ECTI Transactions on Computer and Information Technology
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

SPARQL Query Optimizations for GPU RDF Stores

Pisit Makpaisit[1] and Chantana Chantrapornchai[2]

ABSTRACT

The SPARQL query time optimization is one of the challenges for the Resource Description Framework (RDF) data store. Even though Graphic Processing Units (GPU) can be used to accelerate query processing, it has weaknesses from the GPU memory transfer overhead. In this work, we propose optimization techniques for the GPU RDF store. In particular, we present the query optimization technique, which results in a reduction in the memory transfer time. In particular, the first approach is to use an empty interval filter that considers the empty data range to eliminate unneeded data during the uploading process. Secondly, we provide the heuristic query execution planner for generating an execution plan which suits our GPU RDF store and the filter technique. Our experiments on the WatDiv benchmark show that the proposed methods perform well on benchmark queries. The total query processing time on GPU is improved on average by 30% compared to the random plan generators.

Article information:

Keywords: SPARQL, GPU, RDF, Query Optimization

Article history:
Received: April 22, 2023
Revised: May 20, 2023
Accepted: May 27, 2023
Published: June 3, 2023 (Online)

DOI: 10.37936/ecti-cit.2023172.252463

1. INTRODUCTION

SPARQL is a standard query language for retrieving and manipulating RDF (Resource Description Framework) data. It has been widely adopted and is supported by many platforms. Since mainly, it has been used to query large RDF datasets, SPARQL query acceleration is necessary to reduce the response time.

In [1], VEDAS, an RDF store, uses the GPU to improve its retrieval system capabilities. VEDAS is based on column-based representation with indices that can decrease storage size and provide fast access with indexing. Thus, the query time can be reduced by utilizing the GPU for processing the large join.

GPU is superior for retrieving the RDF data since its massive processing unit can speed up the query time compared to the traditional CPU system. However, it requires the data to reside in the GPU memory before processing. Thus, the data uploading process is a weakness of such a system. When large data are uploaded and the processing time on the GPU is fast, the whole computation time leads to a low speedup gain. It also affects many other aspects:

1. It increases GPU memory allocation time.
2. It may occupy the GPU memory, leaving a small space for processing.
3. It increases the join time because of the large input size.

In addition, we found that the query execution planner (QEP) is a crucial component for the effectiveness of the GPU RDF store. With a good planner, data to upload may be reduced, thus, optimizing the upload time.

While many query optimization strategies concentrate on decreasing the join time or creating an execution plan that is join-friendly, the query optimization for the GPU store should focus on both the upload and join stages. The GPU join time is fast, but the upload time creates a significant overhead. In this work, we improve VEDAS by constructing the new GPU RDF store query optimizer as follows:

- We utilize the empty interval filter (EIF) technique for decreasing the data size to upload to GPU memory.
- We propose a non-statistical heuristics-based query execution planner that is suitable for GPU processing and our filtering technique.

2. RELATED WORKS

Hardware acceleration is an approach to accelerating the application with special hardware that is more efficient than the CPU for a given task. Hard-

[1,2] The authors are with High Performance Computing and Networking Center, Kasetsart University, Bangkok, Thailand. E-mail: pisit.mak@ku.th and fengcnc@ku.ac.th
[2] The corresponding author.

ware accelerators can be found in various forms, for example, FPGA, GPU, DSP, TPU, etc. This work emphasizes the use of the GPU as an accelerator. Numerous studies utilize GPU acceleration to enhance the performance of SPARQL queries and RDF stores. These works focus on how to represent the RDF data on GPU and how to implement the complete system for query execution [2] [3] [4] [5] [6]. Most GPU-based systems use the numeric ID to represent the IRI and literal in triple. They also used the tensor-based representation for storing the RDF data. Pattern matching, like regular expressions, can also be deployed on the FPGA accelerator to speed up the processing time [7].

Query optimization is an essential process for any database management system. It makes DBMS determine an efficient way to process the query and reduce time and resources for processing. The two most widely used approaches in RDF stores are the statistical approach and the heuristic approach.

Characteristic set (CS) [8] is one of the statistical approaches. It captures the semantics of data by creating sets of all predicates with the same subjects. The characteristic set can use to estimate the cardinality of the star-join. Meimaris et al. proposed extended characteristic sets (ECS) [9] that are improved from the characteristic sets. It focused on the triple level instead of the node level in CS by capturing characteristic sets of subjects and objects. Traveling Light [10] proposed the low-overhead statistic-based optimizer. It uses only selectivity statistics for predicates to overcome the great overhead problem in the statistical approach. It updates the cost model estimator when getting new queries. If the statistics are not provided the very first time, a heuristics-based optimizer such as [11] is used. Marios and George [12] proposed the distance-based query optimization. The idea is to transform the SPARQL query into the vector and use the vector distance to optimize the query. This work focused only on left-deep join and tried to reorder the triple for join. The drawbacks of the statistical approach are that it assumes data independence. The estimation of the cardinality may deviate significantly from the actual value. It also propagates the error when the number of joins increases. The statistical approach process is both computationally and memory intensive, whereas heuristics are lightweight and consume less memory.

