applied sciences

MDPI

Article

An Efficient Distributed SPARQL Query Processing Scheme Considering Communication Costs in Spark Environments

Jongtae Lim [1], Byounghoon Kim [1], Hyeonbyeong Lee [1], Dojin Choi [1], Kyoungsoo Bok [2] and Jaesoo Yoo [1,*]

[1] Department of Information and Communication Engineering, Chungbuk National University, Chungdae-ro 1, Seowon-Gu, Cheongju 28644, Korea; jtlim@chungbuk.ac.kr (J.L.); bhkim@chungbuk.ac.kr (B.K.); lhb@chungbuk.ac.kr (H.L.); mycdj91@chungbuk.ac.kr (D.C.)

[2] Department of SW Convergence Technology, Wonkwang University, Iksandae 460, Iksan 54538, Korea; ksbok@wku.ac.kr

* Correspondence: yjs@cbnu.ac.kr or yjs@chungbuk.ac.kr; Tel.: +82-43-261-3230

Abstract: Various distributed processing schemes were studied to efficiently utilize a large scale of RDF graph in semantic web services. This paper proposes a new distributed SPARQL query processing scheme considering communication costs in Spark environments to reduce I/O costs during SPARQL query processing. We divide a SPARQL query into several subqueries using a WHERE clause to process a query of an RDF graph stored in a distributed environment. The proposed scheme reduces data communication costs by grouping the divided subqueries in related nodes through the index and processing them, and the grouped subqueries calculate the cost of all possible query execution paths to select an efficient query execution path. The efficient query execution path is selected through the algorithm considering the data parsing cost of all possible query execution paths, amount of data communication, and queue time per node. It is shown through various performance evaluations that the proposed scheme outperforms the existing schemes.

Keywords: SPARQL; Apache spark; RDF; distributed query processing; communication cost

Citation: Lim, J.; Kim, B.; Lee, H.; Choi, D.; Bok, K.; Yoo, J. An Efficient Distributed SPARQL Query Processing Scheme Considering Communication Costs in Spark Environments. *Appl. Sci.* **2022**, *12*, 122. https://doi.org/10.3390/app12010122

Academic Editor: Elisa Quintarelli

Received: 7 December 2021
Accepted: 21 December 2021
Published: 23 December 2021

1. Introduction

The semantic web allows computers to understand and manipulate the meaning of documents [1–3]. The semantic web has emerged to process various resources and data management by computers automatically. As studies on semantic web services were conducted actively, a Resource Description Frame (RDF) that can manage resources over the web was studied [4–6]. RDF is a standard metadata model that expresses resource information in the semantic web. The RDF identifies a relationship between expressed values and processes data intelligently. RDF forms a graph through a triple structure, which is expressed with the subject (S), predicate (P), and object (O) [7–10].

As the volume of the RDF graph increases over the semantic web, some studies on SPARQL Protocol and RDF Query Language (SPARQL), which is a language that can query and process RDF data, were conducted [11–14]. SPARQL is an RDF query language, which consists of PREFIX, SELECT, and WHERE clauses [15–17]. PREFIX means an RDF graph set used in a query. SELECT means variable expressing query results. WHERE means conditions to process the query. There are other clauses: ORDER BY clause to designate an order of query results, LIMIT clause that designates how many records are displayed in the query result, and OFFSET clause that designates from which lines the query result starts. As web-based services increase in scale, it becomes impossible to store the entire RDF graph data, which is used in services in a repository based on a single node. Moreover, performance degradation occurs when a large scale of RDF graph is queried and processed [18–24]. Thus, not only a scheme of how to store RDF graph in the distributed manner but also a query processing scheme of the stored RDF graph are required [24–29].

The existing query processing schemes that divide a SPARQL query into subqueries and process the divided subqueries were proposed for SPARQL query processing [30–38].

As a result, communication amount is increased, and the cost of query processing becomes higher since many joins occur or a large amount of data are generated when fetching results from the divided data in various environments in which data are divided [39,40]. In the existing schemes, query processing is conducted by searching many nodes and finding data accordingly during SPARQL query requests over the peer-to-peer (P2P) environment. The data transmission and join cost occurs in order to generate result data from the distributed result data during query processing through searching data via many nodes [41]. Another scheme in the past considered the join and communication costs during query processing. However, it assumed an environment where data were replicated in a few nodes. The data communication and join cost can be reduced if data are replicated and stored rather than distributed and stored. However, the existing schemes cannot be utilized in a data distributed environment if all RDF graphs cannot be stored in a single node [42]. More recently, a scheme that reduces the disk I/O cost during data parsing and join cost during query processing in the Spark environment, which was a distributed in-memory platform, was proposed [43–45]. However, it does not consider the communication cost during query processing in a distributed environment, resulting in a large amount of cost generated during query processing of a large scale of RDF graph [43].

In this paper, we propose a distributed SPARQL query processing scheme considering Spark environments to reduce join and communication costs that occurred during query processing, which are problems in existing schemes. The proposed scheme reduces disk I/O cost through query processing in the Spark environment, which is a distributed in-memory environment. In addition, it reduces join and communication costs during query processing through the query execution paths considering data communication costs and various costs that occur during query processing. A single SPARQL query is divided into multiple subqueries based on the WHERE clause of SPARQL, and subqueries are grouped for distributed query processing. The data communication cost is reduced during query processing through grouping. The grouped subqueries select optimal query execution paths through the algorithm considering data parsing, data communication and join costs, and queue time. The unnecessary join and queue time can be reduced by performing query processing through the selected query execution paths.

The rest of this paper is organized as follows. In Section 2, previous studies related to the query processing scheme proposed in this paper are described. In Section 3, the query processing scheme proposed in this paper is explained in detail. In Section 4, performance evaluation on existing and proposed schemes is conducted. Finally, in Section 5, conclusions and future study plans are presented.

2. Related Work

In ref. [39], Leida and Chu proposed a distributed SPARQL query processing scheme in a distributed grid environment. A SPARQL query was divided based on the WHERE clause, and a single atom was created for each condition and transferred to a few nodes according to the query performing plan. For each atom, the weight of each (S, P), (S, O), and (P, O) was calculated, and the lowest value was chosen as the atomic weight in the calculation. Once the weight was calculated, a query path, in which atoms that are able to connect to each of the atoms are combined, was created. The created query path was then divided into many query paths (as many as the number of the nodes uniformly based on the weight of the atoms). The divided query paths were then transferred to the nodes randomly to process the query.

In ref. [41], Zhou et al. proposed a query processing scheme when RDF graphs were stored in a distributed manner in P2P environments. A SPARQL query was divided based on the WHERE clause, and a single subquery was created for each condition. Each of the subqueries searches to see whether it is connected to the node that stores the data through the index starting from the SPARQL query-requested node. Each of the subqueries verifies whether query processing can be conducted in a node through the index, and the index is searched using the hash function based on the values of constants that propose a triple

pattern variable of the subquery. The node that stores the data is verified through the index, and a query is transferred to the data node to process a query of preferred data. It then receives the query result to complete the query processing. If no index information of the subqueries is found in the node, subqueries are transferred to the neighboring node in the clockwise direction.

In ref. [42], Hammoud et al. proposed a SPARQL query processing scheme while a large amount of RDF graph was replicated in each of the slaves in the master–slave environment. It proposed a distributed SPARQL query processing scheme in the replicated distributed environment to reduce the join and communication costs that occur in a distributed environment. It was performed assuming that all RDF graphs can be stored in a single node. All RDF graphs were stored in a single node, and a single node can perform a more efficient query processing scheme if SPARQL queries are processed through replication.

