Efficient Processing of Top-k Joins in MapReduce

Mei Saouk, Christos Doulkeridis, Akrivi Vlachou
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
chawkmay@gmail.com, cdoulk@unipi.gr, avlachou@aueb.gr

Kjetil Nørvåg
Department of Computer and Information Sciences
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
noervaag@idi.ntnu.no

***Abstract*—Top-k join is an essential tool for data analysis, since it enables selective retrieval of the k best combined results that come from multiple different input datasets. In the context of Big Data, processing top-k joins over huge datasets requires a scalable platform, such as the widely popular MapReduce framework. However, such a solution does not necessarily imply efficient processing, due to inherent limitations related to MapReduce. In particular, these include lack of an early termination mechanism for accessing only subset of input data, as well as an appropriate load balancing mechanism tailored to the top-k join problem. Apart from these issues, a significant research problem is how to determine the subset of the inputs that is guaranteed to produce the correct top-k join result. In this paper, we address these challenges by proposing an algorithm for efficient top-k join processing in MapReduce. Our experimental evaluation clearly demonstrates the efficiency of our approach, which does not compromise its scalability nor any other salient feature of MapReduce processing.**

I. Introduction

Rank-aware query processing is essential for large-scale data analytics, since it enables selective retrieval of a bounded set of the k best results according to user-specified ranking function. In real-life applications, ranking needs to be performed over combined records that stem from different input relations, using a query known as *top-k join*:

SELECT some attributes
FROM T_1,T_2,...,T_m
WHERE join condition AND selection predicates
ORDER BY $f(T_1.s_1, T_2.s_2, ..., T_m.s_m)$
LIMIT k

where the resulting join tuples are ordered using a scoring function $f()$ (*order by* clause) and the top-k answers based on their scores are returned to the user (*limit* clause). From the aspect of the database system, the challenge associated with ranked queries is to efficiently process the query by accessing only a handful of carefully selected tuples, without examining all input relations in their entirety.

In the era of *Big Data*, the importance of top-k join processing is paramount, since it is practically infeasible for users to inspect large unranked query result sets, simply due to their extreme volume. Yet, despite the significance of the problem, there is a lack of fully parallelized algorithms that operate efficiently at scale and provably return the correct and exact result. To the best of our knowledge, existing solutions for top-k joins in the context of MapReduce either cannot guarantee retrieval of k results (RanKloud [1]), or rely on the availability of a distributed data store that provides random access and execute the query in a centralized fashion (approach described in [2]).

Motivated by this observation, in this paper, we address the problem of processing top-k joins in the popular programming model of MapReduce over data stored in HDFS. We start by demonstrating that a simple parallel MapReduce algorithm that solves the problem has limitations with respect to performance. Then, we propose a new framework for top-k join processing in MapReduce that includes several optimizations at different parts of MapReduce processing, and boosts the performance of top-k join processing.

In a nutshell, this paper makes the following contributions:

- We present a novel framework for top-k join processing in MapReduce. Salient features of out framework include that it is *built on top of "vanilla" Hadoop*, without changing its internal operation, and that our solution consists of *one single, fully parallelized MapReduce job*, thereby exploiting parallelism as much as possible and avoiding the initialization overhead of chained MapReduce jobs.
- We advocate the use of data summaries, in the form of histograms, that are efficiently created using one-pass algorithms, in order to design efficient algorithms that solve the parallel top-k join problem.
- We equip our framework with several techniques based on the constructed histograms, whose combination is responsible for improved performance, including early termination, load balancing, and selective access of disk-resident data.
- We empirically evaluate our framework by experiments in a medium-sized cluster consisting of 12 nodes, showing that we obtain orders of magnitude improvements in execution time.

The rest of the paper is structured as follows: Section II provides an overview of related work. Section III describes preliminary concepts and the problem statement. Section IV presents a basic solution in MapReduce and highlights its limitations. Then, in Section V, the proposed framework for top-k joins is presented. In Section VI we presents an extension to our approach that significantly improves performance. Section VII demonstrates the results of our experimentation, and Section VIII concludes the paper.

II. Related Work

In *centralized databases*, many works have studied the problem of efficient top-k join processing, including J* [3], NRA-RJ [4], rank-join algorithm [5], and DEEP [6].

In the context of *distributed* data sources, Fagin et al. [7] study equi-joins of ranked data when the joining attribute is a unique identifier present in all relations (i.e., one-to-one join). However, in our work, we are interested in a generalization of this problem, focusing on arbitrary user-defined join attributes between relations (many-to-many join). Only a few studies have focused on this problem. Some of the previous approaches, including PJoin [8] and [9], are based on sampling to estimate score bounds that prune tuples which cannot belong to the result set. A more efficient approach is DRJN [10], where a two-dimensional histogram for each relation keeps the distribution of scores (number of tuples within each range) for each join value (or for ranges of values). This histogram is distributed to all participating nodes, and by performing a join on the histograms it is possible in a distributed way to calculate bounds and determine which tuples can participate in the final result, thus keeping the amount of tuples to communicate to a minimum.

Two approaches have been proposed for top-k join in the context of *MapReduce*. *RanKloud* [1] computes statistics (at runtime) during scanning of records and uses these statistics to compute a threshold (the lowest score for top-k results) for early termination. In addition, a new partitioning method is proposed, termed uSplit, that aims to repartition data in a utility-sensitive way, where utility refers to the potential of a tuple to be part of the top-k. The main difference to our work, is that RanKloud cannot guarantee retrieval of k results, while our approach aims at retrieval of the exact result. In [2] Ntarmos et al. study the problem of how to efficiently perform rank join queries in NoSQL databases. Their main approach is based on creating a two-level statistical datastructure (BFHM) that uses histograms at the first level, and Bloom filter on the second level. The BFHM index is stored in HBase, and top-k joins are executed by retrieving and processing (part of) this index structure. In contrast to our work, [2] relies on having the data stored in a database for efficient random access, while our approach can hand data files directly. Another important difference is that their approach for actually executing the query is centralized and executed by one coordinator, while our computation of the result is performed in parallel by all nodes.

