Big data multi-query optimisation with Apache Flink

Radhya Sahal*
Department of Computer Science, Computers and Engineering College, Hodeida University, Yemen and
Department of Computer Science, Faculty of Computers and Information, Cairo University, Egypt
Email: radhya.sahal@grad.fci-cu.edu.eg
*Corresponding author

Mohamed H. Khafagy
Department of Computer Science, Faculty of Computers and Information, Fayoum University, Egypt
Email: mhk00@fayoum.edu.eg

Fatma A. Omara
Department of Computer Science, Faculty of Computers and Information, Cairo University, Egypt
Email: f.omara@fci-cu.edu.eg

Abstract: Big data analytic frameworks, such as MapReduce, Spark and Flink, have recently gained more popularity to process large data. Flink is an open-source Apache-hosted big data analytic framework for processing batch and streaming data. For historical data processing (batch), Flink’s query optimiser is built based on techniques which have been used in the parallel database systems. Flink query optimiser translates the queries into jobs which are repeatedly submitted with similar tasks. Therefore, exploiting the similarity of tasks can avoid redundant computation. In this paper, Flink multi-query optimisation system, Flink-MQO, has been proposed and built on top of Flink software stack. It is considered as an add-on to Apache Flink to optimise multi-query based on data sharing. The Flink-MQO system exploits the data sharing opportunities of selection operators to eliminate the redundancy and duplication of data in-network movement of multi-query. Experimental results show that the exploiting of shared selection operators in big data multi-query can provide promising query execution time. Therefore, Flink-MQO system can potentially be used in the stream processing to improve the performance of the real-time applications.

Keywords: big data; Flink; batch processing; multi-query optimisation; MQO; sharing opportunity; selection predicates filter.
1 Introduction

Big data has been spread rapidly in many domains such as information systems. On the other hand, the distributed computing is growing every day with the increasing of workstations power and dataset size. Therefore, the development and implementation of the distributed systems for big data applications are considered challenge (Akerkar, 2013; Gkoulalas-Divanis and Labbi, 2014; Chen and Zhang, 2014). One of the famous frameworks that have been emerged for big data processing is MapReduce, which is firstly introduced by Google in 2004 (Dean and Ghemawat, 2008). The main concept of MapReduce is to abstract the details of a large cluster of machines to facilitate the computation on large datasets. Recently, much of the research works in academia and industry have proposed new data processing systems such as Flink and Spark to improve the performance of analytical data applications including query optimisation techniques (Babu and Herodotou, 2013; Spangenberg et al., 2015; Eiras-Franco et al., 2016).

Notwithstanding, over the past couple of years, MapReduce has become the dominant computing platform that makes big data easier to be processed, there is no single computing platform could solve the problems of the real-world datasets, or even address every situation. Although the Hadoop ecosystem has grown and matured for several
years, it cannot hide the problems of the MapReduce computing platform like inefficient processing of iterations or affect performance for data scientists (Lee et al., 2012). Thereby, the emerging of the second version of Hadoop (i.e., YARN resource manager) has been introduced to reduce processing speed. In particular, YARN allows alternative computing platforms to be presented such as Apache Spark and Apache Flink. These alternative open-source big data analytical systems utilise memory usage efficiently and avoid the materialisation of intermediate results on disk that is typical of MapReduce jobs (Spangenberg et al., 2015; Park, 2013). Recently, Apache Flink has emerged for processing streaming and batch data. It combines the low latency stream processing with high-throughput batch processing. Briefly, Flink provides three types of APIs; DataSet API, DataStream API and Table API; for batch processing, streaming processing and SQL-like data analysis respectively (Apache Flink, 2016a).

Big data could be classified into different categories based on five aspects; data sources, content format, data stores, data staging and data processing (see Figure 1) (Hashem et al., 2015). The work in this paper concerns about transaction data, structure formatting, key/value data store where the queries are transformed into data stages including a set of jobs run on Flink batch data processing.

