Efficient Distributed Top-k Query Processing with Caching

Norvald H. Ryeng, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg

Norwegian University of Science and Technology
Department of Computer and Information Science
Trondheim, Norway
{ryeng,vlachou,cdoulk,noervaag}@idi.ntnu.no

Abstract. Recently, there has been an increased interest in incorporating in database management systems rank-aware query operators, such as top-k queries, that allow users to retrieve only the most interesting data objects. In this paper, we propose a cache-based approach for efficiently supporting top-k queries in distributed database management systems. In large distributed systems, the query performance depends mainly on the network cost, measured as the number of tuples transmitted over the network. Ideally, only the k tuples that belong to the query result set should be transmitted. Nevertheless, a server cannot decide based only on its local data which tuples belong to the result set. Therefore, in this paper, we use caching of previous results to reduce the number of tuples that must be fetched over the network. To this end, our approach always delivers as many tuples as possible from cache and constructs a remainder query to fetch the remaining tuples. This is different from the existing distributed approaches that need to re-execute the entire top-k query when the cached entries are not sufficient to provide the result set. We demonstrate the feasibility and efficiency of our approach through implementation in a distributed database management system.

1 Introduction

Nowadays, due to the huge amount of available data, users are often overwhelmed by the variety of relevant data. Therefore, database management systems offer rank-aware query operators, such as top-k queries, that allow users to retrieve only a limited set of the most interesting tuples. Top-k queries [5, 8, 15] retrieve the k tuples that best match the individual user preferences based on a user-specified scoring function. Different scoring functions express the preferences of different users. Several applications benefit from top-k queries, including web search, digital libraries and e-commerce. Moreover, the high distribution of data raises the importance of supporting efficient top-k query processing in distributed systems.

In this paper, we propose a cache-based approach, called $ARTO$[1], for efficiently supporting top-k queries in distributed database management systems.

[1] Algorithm with Remainder TOp-k queries.

In large-scale distributed systems, the dominant factor in the performance of query processing is the communication cost, measured as the number of tuples transmitted over the network. Ideally, only the k tuples that belong to the result set should be fetched. Nevertheless, in the case of top-k queries, a server cannot individually decide which of its top-k local tuples belong to the global top-k result set of the query. In order to restrict the number of fetched tuples and reduce the communication costs, we employ caching of result sets of previously posed top-k queries. Each server autonomously maintains its own cache and only a summary description of the cache is available to any other server in the network.

In general, a top-k query is defined by a scoring function f and a desired number of results k, and these parameters differ between queries. Given a set of cached top-k queries in the system and a new query, the problem is to identify whether the results of cached queries are sufficient to answer the new query. To deal with this problem, we apply techniques similar to those of the view selection problem in the case of materialized views [15] in centralized database systems. Based on the cached queries, we need to decide whether the cached results *cover* the results of a new query. In this case, the query is answered from the cache and no tuples need to be transferred over the network. However, the major challenge arises when the query is not covered by the cached tuples.

Different from existing approaches [20] that require the servers to recompute the query from scratch, we do not evaluate the entire query, but we create a *remainder query* that provides the result tuples that are not found in cache. More detailed, we split the top-k query into a top-k' query ($k' < k$) that is answerable from cache, and a remainder next-$(k - k')$ query that provides the remaining tuples that were not retrieved from the top-k' query. To further optimize the query performance, we deliberately assign the top-k query to the server that is expected to induce the lowest network cost based on the locally cached tuples. To summarize, the contributions of this paper are:

- We propose a novel framework for distributed top-k queries that retrieves as many tuples k' ($k' < k$) as possible from the cache, and poses a *remainder query* that provides the remaining $k - k'$ tuples that are not found in cache.
- We present a novel method for efficiently computing remainder queries, without recomputing the entire top-k query.
- We propose a server selection mechanism that identifies the server that owns the cache with the most relevant entries for a given query.
- We evaluate our approach experimentally by integrating ARTO in an existing distributed database management system [14], and we show that our method significantly reduces communication costs.

The rest of this paper is organized as follows. In Section 2, we explain how this paper relates to previous work in this area. Section 3 presents preliminary concepts, and Section 4 presents our framework for distributed top-k query processing. Answering top-k queries from cache is outlined in Section 5. The remainder queries are described in Section 6, while Section 7 presents the server selection mechanism. Results from our experimental evaluation are presented in Section 8, and in Section 9 we conclude the paper.

