumps. Online; accessed 20 October 2020
12. Vdovjak, R., Houben, G.-J., Stuckenschmidt, H., Aerts, A.: In: Staab, S., Stuckenschmidt, H. (eds.) RDF and Traditional Query Architectures, pp. 41–58. Springer, Berlin, Heidelberg (2006)
13. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment **1**(1), 647–659 (2008)
14. Neumann, T., Weikum, G.: The rdf-3x engine for scalable management of rdf data. The VLDB Journal **19**(1), 91–113 (2010)
15. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: Proceedings of VLDB (2001)
16. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. Proceedings of the ACM SIGMOD International Conference on Management of Data (2014)
17. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.-C.N.: A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs (2021). 2102.13027
18. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: GStore: Answering SPARQL queries via subgraph matching. Proc. VLDB Endow. **4**(8), 482–493 (2011)
19. Zeng, L., Zou, L.: Redesign of the gStore system. Frontiers of Computer Science **12** (2018)
20. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple indexing for semantic web data management. In: VLDB, Auckland, New Zealand (2008)
21. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web. WWW '08, pp. 595–604. Association for Computing Machinery, New York, NY, USA (2008). doi:10.1145/1367497.1367578. https://doi.org/10.1145/1367497.1367578
22. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: A fast and compact system for large scale RDF data. Proceedings of the VLDB Endowment **6**, 517–528 (2013)
23. Jamour, F., Abdelaziz, I., Kalnis, P.: A demonstration of MAGiQ: matrix algebra approach for solving RDF graph queries. Proceedings of the VLDB Endowment **11**, 1978–1981 (2018)
24. Xiaowang, Z., Zhang, M., Peng, P., Song, J., Feng, Z., Zou, L.: gSMat: A scalable sparse matrix-based join for SPARQL query processing (2018)
25. Feng, J., Xiaowang, Z., Feng, Z.: MapSQ: A mapreduce-based framework for SPARQL queries on gpu (2017)
26. Galkin, M., Endris, K., Acosta, M., Collarana, D., Vidal, M.-E., Auer, S.: SMJoin: A multi-way join operator for SPARQL queries. (2017)
27. Feng, J., Meng, C., Song, J., Zhang, X., Feng, Z., Zou, L.: SPARQL query parallel processing: A survey. In: 2017 IEEE International Congress on Big Data (BigData Congress), pp. 444–451 (2017)
28. Ren, T., Rao, G., Zhang, X., Feng, Z.: SRSPG: A plugin-based spark framework for large-scale RDF streams processing on gpu. In: ISWC Satellites (2019)
29. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on Spark. Proc. VLDB Endow. **9**(10), 804–815 (2016)
30. Peng, P., Zou, L., Özsu, M.T., Chen, L., Zhao, D.: Processing SPARQL queries over distributed RDF graphs. The VLDB Journal **25**, 1–26 (2016)
31. NVIDIA: Thrust. https://docs.nvidia.com/cuda/thrust/index.html. Online; accessed 23 October 2020

32. Beckett, D.: The design and implementation of the Redland librdf RDF API Library. In: Proceedings of WWW10. Springer, Hong Kong (2001)
33. NVIDIA: Relational Joins. https://moderngpu.github.io/join.html. Online; accessed 24 October 2020
34. Neumann, T., Weikum, G.: The RDF3X engine for scalable management of RDF data. The Vldb Journal - VLDB **19**, 91–113 (2010)
35. Thomas Neumann: RDF3X. http://www.mpi-inf.mpg.de/~neumann/rdf3x. Online; accessed 24 October 2020
36. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of rdf data management systems, vol. 8796, pp. 197–212 (2014)
37. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2), 158–182 (2005). doi:10.1016/j.websem.2005.06.005. Selcted Papers from the International Semantic Web Conference, 2004
38. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A survey and experimental comparison of distributed sparql engines for very large rdf data. Proc. VLDB Endow. **10**(13), 2049–2060 (2017). doi:10.14778/3151106.3151109
39. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In: Proceedings of the 19th International Conference on World Wide Web. WWW '10, pp. 41–50. Association for Computing Machinery, New York, NY, USA (2010). doi:10.1145/1772690.1772696. https://doi.org/10.1145/1772690.1772696

Figures

```
<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> .
```

Figure 1

N-Triples example.