Figure 1  Big data classification (see online version for colours)

Source: Hashem et al. (2015)

By growing big data frameworks for interactive analytics, big data query processing has been heavily invested using various optimisation techniques such as parallelisation, indexing, materialisation (i.e., reusing previous work). In particular, the MapReduce-based systems for analytical tasks (i.e., query processing) has some performance limitations for multi-shared queries such as redundant and wasteful processing shared dataset (Nykiel et al., 2010; Elghandour and Aboulnaga, 2012; Wang and Chan, 2013). On the other hand, multi-query optimisation (MQO) is considered an essential keyword of query processing in the database systems which recalled into big data analysis systems such as MapReduce-based systems for query processing (Qian et al., 2015; Dong et al., 2015). Over about two decades, MQO has been extensively
studied and demonstrated to be an effective technique in both RDBMS and MapReduce to identify and exploit the common sub-expressions (CSEs) among queries to improve the overall query evaluation (Park and Segev, 1988; Sellis, 1988).

Figure 2 Methodologies for big data MQO (see online version for colours)

More specifically, big data MQO could be classified into two methodologies; sharing data and sharing work (see Figure 2) (Psaroudakis et al., 2013). The sharing data methodology exploits the shared scan and selection predicates to avoid redundant computation of the same data for the ongoing queries. However, the sharing work methodology exploits the shared join and shared aggregation operators. In the case of join query, a shared join operator could be exploited to avoid repeated long join execution where the shared aggregations avoid calculating a redundant computation such as summation. Regarding the work in this paper, the sharing data MQO methodology is considered (see Figure 2).

Figure 3 Sharing for big data MQO (see online version for colours)

Substantially, the goal of this paper is to investigate the exploiting sharing data (i.e., selection operators) on Flink batch processing to gain significant performance improvement for big data multi-query. Accordingly, big data multi-query execution time
on Flink is improved by avoiding redundant selections (i.e., the answer from a previous query). In particular, the reused-based opportunities among the shared multi-query are exploited to overcome the useless data loading and unnecessary data processing on Flink (Nykiel et al., 2010). Despite its many features of Flink, querying on redundant data is considered wasteful work and even impractical. Therefore, considering the sharing data techniques among big data multi-query becomes essential especially for I/O-intensive applications. In the case of sharing data on Flink, smaller data size could be loaded to optimise multi-query, then the in-memory computing manner could be used for speeding up the shared queries. In accordance, the data sharing techniques reduce the computation time over I/O intensive applications by fetching the desired results with minimum cost (see Figure 3). Ultimately, by decreasing the cost of big data query processing, charges incurred could be reduced monetary while utilising the big data processing infrastructure (Lefevre et al., 2014a, 2014b; Dokeroglu et al., 2014; Liu et al., 2016).

Three reasons have motivated us to study big data MQO on Flink batch processing. Firstly, although the in-memory big data computing (i.e., Spark and Flink) is faster than the in-disk computing (i.e., MapReduce), the memory size is still limited, so decreasing large dataset which needs to be processed is necessary. Secondly, there is no clear-cut winner between Spark and Flink and there is not one big data analytic framework for all data types, sizes and job patterns (Marcu et al., 2016). Thirdly, for historical data processing (batch), Flink’s query optimiser is built based on techniques from parallel database systems such as plan equivalence, cost modelling and interesting property propagation (Carbone et al., 2015). Exploiting the similar tasks can offer the possibilities of reusing the previous results and avoid redundant computations. Therefore, the main goal of this paper is to study big data MQO on Flink with sharing data opportunities consideration. To this end, the Flink-MQO system is a software component which built on top of Flink and implements two existing works; the MRShare and relaxed MRShare techniques (Nykiel et al., 2010; Wang and Chan, 2013). It is considered as an add-on to Apache Flink to optimise multi-query based on data sharing. The Flink-MQO system can exploit the equal and overlap shared predicates operators among big data multi-query. Furthermore, it investigates that Flink batch processing for big data MQO can gain more optimisation using predicate-based filters by reducing redundant filtering tasks.

The rest of this paper is organised as follows. Related work is described in Section 2. Overview of MQO MapReduce-based systems and the predicate-based filter on shared data are introduced in Section 3. The proposed Flink-MQO system is presented in Section 4. The experimental evaluation is presented in Section 5. Finally, the conclusions are presented in Section 6.

2 Related works

In the domain of Hadoop-based systems, several research work efforts have been made for optimising big data analysis, especially MQO. These research work efforts can be broadly classified into two categories; concurrent MQO and non-concurrent MQO. Concurrent MQO is similar to the MQO of the relational databases (Olston et al., 2008a). It tries to find shared parts among multiple queries that include scan, computation, shuffling and so forth to maximise the sharing benefit (Nykiel et al., 2010; Wang and
The non-concurrent MQO resembles the materialised view techniques. It materialises the intermediate and final computation results and uses them to answer the queries (Elghandour and Aboulnaga, 2012).

