A comparison of most recent MapReduce joins algorithms

Sawsan Alodibat
Jordan University for Science and Technology
Irbid, Jordan
smalodibat15@cit.just.edu.jo

Enas Al-Shraida
Jordan University for Science and Technology
Irbid, Jordan
eialshrida15@cit.just.edu.jo

Abstract: **In this interesting line of research, an attempt has been to overview different parallel processing platforms that implement *MapReduce* jobs. This survey provides a wide-ranging analysis of work and publications related to MapReduce framework to data, and it also can be used as a basis for further research and examination. The scope of this survey is focused on pre-processing, pre-filtering, partitioning, replication, load balancing, performance, memory space, communication cost, and query processing and optimization aspects in the light of big data analysis in *MapReduce*. Moreover, a set of efficient optimized and improved approaches in the context of analytical query processing and optimizing using *MapReduce*. It provides an added value to current research published yearly by introducing a comprehensive classification of recently presented papers in the era of join types using *MapReduce*. From data-centric perspective, the main topic of this approach is intended to highlight the importance of traditional problems of data management and analysis in the regard of efficient big data processing and analysis approaches.**

Keywords: MapReduce, Hadoop, join types, multi-way join, theta-join, KNN join, top-k join, graph similarity join, semi join, filter join, bloom join, intersection join.

I. INTRODUCTION

There are many systems have been developed primarily to adopt big data analysis such as *Yahoo's PNUTS*, *Twitter Storm*, *LinkedIn's Kafka*, and especially *Google's MapReduce*. MapReduce, because of its simplicity, transformed the receiving of big data and large-scale processing; it becomes the most common framework used for vast datasets analysis based on machine learning techniques. *Apache Hadoop* is an open source which implements MapReduce framework and it has performed high popularity in both academia and industry due to its widespread usage [7].

MapReduce implementation in *DBMS* supports a set of functions: storage management, data partitioning, data compression, storage management, query optimization, and indexing. *Hadoop DB* presents the strategies of partitioning and indexing for parallel *DBMSs* based on *MapReduce* framework. *Hadoop DB* architecture includes three layers: top layer, middle layer, and bottom layer. In top layer, *Hive* is extended to convert queries to *MapReduce* jobs. In middle layer, *MapReduce* infrastructure and *HDFS* are implemented including fault tolerance, shuffling data between nodes, and caching intermediate files. In bottom layer, there are a set of computing nodes distributed in side layer to run individual instance of *PostgresSQL DBMS* and to store data [17].

Hadoop is an open source implementation of *MapReduce* which is the most common framework increasingly used by many companies including huge number of users. *Hadoop* is mainly compound of two parts: Hadoop Distributed File System (*HDFS*) and *MapReduce* to achieve distributed processing. *Hadoop* contains various servers: Job-Tracker and Task-Tracker to perform *MapReduce*, and *Name-Node*, *Secondary Name-Node*, and *Data-Node* to manage *HDFS*. *MapReduce* supports parallel processing of vast datasets; it includes two functions: Map function and Reduce function. Any job which has to be performed by *MapReduce* should go through these two phases. Map function is also called mapper which takes input including key-value pairs. It also performs some computational processes on the input to produce intermediary outputs formed also with key-value pairs. While Reduce function, which is also called reducer processes the obtained results from Map function; the data are shuffled to perform reduce phase. Shuffle phase sometimes takes time, network bandwidth, and other resources more than two main functions, Map and Reduce [7].

Data is stored by default in HDFS which consists of several Data-Nodes to store data. It also consists of Name-Node, a master node, to monitor Data-Nodes and maintain all Meta data. Data in HDFS is separated into multiple chunks that contain different Data-Nodes and equivalent in size. Two system processes are established, Job-Tracker and Task-Tracker, in MapReduce runtime. Job-Tracker is responsible to split a job into two phases: map and reduce that the user defines. It also arranges all tasks among different Task-Trackers. After that, Task-Tracker accepts the job and starts to process tasks assigned to map reduce functions. Task-Tracker will take a data chunk defined by Job-Tracker and apply map task on. Once every map task completes, all intermediate results are grouped into reduce tasks in order to obtain the results [18].

HDFS is a distributed file designed to store big data files in a stream data form with access pattern. It is designed to recognize and respond individual machines failures since it is potential to work on commercial hardware. The main workflow is as follows: data are copied to *HDFS* to perform *MapReduce*, and then results are also copied from *HDFS*. So HDFS is usually not the key storage of data. This typical

workflow scenario of using *HDFS* obeys to an access model called write-once read-many. In this model, random access to file parts is essentially costly in comparison with sequential access since *HDFS* is optimized for streaming access of large files. Files are possible to be only appended; there is no file update support [7].

A. Query optimization

Query plan optimization using many plan generations and selection algorithms can be performed and developed to find optimal plan for relational *DBMSs*. In addition, *MapReduce* system can further improve these optimization algorithms. Query optimization algorithms and more elaborate algorithms are needed since *MapReduce* jobs usually run longer than relational queries. Additionally, query execution time and query optimization time should be balanced to run fast relational optimization algorithms. To reduce the plan search space and to pipeline data between operators, only left deep plans are typically considered in most relational *DBMSs*. Query execution is more significant for efficiency so there will be no pipeline between the original *MapReduce* and the operator [17].

In this paper, we considered a set of papers related to MapReduce published early in main database journals and conferences from 2009 to 2016. We attempt to analyze the limitations of existing surveys' approaches related to *MapReduce* in order to outline their shortcomings and to make a comparison between them. In addition, we aim to define major encountered problems in terms of *MapReduce* tasks processing in order to provide categorization of entire work and research comprehensively according to the addressed problems. The main contribution of this paper is to present a powerful citation of current problems and their potential solving techniques and to talk about future work to improve novel systems in terms of *MapReduce* processing tasks. In our survey, we focus on the improvements of *MapReduce* framework by reviewing the primary *MapReduce* framework and its multiple implementations. Different approaches have been implemented using *MapReduce* framework since it has no real specification of the way of implementing components. Therefore, we compare the design and features of different well known implementations of *MapReduce* framework.

II. Background/Literature review

In the following section, an overview is introduced to provide many techniques and methods presented in the literature in terms of *MapReduce* performance improvement. We organize the categorized approaches of *MapReduce* improvement in a specified classification based on the introduced improvement.

Many purposes have been realized to improve the usefulness of database operators via *MapReduce* algorithms especially in intensive applications. In *MapReduce* framework, Map function is able to easily support simple operators such as select and project, but it cannot achieve theta-join, equi-join, multi-way join, and similarity join [17].

A. Multi way join

Multi-way join is more complex join implementation than binary join. It can be implemented either using only one *MapReduce* job which is called replicated join or using multiple *MapReduce* jobs (one job for every join). Multiple *MapReduce* jobs are used to execute *multi-way* join by achieving a series of *equi-joins*. Every single equi-join is performed by one *MapReduce* job, and every result of one *MapReduce* job passes to next *MapReduce* job as input. Usually, several join orders can lead to different performance based on different query plans that can be generated [17].

In the paper of [20], a multi-way join was presented to compute a set of matrix multiplications among several relations. The proposed algorithm can reduce the number of binary multiplications by taking the advantage of multi-way join operation. The proposed algorithm was implemented based on MapReduce framework, which provides us an ability to achieve the scalability of large matrix multiplication. The paper took a different perspective differs than several papers have investigated matrix multiplication using MapReduce. In the paper, the concept of parallelism was employed in the expansion of the problem from binary multiplication to *n*-ary multiplication of the whole equation. The multiplication was translated into a join operation in database systems to facilitate the efficiency of the matrices storage and to easier matrices multiplication of the most common matrices in graph data. Three types of algorithms were implemented: *S2*, *P2*, and *PM*. The experiments were processed on real world graph data in the paper have demonstrated the capacity of the parallel m-way join to enhance the process of matrix multiplication. Because of using the raw key implementation, the parallel two-way join algorithm can balance the intra-operation parallelism and inter-parallelism approaches.

In the paper of [12], *three-way joins* on MapReduce was studied in order to utilize distributed computation of joins using clusters of many machines for efficient graph algorithms. A state-of-the-art MapReduce multi-way join algorithm was shown in the paper to provide the appropriateness of using it with huge datasets. The aggregation step can be integrated into a cascade of two-way joins if the join result needs to be summarized or aggregated to get more efficiency. In the paper, the focus was on three-way joins for MapReduce specially for social networks analysis. However, the result of the join should be preferably cascaded of two-way joins to reduce the communication cost. Multi-way join algorithms are divided into three sub-types including Replicated join, Star join, and Theta-join as shown as follows:

- *Replicated join*

Replicated join is performed by as a single *MapReduce* job to perform multi-way joins. There is a special case of replicated join called star join that perform all join conditions on the same attribute or a set of same attributes [17].