In ref. [43], Chen et al. proposed a distributed SPARQL query processing scheme in a Spark environment. Apache Spark is a general-purpose high-performance distributed platform [43–45]. As with other functions, Spark can process MapReduce tasks and streaming processing. The existing platforms that provide analysis functions on big data have degraded performance because they are run on a disk basis during query processing. However, Spark enables faster analysis on big data as query processing can be conducted on a faster memory basis. All tasks in Spark are performed through resilient distributed datasets (RDDs). An RDD is a set of distributed data (objects) that cannot be modified [46,47]. All tasks inside Spark are processed by creating a new RDD, modifying existing RDDs, or calling operations (functions or methods) in RDDs to calculate the results [48]. The requested SPARQL query excludes the conditions that include predicates such as type and class, which are unnecessary query types, during query processing and is transformed to TR-SPARQL. It is then divided into a few subqueries based on the WHERE clause of SPARQL. The divided subqueries create all connectable query paths. The data parsing cost, join cost, and communication cost is calculated for each of all the created query paths, and the least-cost query path is selected. Query processing is performed in the Spark distributed memory environment through the selected query path, and subqueries are transformed to a Resilient Discreted SubGraph (RDSG) format to perform a join action with other RDSGs.

Previously, SPARQL query processing schemes were proposed in various environments. A query processing using the existing schemes has various characteristics. However, not considering the data transfer and the disk I/O may be a problem when a large amount of RDF graph data cannot be stored in a single node anymore by limitation of memory and storage. The reason is that a lot of data transmission and disk I/Os occur to process queries in a distributed storage environment. In particular, graph processing creates a lot of join operations that cause a lot of data transmission and disk I/Os. In refs. [32,34], a large amount of RDF graphs were distributed, and a query processing scheme was conducted in accordance with a distributed stored environment. However, it did not consider data communication and join costs incurred during query processing. Thus, the query processing cost increased as the RDF graph and queries were more complex or larger during query processing. [42] performed a scheme considering data communication and join costs incurred during query processing. However, a system structure proposed in [42] was a scheme in a replicated environment rather than distributed stored environment. Although a scheme in a replicated environment was proposed assuming that all RDF graphs can be stored in a single node, a problem occurs if a large amount of RDF graphs cannot be stored in a single node anymore.

3. The Proposed Distributed SPARQL Query Processing Scheme

In Section 3, we introduce a distributed SPARQL query processing scheme considering Spark environments to reduce join and communication costs that occurred during query processing. As volumes of the RDF graphs increase, distributed storage and an efficient query processing scheme are required. However, the existing schemes did not consider

the communication and join costs incurred during query processing, entailing a large cost during query processing of a large amount of RDF. In addition, the disk I/O cost increases as data size increases. Thus, a query processing scheme is needed in consideration of the many costs incurred during query processing. We proposed a distributed stored RDF graph query processing scheme in the Spark environment. It reduces disk I/O costs through query processing in the Spark environment, which is a distributed in-memory environment. It also reduces the communication cost through grouping the divided SPARQL subqueries when a query on distributed RDF graph is processed. In addition, join, and communication costs are reduced by setting a query processing order through efficient query execution paths.

3.1. Overall Architecture

The proposed scheme is a master–slave structure in the Spark cluster environment, as shown in Figure 1. RDF is a directed graph $G = (V, E)$ represented by triple patterns such as subject–predicate–object, where the vertices V are the subject and the object, and the directed edges E are the predicates expressing the relation between the subject and object. A client sends a request for SPARQL query processing to the master, and the master divides the SPARQL query into subqueries SQ_i in the query decomposition. The divided subqueries SQ_i are grouped based on the criterion of related nodes. For the grouped subqueries GQ_j, all possible query execution paths are created that are query-processable in the query execution path generation. From the created possible performing query paths, the optimal query execution path is identified through the algorithm that considers join and communication costs and queuing time in each of the query execution paths. The join and communication costs can be reduced during query processing by processing a query through the optimal query execution path. The subqueries that are grouped into slaves according to the selected query execution path are transferred as they are transformed into each of the RDD, and RDF subgraph SG_i that are matched with the subqueries are parsed. For each of the subqueries, the intermediate results IR_i are produced after data parsing, and query processing is performed through data communication and join according to the query execution path. After query processing is complete, a slave that owns the result sends the results to the master. Once the master receives the result from the slave, it sends the result to the client to complete the query processing.

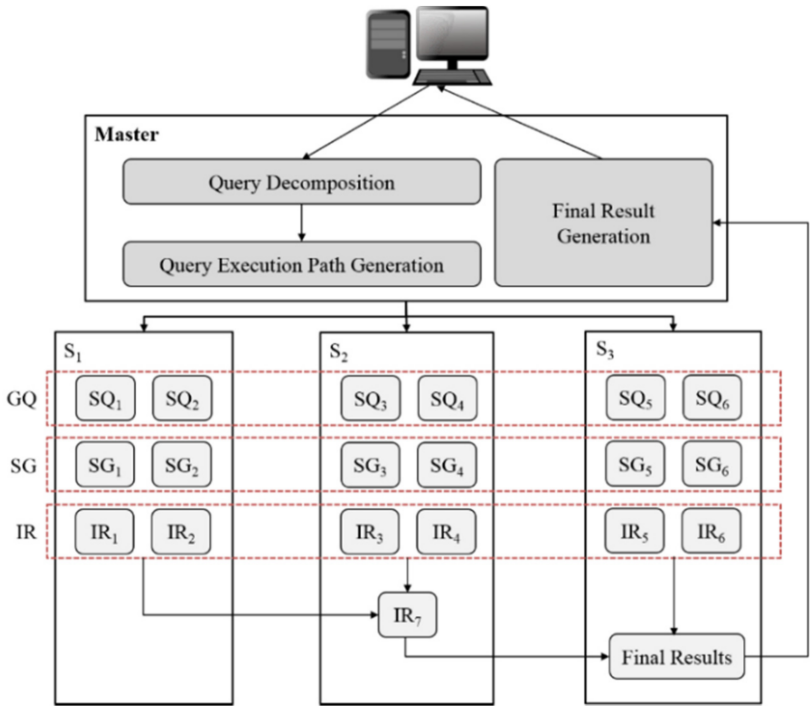


Figure 1. Overall architecture of the proposed scheme.

3.2. Query Decomposition

A SPARQL query that is optimized to distributed data is divided into subqueries and transferred for query processing of RDF graph in a distributed environment. If queries that are unsuitable to distributed data are transferred and processed, efficient query processing cannot be conducted, and disk I/O cost and data communication cost are increased during query processing. In addition, communication cost increases as the number of subqueries increases during query processing after a SPARQL query is divided into subqueries. Thus, efficient query processing that can reduce the communication cost through grouping of subqueries is conducted.

For example, if the RDF graph is distributed over a few nodes, as shown in Figure 2, only N_1 is searched to know the job of the person called "Kim." However, the communication cost has to be increased because slave nodes N_1, N_2, and N_4 have to be searched once a SPARQL query is requested to find the living city of "Kim." As shown in the above example, query decomposition is needed to reduce the communication cost and achieve efficient query processing.