MRShare is a concurrent sharing framework where I/O cost is dominant (Nykiel et al., 2010). So, it considers the sharing opportunities of scans, map output and map functions sharing. However, the relaxed MRShare relaxes and generalises the overlapped queries to increase sharing opportunities over a single job (Wang and Chan, 2013). According to relaxed MRShare, the shared map input scan and map output have been studied and algorithms have been introduced to select a computation plan for a batch of jobs in the MapReduce context. Also, additional optimisation techniques (i.e., GGT and MT) have been introduced in the relaxed MRShare. Moreover, a comparative study of MRShare and relaxed MRShare techniques using predicate-based filters on MapReduce is introduced in Sahal et al. (2016). According to the comparative results, it is found that the relaxed MRShare technique outperforms MRShare technique with respect to query execution time for shared data regarding predicate-based filters in MapReduce. Furthermore, the comparative study has addressed the MQO regarding predicate-based filters in MapReduce which considered as in-disk computing big data analytical system, while this work provides a broad comparison of MQO using predicate-based filters in Flink which considered as in-memory computing big data analytical system.

ReStore system is one of the non-concurrent sharing systems which built on top of Pig to optimise query evaluation using materialised results (Elghandour and Aboulnaga, 2012). It uses heuristics algorithm to select the suitable materialised results even for the complete or part of the map and reduce the output of each job. The produced materialised output by ReStore system might not be reused in case of the query workloads which are not repeated which causes storage overheads.

A work in Lefevre et al. (2014a, 2014b) considers the reusing results which are stored by exploiting MapReduce intermediate results for failure resilience reasons as materialised views, where semantic user-defined functions (UDF) models based on Hive have been used to enable effectively reuse views where subsequent queries can be evaluated faster. On the other side, a MQO framework, SharedHive, is proposed to transform a set of correlated HiveQL into new optimised queries sets on sharing scan and computation tasks (Dokeroglu et al., 2014). The reused-based optimisation has been addressed for Pig scripts by proposing PigReuse which identifies and reuses CSEs occurring in Pig Latin scripts. Then, it selects the best ones to be merged based on a cost-based search process which is implemented with the help of a linear program solver (Camacho-Rodriguez et al., 2014). Also, the MOTH system is proposed to exploit the coarse granularity of fully and partial reused-based opportunities in big data MQO. The main principle of the proposed MOTH system is to investigate the coarse-grained reused-based opportunities horizontally (i.e., non-equal tuples size) and non-uniform data distribution vertically (i.e., tuples number) using metadata and histogram with slow storage consideration (Sahal et al., 2017).

3 MapReduce-based MQO overview

In this section, the state-of-art MapReduce-based systems for MQO will be introduced and then the predicate-based query operator will be discussed.
3.1 MQO in MapReduce

MapReduce framework has two simple user-friendly interfaces; map and reduce functions. Moreover, MapReduce supports query processing by introducing high-level declarative languages such as Hive and Pig (Thusoo et al., 2009, 2010; Olston et al., 2008b). Regarding this work, multiple queries are translated to MapReduce jobs with similar tasks running over the same dataset (Doulkeridis and Nørvåg, 2014). Conventionally, the similar tasks are executed independently which incur costly redundant processing. Thereby, the sharing and materialisation techniques are proposed to exploit the sharing common parts of queries to eliminate wasteful processing over the same dataset (Alhajj and Polat, 1999). MRShare and relaxed MRShare techniques are the well-known MapReduce-based solutions for big data which will be overviewed as follows.

3.1.1 MRShare

MRShare is the state-of-art work of MQO in MapReduce which considered as a concurrent sharing system that can share portions of identical works to avoid wasteful processing (Nykiel et al., 2010). The key idea of MRShare technique is that the similar jobs are grouped into one job which performs only once (Doulkeridis and Nørvåg, 2014; Nykiel et al., 2010; Wang and Chan, 2013). According to MRShare, three sharing opportunities are considered:

1. **Sharing scans**: scan only once the same map tasks over the same input file.
2. **Sharing map outputs**: performs only one transform tasks for the key-value pairs of the same mapping tasks.
3. **Sharing map functions**: avoid redundancy computation for a batch of queries which executed at the same time.