Two of the key contributions in our approach are based on *early termination* and *load balancing*. Early termination in the context of MapReduce is avoiding reading the whole data set when only part of it (typically some fraction of the start of the data files) is needed in order to produce the result, while load balancing is important in order to minimize the job completion time, i.e., no Reduce task is overloaded and delays the completion of the job. The most common solution in previous work to early termination, is to 1) introduce a new input provider stage before the Mapper that controls reading of new splits to be processed by MapReduce, and 2) let one of the subsequent stages provide this input provider with information on whether more data should be read or not [11], [12]. Load balancing is typically performed by partitioning the data so that all Reducers receive approximately the same amount of data, or all Reducers have to perform the same amount of work [13], [14], [15], [16]. As will be described in more detail below, in contrast to previous approaches, we consider the problem as tasks with estimated finishing times, and base our approach on solving the *multiprocessor scheduling problem* [17]. For a more detailed discussion on early termination and load balancing in MapReduce, we refer to [18].

III. Preliminaries and Problem Statement

In this section we give a brief overview of MapReduce and HDFS, we define the type of queries we will focus on, and we formulate the problem statement.

A. MapReduce and HDFS

Hadoop is an open-source implementation of MapReduce [19], providing an environment for large-scale fault-tolerant data processing. Hadoop consists of two main parts: the HDFS distributed file system and MapReduce for distributed processing.

Files in HDFS are split into a number of large blocks which are stored on DataNodes, and one file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. The maintenance of mapping from file to block, and location (DataNode) of block, is handled by a separate NameNode. One important aspect of HDFS, is that HDFS is optimized for streaming access of large files, and as a result random access to parts of files is significantly more expensive than sequential access.

A task to be performed using the MapReduce framework has to be specified as two steps: the *Map* step as specified by a Map function takes input (typically from HDFS files), possibly performs some computation on this input, and distributes it to worker nodes, and the *Reduce* step which processes these results as specified by a Reduce function. An important aspect of MapReduce is that both the input and output of the Map step is represented as key-value pairs, and that pairs with same key will be processed as one group by the Reducer.

B. Top-k Queries and Top-k Joins

Given an input table or relation T with n scoring attributes, we use τ to represent a record (or tuple) of T, and $\tau[i]$ refers to the i-th scoring attribute ($i \in [1, n]$). A *top-k* query $q(k, f)$ returns the k best query results, based on a monotone scoring function f. When applied to relation T, the result of a top-k query $q(k, f)$ is a set of k records $\tau_1, \ldots, \tau_k$ of T with minimum scores, i.e., values of $f(\tau)$. Notice that without loss of generality, in this paper, records with minimum scores are considered best.

Quite often in rank-aware processing, the top-k results of a join of two (or more) input relations are requested. This

Symbols	Description
T_i	Input table (relation) $T_i(a_i, s_i, \ldots)$
a_i	Join attribute of table T_i
s_i	Scoring attribute of table T_i
τ	Record of T_i ($\tau = \langle \tau.id, \tau.a_i, \tau.s_i \rangle$)
k	Value of top-k
f	Function of top-k
$q(k, f, T_0, T_1)$	Top-k join query
b_i	Estimated score bound for table T_i
$H(T_i)$	Histogram of table T_i
R	Number of Reducers
T_i^{idx}	Index file of table T_i

TABLE I
OVERVIEW OF BASIC SYMBOLS.

is treated as an operator, called *top-k* join query, and can be answered in a straightforward (albeit costly) way by first performing the join and then ranking the join records by means of the scoring function. However, this results in wasteful processing, therefore efficient algorithms have been proposed that interleave the join with ranking.

In the context of this paper, we consider that an input relation (table) T_i contains a join attribute a_i, a scoring attribute s_i, as well as any number of other attributes, which are omitted in the subsequent presentation for simplicity. Thus, T_i consists of records τ described by a unique identifier ($\tau.id$), a join attribute value or join value ($\tau.a_i$), and a value of scoring attribute ($\tau.s_i$). We focus our attention on binary[1] many-to-many top-k equi-joins, where the input tables T_0 and T_1 are joined on a join attribute $a_0 = a_1$ and a combination of the scoring attributes (s_0 and s_1) of both relations is used as input to the scoring function f in order to produce the top-k join records. For a quick overview of the basic symbols used in this paper, we refer to Table I.

C. Problem Statement

We are now ready to formulate the problem addressed in this paper.

Problem 1. ***(Parallel Top-k Join)*** *Consider two input tables T_0 and T_1 with join attribute a_0, a_1 and scoring attribute s_0, s_1 respectively, which are horizontally partitioned over a set of machines. Given a top-k join query $q(k, f, T_0, T_1)$ defined by an integer k and a monotone scoring function f that combines the scoring attributes s_0 and s_1 to produce scores for join records, the Parallel Top-k Join problem requires to produce the top-k join records with minimum scores.*

In the context of MapReduce, input tables T_0 and T_1 are split in HDFS blocks and stored in HDFS following the concept of horizontal partitioning. A record τ in each file is of the form ($\tau.id, \tau.a_i, \tau.s_i$), where $\tau.id$ is a unique identifier, $\tau.a_i$ is the join attribute, and $\tau.s_i$ is the scoring attribute. In addition to this triple, each line may in general contain other attributes of the record τ of arbitrary length. Thus, in the general case, each DataNode stores only a subset of the records of each relation, and the problem is to design an algorithm that consists of a Map and Reduce phase, in order to compute the top-k join efficiently in a parallel manner.

Finally, we note that the most costly part of parallel processing of the top-k join is the computation of top-k join records per join value. Therefore, in this paper, we focus on providing a fully parallelized solution to this problem. The final step to obtain the top-k join results needs to process $k \cdot m$ join records (where m represents the number of distinct join values), which is typically orders of magnitude smaller than the size of the initial table T_i, i.e., $k \cdot m << |T_i|$. Therefore, this computation can be easily performed by a centralized program that processes these individual top-k results without significant overhead.

IV. A First Baseline Solution

In this section, we describe a basic solution in MapReduce that adopts the technique known as Reduce-side join. Then, we present the limitations and weaknesses of the basic solution that guide our design of more efficient algorithms for top-k join processing in MapReduce.

A. Basic Solution in MapReduce

We sketch a Reduce-side join algorithm, termed ***RSJSimple***, that computes the correct top-k join result. Figure 1 shows an example of the data flow in the execution of ***RSJSimple***.

