applied sciences

MDPI

Article

Optimization of the Join between Large Tables in the Spark Distributed Framework

Xiang Wu and Yueshun He *

School of Information Engineering, East China University of Technology, Nanchang 330013, China; x81367828@163.com
* Correspondence: heys@ecut.edu.cn

Abstract: The Join task between Spark large tables takes a long time to run and produces a lot of disk I/O, network I/O and disk occupation in the Shuffle process. This paper proposes a lightweight distributed data filtering model that combines broadcast variables and accumulators using RoaringBitmap. When the data in the two tables are not exactly matched, the dimension table Key is collected through the accumulator, compressed by RoaringBitmap and distributed to each node using broadcast variables. The distributed fact table data can be pre-filtered on the local server, which effectively reduces the data transmission and disk reading and writing in the Shuffle phase. Experimental results show that this optimization method can reduce disk usage, shorten the running time and reduce network I/O and disk I/O for Spark Join tasks in the case of massive data, and the effect is more obvious when the two tables have a higher incomplete matching degree or a fixed matching degree but a larger amount of data. This optimization scheme has the advantages of being easy to use, being easy to maintain and having an obvious effect, and it can be applied to many development scenarios.

Keywords: Join; Spark; Shuffle; optimization method; RoaringBitmap

check for updates

Citation: Wu, X.; He, Y. Optimization of the Join between Large Tables in the Spark Distributed Framework. *Appl. Sci.* **2023**, *13*, 6257. https://doi.org/10.3390/app13106257

Academic Editor: Antonio Fernandez Caballero

Received: 24 March 2023
Revised: 6 May 2023
Accepted: 13 May 2023
Published: 19 May 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

With the rapid development of the Internet in recent years, the era of big data has arrived. After years of development, a large number of new high-performance technologies have emerged in the field of big data, such as Apache Spark [1–3] and Apache Flink [4], which are stronger than MapReduce [5,6] in terms of query and computation performance and which have become powerful tools for big data acquisition, storage, analysis and presentation. Big data analysis technology plays a key role in various industries. Asad et al. [7,8] studied the importance of big data analysis technology in enterprises.

Spark is a fast, universal, scalable and highly available big data analysis search engine developed based on Scala. It has upgraded its performance based on the MapReduce model. Developers can deploy Spark on a large number of servers to form clusters that efficiently process data. The core technology of Spark is the use of resilient distributed datasets (RDD) [9]. The data are distributed in the form of RDD on each server for management, to achieve data parallelization and distributed processing. During the data repartitioning process of the Spark task, if data are moved across nodes, Shuffle is generated, as shown in Figure 1. Shuffle is a bridge between Map and Reduce. It corresponds the Map output to the Reduce input and involves serialization and deserialization, cross-node network I/O and disk read/write I/O. If a complex service logic has Shuffle, the next stage can be executed only after the previous stage produces a result. In the mass data Join task of distributed architecture, the data interaction between servers will inevitably generate Shuffle, which means a large number of serialization–deserializations, cross-node network I/Os and disk read and write I/Os.

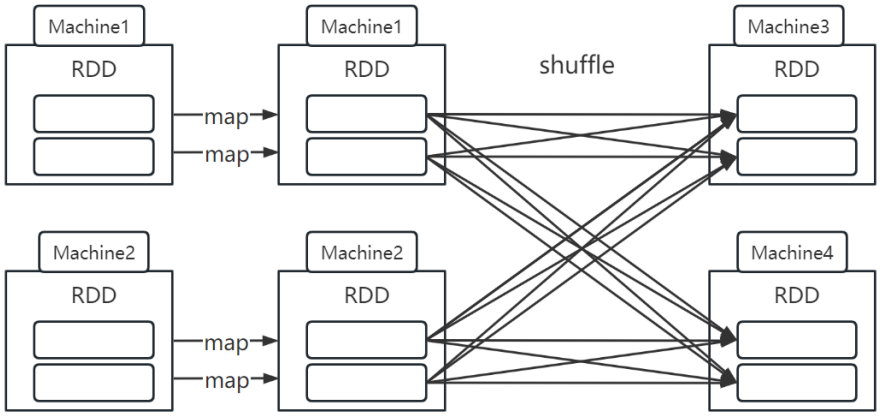

Figure 1. The Spark RDD dependency displayed in a Shuffle structure diagram.

Shuffle consists of Shuffle write and Shuffle read phases, as shown in Figure 2. During Shuffle, a large amount of intermediate data is migrated to disks for a long time, and a large amount of network I/O is generated, affecting the overall performance of the Spark job.

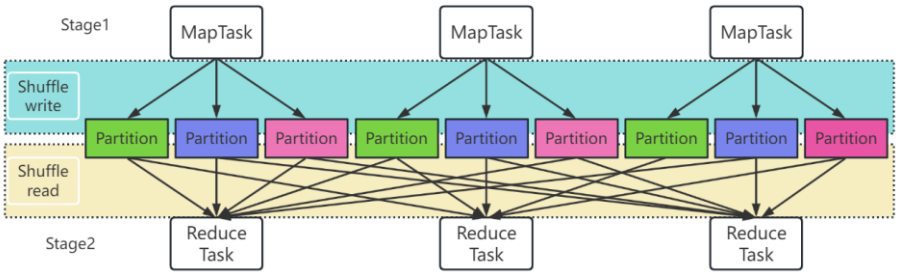

Figure 2. Interaction between Shuffle write and Shuffle read.

When optimizing the Spark performance, we should not only pay attention to the execution time of tasks but should also pay attention to the network IO and disk read and write. The proper optimization is not only to reduce the uptime but also to reduce the network I/O load, disk usage, and so on. In actual development, the number of Spark tasks ranges from as few as 100 to as many as thousands. In this case, performance optimization of key tasks is extremely important. Proper performance optimization can ensure running efficiency and save resources, and helps to avoid negative effects caused by excessive operation data.

In order to solve the problems of a long running time, excessive network IO load and high disk occupancy of Spark Join tasks between large tables, this paper proposes a lightweight distributed data filtering model using RoaringBitmap [10] to combine broadcast variables and accumulators when the data in two tables are not completely matched. This optimization is theoretically analyzed and experimentally verified. The implementation results show that this method effectively reduces the running time of Spark Join and effectively reduces the data transfer and disk read and write in the Shuffle phase. The overall performance of Spark Join tasks is improved.