Markus et al. [13] used some heuristic rules for selectivity estimation. They noticed that $sel(S) < sel(O) < sel(P)$ and bounds join are more selectivity than unbound joins. Petros et al. [11] propose the heuristic set for query optimization. The rules are based on the observed data. For example, some triple pattern is more selective than another pattern, or triple with literal is more selective, etc.

The modern approach uses RDF metadata, e.g., shape information, to estimate the cardinality and selectivity [14], [15]. However, this approach requires the available graph constraint information files that are not always available.

3. RDF AND SPARQL

Resource Description Framework (RDF) is a standard for representing information about resources on the web. It is used in many fields, such as knowledge management, the semantic web, and life science. The RDF data represent the relationship among data. Each row of data in RDF is called a *triple*. A triple is stored in the format *Subject Predicate Object* or can be written as $S\ P\ O$ in short. An example of RDF data is shown in Listing 1.

Listing 1: RDF Example

```
Eric :knows John .
John :knows Alice .
Eric :name "Eric" .
John :name "John" .
John :email "john@ku.ac.th" .
...
```

The domain of subject and predicate is IRI which is used to identify resources on the web. The object can be IRI or literal with many types, such as string, number, date, etc. Given I as a set of IRI and L is a set of literals, the triple (s, p, o) is a subset of $I \times I \times (I \cup L)$.

SPARQL query consists of one or more triple patterns to specify the data to be retrieved. For the triple pattern in the RDF data, each term can be replaced by the variable (which leads with ?). A variable represents an unknown value that can match any RDF data. Listing 2 shows the SPARQL example with three triple patterns. The set of triple patterns in the query can be called the basic graph pattern (BGP).

Listing 2: SPARQL Query Example

```
SELECT * WHERE {
    Eric :knows ?p .
    ?p :name ?n .
    ?p :email ?e .
}
```

SPARQL query can be represented with the labeled directed graph, called *query graph*. The corresponding query graph of Listing 2 is shown in Figure 1.

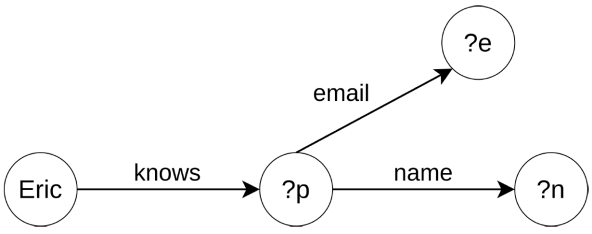


Fig.1: Query Graph of SPARQL in Listing 2.

4. SYSTEM OVERVIEW

VEDAS framework contains the GPU RDF store that represents triples in the column-based with indices. The index is the CSR style representation for compressing the data and reducing the time to access the data to upload and join. VEDAS requires just two index types (POS and PSO) for the query set or enables the full permutation indices, which require more memory space. The system architecture is shown in Figure 2. The responsibility of data loader is to parse the RDF data from files and convert each term to an integer ID. It also uses the GPU to sort triples and construct the indices. After the loader performs the loading task, all data and indices are stored in the data storage component. We use the in-memory storage approach to keep the ID and indices. The query processor receives the input SPARQL query, parses, and converts it to IDs. Query planner constructs the execution plan forming the basic graph pattern (BGP) of a query. The query executor uses the execution plan from the planner and completes the query processing on the GPU.

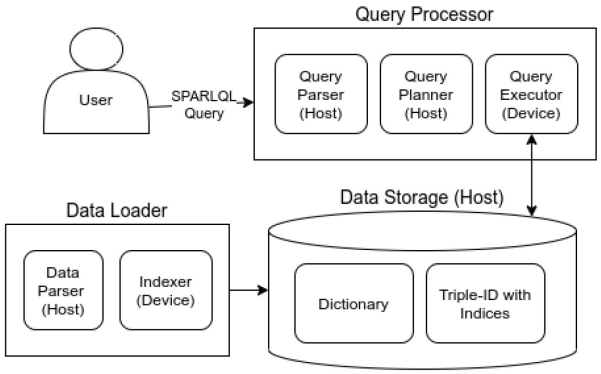


Fig.2: Components of VEDAS.

