Parallel and Distributed Processing of Reverse Top-k Queries

Panagiotis Nikitopoulos[1], Georgios A. Sfyris[2], Akrivi Vlachou[3], Christos Doulkeridis[4] and Orestis Telelis[5]

Department of Digital Systems
School of Information and Communication Technologies
University of Piraeus
Piraeus, Greece
{[1]*nikp,*[4]*cdoulk,*[5]*telelis*}*@unipi.gr,* [2]*george.sfiris@gmail.com,* [3]*avlachou@aueb.gr*

***Abstract*—In this paper, we address the problem of processing reverse top-k queries in a parallel and distributed setting. Given a database of objects, a set of user preferences, and a query object q, the reverse top-k query returns the subset of user preferences for which the query object belongs to the top-k results. Although recently, the reverse top-k query operator has been studied extensively, its CPU-intensive nature results in prohibitively expensive processing cost, when applied on vast-sized data sets. This limitation motivates us to explore a parallel processing solution, to enable reverse top-k query evaluation over GBs of data in reasonable execution time. To the best of our knowledge, this is the first work that addresses the problem of parallel reverse top-k query processing. We propose a solution to this problem, called DiPaRT, which is based on MapReduce and is provably correct. DiPaRT is empirically evaluated using GB-sized data sets.**

***Index Terms*—reverse top-k, distributed, parallel**

I. Introduction

Preference-aware databases have attracted wide attention recently, due to the increased significance of personalization and ranking for real-life applications. The most well-known operator is the *top-k query*, the use of which is ubiquitous in modern systems. Given a database of objects described by a set of numerical scoring attributes and a user with a preference (scoring) function defined over these attributes, a top-k query retrieves the k objects with the best scores for the particular preference function. In the model that is widely used in related work and in practice, users express their preferences through linear top-k queries, defined by assigning a weight to each of the scoring attributes, indicating the importance of each attribute to the user. Assuming a stored set of user preferences, and a query q, the *reverse top-k query* [15] returns the subset of users for which q belongs to their top-k results.

Even though several centralized algorithms [6], [14]–[18] for reverse top-k query processing have been proposed, they typically entail prohibitively expensive processing cost, when confronted with very large (GB-sized) data sets; we demonstrate this experimentally in Section V-A. This is due to the CPU-intensive nature of reverse top-k processing. This shortcoming motivates us to explore parallel processing solutions for evaluating reverse top-k queries over large-sized input data sets.

We assume a generic parallel setting, where data is horizontally partitioned among nodes. Hence, each participating node has access to a disjoint subset of data objects and user preferences. Based on this setup, we design a parallel algorithm that consists of three phases: in the first phase, nodes perform local processing on subsets of data objects and user preferences and produce local results; in the second phase, the local results are re-partitioned and distributed to nodes in order to perform the computation entirely in parallel; lastly, in the third phase, result merging takes place in order to deliver the final result. To the best of our knowledge, the problem of parallel and distributed processing of reverse top-k queries has not been studied before.

As an application scenario, consider any popular online shop that has registered the profiles (preferences) of millions of individual users, who retrieve ranked results from millions of data items. The online shop is interested in designing a focused, personalized marketing strategy, in order to discover the subset of its customers (the reverse top-k result), which would consider buying a product of interest (the query object). Such a data analysis process is a batch processing task that is performed, after having collected the historical data of user preferences and product descriptions.

Our contributions can be summarized as follows:

- We introduce and state formally the problem of parallel and distributed reverse top-k query processing (Sect. III).
- We propose a parallel processing algorithm, called DiPaRT (Distributed and Parallel Reverse Top-k algorithm), which is based on MapReduce and computes the correct reverse top-k query result set (Sect. IV).
- We provide an implementation of DiPaRT in Hadoop, and evaluate it using large-sized data sets (Sect. V).

Also, Sect. II presents preliminary concepts, Sect. VI reviews related work, and Sect. VII concludes our study.

II. Preliminaries

Let D be an n-dimensional data space, where each dimension $i = 1, \ldots, n$ corresponds to a numerical non-negative scoring attribute. Let $S \subseteq D$ denote a set of database objects. Each $p \in S$ is a point $p = \{p[1], \ldots, p[n]\}$, where $p[i]$ is a value on dimension i.