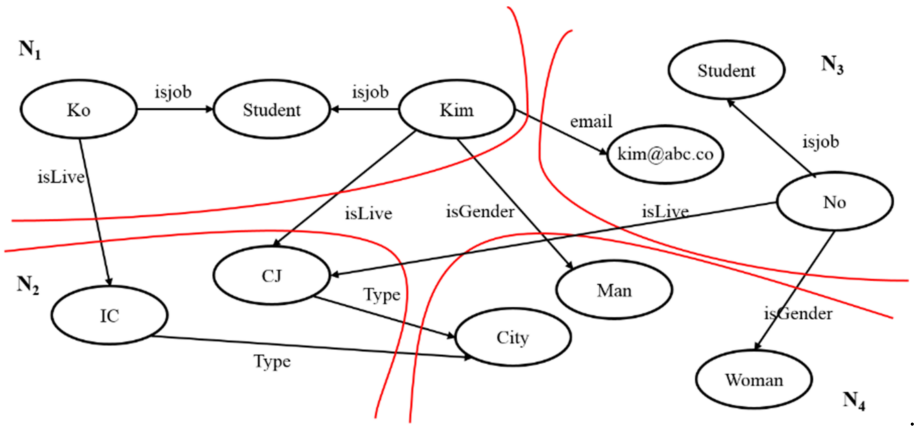


Figure 2. Example of an RDF graph.

When a query is requested, the query is divided into multiple subqueries based on the WHERE clause. The divided subqueries are grouped through the proposed RDF index first. The subqueries grouped through the index are then second grouped with related subqueries among S, P, and O. The data communication cost is reduced during query processing through the second grouping. In addition, the second-grouped queries are stored as a format of RDD and utilized. This can not only reduce the disk I/O cost incurred in existing schemes by storing queries at the RDD format inside the Spark but also achieve more efficient query processing than existing schemes through query processing in the in-memory environment.

A query that is optimized to distributed data should be sent to process RDF graph queries in a distributed environment. If queries that are unsuitable to distributed data are transferred and processed, efficient query processing cannot be conducted, and disk I/O cost and data communication cost are increased during query processing. Thus, efficient query processing can be conducted by decomposing a SPARQL query into subqueries. As shown in Figure 3, once the SPARQL query is entered, the WHERE clause in the SPARQL is divided. The WHERE clause is decomposed into subqueries. After being decomposed into subqueries, related nodes are searched through the index. The index is composed based on S, P, and O about all triple data. It has information about S, P, and O of each record, and information about all combinations of S, P, and O such as SP, PO, SO, or SPO, etc., are also stored. The ID numbers of the nodes that store the information are also stored. The index used is different according to the characteristics of the subqueries. For example, if the subject index's type is Name, all of the Name information is stored, and a node where the Name is stored is also stored along with the ID number of the node. If a data record

called Name is stored in multiple nodes, plural ID numbers of the nodes are stored. For example, if there is a query (Kim, isJob, ?A) among the subqueries, since "?A" in the object is a variable, the value is not known. In such a case, although the object is not known, the subject and predicate are known, thus that it can be searched through the SP index out of the indices.

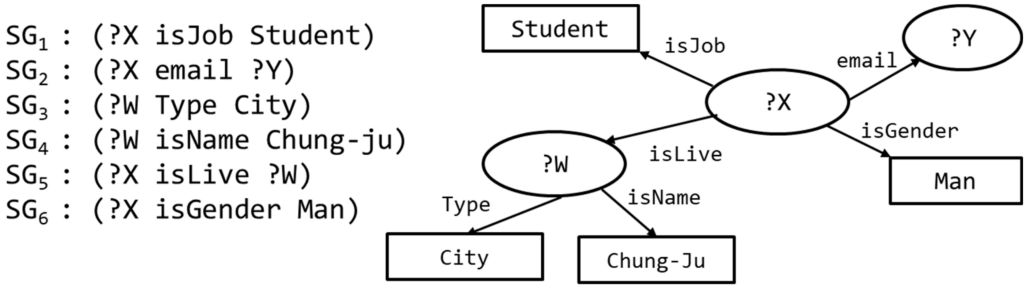


Figure 3. Example of a SPARQL query.

A query can be efficiently processed in the distributed stored RDF environment by decomposing a query entered by the client. However, in order to produce a final result when a single query is decomposed into multiple subqueries, intermediate results decomposed through communication should be made into a single final result with regard to query processing results of subqueries. A large amount of communication cost is incurred to have a single final result. To overcome this, subqueries are grouped with related subqueries, thereby reducing the communication cost during query processing.

If only related nodes of subqueries are known through the index, subqueries are only sent to the related nodes. For only adjacently connected subqueries, related node ID are compared. For example, if SG_1 is related to N_1 and N_2 and SG_2 is related to Nodes N_1, N_2, N_3, and N_4, SG_1 only knows predicate and object. Thus, SG_1 has a small number of related nodes compared to other subqueries. On the other hand, SG_2, which knows only predicates, may have many related nodes depending on how the stored data are distributed. In such a case, the communication cost may increase according to the number of related nodes during query processing. Thus, it is important to reduce the number of related nodes of SG_2 compared to that of SG_1, which is an adjacent subquery. The unnecessary data transfer during query processing and data parsing cost can be decreased by reducing the number of related nodes of SG_i through only adjacent nodes that are shared in common via the comparison of related nodes of SG_1 and SG_2.

Grouping of related nodes can be conducted through the related node and adjacent subqueries. In addition, related and unrelated subqueries are distinguished even inside each node. During grouping, S, P, and O of each SQ_i are compared. The second grouping is conducted based on SQ_i that have the same S, P, or O, or the same variables during the comparison. The join cost can be reduced by query processing of related SQ_i though second grouping. Figure 4 shows the subquery grouping by the node. In N_1, N_2, and N_3, only a single group is made since grouping is conducted through common variables in subqueries included in the related nodes. In contrast, in N_4, although SQ_3, SQ_4, and SQ_6 are related to N_4, SQ_3 and SQ_4 can be grouped into a single group through the "?W" variable. However, SQ_6 does not have S, P, and O that are related to SQ_3 and SQ_4, which cannot be grouped into a single group. Thus, in N_4, two groups are produced.

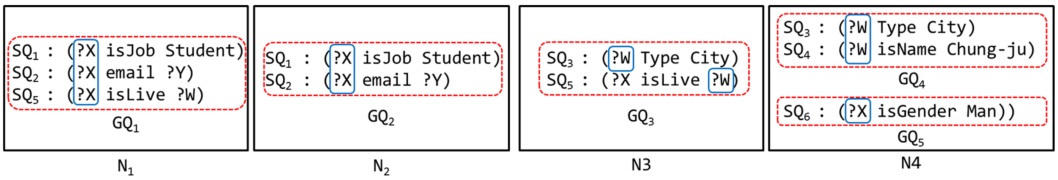


Figure 4. Grouping of subqueries by node.

3.3. Query Processing Procedure

The communication and join costs incurred when distributed intermediate results are combined into a single final result during query processing in the distributed stored RDF environment. In addition, a query processing order and joining method can increase or decrease the cost of query processing. Thus, the optimal query execution path is created to reduce the join and data communication costs incurred during query processing in the query processing execution path. A query execution path is created in consideration of data parsing, communication cost between nodes, and queuing time to reduce the query processing cost incurred during query processing, which is not taken into consideration in existing schemes. The join and communication costs can be reduced by performing query processing through the created query execution paths.

Figure 5 shows the overall query processing procedure. First, when a query is inputted, the proposed scheme decomposes the query. Once the subqueries grouped from query decomposition are entered, all possible query execution paths are generated. After all possible query execution paths are generated, the cost of performing the query paths is calculated through the proposed algorithm. The optimal query execution path is selected through the cost calculation, and queries are processed through the selected query execution path. We obtain a final result.