In detail, VEDAS has three basic operations to execute the query: upload, join, and index swap. Upload operation uploads the data from CPU memory to GPU memory by scanning from indices. The join operation performs the relational outer join of the upload data or intermediate results (IR) that already reside on the GPU memory. Because VEDAS uses sort-merge join in the join process, it is necessary to ensure that the column used for the joining is arranged in the sorted order. The process of sorting a column before joining is referred to as the index swap operation. Based on our research, the GPU method is more effective than the CPU in terms of accelerating the processing time by utilizing the massive amount of cores. The time required for joining can be significantly reduced. However, the time of uploading data will incur the overhead. The query with a low computation-upload ratio leads to low speed up from the CPU system. Hence, we apply the optimization technique, called pre-upload filter, to reduce the number of upload rows. However, some query still has a large upload portion of time.

In this work, we propose solutions to tackle the problem of large upload size problem. Reducing the upload time may also lead to reducing the time needed for GPU memory allocation and decreasing the input size for joining.

5. QUERY OPTIMIZATION

The large upload time is typically caused by the high selectivity join, i.e., the input size of join operations is large compared to the join result size. The pre-upload filter of our previous work can reduce the size by the following method. For IR R_1 and R_2 that the first column data range from (lb_1, ub_1) and (lb_2, ub_2), respectively, The pre-upload filter selects only the data in the range $(\max(lb_1, lb_2), \min(ub_1, ub_2))$. This means we can filter out the out-of-the-range data that are definitely not the join results. In addition, we propose a new technique to reduce the upload data similar to the pre-upload filter, but it works with the inner data range. This technique is known as an *Empty Interval Filter*.

A good planner can find the best plan to eliminate the unnecessary upload to the GPU memory. Thus, the planner for the GPU SPARQL executor is also very crucial. The proposed algorithm for QEP is not based on statistics or metainformation of the data. Therefore, collecting the metadata is not required during the loading process. It can run with low overhead to generate a good plan for the GPU RDF store.

5.1 Empty Interval Filter

The empty interval is the range of data that does not include the intermediate results. For example, data $\{1, 2, 3, 7, 13, 14, 22\}$ has three empty intervals $(4, 6)$, $(8, 12)$, and $(15, 22)$. Sometimes, the data has a large empty interval that can be a chance to filter out in the other IR. This is the intuition for an empty interval filter (EIF).

Let $EI =< l_1, u_1, l_2, u_2 >$ be the first two largest empty intervals $[l_1, u_2]$ and $[l_2, u_2]$ of variable $?v$ that are computed from the intermediate result. The EI is stored for every joined variable. Before uploading the new data to GPU, the system uses both pre-upload filter and empty interval filter EI to filter out the data that cannot be joined. The system scans indices and uploads only out-of EI range data. The algorithm to find the largest empty interval is to construct the difference array that stores the difference of consecutive values, and finds the maximum value among them. These two steps can be processed in parallel by GPUs. The system updates EI after the join by $?v$ and performs index swapping to use $?v$ as an index. It does not update every time after the join time, but updates only if this variable still can

Algorithm 1 Update Empty Interval

Input: Empty Interval EI, New Interval EI_{new}
Output: Merged Interval EI_{merged}

$L \leftarrow []$
$L.append((lb_1(EI), ub_1(EI)))$
$L.append((lb_2(EI), ub_2(EI)))$
$L.append((lb_1(EI_{new}), ub_1(EI_{new})))$
$L.append((lb_2(EI_{new}), ub_2(EI_{new})))$
Sort the bound pair in L in an ascending order
Merge the adjacency bound pair in L if they are overlapped
$EI_{merged} \leftarrow$ two largest bounds from L
return EI_{merged}