- *Star join*

Star join can be implemented by one *MapReduce* job by setting the map output key to be the join attribute and deploying load balanced if needed [17].

- *Theta join*

Theta-join or *θ-join* is a join operator contains one of the following join conditions: (<, >, =, <=, >=, or !=) [17]. In real practices, more specifically in complex relations, *multi-way theta-join* queries are powerful. The most challenging task is to minimize the total processing time span through the best schedule sequence of execution of *MapReduce* jobs by mapping *multi-way theta-join* query [34].

In the paper [21], a proposed algorithm to implement theta-join as a single *MapReduce* job was presented. The implementation was achieved without changing *MapReduce* framework by constructing proper functions of Map and Reduce. The goal of the paper is to minimize job completion time. To do this, an appropriate join *matrix-to-reducer mappings* was used to define a great diversity of join implementations. An algorithm was proposed called *1-Bucket-Theta* that uses *matrix-to-reducer mappings* on any join which has output significantly fraction of cross product and on cross product especially. Moreover, even though the proposed algorithm, *1-Bucket-Theta*, is slower than other faster algorithms, other algorithms cannot be identified as usable unless knowing the join result in advance or performing expensive an analysis. The proposed algorithm consists of M-Bucket class of algorithms that can exclude large regions of join matrix and reduce input-related costs to improve running time for any theta-join. The proposed approach does not need to change *MapReduce* model; it supports any theta-join in a single *MapReduce* job. Indeed, the proposed algorithm can be integrated with high level programming languages on top of *MapReduce*. There are better algorithms that *1-Bucket-Theta* for selective join conditions. On the other hand, these algorithms include an essential fraction of the join matrix cells that are unassigned to any reducer. In practice, finding enough of such matrix cells can be impossible or computationally very expensive due to complex user-defined join conditions and insufficient input statistics. Due to the lack of proof that better matrix-to-reducer mapping does not miss any output tuple, we cannot use it.

Extending current solutions from traditional distributed and parallel databases for *multi-way theta-join* queries is relatively difficult to fit huge data volumes. In the paper of [34], a study was conducted from cost effective perspective of the problem of efficient processing of *multi-way theta-join* queries based on *MapReduce* identification and scheduling. Efficient processing of *multi-way theta-join* has not never been fully explores although of many works have been done using *key-value* pair-based programming model to support join operations. The most challenging task is to minimize the total processing time span through the best schedule sequence of execution of *MapReduce* jobs by mapping *multi-way theta-join* query. The main solution provided in the paper includes two parts: using only single *MapReduce* job for efficient execution of *chain-typed theta-join*, and how to execute single *MapReduce* job or a set of *MapReduce* jobs in a certain order and the corresponding cost metrics. The method can achieve substantial improvement of the join processing efficiency compared to other widely adopted solutions. In fact, the work introduced for the first time the exploration and evaluation of *multi-way theta-joins* using *MapReduce*. In the work, a set of rules were established to decompose a *multi-way* join query, in order to evaluate the cost model to execute *multi-way* join query for both single *MapReduce* job and multiple *MapReduce* jobs. Thus, extensive experiments were conducted to validate the proposed cost model and the solution of *multi-way theta-join* queries, and to compare with the state-of-art solutions in terms of query evaluation efficiency. A *Hilbert* curve based space partition method was proposed in the paper to reduce the volume of copying data over network and to adjust the reduce tasks workload. Certainly, the proposed schema in resource restricted scenarios for scheduling can help to achieve the evaluation of complex join queries a near optimal time efficiency.

In the paper [13], a binary theta-join and pre-processing clustering algorithm were introduced in MapReduce framework. The optimal trade-off between the communication cost and the size of the input can be reached using the best-known algorithm which has high join selectivity. Thus, the improvements of the state-of-the-art have been presented when the join selectivity is low. In addition, load imbalance was considered across reducers to decrease the communication cost and the maximum load of a reducer. The proposed algorithm in the paper is based on *1-Bucket-Theta* and *M-Bucket*. *1-Bucket-Theta* requires minimal statistical information and examines all tuples pairs, making it the most generic algorithm. *M-Bucket-I* is better than *1-Bucket-Theta* in cases that the join selectivity is small. The worst-case behavior of 1-Bucket Theta matches the lower bounds for the binary theta-join problem, so an analysis was performed. Clustering histogram buckets were performed to improve these algorithms by achieving more efficient partitioning of histogram buckets to reducers. The imbalance across reducers, the maximum reducer input, and the replication rate are the main factors of the efficiency. In the paper, the results have revealed that load imbalance is not significantly affected by improving the replication rate and maximizing reducer input. The main difference between M-Bucket-O and M-Bucket-I is that the earlier aims to minimize the maximum reducer output, whereas the last aims to minimize the maximum reducer input. *Join Matrix (JM)* is used to operate *M-Bucket-I* partitioner. *JM* includes each cell corresponds to pair of histogram buckets; trying to create a region for each single reducer and fit cells in these regions. The main goal is to improve the quality of the partitioner phase by reducing rows and columns of *JM*. The results confirmed that the proposed partitioning algorithm provides up to 59% better time performance. Once the selectivity becomes lower and the number of the band condition increase, the improvements become more significant. However, the approach is not intrusive; it can be integrated with the existing state-of-the-art.

In the paper [31], a proposed algorithm called Strict Even Join (*SEJ*) was designed to partition multi-way theta joins into smaller groups and selects the best one based on one MapReduce job. Therefore, by calling *SEJ* algorithm, a dynamic algorithm is elaborated to optimize the multi-way theta joins. The experiments have proved the feasibility and efficiency of the proposed randomized algorithm. A method called *largrangian* was used to minimize the communication cost between map and reduce functions and to compute the

estimated results per relation. The experiments in the paper have shown the efficiency and the stability of the proposed algorithm in terms of multi-way joins using one MapReduce job rather than cascades of two-way joins.

B. Equi join

Equi-join is a special case of theta-join where join condition can be only "=". *MapReduce* implementation follows strategies of earlier parallel database implementation on equi-join operator [17]. Equi-join exploits *MapReduce* key-equality which requires more complex join based data flow management. *MapReduce* provides balancing between mapper nodes easily due to its simplicity nature. However, in some cases, standard equi-join algorithm could delay job completion whether a reducer receives a larger shared work. For this reason, balance load between reducers can resolve the issue by minimizing the greater amount of work allocated to a reducer and then minimizing job completion time [21]. Equi-join implementation has four variant types: repartition join, map-only join, reduce-only join and semi join [17].

- *Repartition join*

The default join algorithm and the most basic *equi-join* implementation for *MapReduce* in *Hadoop* is repartition join which is the most general join method that can be implemented as one *MapReduce* job. In repartition join, map phase repartitions two tables and then tuples are shuffled with the same key. After that, the result of map phase is assigned to the same reducer which joins the generated tuples [17].

- *Map-only join*

Map-only join consists of only map phase; it partitions input data based on the join key and then shuffles it to the reducers. *Map-only* join can be implemented on co-partitioned relations based on the join key [17]. Map-side join is an algorithm without Reduce phase [24]. The data sets in addition to their partition are sorted by the same ordering. The two sets of data pre-partitioned into the same number of splits by the same partitioner. This algorithm buffers all records with the same keys in memory, as is the case with skew data may fail due to lack of enough memory [24].

- *Replication side join*

Reduce-side join is an algorithm which performs data pre-processing in Map phase, and direct join is done during the Reduce phase [24]. The preprocessing is sorting for the keys. Semi-joins filtering is used to filter the original data. The partitioner must split the nodes by the key. The reducer should have enough memory for all records with a same key. It is the most time-consuming, because it contains an additional phase and transmits data over the network from one phase to another. The algorithm has to pass information about source of data through the network [24].

- *Semi join*

Semi-join can be implemented on *MapReduce* even it has been well studied in parallel database systems. It is efficient when the result of semi join is relatively small since it requires several *MapReduce* jobs and the result of semi join must be implemented first [17].