First, the Mappers (which execute the Map function) access the input tables T_i record-by-record ($\tau = \langle \tau.id, \tau.a_i, \tau.s_i \rangle$). The input to the Map function (Algorithm 1) consist of a key-value pair, where the key is a unique identifier, and the value is the complete record τ. It outputs each key-value pair using a new CompositeKey that consists of (a_i, s_i, i) and the input record τ as value (cf. Figure 1(a)). The tag i is a value that indicates the table from which τ originates. The output key-value pairs of the Map phase are grouped by join value ($\tau.a_i$) and assigned to Reduce tasks using a customized Partitioner. Also, in each Reducer, we need to order the records within each group in ascending order of score ($\tau.s_i$), this is achieved through the use of the composite keys for sorting. The output of the Reduce phase is records of the form $a, \tau.id, \tau'.id, f(\tau, \tau')$, as shown in Figure 1(e)). Also, notice the unbalanced allocation to the two Reducers.

The combination of these techniques enables each Reduce task (Algorithm 2) to take as input all the records associated with a specific value of the join attribute, and perform the top-k join for each such join value independently of the other Reduce tasks. Moreover, due to the sorted access to records by ascending score, it suffices to read in memory (M_0 and M_1) only as many records as k from each input table (line 10), since any other record cannot produce a top-k join result (i.e., it will always produce a join record with worse score).

The actual computation of the top-k join for a given join value locally at the Reducer is straightforward, by joining the records of M_0 and M_1 and keeping the best k join records. Essentially, the Map phase groups the records of the two tables

[1]Notice that this is not restrictive and that our approach can be generalized for multi-way joins, but we only focus on binary joins for simplicity.

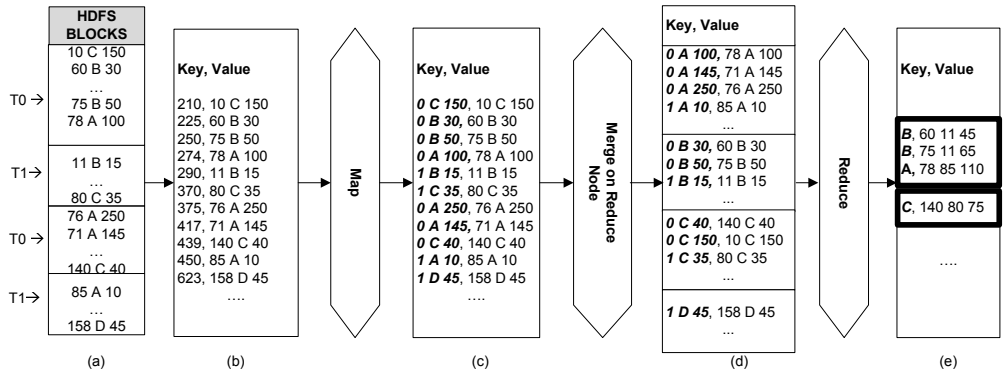


Fig. 1. Example of data flow in the execution of *RSJSimple*.

based on the join value, while the Reduce phase performs the top-k join operation for each such group individually. This produces as output the top-k join records for each join value.

In order to compute the k join records of the final result $q(k, f, T_0, T_1)$, we need to process the top-k join records of each join value and keep the global top-k join records independently of the join value. Given the fact that the individual top-k join records per join value are small in size (i.e., at most k records per join value), this computation is performed in the driver program of the MapReduce job that takes as input the outputs of the Reduce tasks without significant overhead.

Algorithm 1 *RSJSimple*: Map phase

```
1: Input: T_0, T_1
2: Output: records of T_0, T_1 based on join value
3: function MAP(τ (τ.a_i, τ.s_i): input record of table T_i)
4: if (τ ∈ T_0) then
5:     τ.tag ← 0
6: else
7:     τ.tag ← 1
8: end if
9: output ⟨(τ.a_i, τ.s_i, τ.tag), τ⟩
10: end function
```

Algorithm 2 *RSJSimple*: Reduce phase

```
1: Input: A subset of join values key_1,key_2,... with the associated
   sets of records V_1,V_2,...
2: Output: top-k records for join value key
3: function REDUCE(key, V: Set of records with join value:
   key sorted in ascending order of score)
4: for (τ ∈ V) do
5:     if (τ.tag = 0) then
6:         Load τ in M_0
7:     else
8:         Load τ in M_1
9:     end if
10:    if (M_0.size() ≥ k) and (M_1.size() ≥ k) then
11:        break
12:    end if
13: end for
14: output ⟨RankJoin(k, f, M_0, M_1)⟩
15: end function
```

B. Limitations of the Basic Solution

RSJSimple provides a correct solution to the parallel top-k join problem, however it has severe limitations with respect to performance. First, it accesses both input tables in their entirety, even though intuitively a much smaller set of records suffices to produce the correct result. In other words, it clearly results in wasting resources, in terms of disk accesses, processing cost, as well as communication. Ideally, we would like to access only a few HDFS blocks selectively and also terminate processing of the Map phase as soon as we have identified that the already accessed records are guaranteed to produce the correct result. Second, the assignment of Map output keys (join values) to Reduce tasks is performed at random, since it does not use any knowledge regarding the number of records associated with each join value. This can lead to unbalanced work allocation to the different Reduce tasks, thereby delaying the completion of the job, since the job completion time is determined by the slowest Reduce task. Ideally, we would like to intentionally assign join values to Reduce tasks, by exploiting knowledge of the join value distribution, so load balancing in the Reduce phase can be achieved.

V. A Framework for Top-k Joins in MapReduce

In this section, we present the proposed framework for top-k join processing in MapReduce. Salient features of our framework include that it is *built on top of "vanilla" Hadoop*, without changing its internal operation, and that our solution consists of *one single, fully parallelized MapReduce job*, thereby exploiting parallelism as much as possible and avoiding the initialization overhead of chaining MapReduce jobs.

A. Overview

The two input tables T_0 and T_1 are uploaded and stored as separate files in HDFS, sorted in ascending order based on scoring attribute[2]. In addition, for each input table, we compute

[2]Recall that, in this paper, without loss of generality, we retrieve the top-k join records with minimum aggregate values.