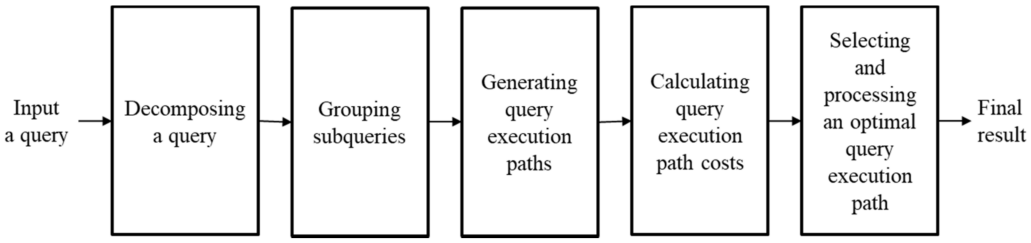


Figure 5. Query processing procedure.

Figure 5. Query processing procedure.

We cannot know if it is possible to process a query using which query execution path when processing it in the distributed stored RDF environment. Thus, we create all possible query execution paths based on the subqueries. A large amount of the join and communication costs can be incurred during query processing performed according to the created query execution paths. Thus, an efficient and optimal query execution path is selected from all possible executable query paths to process a query. The join and communication costs can be reduced during query processing by processing a query through the selection of the selected query execution path.

The optimal query path is selected through the cost calculation of the created query paths. First, the query cost of the group by node is calculated via Equation (1). $Parse(SQ_i)$ refers to the cost of data parsing for the SQ_i in the corresponding node. The query processing cost when the number of SQ_i in the corresponding node is only one is $Result(SQ_i)$. However, if the number of SQ_i is more than one, the query processing cost of the group by node is calculated including join cost $Join(Parse(SQ_i) + Result(SQ_{i-1}))$ among SQ_i.

$$Result(SQ_i) = \begin{cases} Join(Parse(SQ_i) + Result(SQ_{i-1})), & i > 1 \\ Parse(SQ_i), & i = 1 \end{cases} \quad (1)$$

The cost is calculated by dividing a case when the number of subqueries is one and two or more according to the group by node. When subqueries are connected adjacently inside the group, even if the number of subqueries is more than two, it is regarded as a single subquery to calculate the cost.

Once the cost for each group inside each node is calculated, all possible query paths are created through Equation (2) (ETE: Estimated Time Equation algorithm) based on the join variables between groups by node overall. After all the query paths are determined based

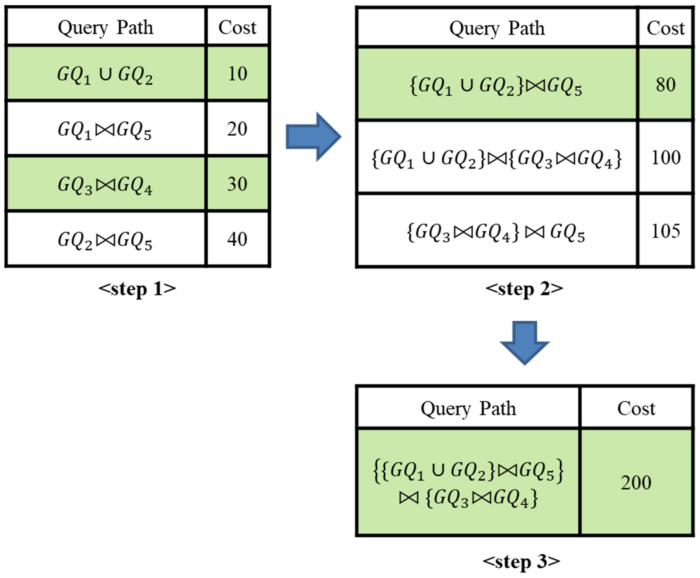
on the join variables between groups, the cost of the query paths is calculated. To calculate Equation (2), which node is used to process GQ_i (Groups by node) should be determined as presented in Equation (3). If GQ_1 and GQ_2 can be joined, and which GQ_i is moved is determined through Equation (3) to process a query. The cost is compared via Equation (3) between GQ_1 to $GQ_2\left(T^r_{GQ1-GQ2}\right)$ or from GQ_2 to $GQ_1\left(T^r_{GQ2-GQ1}\right)$, and GQ is moved to the less-cost direction. Once the path of GQ move is determined, whether it is a join operation $\left(T^j_{GQ1-GQ2}\right)$ or union operation $\left(T^u_{GQ1-GQ2}\right)$ is determined according to GQ_1 and GQ_2. For example, if GQ_1 and GQ_2 are not the same SG and have a join variable called B(X), a query path cost can be calculated through join operation. On the other hand, if GQ_1 and GQ_2 have the same SG, the query path cost is calculated through the union operation.

$$T^{total}_{GQ1-GQ2} = \begin{cases} T^{wait}_{GQ1-GQ2} + T^{j}_{GQ1-GQ2} \\ T^{wait}_{GQ1-GQ2} + T^{u}_{GQ1-GQ2} \end{cases} \tag{2}$$

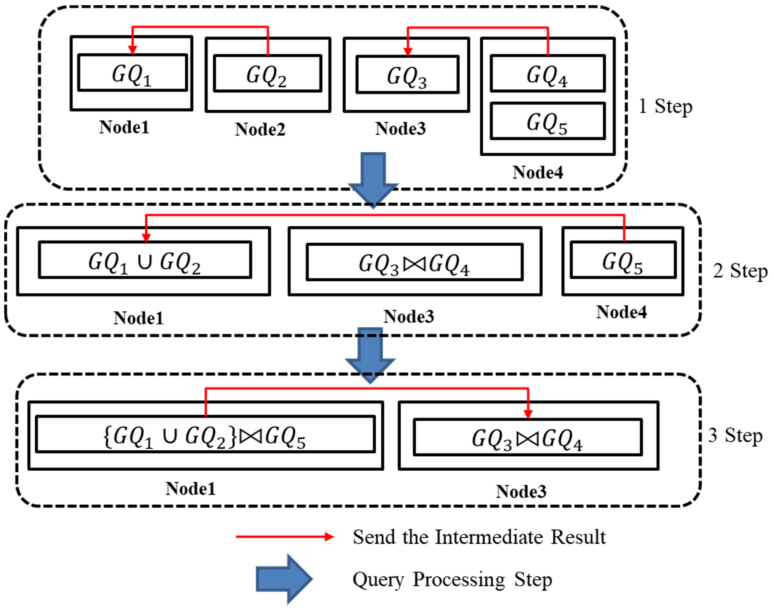
$$T^{wait}_{GQ1-GQ2} = \max\left\{\left(T^l_{GQ2} + T^r_{GQ2-GQ1}\right),\ T^l_{GQ1}\right\} \tag{3}$$

Once the join and union costs between groups by each node are calculated, the calculated costs are stored in the table in ascending order. The least cost of the query path stored in the table is selected first. When all the groups by all nodes are selected, the cost is calculated again through Equation (2) using the selected query path. Finally, this calculation is iterated until all groups by all nodes generate a single query path. Once the iterations are complete, the query paths determined through Equation (2) are sorted out sequentially to create a query execution path. Figure 6 shows the operation of the ETE algorithm. The cost is calculated through the ETE algorithm for the groups in each node according to whether the join operation can be performed. The least $GQ_1 \cup GQ_2$ is selected after comparing the calculated costs, and $GQ_3 \bowtie GQ_4$ is selected, which is the least cost calculated with groups other than the selected groups by node. Step 1 in Figure 6a shows the first iteration of the ETE algorithm, which is iterated until the order is determined after all groups are selected. Figure 6b shows the query processing order based on the groups by node.