3.1.2 Relax MRShare

Although MRShare is considered as the state-of-the-art work in MQO MapReduce-based, there is a set of research works have been proposed to extend MRShare. The relaxed MRShare technique has been proposed to relax and generalise MRShare by exploiting the overlapping among queries to increase the sharing opportunities in big data MQO (Wang and Chan, 2013). Relative to the MRShare system, the relaxed MRShare system has introduced additional optimisation techniques (i.e., generalised grouping technique and materialisation technique) to exploit the overlapping among shared jobs. More details regarding MRShare vs. relaxed MRShare are illustrated in Guoping (2014) and Sahal et al. (2016).

3.2 Selection operators on sharing data

The sharing data including selection queries can minimise data filtration significantly for multi-query processing by reducing I/O bottleneck and network traffic over big data infrastructure. According to the context of this paper, the shared predicate-based filters are concerned to be run over Apache Flink. Therefore, these predicates operators can improve the performance by applying filtration tasks only once on the same input files.
A comparative study has been performed in our previous work for predicate-based filters among multi-query and it has confirmed that the relaxed MRShare system significantly outperforms the MRShare for sharing data on MapReduce environment (Sahal et al., 2016). However, this work has studied that an efficient in-memory computing platforms still need more optimisation to reduce data which will be processed to answer big data multi-query.

4 The proposed Flink-MQO system

In this section, an overview of Flink batch processing will be presented. Then, the proposed Flink-MQO system will be introduced.

4.1 Flink batch processing overview

Apache Flink is an Apache-hosted big data analytic framework that implements a universal dataflow engine to perform both stream and batch big data analytics (Carbone et al., 2015). It follows a computing paradigm that embraces in-memory computing for big data analysis. In-memory computing refers to the use of direct memory instead of disks (i.e., MapReduce) for data processing whether stream or batch. Therefore, the application performance can be improved by storing and processing data directly from RAM. Furthermore, the overhead of accessing the disk can be reduced in the case of processing unbounded data stream for real-time analytics. According to this paper, Flink batch processing of MQO over bound static data is addressed. For the data perspective, the bounded data such as the traditional data warehouses is a special case of an unbounded data stream, so the traditional data warehouse systems which processed on the hard disk is shifted to the in-memory computing analytical system such as Flink (Carbone et al., 2015; Shukla et al., 2015). For the processing perspective, the batch processing is considered theoretically as a special case of streaming processing where the stream is finite and the order and time of records do not matter (i.e., all records implicitly belong to one all-encompassing window). In particular, Flink treats the batch processing especially, by optimising their execution using a query optimiser and by implementing blocking operators that gracefully spill to disk in the absence of memory (Mohammed et al., 2016; Sayed and Khafagy, 2015).

Apache Flink provides three types of APIs; DataStream API, DataSet API and Table API, for streaming processing, batch processing and SQL-like data analysis respectively (Apache Flink, 2016a). The DataStream API provides the means to keep recoverable state and partition, transform and aggregate data stream windows. In contrast, DataSet API is added to Flink to support batch processing over static datasets. Furthermore, the Flink batch processing DataSet APIs uses specialised data structures, algorithms and operators like join and grouping and uses dedicated scheduling strategies. Recently, Flink provides an API, namely Table API, which allows specifying operations using SQL-like expressions for relational stream and batch processing. The Flink Table API allows the developers to write their queries which execute over a set of a relational table abstraction. Each relational table can be created from external data sources or existed DataSets and DataStreams where the relational operators such as selection, aggregation and join can be on tables (Apache Flink, 2016b, 2016c).
4.2 The proposed Flink-MQO system

Again here, the goal of this paper is to investigate the exploiting sharing data (i.e., selection operators) on Flink batch processing to gain significant performance improvement for big data multi-query. Thereby, the proposed Flink-MQO can reduce the big redundant filtration tasks in multi-query regarding shared selection predicate operators. Consequently, the Flink-MQO system is designed to behave as ordinary Flink software stack including MRShare and relaxed MRShare techniques (Nykiel et al., 2010; Wang and Chan, 2013). Figure 4 depicts the proposed Flink-MQO system which built on top of the Table API, DataSet API and batch processing environment in Flink software stack. In particular, the proposed Flink-MQO system can optimise the multiple queries over static data with considering the optimised data sharing opportunities of multi-query. The proposed Flink-MQO system consists of two layers; multi-query optimiser and the underlying Apache Flink. The modules of multi-query optimiser layer will be described as follows.