In the paper [3], a study of the properties *hash-based* and *sort-based* equi-join algorithms was focused in case of fully joining datasets loaded into the main memory. In large high performance distributed data processing system, building block of a single node setting is very important factor. When running analytical data processing services on hardware shared among parallel services, memory footprint is an important deployment consideration. The critical contributions of the work are: studying the impact of memory footprint for each join algorithm on the number of parallel queries can be achieved, in addition to improving query response time through allowing system implementers and query optimizers to use the optimal join algorithm. In addition, the impact of two physical characteristics of join algorithms regarding their input and output (data being hash partitioned on the join key and data being pre-sorted on the join key) was considered in the paper to measure the performance. To optimize complex query pipelines with multiple joins. In general, equi-join is expensive process and the improving the overall performance of main memory data processing is relatively a challenging task. The results showed that hash-based join algorithm performs faster than sort-based join algorithms in most cases. Thus, the hash-based algorithm consumes smaller memory footprint compared to sort-based algorithms. When join inputs is already sorted, sort-based algorithms become competitive. The main conclusion of the paper is that considering the physical characteristics of the input and output is required to achieve the best response time and consolidation for main memory equi-join processing.

C. Similarity join

Similarity join is one of many applications of join conditions where the results are similar to the join condition value but not equal to exact value. Therefore, there have been many proposed algorithms to find *top-k* most similar pairs, *k-nearest* returned tuples from two relations, and *KNN* join which finds the similarity between tuples based on their distances [17]. String similarity joins have received considerable interest. String similarity join is widely applied that aims to find all string pairs based on user defined threshold and a given similarity function [25].

In the paper [27], an efficient set-similarity join algorithm was proposed based on MapReduce to achieve joins in parallelism. For end-to-end set-similarity joins, a three-stage approach was proposed that takes a set of records as input and provides a set of joined records according to the set-similarity condition. In order to minimize the need for replication and to balance the workload, an efficient data-nodes partitioning technique was proposed. Both *self-join* and *R-S* joins were used to control the amount of data-nodes in main memory. The data still does not fit into main memory of a node even of the use of the most fine-grained partitioning. Extensive experiments were conducted to get results along with the increasing size of real data sets in order to estimate the scaling up and the speed up of the proposed algorithm and their properties. By exploiting the properties of the MapReduce framework, a discussion of different ways efficiently applied

was introduced in terms of multiple inputs, replication of join, and partitioning.

String similarity joins have received considerable interest to design new algorithms called *MGjoin* with the assistant of an inverted index. String similarity join is widely applied that aims to find all string pairs based on user defined threshold and a given similarity function. In the paper of [25], *two-step-filter-and-refine* was adopted by the proposed algorithm to identify similar string pairs adopted approach. The proposed algorithm can generate candidate pairs based on inverted index and verify the candidate pairs based on similarity join. On the other hand, the proposed algorithm could result in high verification cost caused by poor filtering power or greater power of filtering computational cost. The proposed approach was the first work to explore multiple prefix filtering method was performed based on different orders and a parallel extension of the algorithm. Extensive experiments were conducted and have shown that the proposed approach outperforms other approaches mainly state-of-the-art methods in terms of scalability and efficiency.

In the paper of [6], a scalable string similarity join called *MASSJOIN* was presented based on MapReduce. The proposed approach supports both character based similarity functions and set based similarity functions. Existing partition based signature scheme was extended to perform set based similarity functions, which generates key-value pairs by utilizing the signatures. Using the proposed approach, key-value pairs were merged in order to reduce transmission cost and the number of key-value pairs. Therefore, *light-weight* filter units were incorporated into *key-value* pairs in order to improve the performance and omit the factors of increasing transmission cost. The significance of the proposed method was shown by conducting extensive experiments; the results proved that the performance of the proposed method is better than the state-of-the-art approaches.

D. kNN join

KNN join is useful tool mostly used in data mining applications and spatial multimedia databases. It can produce *K Nearest Neighbors* (*KNN*) from one relation for every point in another relation. Performing *KNN* joins efficiently is a challenging task since it involves both the join and *NN* search. Hence, the applications continue to expand with the amount of data need to process. *KNN* execution on large data stored in *MapReduce* is the main challenging and interesting task since it frequently needed in practice [30]. *KNN* join is costly operation since *NN* search and join are expensive especially when datasets are in large multi-dimensions. There has been little research on parallel *KNN* joins in large data since it incrementally increases being exponential rate of datasets and a challenging task. On the other hand, there have been many parallel algorithms in *MapReduce* for *equi-joins*, *similarity joins*, *theta-joins*, and *spatial range joins*. Hence, many challenging and interesting problems were encountered regarding implementing *KNN* joins in *MapReduce* [30]. K Nearest Neighbour *KNN* join is a primitive operation commonly implemented by various applications of data mining. *KNN* join is designed to find *k* nearest neighbours from one dataset for every object in another dataset. However, *KNN* is an expensive operation since it combines *k* nearest neighbour query and join operation. Moreover, performing *KNN* join on centralized machine is difficult with the increasing volume of data [18]. In many application domains, *K* Nearest Neighbours is one of the popular methods used to achieve query point or a set of query points namely KNN-join. Many problems have received much effort to resolve and to adopt changes to the database specially in stand-alone systems and spatial databases. These problems may limit the efficiency of relational database management system large data applications [28]. Typically, KNN join operation correlates a data object located in one dataset with the corresponding k nearest neighbor located in the same or different dataset [29].

In the paper [30], a novel algorithm was proposed in order to implement parallel *KNN* joins on large data using *MapReduce* demonstrated by *Hadoop*. The extensive experiments in the paper have demonstrated the scalability, efficiency, and effectiveness of the proposed methods in large and synthetic datasets. *KNN* join is costly operation since *NN* search and join are expensive especially when datasets are in large multi-dimensions. In the work based on previous observation, a motivation pays an attention to explore the problems associated with *KNN* joins execution on large data in *MapReduce*. First, *Block Nested Loop Join (BNLJ)* was the basic approach was proposed and then it was improved using R-tree indices. The basic approach does not scale well for large and multidimensional data due to the quadratic number of partitions produced (number of dataset input blocks and reducers). *MapReduce* friendly was introduced to handle this limitation. *MapReduce* friendly is an approximate algorithm dependent to multi-dimensional datasets mapping into single dimension. For example, transforming *KNN* joins and space-filling curves are converted to a set of single dimension range searches. The proposed algorithms presented in the work were applied in *MapReduce* framework and the above issues raised in *Hadoop* were handled. The extensive experiments conducted in the work have been implemented over large real datasets, and the results confirmed good approximation quality which constantly outperforms the basic approach. Parallel *KNN* join in *MapReduce* was studied in the work including proposing exact *H-BRJ* and *H-zKNNj* approximate algorithms.

In the paper [18], an investigation to perform *KNN* join using *MapReduce* was presented. In map phase, cluster objects are divided into groups, and then *KNN* join is performed on each group of objects independently in reduce phase. Hence, the proposed mapping mechanism is designed to exploit distance-filtering rules using *Voronoi-diagram-based* partitioning method in order to minimize computational and shuffling costs. Two approximate algorithms were proposed to reduce number of replicas and then reduced shuffling cost. The primary contributions of the paper are: presenting an implementation of KNN joins for multi-dimensional and large volume datasets using *MapReduce* framework without any modification. Additionally, in order to perform *KNN* join, an efficient mapping method is designed to divide objects into groups; every group is processed by a reducer. The distances between data partitions are more closely between groups and reduce number of replicas. Moreover, a cost model was developed to compute the number of replicas resulted from

shuffling process. The extensive experiments have been conducted demonstrate the efficiency, robustness, and scalability of proposed methods.

In the paper of [28], a new method to achieve both *KNN* and *KNN* join in relational database was integrated with further query conditions. The main purpose was to design an algorithm that has the least impact and trivial changes to relational algorithms of database engine. The proposed algorithm uses *SQL* operators that generate the best plan to be used by query optimizer without radical changes to the database. The proposed approach is guaranteed to find the best-estimated *KNN* exactly in logarithmic cost in terms of number of block accesses required using only a small number of random shifts for databases in any fixed dimension. The extensive experiments have been conducted have demonstrated the efficiency and practicality of the proposed approach mainly on large, real, and synthetic datasets.

In general, *KNN-join* is designed to handle static datasets but not frequently updated datasets, whereas *KNN-join* is an expensive operation since it applied on high dimensional data. In the paper of [29], a novel *KNN* join method was proposed namely *KNN-join+* to provide an effective *KNN* results with no significant changes to high dimensional datasets. Additionally, the proposed method guarantees to answer KNN queries of the advanced applications with the least workload. The results have revealed the effectiveness of the *KNN-join+* to fast process high dimensional *KNN* join queries in both static and dynamic datasets. The proposed approach outperforms the existing indexing techniques by providing the excellent and scalable choice to handle dynamic dimensional data when dimensions are high especially in terms of sequential scan.