Once the query processing execution path is determined, the master sends the query processing execution path to each node and RDDs to the slave. The slaves that receive the RDDs perform query processing tasks according to the query processing execution path. The node that has the final result sends it to the master again to finish the query processing.



(**a**)



(**b**)

Figure 6. ETE algorithm operation: (**a**) step to select the optimal query path; (**b**) query processing order for each node.

4. Performance Evaluation

The performance evaluation compares the execution results between the master–slave system structure proposed in this paper and an existing scheme. The compared existing scheme was a distributed SPARQL query processing-related scheme over the Spark environment. The existing scheme considered distributed storage of a large amount of RDF data, but it did not consider data communication and join costs. Table 1 shows an experimental environment for the performance evaluation. Four virtual nodes were built. One was used as a master, and the other three nodes were used as slaves. The performance was compared between the SPARQL query processing scheme [43] over the Spark environment, which is an in-memory environment, and the existing scheme.

Table 1. Experimental environment.

Parameter	Value
Processor	Intel(R) Core(TM) i5
Memory	64 GB
No. of nodes	Master 1, Slave 3
Engine used	Spark 2.4.5

For the performance evaluation, LUBM dataset [49] and DBpedia dataset [50] were used. Approximately 230,000, 430,000, and 910,000 records of RDF graph from the LUBM dataset and 300,000 records of RDF graph from the DBpedia dataset were used, which were distributed and stored in the three slave nodes. For SPARQL queries, four queries from the LUBM dataset and three queries from the DBpedia dataset were used in accordance with the dataset used in this performance evaluation. Q1 and Q2 in the LUBM dataset had a very small number of intermediate results from the nodes, entailing that data transfer between nodes was small. Q3 was only considered when query processing was performed in nodes whose number of necessary data for query processing was two. Q4 had a large number of intermediate results from the nodes, and a large amount of data were transferred during data transfer. In addition, Q1, Q2, and Q3 from DBpedia had a different size of data used during query processing.

Figure 7 shows the comparison results of performances of the proposed scheme and when grouping of subqueries is not considered during query decomposition (group) and when optimal query execution path is not considered in the query execution path (optimal query execution). The performance was also compared with four different queries at the same data size. The query processing time was compared according to queries without considering query decomposition and query execution path in the proposed scheme. The performance evaluation results showed that Q1 and Q2 had a relatively small size of intermediate data, resulting in a similar query processing time with that of the proposed scheme was revealed, while Q3 and Q4 exhibited a performance difference between the proposed and existing schemes according to the intermediate data size and the number of subqueries. When grouping was not considered in query decomposition, the intermediate data size and join cost was increased in the existing scheme compared to that of the proposed scheme. Similarly, when the optimal query execution path was not considered in the query execution path, the communication cost was increased in the existing scheme compared to that of the proposed scheme.

Figure 7. Performance difference by considering query decomposition and query execution path.

Figure 8 shows the comparison results of query processing time between the existing and proposed schemes. Performances were compared through different queries while changing a data size through various datasets. The proposed scheme showed better performance results than the existing scheme as queries became more complex and more intermediate results were obtained through corresponding queries. In addition, as the number of nodes during query processing increased, performance improved. However, as the number of used nodes decreased, no significant difference in performance was found. The query processing time was increased in the existing scheme as a query became more complex since the communication and join costs were taken into consideration during query processing. The proposed scheme removed this problem thus that it achieved performance improvements of around 15% on average through the query processing scheme considering the communication and join costs.

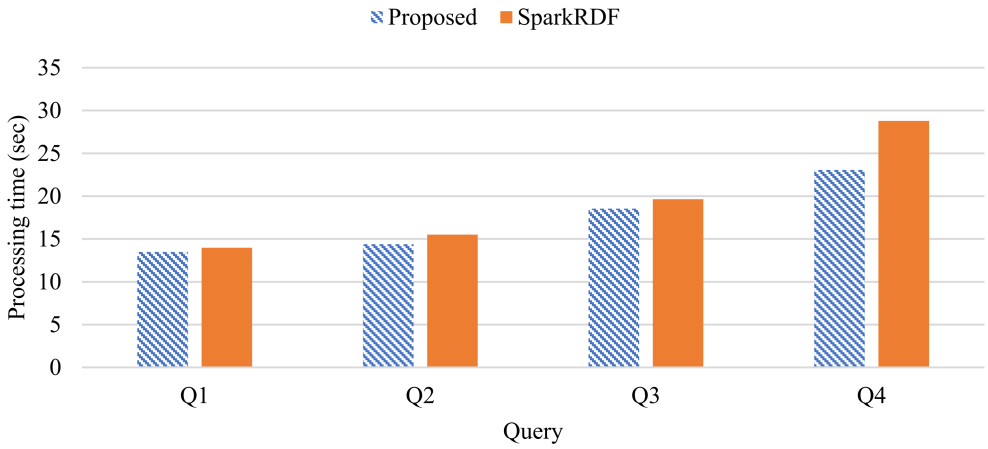
Table 2 presents the comparison results of query processing time incurred per node between the existing and proposed schemes through the LUBM dataset. Data parsing time was measured in a node that executed the query the first time according to the query processing path while processing time of data received from other nodes and parsed data from the current node was measured in other nodes and compared. The performance evaluation was conducted based on four different queries, the same as in the first performance evaluation. For Q3, query processing was conducted based on two nodes, in which query processing time was compared between N1 and N2. The performance evaluation results showed that processing time was calculated differently by the node according to the query execution path between the existing and proposed schemes, and query processing time was increased according to the data size used in join operation in all nodes.

Table 2. Query processing time per node.

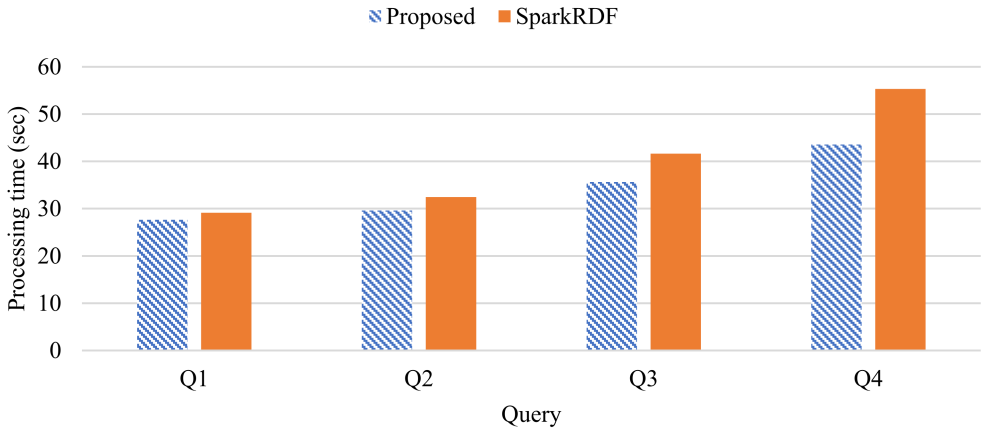
(s)	N1		N2		N3	
	Existing Scheme	**Proposed Scheme**	**Existing Scheme**	**Proposed Scheme**	**Existing Scheme**	**Proposed Scheme**
Q1	2.578	2.557	2.532	2.542	3.421	3.411
Q2	2.571	2.671	2.515	2.424	2.997	2.987
Q3	3.784	3.532	3.564	3.487	X	X
Q4	7.601	7.710	7.331	6.109	1.927	1.887

