Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 63 (2007) 315–335
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Efficient top-k processing in large-scaled distributed environments

Keping Zhao [a], Yufei Tao [b,*], Shuigeng Zhou [c]

[a] *Microsoft China, 3 Hong Qiao Road, Shanghai 200030, China*
[b] *Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, New Territories, Hong Kong*
[c] *Department of Computer Science and Engineering, Fudan University, 220 Handan Road, Shanghai 200433, China*

Received 5 October 2006; received in revised form 30 January 2007; accepted 2 March 2007
Available online 2 April 2007

Abstract

The rapid development of networking technologies has made it possible to construct a distributed database that involves a huge number of sites. Query processing in such a large-scaled system poses serious challenges beyond the scope of traditional distributed algorithms. In this paper, we propose a new algorithm BRANCA for performing top-k retrieval in these environments. Integrating two orthogonal methodologies "semantic caching" and "routing indexes", BRANCA is able to solve a query by accessing only a small number of servers. Our algorithmic findings are accompanied with a solid theoretical analysis, which rigorously proves the effectiveness of BRANCA. Extensive experiments verify that our technique outperforms the existing methods significantly.

Keywords: Top-k; Distributed database; Caching

1. Introduction

The rapid development of communication technology has significantly reduced the cost of maintaining a fast and reliable network. This, in turn, provides the opportunity of constructing a large-scaled distributed database, where the underlying data is partitioned onto a sizable number (e.g., thousands) of sites. Query processing in such systems encounters numerous difficult issues that do not exist in a conventional distributed system, where the number of sites is much smaller. In particular, we can no longer assume that the participating servers are aware of each other, as is a prerequisite of most text-book distributed solutions. In particular, in many applications, a server only keeps information about its neighboring servers in a topology, such that adding/removing a server requires updating the information only in a small part of the network.

* Corresponding author. Tel.: +852 26098437; fax: +852 26035024.
E-mail addresses: kepingz@microsoft.com (K. Zhao), taoyf@cse.cuhk.edu.hk (Y. Tao), sgzhou@fudan.edu.cn (S. Zhou).
URL: http://www.cse.cuhk.edu.hk/~taoyf (Y. Tao).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2007.03.012

We are interested in top-k processing in such scenarios. As an example, consider a *real-property information system* that integrates the records of houses recently sold in a state. The records are periodically (e.g., every week) appended by realty agencies, and conform to a universal relational schema such as PROPERTY(*address*, *size*, *NSR*, *NAQ*, ...), where *NSR* and *NAQ* represent "neighborhood security rating" and "neighborhood air quality", respectively. The objective is to efficiently answer queries of the form:

SELECT *address* FROM PROPERTY
ORDER BY $0.3 \cdot size + 0.5 \cdot NSR + 0.2 \cdot NAQ$ DESC
STOP AFTER 10

which returns the top-10 houses maximizing a function that computes a weighted sum of the columns of PROPERTY (e.g., the weight on *NSR* is 0.5). The system should support queries with any weighting.

1.1. Problem definition

We consider a distributed database that integrates a large number of servers in a network. Our discussion assumes the existence of a *logical topology*, which is an acyclic graph with each vertex corresponding to a server. Such a topology should be distinguished with the underlying physical topology (which refers to the physical connections among servers). The acyclic graph can be easily implemented, by requiring each vertex to remember the IP addresses of its adjacent vertices. In the sequel, all occurrences of "topology" refer to the logical topology.

The database contains a relation R with d numeric *ranking attributes*. Given a tuple o, we denote its value on the ith ($1 \leqslant i \leqslant d$) attribute as $o[i]$. Without loss of generality, we assume that each $o[i]$ falls in the range $[0,1]$; equivalently, the d-dimensional vector $\vec{o} = \{o[1], o[2], \ldots, o[d]\}$ distributes in the unit *ranking space* $[0,1]^d$. Each server X stores a horizontal partition of R, denote it as $X.ds$, which is a subset of the data in R. Therefore, R equals $\cup_{\text{each server } X} X.ds$. Each server can store an *arbitrary fraction* of R. This is different from many P2P systems [31,33,25], where each server is responsible for retaining only tuples falling in a particular region (i.e., the server's "jurisdiction region") of the ranking space.

