Hindawi
Complexity
Volume 2020, Article ID 6617149, 25 pages
https://doi.org/10.1155/2020/6617149

Research Article

Exploiting Sharing Join Opportunities in Big Data Multiquery Optimization with Flink

Xiao-Yan Gao,[1] Radhya Sahal ,[2,3] Gui-Xiu Chen ,[4] Mohammed H. Khafagy,[5] and Fatma A. Omara[2]

[1]*School of Mathematics and Statistics, Yulin University, Yulin 719000, China*
[2]*Faculty of Computers and Information, Cairo University, Cairo, Egypt*
[3]*Faculty of Computer Science and Engineering, Hodeidah University, Hodeidah, Yemen*
[4]*School of Mathematics and Statistics, Qinghai Normal University, 810008 Xining, China*
[5]*Faculty of Computers and Information, Fayoum University, Fayoum, Egypt*

Correspondence should be addressed to Radhya Sahal; radhya.sahal.dsi@gmail.com and Gui-Xiu Chen; chenguixiu@qhnu.edu.cn

Received 22 October 2020; Accepted 8 November 2020; Published 7 December 2020

Academic Editor: Ahmed Mostafa Khalil

Multiway join queries incur high-cost I/Os operations over large-scale data. Exploiting sharing join opportunities among multiple multiway joins could be beneficial to reduce query execution time and shuffled intermediate data. Although multiway join optimization has been carried out in MapReduce, different design principles (i.e., in-memory Big Data platforms, Flink) are not considered. To bridge the gap of not considering the optimization of Big Data platforms, an end-to-end multiway join over Flink, which is called Join-MOTH system (J-MOTH), is proposed to exploit sharing data granularity, sharing join granularity, and sharing implicit sorts within multiple join queries. For sharing data, our previous work, Multiquery Optimization using Tuple Size and Histogram (MOTH) system, has been introduced to consider the granularity of sharing data opportunities among multiple queries. For sharing sort, our previous work, Sort-Based Optimizer for Big Data Multiquery (SOOM), has been introduced to consider the implicit sorts among join queries. For sharing join, additional modules have been tailored to the J-MOTH optimizer to optimize sharing work by exploiting shared pipelined multiway join among multiple multiway join queries. The experimental evaluation has demonstrated that the J-MOTH system outperforms the naive and the state-of-the-art techniques by 44% for query execution time using TPC-H queries. Also, the proposed J-MOTH system introduces maximal intermediate data size reduction by 30% in average over Hadoop-like infrastructures.

1. Introduction

Recently, data size has become too big and complex to capture, store, and process for insightful analysis among industry influencers, academicians, and other prominent stakeholders. The value of Big Data remains a challenge across different application domains such as Governance [1], Agriculture [2], Industry [3, 4], Culture [5], Robotics [6], Healthcare [7], and other. In smart governments, the biggest challenges are the integration and interoperability of Big Data across different government departments and related organizations. Also, Big Data analysis platforms and tools are involved in smart farming to improve the productivity of agriculture. Furthermore, an industrial IoT data processing poses some unique challenges when endeavoring to make Big Data processing a reliable solution for Industry 4.0, which is a term that describes the fourth generation of industrial activities enabled by the systems of smart data analytics and Internet-based solutions. Moreover, healthcare sector deals with a huge amount of data that holds health-related information generated from sensors within wearable devices (i.e., a healthcare Cloud of Things (CoT)) scenario. This health database is difficult to analyse and shows patterns which are useful in the medical field.

Substantially, to capitalize on Big Data opportunities among different sectors, the Big Data distributed computing

platforms are growing every day with the increase of workstations' power and the datasets' size. Therefore, the development and implementation of the distributed system for Big Data applications are considered a challenge [8, 9]. One of the famous frameworks that have emerged for Big Data processing is MapReduce, which is first introduced by Google in 2004 [10]. The main concept of MapReduce is to abstract the details of a large cluster of machines to facilitate the computation on large datasets. Recently, many research studies in academia and industry have proposed new data processing systems such as Flink and Spark to improve the performance of analysis in data applications including query optimization techniques [11–13].

Multiquery optimization (MQO) is an essential key process of query processing in the database systems [14]. MQO is an NP-Hard problem, and many algorithms have been developed to solve such problem to define an appropriate execution plan for each query and minimize the total execution time by performing the common tasks only once [15–17]. Currently, MQO problem has re-emerged in the systems of Big Data analysis especially when the datasets to be processed are getting very large. Therefore, optimizing analytical queries becomes an important issue to overcome computation overheads [18, 19]. Thereby, considering the sharing data techniques among Big Data multiquery becomes urgently required *especially for I/O-intensive applications.*

The cost of job execution on Big Data environment is very expensive. Therefore, it should be saved by avoiding redundant computation especially for I/O-intensive queries such as join queries (i.e., two-way and multiway join) [20]. The join queries' execution cost over large datasets can be optimized by reusing the computed results from previous queries. In this respect, our previous work, Multiquery Optimization Using Tuple Size and Histogram (MOTH) System, has been proposed to consider the granularity of fully and partially reused-based opportunities with considering nonequal tuples' size and nonuniform data distribution, to avoid repeated data loading [21]. Also, our previous work, Sort-Based Optimizer for Big Data Multiquery (SOOM) has been proposed to consider the implicit and explicit sorts among aggregations and sort queries, respectively [22]. Functionally, a single Hadoop MapReduce job is performed through three phases: Map, Shuffling, which contains implicit Sorts, and Reduce phase [10, 23]. Although the sharing data opportunities (i.e., fine-grained and coarse-grained reused-based) are exploited and significantly performed to optimize multiquery, they are not capable enough of optimizing multiple join queries. These join queries need to be explored in order to find the sharing opportunities including shared joins and implicit sorts to optimize multiple join especially in the case of storage constrains (i.e., low storage).

Basically, join data using Hive MapReduce as follows: (1) each mapper reads the data from join tables then returns the join key and join value pairs into an intermediate file, (2) each intermediate file is sorted and merged in the shuffle stage using JOIN keys, and (3) each reducer takes this sorted result as input and then it completes the task of join operation (see Figure 1) [24]. Thus, the overhead of disk I/O dominates the time efficiency of join MapReduce jobs [25]. Substantially, the shuffle step is considered expensive since it needs to sort and join all tuples. Therefore, the shuffling operations need to be optimized to improve the join performance and reduce the total intermediate data size of join query [26]. However, exploiting sharing opportunities including loading, sorting, and joining data among multiple join queries is a challenging task. To address this challenge, the MOTH system is extended to Join-MOTH (J-MOTH) system which targets the coarse-grained opportunity of reusing intermediate results to optimize join queries. In particular, the proposed J-MOTH system aims to exploit both sharing data and sharing work (i.e., join and sort) among multiple queries either two-way or multiway join [27]. Functionally, additional modules have been tailored to the J-MOTH optimizer to optimize sharing work by exploiting shared pipelined multiway join and shared implicit sorts among multiple multiway join queries.

Data are enormously generated by IoT devices. Thus, the Complex Event Processing (CEP) optimization becomes an important research problem for the analysis of real-time streaming data. In many real-world applications, the streaming data analysis could be incorporated with historical data analysis to gain a comprehensive analysis of data [28]. This could be done by executing complex queries such as multiway joins on the streaming data as well as historical data for online stream-based analysis and offline batch-based analysis, respectively, on Big Data stream-processing platform such as Apache Flink [29]. Although multiway join optimization has been carried out in Hadoop MapReduce, different design principles (i.e., in-memory Big Data platforms such as Apache Flink) are not considered [30].

The main *contribution of this paper is* the proposed J-MOTH system which is used to

(1) Provide multiway join execution strategy on Flink platform to capitalize on its capabilities

(2) Avoid shuffling of redundant large intermediate results within join and implicit sort operations over in-disk Big Data platforms (i.e., Hadoop MapReduce) especially in the case of slow storage and space constraints

(3) Evaluate multiple two-way and multiway join execution strategy on MapReduce and Flink

The rest of this paper is organized as follows. Related work is summarized in Section 2. The details of the proposed J-MOTH system's modules are described in Section 3. The functions of the J-MOTH system for two-way join and multiway join queries are introduced in Section 4 and Section 5, respectively. The J-MOTH system experimental evaluation is provided in Section 6. Finally, conclusions and future work are presented in Section 7.

2. Related Work

Several research works in the domain of MapReduce-based systems have been carried out to optimize Big Data analysis,

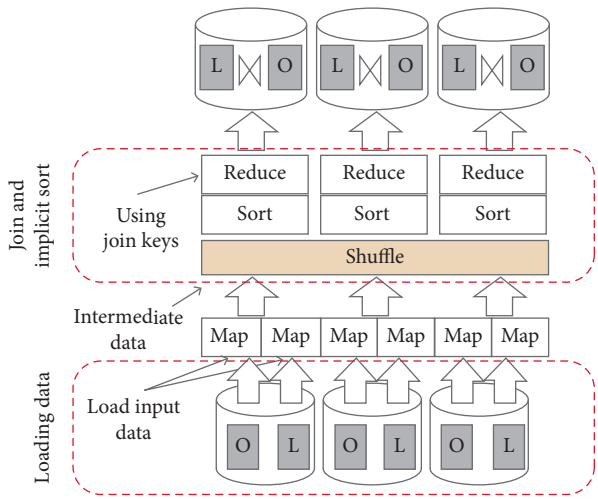

Figure 1: MapReduce join query execution.

especially multiquery optimization. The existed research can be broadly classified into two categories, namely, sharing multiquery optimization and join optimization.

2.1. Sharing Multiquery Optimization. MRShare is a concurrent sharing system which assumes that I/O cost is dominant [18]. Accordingly, it considers the sharing opportunities of scans, Map output, and Map functions sharing. However, the Relaxed MRShare relaxes and generalizes the overlapped queries to increase sharing opportunities into a single job (i.e., generalized grouping technique and materialization technique) [31]. In this respect, a comparative study of MRShare and Relaxed MRShare techniques using predicate-based filters on MapReduce is introduced in [32]. The comparative study has confirmed that the Relaxed MRShare technique significantly improves query execution time for shared data regarding predicate-based filters in MapReduce.

On the contrary, ReStore system is one of the nonconcurrent sharing systems built on top of Pig to optimize query evaluation using materialized results [19, 33]. ReStore system uses heuristics algorithm to select the suitable materialized results even for the complete or part of Map and/or Reduce output of each job. The produced materialized output by ReStore system might not be reused in case the query workloads are not repeated which causes storage overheads. The work in [34, 35] considers the MapReduce reusing intermediate results for failure resilience reasons as materialized views, where semantic UDF models (i.e., User-Defined Functions) based on Hive have been used to enable effectively reusing views where subsequent queries can be evaluated faster. Recently, a vertex-centric graph algorithm, called BIGSUBS, is proposed to iteratively select subexpressions in parallel to be materialized over large workloads [36]. The BIGSUBS algorithm is implemented using SCOPE to assess its effectiveness over production workloads.

2.2. Join Optimization. Different MapReduce join strategies for Big Data analysis are studied and compared [37, 38]. The selection of join strategy is based on data structures and MapReduce programmers. A join query optimizer, AQUA, was proposed using a cost model which is based on precomputed histograms for MapReduce-based processing systems [39]. Furthermore, a bloom filter was extended to optimize join queries for large dataset using MapReduce [40]. On the contrary, JOUM (Join Once Use Many) was proposed to improve the speed up of Hive join query (i.e., MapReduce level tasks) by using pipeline materializing of the full joined star schema [24]. In addition, the improved JOUM which is termed Key/Facts indexing materializes the star schema data and builds an index for this materialized data to optimize join on Hive [41]. Besides, an optimized distributed join algorithm, namely, BlockJoin, has been proposed to reduce shuffling cost of intermediate data by merging relational and linear algebra operations into specialized physical operators [42]. In particular, the BlockJoin algorithm is based on two concepts, index-join and late materialization, which are known in the context of parallel dataflow engines. In addition, an index-based system for reusing data called indexing HiveQL Optimization for join over Multisession Big Data Environment (iHOME) was presented [26]. The proposed iHOME system addresses eight cases of join queries which are classified into three groups: Similar-to-iHOME, Compute-on-iHOME, and Filter-of-iHOME. According to the experimental results of the iHOME system using a benchmark, it was found that the execution time of eight join queries using iHOME on Hive has been reduced.

Regarding the multiway join, a cost-based using histogram on MapReduce, JOMR, with ordering consideration was proposed to rearrange the joinable tables within multiway join query to minimize shuffle time [43, 44]. Also, the simultaneous pipeline technique was introduced in QPipe in advance which can efficiently evaluate query execution plans produced by a multiquery optimizer [45, 46]. Then, the pipelining technique is improved to deliver data to downstream operators more promptly within multiway join which increases the degree of parallelism, improves utilization, and reduces query response time [47]. Michiardi et al. [48] have proposed a method for multiquery optimization which is combined within memory cache primitives to improve the efficiency of data-intensive frameworks. The proposed method produces sharing plans by exploiting common subexpressions while prevailing over memory constraints. They modelled the multiquery optimization cost using a multiple-choice knapsack problem. Furthermore, a general database partitioning method, called GPT, has been proposed for complex analytical queries to improve the query's performance and reduce data redundancy [49]. The GPT method determines an undirected multigraph as a partitioning scheme by considering the trade-off between data redundancy and the number of opportunities for joins processing without shuffling. Besides, an executable Map-Join-Reduce programming model based on Haskel has been introduced [50]. The key idea of the Map-Join-Reduce model is adding a new join module to MapReduce based on filtering-join-aggregation MapReduce to optimize multiway join.

Interestingly, there is no contribution towards multiple join queries optimization over in-memory-based Big Data platform, Flink. Therefore, we aim to design a multiple join optimizer for Flink in order to exploit its in-memory processing capabilities for Big Data analytics. The description of other approaches regarding functionalities, methodologies, and the differences relative to the proposed J-MOTH system are presented in Table 1.