2. Related Work

The performance of most Spark jobs is mainly consumed by the Shuffle process. This process involves a large number of disk I/O operations, serialization and deserialization operations and network data transmission operations. Therefore, to improve the performance of Spark jobs, it is necessary to optimize the Shuffle process.

In terms of load balancing, Ren et al. [11] studied the cross-network reading of Shuffle and the aggregation of partition data among tasks with data dependence. They adopted heuristic prescheduling through SCache, combined with Shuffle size prediction, and bal-

anced the load of each node through load balancing to achieve Shuffle optimization. Li et al. [12] studied the data skew in the Shuffle stage and proposed a Shuffle phase dynamic balance partitioning method based on reservoir sampling to sample and preprocess the intermediate data, predict the overall data skew and provide the overall partitioning strategy for application implementation, thus reducing the impact of data skew on the Spark performance. Kumar et al. [13] studied the search space partitioning strategy of data parallelism. Based on the communication cost-effectiveness pattern mining algorithm, tasks can be allocated fairly and effectively among cluster nodes to reduce the communication cost generated during Shuffle. Choi et al. [14] used SSD to make up for the lack of main memory bandwidth and applied RDD cache strategies with different proportions of Shuffle and storage space to improve the overall performance of the system. Tang et al. [15] proposed an initial adaptive task concurrency estimation algorithm combined with known task input information and actuator memory, realized dynamic memory-aware task scheduling and used two typical benchmarks, light Shuffle-light and heavy Shuffle-heavy, to evaluate the performance, which significantly improved the resource utilization. Zeidan et al. [16] proposed a new spatial divider for the spatial query of large spatial data sets. KNN spatial join query, based on Spark, is used to reduce the spatial query skew and task running time. Zhao et al. [17] studied the cache management strategy in the Dag-aware task scheduling algorithm, and proposed a new cache management strategy called long-run phase set priority to make full use of task dependency to optimize cache management performance in the Dag-aware scheduling algorithm. Tang et al. [18] studied partitioning methods in the Spark framework, considering the partition balance of the intermediate data and the partition balance after the Shuffle operator. The range-based key segmentation algorithm realized slant mitigation in Shuffle and effectively reduced task execution time. Based on the new operators and some new logical and physical rules, they extended the Spark query to achieve task optimization.

In the rational use of resources, Jiang et al. [19] proposed a data management algorithm based on the data mixing stage to effectively reduce the resource occupation and computing response delay based on Spark, which is prone to problems such as insufficient utilization of Spark cluster resources, high computation delay and high task processing delay in the Shuffle stage. The partition-weighted adaptive cache replacement algorithm based on RDD can make full use of memory resources and reduce resource waste effectively. Bazai et al. [20] proposed a data processing method based on distributed data set RDD-based data anonymization technology, based on subtree, which provides effective partition RDD-based method management, improves memory usage, uses cache to frequently reference intermediate values and enhances iteration support. Modi et al. [21] studied the execution of big data queries to realize the sorting and hash aggregation of intermediate data in memory, the exchange of intermediate data to disks and the network transmission of data. Chen et al. [22] proposed a new method of temporal data processing for large events, based on the problem that the computing capacity of distributed systems is limited when processing large-time data and cannot meet the requirements of low delay and high throughput, which effectively realizes large-time data management, operation and real-time response. Shen et al. [23] studied the scalability of Shuffle and designed a new Shuffle mechanism through Magnet, which effectively reduced the data local Shuffle operation and further improved the efficiency and reliability of Shuffle in Spark.

Nowadays, many optimization schemes lack out-of-the-box methods; that is, when the performance of a big data cluster reaches a bottleneck, it needs to be simple, convenient, practical and convenient for later maintenance to break through the performance bottleneck. Although many optimization methods increase some of the performance, they add a lot of unstable factors to the big data cluster. They may not be able to achieve a stable equilibrium state in the actual development process, which may require additional maintenance of the algorithm model and complicate the development. Many practical problems can be solved by using appropriate algorithm models. Qalati et al. [24] used a partial least squares structural equation model to analyze data and obtained the influencing factors of energy

saving intention and actual behavior. The optimization scheme used in this paper can achieve the effect out of the box in Spark Join tasks between large tables, and the effect is obvious, the stability is strong and the maintenance is easy, and it can be applied to many development scenarios.

3. Related Technologies

3.1. RoaringBitmap Algorithm

RoaringBitmap is composed of binary data structure, using bit as the unit to store data, so the data compression rate is very high. To store 4 billion data of type int, the data size is 14.9 GB for normal storage and 512 MB for RoaringBitmap storage. RoaringBitmap storage is about 30 times smaller than normal storage.

RoaringBitmap uses a bucket mechanism to save space. The int data are divided into 2^{16} buckets. The first 16 bits of binary data are used as bucket numbers. Each bucket has a Container for storing the last 16 bits of binary data. A RoaringBitmap is a collection of Containers. When storing data, the first 16 bits of data are numbered to find the corresponding Container. If the corresponding Container is not found, the corresponding Container is created and the last 16 bits of data are put into the Container. As shown in Figure 3, the value of 20 is saved into RoaringBitmap, and the value of the first 16 bits is 0 through calculation. Therefore, the corresponding Container number is 0. After obtaining the corresponding Container, the calculated value of the last 16 bits of 20 is set into the corresponding Container.

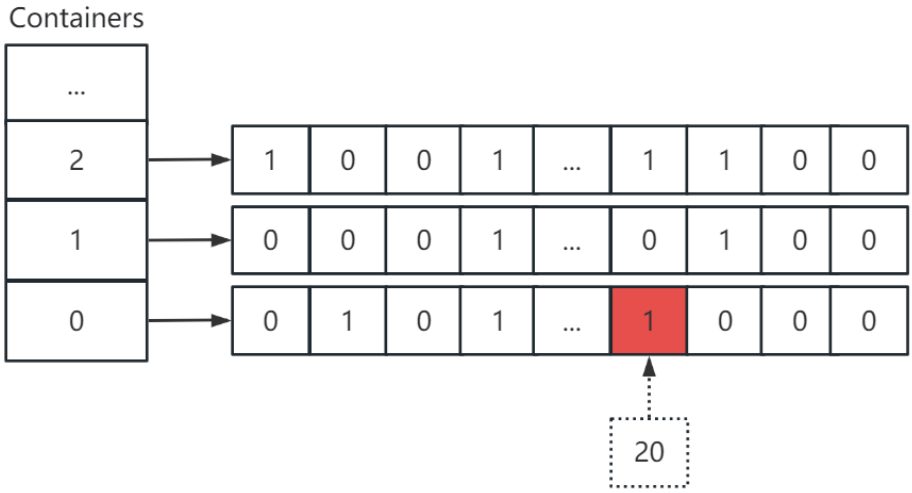

Figure 3. RoaringBitmap storage mode.

