Article

JQPro:Join Query Processing in a Distributed System for Big RDF Data Using the Hash-Merge Join Technique

Nahla Mohammed Elzein [1], Mazlina Abdul Majid [2], Ibrahim Abaker Targio Hashem [3,*], Ashraf Osman Ibrahim [4,5,*], Anas W. Abulfaraj [6] and Faisal Binzagr [7]

1 Faculty of Computer Science, Future University, Khartoum 10553, Sudan
2 Faculty of Computing, University Malaysia Pahang, Pekan 26600, Malaysia
3 Department of Computer Science, College of Computing and Informatics, University of Sharjah, Sharjah 27272, United Arab Emirates
4 Data Science Programme, Faculty of Computing and Informatics, Universiti Malaysia Sabah, Kota Kinabalu 88400, Malaysia
5 Advanced Machine Intelligence Research Group, Universiti Malaysia Sabah, Kota Kinabalu 88400, Malaysia
6 Department of Information Systems, King Abdulaziz University, P.O. Box 344, Rabigh 21911, Saudi Arabia
7 Department of Computer Science, King Abdulaziz University, P.O. Box 344, Rabigh 21911, Saudi Arabia
* Correspondence: ihashem@sharjah.ac.ae (I.A.T.H.); ashrafosman@ums.edu.my (A.O.I.)

Abstract: In the last decade, the volume of semantic data has increased exponentially, with the number of Resource Description Framework (RDF) datasets exceeding trillions of triples in RDF repositories. Hence, the size of RDF datasets continues to grow. However, with the increasing number of RDF triples, complex multiple RDF queries are becoming a significant demand. Sometimes, such complex queries produce many common sub-expressions in a single query or over multiple queries running as a batch. In addition, it is also difficult to minimize the number of RDF queries and processing time for a large amount of related data in a typical distributed environment encounter. To address this complication, we introduce a join query processing model for big RDF data, called JQPro. By adopting a MapReduce framework in JQPro, we developed three new algorithms, which are hash-join, sort-merge, and enhanced MapReduce-join for join query processing of RDF data. Based on an experiment conducted, the result showed that the JQPro model outperformed the two popular algorithms, gStore and RDF-3X, with respect to the average execution time. Furthermore, the JQPro model was also tested against RDF-3X, RDFox, and PARJs using the LUBM benchmark. The result showed that the JQPro model had better performance in comparison with the other models. In conclusion, the findings showed that JQPro achieved improved performance with 87.77% in terms of execution time. Hence, in comparison with the selected models, JQPro performs better.

Keywords: semantic web; distributed computing; RDF; big data; SPARKSQL

MSC: 68P05

Citation: Elzein, N.M.; Majid, M.A.; Hashem, I.A.T.; Ibrahim, A.O.; Abulfaraj, A.W.; Binzagr, F. JQPro:Join Query Processing in a Distributed System for Big RDF Data Using the Hash-Merge Join Technique. *Mathematics* **2023**, *11*, 1275. https://doi.org/10.3390/math11051275

Academic Editors: Codruta Mare, Ioana Florina Coita and Christophe Chesneau

Received: 29 December 2022
Revised: 21 February 2023
Accepted: 27 February 2023
Published: 6 March 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Over the last several years, Semantic Web technology in various research fields has grown, with persistent growth in its information collection. Hundreds of millions of triple data sets are now readily accessible by web indexing, creeping, and network connectivity initiatives [1]. Large-scale data can be collected from the Internet of Things (IoT), mobile devices, traffic management, network monitoring, sensor applications, manufacturing processes, blogging, and emails. Usually, the current size of this data ranges from terabytes to petabytes. Hence, dealing with this amount of data is challenging due to the required resilient and large-scale methods. The Semantic Web, which is endorsed by the world, promotes concepts and international standards in data envelopment analysis at common web open interfaces [2]. The demand for complex multiple RDF queries is becoming significant

with the increasing number of RDF triples. Such complex queries occasionally produce many common subexpressions. A single query or several runs mostly as batches. It is therefore extremely challenging to reduce the amount of RDF queries and transmission time for a vast number of related data collected in the publicly available typical distributed setting.

The recent rise in the amount of data collected is astonishing, with the growth rate surpassing the capacity to design appropriate data storage and analysis systems for effective data processing. Resource Description Framework (RDF) technologies commonly use an interactive data structure for Semantic Web. Nowadays, RDF triples are becoming web-scale graph datasets available via the internet, such as DBpedia, UniProt RDF, LUBM, and YAGO2 [3,4]. Therefore, RDF datasets that surpass hundreds of thousands of triples data are on the increase in various RDF databases. Figure 1 shows an example of the RDF graph. RDF databases consist of a triple, which is an atomic data entity called a graph data model, and are used for SPARQL queries. The W3C standard RDF query language is SPARQL, which can be used with a large number of data sets to process your database in RDF format. SPARQL queries may take different forms in practice, as shown in Figure 2. The main SPARQL queries with a general modification that sets, then rearranges [5].

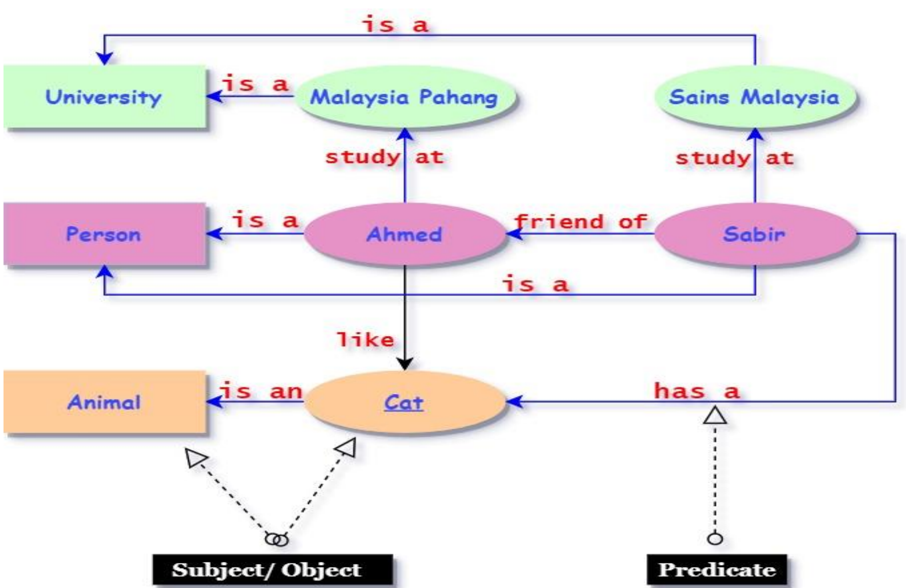

Figure 1. Example of the RDF graph.

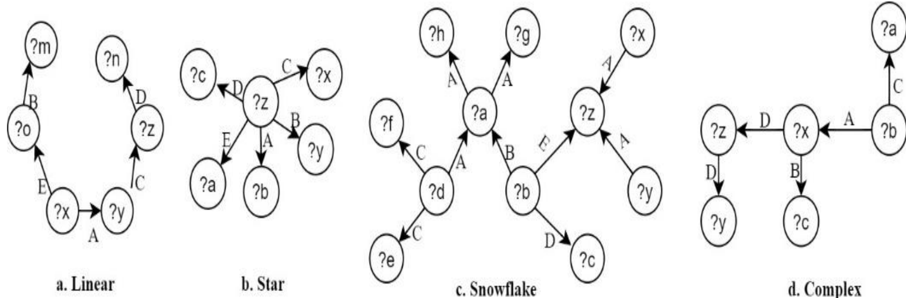

Figure 2. SPARQL BGP queries difference forms.

Early RDF data management research activities resulted in powerful big RDF systems such as RDF-3X. Although central data storage is not well suited to complicated web-scale RDF queries, as such, distributed RDF management systems are implemented by splitting RDF data between multiple device nodes and transmitting queries [6]. A query from SPARQL is subdivided into several sub-requests that are evaluated separately by each node. When data are distributed, intermediate results can be shared during query evaluation [7].

Distributed processing models were used to optimize RDF queries, such as map-side joins, heuristic distributed plan generation, reordering, cost-based optimizations, and join-optimistic triple reordering, which demonstrated that the runtime of large RDF data and the complexity of the query statement were relatively high [8–10]. As such, a high number of self-joins on the relational table are needed for the querying of RDF triple store databases that poorly scale into state-of-the-art RDF databases, which is time-consuming [11–13]. Additionally, most existing models have used parallel access to node-distributed RDF triples to distribute the entire workload among nodes containing replicated RDF data. However, parallel processes are complicated because of a sequential job, which results in multiple RDF triples running longer. This is more complicated, particularly if a large amount of RDF data is running [14].