3. The Proposed J-MOTH System

Interestingly, even traditional key RDBMS players such as Oracle are now focusing on NoSQL products. There are many other comparable open source initiatives based on a key-value model including column-based databases (e.g., HBase and Cassandra) and document-based databases (e.g., MongoDB and CouchDB). Besides, some SQL-like engines are also based on a key-value model coupled with cluster processing frameworks. For example, MapReduce-based Hive and Pig, SparkSQL, and Table API are developed on top of MapReduce, Spark, and Flink, respectively. In particular, these SQL-like engines are developed to run across a large number of clusters to achieve high throughput for NoSQL over large-scale data without a predefined schema. Furthermore, the NoSQL data-driven applications make the enterprises and researchers realize a competitive advantage of being able to act on high volume data using NoSQLkey-value based query engines, besides Big Data platforms. For instance, as Cassandra has become one of the most widely used NoSQL databases, it has been used for NoSQL engine and storage within a general architecture for modular time series management called ModelarDB, where Spark is used for query processing [56].

In spite of the evolution of NoSQL and fast keyed-value retrievals, join queries as one of the most common complex queries still incurs more disk I/Os which are needed to get the desired data (e.g., in typical queries involving large scans of rows). These NoSQL join queries which are often called NoJoin are mostly optimized by performing early sort operations to reduce data shuffling. Furthermore, for multiple join queries deployed on Big Data query engines, the redundant computations including data joining and data shuffling could be eliminated by simply reusing intermediate query results (i.e., sorted and joined results) for shared queries. Therefore, the proposed MOTH system has been extended to the J-MOTH system to optimize Big Data multiple join queries including two-way and multiway joins, by exploiting sharing data granularity, sharing join granularity, and sharing implicit sorts within multiple join queries.

3.1. J-MOTH System Modules. The proposed J-MOTH system consists of two layers which are J-MOTH multiquery optimizer and Big Data Analytical Platforms including SQL-Like query interfaces such as Table API for Flink and Hive for Hadoop MapReduce (see Figure 2). We have started with MapReduce (i.e., the state-of-the-art Big Data processing framework) to develop J-MOTH system modules for multiway join optimizer over Flink.

The J-MOTH system has been developed using modular approach. Thus, it is easy to scale out for new proposed multiquery optimization algorithms, as well as new SQL-like query interfaces and Big Data platforms to improve its performance. The main modules of the J-MOTH system are Query Explorer, MOTH Optimizer, J-MOTH Optimizer (i.e., including Join Optimizer and Sort Exploiter), and Query Rewriter. These modules interact together to generate an optimized multiquery execution plan.

3.1.1. Query Explorer Module. The function of Query Explorer module is exploiting additional sharing opportunities among multiquery. It branches each input query using MOTH Query Parser into two main parts which are data and work. The data part includes predicate filters, while the work part includes join and sort operations.

3.1.2. MOTH Reused-Based Query Optimizer Module. Our proposed MOTH system is included as a module in the extended J-MOTH system to exploit sharing data (see Figure 3) [21]. It contains a set of modules such as Query Parser, Sharing Classifier, Reused-based Optimizer, and Query Rewrite. The Query Parser parses the input query, while the sharing classifier classifies the input queries into a set of sharing data queries' list and nonsharing data queries' list. The Reused-based Optimizer optimizes the shared queries based on the granularity of sharing using histogram and metadata for nonuniform data distribution and variable tuple sizes. The Query Rewrite rewrites the final sharing data plan which is submitted back to Hadoop cluster to be executed.

3.1.3. J-MOTH Reused-Based Query Optimizer Module. The J-MOTH optimizer consists of two submodules which are Join Optimizer and Sort Exploiter. The Join Optimizer module leverages the JOMR technique (i.e., Join Order in MapReduce) and SP technique (i.e., Simultaneous Pipeline) for join ordering MapReduce optimization, and simultaneously pipelines join queries' execution, respectively [27, 43]. Furthermore, the J-MOTH optimizer extends the MOTH Reused-based Estimator to estimate all reused-based opportunities of sharing work including join and sort operations. Also, it extends the MOTH Reused-based Enumerator to select the cheapest reused-based opportunity to optimize multiple join queries. A brief description of Sort Exploiter is as follows. Sort Exploiter: although the proposed MOTH system exploits the coarse-grained reused-based opportunities among shared multiquery with respect to nonuniform data distribution, it incurs time overheads because of implicit sort operations (i.e., ORDER BY). Therefore, the Sort Exploiter module of J-MOTH system is developed based on our proposed SOOM system [22]. The Sort Exploiter module is responsible for exploiting implicit sort operations among multiple shared join queries. Then, it performs the shared ORDER BY only once in the starting execution of multiquery execution plans.

Table 1: The comparison of the J-MOTH system with other proposed solutions.

No.	Reference	Functionalities description	Query type	Storing type	Data distribution	Methodologies	Difference relative to J-MOTH system
1	MRShare: sharing across multiple queries in MapReduce [18]	The concurrent-sharing framework that exploits sharing opportunities among multiple jobs by grouping them into a single job	Aggregation	Temporary	Uniform	Grouping shared queries	The sharing optimization of similar work and overlapped work in MRShare and relaxed MRShare, respectively, are considered fine-grained sharing
2	Multiquery optimization in MapReduce framework [31]	Relaxing and generalizing MRShare overlapping queries to increase sharing opportunity into a single job	Aggregation	Temporary	Uniform	Grouping materialization of overlapped queries	
3	Restore: reusing results of MapReduce jobs [19]	Nonconcurrent sharing system that optimizes query evaluation using materialized results produced by pig workflows of MapReduce jobs	N/A	Permanent	Uniform	Materialization	The worst case of using both of HOME and ReStore systems is that the output of a full or subjob is not reused by future queries; thus, these existing systems suffer from big data storage limitation which incurs a high cost to buy additional physical storages or rent extra virtual storages in big data environment
4	HOME : HiveQL optimization in multisession environment [51]	HOME system improves Hadoop performance by storing data of previous results and using it in the next run for the same session or for a different session	Selection aggregation join	Permanent	Uniform	Materialization	
5	Improving the performance of Hadoop Hive by sharing scan and computation tasks [52]	MQO framework, SharedHive, improves the overall performance of Hadoop by grouping correlated HiveQL queries into a new set of queries	N/A	Temporary	Uniform	Grouping	It exploits the temporary fine-grained correlated queries to optimize multiquery for only one session over Hive
6	Reuse-based optimization for pig Latin [53]	The PigReuse system operates common subexpressions using the algebraic representations of pig Latin scripts and reuse-based algorithms to share its results	N/A	Temporary	Uniform	Reused-based	It reutilizes the fine-grained reused-based opportunities to optimize multiquery over pig without considering data distribution
7	Exploiting Soft and Hard Correlations in big data query optimization [54]	EXORD system has been proposed to exploit the granularity of data correlations (i.e., soft and hard) to improve big data query optimization	N/A	Temporary	Uniform	Materialization	It deals with multiquery sharing and reuses fine-grained computed results based on sharing correlations
8	JOUM : an indexing methodology for improving join in Hive star schema [24]	JOUM (join once use many) improves the speed-up of Hive join query tasks by using pipeline materializing of the full joined star schema	Join	Permanent	Uniform	Materialization	It does not exploit sharing among join queries; it optimizes individual incoming join query based on the prejoined schema

Table 1: Continued.

No.	Reference	Functionalities description	Query type	Storing type	Data distribution	Methodologies	Difference relative to J-MOTH system
9	Optimizing join in Hive star schema using Key/Facts indexing [41]	Proposing key/facts indexing to materialize the star schema data and build an index for this materialized data to optimize JOIN on Hive	Join	Permanent	Uniform	Materialization	It does not exploit sharing among join queries; it optimizes individual incoming join query based on the indexed prejoined schema
10	BlockJoin: efficient matrix partitioning through joins [42]	Proposing an optimized distributed join algorithm, namely, BlockJoin to reduce shuffling costs of intermediate data by merging relational and liner algebra operations into specialized physical operators	Join	Temporary	Uniform	Materialization and index join	It does not exploit sharing among join queries; it optimizes individual incoming join query based on the indexed and materialization strategy (i.e., early and late) based on the shape of tables
11	Wide table layout optimization based on column ordering and duplication [55]	Proposing a fine-grained cost model for column accesses and column duplication to optimize HDFS I/O cost of a query workload	Join	Temporary	Uniform	N/A	It does not exploit sharing among join queries; it implements the fine-grained cost model using simulated annealing-based column ordering algorithm to find the approximated optimal column orders and combines with storage constrained column duplication algorithm to optimize I/O throughput
12	Selecting subexpressions to materialize at datacenter scale [36]	Proposing a vertex-centric graph algorithm called BIGSUBS to iteratively select subexpressions in parallel to be materialized over very large workloads	Join	N/A	Uniform	Materialization	A subexpression selection is mapped to a bipartite graph labeling problem and then it is solved in an iterative manner
13	In-memory caching for multiquery optimization of data-intensive scalable computing workloads [48]	Proposing a method for multiquery optimization which is combined within memory cache primitives to improve the efficiency of data-intensive frameworks	Join	Temporary	Uniform	N/A	It exploits sharing opportunities by caching distributed relations
14	A parallel query processing system based on graph-based database partitioning [49]	Proposing a novel graph-based database partitioning method called GPT which considers the trade-off between data redundancy and the number of opportunities for joins processing without shuffling	Join	N/A	Uniform	Grouping, sorting, and partitioning	It exploits sharing by using the partitioning method (i.e., hash-based multicolumn (HMC)), while our work considers data granularities and implicit sort to reduce shuffling in multiquery

Table 1: Continued.

No.	Reference	Functionalities description	Query type	Storing type	Data distribution	Methodologies	Difference relative to J-MOTH system
15	An executable specification of map-join-reduce using Haskell [50]	Proposing an executable Map-Join-Reduce programming model based on Haskell, the key idea of the Map-Join-Reduce model is adding a new join module to MapReduce which is based on filtering-join-aggregation MapReduce to optimize multiway join	Join	N/A	Uniform	N/A	Map-Join-Reduce optimizes single multiway join, while J-MOTH considers sharing opportunities among multiple multiway join queries on MapReduce and Flink
16	Exploiting coarse-grained reused-based opportunities in big data multiquery optimization [21]	Proposing the MOTH system to exploit the coarse granularity of full and partial sharing in multiquery on slow storages	Selection projection	Temporary	Nonuniform	Materialization	MOTH system is our previous work, and it tackles sharing data within big data multiquery
17	SOOM : sort-based optimizer for big data multiquery [22]	Proposing the MOTH system to exploit the sharing sort opportunities, including explicit sorts of sort queries and implicit sorts of aggregation queries	Aggregation and sort queries	Temporary	Nonuniform	Materialization	SOOMM system is our previous work, and it tackles sharing data and work within big data multiquery including explicit sorts of sort queries and implicit sorts of aggregation queries
18	J-MOTH : exploiting sharing work for big data multiquery optimization	Proposing the J-MOTH system to exploit the coarse granularity of sharing data besides the implicit sorts in two-way join and pipleined multiway join queries	Join (two-way and multiway)	Temporary	Nonuniform	Materialization	Our contribution in this work

3.1.4. Query Rewriter Module. According to our proposed MOTH system, the final outputs, n queries, are rewritten into two multiquery plans, namely, Nonconcurrent Multiquery and Concurrent Multiquery Plans [21]. The Nonconcurrent Multiquery Plan holds a list of multiple scheduled queries while the Concurrent Multiquery Plan contains multiple simultaneous queries which are executed concurrently to improve multiquery performance. Moreover, some modifications have been introduced for these two plans to support optimized multiple join (see the pseudocode of the proposed J-MOTH system in Algorithm 1–3 for the J-MOTH optimizer, refining shared multiquery and updating weight, respectively).

4. J-MOTH System for Sharing Two-Way Join Queries

In this section, the reused-based opportunity for multiple two-way join queries problem with respect to granular sharing data is described. Then, the J-MOTH system optimizer for two-way join is presented.

4.1. Reused-Based Opportunities for Multiple Two-Way Join Queries' Problem. Multiple join queries might share similar data which means that several join queries can scan and filter the same or part of the same database files. Consequently, there is an opportunity for some queries to reuse the full results of other queries. This opportunity is possible when the queries share the relations, selected attributes, and filtered data. To clarify this, we assume that the input join queries are specified in a high-level query language. Each input join query is modelled as (JR, JA, JCond, JP), where JR is the relation names (i.e., table name(s)), JA is the set of attributes, $JCond$ is the join condition (i.e., a common column between two tables), and JP is the selection predicate filter which is applied over tables to retrieve the data that should be joined. Using the abovementioned notations, the Reused-based Opportunities of shared predicates for Two-Way Join (**ROTWJ**) could be defined as the exploiting of shared predicates (i.e., identical or subset) among multiple two-way join queries which are defined according to this work

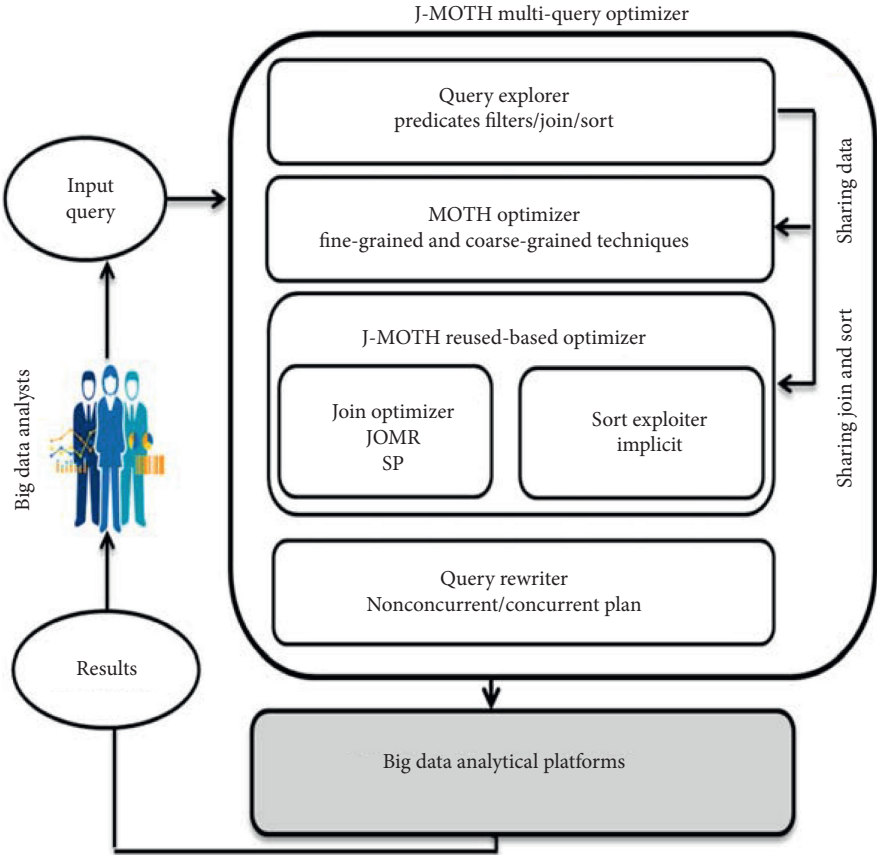

Figure 2: The J-MOTH system architecture.