A server can issue a top-k query at any time, which, in addition to the parameter k, specifies a d-dimensional vector $\vec{q} = \{q[1], q[2], \ldots, q[d]\}$, where $q[i]$ is a positive *weight* on the ith attribute. The vector defines a *preference function* $f_{\vec{q}}$, which computes a score $f_{\vec{q}}(o)$ for every tuple o

$$f_{\vec{q}}(o) = \vec{q} \cdot \vec{o} = \sum_{i=1}^{d} q[i] \cdot o[i] \tag{1}$$

The objective is to find the k tuples in R with the highest scores, and transmit them to the server initiating the query.

1.2. Motivation and contributions

Although distributed top-k processing has been very well studied (see Section 6 for a survey), the existing methods assume *vertical* partitioning (where each server stores an entire column of the relation), and thus are inapplicable in our context. The only top-k solution on horizontally partitioned data is due to Balke et al. [1], who present an algorithm that minimizes the number of tuple transfers. Unfortunately, the minimization does not necessarily result in the least network communication, because two servers may need to exchange many messages before a tuple is transmitted. To alleviate the problem, Balke et al. [1] develop an index which can return the query result earlier, if the *same* query has been processed before. While the index is useful in the settings of [1], it is not in our problem where two queries are rather unlikely to be equivalent (note that equivalence implies that both queries have identical weights on all attributes).

Motivated by this, we develop *branch caching* (BRANCA), an efficient technique for supporting top-k queries in large-scaled distributed databases. The core of BRANCA is the integration of two orthogonal methodologies: (i) "semantic caching" [32] of query results, which are extensively applied in mobile computing, and

(ii) "routing indexes" [12], which constitute a popular technique for fast query processing over the Internet. Specifically, semantic caching improves conventional cache-replacement schemes (such as "least recently used"), by retaining data that is more likely to be retrieved by subsequent queries. A routing index, on the other hand, tags additional information to each entry in a routing table, which serves as a brief summary of the data in the subnet corresponding to that entry. In BRANCA, each participating server caches data fetched from various subnets in the past top-k search. Given a new query, the cache allows a server to forward the query only to a small part of the network that may contain the final results. We accompany our algorithmic findings with a solid theoretical analysis to prove the effectiveness of BRANCA.

The rest of the paper is organized as follows. Section 2 explains the fundamental concept of "branch caches". Section 3 presents the query algorithm of BRANCA, while Section 4 analyzes its performance. Section 5 elaborates the maintenance of branch caches during query processing. Section 6 reviews the previous studies related to this paper, and clarifies their differences from our work. Section 7 contains an extensive experimental evaluation that validates the efficiency of BRANCA in practice. Finally, Section 8 concludes the paper with directions for future work.

2. Branch caches

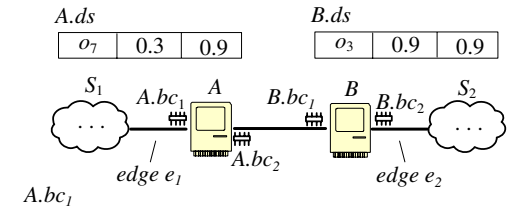
Let X be any server in the underlying topology, and e an edge adjacent to X. The topology becomes two disjoint subgraphs if e is removed. We say that the subgraph, which does *not* contain X, is a *branch subgraph* of X, and denoted as $X.bsg(e)$. When the context is clear, we will use $X.bsg(e)$ also to represent the set of tuples of R that are stored in this subgraph.

Apparently, a server X has as many branch subgraphs as the number of its neighbors in the topology. For each subgraph $X.bsg(e)$ (where e is an edge linking X to one of its neighbors), X maintains a *branch cache* $X.bc(e)$, which retains the results of the previous top-k queries, *with respect to* the data in $X.bsg(e)$. Specifically, each item in $X.bc(e)$ has the form

$$\{\vec{q}, k, result(\vec{q}, k)\}$$

where $\vec{q}$ is the query vector of a past top-k inquiry, and $result(\vec{q}, k)$ the set of k tuples in $X.bsg(e)$ achieving the highest scores for the preference function $f_{\vec{q}}$ (Eq. (1)). We emphasize that the tuples in $result(\vec{q}, k)$ are *not* the top-k result in the entire network – they are only the result in a branch subgraph.

