Distributed and Parallel Databases manuscript No.
(will be inserted by the editor)

Pruning Techniques for Parallel Processing of Reverse Top-k Queries

Panagiotis Nikitopoulos · Georgios A. Sfyris · Akrivi Vlachou · Christos Doulkeridis · Orestis Telelis

Received: date / Accepted: date

Abstract In this paper, we address the problem of processing reverse top-k queries in a parallel 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 scalable parallel processing solution, in order to enable reverse top-k processing over distributed large sets of input data in reasonable execution time. We present an algorithmic framework for the problem, in which different algorithms can be instantiated, targeting a generic parallel setting. We describe a parallel algorithm (DiPaRT) that exploits basic pruning properties and is provably correct, as an instantiation of the framework. Furthermore, we introduce novel pruning properties for the problem, and propose DiPaRT+ as another instance of the algorithmic framework, which offers improved efficiency and scales gracefully. All algorithms are implemented in MapReduce, and we provide a wide set of experiments that demonstrate the improved efficiency of DiPaRT+ using data sets that are four orders of magnitude larger than those handled by centralized approaches.

1 Introduction

Preference-aware databases [16, 5, 13] 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*, whose use is

P. Nikitopoulos[1], G. A. Sfyris[2], A. Vlachou[3], C. Doulkeridis[4], O. Telelis[5]
Department of Digital Systems
School of Information and Communication Technologies
University of Piraeus, 185 34, Greece
E-mail: {nikp[1], cdoulk[4], telelis[5]}@unipi.gr, george.sfiris@gmail.com[2], avlachou@aueb.gr[3]

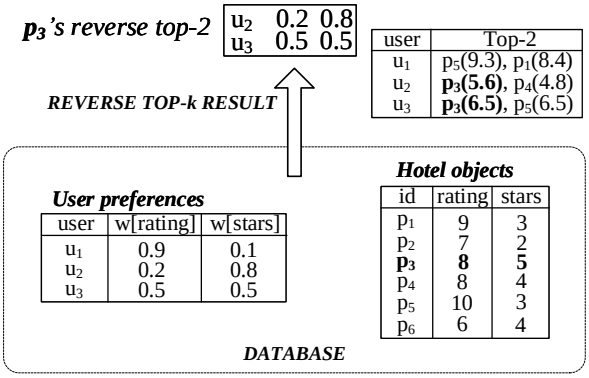


Fig. 1 Example of reverse top-k queries.

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 [5,13,11,14,26,27,29] and in practice, the users express their preferences through linear top-k queries, which are 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* [26] returns the subset of users for which q belongs to their top-k results.

Consider for example, a database containing information about different hotels as well as user preferences, as depicted in Figure 1. For each of the six hotels, the rating and the number of stars are recorded, and maximum values on each attribute are preferable (in the rest of this paper, minimum values will be preferable, without loss of generality). The database also stores the preferences of three users (u_1, u_2, and u_3) in terms of weights on each attribute. User u_1 prefers hotels with high rating values, whereas u_2 is interested in hotels with many stars. User u_3 is indifferent or values equally rating and stars. Also, the reverse top-2 result of hotel p_3 is depicted, which consists of users u_2 and u_3. The reason is that p_3 appears in the top-2 results of both these users.

Even though several centralized algorithms [11,24,26–29] for reverse top-k query processing have been proposed, they typically entail prohibitively expensive processing cost, when confronted with very large data sets, also demonstrated by empirical results in Section 7.1. This is due to the CPU-intensive nature of reverse top-k processing. Furthermore, these studies do not deal with challenges posed by using distributed input sets of data, such as reducing the communication overhead. These shortcomings motivate us to explore parallel processing solutions that scale gracefully with the size of

underlying data and can be used in practice when using the reverse top-k query as a data analysis tool for distributed large data sets.

In previous work [18], we introduced the problem of scalable reverse top-k processing and presented a first solution. In this paper, we present an algorithmic framework for solving the problem, and provide new insights in terms of pruning properties, leading to more efficient solutions. We assume a generic parallel setting, where data is stored horizontally partitioned to nodes. Hence, each participating node has access to a disjoint subset of data objects and user preferences. Based on this setup, we present theoretical properties that allow effective pruning of input data in a parallel way, while also guaranteeing the correctness of the result. Capitalizing on these properties, we design a generic algorithmic framework 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 *repartitioned* and distributed to nodes in order to perform the computation in a completely parallel way; while, in the third phase, *result merging* takes place in order to deliver the final result. Our algorithms can be seen as instances of this framework.

As an application scenario, consider any popular online shop (e.g., Amazon, eBay, etc.) that has registered the profiles of millions of individual users, who search for a specific item to buy among millions of products (e.g. books). User preferences may be retrieved by tracking user activity on the online shop, such as search criteria. Multiple preference vectors might correspond to a single user profile, based on her preferences on a specific time, thus leading to increased amount of data. Preference vectors, as well as data items (product data), are stored distributed in several available servers, in an arbitrary way. The online shop would be 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). The goal of such data analysis process would be to (a) analyze the result in a parallel way, to reduce the processing time required by centralized reverse top-k algorithms, and (b) reduce the communication cost entailed by transferring large sets of data over the network. This process is a batch processing task that is performed after having collected the historical data of user preferences and product descriptions[1].

This paper is an extended version of [18]. The new contributions of this paper are summarized as follows:

- We propose an algorithmic framework for solving the problem of parallel reverse top-k query processing (Section 2.3).
- We present DiPaRT algorithm [18] as an instantiation of our framework, prove its correctness (Section 6.1), and provide its complexity analysis (Section 6.2).

[1] We explicitly state that our work targets offline processing of reverse top-k queries based on all available data objects and user preferences at a given time point.

- We extend a pruning technique (denoted as BB) for reverse top-k query processing to be applicable in the parallel version of the problem (Section 4.1), and present a novel pruning technique (denoted as WV) that leads to improved effectiveness and performance (Section 4.2).
- We introduce and analyze DiPaRT+, an efficient and scalable parallel algorithm, as a second instantiation of our framework, which can be parameterized with a pruning technique to achieve efficiency and scalability (Section 5). We also provide the complexity analysis of DiPaRT+ (Section 6.3).
- We implement our algorithms in MapReduce [7], and demonstrate their merits in terms of efficiency and scalability by means of large-scale experiments under a wide variety of settings (Section 7).

The rest of this paper, is organized as follows: Section 2 describes preliminary concepts along with the problem statement and our generic algorithmic framework for solving the problem. Section 3 presents DiPaRT [18] as instantiation of our framework. Section 4 introduces a set of advanced techniques (BB and WV) for efficiently pruning both data objects and weighting vectors. Section 5 presents DiPaRT+ algorithm as a second instantiation of our framework. Section 6 provides the theoretical analysis of our algorithms. Section 7 demonstrates our experimental evaluation. Section 8 reviews related work and finally, Section 9 concludes our study.

2 Preliminaries & Problem Statement

In this section, we present the preliminary concepts and formally state the problem of parallel reverse top-k processing

2.1 Notation and Definitions

Let D be an n-dimensional data space, where each dimension $i = 1, \ldots, n$ corresponds to a numerical non-negative scoring attribute. Denote by $S \subseteq D$ 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. Without loss of generality, we assume that smaller score values are preferable. Table 1 summarizes our notation.

A *top-k query* is defined with reference to a positive integer k and a scoring function f, that aggregates an object's individual scores 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. 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.

Table 1 Overview of symbols.

Symbols	Description
D	Data space
n	Data dimensionality
$S = \bigcup S_i$	Data set of objects
$W = \bigcup W_i$	Data set of weighting vectors
$p[i]$	Value of point p on dimension i
q	n-dimensional query point
w	Weighting vector
$\mathcal{T}(w,k)$	Result of top-k query defined by vector w
$\mathcal{R}(S,W,k,q)$	Reverse top-k result set
S'_i	Non-dominated objects by q in data fragment S_i
W'_i	Reverse top-k result on S'_i and W_i ($\mathcal{R}(S'_i,W_i,k,q)$)
S''_i	Subset of data objects $S_i \subseteq \bigcup S'_j$
W''_i	Partition of the set of vectors $\bigcup W'_j$
$[l,u]$	A bounding box with lower corner l and upper corner u (BB pruning)
b_j	Vertex j of the bounding polytope b (WV pruning)

A *reverse top-k query* [26] 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. [26]), which assumes that a set of user preferences W is provided.

A commonly used concept in several preference-aware queries is the dominance property. A point $p \in S$ *dominates* another point $p' \in S$, 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. Corollary 1 has been identified in previous work (among others, in [26]) and is derived from the dominance property.

Corollary 1 *Given any two points $p, p' \in S$, 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.*

2.2 Problem Statement

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 $S_i \subseteq S$ ($S_i \cap S_j = \emptyset$, $S = \bigcup S_i$) and $W_i \subseteq W$ ($W_i \cap W_j = \emptyset$, $W = \bigcup W_i$), 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.

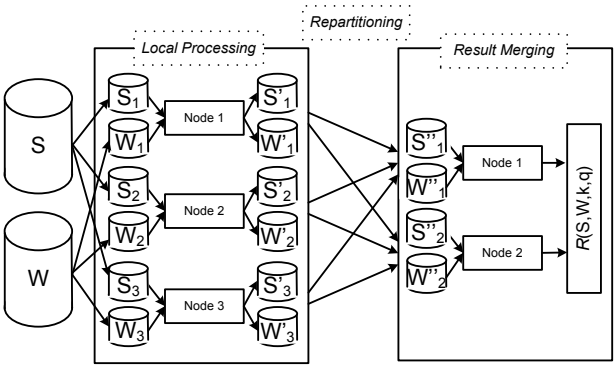


Fig. 2 Parallel reverse top-k query processing.

Problem 1 (Parallel Distributed Reverse Top-k Problem) Given two data sets S and W which are horizontally partitioned in an arbitrary way and distributed over different servers, compute the reverse top-k result: $\mathcal{R}(S, W, k, q)$.