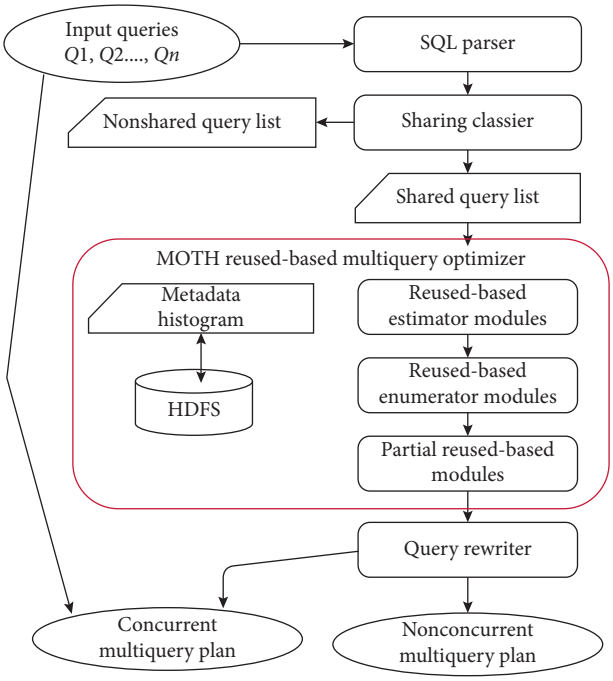

Figure 3: The MOTH system architecture [21].

Input: $Q_{\text{input}} = [Q_1, Q_2, \ldots, Q_n]$
Output: $Q_{\text{nonconcurrent}} = [Q_1, Q_2, \ldots, Q_m]$
$Q_{\text{concurrent}} = [Q_1, Q_2, \ldots, Q_l]$
//Step 1: Query Parsing
(1) $Q_{\text{parsed}} = \textbf{ParseQuery}(Q_{\text{input}})$
//Step 2: Query Explorer
(2) $Q_{\text{Branched-Data}} = \textbf{Explore Da ta}(Q_{\text{input}})$
(3) $Q_{\text{Branched-TwoJoin}} = \textbf{ExploreTWJoin}(Q_{\text{input}})$
(4) $Q_{Branched-MultiJoin} = \textbf{ExploreMWJoin}(Q_{input})$
//Step 3: Sharing Classifier
(5) $Q_{\text{sharedFully}} = \textbf{GetFullShare d}(Q_{\text{Branched-Data}})$
(6) $Q_{\text{sharedPartial}} = \textbf{GetPartialShare d}(Q_{\text{Branched-Data}})$
(7) $Q_{\text{sharedlst}} = Q_{\text{sharedFully}} \cup Q_{\text{sharedPartial}}$
(8) $Q_{\text{nonsharedlst}} = \textbf{GetNonShare d}(Q_{Branche d-Da ta})$
/* Exploiting Sharing Data*/
//**Step 4:** Refine Sharing Data Multiquery
(9) $G_Q = \textbf{RefineShare dM ultiQuery}(Q_{\text{sharedlst}})$
/*call Algorithm S2*/
//**Step 5**: Update Reused-based Opportunity Weight
/*call Algorithm S3*/
Using Histograms and Metadata
(10) $\text{RO}_Q = \textbf{EstimateShare dD istintic}(G_Q)$
(11) $T_Q = \textbf{EstimateShare dT uples}(G_Q)$,
(12) Histogram)
(13) $\text{T}_{Qsz} = \textbf{EstimateShare dT uples}(G_Q, \text{Metadata})$
$\text{WG}_{\text{QDAG}} = \textbf{Weight}G_Q, \text{ROW}(t(RO_Q), |T_Q|, Tsz_Q)$ //**Step 6:** Generate Multiquery Plan
(14) **foreach** $\text{WG}_{Q_k} \in \text{WG}_{\text{QDAG}}$ *do*
(15) **foreach** $V_i \in WG_{Q_k}$ *do* $V_{\text{FP}} = \textbf{getAllFullParentReuse d}(G_{Q_k}, V_i)$
(16) $V_{FP} = \textbf{getAllPartialParentReuse d}(G_{Q_k}, V_i)$
(17) **if** $(V_{\text{FP}} \neq \text{null})$) **then**
(18) $V_{\text{reusecost}_i} = \min(|\text{Weight}_{V_{\text{FP}}}|)$
(19) Else
(20) $V_{\text{reusecost}_i} = \min(|\text{Weight}_{V_{\text{PP}}}|)$
(21) End if
(22) $G_{\text{Qtree}_k} = G_{\text{Qtree}_k} \cup V_i$
/*update $V_i$level based on its query parent*/
(23) $\textbf{Up da teLevel}(V_i)$
(24) End
(25) $\text{RootQuery}_k = \textbf{GetRoot}(V_i)$
(26) End
(27) $\text{PQ}_{lst} = \textbf{getParentQuery}(G_{\text{Qtree}})$
(28) $\text{SQ}_{lst} = \textbf{getSubQuery}(G_{\text{Qtree}})$
(29) $\text{LeafQ}_{lst} = \textbf{getLeafQuery}(G_{\text{Qtree}})$
/* Exploiting Sharing Join*/
//**Step 7:** Optimize multiway join
(30) $\textbf{JOMR}(Q_{\text{Branched-MultiJoin}})$
(31) $\textbf{SP}(Q_{\text{Branched-MultiJoin}})$
//**Step 8:** Sort Exploiter
(32) $\textbf{ExploitImplictSort}(Q_{\text{Branched-MultiJoin}})$
//**Step 9:** Refine and Rewrite Queries for Multiquery Plan
(33) $Q_{\text{nonconcurrent}} = \textbf{RewriteQuery}(\text{PQ}_{lst}, \text{SQ}_{lst})$
(34) $Q_{\text{concurrent}} = \textbf{RewriteQuery}(\text{LeafQ}_{\text{lst}})$
(35) $Q_{\text{concurrent}} = \textbf{RewriteQuery}(Q_{\text{nonsharedlst}})$

Algorithm 1: J-MOTH reused-based multiquery optimizer.

/∗add each shared group into one subgraph∗/
Input: $Q_{\text{sharedlst}} = [Q_1, Q_2, \ldots, Q_m]$
Output: $G_Q = [G_{Q_1}, G_{Q_2}, \ldots, G_{Q_l}]$
//Step 1: initialization
(1) intSSGNO = 1//shared subgraph no
(2) G_{Q1} = new graph( )
(3) foreachQ_i in $Q_{\text{sharedlst}}$do
//Step 2: pick Q_i to add it to suitable subgraph
(4) foreachkfrom 1 to SSGNO do
(5) BooleanValidNode = false
(6) While j inV_k do
(7) If (sharing[i][j]**in**["F", "P"])then
(8) /∗Q_i defined as V_{k_i} and Q_j defined as V_{k_j} in G_{Q_k}
(9) $V_k = V_k \cup V_{k_i}$
/∗ *Weight* could be initially define in terms of full/partial sharing, F or P. ∗/
(10) Weight = sharing[i][j]
(11) G_{Q_k}.addEdge(V_{k_i}, V_{k_j}, Weight)
(12) ValidNode = true
(13) RemoveQ_i from$Q_{\text{sharedlst}}$
(14) End if
(15) End while//end while
(16) End for//all subgraph traverse for sharing Q_i
//Step 3: add Q_i to new subgraph.
(17) If (!ValidNode) then
(18) SSGNO = SSGNO +1
(19) $G_{Q_{\text{SSGNO}}}$ = new graph( )
/∗ $V_{\text{ssGNO}_i}$is a first query node added whose has not share any node in pervious subgraphs but it is share with the rest query in $Q_{\text{sharedlst}}$ ∗/
(20) $V_{\text{SSGNO}} = V_{\text{ssGNO}_i}$
(21) RemoveQ_i from$Q_{\text{sharedlst}}$
(22) End if
(23) End for
(24) End

Algorithm 2: Refine shared multiquery.

Input: $G_{\text{QDAG}} = [G_{\text{QDAG}_1}, G_{\text{QDAG}_2}, \ldots, G_{\text{QDAG}_l}]$
Output: $\text{WG}_{\text{QDAG}} = [\text{WG}_{\text{QDAG}_1}, \text{WG}_{\text{QDAG}_2}, \ldots, \text{WG}_{\text{QDAG}_l}]$
/∗ updating each DAG graph with respect to reused type and length of reused results in terms of histogram, metadata∗/
(1) foreach$G_{\text{QDAG}_k} \in G_{\text{QDAG}}$ do
(2) foreach$V_i \in G_{\text{QDAG}_k}$do
(3) foreach$V_j \in G_{\text{QDAG}_k}$do
(4) RO_{ij} = Weight(V_i, V_j);
(5) RO_Q = **EstimateShare dD istintic**(G_Q)
(6) T_Q = **EstimateShare dT uplesNumber**(G_Q, Histogram))
(7) Tsz_Q = **EstimateShare dT uplesSize**(G_Q, Metadata)
(8) WG_{QDAG} =**Weight**G_Q, ROW (t(RO_Q), $|T_Q|, Tsz_Q$)
(9) **End.**

Algorithm 3: Update reused opportunity weight.

as follows The used acronyms/notations are listed in Table 2

Definition 1. Reused-based Opportunities of Shared Predicates in Two-way Join (ROTWJ):

$$\forall \text{JQ}_i, \quad \text{JQ}_j, \left(\left(\text{JR}_i = \text{JR}_j\right) \wedge \left(\text{JA}_i \subseteq \text{JA}_j\right) \wedge \left(\text{JP}_i \subseteq \text{JP}_j\right)\right) \longrightarrow \text{ROTWJ}\left(\text{JQ}_i, \text{JQ}_j\right). \tag{1}$$

Table 2: List of Acronyms.

Notation	Original
JA	The set of attributes
JCond	The join condition (i.e., a common column between two tables)
JIS	Join implicit sort
JISSO	Join implicit sort shared opportunity
JQ	Join query
JP	The selection predicate
JR	The relation names (i.e., table names)
LISJ	Largest implicit sort join
ROMWJ	Reused-based opportunities of multiway join
ROTWJ	Reused-based opportunities of shared predicates in two-way join
SP	Simultaneous pipeline technique
SSG	Shared sort group
ROW	Weight of reused-based opportunity
$t(\mathrm{RO}_{ij})$	Type of reused-based opportunity of $Q_i$and Q_j (i.e., fully)
T_{Q_i}	Number of tuples inQ_i
Tsz_{Q_i}	Size of tuple (in bytes) inQ_i

When ROTWJ$(\mathrm{JQ}_i, \mathrm{JQ}_j)$ is captured, the shared predicates will be evaluated only once for two join queries Q_i and Q_j such that JP_i will reuse JP_j. As a result, the redundant evaluation of the same selection predicates will be saved.

Example 1. To avoid loss of generality, consider the following three input two-way join queries over the relations *Employees (EmployeeID, Name, Age, Dept) and Orders (EmployeeID, OrderID, OrderDate):*

Q1 SELECT Employees.Name, Orders.OrderID, Orders.OrderDate **FROM** Orders **JOIN** Employees **ON** Orders.EmployeeID = Employees.EmployeeID **WHERE** Employees.Age ≤ 40

Q2 SELECT Employees.Name, Orders.OrderID, Orders.OrderDate **FROM** Orders **JOIN** Employees **ON** Orders.EmployeeID = Employees.EmployeeID **WHERE** Employees.Age ≥ 20 **AND** Employees.Age< 30

Q3 SELECT Employees.Name, Orders.OrderID, Orders.OrderDate **FROM** Orders **JOIN** Employees **ON** Orders.EmployeeID = Employees.EmployeeID **WHERE WHERE** Employees.Age ≥ 30 AND Employees. Age ≤ 35

Each query is divided into two parts, namely, predicate and join which are denoted by JPandJQ, respectively. Accordingly, $(Q_1, Q_2$ and $Q_3)$ are represented as$(\mathrm{JP}_1, \mathrm{JQ}_1)$, $(\mathrm{JP}_2, \mathrm{JQ}_2)$, and $(\mathrm{JP}_3, \mathrm{JQ}_3)$, respectively; their predicate parts are as follows:

SELECT Employees.Name **FROM** Employees **WHERE** Employees.Age ≤ 40

SELECT Employees.Name **FROM** Employees **WHERE** Employees.Age ≥ 20 **AND** Employees.Age ≤ 30

SELECTEmployees.Name **FROM** Employees **WHERE** Employees.Age ≥ 30 **AND** Employees.Age ≤ 35

It is noted that the three queries share their join parts, and there is overlapping among sharing data parts in terms of predicates such as $(\mathrm{JP}_1, \mathrm{JP}_2, \mathrm{JP}_3)$. For this reason, exploiting the shared selection predicates among multiple two-way join queries yield significant cost saving over large datasets. Also, reusing the filtered data rather than the large original input files can reduce the I/O operations as well as the shuffling time through the network.

4.2. J-MOTH Optimizer for Multiple Two-Way Join Queries. In this section, the proposed J-MOTH optimizer targets the opportunistic reusing of the coarse-grained sharing data to optimize multiple two-way join queries. The workflow of J-MOTH modules for two-way join is presented in three phases which are depicted in Figure 4 and listed in the following items as follows.

4.2.1. Phase 1: Exploring Query. The Query Explorer phase investigates the sharing data among multiple two-way join queries. It branches each input of two-way join query into two parts, namely, join and predicates which are denoted as JQ andJP, respectively (see Figure 5). The key idea of this branching is to exploit the shared selection predicates to optimize multiple two-way join queries.