(**a**)

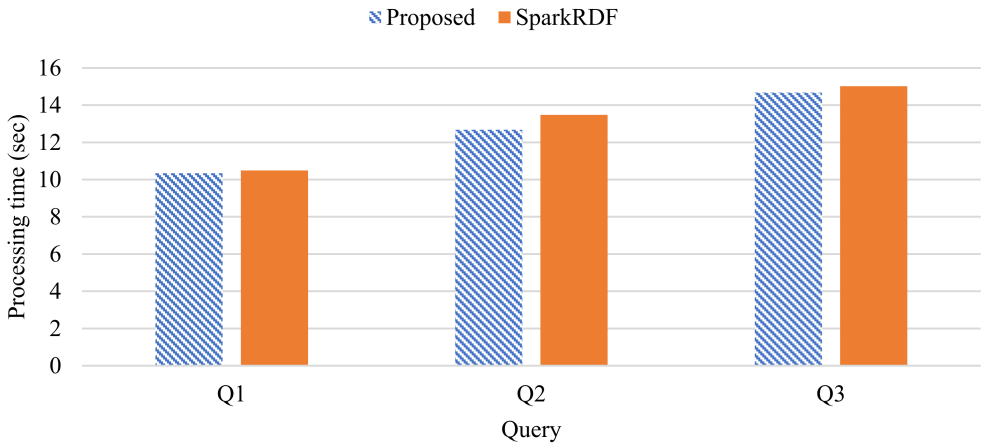
Figure 8. *Cont.*



(**b**)



(**c**)



(**d**)

Figure 8. Query processing time: (**a**) LUBM 230,000 dataset; (**b**) LUBM 430,000 dataset; (**c**) LUBM 910,000 dataset; (**d**) DBpedia 300,000 dataset.

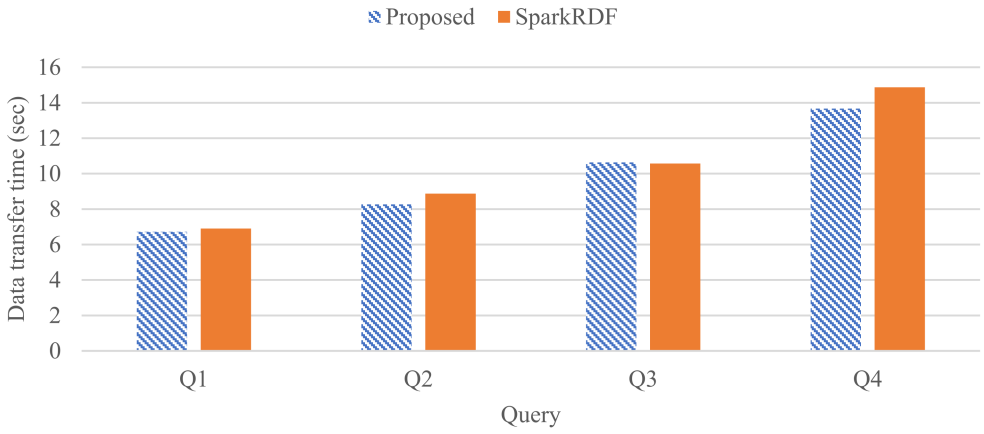
Figure 9 shows the comparison results of data transfer time incurred during query processing between the existing and proposed schemes. Performances were compared through different queries while changing data size through various datasets. Time to transfer the intermediate result data and final result data between nodes to slaves and master was considered. The performance evaluation results showed that data transfer time was increased in the existing scheme as the amount of intermediate data was larger since the query processing order according to the communication cost during query processing was not considered. However, the proposed scheme overcame this problem through the query processing execution path considering the data communication amount and join cost. The proposed scheme thereby reduced a transfer time during query processing on average by about 10% compared to the existing scheme.

Figure 10 shows the comparison results of query processing time according to dataset size between the existing and proposed schemes. In addition to the datasets used previously, approximately 430,000 and 910,000 RDF datasets were used. The performance evaluation was also conducted through two different queries. Figure 10a shows the small communication and join costs due to fewer intermediate results than other queries, whereas Figure 10b shows the large communication and join costs incurred due to larger intermediate results. The performance evaluation results showed that the proposed scheme improved performance considering the communication and join costs during query processing, which were not considered in existing schemes. In particular, the proposed scheme showed a better performance than the existing scheme when the dataset size became larger.

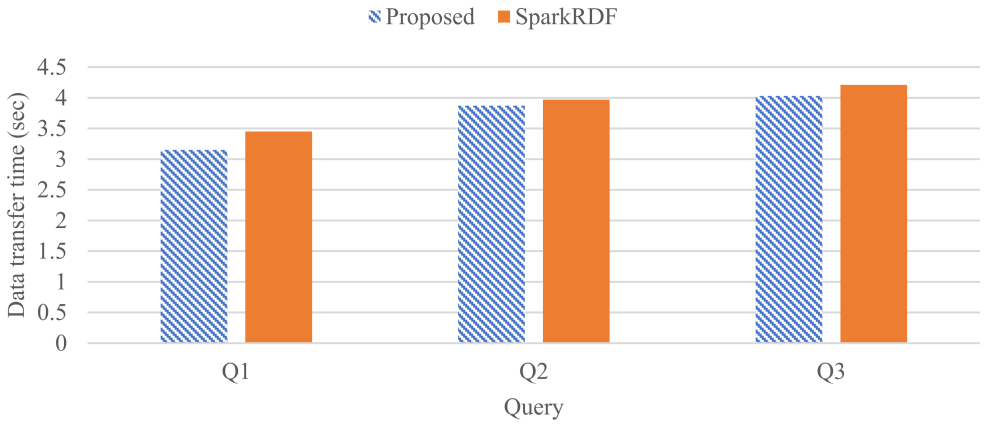
(a)

Figure 9. *Cont.*

(**b**)



(**c**)



(**d**)

Figure 9. Data transfer time: (**a**) LUBM 230,000 dataset; (**b**) LUBM 430,000 dataset; (**c**) LUBM 910,000 dataset; (**d**) DBpedia 300,000 dataset.