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 [28] 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. As already discussed in [18], 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).

2.3 Algorithmic Framework and Research Challenges

Figure 2 provides an overview of the proposed algorithmic framework. In the first phase (*local processing*), 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 (*repartitioning*), 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. In the third phase (*result merging*), each server takes as input sets S_i'' and W_i'' (practically repartitions of sets $\bigcup S_i'$ 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.

Consequently, the parallel reverse top-k problem entails two grand challenges. The *first challenge* is to determine the output sets $S_i' \subseteq S_i$ and $W_i' \subseteq W_i$ such that they are sufficient to compute the correct result set. In other words, S_i' and W_i' need to satisfy the following property: $\mathcal{R}(\bigcup S_i', \bigcup W_i', k, q) = \mathcal{R}(S, W, k, q)$. The *second challenge* is to determine the input sets S_i'' and W_i'' for each server, in such a way that (a) $\bigcup \mathcal{R}(S_i'', W_i'', k, q) = \mathcal{R}(S, W, k, q)$, and

(b) the amount of data exchanged over the network is minimized. Note that the crucial difference of S_i and W_i compared to S_i'' and W_i'' is that the former sets are arbitrary partitions of S and W, whereas S_i'' and W_i'' are partitioned deliberately and computed in parallel on the second phase, in order to ensure the correctness of the final result.

3 The DiPaRT Algorithm

In this section, we present the DiPaRT algorithm [18], as an instantiation of our framework, which exploits the pruning properties outlined next.

3.1 Simple Pruning

The following properties determine subsets of S and W that are sufficient for computing $\mathcal{R}(S, W, k, q)$. The basic idea of DiPaRT is to replicate $S_i' = \{p | p \in S_i \text{ and } q \not\prec p\}$ to all *Reduce* tasks. Then, an arbitrary distribution of local results $W_i' = \mathcal{R}(S_i', W_i, k, q)$ to Reduce tasks is sufficient to provide the correct result.

Dominance-based pruning. According to Corollary 1, 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 data sets 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.

3.2 Local Processing and Pruning

Algorithm 1 presents the *Map* function of DiPaRT. It 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. DiPaRT applies dominance-based pruning for data objects (Section 3.1) by comparing them to the query object q (line 5). The surviving data objects S_i' are maintained in *Map* task's main memory (line 6), along with the weighting vectors $w \in W_i$ (lines 11– 13). As soon as the InputSplit is exhausted, DiPaRT performs vector pruning (Section 3.1) by using the RTA algorithm [26].

In technical terms, DiPaRT implements a customized InputFormat that creates InputSplits (a logical representation of a unit of input work) containing records from both data sets. 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.

Algorithm 1: DiPaRT: Map phase

```
1: Input: S_i, W_i, q, k, r: number of Reduce tasks
2: Output: S'_i, W'_i
3: function MAP(x: input tuple)
4: if x is a data object p then
5:    if p is not dominated by q then
6:       add p to memory S'_i
7:       for j in [1, ..., r] do
8:          output ⟨j, p⟩
9:       end for
10:   end if
11: else {x is a weighting vector w}
12:   add w to memory W_i
13: end if
14: if no more input tuples then
15:   W'_i ← RTA(S'_i, W_i, q, k)
16:   for w ∈ W'_i do
17:      j ← j + 1
18:      output ⟨j%r, w⟩
19:   end for
20: end if
21: end function
```

3.3 Result Merging

In the Result Merging phase, one solution would be to have a single node that computes the final result $\mathcal{R}(\bigcup S'_i, \bigcup W'_i, k, q) = \mathcal{R}(S, W, k, q)$ in a centralized fashion, which is straightforwardly correct. However, our goal is to avoid having a central point of merging intermediate result sets. The *Reduce* function of DiPaRT (Algorithm 2) 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. The final result is the union of these local results.

DiPaRT, however, also comes with a shortcoming: it needs to transfer (*shuffle*) the set $\bigcup S'_i$ multiple times over the network, which incurs high communication cost and also increases the time spent for shuffling. To overcome this limitation, in Section 4 we introduce a set of more sophisticated pruning techniques which efficiently eliminate both data objects and weighting vectors.

4 Advanced Pruning Techniques

In this section, we present our advanced *pruning* techniques that avoid the replication of $\bigcup S'_i$ and reduce the size of W''_i. Interestingly, these techniques enable *parallel merging*, i.e., each group of vectors can be processed in parallel, and a plain union of the results, outputs the correct final result.

Algorithm 2: DiPaRT: Reduce phase

Input: $S''_i = \bigcup S'_j$, $W''_i \subseteq \bigcup W'_j$, k, q
Output: $\mathcal{R}(S'', W'', k, q)$
function $REDUCE(key, V$: Set of values)
for $x \in V$ **do**
  if x is a data object p **then**
    add p to memory S''_i
  else {x is a weighting vector w}
    add w to memory W''_i
  end if
end for
$\mathcal{R}(S'', W'', k, q) \leftarrow RTA(S''_i, W''_i, q, k)$
output $\mathcal{R}(S'', W'', k, q)$
end function

4.1 Pruning Data Objects based on Bounding Boxes

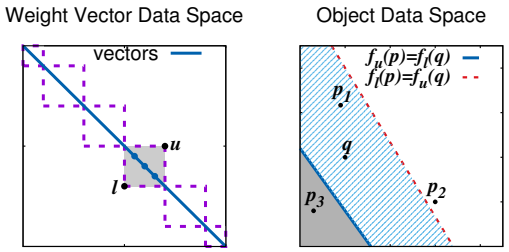
Consider a grouping of weighting vectors that (a) creates r groups $\mathcal{P} = \{W''_1, \ldots, W''_r\}$ that form non-overlapping partitions (i.e., the groups W''_i are disjoint), and (b) the partitions cover the entire weighting vectors space. Given a specific partition W''_i, we can represent it using a box with lower-left corner l and upper-right corner u. The l and u corners are considered as pseudo-vectors, since $\sum_i l[i] < 1$ and $\sum_i u[i] > 1$. The l and u pseudo-vectors of partition W''_i correspond to the enclosing minimum and maximum coordinate values, respectively. Formally, given a partition W''_i, its pseudo-vectors l and u can be calculated as follows: $l = \{min(W''_i[1]), min(W''_i[2]), ..., min(W''_i[n])\}$ and $u = \{max(W''_i[1]), max(W''_i[2]), ..., max(W''_i[n])\}$. The score of any data object $p \in S$ and for any vector $w \in W''_i$ is bound by the values: $f_l(p) \leq f_w(p) \leq f_u(p)$ [28].

The bounds are used to identify groups of vectors W''_i for which q is always ranked higher than p, thus avoiding replication *for a particular group* W''_i. Even though any random grouping of the weighting vectors is sufficient for applying BB pruning (i.e., does not affect the correctness), the efficiency may vary based on the size of the selected box.

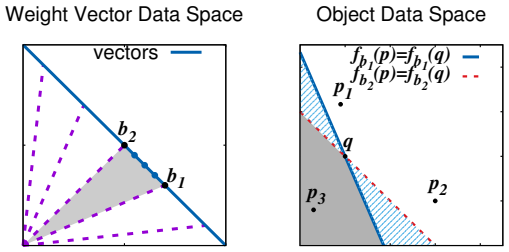
In practice, this grouping can be achieved by partitioning the weight vector space using a *grid-based partitioning* scheme. An example of grid-based partitioning is depicted in Figure 3a for the 2D vector space, where the weight vectors are located on the line $w[1] + w[2] = 1$. For every partition, we can derive a 2D box by calculating the pseudo-vectors l, u. In Figure 3a the grayed box contains a single partition of the vector space, and points l, u correspond to lower and upper corners of the grayed partition respectively.

By exploiting the aforementioned bounding values $f_l(p) \leq f_w(p) \leq f_u(p)$ to the parallel version of the problem, we propose *BB memory-less* and *BB memory-based* techniques for pruning data objects based on a partition W''_i.

BB Memory-less pruning. The following lemma provides a condition for pruning individual data objects for a group of vectors, independently of previously examined data objects.



(a) Grid Partitioning (b) BB Memory-less Pruning



(c) Angle Partitioning (d) WV Pruning

Fig. 3 Pruning examples.

Lemma 1 *Given a reverse top-k query q, a data object p, and a group $W_i'' \in \mathcal{P}$ of weighting vectors represented by a box $[l, u]$, p does not affect the reverse top-k result of q (or q precedes p) for any vector $w \in W_i''$ and can be safely pruned, if: $f_l(p) \geq f_u(q)$.*

Proof Let w denote any vector that belongs to W_i'', thus it is enclosed in the box $[l, u]$ of W_i''. It suffices to show that p has a worse score than q for any such vector w, i.e., $\forall w \in W_i'' : f_w(p) \geq f_w(q)$. Since w in enclosed in $[l, u]$, it holds that $f_w(p) \geq f_l(p)$. It is given that $f_l(p) \geq f_u(q)$, thus we derive that $f_w(p) \geq f_u(q)$. Again, due to w being enclosed in $[l, u]$, it holds that $f_u(q) \geq f_w(q)$. Consequently, it also holds that $f_w(p) \geq f_w(q)$.

Intuitively, this bounding inequality is illustrated in Figure 3b as the dashed line $f_l(p) = f_u(q)$. This line splits the object data space in two parts; the objects that fall in the white part of the space (e.g. p_2) do not affect the ranking of q for all $w \in W_i''$, thus can be safely pruned for W_i''.

BB Memory-based pruning. For each partition W_i'', we maintain a buffer of data objects of length k. Each time a data object p is processed, we check the contents of the buffer and if it contains k data objects $\{p_1, \ldots, p_k\}$ such that the k-th higher value $f_u(p_k)$ is better than the best possible value

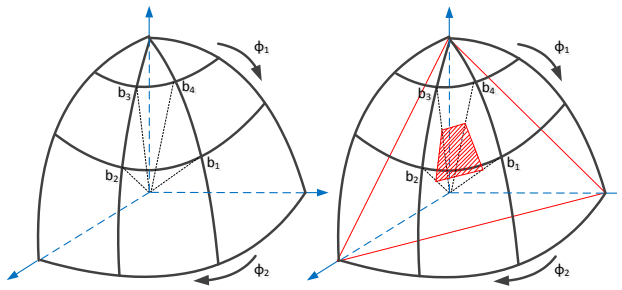


Fig. 4 Angle-based partitioning of space in 3D.