4.2.2. Phase 2: Exploiting Sharing Data (MOTH Reused-Based Optimizer). Regarding multiple two-way join queries, the MOTH reused-based optimizer is used to generate an optimized plan which aims to exploit the possible sharing data (i.e., coarse-grained reused-based opportunities) among two-way join queries. More specifically, the generated plan schedules two-way join queries in a manner that allows computing the shared selection predicates queries only once. Then, it saves them temporarily to be reused immediately by subsequent shared queries in a batch to reduce the overall query evaluation. The steps of generating the optimized plan for multiple two-way join queries using MOTH modules are as follows:

(1) MOTH Sharing Classifier classifies the extracted predicates of the branched input queries into two predefined lists, which are *SharedQueryList* and non-*SharedQueryList.*

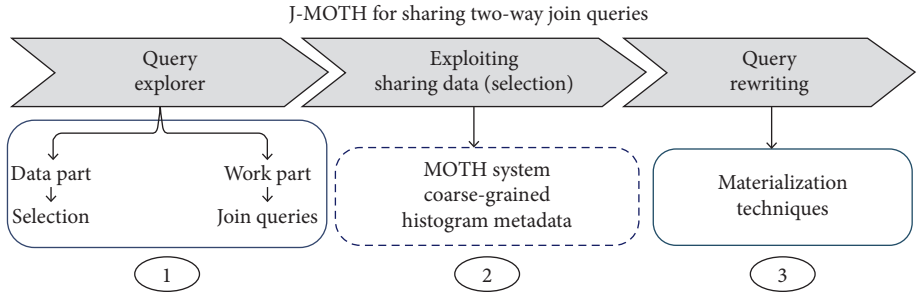

Figure 4: The J-MOTH system phases for two-way join optimization.

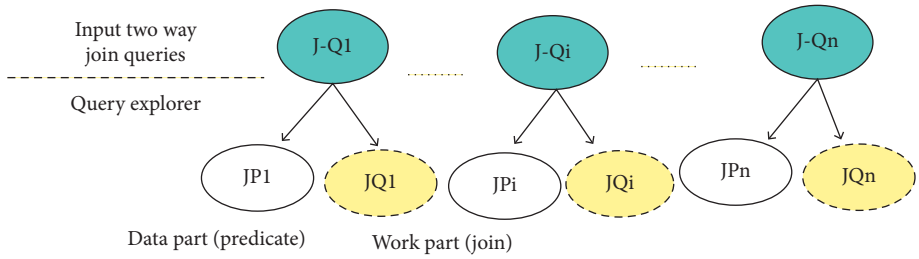

Figure 5: The exploring phase for two-way join queries.

(2) MOTH Reused-based Estimator (i.e., cost model) estimates the reused-based coarse-grained opportunities using MOTH granular technique, *RO*, which is based on metadata and histogram to minimize the loaded and shuffled data size within two-way join queries. The reused-based coarse-grained opportunity of Q_i using Q_j is described in equation (2) as follows:

$$\mathrm{RO}(Q_i) = \mathrm{ROW}\left(t\left(\mathrm{RO}_{ij}\right), \mathrm{Tsz}_{Q_j}, \left|\mathrm{T}_{Q_j}\right|\right), \tag{2}$$

where ROW denotes the weight of reused-based opportunity to retrieve the result (i.e., sharing data) of Q_i by reusing the result of Q_j. $t(\mathrm{RO}_{ij})$ denotes the type of reused-based opportunity of Q_i and Q_j (i.e., Fully). T_{Q_i} and Tsz_{Q_i} denote the number of tuples in Q_i retrieved from the histogram and the size of tuple (in bytes) in Q_j retrieved from metadata, respectively, as described in equations (2) and the following equations:

$$\left|T_{Q_j}\right| = \mathrm{GetTuplesNumberFromHistogram}\left(Q_j\right), \tag{3}$$

$$Tsz_{Q_j} = \mathrm{GetTupleSizeFromMetadata}\left(Q_j\right), \tag{4}$$

see details about the MOTH cost model in [21].

(3) MOTH Reused-based Enumerator receives the estimated reused-based opportunities for the shared predicates and then selects the proper *ParentQuery*(s) to generate multiquery execution plan as follows:

$$\mathbf{Parent}\left(\mathbf{Q_i}\right) = \min\left(\mathrm{RO}\left(Q_{ij}\right)\right), \quad \forall \mathbf{i}, \mathbf{j} \in [1, 2 \ldots, \mathrm{n}], \mathbf{i} \neq \mathbf{j}. \tag{5}$$

Figure 6 shows the generated plan where the qualifying tuples among shared selection predicates are filtered and then fed into two-way join queries. For instance, the *JP*query can be a parent of its pair join query, *JQ* (see highlighted leafs query(s)), as well as other *JP*query. Consequently, reusing the output of the shared queries can reduce the total cost of two-way join queries execution plan rather than using the whole input file to answer such queries.

4.2.3. Phase 3: Rewriting Query. In this phase, the Query Rewriter module rewrites the final outputs, *n* queries, into two plans, namely, Nonconcurrent Multiquery and Concurrent Multiquery. The Nonconcurrent Multiquery Plan holds a list of multiple scheduled queries of sharing data (i.e., sharing predicates, JP). The Query Rewriter module picks leafs (i.e., two-way join queries which are shadowed in Figure 5 within work part) from the generated plan, which can reuse the results of nonconcurrent predicates queries. The Concurrent Multiquery Plan contains multiple simultaneous two-way join queries, JQ, and nonsharing queries *Q* which are executed concurrently to improve multiquery performance. The Two-way Join Multiquery Plan is defined as follows.

Definition 2. Two-way Join Multiquery Plan:

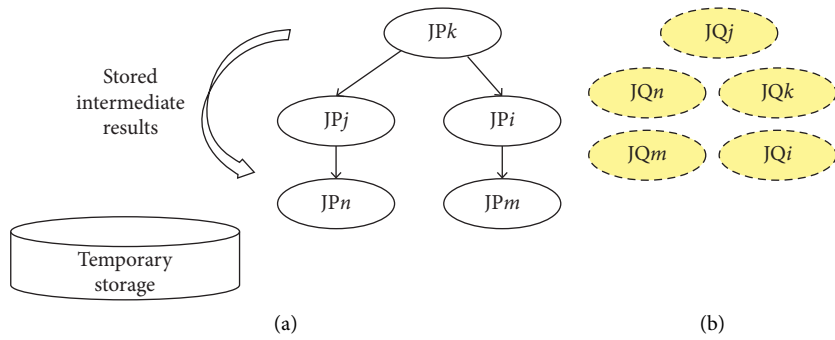

Figure 6: The final generated execution plan of two-way joins queries. (a) Nonconcurrent plan. (b) Concurrent plan.

$$\begin{aligned} \forall Q_i, \quad & (Q_i \in \text{JP}) \longrightarrow Q_i \in \text{NonConQueryList}, \\ \forall Q_i, \quad & (Q_i \in \text{JQ} \vee Q_i \in \text{nonSharedQuery}) \longrightarrow Q_i \in \text{ConQueryList}. \end{aligned} \tag{6}$$

The final optimized multiquery execution plans in terms of nonconcurrent and concurrent are shown in Figure 6. The optimized plans use the temporary storage to store the intermediate results of predicates queries to be immediately reused by multiple two-way join queries. Therefore, the plan eliminates the redundancy which occurs due to accessing the same data multiple times to answer multiple queries.

5. J-MOTH System for Sharing Multiway Join Queries

Recall that the two-way join queries are binary tasks, which means that they can only take two sides, a left side and a right side, unlike the multiway join queries which take more than two sides and can be implemented by linking multiple two-way join tasks together into one job. Regarding a sequence of two-way join tasks, the size of the intermediate joined data could be much larger than that of the final joined answer [57]. Thereby, the data in-network movement time (i.e., shuffle time which is needed to transfer intermediate joined data) has high network I/O cost. According to Big Data multiquery, long query execution times of multiple multiway join queries are incurred by reshuffling the same data multiple times.

According to this work, the proposed J-MOTH optimizer targets the opportunistic reusing of the sharing opportunities to optimize multiple multiway join queries. Particularly, exploiting sharing opportunities including data granularities, joins, and implicit sorts among multiple multiway join queries can be beneficial because it does not process any redundant join results. Also, it avoids extra shuffling costs over the in-disk-based Big Data platform such as Hadoop MapReduce.

5.1. Multiway Joins Reused-Based Opportunities Problem. In this section, we have studied the reused-based opportunities problem among multiple multiway join queries which is presented in two-fold, and it can be described as follows.

5.1.1. Sharing Two-Way Joins in Multiway Join Queries. Similar to two-way join query, we assume that the input multiway join queries are specified in a high-level query language. Each input multiway join query is modelled as (JR, JA, JCond, JP), where JR is the relation names (i.e., table name(s)), JA is the set of attributes, JCond is join condition (i.e., a common column between two tables), and JP is the selection predicate filter which is applied over the table to retrieve data that should be joined. Using the two-way join definition, the multiway joins query can be described as follows:

$$\text{MWJ}_Q = \text{TWJ}_1, \text{TWJ}_2, \ldots, \text{TWJ}_n. \tag{7}$$

Accordingly, the Reused-based Opportunities of Multiway Join (ROMWJ) using the reused-based opportunities of two-way join (ROTWJ) is described as follows:

$$\text{ROTWJ}_1 \subseteq \text{ROTWJ}_2 \subseteq \cdots \subseteq \text{ROTWJ}_i \subseteq \cdots \subseteq \text{ROTWJ}_n. \tag{8}$$

Consequently, the ROMWJ could be described as a set of exploiting ROTWJ among multiple multiway join queries which are defined as follows:

Definition 3. Reused-based Opportunities of Shared Multiway Join (ROMWJ):

$$\forall \text{JQ}_i, \quad \text{JQ}_j, \left(\text{ROTWJ}_i \subseteq \text{ROTWJ}_j\right) \longrightarrow \text{ROMWJ}\left(\text{JQ}_j, \text{JQ}_i\right). \tag{9}$$

When ROMWJ$(\text{JQ}_i, \text{JQ}_j)$ is captured, the shared two-way join will be evaluated only once for two join queries Q_i and Q_j. In particular, if there are some shared two-way join opportunities between Q_i and Q_j which are similar or overlapped, the ROTWJ_j will reuse the joinable results of ROTWJ_i and the redundant join operations will be saved.

Example 2. This example extends the ideas of two-way joins to multiway joins, considering these input HiveQL multiway join queries over TPC-H benchmark which includes *customer, nation, orders,* and *lineitem* relations as follows:

Q1 SELECTc.c_custkey, c.c_name, c.c_address, c.c_phone, c.c_comment, c.c_acctbal, c.c_nationkey, o.o_orderkey, o.o_custkey

FROM customer c

JOIN orders o ON (c.c_custkey = o.o_custkey);

Q2 SELECTc.c_custkey, c.c_name, c.c_address, c.c_phone, c.c_comment, c.c_acctbal, c.c_nationkey, o.o_orderkey, o.o_custkey, l.l_extendedprice, l.l_discount, l.L_RECEIPTDATE, l.L_RETURNFLAG, l.L_COMMENT, l.L_SHIPDATE

FROMcustomer c
JOIN orders o ON (c.c_custkey = o.o_custkey). JOIN lineitem l ON (o.o_orderkey = l.l_orderkey);

Q3 SELECTc.c_custkey, c.c_name, c.c_address, c.c_phone, c.c_comment, c.c_acctbal, c.c_nationkey, o.o_orderkey, o.o_custkey, l.l_extendedprice, l.l_discount, l.L_RECEIPTDATE, l.L_RETURNFLAG, l.L_COMMENT, l.L_SHIPDATE, n.n_name

FROM customer c
JOIN orders o ON (c.c_custkey = o.o_custkey)
JOINlineitem l ON (o.o_orderkey = l.l_orderkey).
JOIN nation n ON (c.c_nationkey = n.n_nationkey)

For simplicity, we assume that the joinable tables have the same orders within the input queries to maximize the sharing join opportunities. Each query is divided into multiple two-way joins where these aforementioned multiway join queries (Q_1, Q_2, Q_3) are represented as follows:

$$\begin{aligned} Q_1&: C \bowtie_{\text{c\_custkey}=\text{o}-\text{custkey}} O \\ Q_2&: C \bowtie_{\text{c\_custkey}=\text{o\_custkey}} O \bowtie_{\text{o\_orderkey}=\text{l\_orderkey}} L \\ Q_3&: C \bowtie_{\text{c}_{\text{custkey}}=\text{o}_{\text{custkey}}} O \bowtie_{\text{o}_{\text{orderkey}}=\text{l}_{\text{orderkey}}} \\ &\quad L \bowtie_{\text{c\_nationkey}=\text{n\_nationkey}} N, \end{aligned} \tag{10}$$

where C, O, L, and N refer to *customer, orders, lineitem,* and *nation* tables, respectively. It can be noticed that these three queries are shared based on the two-way join definition such that ROTWJ_1, ROTWJ_2, ROTWJ_3 in Q_1, Q_2Q_3 can be represented, respectively, as follows

$$\begin{aligned} &ROTWJ_1 = C \bowtie_{\text{c}_{\text{custkey}}=\text{o}-\text{custkey}} O \\ &ROTWJ_2 = ROTWJ_1 \bowtie_{\text{o\_orderkey}=\text{l\_orderkey}} L \\ &ROTWJ_3 = ROTWJ_1 \bowtie_{\text{o\_orderkey}=\text{l\_orderkey}} L \bowtie_{\text{c\_nationkey}=\text{n\_nationkey}} N \end{aligned}$$

OR

$$ROTWJ_2 \bowtie_{\text{c\_nationkey}=\text{n\_nationkey}} N. \tag{11}$$

It can be noted that these reused-based two-way join opportunities are overlapping as follows: $\text{ROTWJ}_1 \subset \text{ROTWJ}_2 \subset \text{ROTWJ}_3$. Obviously, Q_2 can reuse the joined results of the predecessor Q_1 as$Q_1 \prec Q_2$. On the contrary, Q_3 can reuse one of the joined results of the predecessors Q_1 or Q_2 as$Q_1 \prec Q_3$, or $Q_2 \prec Q_3$. As stated in the prior two-way joins, exploiting these shared two-way joins among multiple multiway joins yields significant cost saving in terms of reducing I/O operations, as well as shuffling time through the network. By means of multiway joins, it will be possible to optimize Big Data join processing time besides granular sharing data consideration.

5.1.2. Sharing Implicit Sorts in Multiway Join Queries. Regarding join in Hadoop MapReduce, an implicit sort job is separately performed for each table within multiway join; then, the joins based on the sorted tuples are executed (see Figure 1). Recall that the multiway join is a combination of tuples of n tables such as $T_1, T_2, \ldots,$ and T_n, for each Qi such as

$$T_{i_1} \bowtie_{a_1=a_2} T_{i_2} \bowtie_{a_2=a_3} \cdots \bowtie_{a_{n-1}=a_n} T_{i_n}. \tag{12}$$

Therefore, there are existed multiple Join Implicit Sorts (*JIS*) which are not explicitly seen in the join queries in terms of ORDER BY clause. The *JIS* is defined as follows.