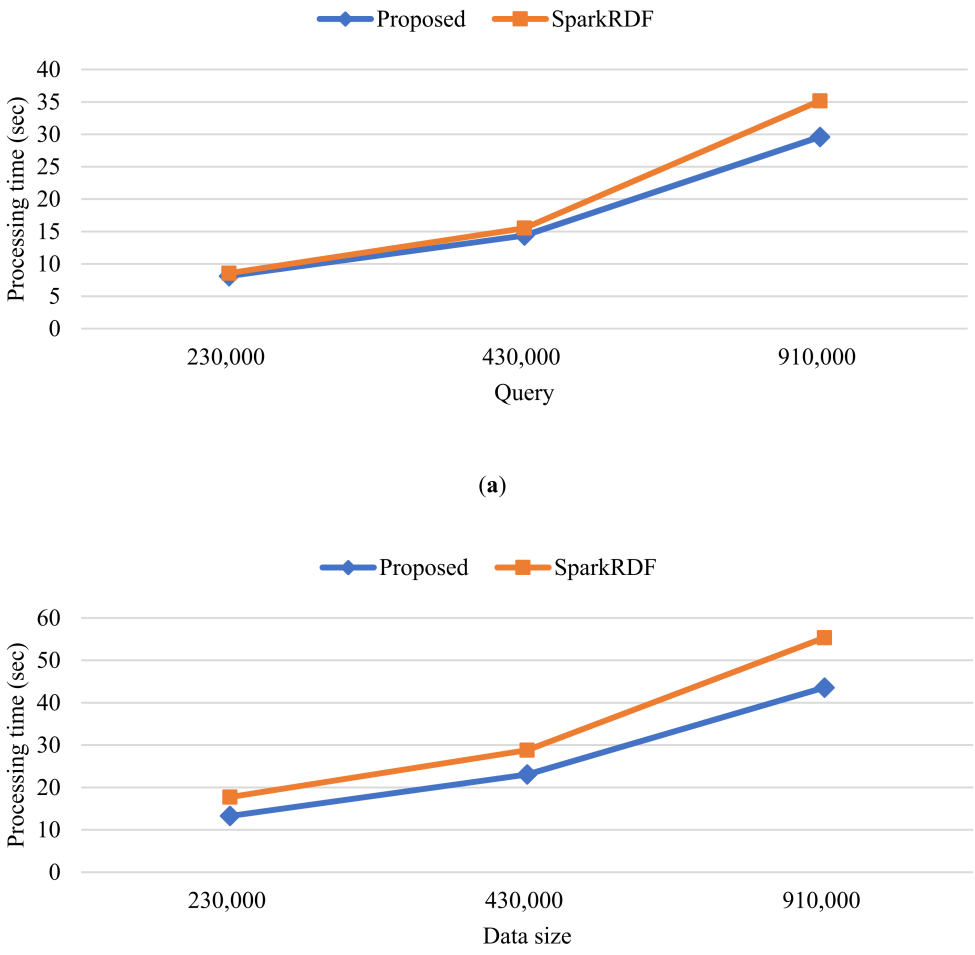


Figure 10. Query processing time by data size: (**a**) comparison through query whose intermediate result is small; (**b**) comparison through query whose intermediate result is large.

5. Conclusions

This paper proposed a distributed SPARQL query processing scheme considering communication costs between nodes in Spark environments. In the proposed scheme, the disk I/O cost incurred during query processing while parsing a large amount of RDF data, which was a problem in existing schemes, was reduced as the disk I/O was processed in the Spark environment (an in-memory environment). In addition, the proposed scheme performed query processing through the query execution path created considering data parsing cost, data communication cost, join cost, and queuing time. The performance evaluation results showed that the proposed scheme improved the performance of query processing time, transfer time, and query processing time in each node by around 15% compared to an existing scheme, which was a distributed SPARQL query processing-related scheme in the Spark environment. In addition, no or large difference in performance was revealed between them, according to the query used. If a query was simple, thus that the number of nodes used during query processing was small, or if the number of query results was smaller, no significant difference in performance was exhibited between the existing and proposed schemes. However, when a query was complex, or the size of query results was large, a performance difference was large. In the near future, a study on load management between nodes during query processing will be conducted.

Author Contributions: Conceptualization, J.L., B.K., H.L., D.C., K.B. and J.Y.; methodology, J.L., B.K., H.L., D.C., K.B. and J.Y.; validation, J.L., B.K., H.L., D.C. and K.B.; formal analysis, J.L., B.K., H.L., D.C. and K.B.; writing—original draft preparation, J.L., B.K., H.L., D.C. and K.B.; writing—review and editing, J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Cooperative Research Program for Agriculture Science and Technology Development (Project No. PJ01624701) Rural Development Administration, Republic of Korea, the MSIT(Ministry of Science and ICT), Korea, under the Grand Information Technology Research Center support program (IITP-2021-2020-0-01462) supervised by the IITP (Institute for Information and communications Technology Planning and Evaluation), Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2014-3-00123, Development of High-Performance Visual Big Data Discovery Platform for Large-Scale Realtime Data Analysis), and Korea Institute for Advancement of Technology(KIAT) grant funded by the Korea Government(MOTIE) (P0008421, The Competency Development Program for Industry Specialist).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Antoniou, G.; Harmelen, F.V. *A Semantic Web Primer*; MIT Press: Cambridge, MA, USA, 2004.
2. Shadbolt, N.; Berners-Lee, T.; Hall, W. The Semantic Web Revisited. *IEEE Intell. Syst.* **2006**, *21*, 96–101. [CrossRef]
3. Carroll, J.; Dickinson, I.; Dollin, C.; Reynolds, D.; Seaborne, A.; Wilkinson, K. Jena: Implementing the Semantic Web Recommendations. In Proceedings of the International Conference on World Wide Web—Alternate Track Papers & Posters, New York, NY, USA, 19–21 May 2004; pp. 74–83.
4. Hassanzadeh, O.; Kementsietsidis, A.; Velegrakis, Y. Data Management Issues on the Semantic Web. In Proceedings of the International Conference on Data Engineering, Arlington, VA, USA, 1–5 April 2012; pp. 1204–1206.
5. RDF 1.1 Concepts and Abstract Syntax. Available online: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (accessed on 7 December 2021).
6. Decker, S.; Melnik, S.; Harmelen, F.V.; Fensel, D.; Klein, M.C.A.; Broekstra, J.; Erdmann, M.; Horrocks, I. The Semantic Web: The Roles of XML and RDF. *IEEE Internet Comput.* **2000**, *4*, 63–74. [CrossRef]
7. Broekstra, J.; Kampman, A.; Harmelen, F.V. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Proceedings of the International Semantic Web Conference, Sardinia, Italy, 9–12 June 2002; pp. 54–68.
8. Picalausa, F.; Luo, Y.; Fletcher, G.H.L.; Hidders, J.; Vansummeren, S. A Structural Approach to Indexing Triples. In Proceedings of the Extended Semantic Web Conference, Heraklion, Greece, 27–31 May 2012; pp. 406–421.
9. Neumann, T.; Weikum, G. The RDF-3X engine for scalable management of RDF data. *VLDB J.* **2010**, *19*, 91–113. [CrossRef]
10. Kang, S.; Shim, J.; Lee, S. Tridex: A lightweight triple index for relational database-based Semantic Web data management. *Expert Syst. Appl.* **2013**, *40*, 3421–3431. [CrossRef]
11. SPARQL 1.1 Overview. Available online: https://www.w3.org/TR/sparql11-overview/ (accessed on 16 December 2021).
12. Kim, K.; Moon, B.; Kim, H. R3F: RDF triple filtering method for efficient SPARQL query processing. *World Wide Web* **2015**, *18*, 317–357. [CrossRef]
13. Hassan, M.; Bansal, K.S. RDF Data Storage Techniques for Efficient SPARQL Query Processing Using Distributed Computation Engines. In Proceedings of the International Conference on Information Reuse and Integration, Salt Lake City, UT, USA, 6–9 July 2018; pp. 323–330.
14. Bonifati, A.; Martens, W.; Timm, T. An analytical study of large SPARQL query logs. *VLDB J.* **2020**, *29*, 655–679. [CrossRef]
15. Kim, K.; Moon, B.; Kim, H. RG-index: An RDF graph index for efficient SPARQL query processing. *Expert Syst. Appl.* **2014**, *41*, 4596–4607. [CrossRef]
16. Huang, J.; Abadi, D.J.; Ren, K. Scalable SPARQL Querying of Large RDF Graphs. *VLDB Endow.* **2011**, *4*, 1123–1134. [CrossRef]
17. Kharrat, M.; Jedidi, A.; Gargouri, F. SPARQL Query Generation Based on RDF Graph. In Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, Porto, Portugal, 9–11 November 2016; pp. 450–455.
18. Wu, B.; Zhou, Y.; Yuan, P. Scalable SPARQL Querying Using Path Partitioning. In Proceedings of the International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 795–806.
19. Hu, C.; Wang, X.; Yang, R.; Wo, T. ScalaRDF: A Distributed, Elastic and Scalable In-Memory RDF Triple Store. In Proceedings of the International Conference on Parallel and Distributed Systems, Wuhan, China, 13–16 December 2016; pp. 593–601.