3.2. Spark Accumulator

The Spark accumulator summarizes data about variables on the Executor side of a cluster to the Driver side. As shown in Figure 4, the accumulator of the Driver side is first serialized and sent to the Executor. Then, the accumulator is used in the Executor to collect data. Finally, the accumulator of each Executor is obtained at the Driver end and the accumulator is merged by the Merge function to obtain the final result.

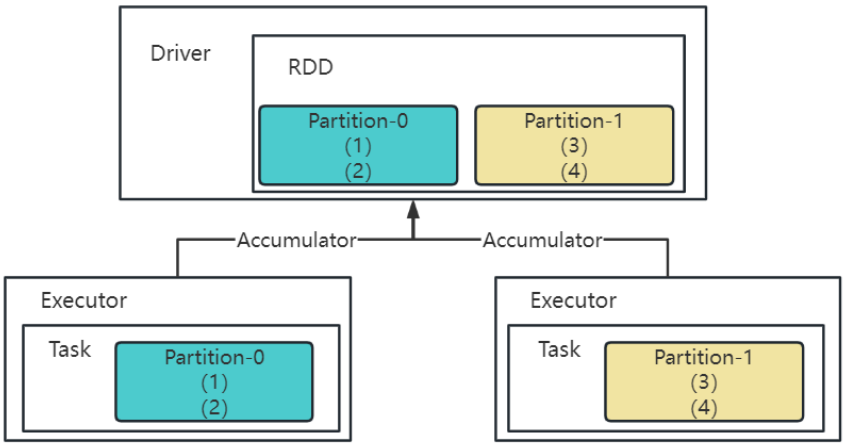

Figure 4. Spark components related to the accumulator.

3.3. Spark Broadcast Variable

The broadcast variable means that one variable is sent to memory on each Executor node associated with the cluster task, as shown in Figure 5. Data information is broadcast to each Executor node, serialized as it is fetched, and deserialized as it is used. Spark tasks can directly read data information from the Executor memory of the local node to prevent data interaction between different tasks from generating a large cross-node network I/O.

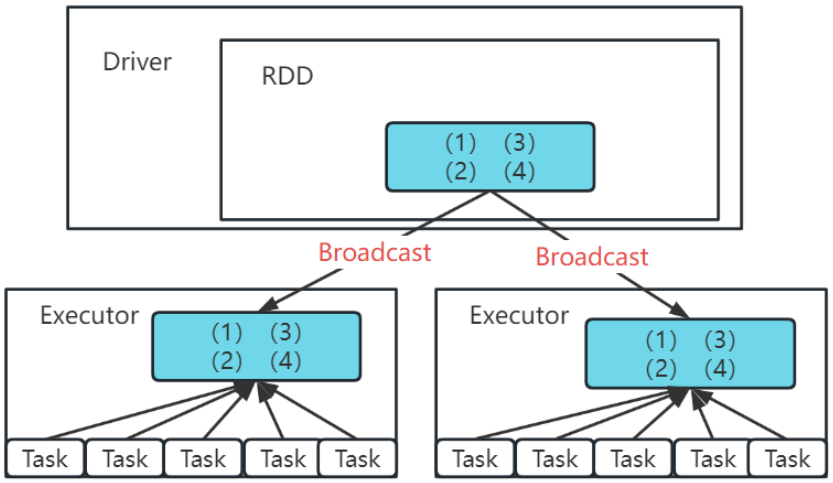

Figure 5. Spark data exchange of broadcast variables.

4. Optimization of Project Analysis

4.1. Cost Optimization Estimation

The execution efficiency of Spark Join tasks is affected by the CPU, memory, disk, runtime configuration and execution code. In the process of evaluating the cost of executing tasks, it is difficult for us to calculate the exact cost of tasks. In the case of a fixed configuration, we only need to estimate the cost of Spark Join tasks before and after optimization to obtain a comparative result, so as to reflect the rationality of the optimization scheme.

The physical execution plan of the Spark Join task based on cost-based optimization (CBO) is a tree structure, the cost of which is equal to the sum of costs of each execution node, as shown in Figure 6:

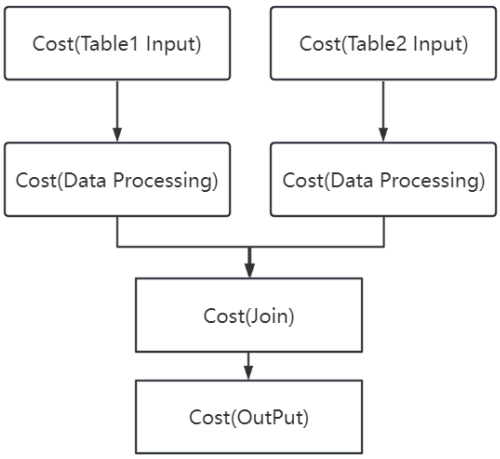

Figure 6. Spark execution node cost.

The cost is equal to the sum of the costs of each execution node, where the highest cost is the Join procedure. In the cost estimation formula of CBO, as shown in Formula (1):

$$Cost = Rows \times Weight + Size \times (1 - Weight) \quad (1)$$

Rows is the number of rows, *Size* is the size of the data, and *Weight* is the weight, which is determined by the spark.sql.cbo.joinReorder.card.weight configuration. In Spark Join, when the data in the two tables are not exactly matched, the weight is fixed, and the fact table is pre-filtered using the optimization scheme before joining, then the rows and size are reduced. According to the CBO estimation formula, the cost before optimization is greater than the cost after optimization. The more data you filter, the lower the cost will be. Lim et al. [25] studied all possible query execution paths in grouping subquery computation overhead and selected effective query execution paths through efficient query algorithms to reduce the cost. Path analysis technology is also applied in all walks of life. Hammami et al. [26] used path analysis technology to test the hypothesis of the dimension of organizational knowledge ability, reveal the various knowledge abilities of the enterprise, and establish the relationship between them. The experimental optimization purpose of this paper is to reduce the Rows and Size before the Shuffle of Join so as to reduce the cost in the maximum cost link. Since the filtering model used in our experiment is very lightweight, it has little impact on the overall cost. After a large amount of irrelevant data have been filtered out in the pre-filtering phase of the experiment, Spark Join tasks may degrade from complex types to simple ones, as shown in Figure 7, greatly reducing the overall cost.