2 Related Work

Centralized processing of top-k queries has received considerable attention recently [2, 5, 8, 15]. For a comprehensive survey of top-k query processing we refer to [16]. Hristidis et al. [15] discuss how to answer top-k queries from a set of materialized ranked views of a relational table. Each view stores all tuples of the relation ranked according to different ranking functions. The idea is to materialize a set of views based on a requirement either on the maximum number of tuples that must be accessed to answer a query, or on the maximum number of views that may be created. When a query arrives, one of these views is selected to be used for answering the top-k query. In [11], the materialized views of previous top-k queries (not entire relations) are used to answer queries, as long as they contain enough tuples to satisfy the new query. For each incoming query, the view selection algorithm chooses a set of views that will give an optimal (in terms of cost) execution of the proposed LPTA algorithm. A theoretical background on view selection is given in [3], providing theoretical guarantees whether a view is able to answer a query or not. However, the algorithms that are presented only allow a query to be answered from views if the views are guaranteed to provide the answer.

In distributed top-k query processing, the proposed approaches can be categorized based on their operation on vertically [9, 12, 17, 6, 18] or horizontally [1, 4, 19, 20] distributed data. In the case of vertically distributed data, any server maintains only a subset of the attributes of the complete relation. Then, each server is able to deliver tuples ranked according to any scoring function that is applied on one or more of its attributes [9, 12, 17]. The TPUT algorithm [6] focuses on limiting the number of communication round-trips, and this work has later been improved by KLEE [18].

In the case of horizontally distributed data, each server stores a subset of the tuples of the complete relation, but for each tuple all attributes are maintained. In [1], a broadcasting technique for answering top-k queries in unstructured peer-to-peer networks is presented. For super-peer topologies, Balke et al. [4] provides a method using indexing to reduce communication costs. This method requires all super-peers to process queries, unless exactly the same query reappears. SPEERTO [19] pre-computes and distributes skyline result sets of super-peers in order to contact only those super-peers that are necessary at query time. BRANCA [20] is a distributed system for answering top-k queries. Caching of previous intermediate and final results is used to avoid recomputing parts of the query. The cache is used much in the same way as the materialized views in [3, 11, 15], but on intermediate results of the query. This means that some servers in the system must process the query from scratch, while others may answer their part of the same query from cache. The main difference between ARTO and other caching approaches, such as BRANCA, becomes clear in the hard cases, when the query cannot be answered by the cache. ARTO still uses the part of the cache that partially answers the query and poses a remainder query for the remaining tuples, without the need to process the query from scratch, as in the case of BRANCA.

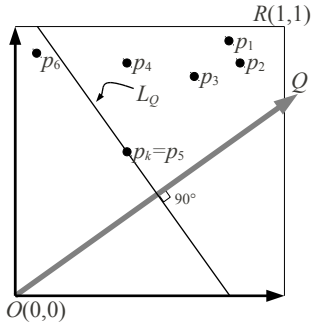

Fig. 1. 2D representation of query and data space.

Finally, our techniques for answering top-k queries relate to stop-restart of query processing [7, 10, 13]. These methods assume that some of the result tuples are already produced and restart processing from where the original query stopped. Our remainder queries differ by not restarting an existing top-k query but a query that was partially answered by cached tuples.

3 Preliminaries

Top-k queries are defined based on a monotone function f that combines the individual scores into an overall scoring value, that in turn enables the ranking (ordering) of tuples. Given a relation R, which consists of n attributes a_i, the result set of a top-k query $Q = \langle R, f, k\rangle$ contains k tuples such that there exists no other tuple in R with better score than the k tuples in the result set. Relation R may be a base relation or the result of an algebra operator, i.e., the result of a join. The most commonly used scoring function is the weighted sum function, also called linear. Each attribute a_i is associated with query-dependent weight w_i indicating a_i's relative importance for the query. Furthermore, without loss of generality, we assume that for any tuple t and any attribute a_i the values $t(a_i)$ are scaled to $[0, 1]$. The aggregated score $f(t)$ for a tuple t is defined as a weighted sum of the individual scores: $f(t) = \sum_{i=1}^{n} w_i t(a_i)$, where $w_i \geq 0$ $(1 \leq i \leq n)$, and $\exists j$ such that $w_j > 0$. The weights represent the relative importance of different attributes, and without loss of generality we assume that $\sum_{i=1}^{n} w_i = 1$. Thus, a linear top-k query Q is defined by a vector $\boldsymbol{w_Q}$ and the parameter k. The ranked tuples can be delivered in either ascending or descending order, but for simplicity, we will only consider descending order in this paper. Our results are also valid in the ascending case.