E. Top-k join

Many applications have used *top-k* similarity join to calculate the most *top-k* similar pairs among different data records in a dataset. Typically, the time performance in *top-k* join is a challenging issue with the increased applications that require processing vast datasets. However, traditional methods cannot easily find the *top-k* pairs in such massive amounts of data [4].

In the paper [14], a new class of queries called *top-k multiple-type integrated* query (*top-k MULTI*) was defined. The main role of *top-k MULTI* query is to treat several data types to find the relevance between the object and the query. It can deal with many data types such as relational, spatial, and textual data types. The main discrimination between traditional *top-k* query and *top-k MULTI* query is that the dependency of component scores on the *top-k MULTI* query to find final scores. Hence, traditional *top-k* spatial keyword query can be considered as an instance of *top-k MULTI* query. In the paper, an integration of the relational data type into the traditional *top-k* spatial keyword query to create *top-k spatial keyword-relational* (*SKR*) query to show the importance of *top-k MULTI* query. Additionally, an investigation of several approaches to process top-k MULTI query (hybrid index and separate index approaches) and *top-k* SKR query was presented. The key issue for *top-k MULTI* query processing is the Scalability due to the multiple data types integrated in a query. In hybrid index approach, all indices for *top-k MULTI* query are built in an integrated form creating multi-level indices. On the other hand, all individual indices are maintained independently in separate index approach. A new query processing method was proposed for the *top-k SKR query* called *Separate SKR* based on separate index approach. Therefore, two methods were presented based on hybrid index approach to the *top-k SKR* query through expanding characteristic methods for the *top-k* spatial keyword query. Finally, a comparison of the results of extensive experiments on *top-k SKR* query using real datasets was performed to measure the efficiency and scalability of the methods from storage and query performance perspectives. The results showed that *Separate SKR* was more efficient and scalable up to 13 times than extended hybrid index methods. Further, *Separate SKR* consumes storage space up to 3 times less than extended hybrid index methods. In conclusion, separate index method can be easily extended to encourage a new data type for *top-k MULTI* query.

The proposed algorithm in [4] namely *RDD-based* can perform *top-k* similarity join over large clusters on high dimensional data sets. In general, *RDD-based* algorithm involves four stages that load multiple high-dimensional records into *HDFS* to find the *top-k* closest pairs ordered based on *Hamming distances* to perform global *top-k* pairs. An efficient distance function based on *Locality Sensitive Hashing* (*LSH*) was developed to increase the process of top-k similarity join and comparisons. All pairs of *LSH* signatures are split into partitions to minimize the amount of data during the *RDD* running time. Therefore, the proposed algorithm is capable to calculate *top-k* closest pairs in parallelism by exploiting a serial computation strategy. The results of experiments have revealed the scalability and effectiveness of *RDD-based* proposed algorithm.

F. Graph similarity join

One of the advanced operations used in a wide range of academic, theoretical, real, and practical applications is to identify clusters or close-knit communities in graphs. Practical algorithmic heuristics are required to efficiently embrace the problem either the theoretical algorithm is computationally or inflexible expensive. A set of significant challenges remain in implementing these heuristics to work for large real world graphs such as irregular data access patterns, compound factors, scale of data, intensive operation computation, and better approximation restriction [26].

In a distributed framework like *MapReduce*, performing graph-analysis is a challenging task. Many approaches have been proposed for graph-analysis of algorithms, but they perform shuffling and storing phases which increase the cost of high communication in *MapReduce* [8]. Graph similarity joins is very important with the advent of massive graph-modelled data, since it is widely applied for many objectives such as data cleaning [5].

In the paper [8], a new design pattern for a family of iterative graph algorithms for *MapReduce* framework. The proposed method separates graph topology from the graph-

analysis results. In each iteration step, each *MapReduce* node existing in the graph participates in graph-analysis task and reads the same partition of the graph locally. Each node also reads all the current analysis results from the distributed file system. Using *merge-join*, the results of iterations are correlated to each graph partition locally, in addition to generate and dump the new improved analysis results in the graph partition to *HDFS*. The algorithm requires only one *MapReduce* job to perform pre-processing graph, and the actual analysis using repartition requires one *map-based MapReduce* job. All partial results are contained in *HDFS* which stores the result of map stage to perform *merge-join* between a partition of the graph and a global file. In detail, the method to perform graph-analysis used parallel *merge-join* between the partition of graph and a global table containing all partial results of each node. The *map-based* approach proposed in the paper outperforms the basic approach since it can improve the performance of graph-analysis. At end, the approach can reduce the communication cost and improve the performance by separating the graph topology from the graph-analysis.

A novel MapReduce-based algorithm called *pClust-mr* was proposed in the paper of [26] for a popular serial graph clustering. Thus, a novel application of the proposed method was developed to cluster biological graphs more specifically to identify dense sub graphs from bipartite graphs. The proposed algorithm uses pipelined MapReduce stages to implement a mixture of shuffling and sorting operations in order to process the edges of the graph as an input. The results have revealed the linear scaling of the time performance on small real world graphs.

In the paper of [5], graph similarity joins are considered under edit distance limitations in order to find the pair of closest to each other lower than a specified threshold. With the use of MapReduce programming model, a scalable algorithm was proposed namely *MGSJoin*, which applies filtering verification framework to perform the most efficient graph similarity join. The main idea of the algorithm is to count the overlapping graph signatures with filtered candidates. Spectral Bloom filters are introduced to minimize the number of key-value pairs with the potential issue of too many key-value pairs in filtering phase. In addition, multi-way join strategy was integrated to increase the efficiency of *GED* calculation for verification based on MapReduce. The proposed algorithm is efficient and scalable with prove of extensive empirical experiments demonstration. In the paper, the main focus is on graph similarity join mainly for graph processing data. For example, suppose there are two graph object sets with distance threshold, and we have to return graph similarity join including all pairs of graph objects contained in these two graphs in terms of the lowest distances between them. In pre-processing of graph data mining, graph similarity join has a wide range of applications. The main contribution of the work is to present MapReduce based graph similarity join algorithm to redesign the current in-memory graph similarity join algorithm. Moreover, large-scale graph datasets can be processed as a resulting baseline of the algorithm. More specifically, Bloom filter capacity was proposed to minimize intermediate key-value pairs. Therefore, optimized verification strategy was presented by multi-way join algorithm that can reduce number of rounds of MapReduce. The results have demonstrated the efficiency and scalability of the proposed algorithm against current solutions with the implementation in real publicly available datasets conducting in wide range of applications.

G. Bloom join

A Bloom filter is a space-efficient probabilistic data structure; that is used to test whether an element is a member of a set. A query returns either "possibly in set" or "not in set". Elements can be added to the set, but not removed. An *empty Bloom filter* is a bit array of *m* bits, all set to 0. There must also be *k* different hash functions defined, each of which maps or hashes some set element to one of the *m* array positions with a uniform random distribution. K is a constant, much smaller than *m*, which is proportional to the number of elements to be added. To *add* an element, feed it to each of the *k* hash functions to get *k* array positions. Set the bits at all these positions to one [19].

It reduces transmission cost. Bloom join with open source map-reduce framework of Hadoop improves the performance of query optimization. The reduce side join applied bloom filters which is inexpensive than map-side join [19]. There are two kinds of cases needing to be considered: two-way joins; that occurs between two data sets, and multi-way joins; that occurs between more than two data sets, and it is implemented by a sequence of two two-way joins [24].

Bloom filter: a type of the map-reduce join, the relation queue is used to decide which relations must be further processed. The memory space needed to store a bloom filter is small compared to the amount of data belonging to the set being tested. For improving the performance of query execution the reduce side join is used with filtering on the map side which generates less I/O cost, but there remain many non-joining tuples after filtering [23]. The individual input records can be processed in parallel. Map function does not only tag the input records but also filters them allowing only some of them to be part of the final map output, there is no replication for the elements. The input to the map function is file split. The hash functions and reduces the total processing cost. It reduces transmission cost, it reduces the amount of data transferred compared to semi-join by utilizing the concept of bloom filters [19]. Two-way join needs less memory space than multi-way join [32].

H. MRFA join

This algorithm, used to manage huge amount of data on large scale systems even for highly skewed data. It is Map/Reduce Frequency Adaptive Join algorithm based on distributed histograms and randomized redistribution approach. The support for fault tolerance and load balancing in Map-Reduce and Distributed File System are preserved if possible: the inherent load imbalance due to repeated values must be handled efficiently by the join algorithm and not by the Map/Reduce framework. Join computation in MRFA-Join proceeds in two map-reduce jobs. First, one phase to compute distributed histograms and to create randomized communication templates to redistribute only relevant data

while avoiding the effect of data skew, Map phase to generate a tagged “local histogram" for input relations then Reduce phase to create join result global histogram index and randomized communication templates for relevant data [10].