Algorithm 2 Generate Execution Plan Tree

Input: Query Graph $Q = (E, V)$
Output: Plan Tree T

for $e \in E$ **do**
    Initialize $e.index1$ and $e.index2$ with unbound variable. If e has only one unbound variable, set $null$ to $e.index2$
    $e.vars \leftarrow \{e.index1, e.index2\}$
end for
while $|E| > 1$ **do**
    $v \leftarrow$ Find best join-star center variable
    $adj \leftarrow$ Set of edge e adjacency to v and $v \in e.vars$
    Sort adj by cardinality in ascending order
    for e adjacent to v **do**
        if e is not merged edge **then**
            $e.treeNode \leftarrow CreateUploadNode(e.treeNode)$
        end if
        if e is first element **then**
            $e_0 \leftarrow e$
        else
            if $e_0.index1 \neq v$ and $e_0.index2 \neq v$ **then**
                $e_0.treeNode \leftarrow CreateIndexSwapNode(e_0.treeNode, v)$
                $e_0.index1 \leftarrow v$, $e_0.index2 \leftarrow null$
            end if
            if $e.index1 \neq v$ and $e.index2 \neq v$ **then**
                $e.treeNode \leftarrow CreateIndexSwapNode(e.treeNode, v)$
            end if
            $e.treeNode \leftarrow CreateJoinNode(e_0.treeNode, e.treeNode, v)$
            $e.vars \leftarrow e.vars \cup v$
        end if
    end for
    for e adjacent to v and $e \neq e_0$ **do**
        $e_0 \leftarrow MergeEdge(e_0, e)$
    end for
end while
return $e_0.treeNode$

Algorithm 3 MergeEdge

Input: Edge e_1, e_2
Output: Edge e_1
1: **if** $e_1.card < 1000$ and $e_2.card < 1000$ **then**
2: $e_1.card \leftarrow \lceil e_1.card \times e_2.card \times \sigma_1 \rceil$
3: **else if** $e_1.card < 1000$ or $e_2.card < 1000$ **then**
4: $e_1.card \leftarrow \lceil e_1.card \times e_2.card \times \sigma_2 \rceil$
5: **else**
6: $e_1.card \leftarrow \lceil e_1.card \times e_2.card \times \sigma_3 \rceil$
7: **end if**
8: $e_1.vars \leftarrow e_1.vars \cup e_2.vars$
9: Merge vertex that is not a shared vertex of e_1 and e_2.
10: Remove e_2 from graph
11: **return** e_1

be used to join. Although the largest interval is processed in parallel on GPU, it still consumes time. α is the threshold for updating, i.e., the update is performed only when the number of results from join or index swap is greater than α.

The update method is described in Algorithm 1. Lines 1-5 are to create a list consisting of 4 empty interval bounds, two from the originals and two from the new values. Line 7 is to merge adjacency bound pairs until no more pairs are overlapped. The term *merge* means union 2 bounds (x_1, y_1) and (x_2, y_2) as $(\min(x_1, x_2), \max(y_1, y_2))$.

5.2 Heuristic Query Execution Planner

Processing data on the GPU can be much faster than utilizing only the CPU. The QEP used in the GPU RDF store system should have low overhead to reduce the total query time. We propose the heuristic approach to archive this goal.

The main idea is to find the most selective triple pattern first. The sooner upload and join the small result triple pattern, the smaller upload size can be derived for the consequence triple patterns. Since we do not rely on any statistical information in the QEP, we collect the exact size for each triple pattern before the start of the planning process. This can be achieved by scanning from indices. This exact size is used to determine which triple pattern should be uploaded first and what to be the next, and so on. When the planner receives the query graph parsed by the query parser, Algorithm 2 generates the execution plan tree. The query executor serializes the plan tree obtained by post-order traversal.

Algorithm 2 generates a plan and modifies the query graph. At first, all edges in the query graphs represent the triple pattern. Each one has its metadata, such as the index for used ($index1$ and $index2$), cardinality($card$), and all variables header($vars$). After creating the join plan node, the edge (triple pattern) to be joined together will be merged. The new edge now represents joined IR. The metadata can also be merged.

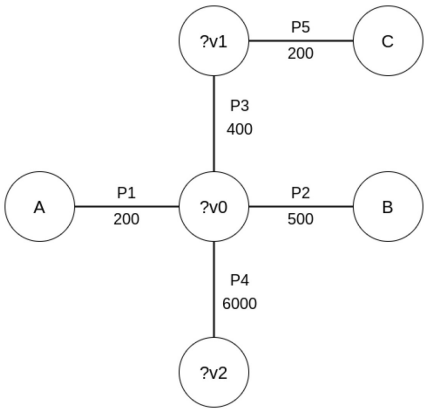


Fig.3: Complex-shaped query graph

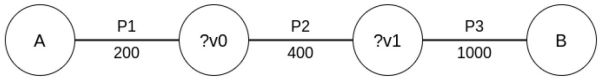
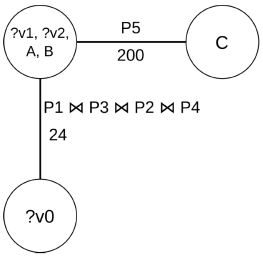


Fig.4: Linear-shaped query graph

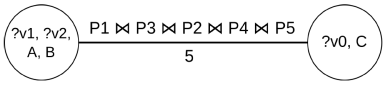
The algorithm attempts to find the best star-join subgraph in the query graph. The best star-join is the one that has the highest number of degrees, or the lowest total cardinality if more than one subgraph with the highest number of degrees exists. For example, the query graph in Figure 3 has two star-shaped centers at variables $?v0$ and $?v1$. In this case, the first iteration of the algorithm chooses $?v0$ center as the best star because it has degree 4 which is more than two compared to $?v1$. Figure 4 has two star-joins which both have degree 2. The star-join with $?v0$ center cardinality is 600 (200 + 400), while $?v1$ is 1,400 (1,000 + 400). Therefore, the algorithm selects $?v0$ first.