A tuple t of R can be represented as a point in the n-dimensional Euclidean space. Furthermore, given a top-k query $Q = \langle R, f, k\rangle$ defined by a linear scoring function, there exists a one-to-one correspondence between the weighting vector $\boldsymbol{w_Q}$ and the hyperplane which is perpendicular to $\boldsymbol{w_Q}$. We refer to the (n-1)-dimensional hyperplane, which is perpendicular to vector $\boldsymbol{w_Q}$ and crosses the kth result tuple, as the *query plane* of $\boldsymbol{w_Q}$, and denote it as L_Q. All points on the

query plane L_Q have the same scoring value for $\boldsymbol{w_Q}$. A 2-dimensional example is depicted in Fig. 1. Processing the top-k query Q is equivalent to sweeping the line L_Q from the upper right corner towards the lower left corner. Each time L_Q meets a tuple t, this tuple is reported as the next result tuple. When L_Q meets the k-th tuple, the complete result set has been retrieved.

4 ARTO Framework

In this paper, we assume a distributed database system where the relations are horizontally fragmented over multiple servers. In more details, each relation R is fragmented into a set of fragments $R_1, R_2, \ldots, R_f$ and each fragment R_i consists of a subset of tuples of the relation R. Our approach is generic and imposes no further constraints on the way partitions are created or whether they are overlapping or not. Furthermore, each server may store fragments of different relations. Any server can pose a top-k query and we refer to that server as *querying server*. During query processing, the querying server may connect to any other server. Thus, no routing paths are imposed on the system other than those of the physical network itself. The only assumption of ARTO is that there exists a distributed *catalog* accessible to all servers, which indexes the information about which server stores fragments of each relation R. Such a distributed catalog can be implemented using a distributed hash table (DHT).

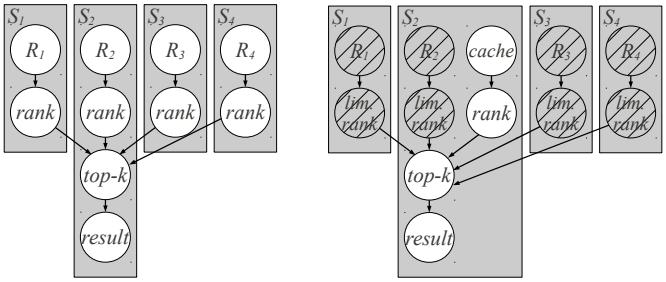

Fig. 2. (a) Query plan for distributed top-k query. (b) Transformed query plan.

To answer a top-k query over a relation R, the querying server first locates those servers that store fragments of R by using the catalog, and constructs a query plan such as the one in Fig. 2(a). In our example, S_2 is the querying server and the relation R is fragmented in four fragments $R_1, \ldots, R_4$ stored on servers $S_1, \ldots, S_4$ respectively. Based on the query plan, each fragment R_i is scanned in ranked order (denoted in Fig. 2(a) as rank), and the top-k operator reads tuples one by one, until the k highest scoring tuples have been retrieved. In more details, the top-k operator maintains a sorted output queue and additionally a list containing the score of the last tuple from each server. Since the tuples read

from R_i are in ranked order, whenever a tuple in the output queue has a higher score than all scores in the list, it can safely be output as a result tuple. Thus, the top-k tuples are returned incrementally. Moreover, the top-k operator reads the next tuple from the fragment R_i with the tuple with the highest score in the list. Therefore, the top-k operator reads as few input tuples as possible from the fragments R_i.

This is the basic approach of answering top-k queries in a distributed data management system. Since it is important to minimize the network cost of query processing, ARTO uses a caching mechanism to take advantage of previously answered top-k queries. Thus, ARTO avoids retrieving tuples from other servers, when the cached tuples are sufficient to answer the new query. To this end, each server maintains its own cache locally, and caches the queries (and their results sets) that were processed by itself. During query processing, the querying server first uses its cache to detect whether the cached tuples are sufficient to answer the given top-k query (see Section 5). Even if the cached tuples are not sufficient, ARTO minimizes the transferred data by using as many cached tuples as possible and retrieving only the missing tuples from the remote servers through the novel use of *remainder queries* (see Section 6). To this end, the query plan is rewritten in order to take advantage of the local cache. The result of such a query transformation is shown in Fig. 2(b). Compared to the initial query plan, the top-k operator additionally retrieves tuples from the cache and performs a limited scan from the relation fragments, thus transferring only tuples that are not cached.

The combination of cached tuples and remainder queries allows ARTO to reduce the number of transferred tuples. The exact number of transferred tuples depends on the similarity of cached queries to the new query. Thus, in order to improve further the query processing performance, we extend ARTO with a *server selection* mechanism, which assigns the new query to the server with the most similar cached query. In order to facilitate this mechanism, each server publishes descriptions of its cached queries in the distributed catalog. Then, the querying server first detects the server with the most similar cached query, and re-assigns the new query to this server (see Section 7).