We illustrate the above concepts using Fig. 1, assuming that relation R has $d = 2$ ranking attributes. Server A, for instance, has two neighbors: B and another server connected by edge e_1 (omitted from the example). Hence, A has two branch subgraphs: (i) $A.bsg(e_1)$, which includes all the servers in set S_1, and (ii) $A.bsg(AB)$, containing B and set S_2. Accordingly, A maintains two branch caches $A.bc_1$ and $A.bc_2$ for subgraphs $A.bsg(e_1)$



$A.bc_1$

	q	k	$result(q,k)$
I_1	{0.7,0.3}	1	{o_1(0.1,0.4)}
I_2	{0.3,0.7}	1	{o_2(0.4,0.1)}

The content of $B.bc_1$ is omitted.

$A.bc_2$

	q	k	$result(q,k)$
I_3	{0.6,0.4}	1	{o_3(0.9,0.9)}
I_4	{0.7,0.3}	3	{o_3(0.9,0.9), o_4(0.9,0.25), o_5(0.6, 0.4)}

$B.bc_2$

	q	k	$result(q,k)$
I_5	{0.3,0.7}	1	{o_6(0.2,0.6)}
I_6	{0.6,0.4}	2	{o_4(0.9,0.25), o_5(0.6,0.4)}

Fig. 1. Branch cache examples.

and $A.bsg(AB)$, respectively. Fig. 1 also demonstrates the content of these caches. The first item (denoted as I_1 in the example) of $A.bc_1$ indicates that, among all the data stored in S_1, the tuple maximizing $f_{\{0.7,0.3\}}$ is o_1 (its values for the two ranking attributes are 0.1 and 0.4, respectively). Similarly, judging from the second item I_4 in $A.bc_2$, we can assert that tuples o_3, o_4, and o_5 have the highest scores for function $f_{\{0.7,0.3\}}$ in the subgraph $A.bsg(AB)$.

Likewise, B also has two branch subgraphs, and the same number of branch caches. Fig. 1 demonstrates the details of $B.bc_2$ (we omit the items in $B.bc_1$ since they are not relevant to the following discussion). Notice that the entries in $A.bc_1$, $A.bc_2$, $B.bc_2$ indicate that at least three queries have been processed before, and they have different $\vec{q}$ and k. Note that it is possible for various caches to retain information about different queries (which is decided by a cache-replacement strategy explained in Section 5).

3. Top-k processing with BRANCA

BRANCA performs top-k retrieval based on the following rationale. First, it attempts to derive as many results as possible from the branch caches (of the servers that have been contacted), and terminates the search as soon as no other tuple can have a higher score than the current results. Second, the algorithm allows every server that participated in the query execution to benefit, by incorporating useful information into its own caches, in order to reduce the cost of future queries. In the sequel, we will first describe the strategy from a high level (Section 3.1), and then clarify the details of pruning (Section 3.2).

3.1. The high level algorithm

Fig. 2 presents the pseudo-code of top-k search with BRANCA. Next, we explain the algorithm using the running example of Fig. 1.

Example 4.1. Consider that server A issues a query with $\vec{q} = \{0.5, 0.5\}$ and $k = 2$. The function in Fig. 2 is invoked as $\texttt{TOPK}(A, \{0.5, 0.5\}, 2, \emptyset)$. Statements 2–4 collect the set S of objects that belong to either $A.ds$, or any of the branch caches of A. In Fig. 1, the content of $A.ds$, which involves a single object o_7, is illustrated on top of the figure. Clearly, $S = \{o_1, o_2, o_3, o_4, o_5, o_7\}$. Then, the algorithm obtains the k (=2) tuples o_3, o_7 from S that maximize the preference function $f_{\{0.5,0.5\}}$ (Statement 5), and place them in $rslt$ (=$\{o_3, o_7\}$).