After we obtain the best star-join variable, all edges with the shared variable vertices are considered. The edges are sorted by cardinality in ascending order. For the triple pattern edge (or still not merged edge), the upload plan node is created. Then, the join plan node is created for the edge connected to the same vertex. The variable v is the variable that

is used to join. If the index of an edge is not v, it generates the index swap plan node (Lines 16 - 22). After creating all nodes to use for the star-join variable vertex, the related edges are merged to the first edge e_0 (Lines 27 - 29). The merged edge algorithm shows in algorithm 3. The merging of 2 edges into a single edge also combines the vertex that connects to both edges. Let edge $e_1 = (v, u)$ and $e_2 = (v, w)$; the vertices w and u are merged. All the edges $e = (x, w)$ are transformed to $e' = (x, u)$. Here, the σ value is the join selectivity factor. We estimate it based on the cardinality of both edges. If both are less than 1,000, we will use $\sigma_1 = 0.001$. Use $\sigma_2 = 0.0001$ when one of them is less than 1,000. Otherwise, we will give $\sigma_3 = 0.000001$. The estimated values are based on the assumption that join selectivity is high when the number of input rows is high and vice versa.



(a) After merging edge adjacent by $?v0$.



(b) After merging adjacent edge by $?v1$.

Fig.5: Steps of the Algorithm.

Figure 5 shows the next steps of algorithm 2 for the complex-shaped query graph in Figure 3. In Figure 5a after merging triple pattern edges P1, P2, P3, and P4, all edges become one joined IR edge. The cardinality of the new edge is 24. The next step is to use the center vertex as a variable to join. In this case, the index variable for join is $v1$. After joining the last two edges, the remaining is shown in Figure 5b. The algorithm terminates because there remains one edge left.

While flat plan or n-ary join has an advantage for building the multi-join execution plan, our QEP only generates the binary bushy-plan, because the binary plan has more chance to update the pre-upload filter and also EIF. The summary of edge metadata used in the generated algorithm is as follows.

- $index1$ and $index2$ maintain the values of the index that can be used. For edges that represent the triple pattern, the values of $index1$ and $index2$ are variables in the triple pattern. $index2$ can be null if the triple pattern has only one variable. For joined IR edge, $index1$ represents the last join variable, and $index2$ is null.
- $vars$ are variables in the IR header. It is used to validate whether the edges can be joined or not. If there is one $v \in vars$ that is the same as the join variable, this edge can be joined.
- $card$ stores the cardinality. The system scans the indices and obtains the number of data rows before the planning stage.
- $treeNode$ is the operation plan tree node that corresponds to the edge. The plan tree root node is stored in the final edge's $treeNode$ when the algorithm has completed.

6. EVALUATION

WatDiv synthetic benchmark dataset [16] is used to evaluate the performance of the proposed optimization techniques. The experiments are performed on the computer with the following specifications: 64 CPU of Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 256 GB of memory. The server contains 4 NVIDIA V100s with 32 GB memory. However, we are only utilizing a single GPU to conduct the experiments.

6.1 Empty Interval Filter Evaluation

To evaluate the EIF, we run the system and collect the query times and upload sizes in 2 cases, with and without the EIF. Table 1 shows that EIF can reduce the upload size and query time of the WatDiv query. Table 2 shows additional results like the number of empty interval updates called and EIF time for each query. For the query that the number of EIF update times is equal to 0, the upload size is zero. From Table 1, EIF can reduce the upload size by about 200 to 1,300,000 rows and, hence, affect query time. The C1 query is a special case in which the executor found the triple pattern with 0-row result, and it propagated the result to other triple patterns. Therefore, the upload size is equal to 0. EIF runs to update the interval in two events, that is, after joining and after index swapping. The linear-shaped query uses each variable to join at once. Hence, EIF can update the interval only after index swapping. EIF is not effective for L class queries. It has some updates for empty interval but cannot reduce the upload size. C class queries also have similar results to the L class. They have many triple patterns, but not many join with the same variables. The exception is for C2, which has a large star-join with very large results (greater than α). For that reason, the update is not triggered. Query S3, S4, S5, and S7 are queries with results of 0 (or 1), and the 0 result is found early; thus, EIF has no role in these queries. Other queries in F and S class gain benefits from EIF. The total query time is lower, even though the EIF overhead was included. From the results, EIF overhead depends on the number of joins with the same variables. For example, the overhead may become large if there are many degree star-joins. However, each update round has a great opportunity to reduce the upload size.