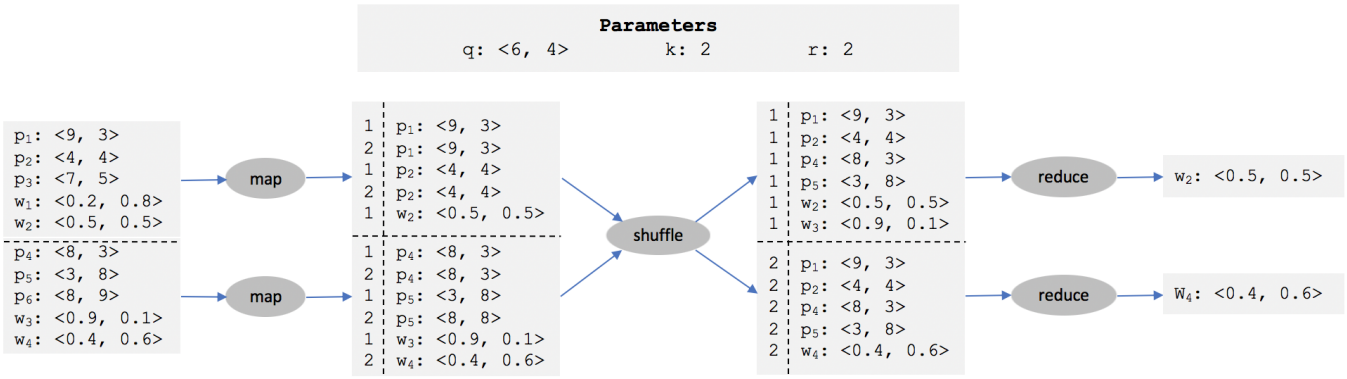


Fig. 1. Running example of DiPaRT algorithm.

A *top-k query* is defined with reference to a positive integer k and a scoring function f that aggregates the individual scores of any object into an overall score. We consider the most commonly used *weighted sum* scoring function $f_w(p) = \sum_i w[i]p[i]$, which associates a query-independent non-negative weight $w[i] \geq 0$, with each dimension i. We assume $\sum_i w[i] = 1$, as weights can be normalized, without consequence to the top-k query result set. Without loss of generality, we assume that smaller score values are preferable. The result set of a top-k query is a subset $\mathcal{T}(w,k) \subseteq S$, satisfying $|\mathcal{T}(w,k)| = k$ and $\forall p_i, p_j$ such that $p_i \in \mathcal{T}(w,k)$, $p_j \in S - \mathcal{T}(w,k)$: $f_w(p_i) \leq f_w(p_j)$. Tie-breaking may be needed for $\mathcal{T}(w,k)$ to be defined precisely, but we do not make any particular assumption with respect to it.

A *reverse top-k query* [15] identifies all weighting vectors for which a query object q belongs to the top-k result set. Formally, given a point q, a positive integer k and two data sets, S and W, of data points and weighting vectors respectively, a vector $w_i \in W$ belongs to the reverse top-k result set $\mathcal{R}(S,W,k,q)$ of q, if and only if $\exists p \in \mathcal{T}(w_i,q)$ such that $f_{w_i}(q) \leq f_{w_i}(p)$. This definition corresponds to the bichromatic version of the reverse top-k query (cf. [15]), which assumes that a set of user preferences W is provided.

The following two important properties that have been identified in previous work (among others in [15]) indicate the connection between top-k and reverse top-k queries.

Corollary 1*: Given any two points $p, p' \in S$, p dominates p', denoted as $p \prec p'$, if (1) $p[i] \leq p'[i]$ on every dimension i; (2) $p[j] < p'[j]$ on at least one dimension j. Consequently, if $p \prec p'$:*

- *$f_w(p) \leq f_w(p')$ for any weighting vector w,*
- *$\mathcal{R}(S,W,k,p') \subseteq \mathcal{R}(S,W,k,p)$ for any set W of weighting vectors.*

III. The Parallel & Distributed RTop-k Problem

In our setting, we consider two data sets S and W arbitrarily partitioned and distributed over different nodes (servers). Each server, in principle, takes as input subsets $S_i \subseteq S$ (where $S_i \cap S_j = \emptyset$, $S = \bigcup S_i$) and $W_i \subseteq W$ (where $W_i \cap W_j = \emptyset$, $W = \bigcup W_i$), and the goal is to compute the reverse top-k result $\mathcal{R}(S,W,k,q)$, while reducing the execution time through parallel processing.