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdf:Description rdf:about="http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#Air">
    <rdfs:subClassOf rdf:resource="http://www.owl-ontologies.com/BiodiversityOntologyFull.owl#
    AbioticEntity"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 2

RDF/XML example.

Subject	Predicate	Object
Bob	knows	Alice
Bob	gender	Male
Alice	birthDate	”1997-05-20”
Alice	founder	YumYum
YumYum	isA	Restaurant
YumYum	hasMenu	”Fried Rice”
Bob	likes	Alice

Figure 3

Triple Example.

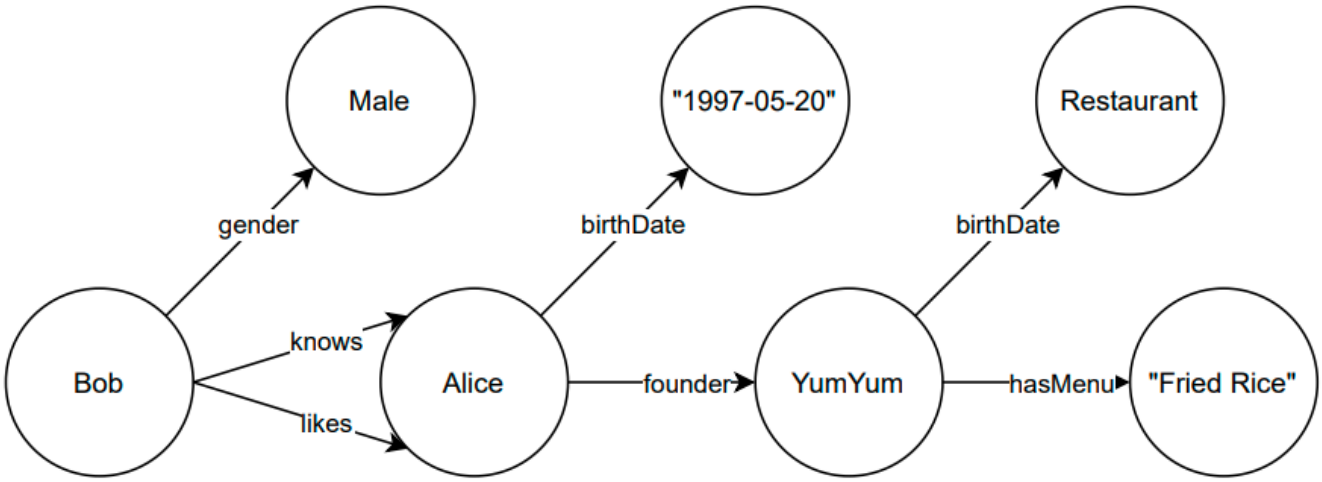


Figure 4

Triple data in graph visualization.