$f_l(p)$ of p, then p can be pruned, as it does not affect the rank of q for W_i''. The buffer is sorted based on increasing $f_u(p_i)$ values, thus the highest value $f_u(p_k)$ corresponds to the k-th object in the order. Also, the buffer is updated if an incoming data object p has a better value $f_u(p)$ than the k-th value $f_u(p_k)$ of the buffer. In this case, p_k is evicted from the buffer, and p is inserted in the buffer. The following lemma establishes the correctness of memory-based pruning.

Lemma 2 *Given a reverse top-k query q, a data object p and a group $W_i'' \in \mathcal{P}$ of weighting vectors represented by a box $[l, u]$, p does not affect the reverse top-k result of q for any vector $w \in W_i''$, if: $\exists\{p_1, \ldots, p_k\} \in S$, such that $\forall i \in [1, k] : f_u(p_i) \leq f_l(p)$.*

Proof Let w denote any vector that belongs to W_i'', thus it is enclosed in the box $[l, u]$ of W_i''. It suffices to show that there exist k other objects with better score than p for any such vector w, i.e., $\exists\{p_1, \ldots, p_k\} \in S$, such that $\forall i \in [1, k] : f_w(p_i) \leq f_w(p)$. Since w in enclosed in $[l, u]$, it holds that $f_w(p) \geq f_l(p)$. It is given that k objects $\{p_1, \ldots, p_k\}$ do exist such that $f_w(p) \geq f_l(p) \geq f_u(p_i)$. For any such point p_i, it also holds that $f_u(p_i) \geq f_w(p_i)$, because w is enclosed in $[l, u]$. Therefore, we derive that there exist k objects $\{p_1, \ldots, p_k\}$ such that $f_w(p) \geq f_w(p_i)$.

4.2 Pruning Data Objects based on Weighting Vectors

BB pruning relies on the use of pseudo-vectors l and u. A shortcoming is that these pseudo-vectors are worst-case approximations of vectors in W_i'', since they are defined by the enclosing minimum and maximum coordinate values respectively. As such, although using the bounding box $[l, u]$ provides cheap pruning, it does not provide the means for maximizing the bounding tightness.

Motivated by this shortcoming, we propose novel tighter bounds. Since these bounds rely on actual bounding vectors, we first need to employ a different partitioning scheme of the weight vector space, namely *angle-based partitioning*. By using angular coordinates on the weight vector space, we can

partition the hyperplane $\sum_{i=1}^{n} w[i] = 1$ in convex polytopes. A convex polytope W_i'' is defined by vectors $b_1, b_2, ..., b_{2^{n-1}}$, whose projections on the plane of the weighting vectors are the vertices of the convex polygon.

An example of this partitioning scheme is illustrated in Figure 4 for the case of a 3-dimensional weight vector space. Given that the weighting vectors in this example are located on the plane $w[1] + w[2] + w[3] = 1$, each angle partition is defined by four vectors b_j, $j = 1, 2, 3, 4$, whose projections on the plane of the weighting vectors are the vertices of a convex polygon, enclosing a group W_i'' of weighting vectors. For the case of 2D weight vector space, an example is also depicted in Figure 3c, where the vectors are located on the line $w[1] + w[2] = 1$. The triangles define line segments that represent partitions of the 2D vector space, while vectors b_1, b_2 are the vertices of the polytope defining the grayed partition. This approach can be generalized and applied in $n > 3$ dimensions too, at the expense of having 2^{n-1} vectors describing the convex polytope.

The principle of *WV pruning* is portrayed by Lemma 3.

Lemma 3 *Let $\mathcal{P}$ denote a convex polytope on the hyperplane $\sum_{i=1}^{n} w[i] = 1$, defined by the preference vectors. For any two points $p, q \in S$, if $f_v(q) \leq f_v(p)$ holds for every vertex v of the polytope $\mathcal{P}$, then $f_w(q) \leq f_w(p)$ holds for every point of $\mathcal{P}$ (q precedes p).*

Proof 1 *Let $w \in \mathcal{P}$ be any point of the polytope. By Carátheodory's theorem (see e.g., [22]), w can be written as the convex combination of at most $n + 1$ vertices of $\mathcal{P}$: that is, there exists a subset $\mathcal{V} = \{v_1, v_2, \ldots, v_d\}$ of $d \leq n + 1$ vertices of $\mathcal{P}$, such that $w = \sum_{j=1}^{d} \lambda_j v_j$, for λ_j,$j = 1, \ldots, d$, satisfying $\sum_{j=1}^{d} \lambda_j = 1$. Then, if $v_j \cdot q \leq v_j \cdot p$ for every $v_j \in \mathcal{V}$, we have $\lambda_j(v_j \cdot q) \leq \lambda_j(v_j \cdot p)$, thus, also, $(\lambda_j v_j) \cdot q \leq (\lambda_j v_j) \cdot p$; summing these latter inequalities over $j = 1, \ldots, d$ yields $w \cdot q \leq w \cdot p$.*

In order to apply Lemma 3, we can simply use the actual vectors b_j obtained by angle-based partitioning, since the projection of each b_j on the plane of the weighting vectors is just $\rho_j b_j$, for some scalar ρ_j. Notice however, that Lemma 3 does not imply the use of a specific partitioning scheme. Hence, WV pruning may exploit any provided non-overlapping angular based partitioning scheme. For example, if the distribution of vectors was available, (e.g., by sampling), the angles could be defined in a way that each partition has a similar number of containing vectors.

Example 1 Intuitively, an example of Lemma 3 in 2D data object space is depicted in Figure 3d, where the lines $f_{b_1}(p) = f_{b_1}(q)$ and $f_{b_2}(p) = f_{b_2}(q)$ defined by partition $W_i'' = [b_1, b_2]$ split the space to three parts; the objects of the white part (e.g., p_1, p_2) can be safely pruned for W_i''. Notice that in comparison to the BB memory-less pruning example (Figure 3b), the volume of the white space is significantly higher for the WV pruning example (Figure 3d), although the partitions $W_i'' = [l, u]$ and $W_i'' = [b_1, b_2]$ are deliberately chosen to be the same: b_1 and b_2 are actually the upper left and lower right corners

of the bounding box $[l, u]$ respectively. In this particular example, the point p_1 can be pruned only by using WV pruning on W_i'', thus demonstrating the effectiveness of this technique.

4.3 Early Termination

Given a partition of weighting vectors W_i'', if there exist at least k data objects p which are ranked higher than q for every $w \in W_i''$ (p precedes q), then this partition has an empty result set. This useful property can be exploited for terminating early the processing of a particular partition W_i'', as long as we have found at least k data objects preceding q. By terminating early the processing of a particular partition, we are able to efficiently perform both data object and weighting vector pruning seamlessly, at no processing cost.

Early termination can be easily achieved by extending either of the previously proposed pruning techniques: for BB pruning we need to count the number of data objects which satisfy $f_u(p) < f_l(q)$, while for WV pruning we need to count the number of data objects which satisfy $f_{b_j}(p) < f_{b_j}(q)$ for all vertices b_j of a convex polytope.

Intuitively, the above property is illustrated in Figures 3b and 3d as the grayed area of the object data space. The data objects in this area (e.g., p_3) precede q and contribute to the counter associated with W_i'' partition. Notice once more, that WV pruning results in a significantly larger grayed area. The shaded area corresponds to points p whose rank with respect to q cannot be determined. WV pruning reduces the space of undetermined ranking, and increases the space corresponding to better (and worse) ranked points than q. In turn, this increases the efficiency of our approach.

5 The DiPaRT+ Algorithm

In this section, we introduce the DiPaRT+ algorithm that capitalizes on the advanced properties introduced in Section 4. In fact, DiPaRT+ is parameterized by the pruning technique employed, hence it can operate using either of the techniques described in Sections 4.1 or 4.2.

5.1 Local Processing and Pruning

The *Map* function of DiPaRT+ is depicted in Algorithm 3. It takes as input subsets $S_i \subset S$, $W_i \subset W$, the query point q, the number k and a partitioning scheme $\mathcal{P}$ of the weighting vector space.

If the input tuple is a data object p (line 4), we check if p is dominated by q, in order to quickly prune it (line 5). If this is unsuccessful, we examine the partitions of $\mathcal{P}$ and try to avoid redistributing p to every partition, by applying either BB or WV pruning.

Algorithm 3: DiPaRT+: Map phase

```
1: Input: S_i, W_i, q, k, P = {W''_1, ..., W''_r}
2: Output: S'_i, W'_i
3: function MAP(x: input tuple)
4: if x is a data object p then
5:    if p is not dominated by q then
6:       for W''_i ∈ P = {W''_1, ..., W''_r} do
7:          if earlyTerminationCounter[i] ≥ k then
8:             continue {prune p}
9:          else if q precedes p for W''_i then
10:            continue {prune p}
11:         else if p precedes q for W''_i then
12:            earlyTerminationCounter[i]++
13:            output ⟨(i, 1), p⟩
14:         else
15:            output ⟨(i, 2), p⟩
16:         end if
17:      end for
18:   end if
19: else {x is a weighting vector w}
20:   i ← {i|w ∈ W''_i and P = {W''_1, ..., W''_r}}
21:   output ⟨(i, 3), w⟩
22: end if
23: end function
```

To this end, we first try to prune p using the Early Termination technique of Section 4.3 (line 7): for the currently examined partition W_i'', we check the associated counter which keeps track of the number of objects preceding q and if it is greater than or equal to k, then we prune p (line 8).