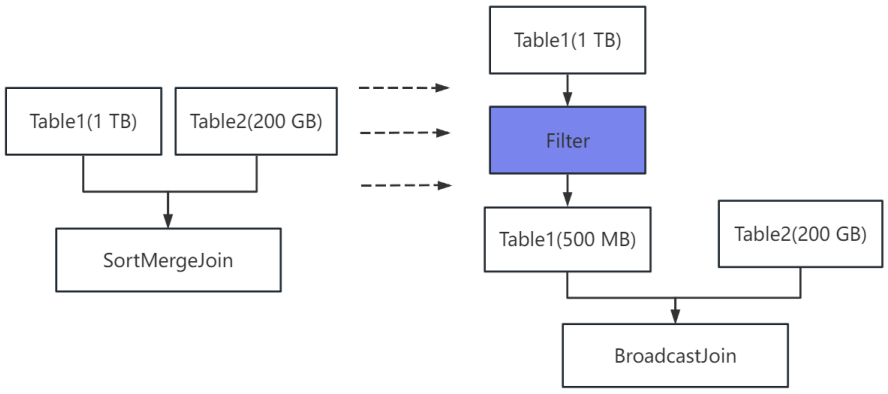

Figure 7. Spark Join degradation.

This paper studied the cost of Shuffle write and Shuffle read in Shuffle of Spark Join. The cost estimate of Shuffle write workflow is shown in Formula (2):

$$Cost_{shuffle\ write} = Cost_{cache} + Cost_{sort} + \sum \left(Cost_{buffer} + Cost_{spill} \right) + Cost_{merge} \quad (2)$$

$Cost_{cache}$ represents the cost of reading data into the cache, $Cost_{sort}$ represents the cost of sorting according to the marked partition, $Cost_{buffer} + Cost_{spill}$ represents the cost of each save to the cache and spill to disk and $Cost_{merge}$ represents the cost of small file consolidation on disk.

It should be noted that the operation processes of Shuffle write and Shuffle read are similar, but Shuffle read needs to establish a network connection and data transfer. When the running memory is sufficient, there will be no spill operation, so no disk file will be generated. When the memory is insufficient, it will also generate sort and spill operations to generate disk files, so the cost calculation of Shuffle read is different in the case of sufficient memory and insufficient memory.

Shuffle read workflow estimates the cost of sufficient memory, as shown in the following equation:

$$Cost_{shuffle\ read} = Cost_{net} + Cost_{cache} \quad (3)$$

Shuffle read Workflow memory shortage cost estimation is shown as follows:

$$Cost_{shuffle\ read} = Cost_{net} + Cost_{cache} + \sum \left(Cost_{buffer} + Cost_{sort} + Cost_{spill} \right) + Cost_{merge} \quad (4)$$

$Cost_{net}$ represents the cost of obtaining data transmitted over the network; $Cost_{cache}$ represents the cost of reading data into the cache; $Cost_{buffer} + Cost_{sort} + Cost_{spill}$ represents the cost of obtaining cache data each time for sorting and then spilling to disk; $Cost_{merge}$ represents the cost of small file consolidation on disk.

The Shuffle write workflow first fetches the data and caches it in memory, then sorts the data, and finally writes the data to disk to generate small files and merges the small files. The size of the data acquired by Shuffle write affects the final size of the data written to disk. The larger the amount of data acquired by Shuffle write, the more data will be written to the disk. Cost estimation involves each step of Shuffle write, but the data cache in the first stage is the key to the cost size. If only a small amount of data are cached, the subsequent cost consumption will be small; if the amount of cached data is large, the subsequent cost will also be large.

Shuffle read mainly involves data network transmission and data caching. The Shuffle read cost is also strongly determined by the size of the data read, but the data read is derived from the data written to disk by the Shuffle write. When the memory is not sufficient, it is also necessary to write data to the disk for temporary storage, increasing the cost.

In this optimization scheme, the amount of data read by Shuffle write is reduced by pre-filtering, so that the overall cost of Shuffle write is reduced, and the amount of data written to disk by Shuffle write is also reduced. When Shuffle write writes fewer data to the disk, Shuffle read needs to read fewer data for network transfer and data caching. At the same time, it also reduces the cache of Shuffle write and Shuffle read data in memory, reduces the utilization of memory and avoids the shortage of memory in the Shuffle read workflow to a greater extent, resulting in a high cost.

4.2. Optimization of Work Content

In the Shuffle process of Spark Join, each node of the cluster writes data to the local disk file through Shuffle write, and Shuffle read obtains the disk file of each node through the network transmission. There are a lot of data interactions, network transfers and file read and write operations, which is why the Shuffle phase is very time and resource consuming.

The optimization scheme in this paper is to preprocess the two tables of Join based on the fact table and dimension table data not completely matching, clean the fact table data

before Shuffle, deal with unnecessary data and only let the data that need to be joined enter the Shuffle phase, which saves resources and reduces the running time to a greater extent.

How to clean the data of each node has become the key to the experiment. Firstly, it should complete the data cleaning task under the condition of limited resources. Secondly, it should have a good cleaning effect in many data sets and should keep the task stable during operation and easy to maintain. According to the requirements of the optimization scheme, the lightweight and highly compressible storage component RoaringBitmap was selected. The accumulator and broadcast variable are used to ensure that RoaringBitmap has high stability, maintainability and efficiency in the process of data loading and data transmission. Therefore, the accumulator, broadcast variable and RoaringBitmap were selected for the Spark pre-filtering task in the experiment.