5 Answering Top-k Queries from Cache

In ARTO, each server autonomously maintains its own cache. More specifically, after answering a top-k query and retrieving the entire result set, the query originator is able to cache the query result. The cache $\mathcal{C} = \{C_i\}$ maintains a set of m *cache entries* C_i. Each cache entry $C_i = \{Q_i, b_i, \{p_1, \ldots, p_{k_i}\}\}$ is defined by a query $Q_i = \{R, f_i, k_i\}$, the tuples $\{p_1, \ldots, p_{k_i}\}$ that belong to the result set of Q_i, and a *threshold* b_i which is the scoring value of point p_{k_i} with respect to f_i, i.e., $b_i = f_i(p_{k_i})$. Consequently, any tuple p of the cache entry C_i has score $f_i(p) \geq b_i$. Notice that the description of a cached entry C_i that is published in the catalog consists only of $\{Q_i, b_i\}$, without the individual result tuples. For the sake of simplicity, we assume that all cache entries refer to the same relation R.

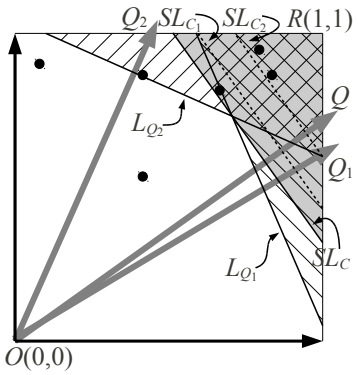

Fig. 3. Cache containing the cache entries of two queries.

Obviously, given a query $Q = \{R, f, k\}$, only cache entries that refer to relation R are taken into account for answering Q.

Fig. 3 shows a server's cache $\mathcal{C}$ that contains two cache entries, C_1 and C_2. Query Q_1 corresponds to a top-3 query, while Q_2 is a top-4 query with different weights. Their corresponding lines, L_{Q_1} and L_{Q_2}, stop at the kth point for each query respectively.

5.1 Basic Properties

In this section, we analyze when the query results of a cache $\mathcal{C}$ are sufficient to answer a top-k query Q. When this situation occurs, we say that the cache *covers* the query. Given a query $Q = \{R, f, k\}$, we identify three cases of covering: (1) a cache entry C_i covers a query defined by the same function ($f = f_i$), (2) a cache entry C_i covers a query defined by a different function ($f \neq f_i$), and (3) a set of cache entries $\{C_i\}$ cover a query defined by a different function ($f \neq f_i$, $\forall i$).

In the first case, if there exists a cache entry C_i such that the weighting vectors that define f and f_i are identical and $k \leq k_i$, then Q can be answered from the result of the cache entry C_i. More specifically, the first k data points of the cache entry C_i provide the answer to Q.

In the second case, we examine if a cache entry covers a query defined by a different function. To this end, we use the concept of *safe area* [3] SA_i of a cache entry C_i.

Definition 1. ***(Safe area)*** *The safe area SA_i of a cache entry C_i with respect to a query Q is the area defined by the right upper corner of the data space and the $(n-1)$-dimensional hyperplane SL_{C_i} that is perpendicular to the query vector, intersects the query plane L_{Q_i}, and has the largest scoring value for Q between all candidate hyperplanes.*

In Fig. 3, the lines that define the safe areas for C_1 and C_2 with respect to Q are shown as SL_{C_1} and SL_{C_2}, respectively. Given a query Q, a cache entry C_i is sufficient to answer a query Q, if it holds that the safe area SA_i of the

cache entry C_i contains at least k data points. This means that there cannot be any other tuples in the result set of Q that have not been retrieved by the query Q_i, because the safe area has been scanned during the processing of Q_i. For example, in Fig. 3, both cache entries are sufficient for answering the query Q for $k = 2$, but none of those is sufficient to answer the query Q for $k = 3$.

The third case deals effectively with the previous situation. Several cache entries need to be combined to answer the top-k query, since a single cache entry is not sufficient. To determine whether a set of cache entries can be used to answer a top-k query, we define the concept of *cache horizon*.

Definition 2. ***(Cache horizon)*** *The cache horizon of a cache $\mathcal{C} = \{C_i\}$ is defined as the borderline of the area defined by the union of query planes L_{Q_i}.*

The cache horizon represents the border between the points that are cached and those that are not. Points behind the cache horizon (towards the right upper corner of the data space) are contained in at least one cached entry, while points beyond the cache horizon (near the origin of the data space) have to be retrieved from the relation R that is stored at different servers. In Fig. 3, the cache horizon is defined by the lines L_{Q_1} and L_{Q_2} and the enclosed area has been examined to answer queries Q_1 and Q_2. In order to determine if the result set of Q is behind the cache horizon and can be answered by combining more than one cache entry, we define the *limiting point* of the cache.