Definition 4. Join Implicit Sort (JIS):

$$\begin{aligned} &\forall Q_i, \quad \exists \text{JIS}(t, a) \mid t \in T,\ a \in A, \\ &\forall Q_i, \quad Q_j, \left(\left(\left(t_{Q_i} = t_{Q_j}\right) \wedge \left(a_{Q_i} = a_{Q_j}\right)\right) \wedge \left(\text{JCond}_{Q_i} = \text{JCond}_{Q_j}\right)\right) \longrightarrow \text{JIS}_{Q_i} = \text{JIS}_{Q_j}, \end{aligned} \tag{13}$$

where each multiway join query contains a set of $\text{JIS}(t, a)$ based on the number of tables in join query and their attributes such as t and a, respectively. Accordingly, the multiple multiway join queries may share similar implicit sorts which allow some multiway join queries to reuse the sorted data of other queries. More specifically, the implicit sort reused-based opportunities between two join queries Q_i and $Q_j$can be captured when the multiway join queries share the following. (1) The same relations (i.e., tables within multiway join query) such as $(t_{Q_i} = t_{Q_j})$, (2) the same corresponding attributes such as $(a_{Q_i} = a_{Q_j})$within join conditions which are implicitly used for sorts, and (3) the join conditions such as $(\text{JCond}_{Q_i} = \text{JCond}_{Q_j})$. Thus, the Join Implicit Sort Shared Opportunity (*JISSO*) can be defined as follows.

Definition 5. Join Implicit Sort Shared Opportunity (JISSO):

$$\forall Q_i, \quad Q_j, \left(\left(\left(\text{JIS}_k \subset \text{JIS}_{Q_i}\right) \wedge \left(\text{JIS}_l \subset \text{JIS}_{Q_j}\right)\right) \wedge \left(\text{JIS}_k = \text{JIS}_l\right)\right) \longrightarrow \text{JISSO}\left(Q_i, Q_j\right), \tag{14}$$

where k^{th} and l^{th} of *JIS(s)* belong to the join implicit sort in Q_i and Q_j, respectively. Therefore, these largest *JISSO* should be exploited among multiple *JISSOs* to optimize multiple multiway join queries.

5.2. J-MOTH Multiway Join Optimizer Overview. As far as we know, the consideration of the coarse-grained sharing data to optimize multiple multiway join queries has not been studied. Therefore, the optimization of multiple multiway join queries which run over larger dataset is addressed through the work in this paper. The key benefit of the J-MOTH multiway join optimizer is to explore opportunistic sharing data to minimize loaded data, as well as exploring sharing join to minimize data in-network movement time by reusing intermediate joined results, besides refining the simultaneous pipeline execution of multiple multiway join.

Substantially, in order to perform the entire multiway join optimization, the sharing data (i.e., fine-grained and coarse-grained) and sharing work (i.e., pipelined two-way joins and implicit sorts) are sequenced into four optimization stages. Figure 7 depicts the progressive optimization of J-MOTH multiway join optimizer through the sharing data and sharing work optimization ordered from left to right.

Firstly, the J-MOTH multiway join optimizer exploits the fine-grained and coarse-grained reused-based opportunities regardless of joinable tables order. And then, it generates an optimized sharing data plan which effortlessly serves multiway join queries by early eliminating the useless data that incurs extra shuffle cost. Secondly, the optimizer orders the tables within each join query using the existing JOMR technique [43]. Thirdly, the multiple implicit sorts can incur time overheads because of redundant internal sort operations. Mostly, exploiting these shared implicit sorts can avoid wasteful shuffled data to optimize multiple multiway joins. Fourthly, SP technique is invoked inside the proposed J-MOTH multiway join optimizer to exploit the overlapped pipelined subplans (i.e., two-way joins) among input queries [27].

5.3. The J-MOTH Multiway Join Optimizer Phases. The J-MOTH multiway join optimizer contains four phases which have been implemented using the J-MOTH modules. These four phases are Query Explorer, MOTH Optimizer, Sort Exploiter, and Query Rewriter; two existed techniques, namely, JOMR and SP (see Figure 8) [27, 43]. These phases integrate together to exploit the maximum reused-based coarse-grained opportunities, shared two-way joins, which will be executed in a pipelined manner depending on ordered joinable tables and implicit sorts to optimize multiple multiway joins over Big Data. Each of these four phases is described below as follows.

5.3.1. Phase 1: Query Exploring. The extended Query Explorer module investigates the sharing among multiple multiway join queries. It branches each input multiway join query into three parts, namely, predicates, implicit sort operations (i.e., ORDER BY), and join, which are denoted as JP, JIS,and *JQ*, respectively (see Figure 9). The key idea of this branching is to exploit shared data such as selection predicates, besides sharing work in terms of shared two-way join and implicit sort to optimize multiple multiway queries.

5.3.2. Phase 2: Exploiting Sharing Data (MOTH Reused-Based Optimizer). Regarding the multiple multiway join queries, the MOTH reused-based optimizer using coarse-grained technique is used to generate an optimized plan for sharing data parts (i.e., the same steps of MOTH optimizer for two-way join are used).

5.3.3. Phase 3: Exploiting Sharing Work. Regarding exploiting sharing work, the two conventional types of multiway join execution plans in DBMS, bushy plan, and left-deep plan are considered (see Figure 10) [43]. The bushy plan executes all the internal joins in parallel and then pushes the results to the last join. The left-deep plan pipelines the subjoins to fulfill the internal results of join query. Thus, it leads to generate cheaper plan than the bushy plan. Therefore, most of DBMSs use left-deep plan in multiway join optimization especially over slow storage because its pipelined processing is simple than that one of the bushy plan [25]. According to the work in this paper, the left-deep plan is considered to exploit sharing work within multiple multiway join queries. The phases of the J-MOTH optimizer are as follows.

(1) Join Order in MapReduce Technique (JOMR). Basically, the JOMR technique is used to rearrange the joinable tables within multiway join query, and then moves the table which needs to scan large input file to the end of join query execution in order to minimize shuffle time. Indeed, two reasons motivate us to invoke the existed JOMR technique [43]. Firstly, the join ordering is extremely sensitive to timing; it reduces shuffling amount of data even if JOMR technique is executed alone in the J-MOTH system. In other words, the different join orders lead to different query plans per single query with significantly different performances. For this, it is important to find the best join order for a multiway join query in the worst Case (i.e., no degree of sharing exhibits among multiple multiway join queries). Secondly, sharing work could be exploited by stating similar joinable tables ordering to increase overlapping degree through the input multiway join queries.

Example 3. For brevity and simplicity, Figure 11 depicts an illustrative example of a single multiway join using TPC-H *Q10* to clarify the JOMR technique idea.

TPC-H Q10

SELECTc.c_custkey, c.c_name, c.c_address, c.c_phone, c.c_comment, c.c_acctbal, c.c_nationkey, o.o_orderkey, o.o_custkey, l.l_extendedprice, l.l_discount, l.L_RECEIPTDATE, l.L_RETURNFLAG, l.L_COMMENT, l.L_SHIPDATE, n.n_name

FROM customer c

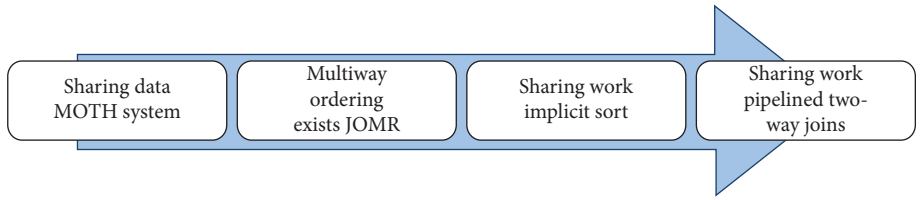

Figure 7: Optimization stages of J-MOTH multiway join optimizer.

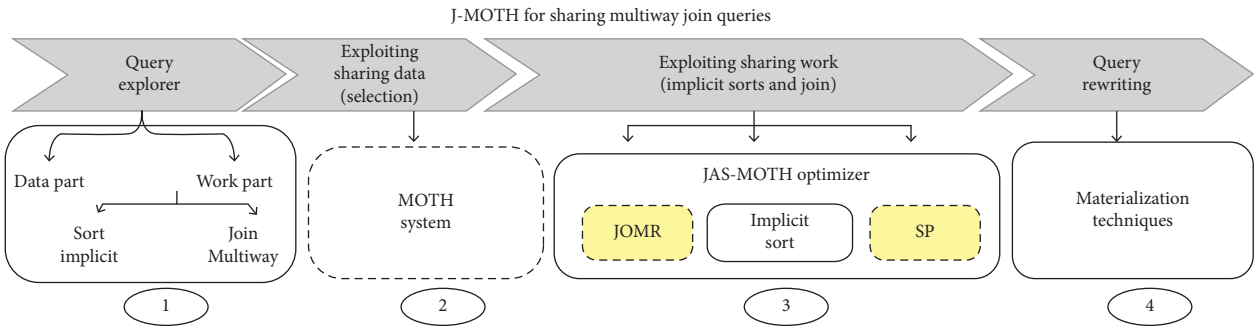

Figure 8: The J-MOTH multiquery optimizer phases for multiway joins.

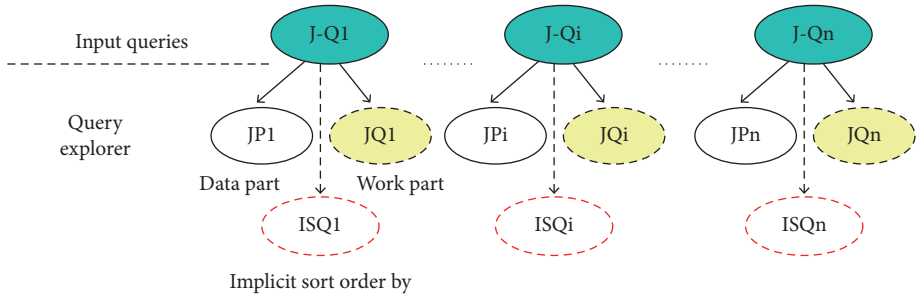

Figure 9: The Query Explorer for multiple multiway join queries.

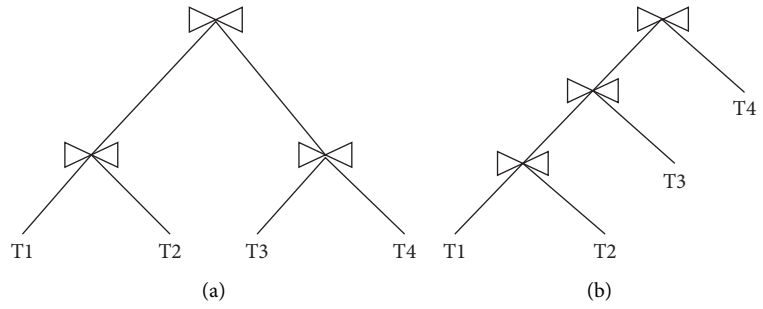

Figure 10: Multiway join execution plan in DBMS. (a) Bushy plan. (b) Left deep plan.

JOIN orders o ON (c.c_custkey = o.o_custkey)
JOINlineitem l ON (o.o_orderkey = l.l_orderkey)
JOIN nation n ON (c.c_nationkey = n.n_nationkey)

The order of tables in the original query, *Q10*, boils down to *customer*, *orders*, *lineitem*, and *nation*. Using the JOMR cost model, the optimized order is *customer, nation, orders*, and *lineitem*. The JOMR optimizes the order by rearranging both of *nation* and orders tables and moves the *lineitem* table to the end of the query because of its large size. This modification of table order can optimize the multiway join query by avoiding large shuffled intermediate data using early join regarding *lineitem* table.

(2) Implicit Sort Exploiter. Regarding the preceding phases, the coarse-grained reused-based opportunities are exploited

to eliminate the redundant filtrations using the proposed MOTH system. And then, the joinable tables are ordered using JOMR technique to reduce the intermediate data by rearranging the tables based on their sizes.

Basically, each file within the join query has its implicit sort which is not seen in the join queries. The large file has the large implicit sort which needs large number of I/Os operations. In the case of shared join, the large files and their implicit sort are shared as well. Besides these optimization phases among multiple multiway join queries, the redundant implicit sorts should be avoided whenever possible due to their expensive operational costs. The key idea of the J-MOTH Implicit Sort Exploiter is to find the largest join implicit sort shared opportunities, *JISSO*(s). So, the J-MOTH Implicit Sort Exploiter selects *JISSO* which has the most expensive sort cost among ordered multiway joins which are ordered by JOMR technique. And then, the Implicit Sort Exploiter performs this implicit sort query at the beginning of the generated plan to avoid redundant implicit sorts in multiway join queries which incur large intermediate data and long shuffling time. The workflow of J-MOTH Implicit Sort Exploiter is as follows:

(1) Identify the join implicit sort, *JIS*, for each table in each multiway join based on the attribute within join condition, such as $\mathrm{JIS}(t_i, a_j)$. For instance, based on the JOMR technique, the join implicit sorts of the tables within the ordered TPC-H *Q10* (i.e., *customer*, *orders*, *lineitem*, and *nation* tables) are as follows:

$$\begin{aligned}&\mathbf{JIS}\left(\text{customer}, c_{\text{custkey}}\right)\mathbf{JIS}\left(\text{customer}, c_{\text{nationkey}}\right),\\ &\mathbf{JIS}\left(\text{orders}, o_{\text{custkey}}\right)\mathbf{JIS}\left(\text{orders}, o_{\text{orderkey}}\right),\\ &\mathbf{JIS}\left(\text{nation}, n_{\text{nationkey}}\right),\ \mathbf{JIS}\left(\text{lineitem}, l_{\text{orderkey}}\right).\end{aligned} \tag{15}$$

(2) Find the shared *JISSO*(s) within the multiway join from the identified *JIS*(s).