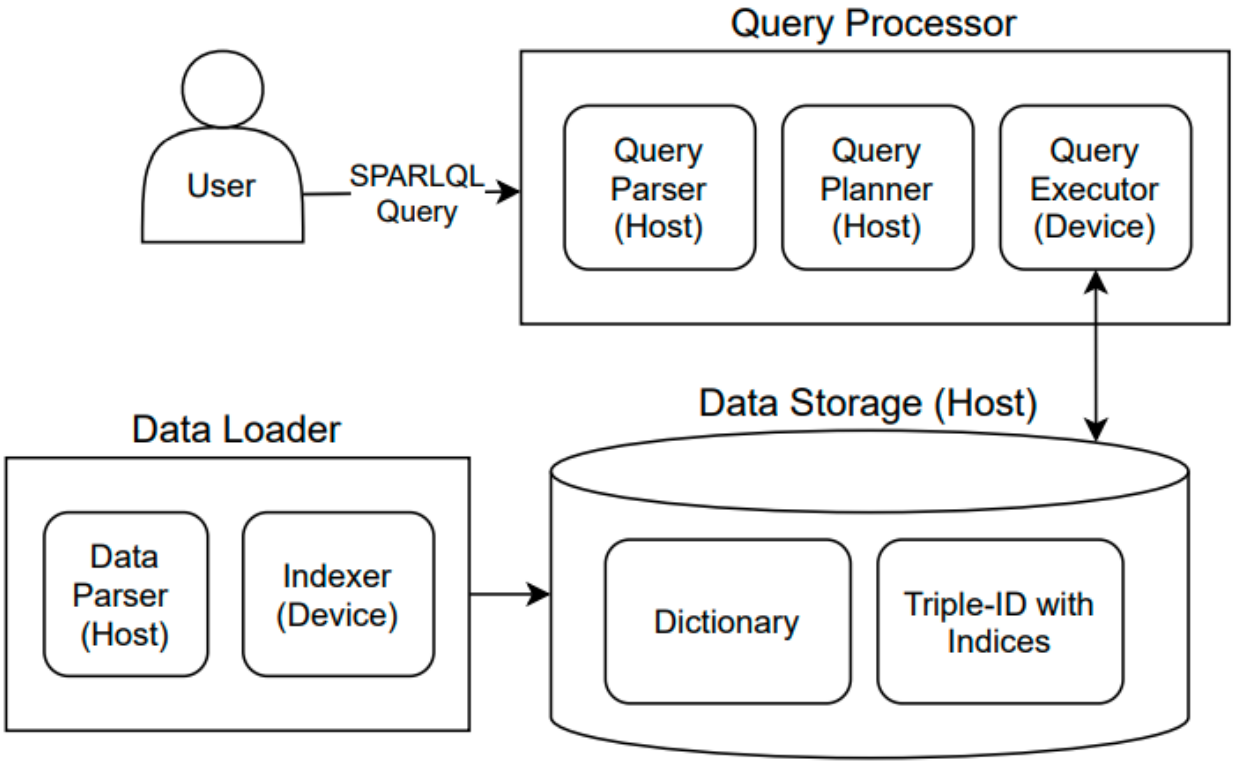


Figure 5

Elements of VEDAS.

Term	*id*	Term	*id*
Alice	1	knows	8
Bob	2	gender	9
YumYum	3	birthDate	10
Male	4	founder	11
"1997-05-20"	5	isA	12
Restaurant	6	hasMenu	13
"Fried Rice"	7	likes	14

(a) Dictionary

Subject	Predicate	Object
2	8	1
2	9	4
1	10	5
1	11	3
3	12	6
3	13	7
2	14	1

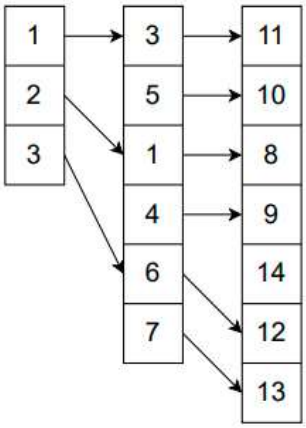
(b) Converted triple-ID

Figure 6

Dictionary and converted Triple-ID example.

Subject	Predicate	Object
1	3	11
1	5	10
2	1	8
2	4	9
2	4	14
3	6	12
3	7	13

(a) After sorting with SOP index



(b) Indexed column-based representation

Figure 7

Triple-id representation and example index.

$?x$	$?y$
1	3

(a) Result for sq_1

$?y$
3

(b) Result for sq_2

Figure 8

Subquery result example.

$?x$	$?y$
1	5
2	3
2	7
4	5
6	2

(a) R_i

$?x$	$?z$	$?w$
2	10	8
6	9	11
7	9	17

(b) R_j

$?x$	$?y$	$?z$	$?w$
2	3	10	8
2	7	10	8
6	2	9	11

(c) $R_k = R_i \ltimes R_j$

Figure 9

Join example.

$?x$	$?y$	$?z$	$?w$
7	6	5	12
7	8	1	4
11	4	5	8
15	9	12	3
13	2	7	2
13	7	5	5
19	6	1	7

(a) Before swapping $?z$

$?z$	$?x$	$?y$	$?w$
1	7	8	4
1	19	6	7
5	7	6	12
5	11	4	8
5	13	7	5
7	13	2	2
12	15	9	3

(b) After swapping $?z$

Figure 10

Swap index by variable ?z.