Definition 3. ***(Limiting point)*** *The limiting point of a cache $\mathcal{C}$ is the point, where the hyperplane $SL_{\mathcal{C}}$ perpendicular to the query vector intersects the cache horizon, when $SL_{\mathcal{C}}$ moves from the right upper corner of the data space towards the origin.*

The area defined by the hyperplane $SL_{\mathcal{C}}$ and the right upper corner of the data space is called *safe area of the cache horizon*. If this area contains more than k data points, then Q can be answered by combining more than one cache entry.

Given a cache $\mathcal{C}$ with m cache entries $C_1, C_2, \ldots, C_m$, the limiting point of the horizon with respect to a query Q can be identified using linear programming. We construct a constraint matrix $\boldsymbol{H}$ and right-hand-side values $\boldsymbol{b}$ from the weights and thresholds of the m cache entries:

$$\boldsymbol{H} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & & & \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Given a query Q, the objective is to maximize $f = \boldsymbol{w_Q}^{\mathrm{T}} \boldsymbol{a}$, subject to the constraints $\boldsymbol{H}\boldsymbol{a} \leq \boldsymbol{b}$ and $0 < a_i < 1, \forall a_i$. The solution of this linear programming problem provides the coordinates of the limiting point. By applying the scoring function f defined by Q, we get the *cache score* $b = f(p)$ of the limiting point p. If at least k cached points p_i exist such that $f(p_i) \geq b$, then the entire result set of query Q is retrieved from the cache entries.

5.2 Cache Replacement Policy

A first observation regarding a cache entry $C_i \in \mathcal{C}$ is that it can become *redundant* due to other cache entries in $\mathcal{C}$. More formally, C_i becomes redundant if its query Q_i is covered by a set of other cache entries. Redundant cache entries can be evicted from the cache without affecting the cache's ability to answer queries. Identifying whether a cache entry C_i is redundant is achieved by solving a linear programming problem. More detailed, the objective is to maximize $\boldsymbol{w_{C_i}}^{\mathrm{T}}\boldsymbol{a}$, subject to $\boldsymbol{H'a} \leq \boldsymbol{b'}$ and $\forall a_j : 0 < a_j < 1$, where $\boldsymbol{H'}$ and $\boldsymbol{b'}$ describe the combined horizon of all cache entries except C_i. Thus, we find the limiting point p of the cache when C_i is ignored. If $b_i > f_i(p)$, the cache entry C_i is redundant and can be removed.

Applying a traditional cache replacement policy, such as LRU, is inappropriate due to the unique characteristics of our cache. The reason is that answering a top-k query from the cache may require combining tuples from more than one cache entry. Consequently, cache entries are utilized collectively, rendering any policy based on usage statistics of individual cache entries ineffective.

Motivated by this, we introduce a new cache replacement policy named *Least-Deviation Angle* (LDA), which is particularly tailored to our problem. After removing redundant entries, LDA determines the priority of a cache entry to be evicted based on deviation from the equal-weights vector $\boldsymbol{e}^{\mathrm{T}} = (1, 1, \ldots, 1)$. For each cache entry C_i, the angle $\theta_i = \arccos(\hat{\boldsymbol{w}_{C_i}} \cdot \hat{\boldsymbol{e}})$ between $\boldsymbol{e}$ and C_i is calculated and used as a measure of deviation. The entry C_i with the largest θ_i is replaced first. Intuitively, LDA penalizes cache entries that have low probability to be highly similar to other queries.

6 Remainder Queries

In the previous section, we described in which cases the entries of the cache are sufficient for retrieving the entire result set of a query Q. When this occurs, no networking cost exists for answering the query. In the case where only $k' < k$ tuples t are retrieved from the cache for which the inequality $f(t) \geq b$ holds (b is the cache score), the cache fails to return the complete result set. Then, instead of executing the entire query Q from scratch, ARTO executes a *remainder query* that retrieves only the $k - k'$ missing tuples and transfers only the necessary tuples to provide the complete result set. We first provide a short discussion showing that is more beneficial to execute a remainder query, rather than restarting a cached query $Q_i = \{R, f_i, k_i\}$ and retrieving additional tuples, so that the k tuples of Q are retrieved. Then, we define the remainder query and explain how it will be processed in order to minimize the network consumption.

6.1 Discussion

In this section, we discuss the issue whether it is more beneficial to restart a query of a cache entry C_i than posing a remainder query. Fig. 4 depicts a cache

containing one cache entry C_1 that covers the data space until the line L_{Q_1} (line DB). A query Q is posed, and the points in the cache entry until the line SL_{C_1} (line AB) are used for answering the query Q. If fewer than k tuples are enclosed in ABR, additional uncached tuples must be retrieved from remote servers. We consider two options for retrieving the remaining tuples. The first alternative is to pose a remainder query that would scan the part $FEBA$ of the data space. Since the query is executed based on the given weighting vector of Q, we can stop after retrieving k tuples exactly, i.e., at the query line L_Q (FE). The other alternative is to restart the cached query Q_1. In this case, we can take advantage of all k_1 data points of the cache entry C_1 (i.e., we save the cost of scanning DBA). On the other hand, in order to be sure that we have retrieved all tuples of the result set of Q we have to scan a larger area at least until the line GE.