If the above was unsuccessful, we check if q precedes p for W_i'' (line 9). For WV pruning, this can be achieved by checking if $f_{b_j}(q) \leq f_{b_j}(p)$ for all vertices b_j of polytope W_i'', while for BB memory-less pruning this translates to checking if $f_u(q) \leq f_l(p)$ for the partition $W_i'' = [l, u]$. When using the BB pruning technique, we also apply BB memory-based pruning during this step, by checking if the k-th object p_k of the associated buffer is better than $f_l(p)$ (i.e. $f_u(p_k) \leq f_l(p)$). In the case any of the above holds true, we are able to prune p for W_i'' (line 10).

If we were not able to prune p, we check if p precedes q for partition W_i'' (line 11). For WV pruning this can be examined by checking if $f_{b_j}(p) < f_{b_j}(q)$ for all vertices b_j of polytope W_i'', while for BB pruning we simply check if $f_u(p) < f_l(q)$. In case the check returns true, we increment the associated Early Termination counter (line 12) and output the data object. We use a composite output key consisting of the partition identifier i of W_i'' and the tuple identifier 1, which facilitates the early termination of *Reduce* task i (line 13). If none of the above holds, we just redistribute data object p to *Reduce* task i using the tuple identifier 2 (line 15).

In case the input tuple is a vector w (line 19), we determine the partition identifier i of $W_i'' \in \mathcal{P}$ that encloses w (line 20). The output composite key (line 21) consists of this partition identifier and the tuple identifier 3, which

Algorithm 4: DiPaRT+: Reduce phase

```
1: Input: S_i'', W_i'', q, k
2: Output: R(S, W, k, q)
3: function REDUCE(key, V: Set of values)
4: for x ∈ V do
5:    if x.tupleIdentifier = 1 then {data object x precedes q}
6:       k = k - 1
7:       if k = 0 then {early termination}
8:          exit
9:       end if
10:   else if x.tupleIdentifier = 2 then {x is a data object}
11:      add x to memory S_i''
12:   else if x.tupleIdentifier = 3 then {x is weighting vector}
13:      if w ∈ Res|Res : RTA(S_i'', w, q, k) then
14:         output ⟨w⟩
15:      end if
16:   end if
17: end for
18: end function
```

indicates the tuple as a weighting vector. Notice that, contrary to DiPaRT, DiPaRT+ does not use reverse top-k query processing to prune vectors in the *Map* phase. Instead, our premise is to benefit by the significantly higher efficiency of the Early Termination technique described in Section 4.3, to prune weighting vectors in the *Reduce* phase, effectively, without any processing cost.

The composite intermediate key consists of two parts: the partition identifier and the tuple identifier. Essentially, the partitioning of intermediate results, is achieved by implementing a customized Partitioner, which only takes into account the partitioning identifier of the composite key. The tuple identifier is exploited by a customized Comparator to apply secondary sorting on the intermediate tuples during the *Shuffle* phase. In the *Reduce* phase, the goal is to have data objects that precede q (tagged with tuple identifier 1) arrive prior to other data objects (tagged with tuple identifier 2), which in turn should arrive prior to candidate weighting vectors (tagged with tuple identifier 3).

5.2 Result Merging

The *Reduce* function of DiPaRT+ is depicted in Algorithm 4 which performs the result merging. The inputs for *Reduce* task i are the intermediate data objects and weighting vectors which have a partition identifier value of i. Due to the secondary sorting, the data objects that precede q are processed first. If k such objects exist (line 7), we can safely deduce that the result of the current partition (and *Reduce* task) is empty and terminate its processing immediately, as discussed in Section 4.3. Notice that in such a case, the *Reduce* task will terminate without performing any additional computation on the remaining tuples. If the *Reduce* task does not terminate early, the remaining data objects

p are accessed and maintained in memory. Then, the candidate weighting vectors are accessed one by one to determine the result set $\mathcal{R}(S_i'', W_i'', k, q)$ using the RTA algorithm [26]. Also notice that the k value might be smaller than its initial value, due to the decrement that might have occurred in line 6.

6 Correctness and Complexity Analysis

In this section, we initially prove the correctness of DiPaRT algorithm. Then, we provide a complexity analysis for both DiPaRT and DiPaRT+, as an analytical comparison between the two approaches.

6.1 Correctness of DiPaRT

In the following, we prove that if every data object is replicated to all servers $S_i'' = \bigcup S_i'$, and $W_i'' \subset \bigcup W_i'$ is an arbitrary disjoint subset of $\bigcup W_i'$, then the correct result set can be produced. This is because the computation of each $w \in W$ is independent of other $w' \in W$ $(w \neq w')$, but still requires all relevant data objects.

Lemma 4 *The pruned sets $\bigcup W_i'$ and $\bigcup S_i'$ determined at the Map phase of DiPaRT are sufficient to produce the correct reverse top-k result set in the Reduce phase, if (1) $S_i'' = \bigcup S_i'$ and (2) $W_i'' \subset \bigcup W_i'$ and $\bigcup W_i'' = \bigcup W_i'$, i.e.: $\bigcup \mathcal{R}(S_i'', W_i'', k, q) = \mathcal{R}(S, W, k, q)$.*

Proof 2 *(Sketch) Initially, let us assume that $\exists w \in \mathcal{R}(S, W, k, q)$ such that $w \notin \bigcup \mathcal{R}(S_i'', W_i'', k, q)$. There are two cases: (a) if $w \in W_i''$, then there exist k objects in $S_i'' = \bigcup S_i'$ that are ranked higher than q w.r.t. w. Since $S_i'' \subseteq S$, it holds that $w \notin \mathcal{R}(S, W, k, q)$, a contradiction. (b) if $w \notin W_i''$, then $w \notin \bigcup W_i'$, which means that w has been eliminated due to vector pruning, thus $w \notin \mathcal{R}(S, W, k, q)$, a contradiction.*

Let us assume that $\exists w \in \bigcup \mathcal{R}(S_i'', W_i'', k, q)$ such that $w \notin \mathcal{R}(S, W, k, q)$. This means that there exist k objects in S that are ranked higher than q w.r.t. w, but there exist fewer than k objects in $S_i'' = \bigcup S_i'$ that are ranked higher than q. Thus, at least one point p that belongs to $S - \bigcup S_i'$ is ranked higher than q. Then, based on the definition of S_i', q dominates p, which in turn means that the score of p cannot exceed the score of q, a contradiction.

6.2 Complexity Analysis of DiPaRT

In terms of space complexity, each *Map* task of DiPaRT requires $O(|W_i|)$ memory consumption plus the space required for storing the R-Trees (required by RTA) of individual S_i' sets. Also, the amount of shuffled data is $O(r \cdot |\bigcup S_i'| + |\bigcup W_i'|)$, where r is the number of *Reduce* tasks. In the *Reduce* phase, the space complexity is $O(|RTree(\bigcup S_i'')| + \frac{|\bigcup W_i'|}{r})$. In summary, the main bottleneck

is the size of $\bigcup S'_i$, which should not exceed the available memory of a *Reduce* task.

Regarding time complexity, each *Map* task needs to process in the worst case $|W_i|$ top-k queries, which corresponds to $O(|W_i| \cdot |S_i|)$ complexity (this is the worst-case scenario for any reverse top-k algorithm). In the Reduce phase, the number of top-k queries that need to be processed is $\frac{|\bigcup W'_i|}{r}$ for r *Reduce* tasks, which corresponds to $O(\frac{|\bigcup W'_i| \cdot |\bigcup S'_i|}{r})$ complexity.

A summary of the complexity analysis of DiPaRT algorithm is depicted in Table 2.

Table 2 Complexity summary of DiPaRT.

	Map task	**Reduce task**	**Shuffle**
Time	$O(\|W_i\| \cdot \|S_i\|)$	$O(\frac{\|\bigcup W'_i\| \cdot \|\bigcup S'_i\|}{r})$	
Space	$O(\|W_i\| + \|RTree(S''_i)\|)$	$O(\|RTree(\bigcup S'_i)\| + \frac{\|\bigcup W'_i\|}{r})$	$O(r \cdot \|\bigcup S'_i\| + \|\bigcup W'_i\|)$

6.3 Complexity Analysis of DiPaRT+

Regarding main memory requirements, each *Map* task of DiPaRT+ consumes $O(k \cdot r)$ memory, while each *Reduce* task requires $O(|RTree(S''_i)|)$ memory. The amount of intermediate results is $O(|\bigcup W''_i| + |\bigcup S''_i|)$. We need to emphasize that while it holds $|S''| = |\bigcup S''_i| = r \cdot |\bigcup S'_i| = r \cdot |S'|$ for DiPaRT, the same is not true for DiPaRT+, since the second algorithm utilizes advanced techniques that are able to prune replicated data objects. For DiPaRT+ it holds that $|S''| \leq r \cdot |S'|$.

In terms of time complexity, a *Map* task might search $O(r)$ partitions in the worst-case to determine the partition of each weighting vector. For data objects, a *Map* task with BB pruning performs $O(r)$ comparisons for memory-less pruning, plus $O(k \cdot r)$ comparisons for memory-based pruning. A *Map* task with WV pruning performs $r \cdot O(2 \cdot |\mathcal{V}_i|) = r \cdot O(2 \cdot 2^{n-1}) = O(r \cdot 2^n)$ comparisons. Each Reduce task processes $O(|W''_i|)$ top-k queries, if it does not terminate early.

A summary of the complexity analysis of DiPaRT+ algorithm is depicted in Table 3.

Table 3 Complexity summary of DiPaRT+.