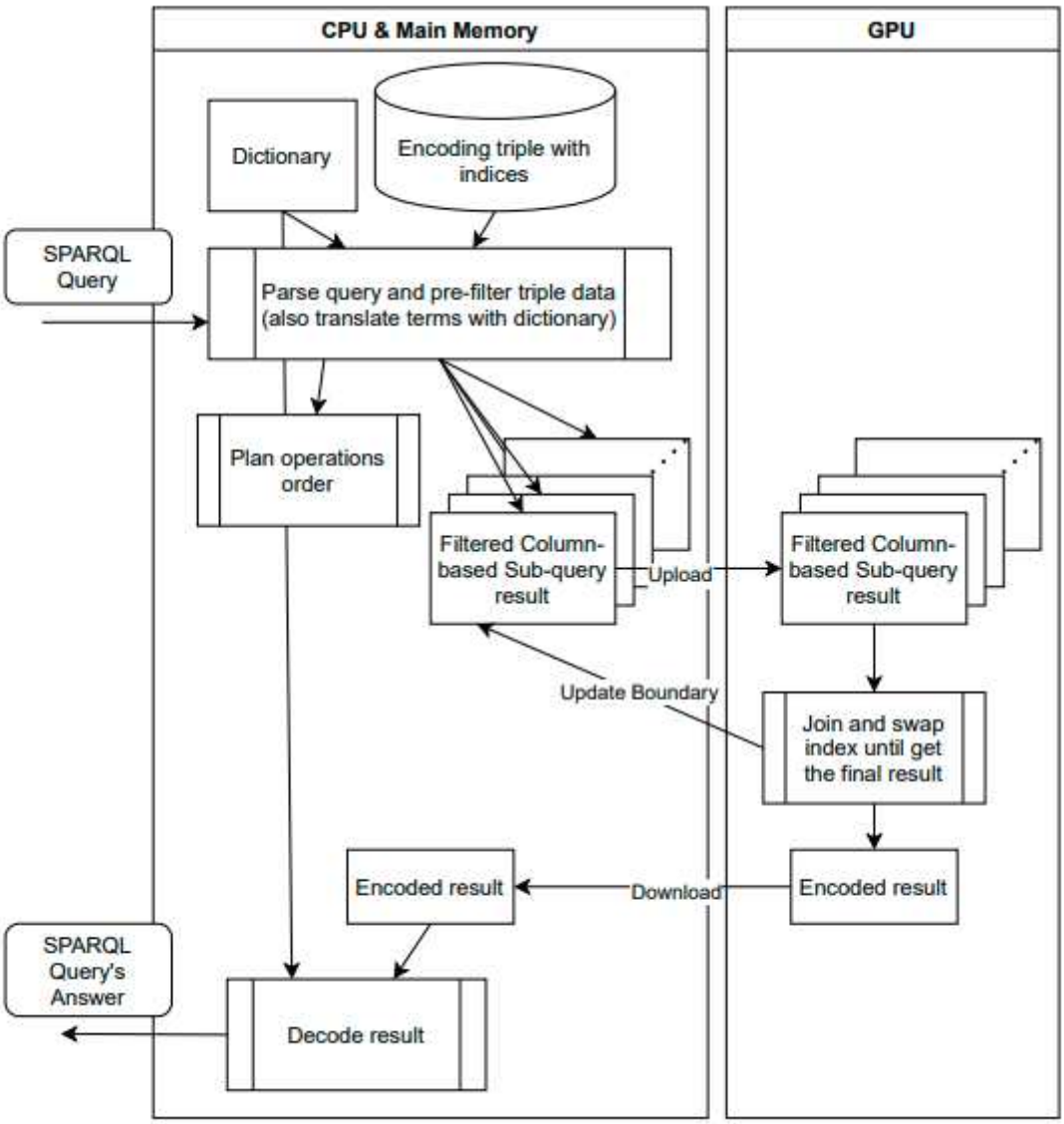


Figure 11

VEDAS's activity diagram to process the SPARQL query.