Figure 4 The Flink software stack including the proposed multi-query optimiser, FLINK-MQO (see online version for colours)

4.2.1 Predicates extractor module

The predicates extractor module parses the queries in terms of tokens (i.e., SQL commands). Then, it extracts the pre-defined selection predicates such as BETWEEN operator and AND and OR operators from WHERE clause.
4.2.2 Reused-based multi-query optimiser module

The reused-based multi-query optimiser is considered the backbone module of the proposed Flink-MQO system which targets the opportunity of reused results for MQO. It generates an optimised multi-query execution plan which exploits the maximum shared selection predicates among input multi-query. More specifically, the generated plan schedules the queries in a manner that the shared selection predicates compute once. Then, it saves them temporary to be reused immediately by subsequent shared queries in a batch to reduce the overall query evaluation. Substantially, the MRShare and relaxed MRShare techniques are implemented to optimise shared selection predicates (Nykiel et al., 2010; Wang and Chan, 2013). Beyond this, the proposed Flink-MQO system can be extended easily by implementing new MQO techniques to apply other MQO criteria.

In particular, the reused-based multi-query optimiser receives a set of predicates which converted into a set of plain queries and then outputs a set of ordered queries in terms of DAG.

Firstly, assume that the input queries are specified in a high-level query language which later translated by Flink Table API into batch Dataset API. Each input query is modelled as \((R, A, P)\) where \(R\) is the relation names (i.e., table name(s)), \(A\) is the set of attributes and \(P\) is the selection predicates filter which applied to retrieve the desired output. Consequently, the shared selection predicates (SSP), could be identical or subset which can be defined by the following definition:

**Definition 1:** Shared selection predicates (SSP)

\[
\forall Q_i, Q_j \left( R_i \subseteq R_j \land A_i \subseteq A_j \land P_i \subseteq P_j \right) \rightarrow SSP(Q_i, Q_j)
\]

In the case of two input queries such as \(Q_i, Q_j\), \(SSP(Q_i, Q_j)\) is evaluated only once and \(P_i\) will reuse \(P_j\) then the redundant evaluation of the same selection predicate will be saved. After SSP among pairs of queries is captured, the reused-based multi-query optimiser estimates the reused-based opportunity of SSP according to the pre-defined estimation criteria. According to the work in this paper, the equality and overlapping estimation regarding MRShare and relaxed MRShare techniques are used respectively (Nykiel et al., 2010, Wang and Chan, 2013). Typically, in the relaxed MRShare technique, the smallest overlapped predicate is chosen from the set of candidate queries to be a parent query (i.e., predecessor node).

Without loss of generality, consider the types of input queries over the relations **Employees** (employeeID, name, age, address, dept, salary) listed as follows: (see online version for colours)

Q1 \(\text{SELECT Name, Age, Salary FROM Employees WHERE Age} \leq 40\)

Q2 \(\text{SELECT Name, Dept, Salary FROM Employees WHERE Age} \geq 20 \text{ AND Age} < 30\)

Q3 \(\text{SELECT Name, Dept, Salary FROM Employees WHERE Age} > 30 \text{ AND Age} \leq 35\)

Accordingly, \((Q_1, Q_2, Q_3)\) are represented as \((R_1, A_1, P_1)\), \((R_2, A_2, P_2)\) and \((R_3, A_3, P_3)\) respectively. It should be noted that three queries are overlapped among sharing data
parts in terms of predicates such as \( (P_1, P_2, P_3) \). So, exploiting the shared selection predicates among input multi-query yields significant cost saving over large datasets. Also, reusing filtered data rather than the original files can reduce the I/O operations through the big data infrastructure. Figure 5 depicts an example of MQO in case of shared selection predicates in the proposed Flink-MQO system. These shared selection predicates are extracted, studied according to sharing opportunities using MRShare and relaxed MRShare techniques and then exploited to optimise multi-query.

**Figure 5** An example of shared selection predicate-based filters on shared multi-query (see online version for colours)

![Diagram of MQO](image)

**Figure 6** The generated multi-query plan using shared predicates (see online version for colours)

![Diagram of Multi-Query Plan](image)

Figure 6 illustrates the generated multi-query plan where the qualifying tuples among shared selection predicates are filtered and then fed into final desired results. In other words, the generated multi-query plan uses the shared selection predicates in direct shared queries which represented as a tree. Thereby, the filtered data can be exploited and reused regardless of query type or existent direct data sharing. Therefore, the generated
multi-query execution plan eliminates the redundancy due to accessing the same data multiple times in different queries.