(3) Estimate the cost of the identified *JISSO*(s) which is defined as a summation of read, shuffling, and write costs using the following equation [18, 31, 39, 58]:

$$\text{Cost}_{\text{JISSO}} = \text{Cost}_{\text{Read}} + \text{Cost}_{\text{shuffel}} + \text{Cost}_{\text{Write}}. \tag{16}$$

(4) The Implicit Sort Exploiter module finds the largest *JISSO* which is termedLargestImplicitSort$_{\text{Join}}$, *LISJ* per each shared join queries' groups to minimize the overall shuffling cost for input multiple multiway joins according to the following equation:

$$\text{LISJ} = \underset{\text{cost}}{\text{argmax}}\left(\text{JISSO}_1, \text{JISSO}_2, \ldots, \text{JISSO}_k\right), \tag{17}$$

where k is the number of *JISSO*s within each group.

(5) The Exploiter represents the Final Implicit Sort vector which holds a set of sort queries (i.e., ORDER BY):

$$\text{FinalImpilictSorts} = \left[\text{LISJ}_{\text{SJG}_1}, \text{LISJ}_{\text{SJG}_2}, \ldots, \text{LISJ}_{\text{SJG}_l}\right], \quad 1 \le l, \tag{18}$$

where l is the number of Shared Join queries Groups, *SJG*. Ultimately, the queries of the *Final Implicit Sort* vector which will be performed before join queries are submitted to the J-MOTH Query Rewriter to generate the final execution plan.

Figure 12 depicts three shared multiway join queries which are ordered by JOMR technique, *Q1*, *Q2*, and *Q3*. It is noted that the join implicit sort shared opportunities are *JIS(B)* and *JIS(c)*, and the largest join implicit sort shared opportunity is *JIS(C)* because of its order. Therefore, the J-MOTH Implicit Sort Exploiter assigns the *JIS(C)* to the *Final Implicit Sort* vector. Accordingly, the Implicit Sort Exploiter finds the largest join implicit sort opportunity based on JOMR ordering and it utilizes the shared ORDER BY queries by performing them only once in the start of multiquery execution plans.

(2) Simultaneous Pipelined Multiway Join (SP). Regarding the Hadoop MapReduce platform, many complex computations which typically involve multiple jobs cannot be expressed as a single MapReduce job [59]. Therefore, the pipeline technique can deliver data to downstream operators more promptly which raises the degree of parallelism, improves utilization, and reduces response time [45]. The SP technique is previously introduced in QPipe which can efficiently evaluate query execution plans produced by a multiquery optimizer [45]. For MapReduce-based query processing, the pipelined query execution can efficiently evaluate the query plans to ensure low query response time such as Pig and Hive [18, 60]. According to the work in this paper, the SP technique is invoked inside the proposed J-MOTH multiway join optimizer to exploit the overlapped subplans (i.e., two-way joins) among input queries [27]. It detects the common subplans among concurrent queries and evaluates only single query node; then, it pipelines the shared intermediate results to multiple nodes at the same time. J-MOTH Multiway Join Reused-based Estimator in SP: as stated in the proposed J-MOTH system for multiway join considering the JOMR technique and the coarse-grained reused-based opportunities of two-way join, ROTWJs, there are multiple parent queries whose joined result can be reused and pipelined to any Q_i. Using the SP technique, the J-MOTH multiway join Reused-based Estimator is implemented as a cost model. In particular, the J-MOTH optimizer estimates the cost of all identified reused-based opportunities of multiway join and ROMWJ_{Q_i}(s) of Q_i included in the parent queries. Therefore, the cost of each reused-based opportunity is calculated by adding read, shuffle, and write costs using equation (19) [18, 31, 39, 58]:

$$\text{Cost}_{\text{ROMWJ}} = \text{Cost}_{\text{Read}} + \text{Cost}_{\text{Shuffle}} + \text{Cost}_{\text{Write}}. \tag{19}$$

J-MOTH Multiway Join Reused-based Enumerator in SP. After that, the J-MOTH Reused-based Enumerator selects the proper ParentQuery, which has the maximum ROMWJ_{Q_i}. The selected ROMWJ_{Q_i} has the largest number of ordered overlapped two-way joins as described below:

$$\mathrm{ROMWJ}_{Q_i} = \mathrm{argmax}\left(\mathrm{ROMWJ}_1, \mathrm{ROMWJ}_2, \ldots, \mathrm{ROMWJ}_k\right), \quad 1 < k < n, \tag{20}$$

where k is the number of the candidate ParentQuery(s) which includes ROMWJs.

Example 4. To clarify how the J-MOTH exploits reused-based opportunities of multiway joins using SP technique, consider five queries, *Q1*, *Q2, Q3, Q4,* and *Q5*. The estimated reused-based opportunities multiway joins, ROTWJusing the Reused-based Estimator submodule, and the selected best ParentQuery(s) using Reused-based Enumerator submodule are presented in Table 3.

For example, the estimated reused-based opportunity of *(Q4, Q2)* is 2 *ROTWJs*. This indicates that *Q4* can reuse two executed two-way join within *Q2*, such as (*T1*⋈*T2*) and ((*T1*⋈*T2*)⋈*T3*). On the contract, the estimated reused-based opportunity of (*Q4, Q3*) is 1. *ROTWJ* indicating *Q4* reuses one executed two-way join within *Q1* such as (*T1*⋈*T2*). Thus, the Reused-based Enumerator module selects *Q2* as a ParentQuery of *Q4* because *Q2* has larger number of two-way join opportunities, which optimizes *Q4* by reducing the execution time for (*T1*⋈*T2*) join. Consequently, the J-MOTH system pipelines the results of *Q2* such as (*T1*⋈*T2*⋈*T3*); then, it performs only one join such as (*Q2*⋈*T4*) instead of performing((*Q2*⋈*T3*)⋈*T4*) in the case of selecting *Q3* as a parent query of *Q4* which takes long execution time caused by large shuffled intermediate data. The J-MOTH multiway join execution plan using SP technique is depicted in Figure 13.

5.3.4. Phase 4: Rewriting Query. The J-MOTH Query Rewriting unites all optimized queries based on exploiting sharing data including predicates and sharing work including implicit sorts and multiway joins. Accordingly, the final execution plan has three types of queries, which are implicit sorts, predicates, and pipelined join queries, all of them executed in two stages (see Figure 14).

During the execution stages of the optimized batch queries, the intermediate results can be shared across multiple queries. For the first stage, the shared predicates and implicit sort queries are executed concurrently. For the second execution stage of the multiway join queries, the intermediate results of the shared predicates and implicit sorts can be shared and materialized across multiple join queries. Furthermore, the joined results of the parent queries can be reused and pipelined to their subjoin queries simultaneously. Therefore, it is noted that each multiway join query is divided into sets of pipelined two-way joins which are sequentially executed over the large dataset. Finally, the optimized J-MOTH plan of multiple multiway join queries are submitted back to Big Data platform to be executed; the results then are written back to HDFS.

Interestingly, to our knowledge, Apache Flink platform does not support multiway join. Through this form of the final execution plan, the shared multiple multiway join queries are supported and efficiently executed on Apache Flink. Therefore, the proposed J-MOTH system is considered the first concrete system that could be integrated into Apache Flink platform to execute optimized multiple shared multiway join queries.

6. Experimental Evaluation

The experiments have been conducted using Hadoop version 2.6.0, and Flink version 1.4.0 on a cluster of 10 nodes configured with a 3 GB of RAM, 2 cores, and 200 GB disk running over Ubuntu Linux 14.04.4 LTS. Moreover, MapReduce Hive version 2.1.1 has been installed on the Hadoop Namenode and Datanodes. In addition, the Hadoop configuration parameter, *io.mapred.task.io.sort.mb*, was tuned between 300 MB and 2 GB based on the number of tuples.

The performance of the proposed J-MOTH system is evaluated using two-way join and multiway join queries.

6.1. Performance Evaluation of Multiple Two-Way Join Queries. The two multiquery execution plans which have been evaluated and compared are (1) the NT plan which runs the queries independently, (2) the J-MOTH plans which depend on the coarse-grained reused-based opportunities. Multiple versions of subset two-way joins of TPC-H Q1, Q3, Q5, Q8, and Q10 are used to evaluate the J-MOTH two-way join query optimizer. Furthermore, to assess the performance of J-MOTH, its total runtime (i.e., runtime of both the sharing data and join queries), MapReduce phases' time, and intermediate data size have been measured under a set of varied data size using 120, 600, and 1200 GBs.

6.1.1. Evaluation of Two-Way Join Strategies in Apache Flink. Apache Flink runtime can execute join queries in various strategies which significantly impacts the performance. It tries to pick a reasonable join strategy automatically, but there is the ability to manually pick a strategy in case of enforcing a specific strategy of join query execution according to the experimental environment specification [61]. Therefore, extra experiments are manually conducted to evaluate the join strategies using TPC-H benchmark running on Apache Flink batch processing. The cheapest join strategy is picked and applied to optimize each two-way join query to improve the overall performance of multiple two-way join queries.

Consequently, the six join strategies of Flink are evaluated using the NT plan and J-MOTH plans to assess the cheapest join strategy (see Table 4). A trivial two-way join query between *Orders* and *Lineitem* tables (i.e., shared selection predicates in *Lineitem*) is evaluated to perform the sharing data, which proves the superiority of the J-MOTH system in the conducted experiments. Table 3 depicts the

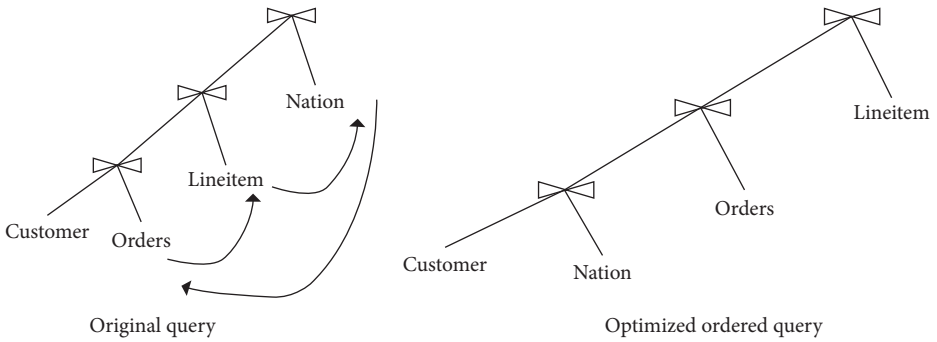

Figure 11: JOMR technique example using TPC-H Q10.

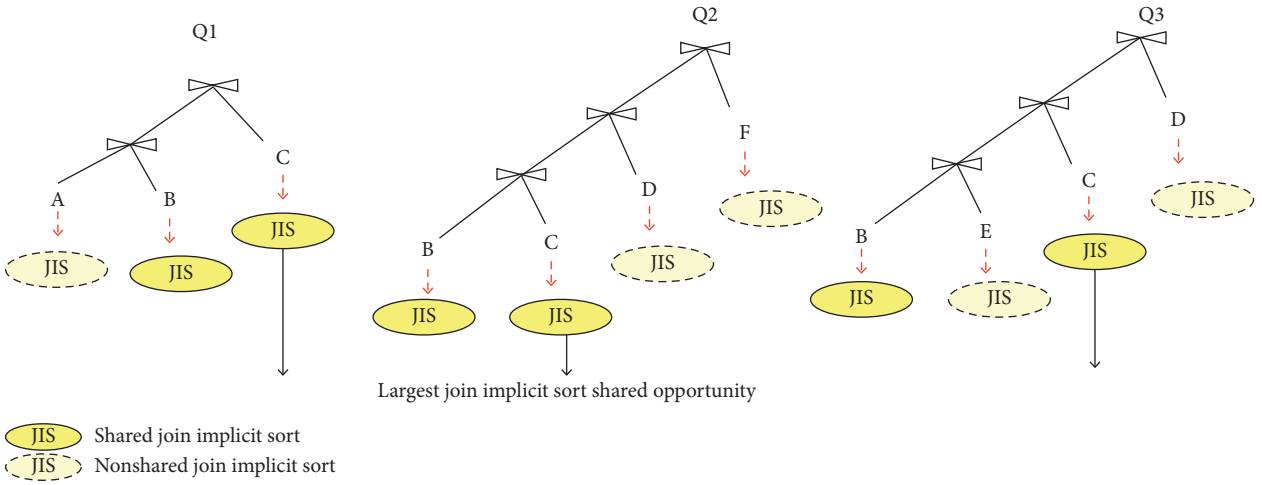

Figure 12: The J-MOTH system Exploiter Example.

Table 3: Example of J-Moth System using SP Technique.

Query No.	Query	Candidate parent query					Estimated reused-based opportunities multiway joins, *ROMWJ*
		Q1	*Q2*	*Q3*	*Q4*	*Q5*	
Q1	*T1⋈T2⋈T4*		0	1	0	0	*Q3*
Q2	*T1⋈T2⋈T3*	0		1	0	0	*Q3*
Q3	*T1⋈T2*	0	0		0	0	–
Q4	*T1⋈T2⋈T3⋈T4*	0	2	1		0	*Q2*
Q5	*T1⋈T2⋈T3⋈T5*	0	2	1	2		*Q2*

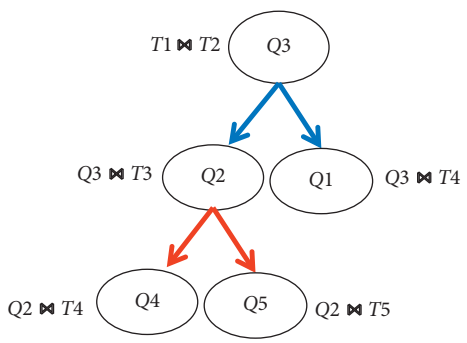

Figure 13: J-MOTH multiway join execution plan using SP technique.

query execution time and the relative improvements for Flink six join strategies.

Also, it is noticed that the REPARTITION_HASH_FIRST join strategy outperforms the other strategies by 28% in average with respect to NT plan. This is because the two input files (i.e., *Orders* and *Lineitem* tables) are large and the Flink runtime builds a hash table from the first input, while the proposed J-MOTH system reduces the shuffled data of the second input, *Lineitem*, by exploiting the shared predicates.