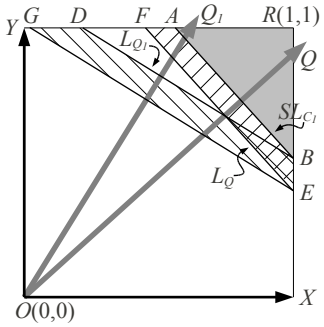

Fig. 4. Areas examined by the remainder query vs. restarting a query Q_1.

If data is uniformly distributed, the number of tuples retrieved is proportional to the area of the data space that is scanned. For the sake of simplicity, we assume that the query line of any query lies in the the upper right triangle of the data space. This means that we have scanned less than half the data space, in order to retrieve the result set of any query, which is an acceptable assumption since usually the values of k are small. In our example, the area of $FEBA$ is smaller than the area of $GEBD$, and the retrieved tuples are expected to be fewer when the remainder query is used. In the following, we prove that this always holds for the 2-dimensional case, when the query line does not cross the diagonal line XY. Similar conclusions can been drawn for arbitrary dimensionality.

Theorem 1. *Given a 2-dimensional data space, if all query lines do not cross the diagonal line XY, a smaller area is scanned if the remainder query is executed than if continuing a cached query.*

Proof. Using the areas of Fig. 4, it suffices to show that the area of trapezoid $FEBA$ is smaller than the area of trapezoid $GEBD$. The two trapezoids share one common side, namely EB. Furthermore, it is always the case that $BD > BA$ and $GE > FE$. Based on Thales' theorem about the ratios of line segments that

are created if two intersecting lines are intercepted by a pair of parallels, we derive that $\frac{FA}{AR} = \frac{EB}{BR}$ (1) and $\frac{GD}{DR} = \frac{EB}{BR}$ (2). From (1) and (2) we conclude that $\frac{FA}{AR} = \frac{GD}{DR}$. Since $DR > AR$, we derive that $GD > FA$. Therefore, three sides of $FEBA$ are smaller than the corresponding three sides of $GEBD$ and the remaining fourth side BE is common. Hence, the area of $FEBA$ is smaller than the area of $GEBD$.

6.2 Processing of Remainder Queries

Given a query Q and a cache score b, a remainder query is defined as $Q' = \langle R, f, k - k', b \rangle$, where k' is the number of cached tuples p such that $f(p) \geq b$. Any server S_i that stores a fragment R_i of the relation R receives the remainder query Q'. Independently from the implementation of the top-k operator at S_i, the server S_i transfers to the querying server only tuples p such that $f(p) \leq b$. Thus, it avoids transferring tuples that are already cached and lie in the safe area of the querying server.

To further limit the number of transferred tuples to the querying server, S_i filters out some of the locally retrieved tuples by using the cache horizon before transferring them. Even though some tuples lie outside the safe area, they are available at the querying server in some cache entry. For example, in Fig. 4, the remainder query has to start scanning the data space from the line SL_{C_1} until k tuples are retrieved, i.e., the remainder query fetches new tuples until the query line L_Q. Nevertheless, the points that fall in the triangle DBA are already available at the querying server in the cache entry C_1. These tuples do not need to be transferred, thus minimizing the number of transferred data. In order for S_i to be able to filter out tuples based on the cache horizon, S_i retrieves the descriptions of all cache entries from the querying server. Then, all tuples p such that there exists a cache entry C_i such that $f_i(p) > b_i$ are not transferred to the querying server, since these tuples are stored locally in the cache. The querying server combines the tuples received from the servers S_i with the tuples in the cache and produces the final result set of the query Q. To summarize, the cache horizon is used to limit the remainder query, which means that the whole cache is exploited and a minimal number of tuples is fetched from other servers.

7 Server Selection

The problem of server selection is to identify the best server for executing the top-k operator. While the rank scan operators must be located at the servers that store the relation fragments, the top-k operator can be placed on any server. Our server selection algorithm assigns the top-k operator to the server that results in the most cost-efficient query execution in terms of network cost.

Intuitively, the best server S^* to process the top-k query $Q = \{R, f, k\}$ is the one that can return as many as possible from the k result tuples from its local cache, thereby reducing the amount of the remaining result tuples that need to be fetched. To identify S^*, we need to inspect the cache entries for each server.