Second, another phase is used to generate join output result by using communications templates carried out in the previous step; Map phase to create a local hash table and to redistribute relevant data using randomized communication templates, then Reduce phase to compute join result. The detailed information provided by distributed histograms, allows to reduce communications costs to only relevant data while guaranteeing perfect balancing processing due to the fact that, all the generated join tasks and buffered data do not never exceed a user defined size using threshold frequencies. This makes the algorithm scalable and outperforming existing map-reduce join algorithms which fail to handle skewed data whenever join tasks cannot fit in the available node's memory. MRFA-Join can also benefit from Map/Reduce underlying load balancing framework in heterogeneous or a multi-user environment since MRFA-Join is implemented without any change in Map/Reduce framework. The overhead related to distributed histograms processing remains very small compared to the gain in performance and communication costs since only relevant data is processed or redistributed across the network [10].

MRFA-Join: a general join framework with filtering techniques in Map-Reduce. To avoid the effect of repeated keys, Map user-defined function should generate distinct output keys even for records having the same join attribute value. The load balance is perfect. To compute the join of two datasets, the input relations are divided into blocks (splits) of data in distributed histograms and to buckets in a randomized key redistribution approach. These splits are also replicated on several nodes for reliability. The detailed information provided by these histograms, communication costs is reduced to only relevant data processing due to the fact that all the generated join tasks and buffered data never exceed a user defined size. The overhead related to distributed histograms processing remains very small compared to the gain in performance and communication costs since only relevant data is processed or across the network. The global redistribution cost is reduced to a minimum [10].

I. Intersection filter

The intersection filters can filter out disjoint elements between two datasets, there are three approaches used to build the intersection filter. First approach is a pair of Bloom filters; specifying the set of intersection by eliminating all disjoint elements between the input datasets. Then filter out the disjoint elements in the input datasets by applying *pair of* bloom filters on the input datasets. Each tuple in one input dataset is queried into the Bloom filter of the other input dataset by *k* hash functions. If its join key is a member of the filter, the tuple containing this key will be returned because the key is a common member of the two input datasets. Otherwise, the tuple will be removed from its dataset because its join key is a disjoint member and this tuple is a non-joining tuple. This approach does not require the filters to have the same size *m* and *k* hash functions. The second approach is Intersection of un-partitioned Bloom filters; this approach is based on the idea that intersecting Bloom filters will produce a result filter called the intersection filter. There is unfortunately little difference between the intersection filter and the intersection of Bloom filters, it is used un-partitioned Bloom filter, only one intersection Bloom filter is used to remove most non-joining tuples from the input datasets instead of using two filters as the first approach. It should use the un-partitioned Bloom filters with the same size *m* and *k* hash functions [23].

The last approach is Intersection of partitioned Bloom filters; in this approach the partitioned Bloom filters are used to create the intersection filter. The size of partitioned Bloom filters can be changed after they are created. The filters may have different sizes but their partitions should have the same size. The intersection filter is generated by intersecting pairs of partitions of two partitioned filters. The intersection filter is generated by intersecting pairs of partitions of two partitioned filters. Two filters with 3 partitions are pair wise intersected with the bit-wise and to produce the result filter including three 4-bit partitions. This intersection filter represents the approximate intersection of the two datasets. If there exists at least one partition of the result filter containing all *m*/*k* bits equal to 0, the two input datasets are disjoint. So the join processing can be finished without doing anything. The pre-processing step is written as a standard map include two jobs running in parallel to process the input datasets (R and S) to build the intersection filter [23].

Intersection filter: Three approaches proposed to compute the intersection filter; intersection to Bloom filters, un-partitioned and Partitioned Bloom Filters. It filters out disjoint elements or non-joining tuples from both datasets, not only on one input dataset [23]. The intersection filter uses hash functions to portion the entire data sets. The memory space for first approach is small [11], but un-partition needs less memory space than partition [23]. The first approach needs to maintain two filters while others require one filter on nodes. A pre-processing enables a dramatic reduction in I/O and computational overhead. It produces much less intermediate data. Join processing in intersection filter can minimize disk I/O and communication costs. It is more effective through a cost-based comparison of join using different approaches. The preprocessing increase total cost, but it is small compared with other algorithms [22].

J. Parallel join: semi join

There are three ways to implement the semi-join operation; a semi-join using bloom-filter, semi-join using selection, an adaptive semi-join. The preprocessing in adaptive and selection semi-join is unique finding keys which are present in two datasets, and the relation queue is used to decide which relations must be further processed in bloom filter [10]. Delete the tuples that will not be used in join by using the filter will reduce the amount of data transferred over the network and the size of the dataset for the join. These filtering techniques introduce some cost, the semi-join can improve the performance but the larger data sets will decrease the performance. There is no replication on the data sets. The additional information about the source of data will increase

the data transferred. Memory space can be large depend on the size of the input data sets, but it can improve performance and reduce the possibility of memory overflow [24].

When a large portion of the data set does not take part in the join, deleting of tuples that will not be used in join significantly it will reduces the amount of data transferred over the network and the size of the dataset for the join. These filtering techniques introduce some cost, so the semi-join can improve the performance of the system only if the join key has low selectivity. There are three ways to implement the semi-join operation [24]. Parallel join is one of the most expensive operations in terms both I/O and CPU costs.

- *A semi join using bloom filter*

There are two jobs to perform the semi join. The Map phase and the Reduce phase. In the Map phase, the keys from one set are selected and added to the Bloom-filter. In the Reduce phase combines the output from Map phase into one. The second job filters only the output of the Map phase, increasing the size of the bitmap will increase the accuracy on this approach, but will increase the amounts of memory space needed. The advantage of this method is it's the compactness. The performance of the semi-join using Bloom-filter highly depends on the balance between the Bloom-filter sizes, which increases the time needed for its reconstruction of the filter in the second job, the large size of the data set can decrease the performance of the join [24].

- *A semi join using selection*

Semi-join with selection extracts unique keys and constructs a hash table. The hash table created in the first step filters the second set. In the context of Map-Reduce, the semi-join is performed in two jobs. Unique keys are selected during the Map phase of the first job and then they are combined into one file during the Map phase. The second job consists of only the Map phase, which filters out the second set. The semi-join using selection has some limitations. Hash table in memory, based on records of unique keys, can be very large, and depends on the key size and the number of different keys [24].

- *The adaptive semi join*

The Adaptive semi join is performed in one job, but filters the original data on the flight during the join. Similar to the Reduce-side join at the Map phase the keys from two data sets are read and values are set equal to tags which identify the source of the keys. At the Reduce phase keys with different tags are selected. The disadvantage of this approach is that additional information about the source of data is transmitted over the network [24].

K. Filter join

This type of join focused on reducing the number of map output records that are not joined. The map output records are replicated multiple times, so filtering out redundant records removes multiple copies of the record in multi-way joins. To join number of datasets simultaneously, some datasets need to be replicated. Replication may degrade the join performance, so it is important to reduce the number of redundant records. There are some filtering techniques to multi-way joins. Multi-way joins can be classified into two types: *common attribute joins* and *distinct attribute joins*. A common attribute join combines datasets based on one or more shared attributes, whereas some relations do not have join attributes in a distinct attribute join [16].

- *Common attribute joins*

The entire input datasets share joins attributes. The input records do not need to be replicated and they can be processed in a similar manner to two-way joins. A set of filters is created and probed in turn depending on the processing order of the input datasets [16]. There is no need to duplicate the entire datasets, the first data set used to make a set of filters to the next dataset. The cost depends on the number of input records, the ratio of the joined records, and the false positive rate of the filters; it is efficient when small portions of records participate in joins. There is no partitioning for input dataset. It needs a small memory space [16].

- *Distinct attribute joins*

Distinct attribute joins required the replication of some input datasets. The star-dim pattern delivered the best performance, chain join has the least performance between them. There are some equations can be used to select the processing order of the input data sets, and to estimate the join cost, because the processing order must be selected carefully because it affects the join cost, but there may be a large search space if the numbers of reducers and the input datasets are large, cost for star-fact is less than chain join but more than star-dim. The replication of input records for their corresponding reducers can be implemented in a similar manner to the data partitioning. The input datasets may not have some join attributes. Thus, some of the datasets with missing attributes need to be replicated because their records may be joined to the input records of other datasets with any values of the missing attributes. The filters can be applied in three patterns: *chain*, *star-fact*, and *star-dim* [16].