4.2.3 Query rewriter module

It rewrites the final outputs \( n \) queries into two plans; non-concurrent multi-query plan and concurrent multi-query plan (see Figure 6). The non-concurrent multi-query plan holds a list of multiple queries which called \( \text{NonConQueryList} \). The queries in \( \text{NonConQueryList} \) will be executed sequentially based on non-ascending order of the rooted trees, since it starts with root queries followed by set of parent queries (Venetis et al., 2014). On the other hand, the leaf queries which can reuse the results of non-concurrent queries, beside non-shared queries are listed in \( \text{ConQueryList} \). The queries in \( \text{ConQueryList} \) are executed simultaneously in concurrent multi-query plan. Consequently, the multi-query plan is defined as follows.

**Definition 2:** Multi-query plans

\[
\forall Q_i, Q_j \in \text{RootQuery} \lor Q_i \in \text{ParentQuery} \rightarrow Q_i \in \text{NonConQueryList}
\]

\[
\forall Q_i \in \text{LeafQuery} \lor Q_i \in \text{nonSharedQuery} \rightarrow Q_i \in \text{ConQueryList}
\]

More specifically, \( \text{RootQuery} \) is the biggest shared selection predicate in input multi-query. Where the \( \text{ParentQuery} \) is the intermediate predicate query which holds a set of predicates and its results can be reused from other predicate queries. The \( \text{RootQuery} \) and \( \text{ParentQuery} \) can be scheduled and then run in strict ordered in non-concurrent multi-query plan. On the other hand, the set of no shared predicates can be run concurrently to improve the overall multi-query performance. The optimised queries are translated using Flink Table API in terms of dataset API and then submitted to Flink batch processing runtime environment. Finally, the desired results are returned to the end users.

4.3 Flink-MQO system implementation

The proposed Flink-MQO system optimises the input multi-query by exploiting the sharing data using the MRShare and relaxed MRShare techniques. The steps of the proposed Flink-MQO system could be described as follows:

1. The **predicate extractor** module receives the input queries (i.e., submitted queries by the big data analysts) and parses them based on the SQL regular expression. Then, the relation names, attribute names and predicates operators are extracted. After that, the parsed input queries are classified based on the extracted relations, attributes and predicates operators into the shared multi-query group and non-shared multi-query group. The shared multi-query group could consist of different groups according to the shared selection data operators.

2. The **reused-based multi-query optimiser** module estimates the equal and overlapped sharing data opportunities (i.e., shared selection predicate operators) in each shared multi-query group using the cost model of the MRShare and relaxed MRShare techniques respectively (Nykiel et al., 2010; Wang and Chan, 2013).
3 The query rewriter module rewrites the shared queries and then it generates the optimised multi-query execution plan which considers the equal and overlapped sharing data among multi-query.

4 The Flink-MQO system submits the optimised multi-query execution plan to Apache Flink. The Flink Table API translates the queries within the optimised multi-query plan into a set of relational operators such as selection predicates. These operators are scheduled into a set of batch processing jobs which submitted to Flink runtime environment. Flink reads the data from big data infrastructure and performs the optimised jobs over the lower data size comparable with the native processing. The final desired outputs are sent back to the big data analysts. Without loss of generality, Algorithm 1 illustrates the pseudo code of the proposed Flink-MQO system which is customised by using the MRShare and relaxed MRShare techniques.