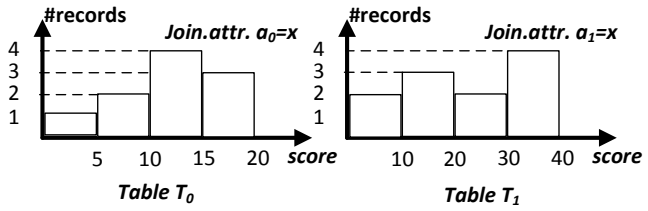


Fig. 2. Example of equi-width histograms of T_0 and T_1 for the same value of join attribute ($a_0 = a_1 = x$).

and store in HDFS histograms $H(T_0)$ and $H(T_1)$ that maintain the number of records of any value of the join attribute for a range of scoring values. Notice that this information can be constructed during data upload of the input tables to HDFS with negligible overhead, as will be demonstrated in the following.

Given a top-k join query, we compute *score bounds* for each input table (based on the histograms) that determine the subset of records sufficient to produce the correct result, prior to job execution. Thus, we can selectively load and process in the Map phase only a small subset of the stored data, and terminate processing of Mappers as soon as records with score values larger than the bounds are encountered. Moreover, we optimize the performance of the Reduce-side join by introducing a load balancing mechanism that distributes join values to Reduce tasks fairly. The resulting approach we call *RSJETLB*.

B. Histogram Construction

Processing data in Hadoop requires data upload, a phase where the entire dataset is read sequentially from an external source and stored in the distributed file system (HDFS). We can exploit this phase, which is mostly I/O-intensive and the CPU is underutilized, in order to seamlessly build histograms in the background. An important requirement is the availability of one-pass construction algorithms, which restricts the potential types of histograms that can be built. The constructed data synopses are stored in HDFS too, so that any MapReduce job can access it prior to or during execution. Typically, the size of histograms is orders of magnitude smaller than that of the original dataset, however an obvious trade-off exists between accuracy and disk size, i.e., higher accuracy can be achieved by constructing larger histograms that consume more disk space.

For our purposes, we choose to build equi-width histograms whose construction is simple and conforms with the one-pass requirement. In more detail, when a record τ ($\tau.a_i, \tau.s_i$) is read during the upload phase, the histogram of join value $\tau.a_i$ is updated by increasing by one the contents of the bin that corresponds to score value $\tau.s_i$. It should be noted however that our approach is able to take advantage of better histograms that are built by more complex algorithms, but their construction cannot be seamlessly done and would require some pre-processing (e.g., another MapReduce job).

Figure 2 depicts equi-width histograms of T_0 and T_1 for a specific common value of the join attribute. For each input table T_i, we create as many such histograms as the individual values of the join attribute. This set of histograms is denoted as $H(T_i)$. For instance, the depicted histogram of T_0 indicates that it contains 10 records with join value $a_0 = x$ in total. Also, the first histogram bin indicates the existence of 1 record with score between 0-5 (we will use the shorthand $[0-5]:1$ from now on). The remaining bins are: $[5-10]:2$, $[10-15]:4$, and $[15-20]:3$.

C. Early Termination

In order to reduce the processing cost of the join, we process only a subset of the input records of both tables that is guaranteed to provide the correct top-k join result. Intuitively, only those records of table T_i with scores lower than a score bound b_i participate in the join and contribute to the top-k join result. Therefore, in order to achieve early termination a method to determine the score bounds b_0 and b_1 so that all records with scores higher than b_i can be discarded as early in the process as possible.

1) Score bound estimation: Given as input only the histograms of both tables, the challenge is to compute correct score bounds b_i for the scores of input records of each table T_i. For this purpose, we employ a variation of the algorithm proposed in [10] for score bound estimation. In practice, this algorithm performs a join over the histograms of the two tables and estimates the number of join results and a range of scores for these results. The objective of this algorithm is twofold: first, to identify histogram bins and the corresponding score ranges that produce at least k join records, and second, to ensure that no other combination of histogram bins can produce join records with smaller score value than the k-th join record. To this end, histogram bins are accessed and joined until: (1) the number of join records exceeds k, and (2) the score of any join record produced by any unseen histogram bin is not smaller than the score of the current k-th join record. We explain the operation of the algorithm using an example.

Example 1. *Consider the histograms depicted in Figure 2 and assume that that the top-k join result (with $k = 1$) is requested using as scoring function the sum. By examining the first bin of each histogram, we know that there exist 2 ($= 1 \times 2$) join records with score in the range $[0-15]$, i.e., $[0-15]:2$. After examining the second bin of each histogram, we additionally know that there exist: $[10-25]:3$, $[5-20]:4$, and $[15-30]:6$. Only after the third bin of T_0 is examined (produced join records not shown here), we can safely stop processing, and report score bounds $b_0 = 15$ and $b_1 = 20$. This is because we already have at least 2 records (i.e., more than $k = 1$) with score $[0-15]$, and any join record that would be produced by combinations of unseen bins of T_0 or T_1 will have a score larger than 15.*

2) Implementing early termination in Hadoop: Assuming that the input tables are stored sorted in HDFS and that also the histograms are available, we create an early termination mechanism to process, in the Map phase, selectively only those input records with score lower than the respective bound.

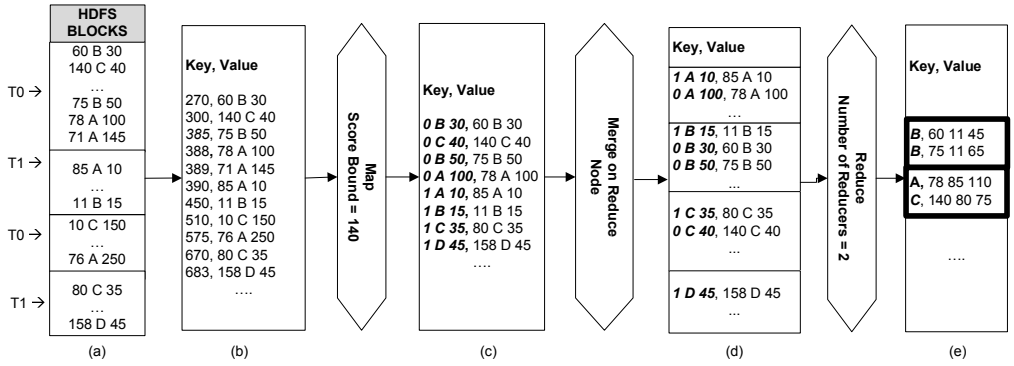


Fig. 3. Example of data flow in the execution of *RSJETLB*.