The naive approach of collecting all data at a central location and performing the reverse top-k query processing using a state-of-the-art centralized algorithm [17] is prohibitively expensive. Thus, we turn our attention to parallel processing solutions. Let us consider a plain approach that computes local reverse top-k results over subsets S_i and W_i, and reports the union of the local results as the final result. In the general case, this approach fails to compute the correct reverse top-k result, since the result set may include weighting vectors that do not belong to the $\mathcal{R}(S,W,k,q)$ (false positives).

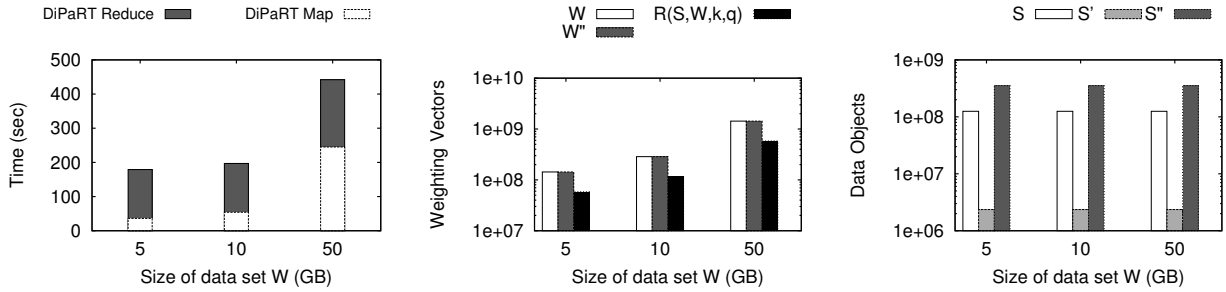
Lemma 1*: There exist instances of the Parallel and Distributed Reverse Top-k Problem wherein the union $\bigcup \mathcal{R}(S_i,W_i,k,q)$ of local reverse top-k results does not provide a correct global reverse top-k solution.*

Proof: (Sketch) Since S_i and W_i are arbitrary subsets of S and W, it is sufficient to show that there exists one partitioning for which the correct solution is not granted. Assume w for which q is ranked in the $k' > k$ position, therefore $w \notin \mathcal{R}(S,W,k,q)$. We can construct a partitioning, such that all data points that are ranked higher than q belong to S_1. Then, for $i \neq 1$ it holds that $w \in \mathcal{R}(S_i,W_i,k,q)$, therefore w is reported as a false positive. ■

IV. The DiPaRT Algorithm

A. Overview

The DiPaRT algorithm is designed as a single MapReduce job and consists of three phases: *Map*, *Shuffle* and *Reduce*. In the first phase (*Map*), each server takes as input arbitrary subsets $S_i \subseteq S$ and $W_i \subseteq W$, and computes output sets $S'_i \subseteq S_i$ and $W'_i \subseteq W_i$, by pruning unnecessary objects. In the second phase (*Shuffle*), the sets $\bigcup S'_i$ and $\bigcup W'_i$ are redistributed to the servers, using intentional assignment or replication, in order to ensure correctness during parallel processing (as indicated by Lemma 1). In the third phase (*Reduce*), each server takes as input sets S''_i and W''_i (that emerge from repartitioning of $\bigcup S'_i$ and $\bigcup W'_i$) and computes part of the reverse top-k result independently, so that a plain union of individual result sets yields the final result.



(a) Time (b) Weighting Vectors (c) Data Objects

Fig. 2. Performance of DiPaRT for various sizes of data set W, using TPC-H.

DiPaRT exploits the following properties to prune data objects and preference vectors during the *Map* phase:

- *Dominance-based pruning.* In any local data partition S_i, data points $p \in S_i$ that are dominated by q do not affect the reverse top-k result and can be safely pruned. Thus, a simple way to define S'_i is to remove from S_i those data objects dominated by q.
- *Vector pruning.* A simple way to determine W'_i from W_i is to remove all weighting vectors $w \in W_i$ that do not belong to the local reverse top-k result $\mathcal{R}(S_i, W_i, k, q)$ based on local datasets S_i and W_i. It is trivial to show that these vectors are guaranteed not to appear in the final reverse top-k result.

B. Implementation

Fig. 1 demonstrates a running example of DiPaRT algorithm for $k = 2$ and $q = \{6, 4\}$. The *Map* function of DiPaRT, takes as input subsets $W_i \subset W$ and $S_i \subset S$, the query point q, the number k and a number r indicating the number of *Reduce* tasks. It applies *dominance-based pruning* for data objects by comparing them to the query object q. In the example of Fig. 1, the data objects p_3 and p_6 are pruned in the two map tasks respectively, since they are both dominated by query point q. The surviving data objects S'_i are maintained in *Map* task's main memory, along with all the weighting vectors $w \in W_i$. As soon as the InputSplit is exhausted, DiPaRT performs *vector pruning* by using the RTA algorithm [15]. For example, in Fig. 1, the first *Map* task prunes weighting vector w_1, as it is not part of the local result set $R(S_i, W_i, k, q)$.