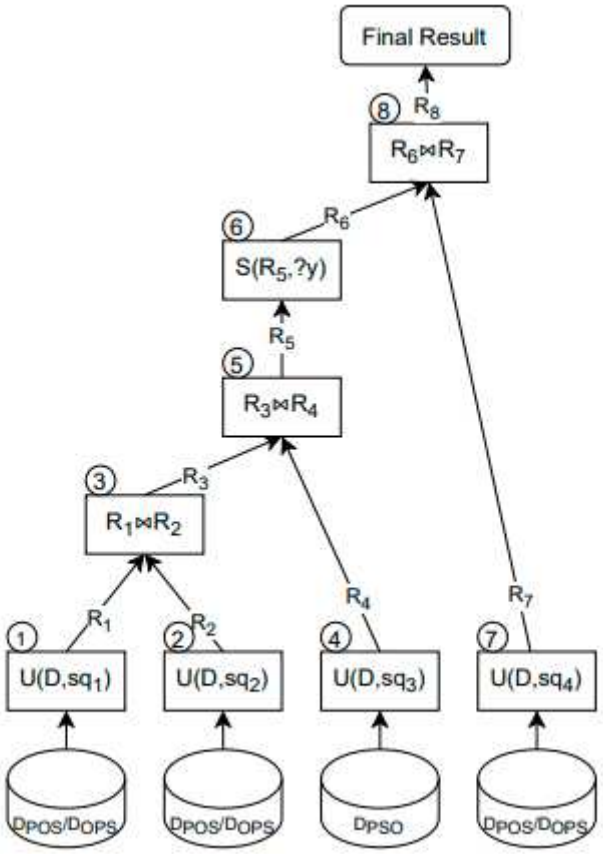


Figure 12

Execution plan for example query.

?x
1
39
42
45
48
50
65
78
730
934
1347

(a) R'_1

?x
48
50
65
78
730
934
1347

(b) R_1

?x
35
42
48
50
65
199
730
934
1322
1768
1998

(c) R'_2

?x
48
50
65
199
730
934
1322

(d) R_2

?x
48
50
65
730
934

(e) R_3

?x	?y
48	101
50	900
65	89
400	203
592	392
730	105
1029	223
1488	2093
1595	2123

(f) R'_4

?x	?y
48	101
50	900
65	89
400	203
592	392
730	105

(g) R_4

?x	?y
48	101
50	900
65	89
730	105

(h) R_5

?y	?x
89	65
101	48
105	730
900	50

(i) R_6

?y
50
101
800
900
934

(j) R'_7

?y
101
800
900

(k) R_7

?y	?x
101	48
900	50

(l) R_8

Figure 13

Intermediate result example of Listing 4.

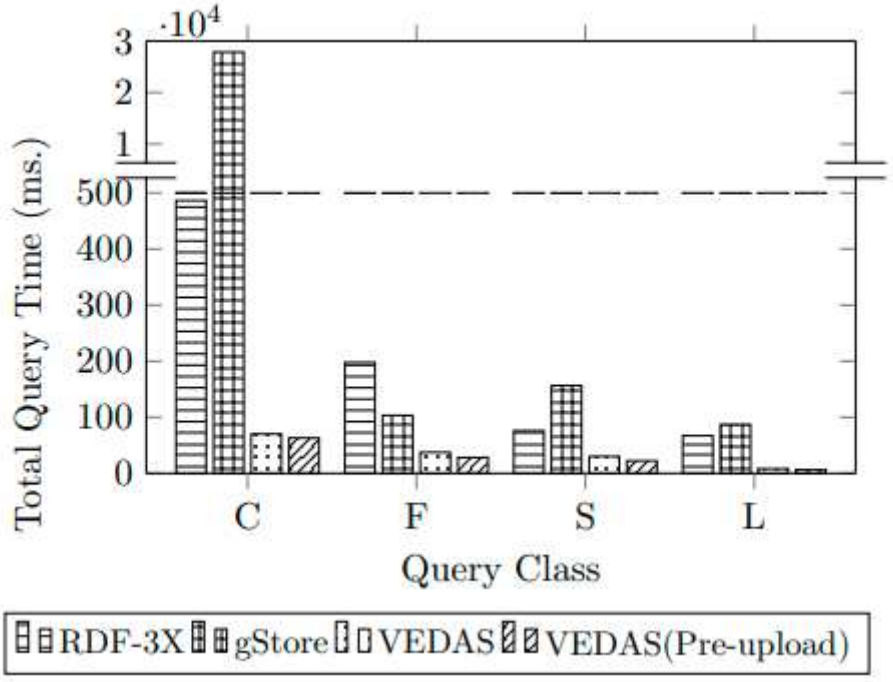


Figure 14

Query Time for 100M.