Algorithm 3 Map phase with early termination

Input: T_0, T_1, b_0, b_1
Output: records of T_0, T_1 with scores lower than b_0, b_1
function $MAP(\tau\ (\tau.a_i, \tau.s_i)$: input record of table $T_i)$
if $(\tau \in T_0)$ **then**
  if $(\tau.s_0 \leq b_0)$ **then**
    $\tau.tag \leftarrow 0$
    output $\langle(\tau.a_i, \tau.s_i, \tau.tag), \tau\rangle$
  end if
else
  if $(\tau.s_1 \leq b_1)$ **then**
    $\tau.tag \leftarrow 1$
    output $\langle(\tau.a_i, \tau.s_i, \tau.tag), \tau\rangle$
  end if
end if
end function

Algorithm 4 *Load Balancing*

Input: A: array of entries join value and number of join results, R: Number of Reducers
Output: $\mathcal{H}$: HashMap with *key* join value and *value* ReducerId
function $BalanceLoad(A, R)$
Initialize max-heap h with contents of A
$count \leftarrow 0$
while $(h \neq \emptyset)$ **do**
  $(v, res) \leftarrow h$.next()
  if $(count < R)$ **then**
    $\mathcal{H}$.put$(v, count)$
    $count++$
  else
    $r \leftarrow$ find Reducer assigned with minimum number of join results $\sum res$
    $\mathcal{H}$.add(v, r)
  end if
end while
return $\mathcal{H}$

It should be noted that the early termination mechanism is implemented on top of Hadoop by extending it, i.e., we do not change the Hadoop core. In this way, our algorithm is portable and compatible with different Hadoop distributions.

Algorithm 3 shows how early termination can be implemented in the Map phase. The algorithm takes as input the score bounds for each input table and accesses the sorted input tables. As long as an input record τ from table T_i has a score lower than the score bound b_i, i.e., $\tau.s_i \leq b_i$, the record is passed on to the Reduce task. It is guaranteed that no record with score higher than the bound can produce a join result that belongs to the top-k join results, so these can safely be discarded, thus significantly reducing the amount of records to be communicated to and processed by the Reducers.

D. Load Balancing

In order to balance the work allocated to the Reduce tasks, it is important to distribute the workload intentionally to the available Reduce tasks. Otherwise, the runtime of the entire job will be dependent on the execution time of the most loaded Reduce task, which may affect the overall performance considerably. To alleviate this problem, we introduce a load balancing mechanism tailored to the problem of top-k joins.

1) Fair assignment of join values to Reducers: In our setting, the workload of a Reduce task is defined by the number of records that it needs to process and join. Since Reduce tasks are assigned join values (and the associated records of any join value), our goal is to find an assignment of the join values to Reduce tasks, such that the maximum number of records that each Reduce task will process is minimized. This is known as the *Multiprocessor Scheduling Problem* [17].

Problem 2. ***(Multiprocessor Scheduling)*** *Given a set J of jobs where job j_i has length l_i and a number of processors m, what is the minimum possible time required to schedule all jobs in J on m processors such that none overlap?*

It is trivial to show that in our case jobs correspond to join values, job length is the number of join records associated with a join value, and processors correspond to Reduce tasks. Unfortunately, the Multiprocessor Scheduling Problem is known to be NP-hard. Therefore, we utilize a heuristic algorithm called *LPT (Longest Processing Time)* which sorts the jobs (join values) based on processing time (number of join records) and then assigns them to the machine (Reducer) with the earliest end time (lowest aggregate number of join results) so far. Moreover, it is shown in [20] that LPT achieves an

upper bound of $4/3-1/(3m)OPT$, where OPT refers to the optimal solution to the problem.

Notice that the reason that makes the LPT algorithm applicable in our setting is that we know the number of records associated with each join value in advance, before the MapReduce job starts. In turn, this is due to the histograms employed, which provide two vital pieces of information for query processing: (a) the score bounds (that allow for early termination), and (b) the number of records associated with each join value and with respect to the score bounds (that enables the load balancing mechanism).

Algorithm 4 shows the pseudocode of the load balancing procedure, which is executed prior to job execution. It takes as input an array A that keeps for each join value the number of join results, and the number of Reducers R. Notice that A is computed together with bound estimation. Also, we point out that A keeps track of join results based only on records with scores smaller than the score bounds, i.e., only those records that will be sent to the Reduce phase. The contents of A are inserted in a max-heap, so that they can be accessed based on descending number of join results. Then, each join value is retrieved from the max-heap and it is assigned to the Reducer with minimal load so far, in terms of aggregate number of join results. Finally, the HashMap $\mathcal{H}$ is returned which contains for each join value (key) the ReducerId (value) to which it is assigned.

Algorithm 5 *RSJETLB*: Map phase

```
1: Input: T_0, T_1, b_0, b_1, H (in DistributedCache)
2: Output: records of T_0, T_1 with scores lower than b_0, b_1
3: function MAP(τ (τ.a_i, τ.s_i): input record of table T_i)
4: r ← H.get(a_i)
5: if (τ ∈ T_0) then
6:     if (τ.s_0 ≤ b_0) then
7:         τ.tag ← 0
8:         output ⟨(τ.a_i, τ.s_i, τ.tag), τ⟩
9:     end if
10: else
11:    if (τ.s_1 ≤ b_1) then
12:        τ.tag ← 1
13:        output ⟨(τ.a_i, τ.s_i, τ.tag, r), τ⟩
14:    end if
15: end if
16: end function
```

2) Implementing load balancing in Hadoop: In order to implement the load balancing, each Mapper needs to have available the assignment of join values to Reducers. To this end, we utilize Hadoop's *DistributedCache* which is a mechanism that allows broadcasting data to all task nodes. In practice, we place the contents of HashMap $\mathcal{H}$ in the *DistributedCache*. Thus, each Mapper can easily retrieve the ReducerId for a given join value. Then, the Map function is modified accordingly in order to output also the ReducerId r, as shown in Algorithm 5. This is achieved by using a CompositeKey that in addition to join key, score and originating table (cf. Section IV-A) additionally contains r. Then, we implement a customized *Partitioner* that checks the CompositeKey of each output record and assigns the record to Reducer r. Notice that the Reduce function is identical to the one used in Algorithm 2.