Table 1: Query time and upload size for each query in WatDiv 200M.

Query	Result (rows)	Query Time (ms.)		Query Size (rows)	
		w/o EIF	w EIF	w/o EIF	w EIF
C1	0	3.86	3.78	0	0
C2	39	74.35	73.95	13093482	13093482
C3	15941	111.88	110.33	96625372	96625372
F1	0	11.19	11.26	1379362	1365846
F2	23	12.15	11.86	3952327	3637590
F3	80	18.01	16.68	5342671	4985858
F4	170	17.76	16.61	4110803	3857933
F5	61	26.88	24.55	9507142	8183969
S1	3	8.45	7.81	890295	715387
S2	5409	8.86	8.48	2549834	2549834
S3	0	0.89	0.87	43421	43421
S4	0	1.47	1.31	66773	66559
S5	0	0.77	0.76	47909	47909
S6	20	7.46	6.53	2662212	2007551
S7	1	1.44	1.32	1421	1421
L1	1	6.49	6.62	2072071	2028710
L2	525	1.69	1.54	87670	87670
L3	6	5.86	5.92	2021762	2021762
L4	655	0.99	0.99	56374	56374
L5	764	2.05	1.91	115029	115029

Table 2: EIF time and upload size for difference α.

Query	Number of EI updates		EIF Time (ms.)		Upload Size (rows)	
	$\alpha =$ 2,500	$\alpha =$ 50,000	$\alpha =$ 2,500	$\alpha =$ 50,000	$\alpha =$ 2,500	$\alpha =$ 50,000
C1	0	0	-	-	-	-
C2	1	6	0.07	1.10	13093475	13093475
C3	0	0	-	-	-	-
F1	1	4	0.24	0.88	1364156	1364156
F2	8	8	1.38	1.42	3637590	3637590
F3	6	6	1.14	1.06	4985858	4985858
F4	8	9	1.69	1.89	3857933	3857933
F5	6	6	1.06	1.13	8183969	8183969
S1	4	4	0.66	0.68	715387	715387
S2	0	0	-	-	-	-
S3	0	0	-	-	-	-
S4	1	1	0.22	0.22	66559	66559
S5	0	0	-	-	-	-
S6	2	2	0.44	0.42	2007551	2007551
S7	0	0	-	-	-	-
L1	2	2	0.39	0.38	2028710	2028710
L2	1	2	0.20	0.48	87670	87670
L3	1	1	0.21	0.22	2021762	2021762
L4	1	1	0.23		56374	56374
L5	1	2	0.21	0.45	115029	114954

6.2 The Effect of α Parameter

Table 2 compares the number of EI updates, EIF time, and upload size when α is 2,500 and 50,000. It turns out that more EI updates do not necessarily lead to a reduction in the upload size. This is because sometimes the same interval is found, and the larger empty interval is not incurred. The results show that $\alpha = 50{,}000$ is not a good value since the corresponding upload size is not decreased. The only case that it can filter out the data is L5. However, it also reduces the size with insignificant query time reduction. Note that the value of 2,500 is a relatively small value compared to the size of after-joined IRs in almost all queries. We chose this value to show that applying EIF a few times is more appropriate. For $\alpha = 50{,}000$, it covers more than 75% of after-joined IRs. Hence, it represents the scenario when EIF is applied to about 75% of the number of join times. From intuition, the α value should depend on the data size and join selectivity. When the RDF file is large or sizable join selectivity, α should be set to the bigger value.

The small α also has one benefit. Because when the join result size is small, it has more probability of finding the large empty interval. In contrast with the large join results, the probability of having a large empty interval is lower.

6.3 Heuristic Query Execution Planner Evaluation

To evaluate the QEP performance, we compare the proposed method with 3 random generators. The L3 and L4 have only 2 triple patterns and can generate only one plan. Therefore we skip to show their results. The other L class queries also have only 3 triple

Table 3: Query time for each query in WatDiv 200M.