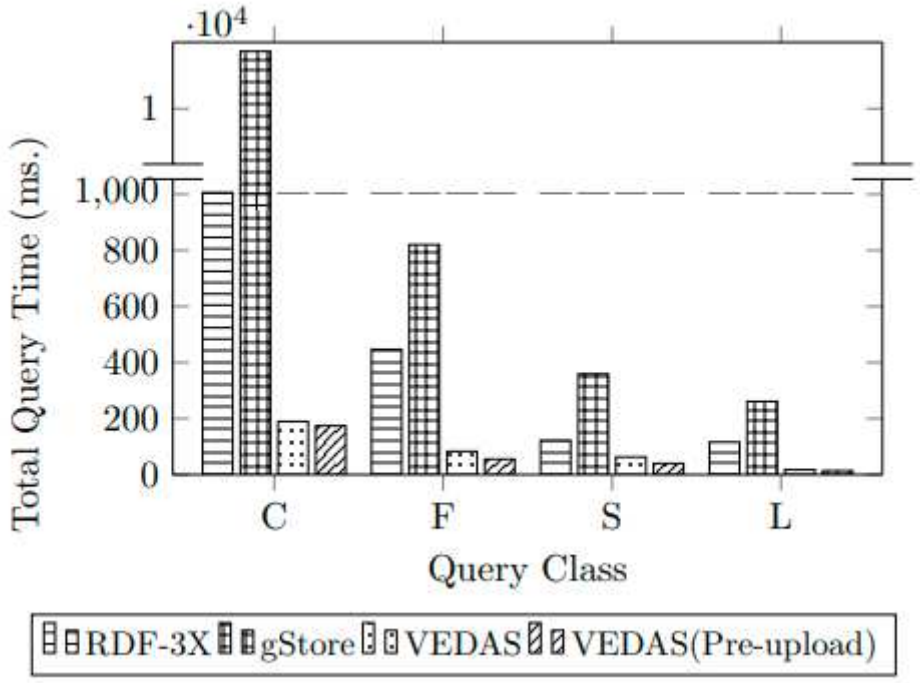


Figure 15

Query Time for 300M.

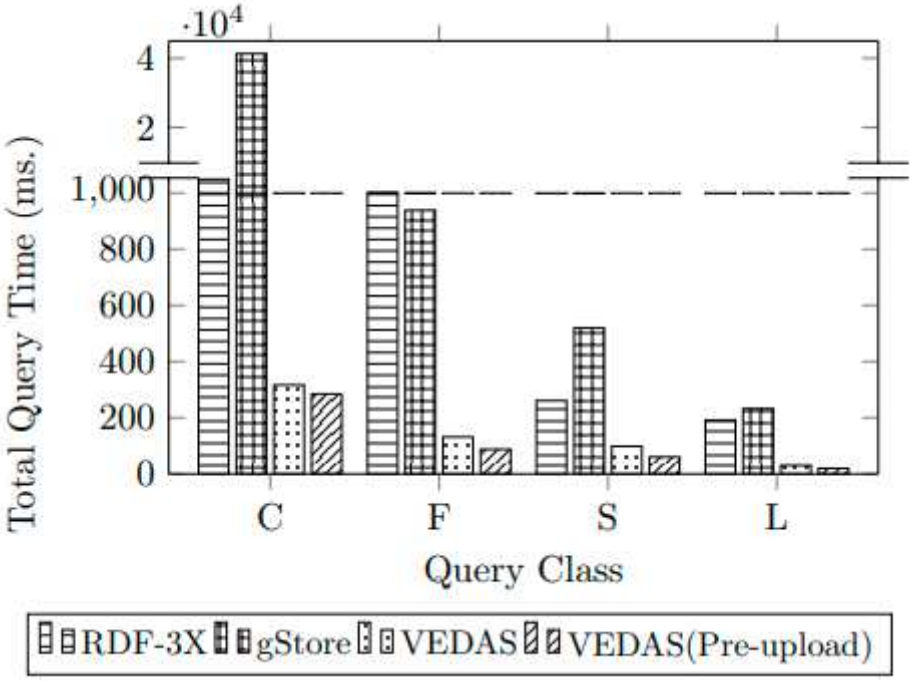


Figure 16

Query Time for 500M.