6.1.2. Effect of Multiple Two-Way Join Queries. A series of experiments are conducted to compare the overall

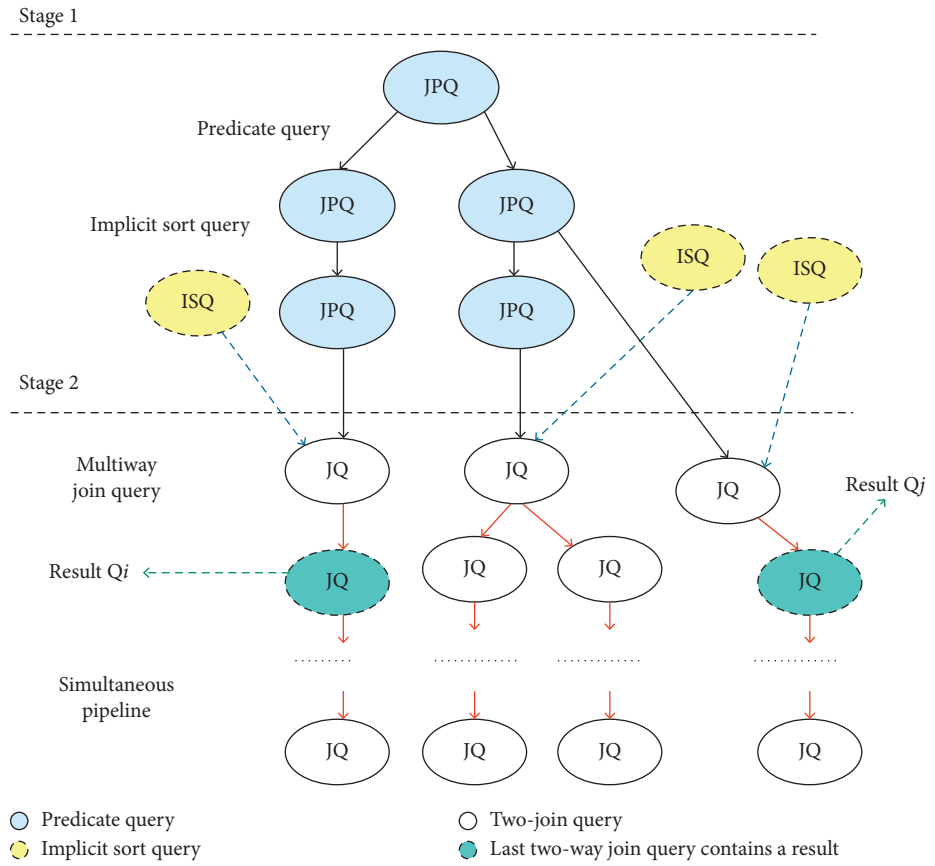

Figure 14: The J-MOTH multiway join optimized plan.

Table 4: The J-Moth Execution Time and Improvement Comparison using Flink Join Strategies.

Strategy	NT execution time (sec)	J-MOTH execution time (sec)	Improvement of J-MOTH wrt NT (%)
BROADCAST_HASH_FIRST	1703	1353	21
BROADCAST_HASH_SECOND	1904	1403	26
REPARTITION_HASH_FIRST	1453	1052	28
REPARTITION_HASH_SECOND	1553	1152	26
REPARTITION_SORT_MERGE	1904	1453	24
OPTIMIZER_CHOOSES	1603	1202	25

performance of the proposed J-MOTH system for multiple two-way join queries using multiple versions of TPC-H queries. The implementation results with respect to the multiquery execution time and its improvement on Hadoop MapReduce and Flink are presented in Figure 15. Accordingly, it is found that the J-MOTH plans using fine-grained and coarse-grained techniques (i.e., FG-T and CG-T) outperform the NT because the NT plan consumes long time for scanning large input files multiple times. In particular, the relative query execution time improvements of FG-T and CG-T plan with respect to NT plan are 14%, 25 and 19%, and 32% for Hadoop MapReduce and Flink, respectively. We attribute these improvements to the exploiting of the shared big selection predicates among multiple two-way join queries which are significantly useful for querying over large datasets. The superiority of Apache Flink improvement manifests in exploiting the coarse-grained reused-based selection predicates besides the cheapest join strategy, REPARTITION_HASH_FIRST.

6.1.3. Average Time of MapReduce Phases. For more investigation, the average times of Hadoop MapReduce phases, Map and Shuffle, which execute the multiple two-way join queries, have been calculated to determine the improvement of data in-network movement time. Table 5 presents the experimental results for the average time of both Map and Shuffle phases, and their improvements. It is noticed that the NT plan affects Map and Shuffle execution time relative to FG-T and CG-T plans. The reason behind this is that the proposed J-MOTH system reduces the data size which is loaded and shuffled by Mapper and Reducer through exploiting the sharing data opportunities (i.e., the shared predicates) among multiple two-way join queries.

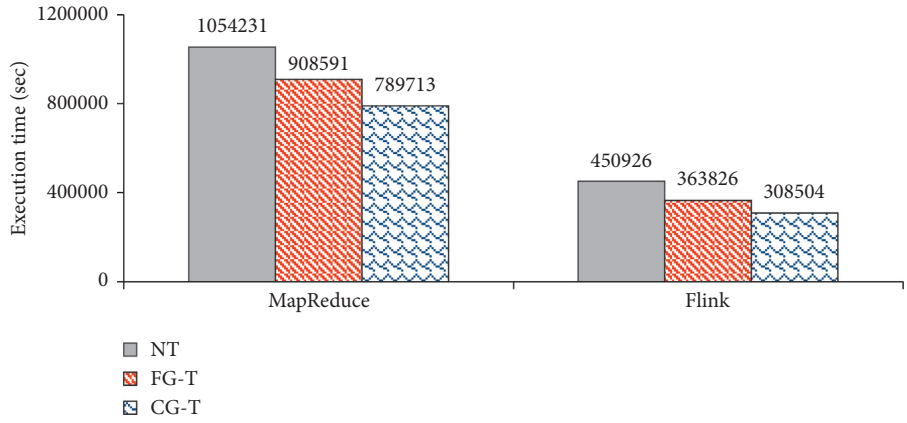

Figure 15: The query execution time two-way join optimizer.

Table 5: The Average time of MapReduce phases for two-way join and their improvement w. r. t NT plan.

Phase	Average time (sec)			Improvement wrt NT	
	NT	FG-T	CG-T	FG-T	CG-T
Map	351	223	184	36%	48%
Shuffle	317	198	134	38%	58%

6.1.4. Effect of Intermediate Data Size. Another performance evaluation metric of the J-MOTH system for multiple two-way join queries is the intermediate data size which is transferred over the network from the machines running the Map tasks to the machines running the Reduce tasks. Table 6 shows the actual intermediate data in terms of GBs number which are shuffled across machines from Mapper to Reducer phase. Practically, the NT plan has the largest intermediate data size because there is no exploiting of the sharing data opportunities among input multiquery. Significantly, the relative intermediate data size improvement of FG-T and CG-T plans with respect to NT plan are 14% and 17% in average, respectively, which can be very meaningful to emphasize the insightful performance regarding exploiting coarse-grained shared predicates among multiple two-way join queries in such large dataset. Therefore, these experiments have demonstrated the robustness of the proposed J-MOTH which is capable of gaining benefits by exploiting sharing data in two-way join queries to avoid redundant big selection predicates, as well as data in-network movement time.

6.2. Performance Evaluation of Multiple Multiway Join Queries. A series of experiments have been conducted to evaluate J-MOTH multiway join execution plans relative to the NT plan which runs the queries independently. As stated before, the Flink does not support multiway join queries, so we have divided each multiway join query into a set of two-way join queries sequences without sharing consideration. For example, in case the input query is $Q = T_1 \bowtie T2 \bowtie T3$, it could be divided into two queries as $Q_1 = T_1 \bowtie T3$ and $Q_2 = Q_1 \bowtie T3$. As discussed earlier, the J-MOTH multiway join optimizer is multiphase optimization which consists of exploiting data granularity (i.e., fine and coarse), sharing implicit sort, sharing pipelined join, sharing ordering pipelined join, and finally leveraging both coarse-grained sharing data and all sharing work phases. Consequently, these phases are categorized into six techniques (i.e., FGSD-T, CGSD-T, SIS-T, SSP-T, O-SSP-T, and CG-SIS-O-SSP-T) as presented in Table 7. Besides, multiple versions of TPC-H $Q1$, $Q3$, $Q5$, $Q8$, and $Q10$ are used to evaluate the proposed J-MOTH multiway join query optimizer against 2 Terabytes.

Moreover, to assess the performance of the J-MOTH multiway join optimizer, the total execution time of multiple multiway join queries and intermediate data size have been measured. Regarding the evaluation of the proposed J-MOTH multiway join optimizer over Flink, the REPARTITION_HASH_FIRST Flink strategy (i.e., it is investigated by J-MOTH two-way join optimizer in the previous section) is selected to execute multiway join queries.

6.2.1. Effect of Multiple Multiway Join Queries. The implementation results with respect to the execution time on Hadoop MapReduce and Flink platforms are presented in Figure 16. Mostly, the multiquery execution time of NT plan is greater than each of J-MOTH generated plans because of the needed time to scan and shuffle large input files multiple times. On the opposite, the multiquery execution time of the J-MOTH plans gradually improves over both Hadoop MapReduce and Flink using sharing data techniques, FGSD-T and CGSD-T, followed by sharing work techniques, SIS-T, SSP-T, O-SSP-T, and finally amalgamating sharing data and sharing work, CG-SIS-O-SSP-T technique. This is because exploiting sharing data besides sharing work can reduce the prolonged intermediate data in-network movement to optimize the execution time of multiway join queries.

Table 6: Total intermediate data size of shared two-way join queries in Hadoop MapReduce.

Intermediate data size (GB)			Improvement wrt NT	
NT	FG-T	CG-T	FG-T	CG-T
82	72	68	14%	17%

Table 7: The J-MOTH multiway join optimization techniques.

No	Sharing type	Technique name	Abbreviation
1	No sharing	Naïve	NT
2	Sharing data	Fine-grained sharing data	FGSD-T
3		Coarse-grained sharing data	CGSD-T
4	Sharing work	Shared implicit sort	SIS-T
5		Shared simultaneous pipeline	SSP-T
6		Ordering-shared simultaneous pipeline	O-SSP-T
7	Sharing data and sharing work	Coarse-grained shared implicit sort- ordering-shared simultaneous pipeline	CG-SIS-O-SSP-T

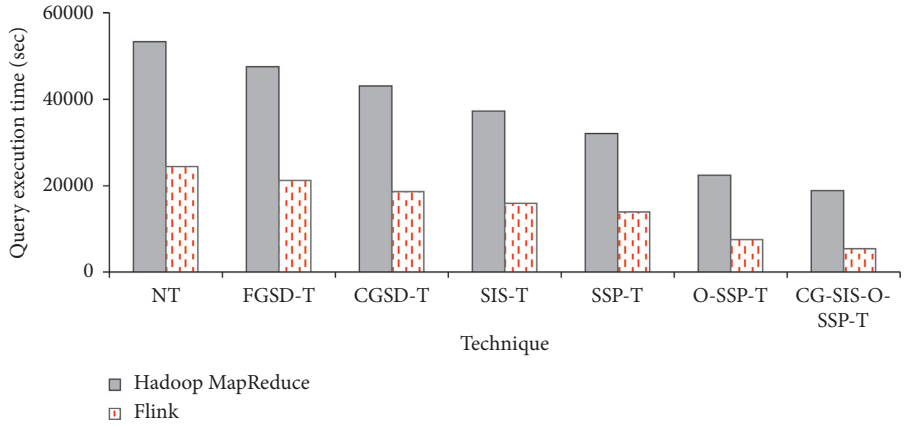

Figure 16: The query execution time of the J-MOTH multiway join optimizer techniques.

It is noticed that the superiority of multiway join queries' execution time has been satisfied over Apache Flink for all conducted experiments with respect to MapReduce because of two reasons: (1) enforcing Flink join strategy to RE-PARTITION-HASH-FIRST strategy for each single two-way join query which is investigated in J-MOTH two-way join optimizer according to our environmental setup and (2) Flink is an in-memory data processing platform which is faster than the in-disk processing (i.e., Hadoop MapReduce).

To study how effectively the proposed J-MOTH system improves multiple multiway join queries relative to NT plan, we have started with sharing data techniques (i.e., FGSD-T and CGSD-T), which achieve 18%, 9% and 21, and 18% for Hadoop MapReduce and Flink, respectively (see Figure 17).

Afterwards, we have used SIS-T technique which exploits sharing implicit sorts. The proposed J-MOTH system accomplishes execution time improvement by 30% and 25% on average for Hadoop MapReduce and Flink, respectively. This means that the duplicate large implicit sorts of the bulk of the tables within multiway join queries have already been avoided. Thus, the J-MOTH system provides more benefits for multiple multiway join queries. Still, there is more possibility for exploiting sharing work opportunities to accelerate multiquery execution time. Therefore, exploiting shared pipelined two-way joins using SSP-T technique improves execution time by 40% and 43% for Hadoop MapReduce and Flink, respectively. Moreover, exploiting shared pipelined two-way joins with ordering consideration using O-SSP-T technique improves the multiway joins execution time by 58% and 69% in average for Hadoop MapReduce and Flink, respectively. Note that the improvements of the execution time using CG-IS-O-SSP-T technique which significantly fulfils the two-fold of the proposed J-MOTH optimizer (i.e., sharing data and sharing work) are 65% and 78% in average for Hadoop MapReduce and Flink, respectively. With more sharing investigation, the improvement of the J-MOTH multiway joins is gradually increased moving from 17% to 46% then 71% for using exploiting sharing data, sharing work, and using both of them, respectively. Consequently, by amalgamating exploiting sharing data and sharing work using our proposed J-MOTH system insightful performance can be gained among multiple multiway join queries over Big Data.