The execution flow of Join for the optimization scheme in this paper is shown in Figure 8. We first create an accumulator and load the RoaringBitmap into it, and then collect the dimension table data keys into the accumulator of the type RoaringBitmap. The RoaringBitmap is broadcast to each node as a Spark broadcast variable and stored in memory. In the filtering phase, each cluster node reads the Key stored in the RoaringBitmap and matches the Key of the current fact table. If the Key of the fact table does not match the value, the data will be deleted. Through the above method, a fact table without redundant data is obtained, and then we Join the data. There are no redundant data in the fact table to enter the Shuffle phase, so as to avoid unnecessary data interaction, network transmission and disk reading and writing generated by Shuffle write and Shuffle read. Thus, efficient and energy-efficient Spark Join tasks can be achieved.

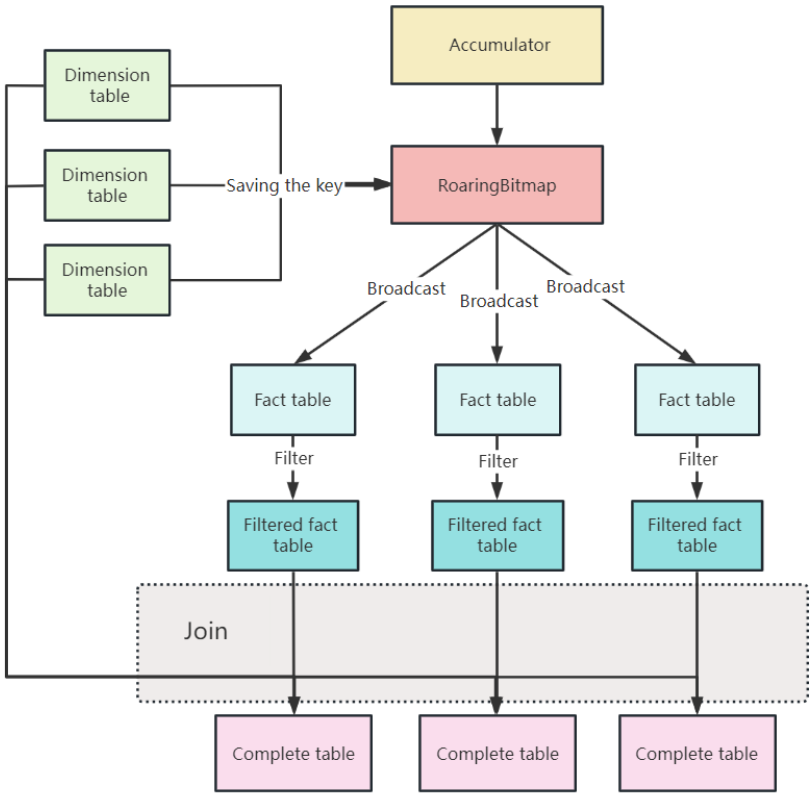

Figure 8. Experiment to optimize specific execution steps.

The storage engine in the experiment running task infrastructure in this paper is based on Spark on hive, and the query engine is based on Spark on yarn. We first store the dimension table and fact table data required for the experiment in the Hive [27] database, and then submit the Spark Join job request from the Spark client, which will submit the job

to Yarn [28]. Finally, Yarn reads the dimension and fact tables to be joined from Hive and performs a distributed Spark Join. Figure 9 shows the Spark task execution architecture.

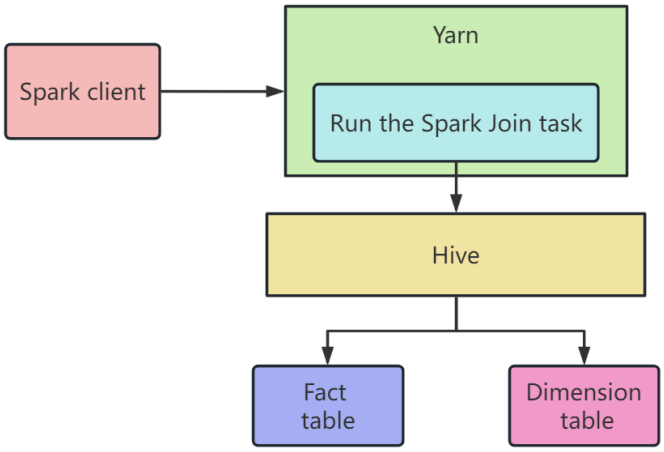

Figure 9. Spark task execution architecture.

In this paper, the comparison is mainly based on three aspects: the task running time before and after optimization, the data size written to disk by Shuffle write and the data size read by Shuffle read. The larger the Shuffle write and Shuffle read phases, the higher the disk footprint, the higher the network IO and the higher the disk IO. If the task running time is shortened after optimization, and the data size of Shuffle write to disk and Shuffle read to disk is reduced, the optimization scheme is very feasible in Spark Join tasks.

5. Experiment

5.1. System Configuration

This optimization experiment is based on Cloudera's Distribution Including Apache Hadoop (CDH) big data platform. The Spark, Hive, Hadoop [29], Zookeeper [30], and Hue components were installed on the CDH big data platform. Spark was used to execute parallel Join tasks, Hive was used to build a data warehouse on Hadoop's HDFS [31] storage engine, Hadoop's Yarn was used to manage resources and schedule tasks on Spark, Zookeeper was used to coordinate components and manage metadata, and Hue was used to build visual queries on Hive to check whether Spark Join data were lost or incorrect. In order to achieve the effect of distributed computing, this experiment involved the setting up of a big data cluster on three Linux servers.

The cluster configuration is shown in Table 1. Altogether, there was a 24-core CPU, 192 GB of memory and 600 GB of hard disk.

Table 1. Cluster configuration.

Server Name	CPU	Memory	Hard Disk
Hadoop201	8-core	64 GB	200 GB
Hadoop202	8-core	64 GB	200 GB
Hadoop203	8-core	64 GB	200 GB

The versions of development tools used by the cluster are shown in Table 2.

Table 2. Development tool versions.

Tool	Versions
Operating System	Centos7.5
CDH	6.3.2
JDK	1.8.0_181
Hadoop	3.0.0 + cdh6.3.2
Hive	2.1.1 + cdh6.3.2
Zookeeper	3.4.5 + cdh6.3.2
Spark	2.4.0 + cdh6.3.2
Hue	4.2.0 + cdh6.3.2

5.2. Testing Dataset

In this paper, we used the TPC-H [32] data set, which is a test set of the TPC-H business intelligence computing test used to simulate decision support applications. At present, this data set is widely used in academia and industry to evaluate performance related to the application of decision support technology.