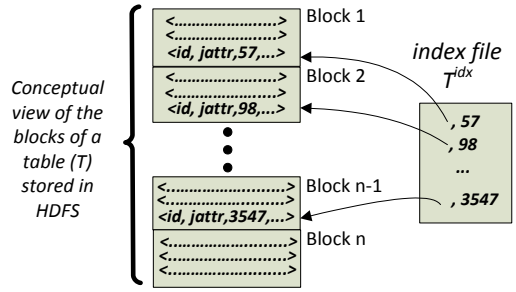


Fig. 4. The index file T^{idx} of a table T stored in HDFS.

Figure 3 presents the data flow of *RSJETLB*. As shown in the figure, the Map function has the score bound $b_0 = 140$ available and uses it to discard records of T_0 with higher scores. In this way, the Map function can terminate its processing as soon as it reads as input a key-value pair with score higher than the score bound. In addition, reduced communication is achieved as well as lower processing load in the Reduce phase. Figure 3(e) shows that 3 join values (A, B, C) are eventually sent to the Reduce phase. When $R = 2$ Reducers are used, the load balancing mechanism manages to fairly assign them to the Reducers, while a randomized partitioning, such as the one used as default by MapReduce, could assign A and B or B and C to the same Reducer.

VI. Selective Data Access

In order to attain performance gains in rank-aware processing, a critical factor is to avoid redundant I/O, since typically a few records suffice to produce the top-k join result. One shortcoming of *RSJETLB* is that although the Mappers only output records that are sufficient to produce to the final result, it still needs to access all blocks (splits) in the input files. In this section we describe how we can achieve only accessing blocks that contain records that are needed for producing the final result. This adds to our framework support for selective data access, which enables reading only part of the input files (i.e., only few HDFS blocks) during query processing.

We implement the functionality of early termination using a customized *RecordReader*. The RecordReader is the object responsible for accessing the input data and creating the key-value pairs that are passed as input to the Map tasks. The default RecordReader accesses the input data in its entirety. Instead, our customized RecordReader takes as input the score bounds for each input table and accesses the sorted input tables.

The key to access only part of the files is to be able to determine, given a particular score, at what point in the file the last record in the file that can contain this score. We achieve this by utilizing an *index file* T_i^{idx} for each input table T_i, that can be used to map between position and score. The index files are stored in HDFS and are typically created

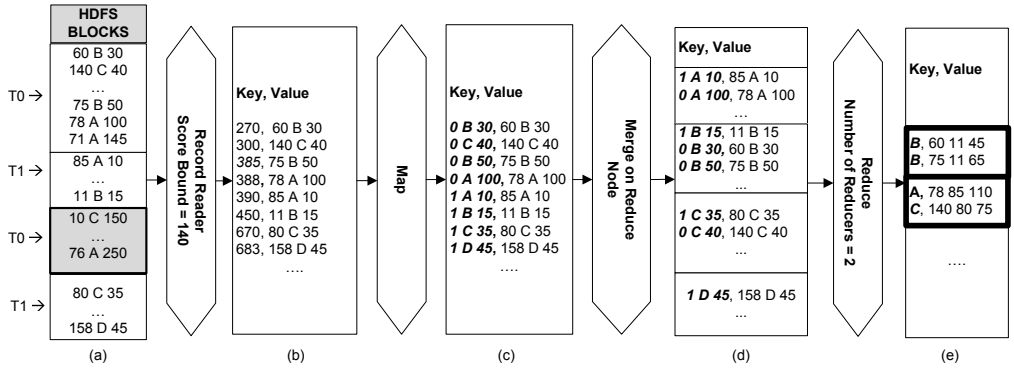


Fig. 5. Example of data flow in the execution of *RSJETIndex*.

during loading of data to HDFS at the same time as histogram construction is performed. Thus the index creation process incurs no additional I/O except for writing the index file. The index files contain records of the form: $(pos, score)$, as many as the number of HDFS blocks needed to store T_i minus one. Essentially, each record keeps the position (measured in bytes from start of the file) and the score of the last record in each HDFS block. Obviously, we do not need to keep this information for the last HDFS block.

Assuming a score bound b_i can be computed that signifies that only those records with score lower than b_i are sufficient to produce the top-k join result, then T_i^{idx} is exploited to identify the exact subset of HDFS blocks of T_i that contain these records. Put differently, T_i^{idx} provides the number of bytes the need to be read from the file of T_i. Figure 4 graphically illustrates an index file with pointers (position in bytes) to the last record of each HDFS block. Capitalizing on this knowledge, we provide our own implementation of an *InputFormat* class, where we override the *getSplits()* method, which is responsible for accessing input data and creating logical units of work (*InputSplits*) for the Map phase. In this way, only the selected blocks are accessed from disk and these blocks are used to form *InputSplits*.

Figure 5 demonstrates the data flow of the resulting algorithm *RSJETIndex*. Essentially, the RecordReader is able to avoid loading HDFS blocks. Assuming again a score bound $b_0 = 140$, the shaded block will not have to be read at all, since we know from the index that its records have scores larger than the bound.

VII. Experimental Evaluation

In this section, we report the results of our experimental study. All algorithms are implemented in Java in our experimental testbed RoadRunner [21].

A. Experimental Setup

Platform. We deployed our algorithms in an in-house CDH cluster consisting of twelve server nodes. There are two types of servers in the cluster: Nodes d1-d8 are of the first type, while nodes d9-d12 belong to the second type. The first server type has 32GB of RAM, 2 disks for HDFS (5TB in total) and 2 CPUs with a total of 8 cores running at 2.6 GHz. The second server type has 128GB of RAM, 4 disks for HDFS (8TB in total) hard disk space and 2 CPUs with a total of 12 cores (24 hyperthreads) running at 2.6 GHz. Each of the servers in the cluster function as DataNode and NodeManager, while one of them in addition functions as NameNode and ResourceManager. Each node runs Ubuntu 12.04. We use the CDH 5.4.8.1 version of Cloudera and Oracle Java 1.7. The JVM heap size is set to 2GB for Map and Reduce tasks. We also configure HDFS with 128MB block size and use a default replication factor of 3.