According to Chantrapornchai and Choksuchat [15] and Jarrar and Dikaiakos [16], path expressions and access to multiple properties for resources with the subject being used in relation tables also require the complexity of requesting graph data. Furthermore, Husain and McGlothlin's [9] study found that there is a gap in the distributed data management repository and that the planer's greed does not consider the selectivity of join. For selective queries, a large overhead is induced, as only works are carried out with MapReduce. Therefore, due to the growing volume of RDF data [17,18] and Linked Data distributed query processing, some difficulties in RDF storage and distributed query processing over Linked Data have been identified [19]. In light of these issues, an effective query processing model is needed that addressed the issue of RDF data.

The objective of this paper is as follows:

- To develop a novel join query processing model for big RDF data.
- To adopt a MapReduce framework in the developed model to aid in RDF data reduction and sorting.
- The proposed JQPro is composed of three newly developed algorithms, which are hash-join, sort-merge, and enhanced MapReduce-join for join query processing of RDF data.
- To process the join query of RDF data by utilizing the HIVE and MapReduce strategy for SPARQL queries.
- The contribution of this paper contains:
- A novel model JQPro in a distributed system for big RDF data using the hash-merge join technique.
- Adopting a MapReduce framework in JQPro, by developing three new algorithms, which are hash-join, sort-merge, and enhanced MapReduce-join for join query processing of RDF data.
- Conducting an extensive experiment on the Lehigh University Benchmark (LUBM) and the Waterloo SPARQL Diversity Test Suite (WatDiv) v06 benchmarks to evaluate and assess the proposed JQPro model.

The rest of the paper is described as follows. Section 2 contains the related works. Section 3 introduces the developed model. Section 4 outlines the general experiment setup, including the benchmarks and datasets utilized. The results and discussion are presented in Section 5. The study is concluded in Section 6.

2. Related Work

In the last few years, a substantial amount of literature has been published on distributed-based processing models for big RDF data, with various models proposed [8–10,20,21]. Abdelaziz, Harbi [22], and Özsu [23] provide a survey and comparative study on distributed SPARQL engines for very large RDF data. The author has selected 22 recent studies concerning distributed RDF data processing. The results of the comparison have been made online for researchers to use as benchmarks. In this section, we summarize some of the existing models developed for big RDF data.

CliqueSquare is a novel optimization technique introduced by Goasdoué, Kaoudi [24] for assessing conjunctive RDF queries in a massively parallel context. The aim is to limit

the number of joins encountered on a root-to-leaf path in the plan to reduce query response time. ExtVP is a revolutionary relational partitioning schema for RDF data presented by Schätzle, Przyjaciel-Zablocki [25] that employs semi-join-based preprocessing, similar to the notion of join indices in relational databases, to efficiently decrease query input size independent of pattern form and diameter. To answer subgraph matching queries on large RDF networks, Xu, Wang [26] present SP-Tree, a novel distributed subgraph matching approach based on the Pregel model. The approach is based on the query graph, which is turned into a spanning variation.

Guo, Gao [27] introduced a model named Leon. Leon is a distributed RDF system that deals with multi-query problems. The idea is to use a characteristically determined partitioning scheme to support complete parallel processing with a set of features to reduce data communication by direct transmission of intermediate results rather than broadcasting. In the context of RDF/SPARQL, Leon reviews the traditional issue of multi-query optimization. In view of the NP-hardness of SPARQL multi-query optimization, the author has presented a heuristic algorithm that partitions the input batch of queries into groups to identify the common sub-query of several SPARQL queries. Gai, Wang's [28] research also looks at the restricted support for queries with more extensive SPARQL specifications, as well as semantic-preserving translation from SPARQL to SQL. The authors provide ROSIE, a runtime optimization system that repeatedly re-optimizes the SPARQL query plan based on the partial incremental query evaluation's real cardinality. During runtime, the model employs heuristic plan generation and a system for identifying cardinal errors in estimation. The authors tackled the problem of suboptimal query plans caused by an error-prone estimation of cardinality. Extensive trials on real-world and benchmarking data revealed that, when compared to state-of-the-art queries, ROSIE regularly outperformed rival models by orders of magnitude.

RDF-3X [29] uses a giant triple table; six B+ clusters must be updated to manage actual updates. However, RDF-3X supports correct SPARQL queries only, and wildcard requests fail to support them. As such, with the underlying RDF repositories, RDF-3X cannot perform web updates effectively. Moreover, for starters, if there are updates of properties in RDF triples in clustered property table-based methods, the re-cluster property has to be performed.

Zou, Özsu [30] created a new gStore approach to effectively answer SPARQL queries. In the developed system, RDF data were processed into a large graph to represent the SPARQL database as a query graph. Additionally, a pruning rule index was developed to achieve scalable and effective processing of queries. For managing web updates over RDF repositories, an efficient maintenance approach was also used. The result of the experiment demonstrates that the accuracy of the system developed was best related to the existing systems. To deal with the missing attribute values in the dataset, the developed system lags.

Hadoop MapReduce is presented in Husain, Doshi [31], as a new framework for manipulating large quantities of RDF data graphs. A high-fault tolerance of the Hadoop distributed file system and MapReduce is automatically accepted via the algorithm on top of the framework to define the baseline processing plan to answer a SPARQL query. Many modern frameworks used for the storage of RDF data do not scale for connected large data because the results do not show the emulation of RDF storage and retrieval.

JOTR [8] has been introduced to distributed Hadoop-based RDF systems as a SPARQL query optimization method. The design is based on the calculation of discrimination and was tested on the LUBM dataset, one of the standard RDF benchmarks. In consideration of the query execution time, large data sets were used to compare the JQPro model with other optimization approaches. The result shows that the JQPro model can deliver prominent results on distributed RDF systems. The RDF triple numbers and size of the LUBM dataset that have been tested are small, so they cannot return the real runtime.

In [32], the authors introduced the RDFox system to efficiently parallelize hash-based joins and produce a suitable parallelization result. Although RDFox supports query evalu-

ation, it is not the major emphasis of the system, and there is no support for intra-query parallelism, which means that each query is assessed in a single thread, for such queries.

Concerning key memory environments, Bilidas and Koubarakis [33] proposed a PARJ system and provided a physical architecture and a query method for using spatial localities to efficiently access the memory. The authors took advantage of totally or partly ordered RDF data through an integrated convergence retrieval approach. They proved that their prototype could store and query RDF graphs up to two billion triples on a computer with 128 GB of main memory and 16 core components. The method exceeds both centralized and distributed state-of-the-art approaches.

MuSe was proposed in [34] as a multi-level storage scheme for big RDF data using MapReduce. Large volumes of RDF data may be swiftly processed using a two-level storage approach called triple pattern matching MapReduce. The MuSe RDF Storage component of Hadoop is utilized to store a significant quantity of data, and MapReduce processes are used to handle the translated SPARQL queries. MuSe is easy to construct and deploy across a Hadoop cluster due to its simplistic architecture; nonetheless, this effort does not explain how MuSe performs with complex queries.

Ref. [35] proposed a multi-way join approach called ADJ, which co-optimizes communication, pre-computing, and computation costs in a one-round multi-way join evaluation. ADJ explores cost-effective partial results to find an optimal query plan and achieves superior performance compared to existing multi-way join methods. The study intended to address the neglect of the computational cost, which can become a bottleneck after communication costs are minimized.

The study [36] focused on the processing of distance join queries (DJQs), which involve both join and distance-based search and are used in various applications, including spatial databases and data mining. The increasing use of spatial data applications has led to the emergence of distributed spatial data management systems (DSDMSs) that use distributed cluster-based computing systems such as Hadoop and Spark. The study compared the performance of two recent DSDMSs, SpatialHadoop and LocationSpark, using existing and new parallel and distributed DJQ algorithms on large spatial datasets. The study finds that SpatialHadoop is efficient for large spatial datasets, while LocationSpark is faster for medium datasets due to in-memory processing but requires higher memory allocation for large datasets. The study also proposes efficient and scalable DJQ algorithms that consider different parameters, such as dataset sizes and the number of computing nodes.

The study of [37] proposed a meta-heuristic optimization-based approach called MOBDC-MR, which involves a binary pigeon optimization algorithm for feature selection and a beetle antenna search with an LSTM model for big data classification. They implemented the proposed method on Hadoop with the MapReduce programming model, which was evaluated using a benchmark dataset. Results show that the MOBDC-MR approach outperforms existing techniques in terms of accuracy and complexity reduction across various dimensions. This paper discusses how big data can be effectively managed and analyzed through feature selection methods to remove unnecessary features that can affect classification results. However, traditional methods are not scalable for massive datasets, so new models are necessary.

Ref. [38] addressed the issue of laggard nodes in the MapReduce architecture, where variations in the operating system environments and input data can result in differences in the number of intermediate data created, affecting the completion time of cloud application tasks. To improve execution performance, the paper proposes a dynamic task adjustment mechanism that uses an intermediate-data processing cycle prediction algorithm to adjust the number of Map and Reduce program tasks based on the processing capabilities of each cloud worker node. This mechanism aims to mitigate the impact of laggard nodes on the Google Cloud Platform (Hadoop cluster). The proposed mechanism was evaluated through a performance analysis and was found to improve processing efficiency by at least 5% for small-scale cloud applications when compared to a simulated Hadoop system.