6.2.2. Effect of Intermediate Data Size. The intermediate data size which is generated during MapReduce phases is evaluated using the J-MOTH system for multiple multiway join

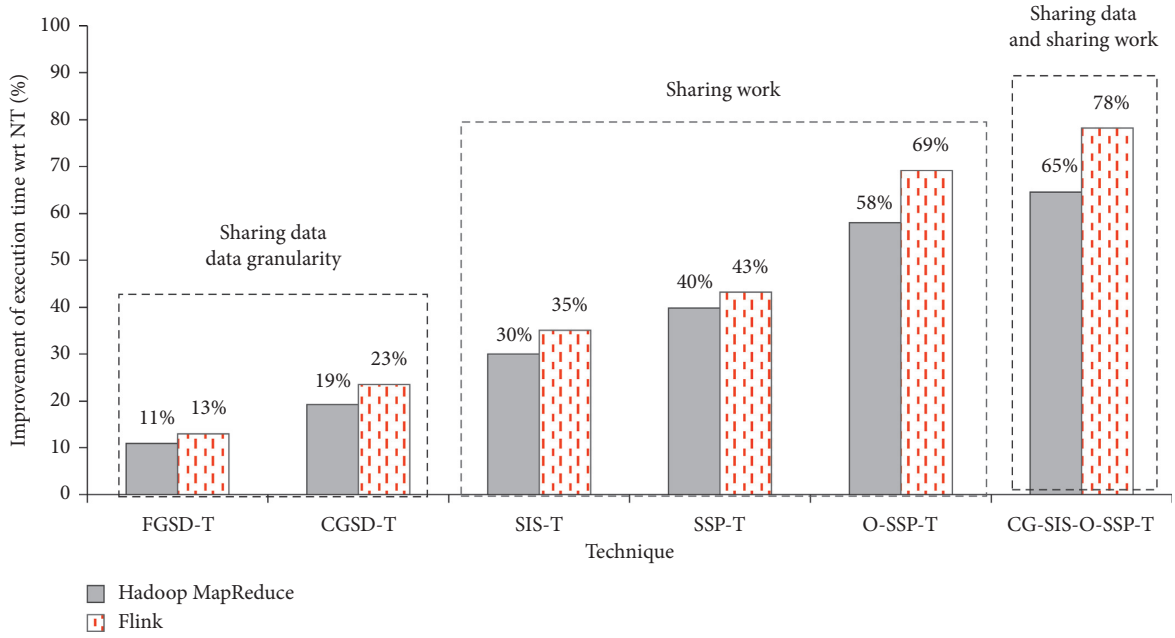

Figure 17: The improvement of query execution time for the J-MOTH multiway join optimizer phases with respect to the NT technique.

Table 8: The intermediate data improvement of the J-Moth multiway join optimization phases.

Technique	NT	FGSD-T	CGSD-T	SIS-T	SSP-T	O-SSP-T	CG-O-SSP-T
Intermediate data size (GB)	478	398	385	366	360	304	279
Improvement with respect to NT	17%	19%	23%	25%	36%	42%	

queries. Table 8 shows the actual intermediate data in terms of shuffled GBs number. Conventionally, the NT plan has the largest intermediate data size because of not exploiting sharing opportunities among input multiquery. Significantly, the improvements of J-MOTH plans relative to NT plan can be very meaningful to emphasize the superior performance regarding exploiting sharing data and sharing work among multiple multiway join in such large dataset.

According to the experimental results in Table 7, it has been found that the generated J-MOTH plans outperform NT plan. Furthermore, the reduction improvement using the J-MOTH plans is increased gradually through more investigation of additional sharing work opportunities such as implicit sorts and shared ordered pipelined two-way joins.

7. Conclusion

The proposed J-MOTH multiway join optimizer is considered as an end-to-end multiway join optimizer strategy on Apache Flink. It exploits shared data, shared joins, and shared implicit sorts within join queries. The proposed J-MOTH system consists of two additional modules to investigate the sharing join. These two additional modules are Query Explorer and Sort Exploiter which integrate with the MOTH system to fulfil the exploiting of coarse-grained sharing data and sharing joins. Regarding the experimental evaluation, it is found that our proposed J-MOTH system outperforms the naive and state-of-the-art techniques (i.e., Relaxed MRShare which is named fine-grained technique). As a future work, we plan to extend the proposed J-MOTH system to address the interactive optimizer which aims to exploit sharing join over data stream-processing (i.e., on-the-fly sharing flavor).

Data Availability

The used data are well-known TPC-H benchmark which is available at http://www.tpc.org/tpch/#:~:text=The%20TPC%20Benchmark%E2%84%A2H,have%20broad%20industry%2Dwide%20relevance.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the Yulin University-Industry Collaboration Project (no. 2019-75-3) and partly supported by the Applied Basic Research Program Funded by Qinghai Province (no. 2019-ZJ-7078).

References

[1] M. N. I. Sarker, M. Wu, and M. A. Hossin, "Smart governance through bigdata: digital transformation of public agencies," in *Proceedings of the 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pp. 62–70, Chengdu, China, 2018.

[2] C. C. Sekhar and C. Sekhar, "Productivity improvement in agriculture sector using big data tools," in *Proceedings of the 2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pp. 169–172, Visakhapatnam, India, 2017.

[3] A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "A cloud-based system for improving retention marketing loyalty programs in industry 4.0: a study on big data storage implications," *IEEE Access*, vol. 6, pp. 5485–5492, 2018.

[4] J. Wan, J. Li, Q. Hua, A. Celesti, and Z. Wang, "Intelligent equipment design assisted by Cognitive Internet of Things and industrial big data," *Neural Computing and Applications*, vol. 32, 2018.

[5] G. Zhang, J. Wang, W. Huang et al., "Big data collection and analysis framework research for public digital culture sharing service," in *Proceedings of the 2015 IEEE International Conference on Multimedia Big Data*, pp. 196–199, Beijing, China, 2015.

[6] L. Carnevale, R. S. Calabrò, A. Celesti et al., "Towards improving robotic-assisted gait training: can big data analysis help us?" *IEEE Internet of Things Journal*, vol. 6, p. 1, 2019.

[7] A. Celesti, M. Fazio, A. Romano, A. Bramanti, P. Bramanti, and M. Villari, "An OAIS-based hospital information system on the Cloud: analysis of a NoSQL column-oriented approach," *IEEE Journal of Biomedical and Health Informatics*, vol. 22, no. 3, pp. 912–918, 2018.

[8] R. Akerkar, *Big Data Computing*, CRC Press, Cleveland, OH, USA, 2013.

[9] A. Gkoulalas-Divanis and A. Labbi, *Large-Scale Data Analytics*, Springer, Geneva, Switzerland, 2016.

[10] J. Dean and S. Ghemawat, "MapReduce," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] S. Babu and H. Herodotou, "Massively parallel databases and MapReduce systems," *Foundations and Trends in Databases*, vol. 5, pp. 1–104, 2013.

[12] N. Spangenberg, M. Roth, and B. Franczyk, "Evaluating new approaches of big data analytics frameworks," in *Proceedings of the International Conference on Business Information Systems*, pp. 28–37, Poznan, Poland, 2015.

[13] C. Eiras-Franco, V. Bolón-Canedo, S. Ramos, J. González-Domínguez, A. Alonso-Betanzos, and J. Touriño, "Multithreaded and Spark parallelization of feature selection filters," *Journal of Computational Science*, vol. 17, no. 3, pp. 609–619, 2016.

[14] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems*, vol. 13, no. 1, pp. 23–52, 1988.

[15] J. Park and A. Segev, "Using common subexpressions to optimize multiple queries," in *Proceedings of the 4th IEEE International Conference on Data Engineering*, pp. 311–319, Chicago, IL, USA, 1988.

[16] G. Papakonstantinou and J. Kontos, "A query-oriented file organization technique," *International Journal of Systems Science*, vol. 5, no. 8, pp. 743–751, 1974.

[17] Z. Ji, Y. Ma, Y. Pang, and X. Li, "Query-aware sparse coding for web multi-video summarization," *Information Sciences*, vol. 478, pp. 152–166, 2019.

[18] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "MRShare," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 494–505, 2010.

[19] I. Elghandour and A. Aboulnaga, "ReStore," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 586–597, 2012.

[20] C. Ming-Syan, P. S. Yu, and W. Kun-Lung, "Optimization of parallel execution for multi-join queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, pp. 416–428, 1996.

[21] R. Sahal, M. H. Khafagy, and F. A. Omara, "Exploiting coarse-grained reused-based opportunities in big data multi-query optimization," *Journal of Computational Science*, vol. 26, pp. 432–452, 2018.

[22] R. Sahal, M. H. Khafagy, and F. A. Omara, "SOOM: sort-based optimizer for big data multi-query," *Big Data*, vol. 8, no. 1, pp. 38–61, 2020.

[23] A. Bechini, F. Marcelloni, and A. Segatori, "A MapReduce solution for associative classification of big data," *Information Sciences*, vol. 332, pp. 33–55, 2016.

[24] H. S. A. Azez, M. H. Khafagy, and F. A. Omara, "JOUM: an indexing methodology for improving join in hive star schema," *International Journal of Scientific & Engineering Research*, vol. 6, pp. 111–119, 2015.

[25] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using MapReduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1184–1195, 2012.

[26] R. Sahal, M. Nihad, M. H. Khafagy, and F. A. Omara, "iHOME: index-based JOIN query optimization for limited big data storage," *Journal of Grid Computing*, vol. 16, no. 2, pp. 345–380, 2018.

[27] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki, "Sharing data and work across concurrent analytical queries," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 637–648, 2013.

[28] R. Sahal, J. G. Breslin, and M. I. Ali, "Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case," *Journal of Manufacturing Systems*, vol. 54, pp. 138–151, 2020/01/01/ 2020.

[29] R. Sahal, J. G. Breslin, and M. I. Ali, "On evaluating the impact of changes in IoT data streams rate over query window configurations," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, Darmstadt, Germany, 2019.

[30] R. Sahal, M. H. Khafagy, and F. A. Omara, "Big data multi-query optimisation with Apache Flink," *International Journal of Web Engineering and Technology*, vol. 13, no. 1, pp. 78–97, 2018.

[31] G. Wang and C.-Y. Chan, "Multi-query optimization in mapreduce framework," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 145–156, 2013.

[32] R. Sahal, M. H. Khafagy, and F. A. Omara, "Comparative study of multi-query optimization techniques using shared predicate-based for big data," *International Journal of Grid and Distributed Computing*, vol. 9, no. 5, pp. 229–240, 2016.

[33] N. Dalvi, C. Re, and D. Suciu, "Queries and materialized views on probabilistic databases," *Journal of Computer and System Sciences*, vol. 77, no. 3, pp. 473–490, 2011.

[34] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "Opportunistic physical design for big data analytics," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 851–862, Houston, TX, USA, 2014.

[35] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "MISO: souping up big data query processing with a multistore system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1591–1602, Houston, TX, USA, 2014.

[36] A. Jindal, K. Karanasos, S. Rao, and H. Patel, "Selecting subexpressions to materialize at datacenter scale," *Proceedings of the VLDB Endowment*, vol. 11, no. 7, pp. 800–812, 2018.

[37] D. Van Hieu, S. Smanchat, and P. Meesad, "MapReduce join strategies for key-value storage," in *Proceedings of the 11th*

International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 164–169, Chonburi, Thailand, 2014.

[38] L. H. Yang, Y. Zhao, Y. Fan, Y. Zhu, and J. Yu, "Peak power modeling for join algorithms in DBMS," *Journal of Computer and System Sciences*, vol. 81, no. 3, pp. 599–614, 2015.

[39] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 12, Santa Cruz, CA, USA, 2011.

[40] M. Yue, H. Gao, S. Shi, and H. Wang, "Join query processing in data quality management," in *Database Systems for Advanced Applications*, pp. 329–342, Springer International Publishing, New York, NY, USA, 2016.

[41] H. S. Abdel Azez, M. H. Khafagy, and F. A. Omara, "Optimizing join in HIVE star schema using key/facts indexing," pp. 1–12, 2017, IETE Technical Report.

[42] A. Kunft, A. Katsifodimos, S. Schelter, T. Rabl, and V. Markl, "BlockJoin: efficient matrix partitioning through joins," *PVLDB*, vol. 10, 2017.

[43] S. S. Mina Shanoda and K. Mohamed, "JOMR: multi-join optimizer technique to enhance map-reduce job," in *Proceedings of the 9th International Conference on Informatics and Systems (INFOS)*, pp. 80–87, Cairo, Egypt, 2014.

[44] M. H. Mohamed and M. H. Khafagy, "Hash semi cascade join for joining multi-way map reduce," in *Proceedings of the SAI Intelligent Systems Conference (IntelliSys)*, pp. 355–361, London, UK, 2015.

[45] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: a simultaneously pipelined relational query engine," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 383–394, Houston, TX, USA, 2005.

[46] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan, "Pipelining in multi-query optimization," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 728–762, 2003.

[47] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Sharing across multiple MapReduce jobs," *ACM Transactions on Database Systems*, vol. 39, no. 2, pp. 1–46, 2014.

[48] P. Michiardi, D. Carra, and S. Migliorini, *In-Memory Caching For Multi-Query Optimization Of Data-Intensive Scalable Computing Workloads*, EDBT/ICDT, Lisbon, Portugal, 2019.

[49] Y.-M. Nam, D. Han, and M.-S. Kim, "A parallel query processing system based on graph-based database partitioning," *Information Sciences*, vol. 480, pp. 237–260, 2019.

[50] J. Ren, L. Liu, F. Liu, and W. Zhou, "An executable specification of map-join-reduce using haskell," *IEEE Access*, vol. 7, pp. 10892–10904, 2019.

[51] M. N. Lu, M. H. Khafagy, and F. A. Omara, "HOME: HiveQL optimization in multi-session environment," in *Proceedings of the 5th European Conference of Computer Science (ECCS14)*, pp. 80–89, Geneva, Switzerland, 2014.

[52] T. Dokeroglu, S. Ozal, M. A. Bayir, M. S. Cinar, and A. Cosar, "Improving the performance of Hadoop Hive by sharing scan and computation tasks," *Journal of Cloud Computing*, vol. 3, pp. 1–11, 2014.

[53] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury, "Reuse-based optimization for Pig Latin," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 2215–2220, Maui, HW, USA, 2016.

[54] H. Liu, D. Xiao, P. Didwania, and M. Y. Eltabakh, "Exploiting soft and hard correlations in big data query optimization," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1005–1016, 2016.

[55] H. Bian, Y. Yan, W. Tao et al., "Wide table layout optimization based on column ordering and duplication," in *Proceedings of ACM International Conference on Management of Data*, pp. 299–314, Portland OR USA, 2017.

[56] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1688–1701, 2018.

[57] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: efficient join query evaluation in a parallel database system," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 63–78, Houston, TX, USA, 2015.

[58] (2015, 21-10-2015). Cost-based optimization in hive-apache hive-apache software foundation. Available: https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive.

[59] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, *MapReduce Online*p. 20, Nsdi, Boston, MA, USA, 2010.

[60] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of MapReduce pipelines," in *Proceedings of 26th IEEE International Conference on Data Engineering (ICDE)*, pp. 681–684, Brisbane, Australia, 2010.

[61] 2017, 16-21-2017). Apache Flink 1.0.3 Documentation: DataSet Transformations. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/batch/dataset_transformations.html#join.