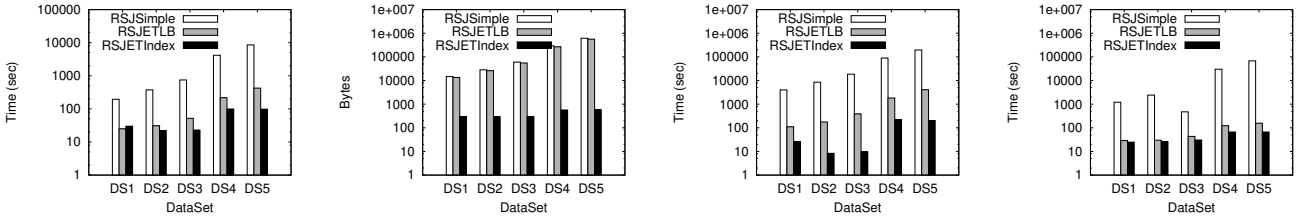
Algorithms. We compare the performance of the following algorithms that are used to compute top-k joins in Hadoop:

- *RSJSimple*: the baseline top-k join algorithm without early termination (Section IV),
- *RSJETLB*: the top-k join algorithm that uses early termination and load balancing (Section V), and
- *RSJETIndex*: the top-k join algorithm that additionally selectively loads HDFS blocks (Section VI).

Datasets and experimental parameters. In order to experiment with datasets of sufficiently high number of records, we used a synthetic data generator to produce large input datasets. We vary the size of input tables T_i from 1GB to 0.25TB. We use the following synthetic data distribution for generating the scoring attributes of relations: (a) uniform (UN), and (b) skewed (zipf distribution) with varying parameter of skewness 0.5 and 1, denoted as ZI0.5 and ZI1.0 respectively. We vary the number of distinct join values in each table (from 100 to 2,000), thereby affecting the join selectivity to study its effect on our algorithms. We also perform an experiment where the distribution of the values in the join attribute is varied to study this effect too.

In addition, we also vary the number of requested results (k) from 10 to 500. In all cases we set the number of Reduce tasks R equal to 10, since we noticed no significant gain in performance for our framework when increasing R. In all experiments, we use the sum as scoring function.

Metrics. The main metric used is the total execution time for each job. In addition, we measure the CPU time spent in Map and Reduce phases, as well as the size of InputSplits for

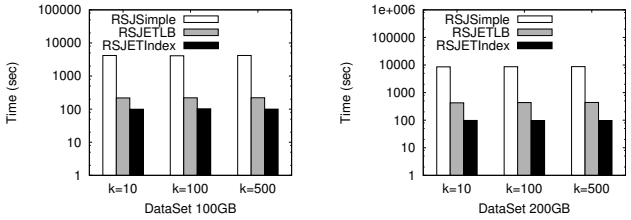


(a) Execution time. (b) Size of InputSplits. (c) CPU time Map. (d) CPU time Reduce.

Fig. 6. Scalability study of all algorithms with dataset size.

Dataset	Size T_0, T_1	Distinct join values
DS1	5.7GB, 5.8GB	500
DS2	11GB, 11.3GB	1,000
DS3	23GB, 24GB	2,000
DS4	115.8GB, 118.8GB	1,000
DS5	240.9GB, 251.2GB	2,000

TABLE II
DATASETS USED FOR SCALABILITY STUDY.



(a) Execution time (100GB). (b) Execution time (200GB).

Fig. 7. Performance of all algorithms when varying the value of k.

each job. Unless explicitly mentioned otherwise, all results on y-axis are depicted using log-scale. It should be noted that the top-k join query that we evaluated produces the top-k join tuples per join value. The final result is easily obtained by merging the top-k tuples per attribute to the final top-k join result.

B. Experimental Results

1) Scalability study: For this experiment, we created five different datasets of varying size denoted as DS1 – DS5. The respective parameters of these datasets are reported in Table II. All datasets are generated with ZI0.5 in the scoring attribute. We used $k = 10$ in this experiment.

Figure 6 reports the results of our scalability study with dataset size. In Figure 6(a), the total execution time is shown, and *RSJETLB* is 1 to 1.5 orders of magnitude better than *RSJSimple*. Also, when the size of datasets is increased, it becomes clear that the algorithm *RSJETIndex* that incorporates all our techniques outperforms the other algorithms by a large margin. For our largest setup (DS5) consisting of 0.5TB input data in total, *RSJETIndex* is two orders of magnitude faster than *RSJSimple*, and more than five times faster than *RSJETLB*. Another strong point that should be mentioned is that the gain of our algorithms compared to *RSJSimple* increases for larger datasets. This is a strong witness in favor of their scalability for large inputs.

Figure 6(b) explains the advantage of the technique that selectively loads data blocks to be processed by the job. In particular, *RSJETIndex* requires to load significantly fewer data compared to the other algorithms. In the largest setup, *RSJETIndex* loads three orders of magnitude fewer data (measured in Bytes) than all other algorithms. This is the main factor that contributes to its superiority compared to the other algorithms. In contrast, *RSJETLB* accesses almost the same data from disk as *RSJSimple*. However, it performs faster due to processing and shuffling fewer data, as a result of the early termination employed.

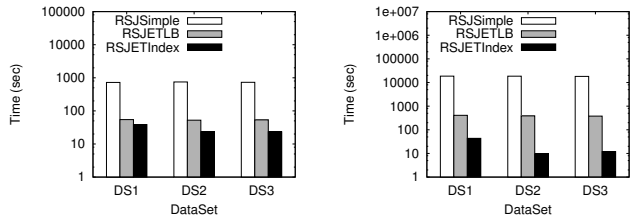
Figures 6(c) and 6(d) show that the advantage of the algorithms that use early termination is also sustained when examining the CPU processing time spent in Map and Reduce phases respectively. In particular, in Figure 6(c), we observe that the algorithms using the early termination technique are faster than *RSJSimple* because of the reduction of the number of input records in the Map phase. In addition, *RSJETIndex* is even faster as it launches even fewer Map tasks than *RSJETLB*.

This experiment is our main result, as it clearly demonstrates the performance gains of our framework compared to a baseline solution, when scalability with respect to dataset size is considered.