### Algorithm 1: Flink-MQO system

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: $Q_{input} = [Q_1, Q_2, \ldots, Q_n]$</td>
</tr>
<tr>
<td>2</td>
<td>Output: $Q_{MRShare output} = [Q_1, Q_2, \ldots, Q_n]$</td>
</tr>
<tr>
<td>3</td>
<td>$Q_{RelaxedMRShare output} = [Q_1, Q_2, \ldots, Q_n]$</td>
</tr>
<tr>
<td>4</td>
<td>// Step 1: Predicates extractor</td>
</tr>
<tr>
<td>5</td>
<td>$Q_{parsed} = \text{ParseQuery}(Q_{input})$</td>
</tr>
<tr>
<td>6</td>
<td>$Q_{predicates} = \text{ExtractPredicates}(Q_{parsed})$</td>
</tr>
<tr>
<td>7</td>
<td>// Step 2: Sharing classification</td>
</tr>
<tr>
<td>8</td>
<td>$Q_{SharedGroup} = \text{GetSharedGroup}(Q_{predicates})$</td>
</tr>
<tr>
<td>9</td>
<td>$Q_{nonSharedGroup} = \text{GetnonSharedGroup}(Q_{predicates})$</td>
</tr>
<tr>
<td>10</td>
<td>// Step 3: Reused-based estimator</td>
</tr>
<tr>
<td>11</td>
<td>$Q_{MRShare} = \text{MRShare}(Q_{SharedGroup})$</td>
</tr>
<tr>
<td>12</td>
<td>$Q_{RelaxedMRShare} = \text{RelaxedMRShare}(Q_{SharedGroup})$</td>
</tr>
<tr>
<td>13</td>
<td>// Step 4: Rewriting Flink-MQO plan</td>
</tr>
<tr>
<td>14</td>
<td>for each $Q$ in $Q_{MRShare}$ do</td>
</tr>
<tr>
<td>15</td>
<td>$Q' = \text{RewriteQuery}(Q)$</td>
</tr>
<tr>
<td>16</td>
<td>$Q_{MRShare output} = Q_{MRShare output} \cup Q'$</td>
</tr>
<tr>
<td>17</td>
<td>for each $Q$ in $Q_{RelaxedMRShare}$ do</td>
</tr>
<tr>
<td>18</td>
<td>$Q' = \text{RewriteQuery}(Q)$</td>
</tr>
<tr>
<td>19</td>
<td>$Q_{RelaxedMRShare output} = Q_{RelaxedMRShare output} \cup Q'$</td>
</tr>
<tr>
<td>20</td>
<td>// Step 5: Generate Flink-MQO plan</td>
</tr>
<tr>
<td>21</td>
<td>$Q_{MRShare output} = Q_{MRShare output} \cup Q_{nonSharedGroup}$</td>
</tr>
<tr>
<td>22</td>
<td>$Q_{RelaxedMRShare output} = Q_{RelaxedMRShare output} \cup Q_{nonSharedGroup}$</td>
</tr>
</tbody>
</table>

5 Performance evaluations

In this section, the experimental evaluation of the proposed Flink-MQO system is presented. Starting by describing the experiment setup, dataset and used metrics.
5.1 Experiment setup

The experiments have been performed using Hadoop version 2.6.0 and Flink version 1.1.3 on a cluster of ten nodes. Each node was configured with a 4 GB of RAM, two cores and 200 GB disk and runs Ubuntu Linux 14.04.4 LTS.

5.2 Datasets and queries

The structured data is used through the experiments by generating different files size of TPC-H benchmark. The TPC-H benchmark is designed to measure the performance of database products that support data warehouse applications (Council, 2008). The lineitem table is used and l_discount attribute values (i.e., range attribute) is used as the selection attribute which stores some disjoint of distinct values for predicates operators. A set of synthetic queries generated from the following query template is used:

```
SELECT list of attributes FROM lineitem WHERE a ≤ l_discount ≤ b
```

5.3 Experimental results

In this subsection, the effectiveness of the proposed Flink-MQO system batch processing which implemented in Java is evaluated. We demonstrate the effectiveness of Flink-MQO system for MQO by varying data size (i.e., TPC-H benchmark) regarding tuples number such as 100 million, 250 million, 500 million and 1 billion. More specifically, a comparative study has been done among naive, Flink-MQO MRShare and Flink-MQO relaxed MRShare techniques which denoted by NT, F-MRT and FR-MRT respectively (Wang and Chan, 2013; Guoping, 2014; Nykiel et al., 2010). The multi-query execution plans which have been evaluated and compared are:

1. **NT** plan which runs the queries independently
2. **F-MRT** plan based on the similar sharing data opportunities
3. **FR-MRT** plan based on the similar sharing data opportunities with overlapping consideration.

Finally, two metrics are measured; execution time of queries and the reduction of filtered data. Typically, the execution time of queries is a common metric in data management systems where the execution time of a query is the elapsed time duration from query submission to query completion. While the filtered data indicates the tuples which could be filtered to answer a query running against data files such as the input file or the reused result file. It may be noted that using a different number of queries is ineffective because it does not always guarantee the increasing of sharing opportunities among multi-query.