Ref. [39] proposed a query optimization approach called access plan recommendation, which uses previously created query execution plans to optimize new queries. The query space is divided into clusters, but traditional clustering algorithms take a long time for large datasets. In addition, the study investigated the use of the Apache Spark and Apache Hadoop frameworks in the MapReduce distributed computing model to cluster query datasets of different sizes. The performance is evaluated based on execution time, and the results show that parallel query clustering is highly scalable. Additionally, Apache Spark outperforms Apache Hadoop, achieving an average speed up of 2x.

Ref. [40] proposed a new data partitioning technique called distance-join query processing (DJQs) based on Voronoi diagrams and improved MapReduce algorithms for KNNJQ and KCPQ operations. The effectiveness of these approaches is tested through experiments using real-world datasets, showing that they are efficient, scalable, and robust in SpatialHadoop. SpatialHadoop is an extension of the Hadoop framework that enables better processing of spatial datasets by incorporating global indexing techniques and partitioning data across multiple machines. DJQs, which combine spatial joins with distance-based search, are important operations in spatial applications but are also computationally expensive.

Ref. [41] discussed the importance of the Semantic Web and Big Data Technology for extracting and deriving useful knowledge from the enormous amount of structured and unstructured data on the web. It highlights the need to process this data with powerful and scalable techniques in distributed processing environments such as MapReduce and mentions several distributed RDF processing systems. The article then compares selected RDF query systems using two widely used RDF benchmark datasets, FedBench and LUBM. The results show that the SemaGrow distributed system performs more efficiently than FedX and Splendid, even though the former performs slower on smaller queries.

Ref. [42] proposed a new approach that includes a relational partitioning schema and a distributed RDF data management system to provide efficient and scalable RDF data management in distributed systems. The proposed partitioning schema, property table partitioning (PTP), further partitions an existing property table into multiple tables based on distinct properties to minimize the input size and the number of join operations in a query. The authors also introduce a distributed RDF data management system called S3QLRDF that uses SQL to execute SPARQL queries over the PTP schema and is built on top of Spark. The experimental analysis shows that S3QLRDF outperforms state-of-the-art distributed RDF management systems in terms of query performance and preprocessing costs. By addressing the issues of querying efficiency, optimization for different query patterns, and minimizing pre-processing cost, the proposed approach can lead to improved management of RDF data in distributed systems, with benefits for applications that rely on large-scale semantic data processing.

3. Proposed Join Query Processing (JQPro)

In this section, the developed JQPro propose is discussed in detail. Hence, Figure 3 presents the model in detail.

The JQPro model is composed of two main components, which are the data source and the MapReduce framework. The data source component converts large URL data to RDF format. Hence, the data pre-processor converts the RDF into N-triples serialization formats using the N-triples converter module. This N-triple file of an RDF graph is an input data source, which is loaded as a HIVE query. The users send the query via the HIVE query interface, and the query is then submitted to the MapReduce framework (the second component). As shown in Figure 3, the relationship between the plan generations for query processing and the MapReduce framework. After the data are loaded into HIVE, the input selector gets the data from the storage as triples formats. To reduce the time it takes MapReduce to find the relevant results based on the query, plan generation is used to reduce the amount of data and time taken to execute each query. Since RDF data are extracted from various machines, the performance of each machine is assumed to be

different from others. Therefore, the time to retrieve the data is different depending on the machine selected by the Hadoop framework. Plan generation consists of two developed algorithms. The first algorithm is based on the hash join, where all the selected input is inserted into a hash table using a join relation. The output of Algorithm 1 is then used in Algorithm 2 in which sort-merge join is used to find distinct values of the join attribute for each set of the data that contains the matching of the overall dataset. The combination of the list of join attributes will be used by the modified MapReduce join algorithm to map the selected data using the MapReduce framework.

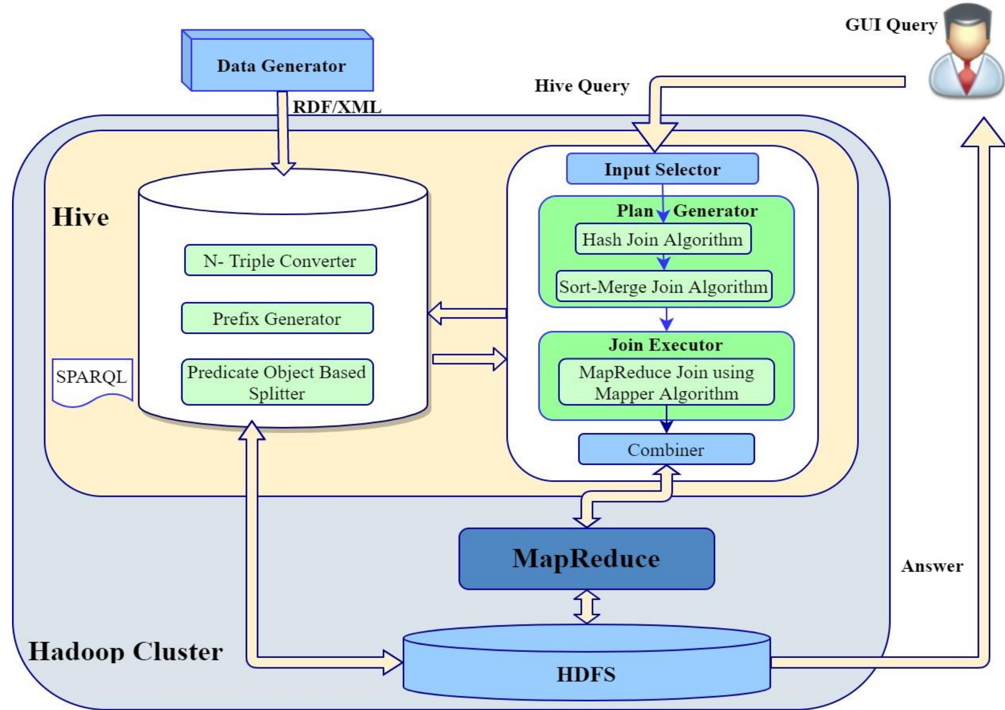

Figure 3. RDF query developed model (JQPro).

In this framework (MapReduce framework), the input selector, which is the first module, maps the HIVE query and generates a plan for query completion. The join executor uses our developed algorithms [43] that are based on the join-merge strategy to find an optimal result to reduce the time (Algorithm 1) while increasing the throughput. After the required queries are joined using the join executor mechanism, the output is shuffled and sorted by the MapReduce Framework to accurately answer the query. As such, the reduce part takes the output of the shuffle and sort using key-value to reduce the results of the query; the outcome will be again submitted to HIVE in order to be returned to the users as an answer (Algorithm 2). The third algorithm is the MapReduce join, which is performed using the developed enhanced MapReduce-join algorithm (Algorithm 3). This algorithm begins by erasing all non-joining variables that are not related to the query. Furthermore, after removing all the non-joining-related queries, the output is shuffled and sorted by the MapReduce Framework to accurately answer the query. As such, the reduce part takes the output of the shuffle and sort using key-value to reduce the results of the query by using the algorithm (Algorithm 3). The outcome will be again submitted to HIVE in order to return an answer to the users. These algorithms are further discussed in this section.

- *Hive*

Hive is an open-source data warehouse software developed by the Apache Software Foundation. It is built on top of Hadoop and enables users to perform SQL-like queries on large datasets stored in the Hadoop Distributed File System (HDFS) or other compatible file systems. Hive uses a language called Hive Query Language (HQL) to process data. HQL is

similar to SQL and allows users to write queries in a familiar SQL-like syntax. However, instead of operating on traditional database tables, Hive queries operate on distributed data stored in Hadoop. It is designed to work with structured and semi-structured data and supports various data formats such as Apache Parquet, ORC, and AVRO. It also supports data partitioning, which allows data to be divided into smaller, more manageable parts that can be processed independently.

One of the key benefits of Hive is its scalability. It can process large amounts of data stored in Hadoop and distribute the workload across multiple nodes in a cluster. Additionally, Hive can integrate with other Hadoop components, such as Pig and HBase, to provide a more comprehensive data processing solution. Overall, Hive is a powerful tool for big data processing and analysis and is widely used in industries such as finance, healthcare, and telecommunications.

- *MapReduce*

MapReduce is a programming model and software framework for processing large datasets in a distributed environment. It was developed by Google and is now an open-source project under the Apache Software Foundation. The MapReduce framework consists of two main phases: the map phase and the reduce phase. In the map phase, input data is divided into smaller chunks and processed in parallel by multiple nodes in a distributed system. Each node applies a map function to its assigned data chunk and generates a set of key-value pairs. In the reduce phase, the key-value pairs are grouped by their keys and processed by a reduce function. The reduce function aggregates the values associated with each key and generates the final output. MapReduce is designed to work with a variety of data sources, including structured, semi-structured, and unstructured data. It can also be used to perform a wide range of data processing tasks, such as filtering, sorting, aggregating, and transforming data.