20. Wang, X.; Yang, T.; Chen, J.; He, L.; Du, X. RDF partitioning for scalable SPARQL query processing. *Front. Comput. Sci.* **2015**, *9*, 919–933. [CrossRef]
21. Galárraga, L.; Hose, K.; Schenkel, R. Partout: A distributed engine for efficient RDF processing. In Proceedings of the International World Wide Web Conference, Seoul, Korea, 7–11 April 2014; pp. 267–268.
22. Guo, X.; Gao, H.; Zou, Z. Leon: A Distributed RDF Engine for Multi-query Processing. In Proceedings of the International Conference on Database Systems for Advanced Applications, Chiang Mai, Thailand, 22–25 April 2019; pp. 742–759.
23. Potter, A.; Motik, B.; Nenov, Y.; Horrocks, I. Dynamic Data Exchange in Distributed RDF Stores. *IEEE Trans. Knowl. Data Eng.* **2018**, *30*, 2312–2325. [CrossRef]
24. Naacke, H.; Curé, O. On distributed SPARQL query processing using triangles of RDF triples. *Open J. Semant. Web* **2020**, *7*, 17–32.
25. Jabeen, H.; Haziiev, E.; Sejdiu, G.; Lehmann, J. Dise: A Distributed in-Memory Sparql Processing Engine over Tensor Data. In Proceedings of the IEEE 14th International Conference on Semantic Computing (ICSC), San Diego, CA, USA, 3–5 February 2020; pp. 400–407.
26. Hassan, M.; Bansal, S.K. S3QLRDF: Property Table Partitioning Scheme for Distributed SPARQL Querying of Large-Scale RDF data. In Proceedings of the IEEE International Conference on Smart Data Services (SMDS), Online, 18–24 October 2020; pp. 133–140.
27. Lu, J.; Yang, C.; Wang, B.; Feng, J. FP-ExtVP: Accelerating Distributed SPARQL Queries by Exploiting Load-Adaptive Partitioning. In Proceedings of the IEEE International Conference on Big Data (Big Data), Online, 10–13 December, 2020; pp. 543–550.
28. Ragab, M.; Eyvazov, S.; Tommasini, R.; Sakr, S. *Systematic Performance Analysis of Distributed SPARQL Query Answering Using Spark-SQL*; IOP Press: Bristol, UK, 2020; pp. 1–21.
29. Kang, X.; Zhao, Y.; Yuan, P.; Jin, H. Grace: An Efficient Parallel SPARQL Query System over Large-Scale RDF Data. In Proceedings of the IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Dalian, China, 5–7 May 2021; pp. 769–774.
30. Leng, Y.; Chen, Z.; Zhong, F.; Li, X.; Hu, Y.; Yang, C. BRGP: A balanced RDF graph partitioning algorithm for cloud storage. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e3896. [CrossRef]
31. Padiya, T.; Bhise, M. DWAHP: Workload Aware Hybrid Partitioning and Distribution of RDF Data. In Proceedings of the International Database Engineering & Applications Symposium, Bristol, UK, 12–14 July 2017; pp. 235–241.
32. Zeng, K.; Yang, J.; Wang, H.; Shao, B.; Wang, Z. A distributed graph engine for web scale RDF data. *VLDB Endow.* **2013**, *6*, 265–276. [CrossRef]
33. Ravindra, P.; Anyanwu, K. Nesting Strategies for Enabling Nimble MapReduce Dataflows for Large RDF Data. *Proc. Int. J. Semant. Web Inf. Syst.* **2014**, *10*, 1–26. [CrossRef]
34. Elzein, N.M.; Majid, M.A.; Hashem, I.A.T.; Yaqoob, I.; Alaba, F.A.; Imran, M. Managing big RDF data in clouds: Challenges, opportunities, and solutions. *Sustain. Cities Soc.* **2018**, *39*, 375–386. [CrossRef]
35. Quilitz, B.; Leser, U. Querying Distributed RDF Data Source with SPARQL. In Proceedings of the European Semantic Web Conferences, Tenerife, Spain, 1–5 June 2008; pp. 524–538.
36. Feng, J.; Meng, C.; Song, J.; Zhang, X.; Feng, Z.; Zou, L. SPARQL Query Parallel Processing: A Survey. In Proceedings of the International Congress on Big Data, Honolulu, HI, USA, 25–30 June 2017; pp. 444–451.
37. Papailiou, N.; Konstantinou, I.; Tsoumakos, D.; Karras, P.; Koziris, N. H2RDF+: High-Performance Distributed Joins over Large-Scale RDF Graphs. In Proceedings of the IEEE International Conference on Big Data, Silicon Valley, CA, USA, 6–9 October 2013; pp. 255–263.
38. Wylot, M.; Hauswirth, M.; Cudré-Mauroux, P.; Sakr, S. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* **2018**, *51*, 84. [CrossRef]
39. Leida, M.; Chu, A. Distributed SPARQL Query Answering over RDF Data Streams. In Proceedings of the International Congress on Big Data, Santa Clara, CA, USA, 27 June–2 July 2013; pp. 369–378.
40. Abdelaziz, I.; Harbi, R.; Khayyat, Z.; Kalnis, P. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *VLDB Endow.* **2017**, *10*, 2049–2060. [CrossRef]
41. Zhou, J.; Bochmann, G.V.; Shi, Z. Distributed Query Processing in an Ad-Hoc Semantic Web Data Sharing System. In Proceedings of the International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, 20–24 May 2013; pp. 687–695.
42. Hammoud, M.; Rabbou, D.A.; Nouri, R.; Beheshti, S.; Sakr, S. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *VLDB Endow.* **2015**, *8*, 654–665. [CrossRef]
43. Chen, X.; Chen, H.; Zhang, N.; Zhang, S. SparkRDF: Elastic Discreted RDF Graph Processing Engine with Distributed Memory. In Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology, Singapore, 6–9 December 2015; pp. 292–300.
44. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache Spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
45. Li, M.; Tan, J.; Wang, Y.; Zhang, L.; Salapura, V. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In Proceedings of the Conference on Computing Frontiers, Ischia, Italy, 18–21 May 2015; pp. 1–8.

46. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
47. Zhang, M.; Chen, R.; Zhang, X.; Feng, Z.; Rao, G.; Wang, X. Intelligent RDD Management for High Performance In-Memory Computing in Spark. In Proceedings of the International Conference on World Wide Web Companion, Perth, Australia, 3–7 April 2017; pp. 873–874.
48. Agathangelos, G.; Troullinou, G.; Kondylakis, H.; Stefanidis, K.; Plexousakis, D. RDF Query Answering Using Apache Spark: Review and Assessment. In Proceedings of the International Conference on Data Engineering Workshops, Paris, France, 16–20 April 2018; pp. 54–59.
49. The LUBM Benchmark. Available online: http://swat.cse.lehigh.edu/projects/lubm/ (accessed on 6 December 2021).
50. DBpedia. Available online: http://wiki.dbpedia.org/ (accessed on 6 December 2021).