Algorithm 1 Server selection

```
1: Input: Query Q = {R, f, k}, Servers S
2: Output: Server S* that will process Q
3: for (∀S_i ∈ S) do
4:    {(Q_j, b_j)} ← catalog.getCacheDesc(S_i)
5:    score(S_i) ← computeLimitingPoint({(Q_j, b_j)})
6:    if (score(S_i) < minScore) then
7:       S* ← S_i
8:       minScore ← score(S_i)
9:    end if
10: end for
11: return S*
```

This operation is efficiently performed using the distributed catalog. In more detail, the catalog can report the descriptions of cache entries $\mathcal{C} = \{C_i\}$ of any server, where a description of C_i consists of $\{Q_i, b_i\}$. Based on this information, the limiting point of the server is calculated, as described in Section 5. Based on the limiting point, we compute the score of each server by applying the function f of the query Q. The server S^* with the smallest score is selected because this server has the largest safe area and therefore is the best candidate to process the top-k query. Algorithm 1 provides the pseudocode for the server selection mechanism.

8 Experiments

In this section, we present an experimental evaluation of ARTO. We have implemented ARTO into the DASCOSA-DB [14] distributed database management system and use this implementation to investigate the effect of different parameters, query workloads and datasets.

Experimental setup. DASCOSA-DB provides a global distributed catalog based on a distributed hash table, and this catalog was used to implement publishing and lookup of cache entries' descriptions. Experiments were performed for varying a) number of servers, b) values of k, and c) cache size. We used three datasets, with uniform, correlated and anti-correlated distributions. Each dataset consisted of 1,000,000 5-dimensional tuples. The datasets were distributed horizontally and uniformly over all servers. A separate querying server issued queries to the system and received the results. The weights of the queries were uniformly distributed.

Each experiment was performed both without caching and with ARTO enabled. In addition, we did experiments with a hit/miss implementation where the cache was used only if it were sufficient to answer the complete query. This is conceptually similar to previously proposed methods, e.g., BRANCA [20]. We measured the number of tuples accessed a) locally on the querying server using its cache, and b) from remote servers.

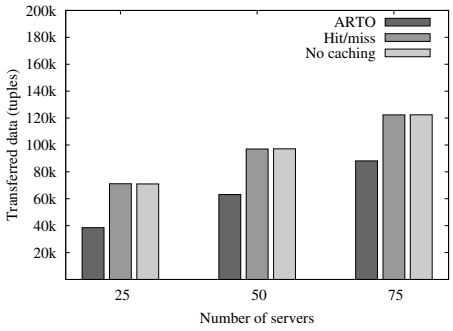

Fig. 5. Transferred data for 1,000 queries and uniform data distribution.

Varying number of servers. In this experiment, ARTO was evaluated with uniformly distributed, correlated and anti-correlated datasets. Each dataset was distributed over 25, 50 and 75 servers. A workload of 1,000 queries with uniformly distributed weights and $k = 50$ were issued. Each server had 10,000 bytes cache, which allows for 4 complete top-50 results to be cached at each server.

Fig. 5 shows the total number of tuples that are transferred over the network for the query workload using a uniform dataset. We observed similar results for correlated and anti-correlated datasets, which hints that the performance of our approach is stable across different data distributions. The combination of server selection with remainder queries causes a major improvement in network communication costs, even with such a small cache size (4 cache entries). The advantage of ARTO is clearly demonstrated when comparing to the hit/miss strategy, which performs poorly, as it requires k tuples in the safe area to use the cache. Since cache misses are frequent, the complete top-k query has to be executed. The results of hit/miss are just barely better than without caching, while ARTO achieves significant improvements.

Varying *k*. The size of k affects the gain that can be obtained from caching. If k is very small, there are not that many remote accesses that can be replaced by local accesses. In this experiment, the caching method was tested with varying values for k. A uniform dataset of 1,000,000 tuples on 25 servers was used. Each site had 10,000 bytes cache. The results displayed in Fig. 6 show how the number of total and local accesses increases with increasing k. ARTO always accesses significantly more local tuples compared to the competitor approaches. Around $k = 100$, the number of local tuples accessed starts to decrease. This is because the cache is of a limited size. With $k = 100$, only two complete top-k results fit in cache. Even in this extreme case, ARTO still manages to access a high percentage of the total tuples from the local cache, thereby saving communication costs.

Cache size. In order to study the effect of cache size in more detail, we performed an experiment where we gradually increased the cache size up to 50,000 bytes, i.e., more than 20 complete results. We fixed $k = 50$ and used a uniform dataset of 1,000,000 tuples on 25 servers. The results are shown in Fig. 7. As the cache size increases, more top-k queries can be cached, thus enlarging the

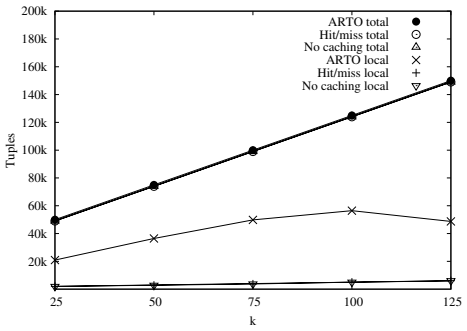

Fig. 6. Results of queries with varying k.

safe area. Consequently, ARTO reduces the number of transferred data (remote tuples accessed). In contrast, the hit/miss strategy always results in cache misses and cannot reduce the amount of transferred data.