One of the key benefits of MapReduce is its scalability. It can process large datasets that are too big to fit into the memory of a single node by distributing the workload across multiple nodes in a cluster. MapReduce can also handle node failures and ensure that data processing tasks are completed even if some nodes fail. Therefore, MapReduce is widely used in industries such as finance, healthcare, and telecommunications for big data processing and analysis. It has become a standard framework for processing large datasets in distributed environments and has been implemented by various big data processing systems, including Apache Hadoop and Apache Spark.

- *Related big data tools*

Computing and big data are distributed hand in hand. Big data enables users to use commodity computing so that distributed queries can be made via many data sets and the result sets returned in time. There are several related big data tools that are commonly used in conjunction with tools such as Hive and MapReduce. Some examples include:

1. Apache Hadoop: Hadoop is a distributed storage and processing framework that provides the underlying infrastructure for tools such as Hive and MapReduce. It is designed to handle large volumes of structured and unstructured data and can be used to store, process, and analyze data across a large cluster of computers.
2. Apache Spark: Spark is a fast and general-purpose cluster computing system that can be used for big data processing, machine learning, and graph processing. It provides a simpler programming model than MapReduce and can be used to process data in memory, which can result in faster performance.
3. Apache Pig: Pig is a high-level platform for creating MapReduce programs used with Hadoop. It provides a simplified programming model for data processing that is based on a language called Pig Latin.
4. Apache Cassandra: Cassandra is a distributed NoSQL database that is designed to handle large volumes of structured and unstructured data across a distributed network of nodes. It provides high availability and scalability and is often used for applications that require real-time data access.

5. Apache Storm: Storm is a distributed real-time processing system that can be used for stream processing and real-time analytics. It can be used to process large volumes of streaming data in real time, making it useful for applications such as fraud detection and sensor data processing.

- *Characteristics of the Hive query used.*

Hive query has some characteristics. The specific characteristics of a query will depend on the requirements of the project and the data being processed. However, in general, some common characteristics of Hive queries include:

1. Hive queries are written in a SQL-like language called HiveQL. This allows developers who are familiar with SQL to work with Hive without having to learn a new programming language.
2. Hive is designed to work with large datasets, so Hive queries are often used to process large volumes of data.
3. Hive is built on top of Hadoop, so Hive queries are often executed in a distributed environment across a cluster of machines.
4. Hive provides a schema-on-read approach to data processing, which means that the schema for the data is inferred at the time the data is read rather than being defined ahead of time.
5. Hive supports a wide range of data formats, including structured, semi-structured, and unstructured data.
6. Hive supports a variety of data storage systems, including the Hadoop Distributed File System (HDFS), Apache HBase, and Amazon S3.

3.1. Algorithm 1. Hash-Join

This algorithm is the hash-join algorithm [44] developed for the input selector module. For this algorithm, the first set of data is loaded along with a table inside the HIVE in the building phase. The large dataset is then scanned and joined with the relevant triple used in the probe phase. In Algorithm 1, P and Q are defined as table partitions in the LUBM and WatDiv datasets. The algorithm sorts both relationships into join qualities and then integrates the sorted relationships. Furthermore, groups of datasets with the same values are sorted in a join column. The sort-merge join algorithm sorts the data sets P and Q on the join qualities and thereafter searches for qualifying tuples p 2 P and q 2 Q by combining the 2 sets. This grouping is exploited by making a comparison between the group R tuples and the S tuples in a similar segment. The hash-join algorithm can be very useful on social media platforms where large amounts of data are generated every second. Social media platforms need to analyze this data to gain insights into user behavior, preferences, and other metrics to improve their services. The hash-join algorithm combines multiple datasets efficiently, even if they are stored in different databases. For example, they can use hash join to combine user data such as demographics, interests, and activity with engagement data such as likes, shares, and comments.

Algorithm 1: Hash-Join Algorithm

```
1. for all pϵ P, do
2.     load p into in memory hash-table H
3. end for
4. For all qϵ Q do
5.     if H contains p matching with q, then
6.          add (p,q) to the result
7.     end if
8. end for
9. end
```

3.2. Algorithm 2: Sort-Merge Join

This algorithm is the sort-merge join algorithm [45]. Firstly, the algorithm will check if the plan generation is less than the query processing. The execution of the query will then be performed. Moreover, the plan generation will be checked again based on equality. If the plan generation and query processing of the RDF datasets are similar, then these components will be added to the P and Q datasets. This process will continue until p and q are equal to the query. This algorithm requires range practitioner (records are subdivided into sorted buckets), all of which are exclusive of each other exclusive. Merge reads from two sets of reducer outputs covering a certain key range. The sorting process groups all tuples with about the same significance in the join column.

Defining partitions or groups of tuples with almost the same value in the join column is therefore easy. This partitioning is used by trying to compare P tuples in a partition just with Q tuples in about the same partition (instead of all tuples in Q), thus further ignoring the enumeration or cross-product of P and Q. This partition-based technique works just to achieve a level playing field.

Sort-merge join can be used to merge data from different sources, such as customer orders and customer information, financial data on income and expenses, and get a complete view of a company's finances, in healthcare it can help electronic health records and lab reports and providers get a complete view of each patient's medical history. Sort-merge join involves sorting the tables based on the common key and then merging the sorted tables by scanning through them in order. It can be used in situations where there are large amounts of data and performance is a concern.

Algorithm 2: Sort-Merge-Join Algorithm

```
1.  p ϵ P; q ϵ Q; gq ϵ Q
2.  while more tuples in inputs, do
3.        while p.a < gq.b, do
4.              advance p
5.        while p.a > gq.b, do
6.              advance gq {a group might begin here}
7.        while p.a == gq.b, do
8.            Q = gq {mark group beginning}
9.            while p.a == q.b, do
10.               add (p, q) to the result
11.               advance q
12.           end
13.           advance p (move forward)
14.       end
15.       gq = q {candidate to begin next group}
16. end
```

3.3. Algorithm 3: Enhanced MapReduce-Join

This algorithm begins by erasing all non-joining variables that are not related to the Q query. Q = {X, Y, VZ, XY, XZ}, where X, Y, VZ, XY, and, XZ represent a set of features with their association for query Q in the running example and erasing the variable non-joining V gives the: Q outcomes = {X,Y,Z,XY,XZ}. This is usually obtained beginning with the first task within the while loop. In Line 4, the variables are sorted out by their respective E-count, and these include: U = {Y; Z; X} with Y and Z having an E-count equal to 1, while the X variable is assigned to an E-count equal to 2. JobJ is usually designated as the storage point for all the join operations, where all the tasks executed are stored accordingly. It is important to note that 'j' denotation represents the identity of the current task. Moreover, the resultant triples of the joins of an existing join are stored in line 6 for a limited period with the variable denoted by 'tmp'. At the loop in line 8, each variable is tested to see whether it can be discarded in full or in part. If so, the results are recorded in the provisional variable

(line 9), and Q (line 11) will be modified. Hence, this will be applied to the immediate job (line 12).

To properly conduct an experiment for the JQPro model with big RDF data, we assume that, when running locally in a real application, different large data applications vary dynamically and consider the size of the cluster used in this analysis. This evaluation assumes that users can scale up or down workload tracks by their needs, whether in data or working scales.

The benchmarks' aim is to measure the performance, time of execution, CPU usage, and performance of the developed join query algorithms. Hence, MapReduce's performance is measured by its execution time [46]. Furthermore, several other factors, such as the number of mapping tasks and reduced tasks, the underlying network, the intermediate shuffled data pattern, and the shuffled data size can significantly influence the work of MapReduce.

Algorithm 3: Enhanced MapReduce-Join Algorithm

1. *Q Remove non joining variables (Q)*
2. *while Q6 = Empty, do*
3. *j 1 // Total jobs number*
4. *U = {u1,..., un} < // All variables sorted with their //E-counts in a non-decreasing order*
5. *Jobj Empty // Join operations list in the current job*
6. *Tmp Empty // The resulting triple patterns are //temporarily stored*
7. *for i = 1 to K, do*
8. *if Can-Eliminate (Q,ui) = true, then // partial or complete elimination possible*
9. *tmp tmp [Join–result(TP(Q,ui))*
10. *Q Q-TP(Q,ui)*
11. *Jobj Jobj[join(TP(Q,ui))*
12. *end*
13. *end*
14. *Q Q [tmp]*
15. *j j + 1*
16. *end*

4. Experiment

In this section, an overview of the experiment is given. The experiment setup, the benchmarks used, and the descriptions of the datasets utilized are presented in detail.