	Map task	**Reduce task**	**Shuffle**
Time	$O(k \cdot r)$ for BB or $O(r \cdot 2^n)$ for WV	$O(\|W''_i\| \cdot \|S''_i\|)$	
Space	$O(k \cdot r)$	$O(\|RTree(S''_i)\|)$	$O(\|\bigcup W''_i\| + \|\bigcup S''_i\|)$

6.4 Discussion

In terms of space complexity, DiPaRT+ is more efficient than DiPaRT, since (a) in the Map Tasks, the former does not need to store the S'_i in RTree and (b) in the Reduce tasks, DiPaRT needs to additionally store the W''_i items. Regarding time complexity, DiPaRT+ is more efficient than DiPaRT in its Map tasks, since it utilizes the advanced pruning techniques, to quickly prune data items and weight vectors. The time complexity of DiPaRT's Map tasks is mostly affected by the expensive reverse top-k computation. In Reduce tasks, the time complexity of DiPaRT+ is greatly dependent on the utilized partitioning scheme, since the size $|W''_i|$ is defined by the selected partitioning scheme. On the other hand, the time complexity of DiPaRT's Reduce tasks is only affected by the number of surviving weight vectors (W'_i) and data objects (S'_i).

The complexity of our algorithms is determined by the sizes of $\bigcup S'_i$, $\bigcup S''_i$, $\bigcup W'_i$ and $\bigcup W''_i$. Since there is no theoretical guarantee of these sizes, we rely on practical heuristics to demonstrate the effectiveness of our proposed pruning techniques. Hence, we provide an extensive experimental study in Section 7 which empirically demonstrates that DiPaRT+ outperforms DiPaRT and both BB and WV pruning have a high pruning effectiveness.

7 Experimental Evaluation

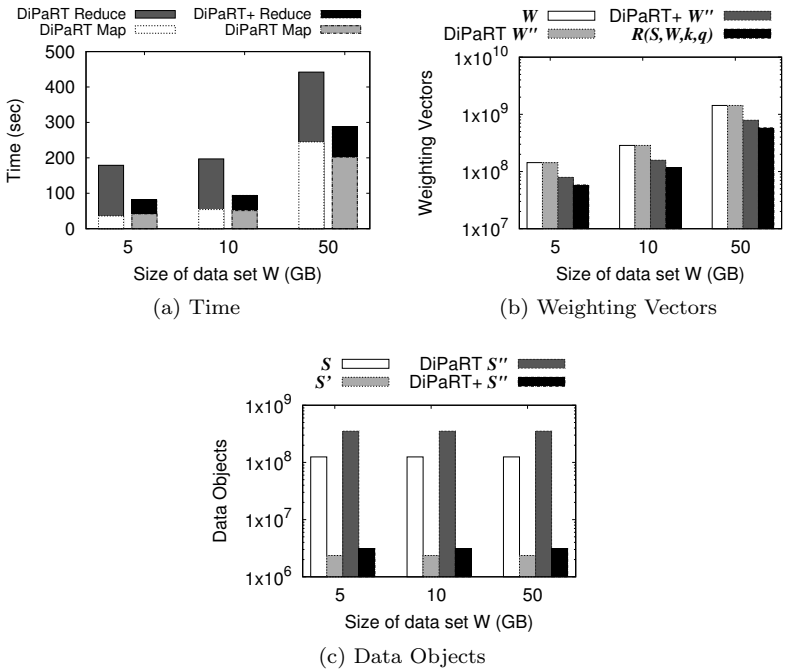
7.1 Limitations of Centralized Algorithms

We demonstrate the limitations of centralized reverse top-k algorithms, RTA [27] and branch-and-bound [28], 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.

7.2 Evaluation of Parallel Algorithms

In this section we evaluate the performance of our proposed parallel solutions. Both DiPaRT and DiPaRT+ are implemented in Java[2], using Apache Hadoop.

[2] Source code available at: `https://github.com/nikpanos/rtopk.distributed`



(a) Time (b) Weighting Vectors

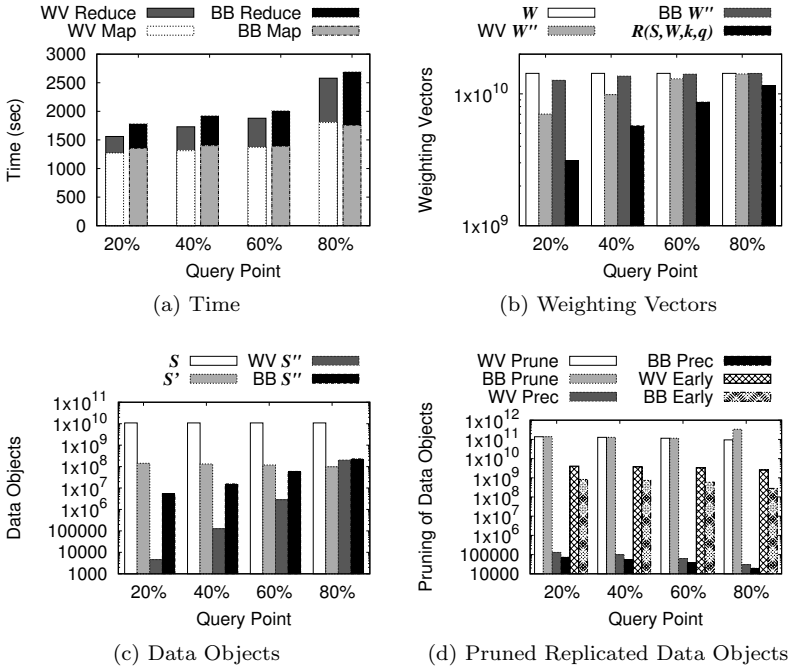
(c) Data Objects

Fig. 5 Comparative performance of DiPaRT vs. DiPaRT+ for various sizes of data set W using TPC-H.

7.2.1 Experimental Setup

Platform. All algorithms are deployed at an in-house CDH 5.12 cluster consisting of 22 nodes with Hadoop 2.6 installed. Nodes 0-8 have 32GB of RAM, 5TB hard disk space and 2 CPUs, each featuring 4 cores running at 2.6 GHz. Nodes 9-21 have 128GB of RAM, 8TB hard disk space and 2 CPUs, each having 6 cores running at 2.6 GHz. On each node Java 8 is installed on Ubuntu 14, and the JVM heap size is set to 2GB for *Map* tasks and 8GB for *Reduce* tasks. We also configured HDFS with 128MB block size and a replication factor of 3.

Data sets. In order to use large-sized data sets for data objects S and user preferences W, we use synthetic data generators. For data objects S we use (a) a customized generator that produces uniform (UN), correlated (CO) or anti-correlated (AC) data distributions (as in [2,24,26]), and (b) the TPC-H generator, generating data for the "Part" and "PartSupp" tables with scale factor of 1000. In the case of TPC-H, the data set S of products is generated by joining the aforementioned tables. We pick only numerical attributes by the joined result set, namely "size", "retailprice", "qty" and "scost". As a result, a 5GB TPC data set is produced by selecting the top 125 million tuples.



(a) Time (b) Weighting Vectors

(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 6 Performance of advanced pruning techniques for various query points q.

Weight vectors w that correspond to user preferences are generated following clustered (CL) or uniform (UN) distributions; the produced values are normalized so that $\sum_{i=1}^{n} w[i] = 1$. For the clustered data set, 100 cluster centroids are picked randomly and the coordinates of generated points follow a Gaussian distribution on each dimension with variance 0.5, and a mean equal to the corresponding coordinate of the centroid.

The sizes of our synthetic non-TPC data sets S and W vary from 5GB to 1000GB for 2 to 6 dimensions. Notice that our default 4-dimensional experimental setup has a combined input size of 1TB (500GB S plus 500GB W). Our largest data set S consists of more than 36 billion data objects, while the largest W data set contains more than 45 billion weighting vectors. These are four orders of magnitude larger than the ones used in the centralized reverse top-k approaches so far (e.g., [24,28,29]). Moreover, we need to emphasize that a baseline solution for processing a reverse top-k query, requires processing a top-k query for each weighting vector in W. Thus, for our largest data set W, more than 45 billion top-k queries need to be processed. This clearly demonstrates the complexity of the reverse top-k query for large data sets and motivates parallel processing with MapReduce. All data sets are stored in HDFS as plain text files.

Queries. Depending on the chosen query point, the size of the result set may vary, thus affecting the efficiency of the evaluated algorithms. As such,

Table 4 Experimental parameters and values.

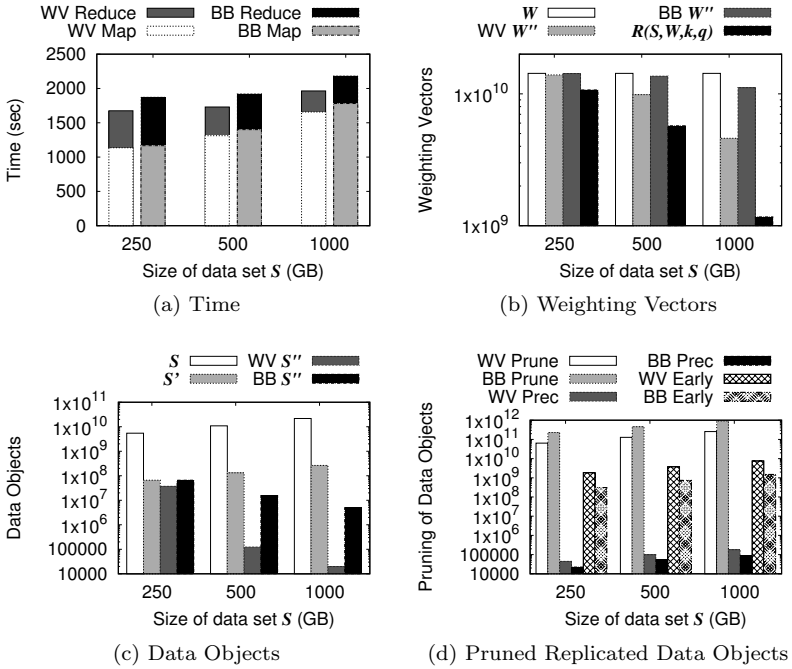
Parameter	**Values**
Data set S distribution	**UN**, CO, AC, TPC-H
Data set W distribution	**UN**, CL
Dimensionality (n)	2, **4**, 6
$\|W\|$ (in GB)	5, 10, 50, 250, **500**, 1000
$\|S\|$ (in GB)	5, 250, **500**, 1000
k	10, **20**, 30
Query point q (ratio $\|\mathcal{R}\|/\|W\|$)	20%, **40%**, 60%, 80%
$\mathcal{P}$ (segments per dimension)	6, 8, **10**(WV), 12, 14, **16**(BB), 18, 20, 22

we experiment with different query points, which produce varying sizes of result sets. We choose four query points, based on the percentage of weighting vectors that are reported in the final result set, namely 20%, 40%, 60% and 80%. After executing this set of experiments, we adopt the 40% query point as the default choice for the rest of the experiments. In cases where a new query point must be generated (i.e., when varying the dimensionality), we select new query points, having 40% result set size ratio.