The first round of experimental data we used was the official data set of TPC-H. The number of data used in the fact table lineitem was 120 million, and the numbers of data used in the dimension table orders were 100,000, 1 million, 5 million, 10 million and 30 million, respectively. The numbers of data after Join were 400,000, 4 million, 20 million, 40 million and 120 million. The orders table has a one-to-many data association with the lineitem table.

In the second round of experimental data, we also used the official data set of TPC-H. In order to realize the complex Join scenario of the many-to-many data association mode, we tested the optimization scheme through different amounts of data when the matching degree was determined. We processed the data of the TPC-H dataset, obtained the orders table with a 1-million-data volume, and copied the data in the table seven times to become the orders table with a 7-million-data volume. The lineitem table with a 10-million-data volume was obtained, and the lineitem table with a 10-million-data volume was replicated 5, 10, 50, 100 and 150 times, respectively, to obtain lineitem tables with 50 million, 100 million, 500 million, 1 billion and 1.5 billion-data volumes. The numbers of data in the orders table Join lineitem table are 140 million, 280 million, 1.4 billion, 2.8 billion and 4.2 billion, respectively. The orders table is used as the dimension table and the lineitem table is used as the fact table in the experiment. The orders table is many-to-many with the lineitem table. The orders table matches 40% of the data in each lineitem table.

5.3. Experimental Results and Analysis

In the experiment, the configuration resources applied for when submitting tasks to Spark were executor-cores 2, num-executors 3, and executor-memory 1 g.

5.3.1. First Round of Experiments

The data volume of the fact table lineitem used in the experiment was 120 million, and the data volume of the dimension table orders was 100,000, 1 million, 5 million, 10 million and 30 million; the data volume of the Join result was 400,000, 4 million, 20 million, 40 million and 120 million, respectively. The orders table has a one-to-many data association with the lineitem table. The Spark distributed computing query framework is used to read Hive data and run it on Yarn for Join operation. The data of each group were tested five times and the average value was obtained. In the data sets of 100,000, 1 million, 5 million and 10 million, the pre-filtering was carried out under the condition of incomplete matching, and the Join time was shortened correspondingly, with the proportion of shortening time being 30.0%, 29.6%, 23.7% and 19.7%, respectively, and the average shortening time was 68.75 s. The average shortening time ratio was 25.75%. The experimental results are shown in Figure 10.

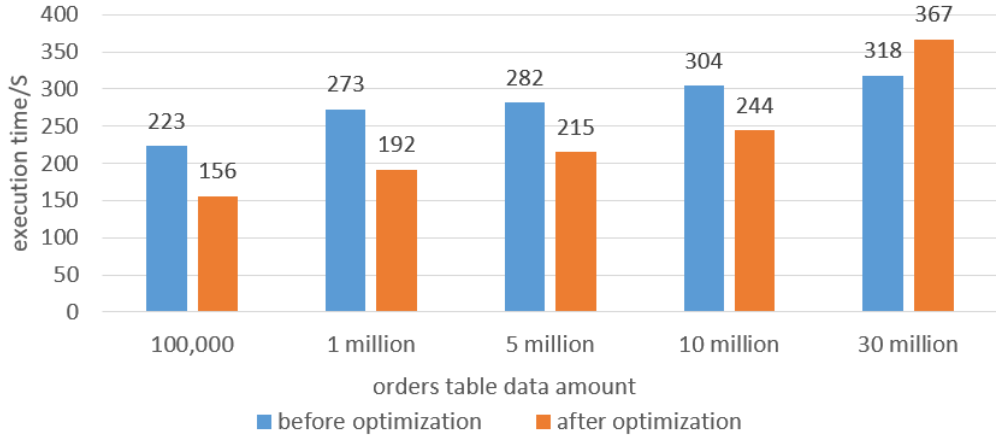

Figure 10. Task run time for one-to-many data.

5.3.2. Second Round of Experiments

The data volume of the orders table used in the experiment was 7 million, and the data volume of the lineitem table was 50 million, 100 million, 500 million, 1 billion and 1.5 billion, respectively. The orders table has a many-to-many data association with the lineitem table. The Spark distributed computing query framework is still used to read Hive data and run it on Yarn for the Join operation. Five experiments were conducted to obtain the average value of each group's data. After pre-filtering, the execution time of the Join task decreased more with an increasing amount of data. The rate of time reduction was 15.1%, 17.0%, 19.7%, 22.0% and 25.2%, respectively. The experimental results are shown in Figure 11.

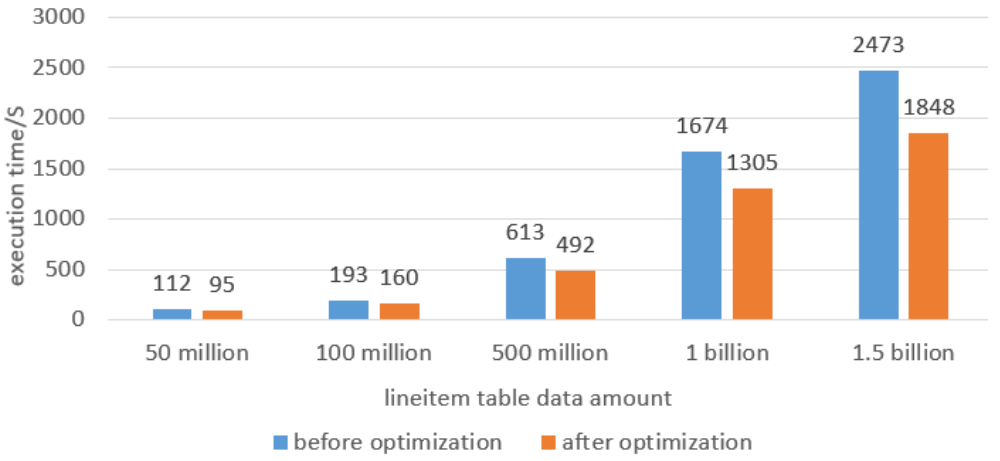

Figure 11. Task run time for many -to-many data.

In addition to the task running speed, we also recorded the Shuffle write data in the Join process. The data volume before optimization was 133 MB, 232 MB, 1024 MB, 2013 MB and 3096 MB, and the data volume after optimization was 74 MB, 114 MB, 434 MB, 834 MB, and 1234 MB, respectively. In the Join process, as the data volume of the optimized Shuffle write task increased, the data volume written to the disk decreased by 44.3%, 50.8%, 57.6%, 58.5% and 60.1%, respectively. The experimental results are shown in Figure 12.