4.1. Experimental Setup

In conducting our experiment, a Hadoop cluster is configured. Hence, each node was installed in a distinct virtual machine (VM). We set up a network connectivity between five computer devices where each computer has a dedicated ethernet cable connected to the switch. Then we installed Hadoop software on each machine to join them into one cluster. Clusters can provide increased processing power and performance by distributing workloads across multiple nodes, allowing for faster processing of large datasets or complex computations.

The cluster (Hadoop) is composed of a master virtual machine and a NameNode, and multiple virtual machine workers running DataNode and a NodeManager. Therefore, DataNode is located in a cluster by the NameNode when an RDF query is placed. Hence, a DataNode contains part of a dataset and communicates with the NameNode constantly to perform a particular task. When the datasets are located within the cluster, the NodeManager starts data processing using the MapReduce framework mechanism, which begins with an input selector, maps, combines, shuffles, and reduces.

The importance of the clustering process can be seen in its ability to reduce the amount of data that needs to be processed. By grouping related data together, an algorithm

can eliminate unnecessary computations and join operations, resulting in faster query processing times and reduced computational overhead. Moreover, the cluster process enables to perform batch processing, which is particularly useful for handling large amounts of data. By clustering together similar data, an algorithm can perform batch processing on subsets of data, reducing the overall processing time required to complete the entire query.

The experiments were performed with 5 VMs installed on the Linux Ubuntu 19 Hadoop Cluster. NameNode and ResourceManager are used by one of the VMs, while DataNode and DataManager are run by the others. Each VM has a 2.80 GHz processor, 8 GB main memory, and 1000 GB of disk space configuration. A high-level query has been utilized with Hadoop version 2.6.0. To specify the data block replication limit, the maximum replication factor "dfs.replication.max" was applied. A representative benchmark set of CPU and IO-intensive applications included in the Hadoop distribution, such as Waterloo SPARQL Diversity Test Suite (Watdiv) and Lehigh University Benchmark (LUBM) has been used for performance analysis to evaluate the MapReduce jobs effectively [17].

4.2. Benchmarks

In this research, two synthetic benchmarks were used, namely LUBM and WatDiv v06 benchmarks.

LUBM: The Lehigh University Benchmark was developed as a standard to systematically facilitate the evaluation of Semantic Web repositories. The purpose of the benchmark is to evaluate the performance of those repositories with extensional queries over a large dataset committed to a single realistic ontology. The benchmark consists of the ontology of the university domain, customizable and repeatable synthetic data, a set of test queries, and various performance metrics. The LUBM features a university-domain ontology, arbitrary-scalable synthetic OWL data, 14 extensional queries representing a variety of properties, and many performance metrics. LUBM has been extended to include a widely adopted RDF and OWL dataset benchmark of approximately 1 GB data size.

WatDiv: The benchmark was developed to measure how an RDF data management system performs on a large variety of SPARQL queries with varying characteristics and selectivity classes. Data generator WatDiv enables users to create their datasets through a description language of the dataset. Thus, users can control which entities are included in their dataset. How well-structured each entity is, how different entities are associated with each other, the probability that a type X entity is associated with a type Y entity, and the cardinality of such associations. The test data for WatDiv is designed using those features. It is possible to generate test datasets of various sizes by executing the data generator with different scale factors.

Our implementation ranged from ten million to one billion RDF triples generated using the WatDiv data generator with scale factors of 10, 100, and 1000 on three datasets, respectively. WatDiv, a more balanced stress testing environment for RDF data management systems with more diverse underlying workloads than other benchmarks, provides the generator with all the different forms of queries. This allowed us to test the performance of the developed algorithm against more fine-grained rivals. Hence, the reason for choosing these two benchmarks in this study is that both of them are the most often used to process RDF datasets as baseline benchmarks for MapReduce.

5. Results and Discussion

In this section, the result is sectionalized into the various sections that represent distinct outcomes of the developed model on the two benchmarks (WatDiv and LUBM).

5.1. Effectiveness of JQPro on WatDiv Benchmark

This section presents the results of JQPro on the WatDiv benchmark. Hence, the results of different queries of size 10 M, 100 M, and 1000 M are given. Hence, these results are illustrated in Figure 4.

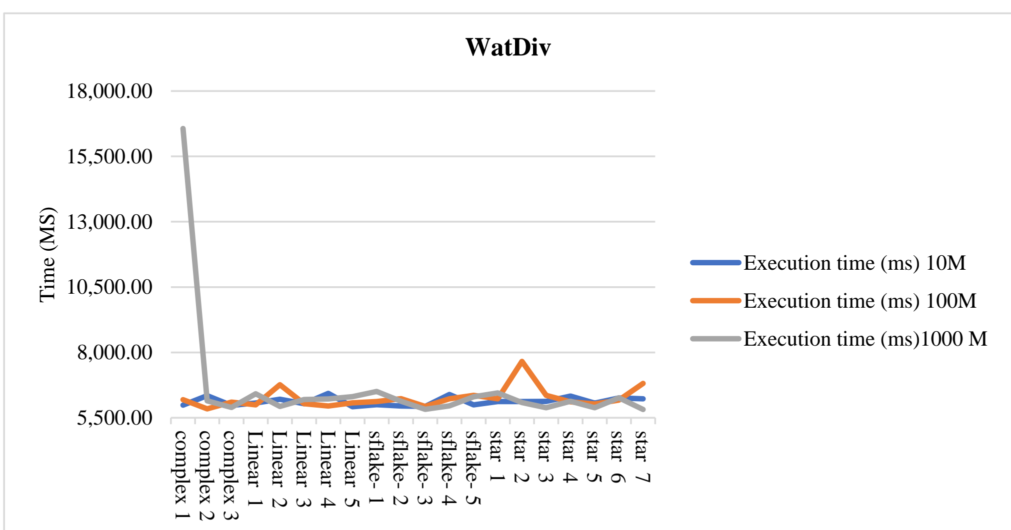

Figure 4. Measurement of the execution time (s), for a number of queries using WatDiv.

Figure 4a highlights the overall average execution time for queries of size 10 M. These queries are complex (1–3), linear (1–5), snowflake-shaped (1–5), and star (1–7) as presented in the figure. From Figure 4a, we observed that the execution time of each query is different; hence, some queries have a higher execution time while others have a lower one. For instance, we observed that, for the star 7 query, the execution time is 6.22 s which is the highest achieved by our developed model (JQPro); this is followed by snowflake 3 with 6.93 s, while the lowest execution time is for query linear 5 with a 5.92 s execution time. The reader should note that the execution times are in seconds. Hence, we conclude that with the WatDiv query of size 10 M, JQPro has performed relatively well. Furthermore, in Figure 4, the same queries were utilized with a query name of size 100 M WatDiv. We observed that star 2 and star 7 are the queries that have the highest execution time. Therefore, with a 5.03 s execution time, the query snowflake-shaped 3 achieved the lowest execution time on size of 100 M. However, star 7 is the second highest with a 6.81 s execution time. The trend is different from the size 10 M, where the query linear 5 is the lowest (as illustrated in Figure 4).

We further present the result based on size 1000 M query with respect to their execution times in Figure 4. Based on the obtained results, we observed that the execution time was significantly reduced for some queries, such as star 7, snowflake-shaped 3. However, the query complex 1 execution time increased rapidly. This can be explained by the CPU-intensive job performed. Hence, with a higher CPU power, the execution time can be further reduced for complex 1 query. This observation is critical as the performance of the developed model (JQPro) is not consistent across various query sizes in the WatDiv benchmark. However, the JQPro model performed very well in this experiment.

The overall results illustrate that with a relatively large number of triples, the execution time of some categories is reduced slightly. This result shows that using JQPro to perform an RDF query can assist in reducing the time it takes to complete a task in comparison to the existing normal query, which only gives fixed times for each result.

Figure 5 shows the results of overall execution time with size data 1000 M triples. Various RDF triple query strategies are based on complex 1, 2, 3, snowflake-shaped 1–5, linear 1–5, and star 1–7. The result indicates that under various queries, the model shows stability in terms of execution time. Moreover, Figure 5 and Table 2 show the queries with size data of 100 M triples, which are complex 1, 2, 3, snowflake-shaped 1–5, linear 1–5, and star 1–7. Each query was performed multiple times to obtain different results based on execution time. This implies that the model demonstrates consistency in terms of execution time. Particularly in the star 2 execution time, the natural test takes longer to complete when the data size decreases, while snowflake-shaped 3 needed more triples and

less execution time. Figure 5 and Table 1 with size data 10 illustrate that the execution time of different queries, which are reduced on the basis of the execution time of the application, is reduced significantly for some queries. However, a modest triples throughput exhibits the maximum execution time in snowflake-shaped 3 and linear 3, as shown in Figure 5. As for the less execution time, it was at complex 1 and 3, snowflake-shaped 2, and linear 5, and the volume of data was also moderated. In contrast, the highest quantity of triple throughput is shown in complex 1, while the execution time was relatively less, and with increasing execution time, reduced triple throughput was in star 7.