Partitioning. The DiPaRT algorithm operates on a random partitioning of weight vectors, thus we simply experiment with varying the number of evenly distributed partitions. On the contrary, DiPaRT+ can exploit two partitioning schemes, namely the angle-based and grid-based partitioning. The former splits each angular dimension of the n-sphere to c evenly distributed segments producing c^{n-1} partitions, while the latter splits each Cartesian dimension to c segments resulting to c^n evenly distributed partitions. In grid based partitioning some partitions are pruned away because they contain no weighting vectors, thus resulting to fewer than c^n partitions. In the following, we first experiment with different values of c for both angular and grid based partitioning, in order to find the best-performing values that will be used as default. Notice that for both DiPaRT and DiPaRT+, the number of partitions is a crucial choice which defines the number of *Reduce* tasks to be executed, thus highly affecting performance.

Metrics. Our main performance metric is the *execution* time needed for each experiment (job) to complete. We report the total time divided into two parts, namely *Map* and *Reduce*, as indicators of the workload assigned to the corresponding phases.

The rest of the metrics quantify the pruning effectiveness achieved for W and S data sets, in the various stages of query processing. We report the number of vectors: (a) given as input W to the job, (b) survived by early termination of *Reduce* tasks (indicated as W''), and (c) reported to the final result set (denoted $R(S, W, k, q)$). Furthermore, we report the number of data objects: (a) given as input S to the job, (b) survived by the dominance pruning (denoted S') and (c) given as input to the non-early-terminated Reduce tasks (denoted S''). Notice that unlike the former two data objects metrics, the size of S'' indicates replicated data objects. Additionally we measure the



(a) Time (b) Weighting Vectors

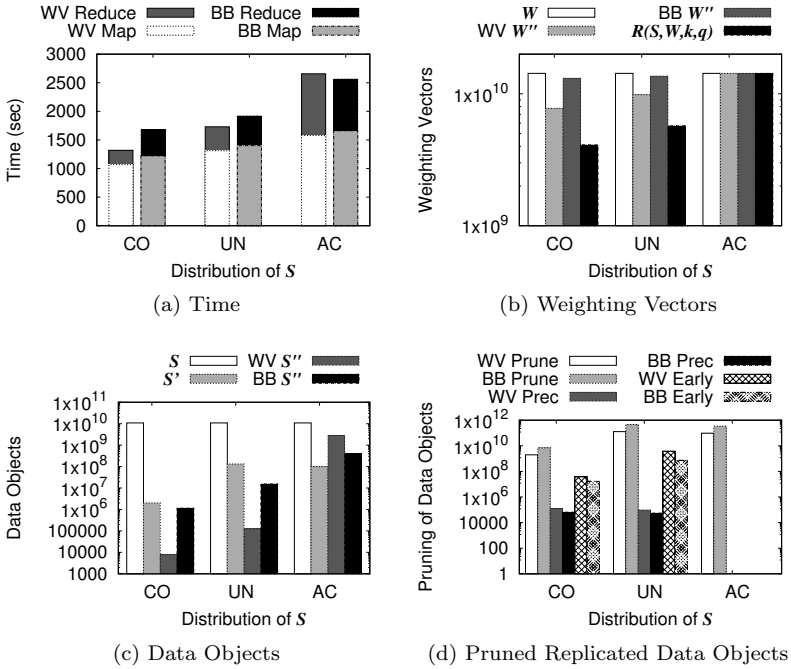
(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 7 Performance of advanced pruning techniques for various sizes of data set S.

individual sizes of pruned replicated data objects, to demonstrate a direct comparison between the various pruning techniques introduced in this paper. More specifically, we report the number of replicated data objects that were pruned by: (a) either BB or WV pruning, (b) precedence pruning performed in Map phase and (c) early termination pruning of *Reduce* tasks.

To demonstrate the merit of early termination, we also report the percentage of *Reduce* tasks that terminated early. Notice the use of log-scale in all charts, except from the ones reporting execution time.

Evaluation Methodology. The limitations of centralized approaches have already been discussed in Section 7.1. Thus, we turn our attention to DiPaRT and DiPaRT+, providing a head-to-head comparison between them, in order to clearly present the advantages of DiPaRT+. Then, we focus on DiPaRT+ and conduct a thorough sensitivity analysis that demonstrates both the efficiency and scalability of our proposed algorithm. For both pruning techniques, BB and WV, discussed for DiPaRT+, we vary the sizes of both data sets S and W, their data distributions, the dimensionality n, the value of k, the partitioning scheme $\mathcal{P}$ and the query point q. The parameter values are shown in Table 4, with default values in bold. In each experiment, we vary a single parameter, while setting the others to their default values.



(a) Time (b) Weighting Vectors

(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 8 Performance of advanced pruning techniques for various distributions of data set S.

7.2.2 Comparison of DiPaRT against DiPaRT+

For this set of experiments, we employ the 4-dimensional TPC-H data set S consisting of 125 million data objects, while we vary the size of the data set W from 5 to 50 GBs using UN distribution. The parameter k is set to 20, and the query point is selected to have 40% result ratio. DiPaRT is configured to run with 150 partitions (i.e., *Reduce* tasks), whereas DiPaRT+ is set with 216 angular partitions. These values have been identified experimentally and are the best in terms of execution time for both algorithms. Notice that DiPaRT's performance degrades for larger number of partitions due to the network overhead produced by increased replication of $\bigcup S'_i$.

Figure 5 demonstrates the comparative performance of our algorithms. In Figure 5a, the total length of the bar corresponds to the execution time of each algorithm, and it is split to show the cost of *Map* and *Reduce* individually. DiPaRT+ is always faster than DiPaRT for all sizes of the W data set. The *Map* phase of DiPaRT+ is slightly more expensive than its *Reduce* phase, due to the pruning of data objects performed in *Map* tasks; this effective pruning however, enables its *Reduce* tasks to either terminate early, or process fewer data objects in their top-k queries. The increase of *Map* phase workload appearing at the 50GB data set W, is mainly due to the increased cost of loading the input data from HDFS.

Figure 5b shows that the early termination technique employed in DiPaRT+ prunes more weighting vectors. Interestingly, this pruning effectiveness comes at a much lower processing cost, as it manages to terminate early approximately half of DiPaRT+'s *Reduce* tasks. Figure 5c shows that the number of replicated data objects (S'') processed by *Reduce* tasks of DiPaRT+ is nearly three orders of magnitude fewer than those of DiPaRT. This is due to the pruning of replicated data objects employed in DiPaRT+. In summary, as expected, DiPaRT+ consistently outperforms DiPaRT.

Moreover, we tested DiPaRT by using an even larger set of weighting vectors W (500GB), and noticed that it requires more than 5 hours of processing time, whereas DiPaRT+ scales gracefully for larger input sizes, as will be shown in Section 7.2.3. Therefore, in the following, we turn our attention to the sensitivity analysis of DiPaRT+, for experimenting with larger input volume of data.

7.2.3 Sensitivity Analysis

In this section, we conduct a large variety of experiments to study the performance and pruning effectiveness of DiPaRT+, when using either BB or WV pruning.

Varying q. In Figure 6, we test the performance of DiPaRT+ for queries with increasing size of result sets. Figure 6a shows that DiPaRT+ with WV pruning constantly outperforms BB pruning. The *Map* phase of WV is more expensive than BB, as expected by the complexity analysis of DiPaRT+. However, the higher *Map* pruning effectiveness performed by WV, results in much lower workload of its *Reduce* phase. In Figure 6b, WV achieves much better pruning effectiveness for weighting vectors, while Figure 6c shows that more data points are given as input to *Reduce* tasks as the result size is increased. Furthermore, as shown in Figure 6d, the effectiveness of both WV and BB pruning decreases for larger result set sizes, having also fewer data objects pruned by early termination. Notice that BB pruning is configured to use more partitions in order to achieve its best possible performance. Hence, BB pruning experimental results have larger sizes of replicated data objects.

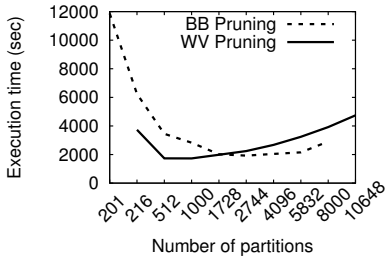
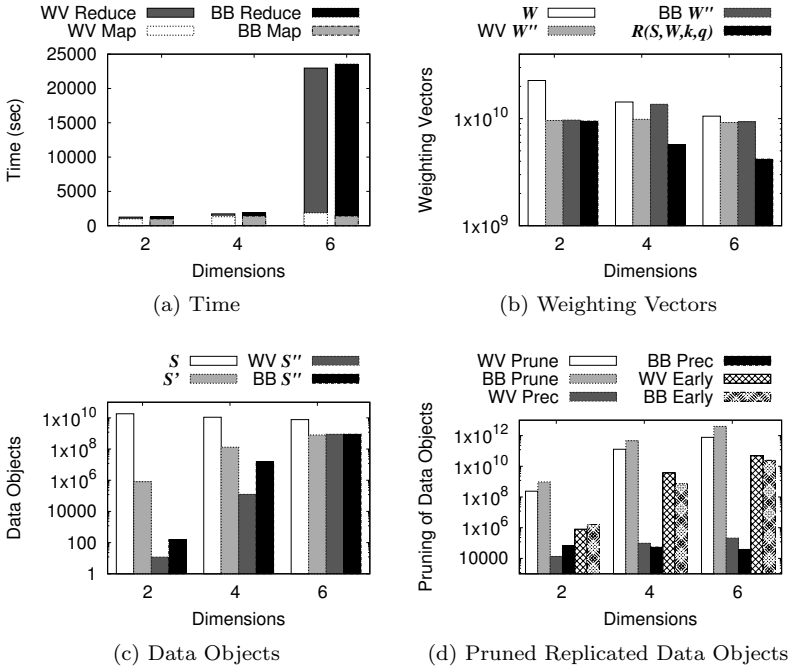


Fig. 9 Performance of advanced pruning techniques for various number of partitions.