In technical terms, a customized InputFormat that creates InputSplits (a logical representation of a unit of input work) containing records from both data sets, is implemented. This customized InputFormat provides input data to the *Map* function, and is configured to first provide objects $p \in S_i$ and then, vectors $w \in W_i$. Also, the size of the input data sets $|W_i|+|S_i|$ is limited to be at most 128 MBs, to enable temporary storage of both sets in the main memory of a *Map* task.

In the *Shuffle* phase, DiPaRT replicates $\bigcup S'_i$ to all *Reduce* tasks, while it distributes $\bigcup W'_i$ arbitrarily to *Reduce* tasks, by exploiting the key of each intermediate tuple, as shown in Fig. 1.

In the *Reduce* phase, our goal is to avoid having a central point of merging intermediate result sets. The *Reduce* function of DiPaRT takes as input a partition of $\bigcup W'_i$ and the entire $\bigcup S'_i$, along with the value k and the query point q, and produces the local reverse top-k result, using the RTA algorithm. In the running example of Fig 1, all *Reduce* tasks take as input the entire $\bigcup S'_i$ (namely p_1, p_2, p_4, p_5) and a partition of weighting vectors (namely w_2, w_3 in the first *Reduce* task and w_4 in the second). Then, each *Reduce* task computes the local result independently. The final result is the union of these local results.

V. Experimental Evaluation

A. Limitations of Centralized Algorithms

We demonstrate the limitations of centralized reverse top-k algorithms, RTA [16] and branch-and-bound [17], when confronted with really big data sets. We implemented both algorithms in Java 7, and deployed them on a machine with a 4-core CPU running at 3.6GHz and 16GB of RAM. Both algorithms were tested with input 4-dimensional data sets of 10GB (5GB S and 5GB W) uniformly (UN) distributed, and using a reverse top-k query with k=20 that returns 40% of W as result. The branch-and-bound algorithm required 4 hours of pre-processing to build the R-Trees for both data sets S and W, plus 48 hours to report the final result set. RTA did not report the result set after 48 hours. Hence, it is clear that centralized processing is not a feasible solution in the case of massive volumes of input data. Also, these experiments demonstrate that the reverse top-k query is a costly query operator, and its parallelization makes processing large-sized data sets more practicable.

B. Evaluation of DiPaRT

DiPaRT is implemented in Java, and deployed to an in-house CDH 5.12 cluster consisting of 22 nodes with Hadoop 2.6 installed. We generated data for "Part" and "PartSupp" TPC-H tables with a scale factor of 1000. By joining these tables and keeping only numerical attributes ("size", "retailprice", "qty", "scost"), we produced a 4-dimensional 5GB data set S consisting of 125 million products p. The set of weighting vectors W that correspond to user preferences were generated following a uniform (UN) distribution. We used data sets W

of various sizes from 5 to 50 GBs. The parameter k was set to 20, and the query point was selected again to have 40% result ratio. DiPaRT was configured to run with 150 partitions (i.e., *Reduce* tasks) since this setting was identified empirically to be the best in terms of execution time.

Fig. 2 demonstrates the performance results of DiPaRT, when increasing the size of data set W. In Fig. 2a, the total length of each bar corresponds to the execution time of each experiment, and it is split to show the cost of *Map* and *Reduce* phases individually. Overall, the total processing time increases 2.5 times when using 10 times larger input sets of weighting vectors. More specifically, the cost of reading more data from disk in the *Map* phase naturally increases for larger data sets. Moreover, the reverse top-k algorithm used in the *Map* phase requires more processing time to evaluate the higher number of (input) weighting vectors. As depicted in Fig. 2b the effectiveness ratio of *vector pruning* is the same for all experiments conducted. Also, the result set size increases proportionally to the input size of data set W, since the latter follows a UN distribution. This results to increased processing cost of the *Reduce* phase of DiPaRT, for larger sizes of input data, since it needs to process more weighting vectors. Fig. 2c shows that the amount of data objects remains constant during this set of experiments: approximately 2 million objects survive the *dominance-based pruning* (indicated as S'_i) and they are replicated to all 150 partitions of our setup (indicated as S''_i).