2) Varying the value k: Figure 7 shows the results when increasing the value of k from 10 to 500, for the two larger datasets of 100GB and 200GB. The main conclusion is that the value k does not significantly affect the total execution time of the algorithms. This is in accordance with earlier results that studied top-k joins in the literature, only it becomes more evident in our context of Big Data, since the value of k is orders of magnitude smaller than the number of records in the input tables. Therefore, we shall restrict the focus of our empirical study in other experimental parameters in the following.

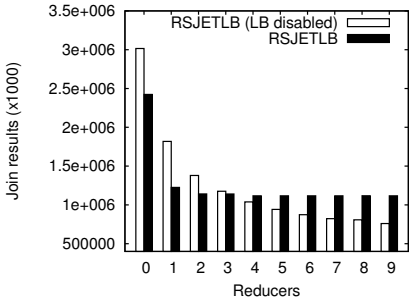
3) Varying the data distribution on score attribute: For this experiment, we use a dataset of 20GB with distinct join values 2,000 and k=10. We create three datasets DS1, DS2 and DS3, with varying distribution on the scoring attribute; DS1 is UN, DS2 is ZI0.5, and DS3 is ZI1.0.

Figure 8 shows the results when varying the data distribution on scoring attribute. Figure 8(a) depicts the overall execution time, which is reduced when increasing the skewness on the data distribution of the scoring attribute. This is due to the fact that in the case of more skewed scoring distributions, the



(a) Execution time. (b) CPU time Map.

Fig. 8. Performance of all algorithms when varying the data distribution on scoring attribute.



(a) Load per Reducer.

Fig. 9. Effect of load balancing on Reducers.

algorithms that use early termination need to process fewer input records to retrieve the top-k join result. Put differently, the computed score bounds have lower values, thus fewer records need to be processed. Figure 8(b) shows the CPU time spent in the Map phase, which follows the same trend as described above. In all cases, our algorithms significantly outperform *RSJSimple*, and among our variants *RSJETIndex* is consistently the better performing algorithm.

4) Effect of load balancing: Finally, we demonstrate the effect of load balancing by a small experiment concerning datasets of 1GB where the distribution of the join attribute is skewed (ZI0.5). Figure 9 depicts the number of join results assigned to each Reducer for two variants of algorithm *RSJETLB*: the one described in Section V-D (depicted with black bars) and one with the load balancing disabled (depicted with white bars). The latter uses only early termination and assigns Map output keys to Reducers using Hadoop's default hash-based partitioning, which essentially operates as a random partitioning. Notice that the y-axis is not in log scale. The result shows that our load balancing assigns join results to Reducers in a more uniform way, thereby allocating the work in a more fair way. Moreover, it is clear that there is one Reducer in the variant where load balancing is disabled that has more load than any Reducer of the other using load balancing. For completeness, we note that the high load imposed on Reducer 0 (white bar) is due to a single join value that has too many join results, whereas the same join value is assigned to Reducer 0 also when load balancing is disabled (black bar), but this Reducer is assigned additional join values due to the random assignment. This experiment demonstrates the benefit of employing our load balancing technique.

VIII. Conclusions

In this paper, we presented a framework for processing top-k joins in MapReduce. Distinguishing features of our algorithms include the computation of the top-k join in a fully parallelized way using a single MapReduce job, as well as their implementation on top of "vanilla" Hadoop. Our approach is based on the use of data summaries, in the form of histograms, which are computed at data upload time with zero overhead. To achieve performance gains we make use of various techniques in our framework, including early termination, load balancing, and selective access to data. Our experimental study demonstrates the efficiency of our framework over different setups.

References

[1] K. S. Candan, J. W. Kim, P. Nagarkar, M. Nagendra, and R. Yu, "RanKloud: scalable multimedia data processing in server clusters," *IEEE MultiMedia*, vol. 18, no. 1, pp. 64–77, 2011.

[2] N. Ntarmos, I. Patlakas, and P. Triantafillou, "Rank join queries in NoSQL databases," *PVLDB*, vol. 7, no. 7, pp. 493–504, 2014.

[3] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs," in *Proc. of VLDB*, 2001, pp. 281–290.

[4] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Joining ranked inputs in practice," in *Proc. of VLDB*, 2002, pp. 950–961.

[5] ——, "Supporting top-k join queries in relational databases," in *Proc. of VLDB*, 2003, pp. 754–765.

[6] K. Schnaitter, J. Spiegel, and N. Polyzotis, "Depth estimation for ranking query optimization," in *Proc. of VLDB*, 2007, pp. 902–913.

[7] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proc. of PODS*, 2001, pp. 102–113.

[8] K. Zhao, S. Zhou, and A. Zhou, "Towards efficient ranked query processing in peer-to-peer networks," in *Proc. of Cognitive Systems*, 2005, pp. 145–160.

[9] J. Liu, L. Feng, and H. Zhuge, "Using semantic links to support top-*K* join queries in peer-to-peer networks," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 15, pp. 2031–2046, 2007.

[10] C. Doulkeridis, A. Vlachou, K. Nørvåg, Y. Kotidis, and N. Polyzotis, "Processing of rank joins in highly distributed systems," in *Proceedings of ICDE*, 2012, pp. 606–617.

[11] R. Grover and M. J. Carey, "Extending map-reduce for efficient predicate-based sampling," in *Proceedings of ICDE*, 2012, pp. 486–497.

[12] N. Laptev, K. Zeng, and C. Zaniolo, "Early accurate results for advanced analytics on MapReduce," *PVLDB*, vol. 5, no. 10, pp. 1028–1039, 2012.

[13] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *Proceedings of SIGMOD*, 2010, pp. 495–506.

[14] A. Metwally and C. Faloutsos, "V-SMART-Join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors," *PVLDB*, vol. 5, no. 8, pp. 704–715, 2012.

[15] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *Proceedings of SIGMOD*, 2011, pp. 949–960.

[16] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," *PVLDB*, vol. 5, no. 10, pp. 1016–1027, 2012.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] C. Doulkeridis and K. Nørvåg, "A survey of analytical query processing in MapReduce," *VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. of OSDI'2004*, 2004, p. 10.

[20] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[21] C. Doulkeridis, A. Vlachou, P. Nikitopoulos, P. Tampakis, and M. Saouk, "The RoadRunner framework for efficient and scalable processing of big data," in *Proceedings of PCI*, 2015, pp. 215–220.