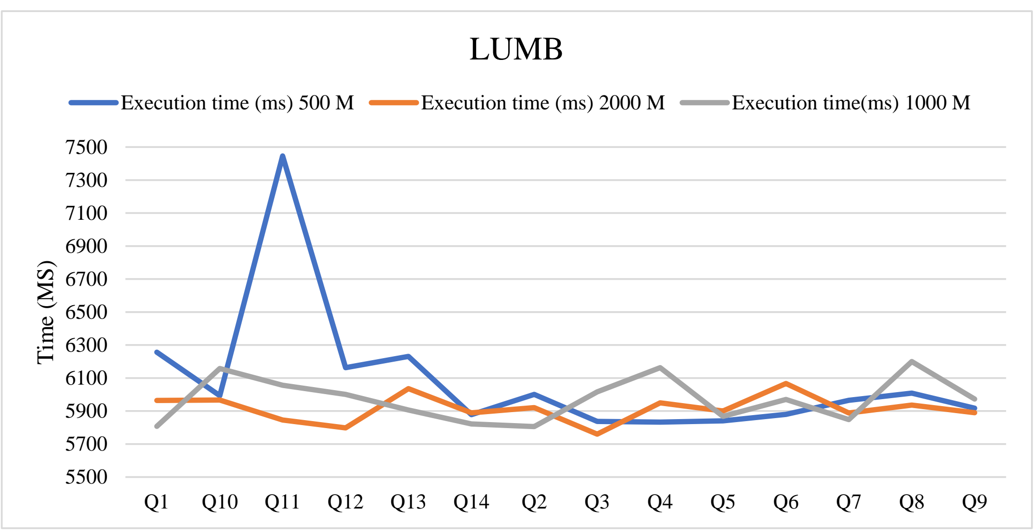

Figure 5. Measurement of the execution time (s), for a number of queries using LUMB.

In addition, the comparison between complex queries with different sizes of WatDiv triples increased in terms of execution time as the number of triples increased. However, the average execution time is higher than the regular queries. The WatDiv Triple execution time of various sizes for snowflake queries is slightly higher. Significantly, the total time efficiency of execution is higher than the existing performance.

5.2. Effectiveness of JQPro on LUBM Benchmark

In this section, the results of JQPro on the LUBM benchmark are presented. The RDF data for this experiment are processed using different sizes of arranged triples (500 M, 1000 M, and 2000 M). Furthermore, numerous query categories, which involve fourteen queries, are chosen. The results show that these queries perform better than normal RDF queries found in the literature.

Figure 5 presents the results of the JQPro experiment on the LUBM benchmark. As indicated in Figure 5, 14 queries are performed to measure execution time, and it shows that the execution time of some queries improved. Hence, Figure 5 highlights the overall average execution time for queries of size 500 M. From Figure 5 we observed that query Q11 has the highest execution time. Hence, the best-performing query on a 500 M query size is the query Q4, followed by Q3, Q5, Q14, and Q6, respectively. These queries achieved low execution times. From Figure 5, the overall average execution time for queries of size 1000 M is presented. Hence, the experiment was also conducted with 14 queries. Each query is performed to obtain different results based on execution time. The results show that some of the queries (e.g., Q2, Q1, Q14, Q7, and Q5) provide low execution times. However, queries such as Q8, Q4, and Q10 have a high execution time.

The result of the execution time for queries of size 2000 M is also presented in Figure 5. Based on the experiment conducted, we observed that some queries, such as Q3 and Q12, provided the lowest execution time. However, Q6 and Q13 have the highest execution times, as presented in Figure 4. The result obtained in this section with respect to the

experiment on the LUBM benchmark is good, with no more than 6.6 s of execution time observed in all queries. These observations are important and are key to solidifying the result obtained in this section on the LUBM benchmark.

5.3. Cross-Comparison of JQPro with the Existing State-of-the-Art

In this section, the cross-comparison results between the JQPro model and four models (gStore, RDF-3X, RDFox, and PARJ) are presented with respect to the WatDiv and LUBM benchmarks. The result is given for two facets, which are the execution time and the throughputs. The results of execution time and throughput are discussed, which demonstrate the total efficiency of our developed model. The analysis of our findings is presented through several figures indicating their values.

5.3.1. Comparison on WatDiv Benchmark

The evaluation is required to confirm the reliability of the results based on our developed model. Evaluation of the JQPro model results from the experiments and comparison with two existing models, gStore and RDF-3X, is presented. From Table 1, the reader can see the cross-comparison of JQPro with the selected models.

Table 1. Comparison between our model and presented system for execution time–WatDiv.

Query (MS)	JQPro	gStore	RDF-3X
Linear	6144	20,128	16,282.1
Star	6073	10,808.7	3820.6
Snowflake-shaped	6216	29,204.8	8405.6
Complex	9535	15,447	11,980.3

The queries were also highlighted in the table. From the result, based on the WatDiv benchmark, we observed that generally, JQPro has a lower execution time with a great margin. Hence, based on this result, the JQPro model performs better in comparison to the existing state-of-the-art.

Figure 6 shows the execution time (mean) of a number of queries using WatDiv Benchmark with JQPro, gStore, and RDF-3X in Hadoop cluster nodes with linear, star, snowflake-shaped, and complex queries of a large number of triples. This shows that if the number of tasks increases, the completion time of the overall processing is also reduced.

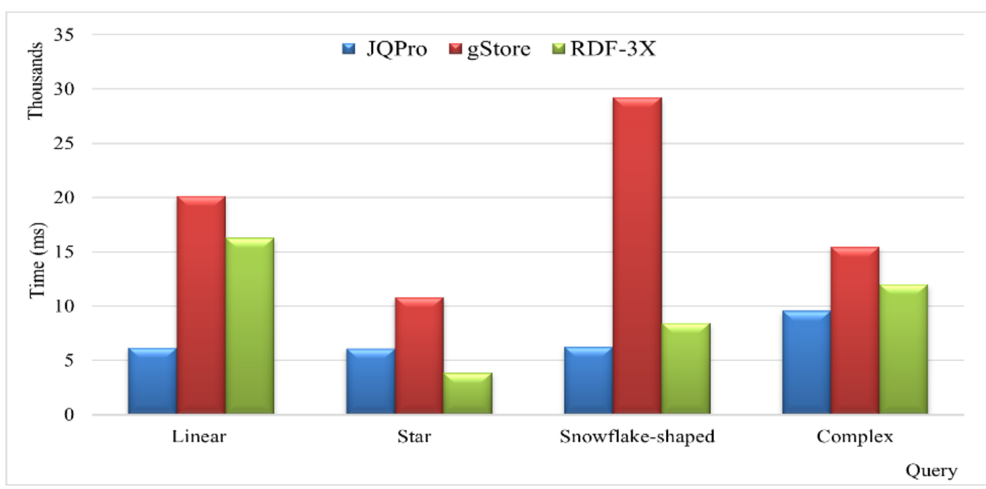

Figure 6. Execution time linear queries–WatDiv.

In the first scenario, gStore and RDF-3X are used on Hadoop nodes without tuning Hadoop parameters or the JQPro model. The result shows that JQPro outperforms the two

popular algorithms, gStore and RDF-3X. The blue color represents the result of the JQPro model based on the joined approach.

5.3.2. Comparison on LUBM Benchmark

Figure 7 and Table 2 compare three models, namely RDF-3X, RDFox, and PARJ, with the JQPro model. The result shows that JQPro has outperformed the mentioned models when queries Q1 to Q3 and Q7 to Q10 are used on the LUBM benchmark. Looking at the figure and table, one can see that JQPro outperformed all the compared models on some queries for the LUBM benchmark. This is the same for the WatDiv benchmark as well. Hence, we conclude that the JQPro model is very effective in handling multiple join queries.

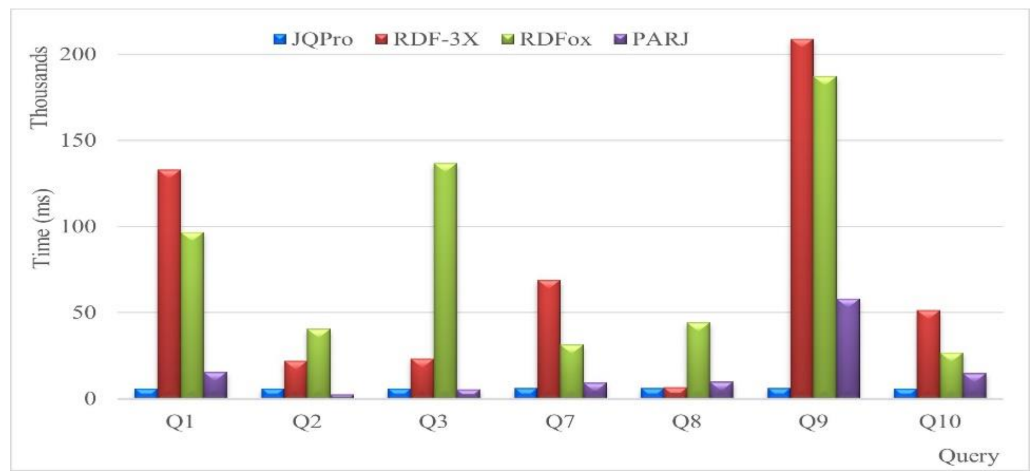

Figure 7. Execution time–LUBM benchmark.