- *Chain*

The chain pattern creates filters similar to common attribute joins, except that each set of filters is created for a different join attribute [16].

- *Star fact*

The star-fact pattern creates filters using the dataset with both join attributes and uses the filters to process the other datasets [16].

- *Star-Dim*

The star-dim pattern creates filters using the datasets with missing join attributes and uses the filters to process the other dataset [16].

L. SJMR: parallelizing spatial join

Spatial join merges two spatial data sets with a spatial relationship between the objects. Spatial join is commonly used in applications such as spatial robotics, DBMS, and game programming. Given two sets of multidimensional objects in Euclidean space, a spatial join query can discover all pairs of

objects satisfying a given spatial relation, such as intersection. The input dataset is preprocessed to extract some key attributes. The input datasets are replicated to several partitions at Map stage. The filter is used to remove the tuples that cannot be parts of the result. This algorithm depends on good load balancing strategies. The grid partitioning method is used to dived the random data among *n* processors, the performance of SJMR improves by the increasing of node number, it's performance is high compared with PPBSM algorithm. Reduce task number and memory size of each node is increased to make the memory size large enough to filter and refine all in memory without writing operations so it needs large memory space. This method could only proceed when Reduce stage has finished completely, so its cost is high [33].

M. Massively parallel sort merge (MPSM) joins

MPSM join is a new sort-based parallel join method scaling almost linearly with the number of clusters. Therefore, this sort-based join outperforms hash-based parallel join algorithms on modern multi-core servers. Sort-based algorithms formed the basis for multi-core optimization in recent proposed approaches [1]. There are three type of this algorithm:

- *B-MPSM algorithm*

It is the basic form of MPSM algorithm, which is unaffected by any kind of skew. It allows some similarity to fragment and replicate distributed join algorithms. It only replicates merge join scans of the threads/cores but does not duplicate any data [1].

- *P-MPSM algorithm*

It is an improved MPSM version based on range partitioning of the input by join keys [1].

- *D-MPSM algorithm*

The MPSM can be effectively modified to non-main memory scenarios, in which intermediate data must be written to disk [1].

B-MPSM starts by generating sorted runs in parallel. There is no data duplication but instead data is partitioned into equal sizes. B-MPSM performs a number of worker threads of sort-merge joins in parallel. At result, it a large memory space is needed. P-MPSM uses join keys to split the input data sets. D-MPSM uses a small part of memory to store the data during join processing so it is a RAM-constrained version that spools runs to disk. The performance depends on the number of threads running in parallel. MPSM adds only very little overhead to the overall join processing. The algorithms take advantage of massive thread parallelism, fast inter-processor communication through local memory. Memory space needed for B-MPSM and P-MPSM are large but it is small for D-MPSM [1].

After deeply reviewed papers and works in the field of MapReduce join algorithms, we are motivated to make a comparison between these investigated papers. The comparison will be based on the following criteria (technical aspects of join algorithms): pre-processing, filtering, partitioning, replication, load balancing, performance, memory space, communication cost, query processing, and query optimization. Compared information is displayed in Table 1. It shows different MapReduce join algorithms from literature analyzed from the above-mentioned perspectives.

We classify MapReduce join algorithms into several categories including Multi way join, Equi-join, Similarity join, and Bloom join filter. Further, Multi way join algorithms are also divided into: two-way or three-way; replicated join, and star join, or theta-join. In Equi-join, the algorithms are also subdivided into: repartition join, map-only join, replication side, and semi join. Moreover, similarity join algorithms are classified into: top-k, KNN, string similarity, and graph similarity. Lastly but not least, Bloom join filter is either intersection filter or parallel join.

Table 1: a comparison of the investigated papers in MapReduce framework

Join type	Approach	Pre filtering	Pre Processing	Partitioning	Replication	Load balancing	Performance	Memory space	Cost
Theta-join	M-Bucket-Theta [13]	No	Clustering algorithm	Histogram buckets	Replication rate metric	Perfect load balancing	Significant improvements	Small	Reduced cost
	M-Bucket [21]	No	M-Bucket-I's preprocessing	Sampling	Heuristics	Yes	Better performance	Memory-aware	Minimized cost
	MRJ [34]	No	Sampling algorithm	Hilbert curve based space	Avoid repartitioning	Balance workload	Best performance	Round by round	More cost saving
	Random algorithm SEJ [31]	Yes	Clustering algorithm	largrangian method	Join groups	Yes	Stable performance	NA	Reduced cost
k-NN join	PGBJ [18]	Distance filtering	Random Selection	Voronoi diagram	Minimized # replicas	Workload balance	Poor performance	Large memory	Reduced costs
	kNNJoin+ self-join [29]	No	K-means Selection	Euclidean distance	No	Multiple KNN queries	Improved performance	Intermediate memory	Reduced elapsed time
	H-zkNNJ [30]	No	Sampling	Z-value based	Shifted copies	Distributed Cache	Best performance	Low memory space	Better than quadratic
	zχ-kNN [28]	Gorder	Randomly shifted	Clustering method	Repeated Euclidean	No	Very good performance	Small space Overhead	Logarithmic page accesses
Top-k join	top-k MULTI query [14]	No	Separate index	Rank-aware B+-tree	Nth-level index	No	More efficient	Less storage	Minimized elapsed time
	RDD-based algorithm [4]	Filter transformation	Locality Sensitive Hashing	Bucket based	LSH-based distance function	NA	Better performance	Fault-tolerant shared memory	Θ(n2)
Graph similarity join	MGSJoin [5]	+SBF Bloom filter	Sampling	Hash-based	Multi-way join	No load balancing	Significantly improved	In-memory join	Reduced cost
	pClust-mr [26]	Group/sort	Shuffling and sorting	Clustering	No	Yes	Optimal value	Efficient	Relatively low
	Map-based graph analysis [8]	Compute the metadata	No shuffling and sorting	User-defined partitioning	DFS sequence	No	Improved performance	Small enough	Low cost
String similarity join	MASSJOIN [6]	Light-weight filter	Sorting	Set-based similarity	Token frequencies	No	Significantly Improved	In-memory similarity join	Large cost
	PPJoin+ [27]	Prefix filtering	Grouping	Customize hashing	Minimum replications	Balance workload	Good performance	Optimized memory footprint	Increased cost
	MGJoin [25]	Prefix filtering	Tokenization and sorting	Data set partitioning	Single global ordering	Balance workload	Better performance	Inefficient in-memory join	Increased with data scale

Join type	Approach	Pre filtering	Pre Processing	Partitioning	Replication	Load balancing	Performance	Memory space	Cost
Multi-way join	**M-way: S2, P2, and PM [20]**	**No**	**Shuffling**	**MR-based binary matrix**	**Serial two-way join**	**Balances operation**	**Best performance**	**Greater network**	**Sorting overhead**
	Three-way join [12]	No	Sorting and shuffling	NA	Cascade of two-way joins	No	Better	NA	Reduced cost
Equi-join	STRSM [3]	No	Sorting	Hash-based	NA	Balanced work	Lowest response time	Smallest memory footprint	Lower total cost
Bloom filter	Two-way Join [2] [19]	Yes	Relation queue	Hash-based	Reduced	N/A	Not efficient	Small	Reduced
	Multi-way Join [32]	Yes	Relation queue	Hash-based	Reduced	N/A	Not efficient	Larger /two-way	Larger/ two-way
MRFA-Join	Distributed histograms [10]	Yes	Key	Blocks-based	Yes	Perfect	High	Large	Small
	Randomized redistribution [10]	Yes	Key	Buckets-based	Yes	Small	High	Large	Small
Intersection Filter	MapReduce [23]	Bloom filters	Relation queue	Hash-based	Reduced	N/A	High	Small	Reduced
	MapReduce [11]	Un-partition	Relation queue	No	Reduced	N/A	High	Less space	Reduced
	MapReduce [22]	Partition	Relation queue	K arrays-based	Reduced	N/A	High	Large	Reduced
Parallel Join	Semi-Join [24]	Bloom-filter	Relation queue	Hash-based	No	N/A	High/size	Large	High
	Map-Side join [15]	Selection Adaptive	Keys Yes	Yes yes	No No	N/A	High/size	Large	High
		Yes	Keys	Yes	No	N/A	Less/semi	Large	High
	Reduce-Side join [15]	Yes	Yes	Yes	No	N/A	Less/semi	Large	High
Filter Multi way join	Common attribute [16]	Yes	No	No	No	N/A	High	Small	Small
	Distinct Attribute [16]	Chain	Yes	Yes	Yes	N/A	Least	Large	High
		Star-Fact	Yes	Yes	Yes	N/A	Less/dim	Large	Less/chain
		Star-Dim	Yes	Yes	Yes	N/A	High	Large	Small
SJMR Join	Parallel join [33]	Yes	Yes	Yes	Yes reduced	Good	High	Large	High
MPSM Parallel Join	B-MPSM [1]	No	Yes	Yes	No	Small	High	Large	High
	P-MPSM [1]	No	Yes	Yes	No	Small	High	Large	High
	D-MPSM [1]	No	Yes	Yes	No	Small	High	Small	High