Algorithm TOPK (X, $\vec{q}$, k, Z)

X: the server on which the algorithm is being executed
$\vec{q}$: the query vector of the top-k inquiry
k: as in the previous sentence
Z: the server from which the query is relayed to X (set to $\emptyset$ if the query is initiated by X)

1. $rslt = \emptyset$ //the query result
2. $S = X.ds$
3. for each neighbor Y of X that is different from Z
4. $S = S \cup \{$the tuples in $X.bc(XY)\}$
5. $rslt$ = the k tuples in S that maximize $f_{\vec{q}}$ //in case S has fewer than k tuples, $rslt = S$
6. $list_{contact} = \emptyset$ //list of servers to be contacted
7. for each neighbor Y of X that is different from Z
8. if ($rslt$ has fewer than k tuples) or (PRUNE-BRANCH($X.bc(XY)$, $\vec{q}$, k, $rslt$) = false)
   //PRUNE-BRANCH is presented in Figure 3
9. $list_{contact} = list_{contact} \cup \{Y\}$
10. for each server Y in $list_{contact}$ do simultaneously
11. $rslt_Y$ = TOPK(Y, $\vec{q}$, k, X) //recursively request results from Y
12. $rslt = rslt \cup rslt_Y$
13. $X.bc(XY) = X.bc(XY) \cup \{\vec{q}, k, rslt_Y\}$ //cache the results
14. if the cache space overflows then CACHE-REPLACE //presented in Figure 6
15. return the k tuples in $rslt$ maximizing $f_{\vec{q}}$

Fig. 2. The top-k algorithm.

Next, TOPK (Statements 6–9) inspects whether each branch subgraph of A can be eliminated from further consideration. This is accomplished by function PRUNE-BRANCH which, as analyzed in the next section, makes the decision according to the data in *rslt* and the corresponding branch cache. Continuing the example:

- (*The first application of* PRUNE-BRANCH) For the neighbor of A in S_1 (see Fig. 1), PRUNE-BRANCH (Statement 8) is invoked with parameters $A.bc_1$, $\{0.5, 0.5\}$, 2, $\{o_3, o_7\}$. The function returns true (the reason will be clear after the next section), meaning that if a tuple o in S_1 belongs to the final top-k set, o must necessarily appear in $A.bc_1$. Hence, S_1 does not need to be explored.
- (*The second application*) For the neighbor B of A, PRUNE-BRANCH is invoked with parameters $A.bc_2$, $\{0.5, 0.5\}$, 2, $\{o_3, o_7\}$. This time, PRUNE-BRANCH returns false, and hence, B is added to $list_{contact}$.

For each server Y in $list_{contact}$, TOPK *simultaneously* requests the top-k result $rslt_Y$ in the branch subgraph $A.bsg(AY)$. In Fig. 1, $list_{contact}$ has a single server B, and thus, A sends the query to B. This is performed at Statement 11, which has the form $rslt_B = \texttt{TOPK}(B, \{0.5, 0.5\}, 2, A)$ here.

The execution of TOPK on server B proceeds in the same manner as described earlier, except that the branch connected by edge AB is ignored (note the part "different from Z" in Statements 3 and 7). Specifically, at Statements 1–5, it obtains a set *rslt* containing two objects $\{o_3, o_4\}$ in $B.ds \cup B.bc_2$ that maximize $f_{\{0.5, 0.5\}}$ (i.e., $B.bc_1$ is not taken into account). Then,

- (*The third application*) At Statement 8 (for the neighbor of B in S_2), PRUNE-BRANCH is invoked with parameters $B.bc_2$, $\{0.5, 0.5\}$, 2, and $\{o_3, o_4\}$. The function returns true, pruning the branch subgraph S_2.

Therefore, $list_{contact} = \emptyset$ at Statement 10, and Statements 11–13 are not executed. Since the branch caches of B do not have any changes, Statement 14 has no effect. TOPK finishes on server B, and returns $\{o_3, o_4\}$.