Table 2. Comparison between our solution and presented system for execution time–LUBM.

Query (MS)	JQPro	RDF-3X	RDFox	PARJ
Q1	6033	132,951	96,677	15,369
Q2	5975	21,870	40,368	2437
Q3	5957	23,179	136,554	5338
Q7	6151	68,769	31,180	9213
Q8	6264	6485	44,144	9899
Q9	6081	208,839	187,192	58,082
Q10	6054	51,235	26,690	14,606

5.4. Overall Results

Performance evaluation of the execution time for all four models of various data sizes using the benchmarks (WatDiv and LUBM) was presented. The gStore was 18,897.125 ms on average for WatDiv queries. The RDF-3X was 12,222.66 ms on average for WatDiv; it was also 73,332.57 ms in LUBM, which is less efficient in LUBM than WatDiv.

The LUBM queries (Q1, Q2, Q3, Q7, Q8, Q9, and Q10) have a mean run time of 80,400.71 and are higher than the other three strategies. PARJ has taken an average of 21,433,80 for LUBM queries (Q1, Q2, Q3, Q7, Q8, Q9, and Q10), and its performance is lower than the other three strategy frameworks. JQPro uses a lower runtime of 6073 ms for the Star query for the WatDiv benchmark and 5957 ms for the Q3 query of the LUMB benchmark in comparison with the existing models. The results in this section demonstrate a percentage improvement in JQPro execution time for the developed model. The improvement percentage is calculated according to the time execution by each model.

Performance evaluation of the execution time frameworks for all four techniques with various data sizes comes from the benchmarks WatDiv and LUBM. The gStore took 18,897.125 ms on average for WatDiv queries. The RDF-3X was taken in 12,222.66 ms by

queries in WatDiv; it was also 73,332.57 ms in LUBM, which is therefore less efficient in LUBM than WatDiv. The LUBM queries (Q1, Q2, Q3, Q7, Q8, Q9, and Q10) have a mean run time of 80,400.71 and are higher than the other three strategies. PARJ has taken an average of 21,433,80 for LUBM queries (Q1, Q2, Q3, Q7, Q8, Q9, and Q10), and its performance is lower than the other three strategy frameworks. Evidently, JQPro uses a lower runtime, at 6073 ms for the Star query WatDiv and 5957 ms for the Q3 query LUMB, compared with other existing systems. The improvement percentage is calculated according to the time spent executing each framework using Equation (1) the percentage change is measured. It is an easy and well-known equation for identifying improvements for a technique in the percentage of performance testing.

$$PIM = \frac{TE_i - TS}{TE_i} \times 100 \quad (1)$$

where,

PIMi improves the JQPro system by percentage versus new technology;

TEi is the average time the *i*th existing framework has been implemented;

TS is the JQPro framework's average execution time.

In both WatDiv and LUBM datasets, JQPro generally showed faster execution times compared to the other systems (gStore, RDF-3X, and RDFox for LUBM only). Specifically, for WatDiv, JQPro outperformed gStore and RDF-3X for all query types, with an average improvement. For LUBM, JQPro also outperformed gStore and RDF-3X, with an average improvement. These results suggest that JQPro may be a promising alternative to others in terms of query performance for large-scale RDF datasets.

6. Conclusions and Future Direction

For big RDF data, join query processing is an important aspect. With the increase in RDF data size, the query execution time also increases drastically. Hence, in this situation, an effective join query processing model/framework is imminent and crucial to help in the execution time reduction. Furthermore, various performance issues such as hardware failure, software errors, machinery, and data heterogeneity have posed a big challenge to RDF data analytics.

The performance tuning of RDF query algorithms is therefore essential to ensure the performance of algorithms and frameworks in a different environment. In this paper, we developed a novel join query processing model for big RDF data, called JQPro.

The JQPro model adopted a MapReduce framework to aid in RDF data reduction and sorting. JQPro is composed of three newly developed algorithms, which are hash-join, sort-merge, and enhanced MapReduce-join for join query processing of RDF data (as discussed in Section 3). The key objective of the JQPro model is to process the join query of RDF data by utilizing the HIVE and MapReduce strategies for SPARQL queries. Our evaluation using the Lehigh University Benchmark (LUBM) and the Waterloo SPARQL Diversity Test Suite (WatDiv) v06 benchmarks shows that the JQPro model outperforms the existing state-of-the-art. These algorithms determine the starting time of the task by running the task. The evaluation result further showed that the execution time of multiple tasks in a parallel system is reduced. Lastly, the findings showed that JQPro achieved an improved performance of 87.77% in terms of execution time. The performance result is based on the overall average of multiple experiments that have been conducted using the benchmarks. Hence, in comparison with the selected models, JQPro performs better. Deploying and testing our proposed solution in practical use-case applications such as healthcare, social media, and open data can provide valuable insights into how the solution works in a real-world setting. These industries generate massive amounts of data that can be challenging to analyze. Advanced data processing and analysis techniques such as those proposed can help provide meaningful insights and improve decision-making processes.

There are several potential areas for improving the performance and capabilities of JQPro, a tool for processing join queries on large RDF datasets. One approach is to

incorporate more advanced distributed computing frameworks such as Apache Spark or Apache Flink to handle large-scale data processing tasks more efficiently. Another approach is to explore new techniques for data compression and storage to reduce the amount of data that needs to be processed. Additionally, integrating JQPro with other big data tools and platforms such as Hadoop or NoSQL databases could provide new opportunities for optimization and performance improvements. A further area for research is investigating the use of machine learning techniques to optimize join query processing in distributed systems, such as using reinforcement learning algorithms to select join algorithms and keys and partition data for efficient processing.

The JQPro model has limitations that need to be addressed, such as a limited range of query operations, reliance on the MapReduce framework, and being primarily tested on the LUBM benchmark. More advanced query capabilities may be required in certain scenarios, and other distributed computing frameworks such as Apache Spark or Flink may provide better performance and scalability. Further evaluation and testing on different datasets and query types are needed to fully assess JQPro's capabilities and limitations.

Author Contributions: Conceptualization, N.M.E. and I.A.T.H.; Methodology, N.M.E., M.A.M. and I.A.T.H.; Software, N.M.E.; Investigation, A.O.I. and F.B.; Writing—original draft, N.M.E.; Writing—review & editing, A.O.I., A.W.A. and F.B.; Visualization, A.O.I. and A.W.A.; Supervision, M.A.M. and I.A.T.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Acknowledgments: Authors acknowledge thanks to the Faculty of Computing, University Malaysia Pahang and Faculty of Computing and Informatics, University Malaysia Sabah.

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

References

1. Hernández-Illera, A.; Martínez-Prieto, M.A.; Fernández, J.D. RDF-TR: Exploiting structural redundancies to boost RDF compression. *Inf. Sci.* **2020**, *508*, 234–259. [CrossRef]
2. Ruta, M.; Scioscia, F.; Bilenchi, I.; Gramegna, F.; Loseto, G.; Ieva, S.; Pinto, A. A multiplatform reasoning engine for the Semantic Web of Everything. *J. Web Semant.* **2022**, *73*, 100709. [CrossRef]
3. Freitas, A.; Curry, E.; Oliveira, J.G.; O'Riain, S. Querying heterogeneous datasets on the linked data web: Challenges, approaches, and trends. *IEEE Internet Comput.* **2011**, *16*, 24–33. [CrossRef]
4. Mountantonakis, M.; Tzitzikas, Y. Content-based Union and Complement Metrics for Dataset Search over RDF Knowledge Graphs. *J. Data Inf. Qual. (JDIQ)* **2020**, *12*, 1–31. [CrossRef]
5. Consortium, W.C.W.W.W. SPARQL Query Language for RDF. 2008. Available online: http://www.w3.org/TR/rdf-sparql-query (accessed on 23 June 2022).
6. Yao, Z.; Chen, R.; Zang, B.; Chen, H. Wukong + G: Fast and Concurrent RDF Query Processing Using RDMA-Assisted GPU Graph Exploration. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *33*, 1619–1635. [CrossRef]
7. Dong, X.; Yu, B.; Sun, H. Design and Implementation of SPARQL Engine Based on Heuristic Algorithm. In Proceedings of the 2022 11th International Conference of Information and Communication Technology (ICTech), Wuhan, China, 4–6 February 2022; IEEE: Piscataway, NJ, USA, 2022.
8. Chawla, T.; Singh, G.; Pilli, E. JOTR: Join-Optimistic Triple Reordering Approach for SPARQL Query Optimization on Big RDF Data. In Proceedings of the 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Bengaluru, India, 10–12 July 2018; IEEE: Piscataway, NJ, USA, 2018.
9. Husain, M.; McGlothlin, J.; Masud, M.M.; Khan, L.; Thuraisingham, B.M. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.* **2011**, *23*, 1312–1327. [CrossRef]
10. Peng, P.; Ge, Q.; Zou, L.; Ozsu, M.T.; Xu, Z.; Zhao, D. Optimizing Multi-Query Evaluation in Federated RDF Systems. *IEEE Trans. Knowl. Data Eng.* **2019**. [CrossRef]
11. Abadi, D.J.; Marcus, A.; Madden, S.R.; Hollenbach, K. SW-Store: A vertically partitioned DBMS for Semantic Web data management. *VLDB J.* **2009**, *18*, 385–406. [CrossRef]
12. De Virgilio, R.; Del Nostro, P.; Gianforme, G.; Paolozzi, S. A scalable and extensible framework for query answering over RDF. *World Wide Web* **2011**, *14*, 599–622. [CrossRef]
13. Karvinen, P.; Díaz-Rodríguez, N.; Grönroos, S.; Lilius, J. RDF stores for enhanced living environments: An overview. In *Enhanced Living Environments*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 19–52.