Query	Query Time (ms.)			
	Random1	Random2	Random3	Proposed
C2	82.61	101.09	221.65	73.95
C3	112.42	99.86	110.53	110.33
F1	10.67	11.46	12.46	11.26
F2	14.82	12.82	14.41	11.86
F3	18.43	20.35	27.26	16.68
F4	17.57	16.37	17.17	16.61
F5	28.03	31.70	35.66	24.55
S1	32.96	57.79	49.45	7.81
S2	8.94	9.33	9.03	8.48
S3	1.57	5.20	5.21	0.87
S4	5.50	6.03	5.59	1.31
S5	1.86	3.70	4.22	0.76
S6	8.58	6.77	8.39	6.53
S7	1.71	1.21	1.39	1.32
L1	7.75	6.73	-	6.62
L2	3.98	1.50	-	1.54
L5	1.89	3.78	-	1.91

Table 4: Upload size for each query in WatDiv 200M.

Query	# of Triple Pattern	Upload Size (rows)			
		Random1	Random2	Random3	Proposed
C2	10	13380287	13737013	13736978	13093482
C3	6	96625702	96625742	96625778	96625372
F1	6	1379822	1365844	1379820	1365846
F2	8	3975622	3946654	3964968	3637590
F3	6	6022312	6065061	7100481	4985858
F4	9	4448324	3624832	4275890	3857933
F5	6	9602952	9301637	9727257	8183969
S1	9	9439858	14517113	10398593	715387
S2	4	2549849	2550300	2550328	2549834
S3	4	93525	1280566	1281124	43421
S4	4	1465906	1474783	1475431	66559
S5	4	348093	497964	498074	47909
S6	3	2963986	2007551	2964025	2007551
S7	3	1421	28227	28227	1421
L1	3	2075731	2072071	-	2028710
L2	3	472848	87670	-	87670
L5	3	115029	500209	-	115029

Table 5: Join size for each query in WatDiv 200M.

Query	# of Triple Pattern	Join Size (rows)			
		Random1	Random2	Random3	Proposed
C2	10	22533816	49169901	116160434	22404389
C3	6	6738829	6236316	6633324	6524044
F1	6	1392250	1382296	1735697	1374000
F2	8	3776024	3532199	3870196	3367571
F3	6	6035511	6774502	9094598	5086956
F4	9	4548199	3582957	4361602	3832034
F5	6	9716444	11280979	13426639	8567633
S1	9	10527547	18933108	18062227	774038
S2	4	2600207	2586603	2627141	2580938
S3	4	95274	1286678	1407047	43421
S4	4	1468911	1511922	1488903	66699
S5	4	358220	503063	589464	47909
S6	3	2965641	2280871	2972342	2280871
S7	3	1422	28229	29645	1422
L1	3	2290569	2072077	-	2072077
L2	3	487452	102674	-	102674
L5	3	130033	520103	-	130033

patterns and 2 possible plans. We generate all 2 unique plans for each one. In all tests, the empty interval filter is enabled and set α to 2,500. Each query is run 10 times and the best time is selected for the comparison. Table 3 shows the query time for each planner. Tables 4 and 5 show the upload and join sizes, respectively. The definition of upload size is the number of rows that are uploaded from the CPU host to the GPU. To simplify the problem, we ignore the number of columns to be uploaded and consider only the number of rows. The join size becomes the summation of the number of rows of 2 inputs and disregards the number of columns.

For the small number of triple pattern queries (S6, S7, L1, L2, and L5), The results show that the proposed QEP selects the best plan. Our QEP is also better than the random plans in other queries except for F4. It has the minimum number of uploads and join size among all plans, and leads to the minimum query time. The interesting point is that for the result of the S class, the query that has a small number of resulting rows or 0 rows, the proposed method is efficient. The reason is it early performs the very selective join to execute. The result size of the early steps is small or 0. Thus, the next upload with a pre-upload filter can be small.

Table 5 shows that when the upload size is decreased, the join size is also decreased. As a result of reducing the upload size, the join size and join execution time are decreased. However, reducing the upload size is more effective than reducing the join size in the aspect of reducing the total query time.

7. CONCLUSIONS

This work proposes two SPARQL query optimizations suitable for the VEDAS GPU RDF store. Two approaches are demonstrated. First, we eliminate the unused data before uploading by using the empty interval filter. This optimization is to collect the empty interval for each join variable and use it to prune out the data. Secondly, the heuristic approach QEP, which has low overhead and requires no metadata, is proposed. QEP tries to find the most proper star-join in the query graph and generates the execution plan. It finds the largest degree of star-join and summation of cardinality, which guides to the good-enough plan or best plan in some cases.

The experiments show that empty interval filter can reduce upload size up to 24.60% and 12.47% for query time. EIF works well in conjunction with the pre-upload filter in the original system. It can reduce the upload size in most F and S classes in the tested WatDiv benchmark. α parameter in EIF is suggested to be a small value to reduce overhead. However, there is an opportunity to find the optimal value in the future.