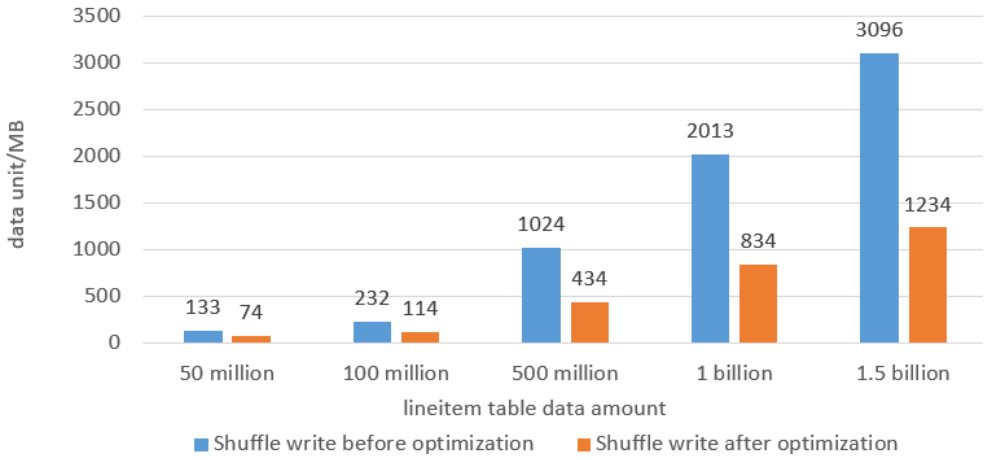

Figure 12. Amount of Shuffle write data in a many-to-many data Join.

During Shuffle write, there is also a corresponding Shuffle read. Shuffle read data were recorded in the experiment. Before optimization, Shuffle read 133 MB, 232 MB, 1024 MB, 2013 MB and 3096 MB from the disk. After optimization, Shuffle read 74 MB, 114 MB, 434 MB, 834 MB and 1234 MB, respectively. In the Join process, as the data volume of the optimized task increased, the data volume read by Shuffle read decreased by more. The data read from the disk decreased by 44.3%, 50.8%, 57.6%, 58.5% and 60.1%, respectively. The experimental results are shown in Figure 13.

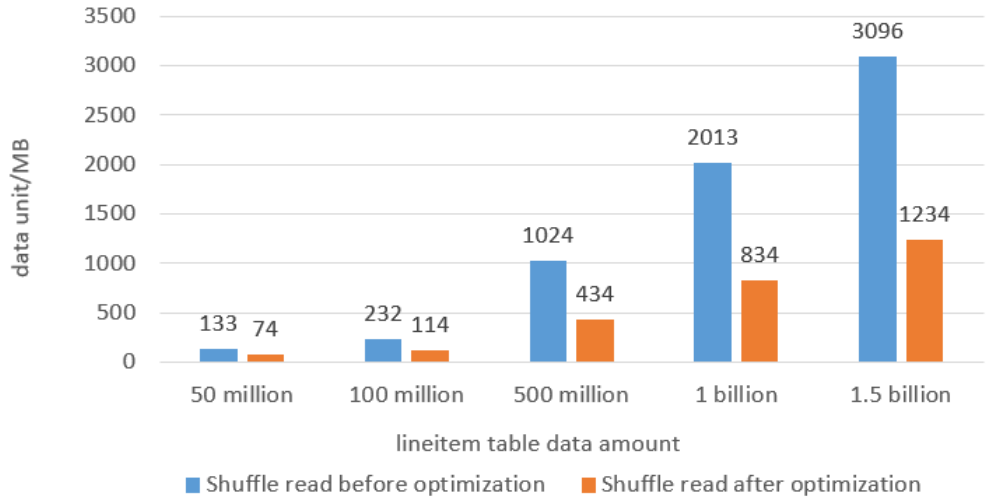

Figure 13. Amount of Shuffle read data in a many-to-many data Join.

5.3.3. Summary and Analysis of Experiments

An experimental comparison between the Spark Join task before optimization and the optimized Spark Join task was carried out. In the first round of experiments, it can be seen from the experimental results that when the data matching degree of the two tables continues to decrease, the running time of more tasks can be shortened by our optimized scheme, and the proportion of shortened time increases. In the second round of experiments, when the matching degree of the two tables is fixed, when the amount of data in the lineitem table increases and the amount of data after Join increases, the optimized tasks can shorten running time more. After optimization, as the amount of data increases, the amount of data written to disk in Shuffle write phase is reduced more and the reduction proportion increases. The amount of data read from disk in Shuffle read phase is reduced more and the reduction proportion increases.

Experiments show that the optimization scheme reduces the running time and reduces the resource consumption of Spark tasks when the data of the two big tables are not exactly matched; it also reduces the amount of operation data in Shuffle write and Shuffle Read phases to reduce network I/O, disk I/O and disk consumption.

6. Conclusions

The Join process between Spark large tables consumes a lot of resources. This paper proposes a data filtering model using RoaringBitmap as the main Spark accumulator and broadcast variables as the auxiliary. Using this filtering model eliminates the irrelevant data in the process of distributed interaction with a very small storage cost, avoiding unnecessary data processing in the Shuffle phase leading to resource consumption. Compared with other optimization schemes, this optimization scheme pays more attention to the simplicity, maintainability and versatility of the optimization method, considers the running time, disk I/O, disk occupation and network I/O, and pays more attention to the overall performance of the Join task. Therefore, a lightweight, maintainable, and extensible combination of RoaringBitmap, accumulator and broadcast variable is adopted. In the experiments on this optimization scheme, the Spark Join task completes in less time, with less disk consumption, lower disk I/O and lower network I/O. The optimization scheme can be applied in many development scenarios; when the two tables have a higher degree of incomplete matching or a fixed degree of matching but a larger amount of data, the effect is more obvious.

Author Contributions: Conceptualization, X.W.; methodology, X.W.; validation, X.W.; investigation, X.W.; data curation, X.W.; writing, X.W.; supervision, Y.H.; project administration, Y.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Key Research Projects of Jiangxi Province, grant No. 20224BBC41001 and Jiangxi Key Laboratory of Cybersecurity Intelligent Perception, grant No. JKLCIP202203, JKLCIP202204.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original data used in this study can be downloaded from the Transactionprocessing Performance Council website (http://www.tpc.org), accessed on 5 March 2023.