Now the control returns to Statement 11 at server A, with $rslt_B$ (i.e., the $rslt_Y$ in the statement) set to $\{o_3, o_4\}$. The algorithm proceeds by incorporating (Statement 12) the two objects of $rslt_B$ into *rslt*. Since *rslt* was $\{o_3, o_7\}$ before A relayed the query to B, after incorporating $rslt_B$, *rslt* becomes $\{o_3, o_7, o_4\}$. At Statement 13, $rslt_B$ is inserted in $A.bc_1$, creating an item $\{\{0.5, 0.5\}, 2, \{o_3, o_4\}\}$ in $A.bc_1$. If, after the insertion, the space allocated for caching is exhausted, a cache-replacement algorithm (the topic of Section 5) is invoked to expunge the least useful information, taking into account all branch caches of A (Statement 14). Finally, the query results are the two objects o_3, o_7 in *rslt* that maximize $f_{\{0.5, 0.5\}}$.

The performance of TOPK is determined by the effectiveness of PRUNE-BRANCH, which was applied three times in the above execution. In the next section, we will discuss the pruning details for each application.

3.2. Pruning with branch caching

Before explaining PRUNE-BRANCH, we first tackle a relevant problem. Specifically, let X be a server, $X.bsg$ any of its branch subgraphs, and $X.bc$ the branch cache responsible for $X.bsg$. *Given a query vector* $\vec{q}$ *and a value* λ, *can any tuple* o in $X.bsg$ satisfy the following conditions simultaneously: (i) o has a score $f_{\vec{q}}(o) > \lambda$, and (ii) o does not appear in $X.bc$?

The answer to the above question is directly related to the pruning strategies of PRUNE-BRANCH. To understand this, imagine that we have a top-k query $\vec{q}$, and λ is the kth highest score observed from all the tuples that have been accessed. Now server X wants to decide whether the subgraph $X.bsg$ may have any tuple o that has not been considered, but its score is higher than λ. If o does not exist, $X.bsg$ can be safely pruned, because no data there can possibly update our current top-k set. Note that o should *not* belong to $X.bc$. This is because $X.bc$ is contained locally in X, and thus, all the data in $X.bc$ has already been considered. Therefore, we need to search $X.bsg$ only if our target question has a positive answer.

Without loss of generality, we assume that $X.bc$ includes c cache items:

$$\{\vec{q}_1, k_1, result(\vec{q}_1, k_1)\}, \ldots, \{\vec{q}_c, k_c, result(\vec{q}_c, k_c)\}$$

For each $i \in [1, c]$, denote λ_i as the lowest score (with respect to $f_{\vec{q}_i}$) of the k_i tuples in $result(\vec{q}_i, k_i)$. For instance, $c = 2$ for the branch cache $B.bc_2$ in Fig. 1. According to the first item in $B.bc_2$, $\vec{q}_1 = \{0.3, 0.7\}, k_1 = 1$,

$result(\vec{q}_1, k_1) = \{o_6\}$, and $\lambda_1 = f_{\vec{q}_1}(o_6) = 0.48$. Similarly, by the second item of $B.bc_2, \vec{q}_2 = \{0.6, 0.4\}$, $k_2 = 2, result(\vec{q}_2, k_2) = \{o_4, o_5\}$. Between o_4 and o_5, the latter has a lower score for $f_{\vec{q}_2}$, leading to $\lambda_2 = f_{\vec{q}_2}(o_5) = 0.52$.

The vector $\vec{q}_i$ and value λ_i ($1 \leqslant i \leqslant c$) decide a d-dimensional plane $\mathscr{H}(\vec{q}_i, \lambda_i)$, whose equation is $\vec{q}_i \cdot \vec{x} = \lambda_i$, where $\vec{x}$ is a variable in the (d-dimensional) ranking space. $\mathscr{H}(\vec{q}_i, \lambda_i)$ divides the ranking space into two half-spaces: one contains the origin of the space, and the other does not. We represent these half-spaces as $\mathscr{H}^-(\vec{q}_i, \lambda_i)$ and $\mathscr{H}^+(\vec{q}_i, \lambda_i)$, respectively. Formally, $\mathscr{H}^-(\vec{q}_i, \lambda_i)$ covers all the points $\vec{x}$ satisfying