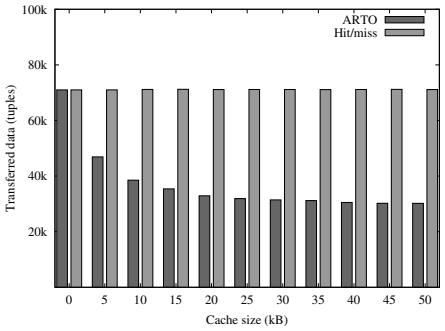

Fig. 7. Results of queries with varying cache size.

9 Conclusion

In this paper, we present ARTO, a novel framework for efficient distributed top-k query processing. ARTO relies on a caching mechanism that reduces the network communication costs significantly by retrieving as many tuples as possible from the local cache. In order to retrieve the missing tuples, we define the remainder query that transfers only the tuples that are not stored in the cache by filtering out tuples based on the cache horizon. Moreover, ARTO provides a server selection mechanism that assigns a new top-k query to the most promising server. We have implemented our framework in the DASCOSA-DB database management system. The results of the experiments show considerable improvements in network communication costs.

References

1. Akbarinia, R., Pacitti, E., Valduriez, P.: Reducing network traffic in unstructured P2P systems using top-k queries. Distributed and Parallel Databases 19(2-3), 67–86 (2006)
2. Akbarinia, R., Pacitti, E., Valduriez, P.: Best position algorithms for top-k queries. In: Proceedings of VLDB (2007)
3. Baikousi, E., Vassiliadis, P.: View usability and safety for the answering of top-k queries via materialized views. In: Proceedings of DOLAP (2009)
4. Balke, W.T., Nejdl, W., Siberski, W., Thaden, U.: Progressive distributed top-k retrieval in peer-to-peer networks. In: Proceedings of ICDE (2005)
5. Bruno, N., Chaudhuri, S., Gravano, L.: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM Trans. Database Syst. 27(2), 153–187 (2002)
6. Cao, P., Wang, Z.: Efficient top-k query calculation in distributed networks. In: Proceedings of PODC (2004)
7. Chandramouli, B., Bond, C.N., Babu, S., Yang, J.: Query suspend and resume. In: Proceedings of SIGMOD (2007)
8. Chaudhuri, S., Gravano, L.: Evaluating top-k selection queries. In: Proceedings of VLDB (1999)
9. Chaudhuri, S., Gravano, L., Marian, A.: Optimizing top-k selection queries over multimedia repositories. IEEE Trans. on Knowledge and Data Engineering 16(8), 992–1009 (2004)
10. Chaudhuri, S., Kaushik, R., Ramamurthy, R., Pol, A.: Stop-and-restart style execution for long running decision support queries. In: Proceedings of VLDB (2007)
11. Das, G., Gunopulos, D., Koudas, N., Tsirogiannis, D.: Answering top-k queries using views. In: Proceedings of VLDB (2006)
12. Güntzer, U., Balke, W.T., Kießling, W.: Optimizing multi-feature queries for image databases. In: Proceedings of VLDB (2000)
13. Hauglid, J.O., Nørvåg, K.: Proqid: partial restarts of queries in distributed databases. In: Proceedings of CIKM (2008)
14. Hauglid, J.O., Nørvåg, K., Ryeng, N.H.: Efficient and robust database support for data-intensive applications in dynamic environments. In: Proceedings of ICDE (2009)
15. Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER: A system for the efficient execution of multi-parametric ranked queries. In: Proceedings of SIGMOD (2001)
16. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 40(4) (2008)
17. Marian, A., Bruno, N., Gravano, L.: Evaluating top-k queries over web-accessible databases. ACM Trans. Database Syst. 29(2), 319–362 (2004)
18. Michel, S., Triantafillou, P., Weikum, G.: KLEE: A framework for distributed top-k query algorithms. In: Proceedings of VLDB (2005)
19. Vlachou, A., Doulkeridis, C., Nørvåg, K., Vazirgiannis, M.: On efficient top-k query processing in highly distributed environments. In: Proceedings of SIGMOD (2008)
20. Zhao, K., Tao, Y., Zhou, S.: Efficient top-k processing in large-scaled distributed environments. Data and Knowledge Engineering 63(2), 315–335 (2007)