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

References

1. Salloum, S.; Dautov, R.; Chen, X.; Peng, P.X.; Huang, J.Z. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* **2016**, *1*, 145–164. [CrossRef]
2. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10), Boston, MA, USA, 22–25 June 2010*; HotCloud: Berkeley, CA, USA, 2010.
3. 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]
4. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bull. Tech. Comm. Data Eng.* **2015**, *38*, 28–38.
5. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
6. Dean, J.; Ghemawat, S. MapReduce: A flexible data processing tool. *Commun. ACM* **2010**, *53*, 72–77. [CrossRef]
7. Asad, M.; Asif, M.U.; Khan, A.A.; Allam, Z.; Satar, M.S. Synergetic effect of entrepreneurial orientation and big data analytics for competitive advantage and SMEs performance. In Proceedings of the 2022 International Conference on Decision Aid Sciences and Applications (DASA), Chiangrai, Thailand, 23–25 March 2022.
8. Asad, M.; Asif, M.U.; Bakar, L.J.; Altaf, N. Entrepreneurial orientation, big data analytics, and SMEs performance under the effects of environmental turbulence. In Proceedings of the 2021 International Conference on Data Analytics for Business and Industry (ICDABI). Sakheer, Bahrain, 25–26 October 2021.
9. 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 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Jose, CA, USA, 25–27 April 2012; pp. 15–28.

10. Chambi, S.; Lemire, D.; Kaser, O.; Godin, R. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exp.* **2016**, *46*, 709–719. [CrossRef]
11. Ren, R.; Wu, C.; Fu, Z.; Song, T.; Liu, Y.; Qi, Z.; Guan, H. Efficient shuffle management for DAG computing frameworks based on the FRQ model. *J. Parallel Distrib. Comput.* **2021**, *149*, 163–173. [CrossRef]
12. Li, C.; Cai, Q.; Luo, Y. Data balancing-based intermediate data partitioning and check point-based cache recovery in Spark environment. *J. Supercomput.* **2022**, *78*, 3561–3604. [CrossRef]
13. Kumar, S.; Mohbey, K.K. A Utility-Based Distributed Pattern Mining Algorithm with Reduced Shuffle Overhead. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *34*, 416–428. [CrossRef]
14. Choi, J.; Lee, J.; Kim, J.S.; Lee, J. Optimization Techniques for a Distributed In-Memory Computing Platform by Leveraging SSD. *Appl. Sci.* **2021**, *11*, 8476. [CrossRef]
15. Tang, Z.; Zeng, A.; Zhang, X.; Yang, L.; Li, K. Dynamic memory-aware scheduling in spark computing environment. *J. Parallel Distrib. Comput.* **2020**, *141*, 10–22. [CrossRef]
16. Zeidan, A.; Vo, H.T. Efficient spatial data partitioning for distributed *k*NN joins. *J. Big Data* **2022**, *9*, 77. [CrossRef]
17. Zhao, Y.; Dong, J.; Liu, H.; Wu, J.; Liu, Y. Performance improvement of dag-aware task scheduling algorithms with efficient cache management in spark. *Electronics* **2021**, *10*, 1874. [CrossRef]
18. Tang, Z.; Lv, W.; Li, K.; Li, K. An intermediate data partition algorithm for skew mitigation in spark computing environment. *IEEE Trans. Cloud Comput.* **2018**, *9*, 461–474. [CrossRef]
19. Jiang, K.; Du, S.; Zhao, F.; Huang, Y.; Li, C.; Luo, Y. Effective data management strategy and RDD weight cache replacement strategy in Spark. *Comput. Commun.* **2022**, *194*, 66–85. [CrossRef]
20. Bazai, S.U.; Jang-Jaccard, J.; Alavizadeh, H. Scalable, high-performance, and generalized subtree data anonymization approach for Apache Spark. *Electronics* **2021**, *10*, 589. [CrossRef]
21. Modi, A.; Rajan, K.; Thimmaiah, S.; Jain, P.; Mann, S.; Agarwal, A.; Shetty, A.; Gosalia, A.; Partho, P. New query optimization techniques in the Spark engine of Azure synapse. *Proc. VLDB Endow.* **2021**, *15*, 936–948. [CrossRef]
22. Chen, Z.; Yao, B.; Wang, Z.J.; Zhang, W.; Zheng, K.; Kalnis, P.; Tang, F. ITISS: An efficient framework for querying big temporal data. *GeoInformatica* **2020**, *24*, 27–59. [CrossRef]
23. Shen, M.; Zhou, Y.; Singh, C. Magnet: Push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.* **2020**, *13*, 3382–3395. [CrossRef]
24. Qalati, S.A.; Qureshi, N.A.; Ostic, D.; Sulaiman, M.A. An extension of the theory of planned behavior to understand factors influencing Pakistani households' energy-saving intentions and behavior: A mediated–moderated model. *Energy Effic.* **2022**, *15*, 40. [CrossRef]
25. 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.* **2021**, *12*, 122. [CrossRef]
26. Hammami, S.M.; Ahmed, F.; Johny, J.; Sulaiman, M.A. Impact of knowledge capabilities on organizational performance in the private sector in Oman: An SEM approach using path analysis. *Int. J. Knowl. Manag. (IJKM)* **2021**, *17*, 15–18. [CrossRef]
27. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Murthy, R. Hive: A Warehousing Solution over A Map-Reduce Framework. *Proc. VLDB Endow* **2009**, *2*, 1626–1629. [CrossRef]
28. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013.
29. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010.
30. Hunt, P.; Konar, M.; Junqueira, F.P.; Reed, B. ZooKeeper: Wait-free coordination for internet-scale systems. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC'10), Boston, MA, USA, 23–25 June 2010.
31. Borthakur, D. HDFS architecture guide. *Hadoop Apache Proj.* **2008**, *53*, 2.
32. Ivanov, T.; Rabl, T.; Poess, M.; Queralt, A.; Poelman, J.; Poggi, N.; Buell, J. Big data benchmark compendium. In Proceedings of the 7th TPC Technology Conference, Kohala Coast, HI, USA, 31 August–4 September 2015.

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.