III. RESULTS AND DISCUSSION

In terms of theta-join, we investigated four of the most recent papers in MapReduce framework: *M-Bucket-Theta* [13], *M-Bucket* [21], *MRJ* [34], and *Random algorithm SEJ* [31]. *M-Bucket-Theta* in [13] a binary theta-join and pre-processing clustering algorithm were introduced. In addition, load imbalance was considered to decrease the communication cost and the maximum load of a reducer. The proposed algorithm in the paper is based on 1-Bucket-Theta that requires minimal statistical information and examines all tuples pairs. The results confirmed that the proposed partitioning algorithm provides up to 59% better time performance. However, [31] still perform join performance less than *M-Bucket* [21] that implements theta-join as a single *MapReduce* job without changing *MapReduce* framework. It implements memory-aware approach, minimizes total cost, and has better performance. Another examined paper was *MRJ* [34] that provides a solution using only single *MapReduce* job for efficient execution of *chain-typed theta-join* near optimal time efficiency. The method can achieve substantial improvement of the join processing efficiency compared to other adopted solutions. A *Hilbert* curve based space partition method was proposed in the paper to reduce the volume of copying data over network and to adjust the reduce tasks workload. The last approach is Random algorithm *SEJ* [31] that performs efficient multi-way joins using one MapReduce job rather than cascades of two-way joins due to the use of *largrangian* method to reduce communication cost and to increase performance.

In terms of *KNN* join, we investigated four of the most recent papers in MapReduce framework: *PGBJ* [18], *kNNJoin+* self-join [29], *H-zkNNJ* [30], and *zχ-kNN* [28]. The proposed method in [18] was designed based on mapping mechanism that exploits distance-filtering rules using *Voronoi-diagram-based* partitioning method in order to minimize computational and shuffling costs. *PGBJ* achieved poor performance due to the large memory consumption. However, *kNNJoin+* self-join proposed in [29] is more effective that results with no significant changes to high dimensional datasets and with the least workload. , *kNNJoin+* outperforms other indexing techniques by providing the excellent and scalable choice to handle dynamic dimensional data when dimensions are high. Another examined approach is *H-zkNNJ* in [30] that proposed *Block Nested Loop Join (BNLJ)* for scalability requiring only linear number of blocks, dataset input and reducers. The problems and issued addressed in the study are: the issue of reducing the amount of communication occurred between *Map* and *Reduce* phases, the issue of performing random shifts in *MapReduce*, and the issue of designing a good partition over single dimensional *z-values* for joins purpose. In [28], *zχ-kNN* uses *SQL* operators that generate the best plan to query optimizer without radical changes to the database. zχ-kNN approach is guaranteed to find the best-estimated *KNN* exactly in logarithmic cost in terms of number of block accesses.

In terms of top-k join, we investigated two of the most recent papers in MapReduce framework: *Top-k MULTI* query [14] and *RDD-based* algorithm [4]. First, *Top-k MULTI* query in [14] performed efficient and scalable storage and query performance using *Separate SKR*. Further, *Separate SKR* consumes storage space up to 3 times less than extended hybrid index methods. However, *RDD-based* algorithm proposed in [4] is based on *Hamming distances* and distance function based on *Locality Sensitive Hashing*. *RDD-based* algorithm can perform *top-k* similarity join over large clusters on high dimensional data sets.

In terms of Graph similarity join, we investigated three of the most recent papers in MapReduce framework: *MGSJoin* [5], *pClust-mr* [26], and Map-based graph analysis [8]. *MGS* Join in [5] applies filtering verification framework in graph similarity join mainly for graph processing data. *MGSJ* join was presented to redesign the current in-memory graph similarity join algorithm and to use the capacity of Bloom filter capacity to minimize intermediate key-value pairs. Therefore, it used an optimized verification strategy by multi-way join algorithm to reduce number of rounds of MapReduce which results in more efficient and scalable algorithm against other solutions. *PClust-mr* in [26] was proposed for serial graph clustering using pipelined MapReduce stages to implement a mixture of shuffling and sorting operations. *PClust-mr* results in linear scaling of the time performance on small real world graphs but it still needs to be improved for large graphs. Map-based graph analysis in [8] requires only one *MapReduce* job to perform pre-processing graph. It uses parallel *merge-join* to generate and dump the new improved analysis results in the graph partition to *HDFS*. It outperforms other approaches due to the improved performance of graph-analysis and the reduced communication cost by separating the graph topology from the graph-analysis.

In terms of String similarity join, we investigated three of the most recent papers in MapReduce framework: *MASSJOIN* [6], *PPJoin+* [27], and *MGJoin* [25]. *MASSJOIN* in [6] supports both character based similarity functions and set based similarity functions which generates key-value pairs by utilizing the signatures. *MASSJOIN* can reduce transmission cost and the number of key-value pairs using *lightweight* filter units to improve the performance better than others and to omit the factors of increasing transmission cost. *PPJoin+* in [27] is a *three-stage* approach and an efficient data-node partitioning technique proposed to minimize the need for replication and to balance the workload for end-to-end set-similarity joins. However, *PPJoin+* still does not fit into memory with the increasing size of real data sets even with the use of *self-join* and *R-S joins* to control the amount of data-nodes in main memory. *MGJoin* in [25] results in high verification cost caused by poor filtering power or greater power of filtering computational cost. *MGJoin* adopted *two-step-filter-and-refine*, which was the first work to explore multiple prefix filtering method based on different orders and a parallel extension of the algorithm.

In terms of Multi-way join, we investigated two of the most recent papers in MapReduce framework: *m-way: S2, P2, and PM* [20] and *three-way join* [12]. M-way in [20] uses three types of algorithms were implemented: *S2, P2, and PM*. *M-way* join can reduce the number of binary multiplications by taking the advantage of multi-way join operation. It has demonstrated the capacity of the parallel m-way join to

enhance the process of matrix multiplication differs than the rest of papers. It also can balance the intra-operation parallelism and inter-parallelism approaches because of using the raw key implementation and parallel two-way join algorithm. *Three-way join* in [12] utilizes distributed computation of joins using clusters of many machines for efficient graph algorithms. It uses a cascade of two-way joins if the join result needs to be summarized or aggregated to get more efficiency. However, the result of the join should be preferably cascaded of two-way joins to reduce the communication cost.

In terms of *Equi-join*, we investigated only one of the most recent papers in MapReduce framework: *STRSM* [3]. It studied the impact of memory footprint for each join algorithm on the number of parallel queries to improve query response time. It allows system implementers and query optimizer to use the optimal join algorithm and to optimize complex query pipelines with multiple joins. However, hash-based join algorithm performs faster and consumes smaller memory footprint compared to sort-based algorithms in most cases.

Two-way join is less efficient than the improved repartition join especially if the size of the relation is small. Therefore, two-way join needs additional Map-Reduce rounds to build the bloom-filters. As a result, it is more efficient than improved repartition join. However, when the size of the relation grows to over 50 million records, the bloom-filters can filter a lot of useless data to save network overhead and processing overhead. The bloom-filter can be used to filter useless data and eventually improve the efficiency of the two-way join and multi-way joins [24].

Bloom Filter, which works better than Semi-Join, reduces amount of data transfer between different sites and performs efficient query processing. Bloom join with open source map-reduce framework of Hadoop improves the performance of query optimization [19].

MRFA-Join algorithm ensures of perfect balancing properties during all stages of join computation [10]. The intersection filter has an extra cost for the preprocessing step, but its efficiency in space-saving and filtering often outweighs these shortcomings [23]. However, its performance is least compared with other join algorithms like bloom join and Reduce-Side-Join [22].

The un-partitioned intersection filters seem more efficient than the joins using the partitioned intersection filter because of their filtering performance. However, the partitioned intersection filter can easily discover disjoint datasets on a join key column and stop the join processing [22].

Common attribute filter joins and distinct attribute filter joins significantly outperformed the repartition join on the other hand, MFR-Join outperforms them and the semi-join with bloom filters [16]. Moreover, common attribute filter joins and distinct attribute filter joins improve the execution time significantly by reducing the amount of intermediate results when small portions of input datasets are joined [16].