(a) Time (b) Weighting Vectors

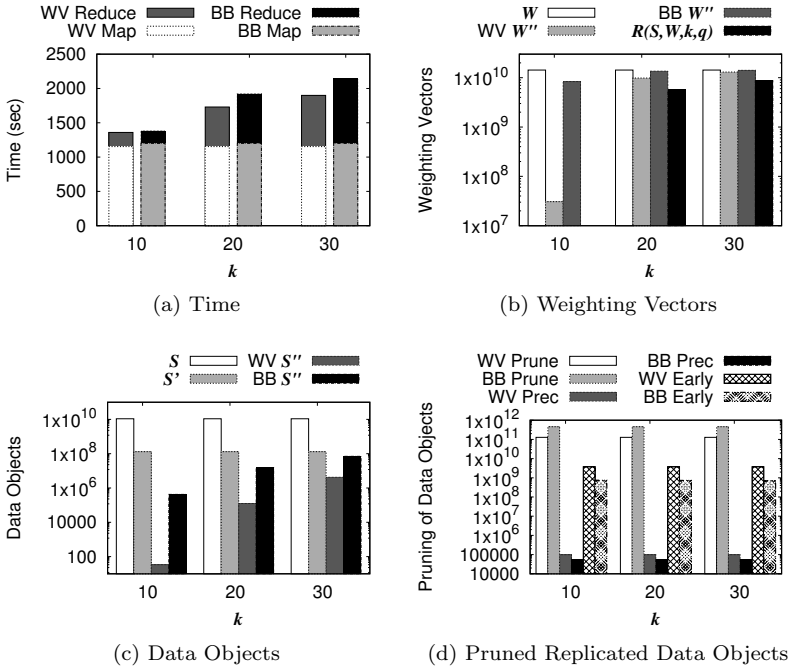
(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 10 Performance of advanced pruning techniques when increasing the dimensionality.

Varying number of partitions. An important parameter for both BB and WV is the number of partitions, as it directly affects the number of *Reduce* tasks. Thus, in Figure 9, we evaluate the performance of both pruning techniques for different numbers of partitions. First, we see that WV is more stable than BB, whose performance degrades for smaller number of partitions. This is due to its limited capability of effectively pruning data points. Second, we see that the best performance is achieved at 1000 partitions for WV, and at 3476 partitions for BB. Third, the best performance of WV is better than the one of BB. For the remaining experiments, we use the values above as default, unless stated otherwise.

Both BB and WV pruning techniques appear to have an optimal number of partitions. For WV, this optimal number is less than that of BB pruning, due to the former's ability to use tighter bounds. By using too many partitions, both pruning techniques have increased resource requirements since their Map tasks' time complexity depends on the number of chosen partitions (Section 6.3). On the other hand, by using too few partitions, the corresponding WV and BB pruning bounds are less tight, resulting to reduced pruning effectiveness and performance.

Varying $|S|$. Figure 7a shows that the impact of increased size of S on the performance of DiPaRT+ for any pruning (WV or BB) is rather small. As we increase $|S|$ from 250GB to 1TB, the execution time increases only slightly.



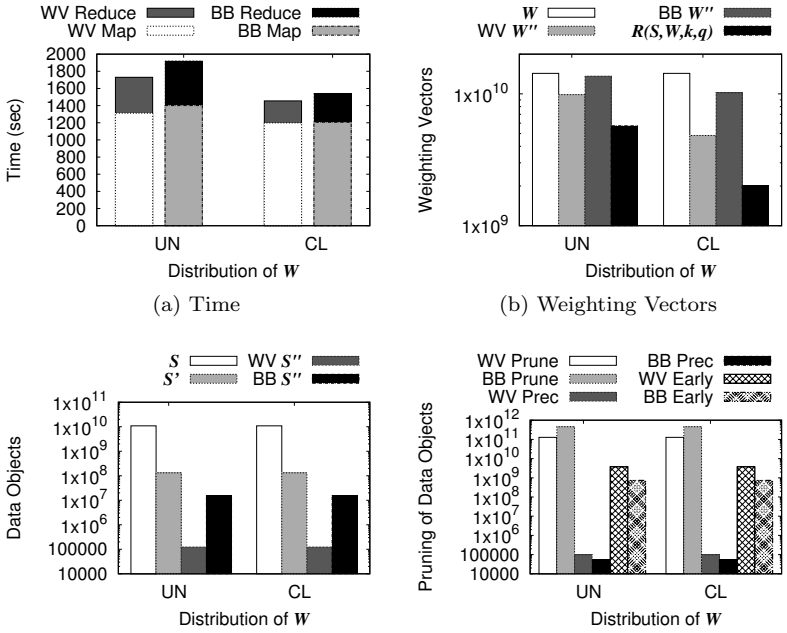
(a) Time (b) Weighting Vectors

(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 11 Performance of advanced pruning techniques for various values of top-k.

Map tasks require more time to load larger data sets from HDFS, affecting the overall performance of the job. However, the workload of the *Reduce* phase is greatly decreased for larger sizes of S, thus mitigating the impact of *Map* tasks to the overall performance. The reason that the *Reduce* phase is faster for larger $|S|$ is that more *Reduce* tasks terminate early for larger sizes of S. WV pruning constantly outperforms BB pruning for this set of experiments, as the former outputs fewer weighting vectors and data objects to its *Reduce* tasks. Figure 7b shows that as $|S|$ increases, the effectiveness of WV pruning weighting vectors increases as well. Similarly, WV is able to prune more replicated data objects, for larger sizes of data set S, as depicted in Figure 7d.

Varying the data distribution of S. Figure 8a shows that for CO and UN data distributions, WV pruning performs better than BB, whereas this behavior is reversed for AC. Notice that this is the only setup in our experimental study, where BB pruning outperforms WV. This situation is explained by the fact that none of the *Reduce* tasks of WV manages to terminate early in the case of AC data distribution. The result size ratio for AC, is close to 100% of $|W|$, explaining the fact that zero *Reduce* tasks terminate early. On the other hand, the CO distribution has a lower result set size, enabling WV to prune many more weighting vectors than BB, as shown in Figure 8b. Also for CO, the amount of data objects given as input to *Reduce* phase is much smaller for WV pruning (Figure 8c). In contrast, in the case of AC, the input



(a) Time (b) Weighting Vectors

(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 12 Performance of advanced pruning techniques when varying the data distribution of W.

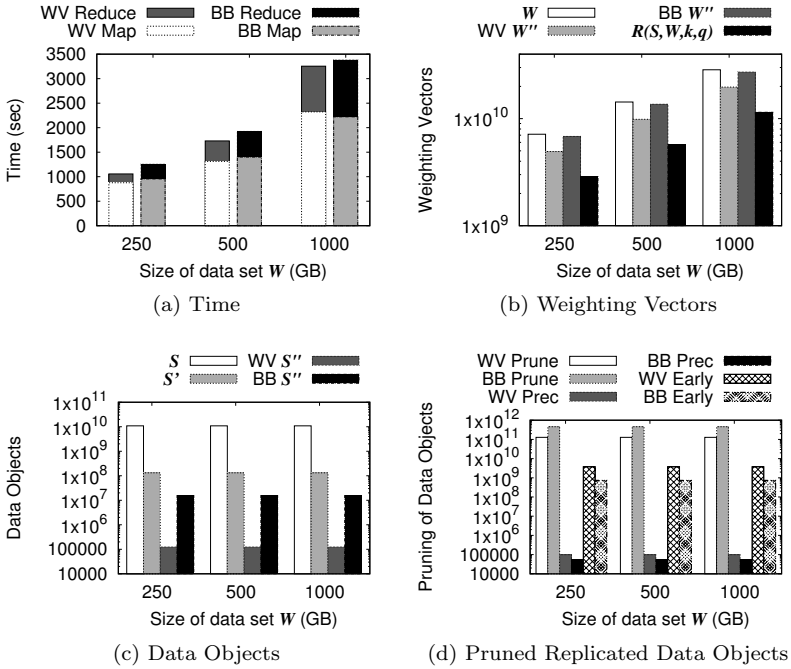
for the *Reduce tasks* of WV is larger. The effectiveness of data objects pruning of BB is justified by the fact that BB pruning uses a more detailed partitioning scheme (i.e., larger number of partitions), enabling the pruning of more data objects, as shown in Figure 8d.

Table 5 Values used while varying dimensions.

		2D	**4D**	**6D**
$\|S\|$	data objects	18B	11B	8B
	file size	500GB	500GB	500GB
$\|W\|$	weight vectors	23B	14B	11B
	file size	500GB	500GB	500GB
$\|\mathcal{P}\|$	angle based	300	1000	1024
$\|\mathcal{P}\|$	grid based	1196	3476	4971

Varying dimensionality. Table 5 demonstrates the parameters and data set sizes used in this set of experiments. For comparative purposes, we keep the size of all data sets equal to 500GB, each containing different number of data objects and vectors, depending on dimensionality.

Figure 10a shows that higher dimensional data sets require much higher execution time. This is expected for top-k and reverse top-k queries, since



(a) Time (b) Weighting Vectors

(c) Data Objects (d) Pruned Replicated Data Objects

Fig. 13 Performance of advanced pruning techniques for various sizes of data set W.

pruning becomes more difficult for higher dimensions. Figure 10c also shows that the size of S'' gets larger for more dimensions. This means that pruning of S is less effective for more dimensions, especially for BB pruning method (depicted in Figure 10d). The main performance impact, in the case of 6-dimensional data, is the fact that the *Reduce* phase gets highly overloaded by data points S.