14. Ranichandra, C.; Tripathy, B. Architecture for distributed query processing using the RDF data in cloud environment. *Evol. Intell.* **2019**, *14*, 567–575. [CrossRef]
15. Chantrapornchai, C.; Choksuchat, C. TripleID-Q: RDF query processing framework using GPU. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 2121–2135. [CrossRef]
16. Jarrar, M.; Dikaiakos, M.D. A query formulation language for the data web. *IEEE Trans. Knowl. Data Eng.* **2011**, *24*, 783–798. [CrossRef]
17. Hogenboom, A.; Frasincar, F.; Kaymak, U. Ant colony optimization for RDF chain queries for decision support. *Expert Syst. Appl.* **2013**, *40*, 1555–1563. [CrossRef]
18. Tatu, M.; Werner, S.; Balakrishna, M.; Erekhinskaya, T.; Moldovan, D. Semantic question answering on big data. In Proceedings of the International Workshop on Semantic Big Data, San Francisco, CA, USA, 1 July 2016; ACM: New York, NY, USA, 2016.
19. Karnstedt, M.; Sattler, K.-U.; Hauswirth, M. Scalable distributed indexing and query processing over Linked Data. *Web Semant. Sci. Serv. Agents World Wide Web* **2012**, *10*, 3–32. [CrossRef]
20. Choi, P.; Jung, J.; Lee, K.-H. RDFChain: Chain Centric Storage for Scalable Join Processing of RDF Graphs using MapReduce and HBase. In Proceedings of the International Semantic Web Conference (Posters & Demos), Sydney, Australia, 23 October 2013; Citeseer: Princeton, NJ, USA, 2013.
21. Galárraga, L.; Hose, K.; Schenkel, R. Partout: A distributed engine for efficient RDF processing. In Proceedings of the 23rd International Conference on World Wide Web, Seoul, Korea, 8 April 2014; ACM: New York, NY, USA, 2014.
22. Abdelaziz, I.; Harbi, R.; Khayyat, Z.; Kalnis, P. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. In Proceedings of the VLDB Endowment, Munich, Germany, 28 August–1 September 2017; Volume 10, pp. 2049–2060.
23. Özsu, M.T. A survey of RDF data management systems. *Front. Comput. Sci.* **2016**, *10*, 418–432. [CrossRef]
24. Goasdoué, F.; Kaoudi, Z.; Manolescu, I.; Quiané-Ruiz, J.A.; Zampetakis, S. Cliquesquare: Flat plans for massively parallel RDF queries. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Republic of Korea, 13–17 April 2015; IEEE: Piscataway, NJ, USA, 2015.
25. Schätzle, A.; Przyjaciel-Zablocki, M.; Skilevic, S.; Lausen, G. S2RDF: RDF querying with SPARQL on spark. In Proceedings of the VLDB Endowment, New Delhi, India, 12 August 2016; Volume 9, pp. 804–815.
26. Xu, Q.; Wang, X.; Li, J.; Zhang, Q.; Chai, L. Distributed subgraph matching on big knowledge graphs using pregel. *IEEE Access* **2019**, *7*, 116453–116464. [CrossRef]
27. Guo, X.; Gao, H.; Zou, Z. Leon: A Distributed RDF Engine for Multi-query Processing. In *International Conference on Database Systems for Advanced Applications*; Springer: Berlin/Heidelberg, Germany, 2019.
28. Gai, L.; Wang, X.; Wang, T. ROSIE: Runtime Optimization of SPARQL Queries over RDF Using Incremental Evaluation. In *International Conference on Knowledge Science, Engineering and Management*; Springer: Berlin/Heidelberg, Germany, 2018.
29. Neumann, T.; Weikum, G. RDF-3X: A RISC-style engine for RDF. *Proc. VLDB Endow.* **2008**, *1*, 647–659. [CrossRef]
30. Zou, L.; Özsu, M.T.; Chen, L.; Shen, X.; Huang, R.; Zhao, D. gStore: A graph-based SPARQL query engine. *VLDB J.* **2014**, *23*, 565–590. [CrossRef]
31. Husain, M.; Doshi, P.; Khan, L.; McGlothlin, J. *Efficient Query Processing for Large rdf Graphs Using Hadoop and Mapreduce, in Technical Report*; University of Texas Dallas: Dallas, TX, USA, 2009.
32. Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; Banerjee, J. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2015.
33. Bilidas, D.; Koubarakis, M. Scalable Parallelization of RDF Joins on Multicore Architectures. In Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), Lisbon, Portugal, 26–29 March 2019.
34. Chawla, T.; Singh, G.; Pilli, E. MuSe: A multi-level storage scheme for big RDF data using MapReduce. *J. Big Data* **2021**, *8*, 130. [CrossRef]
35. Zhang, H.; Qiao, M.; Yu, J.X.; Cheng, H. April. Fast distributed complex join processing. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 2087–2092.
36. García-García, F.; Corral, A.; Iribarne, L.; Vassilakopoulos, M.; Manolopoulos, Y. Efficient distance join query processing in distributed spatial data management systems. *Inf. Sci.* **2020**, *512*, 985–1008. [CrossRef]
37. Abukhodair, F.; Alsaggaf, W.; Jamal, A.T.; Abdel-Khalek, S.; Mansour, R.F. An intelligent metaheuristic binary pigeon optimization-based feature selection and big data classification in a MapReduce environment. *Mathematics* **2021**, *9*, 2627. [CrossRef]
38. Huang, T.C.; Huang, G.H.; Tsai, M.F. Improving the Performance of MapReduce for Small-Scale Cloud Processes Using a Dynamic Task Adjustment Mechanism. *Mathematics* **2022**, *10*, 1736. [CrossRef]
39. Azhir, E.; Hosseinzadeh, M.; Khan, F.; Mosavi, A. Performance Evaluation of Query Plan Recommendation with Apache Hadoop and Apache Spark. *Mathematics* **2022**, *10*, 3517. [CrossRef]
40. García-García, F.; Corral, A.; Iribarne, L.; Vassilakopoulos, M. Improving distance-join query processing with voronoi-diagram based partitioning in spatialhadoop. *Future Gener. Comput. Syst.* **2020**, *111*, 723–740. [CrossRef]
41. Mohammed, H.H.; Doğdu, E.; Choupani, R.; Zarbega, T.S. Distributed Query Processing and Reasoning Over Linked Big Data. In Proceedings of the Recent Advances in Transdisciplinary Data Science: First Southwest Data Science Conference, SDSC 2022, Waco, TX, USA, 25–26 March 2022; Revised Selected Papers. Springer Nature Switzerland: Cham, Switzerland, 2023; pp. 158–170.

42. Hassan, M.; Bansal, S. S3QLRDF: Distributed SPARQL query processing using Apache Spark—A comparative performance study. *Distrib. Parallel Databases* **2023**, 1–41. [CrossRef]
43. Elzein, N.M.; Majid, M.A.; Fakherldin, M.; Hashem, I.A.T. Distributed Join Query Processing for Big RDF Data. *Adv. Sci. Lett.* **2018**, *24*, 7758–7761. [CrossRef]
44. Pavlo, A.; Paulson, E.; Rasin, A.; Abadi, D.J.; DeWitt, D.J.; Madden, S.; Stonebraker, M. A comparison of approaches to large-scale data analysis. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, Providence, RI, USA, 29 June–2 July 2009; pp. 165–178.
45. Shin, D.K.; Meltzer, A.C. A new join algorithm. *ACM SIGMOD Rec.* **1994**, *23*, 13–20. [CrossRef]
46. Pham, C.M.; Dogaru, V.; Wagle, R.; Venkatramani, C.; Kalbarczyk, Z.; Iyer, R. An evaluation of zookeeper for high availability in system S. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, Dublin, Ireland, 22–26 March 2014; ACM: New York, NY, USA, 2014.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.