5.3.1 Effect of data size

In this subsection, the performance will be examined as a function of data size which expressed as tuples number such as 100 million, 250 million, 500 million and 1 billion. In the conducted experiments, each technique will be examined and evaluated. Figure 7 and Figure 8 illustrate the performance of three multi-query execution plans; **NT**, **F-MRT** and **FR-MRT**; using 100 million, 250 million, 500 million and 1 billion tuples respectively.
Mostly, the multi-query execution time of NT significantly increases much faster than the multi-query execution time of F-MRT and FR-MRT techniques. In particular, the multi-query execution time of NT technique is increased by loading large input file for multiple times. Also, we can observe that FR-MRT technique outperforms the NT and F-MRT technique by increasing the number of tuples of input files.

Figure 7  Flink-MQO execution time for 100 and 250 million tuples (see online version for colours)

Figure 8  Flink-MQO execution time for 500 million and 1 billion tuples (see online version for colours)

Ultimately, another comparison view which concentrates on Flink-MQO system is presented exhaustively in Figure 9. It summarises the performance improvement of F-MRT and FR-MRT techniques with respect to NT technique. According to the results in Figure 9, it is found that FR-MRT technique outperforms F-MRT technique. The total averages of multi-query execution time improvement of the four conducted experiments for F-MRT and FR-MRT techniques with respect to NT are 59% and 63% respectively. Moreover, Figure 10 depicts the improvement of multi-query execution time for the FR-MRT with respect to F-MRT regarding different tuples number. The reason behind the
multi-query performance improvement is that the data sharing regarding overlapping consideration is increased which can reduce the loading data. Obviously, it can expand the predicates operators (i.e., filtering operations) compared to the identical predicates of the F-MRT. Based on these comparisons, it can be argued that FR-MRT technique can optimise multi-query more efficient using Apache Flink batch processing. Worthwhile, not only exploiting the overlapping is beneficial for the proposed Flink-MQO system, but also the overlapping ratios among shared multi-query play a crucial role in Flink-MQO system improving.

**Figure 9** Flink-MQO execution time improvement comparison for F-MRT and FR-MRT wrt NT (see online version for colours)

**Figure 10** Flink-MQO execution time improvement comparison for FR-MRT wrt F-RMRT (see online version for colours)
5.3.2 Effect of filtered data reduction

On the other hand, another benefit of exploiting the reused-based opportunities among multi-query is that the size of filtered data which loaded from static data sources is reduced. The proposed Flink-MQO system reads smaller data size with respect to the NT, which can speed up the queries execution times. On the other words, the filtered data in Flink-MQO system is smaller than that the filtered data in the original input file. Figure 11 depicts the number of filtered tuples for F-MRT and FR-MRT multi-query execution plans with 100 million, 250 million, 500 million and 1 billion tuples. It is noted that the FR-MRT technique can reduce the number of filtered tuples significantly with respect to F-MRT technique. Obviously, the average of improvement of filtered tuples reduction in FR-MRT with respect to F-MRT is 75%. Mostly, the increasing of filtered data reduction for FR-MRT plan is gained by exploiting overlapping of sharing data among input queries. Substantially, the superiority of filtered data reduction will be shown with slight increase of data source size. In consequence, the proposed Flink-MQO system can exploit the sharing opportunities to eliminate the redundancy and duplication of data in-network movement to diminish the multi-query improvement.

Figure 11 Improvement of filtered tuples reduction using 100, 250, 500 million and 1 billion tuples (see online version for colours)

The conducted experiments demonstrate the robustness of the proposed Flink-MQO system by exploiting shared data in multi-query. On the other hand, the sharing data on multi-query does not always guarantee many advantages when the results of the queries being shared would have fewer overlaps. Furthermore, the data size and portions of overlapping through multi-query can significantly improve the performance of the Flink-MQO system by avoiding redundant big selection predicates.

6 Conclusions

In this paper, Flink-MQO system is built on top of Flink batch processing to optimise shared multi-query over the big data. The proposed Flink-MQO system uncovers the benefit of exploiting shared selection predicates on multi-query using two existing sharing techniques; MRShare and relaxed MRShare. It can achieve significant
improvements in multi-query performance by reducing the overheads of redundancy which cause additional wasted work due to the processing of useless data. The experimental evaluation has shown that the Flink-MQO system can optimise the sharing data on a broader class of queries and speed up them compared to the Naive technique. Furthermore, the data size and portions of overlapping through multi-query can improve the Flink-MQO system performance significantly by avoiding redundant filtration tasks. As a future work, we plan to extend the proposed Flink-MQO system to address the effects of sharing ratio through multi-query in the case of batch and streaming processing.

References