Varying k. Figure 11 shows that increasing the value of k, results in higher execution time, mainly due to increased processing cost in the *Reduce* phase, since reverse top-k processing is more costly for higher values of k. This is expected as $|\mathcal{R}(S, W, k, q)|$ also increases with k (shown in Figure 11b). In contrast, the *Map* tasks are not significantly affected by the increase of k, as only memory-based pruning requires maintaining and processing buffers of larger size, but this overhead is negligible. WV pruning proves to be more efficient than BB pruning, regardless of the value of k. The amount of data objects surviving in *Reduce* phase increases for higher values of k, as shown in Figure 11c. However, WV pruning reports at least an order of magnitude fewer data points than BB in the *Reduce* phase, mainly due to the early termination pruning shown in Figure 11d.

Varying the distribution of W. Using clustered distribution for weighting vectors, while keeping the query point fixed, results to smaller result set size, thus increasing the efficiency of DiPaRT+, as depicted in Figure 12a. The

Map phase of DiPaRT+ is naturally unaffected by the distribution of weighting vectors, for both WV and BB pruning, as no processing of weighting vectors takes place. However, the *Reduce* phase benefits by the early termination pruning method for clustered weighting vectors, as more *Reduce* tasks terminate early. Figure 12b shows that WV pruning is more effective than BB, since twice as many weighting vectors are given as input for reverse top-k processing to BB compared to WV.

Varying $|W|$. Figure 13 depicts the scalability of DiPaRT+ when increasing the size of data set W. In terms of execution time, WV is always faster than BB, as depicted in Figure 13a. For both pruning techniques, the increase of execution time is proportional to the increase of $|W|$. The effectiveness of weighting vectors pruning and the result set size are also proportional to $|W|$, as shown in Figure 13b. Pruning techniques of data objects in S, are unaffected by the size of data set W, as shown in Figures 13c, 13d.

8 Related Work

Efficient top-k query processing has attracted much attention in the database research community; for a survey we refer to [14]. Distinguished work includes among others Onion [5], Prefer [13], and branch-and-bound search [25]. Reverse top-k queries [26, 27] have been proposed for assessing the impact of a potential product in the market, based on the number of users that have this product in their top-k results, including a branch-and-bound algorithm [28]. Moreover, the applicability of reverse top-k queries for discovering similar products has been studied in [12]. Also, multiple related topics have been studied, such as why-not questions on reverse top-k queries [10], a unified framework for rank-aware queries [6], as well as query operators (reverse k-ranks query [30] and maximum rank query [17]) with similarities to reverse top-k but different semantics. Recently, the problem of monochromatic reverse top-k in higher dimensions has also been addressed [24].

Evaluation of multiple top-k queries has been studied in [11]. The proposed methods exploit the fact that similar queries share common results to avoid evaluating the top-k queries one-by-one. Another approach for evaluating multiple top-k queries has appeared in [29]. The proposed framework can be employed to process reverse top-k queries efficiently, however it requires to pre-process all top-k queries in W, and then build an index over the k-th ranked objects of each query.

Due to the inherent limitations of various query processing tasks in MapReduce [8], several studies have proposed modifications to its internal operation to improve efficiency. For example, *HadoopDB* [1] is a hybrid system that installs a database system on each node and connect these nodes by means of Hadoop as the task coordinator and network communication layer. *HaLoop* [3] is a system designed for supporting iterative data analysis, which requires changes in terms of caching data, scheduling of iterative tasks to the same nodes to exploit the local caches, etc. *Sailfish* [21] improves performance by reducing

the number of disk accesses. Instead of writing to intermediate files at map side, data is shuffled directly and then written to a file at reduce side, one file per reduce task.

Another category of research papers propose efficient query processing algorithms for MapReduce that operate "on top of" Hadoop, without changing its internals. Our work belongs to this category. In this context, scalable processing of other preference-aware queries, such as skyline and reverse skyline queries, has been studied in [19, 20] for parallel environments (including MapReduce). *RanKloud* [4] has been proposed for top-k retrieval and relies on computing statistics (at runtime) during scanning of records, and then uses these statistics to compute a threshold for early termination. Parallel processing of top-k joins in MapReduce using histograms that enable early termination has been proposed in [23]. In [15], 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 [9].

9 Conclusions

In this paper, we present a parallel and scalable solution to the problem of reverse top-k computation. To this end, we introduce an algorithmic framework and provide implementations of two algorithms as instances of the framework. Our most efficient solution (DiPaRT+) owes its efficiency and scalability to useful pruning properties that eagerly discard data objects and user preferences while not compromising the correctness of the result set. Notable features of our solution include that it operates "on top of" vanilla Hadoop without requiring any changes of its internal operation, and that it provides the correct result in a single MapReduce job thereby saving the overheads related to scheduling chained jobs. Our experiments demonstrate the efficiency of our approach for data sets more than four orders of magnitude larger than those used in the centralized reverse top-k query processing literature by now.

Acknowledgements 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. The authors are grateful to Kjetil Nørvåg (NTNU) for providing access to the cluster infrastructure used for the empirical evaluation reported in the paper.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. PVLDB **2**(1), 922–933 (2009)
2. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of ICDE, pp. 421–430 (2001)

3. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB Journal **21**(2), 169–190 (2012)
4. Candan, K.S., Kim, J.W., Nagarkar, P., Nagendra, M., Yu, R.: RanKloud: scalable multimedia data processing in server clusters. IEEE MultiMedia **18**(1), 64–77 (2011)
5. Chang, Y., Bergman, L.D., Castelli, V., Li, C., Lo, M., Smith, J.R.: The onion technique: Indexing for linear optimization queries. In: Proceedings of the SIGMOD, pp. 391–402 (2000)
6. Cheema, M.A., Shen, Z., Lin, X., Zhang, W.: A unified framework for efficiently processing ranking related queries. In: Proceedings of EDBT., pp. 427–438 (2014)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Doulkeridis, C., Nørvåg, K.: A survey of large-scale analytical query processing in mapreduce. VLDB J. **23**(3), 355–380 (2014)
9. Doulkeridis, C., Vlachou, A., Mpestas, D., Mamoulis, N.: Parallel and distributed processing of spatial preference queries using keywords. In: Proceedings of EDBT, pp. 318–329 (2017)
10. Gao, Y., Liu, Q., Chen, G., Zheng, B., Zhou, L.: Answering why-not questions on reverse top-k queries. PVLDB **8**(7), 738–749 (2015)
11. Ge, S., U, L.H., Mamoulis, N., Cheung, D.W.: 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)
12. Georgoulas, K., Vlachou, A., Doulkeridis, C., Kotidis, Y.: User-centric similarity search. IEEE Trans. Knowl. Data Eng. **29**(1), 200–213 (2017)
13. Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER: A system for the efficient execution of multi-parametric ranked queries. In: Proceedings of SIGMOD, pp. 259–270 (2001)
14. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Computing Surveys **40**(4), 1–58 (2008)
15. Kim, W., Kim, Y., Shim, K.: Parallel computation of k-nearest neighbor joins using mapreduce. In: Proceedings of BigData, pp. 696–705 (2016)
16. Levandoski, J.J., Eldawy, A., Mokbel, M.F., Khalefa, M.E.: Flexible and extensible preference evaluation in database systems. ACM Trans. Database Syst. **38**(3), 17:1–17:43 (2013)
17. Mouratidis, K., Zhang, J., Pang, H.: Maximum rank query. PVLDB **8**(12), 1554–1565 (2015)
18. Nikitopoulos, P., Sfyris, G.A., Vlachou, A., Doulkeridis, C., Telelis, O.: Parallel and distributed processing of reverse top-k queries. In: Proceedings of ICDE, pp. 1586–1589 (2019)
19. Park, Y., Min, J., Shim, K.: Parallel computation of skyline and reverse skyline queries using MapReduce. PVLDB **6**(14), 2002–2013 (2013)
20. Park, Y., Min, J., Shim, K.: Efficient processing of skyline queries using mapreduce. IEEE Trans. Knowl. Data Eng. **29**(5), 1031–1044 (2017)
21. Rao, S., Ramakrishnan, R., Silberstein, A., Ovsiannikov, M., Reeves, D.: Sailfish: a framework for large scale data processing. In: Proceedings of SOCC, p. 4 (2012)
22. Rockafellar, R.T.: Convex Analysis. Princeton Landmarks in Mathematics. Princeton University Press (1997)
23. Saouk, M., Doulkeridis, C., Vlachou, A., Nørvåg, K.: Efficient processing of top-k joins in mapreduce. In: Proceedings of BigData, pp. 570–577 (2016)
24. Tang, B., Mouratidis, K., Yiu, M.L.: Determining the impact regions of competing options in preference space. In: Proceedings of SIGMOD, pp. 805–820 (2017)
25. Tao, Y., Hristidis, V., Papadias, D., Papakonstantinou, Y.: Branch-and-bound processing of ranked queries. Information Systems **32**(3), 424–445 (2007)
26. Vlachou, A., Doulkeridis, C., Kotidis, Y., Nørvåg, K.: Reverse top-k queries. In: Proceedings of ICDE, pp. 365–376 (2010)
27. Vlachou, A., Doulkeridis, C., Kotidis, Y., Nørvåg, K.: Monochromatic and bichromatic reverse top-k queries. IEEE TKDE **23**(8), 1215–1229 (2011)
28. Vlachou, A., Doulkeridis, C., Nørvåg, K., Kotidis, Y.: Branch-and-bound algorithm for reverse top-k queries. In: Proceedings of SIGMOD, pp. 481–492 (2013)

29. Yu, A., Agarwal, P.K., Yang, J.: Processing a large number of continuous preference top-k queries. In: Proceedings of SIGMOD, pp. 397–408 (2012)
30. Zhang, Z., Jin, C., Kang, Q.: Reverse k-ranks query. PVLDB **7**(10), 785–796 (2014)