QEP performs well in most queries. For the small-size result query, our method can reduce many unnecessary uploads to reduce the overall processing time. It also generates the best plan for the query that has a small number of triple patterns.

ACKNOWLEDGMENT

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 would like to thank NVIDIA hardware grant and ARES system from Kasetsart University for providing hardware support for running the experiments.

References

[1] P. Makpaisit and C. Chantrapornchai, "VEDAS: an efficient GPU alternative for store and query of large RDF data sets," *J. Big Data*, vol. 8, no. 1, p. 125, 2021. [Online]. Available: `https://doi.org/10.1186/s40537-021-00513-y`

[2] Y. Kim, Y. Lee, and J. Lee, "An efficient approach to triple search and join of hdt processing using gpu," in *Proc. 7th Int. Conf. Adv. Databases Knowl. Data Appl*, pp. 70–74, 2015.

[3] C. Chantrapornchai and C. Choksuchat, "TripleID-Q: RDF Query Processing Framework Using GPU," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2121-2135, 1 Sept. 2018

[4] F. Jamour, I. Abdelaziz, and P. Kalnis, "A demonstration of MAGiQ: matrix algebra approach for solving RDF graph queries," *Proceedings of the VLDB Endowment*, vol. 11, pp. 1978–1981, 08 2018.

[5] T. Ren, G. Rao, X. Zhang, and Z. Feng, "Srspg: A plugin-based spark framework for large-scale rdf streams processing on gpu," in *ISWC (Satellites)*, pp. 89–92, 2019.

[6] X. Zhang, M. Zhang, P. Peng, J. Song, Z. Feng, and L. Zou, "gsmat: A scalable sparse matrix-based join for sparql query processing," 2018.

[7] Hendarmawan, M. Kuga, and M. Iida, "Streaming Accelerator Design for Regular Expression on CPU+FPGA Embedded System," *ECTI-CIT Transactions*, vol. 16, no. 4, pp. 448–459, Oct. 2022.

[8] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," *2011 IEEE 27th International Conference on Data Engineering*, Hannover, Germany, pp. 984-994, 2011.

[9] M. Meimaris, G. Papastefanatos, N. Mamoulis and I. Anagnostopoulos, "Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization," *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, San Diego, CA, USA, pp. 497-508, 2017.

[10] G. Selvaraj, C. Lutteroth and G. Weber, "Traveling Light — A Low-Overhead Approach for

SPARQL Query Optimization," *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, Laguna Hills, CA, USA, pp. 56-61, 2021.

[11] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz, "Heuristics-based query optimisation for sparql," in *Proceedings of the 15th International Conference on Extending Database Technology*, pp. 324–335, 2012.

[12] M. Meimaris and G. Papastefanatos, "Distance-Based Triple Reordering for SPARQL Query Optimization," *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, San Diego, CA, USA, pp. 1559-1562, 2017.

[13] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in *Proceedings of the 17th international conference on World Wide Web*, pp. 595–604, 2008.

[14] A. Abbas, P. Genev'es, C. Roisin, and N. Layaïda, "Selectivity estimation for sparql triple patterns with shape expressions," in *Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18*. Springer, 2018, pp. 195–209.

[15] K. Rabbani, M. Lissandrini, and K. Hose, "Optimizing sparql queries using shape statistics," in *Advances in Database Technology-24th International Conference on Extending Database Technology, EDBT 2021*. OpenProceedings. org, 2021, pp. 505–510.

[16] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," in *The Semantic Web–ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I 13*. Springer, 2014, pp. 197–212.

Pisit Makpasit received a bachelor's degree in computer science from Thammasat University of Thailand in 2009 and a master's degree in computer engineering from Thailand's Kasetsart University in 2014. Currently, he is a Kasetsart University Ph.D. student in the department of computer engineering. His research interest is parallel computing, especially parallel programming with compiler directives and GPUs. His doctoral research focused on improving the performance of RDF data querying.

Chantana Chantrapornchai received the bachelor's degree in computer science from Thammasat University of Thailand in 1991, the master's degree from Northeastern University at Boston, College of Computer Science, in 1993 and the PhD degree from the University of Notre Dame, Department of Computer Science and Engineering, in 1999. Currently, she is an associated professor with the Department of Computer Engineering, Faculty of Engineering, Kasetsart University, Thailand. Her research interests include parallel computing, big data processing, deep learning application, semantic web, computer architecture, and fuzzy logic. She is also affiliated with HPCNC laboratory. She is currently a principle investigator of GPU Education program and NVIDIA DLI Ambassador at Kasetsart University.