VI. Related Work

A survey of efficient top-k query processing is presented in [8]. Monochromatic and bichromatic reverse top-k queries have been introduced in [15], [16]. Notable studies in this area include: [17] for a branch-and-bound algorithm, [7] for discovering similar products and [14] for monochromatic reverse top-k in higher dimensions. Also, multiple related topics have been studied, such as why-not questions on reverse top-k queries [5], a unified framework for rank-aware queries [2], as well as reverse k-ranks query [19] and maximum rank query [10]. Evaluation of multiple top-k queries has been studied in [6] which avoids sequential calculation of top-k queries by exploiting their common results, and in [18] which is able to calculate a reverse top-k query, but requires an index to be pre-built over the k-th ranked object of each query.

Several studies have proposed scalable processing of other preference-aware queries, that operate "on top of" Hadoop (i.e. without modifying its internal operations [3]). Our work belongs to this category. Parallel skyline and reverse skyline query evaluation is studied in [11], [12]. For top-k retrieval, *RanKloud* [1] has been proposed, which uses statistics, calculated at runtime, to compute a threshold for early termination. Parallel processing of top-k joins in MapReduce using histograms that enable early termination has been studied in [13]. In [9], algorithms for k-nearest neighbor joins in MapReduce are proposed. Ranked spatial preference queries using keywords in the context of MapReduce have been studied in [4].

VII. Conclusions

In this paper, we introduce the problem of parallel and distributed reverse top-k processing. We propose the first parallel solution for this problem, which owes its efficiency to pruning properties that reduce the amount of processed data, without sacrificing the correctness of the result. In our future work, we intend to research on more efficient pruning techniques, aiming at more efficient and scalable algorithms.

VIII. Acknowledgments

This research work has received funding from the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 1667 and under the HFRI PhD Fellowship grant (GA. no. 1059), and from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780754.

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*, 18(1):64–77, 2011.

[2] M. A. Cheema, Z. Shen, X. Lin, and W. Zhang. A unified framework for efficiently processing ranking related queries. In *Proceedings of EDBT.*, pages 427–438, 2014.

[3] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[4] C. Doulkeridis, A. Vlachou, D. Mpestas, and N. Mamoulis. Parallel and distributed processing of spatial preference queries using keywords. In *Proceedings of EDBT*, pages 318–329, 2017.

[5] Y. Gao, Q. Liu, G. Chen, B. Zheng, and L. Zhou. Answering why-not questions on reverse top-k queries. *PVLDB*, 8(7):738–749, 2015.

[6] S. Ge, L. H. U, N. Mamoulis, and D. W. Cheung. Efficient all top-k computation: A unified solution for all top-k, reverse top-k and top-m influential queries. *IEEE TKDE*, 25(5):1015–1027, 2013.

[7] K. Georgoulas, A. Vlachou, C. Doulkeridis, and Y. Kotidis. User-centric similarity search. *IEEE Trans. Knowl. Data Eng.*, 29(1):200–213, 2017.

[8] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):1–58, 2008.

[9] W. Kim, Y. Kim, and K. Shim. Parallel computation of k-nearest neighbor joins using mapreduce. In *Proceedings of BigData*, pages 696–705, 2016.

[10] K. Mouratidis, J. Zhang, and H. Pang. Maximum rank query. *PVLDB*, 8(12):1554–1565, 2015.

[11] Y. Park, J. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using MapReduce. *PVLDB*, 6(14):2002–2013, 2013.

[12] Y. Park, J. Min, and K. Shim. Efficient processing of skyline queries using mapreduce. *IEEE Trans. Knowl. Data Eng.*, 29(5):1031–1044, 2017.

[13] M. Saouk, C. Doulkeridis, A. Vlachou, and K. Nørvåg. Efficient processing of top-k joins in mapreduce. In *Proceedings of BigData*, pages 570–577, 2016.

[14] B. Tang, K. Mouratidis, and M. L. Yiu. Determining the impact regions of competing options in preference space. In *Proceedings of SIGMOD*, pages 805–820, 2017.

[15] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Proceedings of ICDE*, pages 365–376, 2010.

[16] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE TKDE*, 23(8):1215–1229, 2011.

[17] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of SIGMOD*, pages 481–492, 2013.

[18] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-*k* queries. In *Proceedings of SIGMOD*, pages 397–408, 2012.

[19] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *PVLDB*, 7(10):785–796, 2014.