The performance of SJMR algorithm was compared with the performance of PPBSM algorithm; the performance of PPBSM is less than the performance of SJMR algorithm. On the other hand, SJMR algorithm uses a technique called *reference tile method*, which is an improvement to *reference point method*. Instead of using a duplication and elimination operator at the end of SJMR, it is better to avoid producing duplicates online per reduce job. As a result, the filter step in [33] was modified through a simple test applied during the intersection's checking of rectangles. The results are illustrated in Table 1.

IV. CONCLUSION AND FUTURE WORK

We have produced big effort to create this survey that mainly specialized to MapReduce programming model and software framework. MapReduce is intended to facilitate and simplify the processing of massive amount of data through large clusters of commodity hardware in parallelism, reliable, and fault-tolerant manner. We summarized a number of join algorithms in introduced under Map-Reduce framework, and we compared between them based on a set of criteria. Per join type, some of the investigated approaches showed better performance than other approaches. Several join algorithms have presented disparate results in terms of pre-processing, filtering, partitioning, replication, load balancing, performance, memory space, and total cost. The performance of each algorithm depends on the size and duplicates of the input data sets. We can say that the preprocessing step can improve the performance but it requires additional cost such as incorporating filtering techniques. However, we cannot conclude that there is a specific algorithm has the best and perfect performance of the solution.

We will extend this research again by incorporating other comparison criteria and involving other recently published papers. We also can consider other join types that have not explored yet such as block-nested loop join, hash join, symmetric hash join, natural-join, and self-Join. In addition, we have an opportunity to follow-up enhancements shown by the previous algorithms. This survey is ongoing; it has a chance to be continuously up to date according to the improvements and developments to *MapReduce* framework as well as the enhancement of distributed and parallel computing based on MapReduce paradigm. One important opportunity for future work is to benefit every researcher interested in this area of research in order to resolve some encountered problems and shortcomings outlined in this paper. Conversely, this research acts as an initiation point to enhance being proposed algorithms in the field of big data analysis techniques based on MapReduce.

ACKNOWLEDGMENT

We specially thank Dr. Al-badarneh, Amer for his assistance and comments that greatly improved this work, and our parents for their support.

REFERENCES

[1] Albutiu, M. C., Kemper, A., & Neumann, T. (2012). Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, *5*(10), 1064-1075.

[2] Andreas, C. (2010). *Designing a Parallel Query Engine over Map/Reduce* (Doctoral dissertation, Master's thesis, Informatics MSc, School of Informatics, University of Edinburgh).

[3] Blanas, S., & Patel, J. M. (2013, October). Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *Proceedings of the 4th annual Symposium on Cloud Computing* (p. 19). ACM.

[4] Chen, D., Shen, C., Feng, J., & Le, J. (2015). An Efficient Parallel Top-k Similarity Join for Massive Multidimensional Data Using Spark. *International Journal of Database Theory and Application*, *8*(3), 57-68.

[5] Chen, Y., Zhao, X., Xiao, C., Zhang, W., & Tang, J. (2014). Efficient and Scalable Graph Similarity Joins in MapReduce. *The Scientific World Journal*, *2014*.

[6] Deng, D., Li, G., Hao, S., Wang, J., & Feng, J. (2014, March). Massjoin: A mapreduce-based method for scalable string similarity joins. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on* (pp. 340-351). IEEE.

[7] Doulkeridis, C., & Nørvåg, K. (2014). A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal*, *23*(3), 355-380.

[8] Elmasri, R., & Navathe, S. (2009). *Fundamentals of database systems*. 人民邮电出版社.

[9] Gupta, U., & Fegaras, L. (2013, October). Map-based graph analysis on MapReduce. In *Big Data, 2013 IEEE International Conference on* (pp. 24-30). IEEE.

[10] Hassan, M. A. H., Bamha, M., & Loulergue, F. (2014). Handling data-skew effects in join operations using mapreduce. *Procedia Computer Science*, *29*, 145-158.

[11] Jeffrey, M. C., & Steffan, J. G. (2011, June). Understanding Bloom filter intersection for lazy address-set disambiguation. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (pp. 345-354). ACM.

[12] Kimmett, B., Thomo, A., & Venkatesh, S. (2014, July). Three-way joins on mapreduce: An experimental study. In *Information, Intelligence, Systems and Applications, IISA 2014, The 5th International Conference on* (pp. 227-232). IEEE.

[13] Koumarelas, I. K., Naskos, A., & Gounaris, A. (2014, March). Binary Theta-Joins using MapReduce: Efficiency Analysis and Improvements. In *EDBT/ICDT Workshops* (pp. 6-9).

[14] Kwon, H. Y., & Whang, K. Y. (2015). Scalable and efficient processing of top-k multiple-type integrated queries. *World Wide Web*, 1-25. [15] Lee, K. H., Lee, Y. J., Choi, H., Chung, Y. D., & Moon, B. (2012). Parallel data processing with MapReduce: a survey. *AcM sIGMoD Record*, *40*(4), 11-20.

[15] Lee, T., Im, D. H., Kim, H., & Kim, H. J. (2014). Application of filters to multiway joins in mapreduce. *Mathematical Problems in Engineering*, *2014*.

[16] Lee, T., Im, D. H., Kim, H., & Kim, H. J. (2014). Application of filters to multiway joins in mapreduce. *Mathematical Problems in Engineering*, *2014*.

[17] Li, F., Ooi, B. C., Özsu, M. T., & Wu, S. (2014). Distributed data management using MapReduce. *ACM Computing Surveys (CSUR)*, *46*(3), 31.

[18] Lu, W., Shen, Y., Chen, S., & Ooi, B. C. (2012). Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, *5*(10), 1016-1027.

[19] Mahajan, S. M., & Jadhav, M. V. P. BLOOM JOIN FINE-TUNES DISTRIBUTED QUERY IN HADOOP ENVIRONMENT.

[20] Myung, J., & Lee, S. G. (2012, February). Matrix chain multiplication via multi-way join algorithms in MapReduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication* (p. 53). ACM.

[21] Okcan, A., & Riedewald, M. (2011, June). Processing theta-joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (pp. 949-960). ACM.

[22] Phan, T. C. (2014). Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce (Doctoral dissertation, Université Blaise Pascal-Clermont-Ferrand II).

[23] Phan, T. C., d'Orazio, L., & Rigaux, P. (2013, August). Toward intersection filter-based optimization for joins in mapreduce. In *Proceedings of the 2Nd International Workshop on Cloud Intelligence* (p. 2). ACM.

[24] Pigul, A. (2012). Comparative Study Parallel Join Algorithms for MapReduce environment. *Труды Института системного программирования РАН*, *23*.

[25] Rong, C., Lu, W., Wang, X., Du, X., Chen, Y., & Tung, A. K. (2013). Efficient and scalable processing of string similarity join. *Knowledge and Data Engineering, IEEE Transactions on*, *25*(10), 2217-2230.

[26] Rytsareva, I., & Kalyanaraman, A. (2012). An efficient MapReduce algorighm for parallelizing large-scale graph clustering.

[27] Vernica, R., Carey, M. J., & Li, C. (2010, June). Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 495-506). ACM.

[28] Yao, B., Li, F., & Kumar, P. (2010, March). K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *Data engineering (ICDE), 2010 IEEE 26th international conference on* (pp. 4-15). IEEE.

[29] Yu, C., Zhang, R., Huang, Y., & Xiong, H. (2010). High-dimensional knn joins with incremental updates. *Geoinformatica*, *14*(1), 55-82.

[30] Zhang, C., Li, F., & Jestes, J. (2012, March). Efficient parallel kNN joins for large data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology* (pp. 38-49). ACM.

[31] Zhang, C., Li, J., & Wu, L. (2013). Optimizing Theta-Joins in a MapReduce Environment. *International Journal of Database Theory and Application*, *6*(4), 91-107.

[32] Zhang, C., Wu, L., & Li, J. (2013). Efficient processing distributed joins with bloomfilter using mapreduce. *Int J Grid Distrib Comput*, *6*(3), 43-58.

[33] Zhang, S., Han, J., Liu, Z., Wang, K., & Xu, Z. (2009, August). Sjmr: Parallelizing spatial join with mapreduce on clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (pp. 1-8). IEEE.

[34] Zhang, X., Chen, L., & Wang, M. (2012). Efficient multi-way theta-join processing using mapreduce. *Proceedings of the VLDB Endowment*, *5*(11), 1184-1195.