$$\vec{q}_i \cdot \vec{x} \leqslant \lambda_i \tag{2}$$

and for $\mathscr{H}^+(\vec{q}_i, \lambda_i)$:

$$\vec{q}_i \cdot \vec{x} > \lambda_i \tag{3}$$

Intuitively, $\mathscr{H}^-(\vec{q}_i, \lambda_i)$ (or $\mathscr{H}^+(\vec{q}_i, \lambda_i)$) consists of all the points in the ranking space whose scores of $f_{\vec{q}_i}$ are at most (or larger than) λ_i. Similarly, $\vec{q}$ and λ (as in our target problem stated at the beginning of this section) also determine a half-space $\mathscr{H}^+(\vec{q}, \lambda)$ covering all the points $\vec{x}$ with the property:

$$\vec{q} \cdot \vec{x} > \lambda \tag{4}$$

We have:

Lemma 1. *Let $\mathscr{H}^+(\vec{q}_1, \lambda_1), \mathscr{H}^+(\vec{q}_2, \lambda_2), \ldots, \mathscr{H}^+(\vec{q}_c, \lambda_c)$, and $\mathscr{H}^+(\vec{q}, \lambda)$ be the half-spaces described earlier. If $\mathscr{H}^+(\vec{q}, \lambda)$ is completely enclosed in*

$$\mathscr{H}^+(\vec{q}_1, \lambda_1) \cup \mathscr{H}^+(\vec{q}_2, \lambda_2) \cup \cdots \cup \mathscr{H}^+(\vec{q}_c, \lambda_c) \tag{5}$$

then there is no tuple o in $X.bsg$ such that (i) $f_{\vec{q}}(o) > \lambda$ *and* (ii) *o does not appear in $X.bc$.*

Proof. We prove the lemma by contradiction. Assume that there exists such a tuple o. According to condition (i), o appears in $\mathscr{H}^+(\vec{q}, \lambda)$. According to condition (ii), o is not in any $\mathscr{H}^+(\vec{q}_i, \lambda_i)$ for all $i \in [1, c]$ (otherwise, o would be in the result of some query cached in $X.bc$, and hence, should appear in $X.bc$). Thus, we have decided that o falls in $\mathscr{H}^+(\vec{q}, \lambda)$, but not in the region of Formula 5. This contradicts the fact that $\mathscr{H}^+(\vec{q}, \lambda)$ is completely enclosed in the region of Formula 5. □

In the following discussion, we refer to the region represented by Formula 5 as the *pruning region* of branch cache $X.bc$. Examining whether $\mathscr{H}^+(\vec{q}, \lambda)$ falls entirely in the pruning region (so that the subgraph $X.bsg$ can be pruned) is a Linear Programming (LP) problem. To clarify this, we need an alternative form of Lemma 1.

Corollary 1. *If*

$$\mathscr{H}^-(\vec{q}_1, \lambda_1) \cap \mathscr{H}^-(\vec{q}_2, \lambda_2) \cap \cdots \cap \mathscr{H}^-(\vec{q}_c, \lambda_c) \cap \mathscr{H}^+(\vec{q}, \lambda) = \emptyset \tag{6}$$

then there is no tuple o in $X.bsg$ such that (i) $f_{\vec{q}}(o) > \lambda$ *and* (ii) *o does not appear in $X.bc$.*

Proof. The corollary is correct because, $\mathscr{H}^+(\vec{q}, \lambda)$ is contained in the region of Formula 5 if and only if Eq. (6) is true. □

If Eq. (6) does not hold, it follows that there exists a point $\vec{x}$ in the ranking space, such that $\vec{x}$ satisfies c (linear) constraints given by Inequality (2) (for $1 \leqslant i \leqslant c$), and the constraint of Inequality (4). Finding such $\vec{x}$ or claiming its absence can be achieved using a standard LP algorithm. In our implementation, we adopt the Simplex algorithm [18]. Based on the above analysis, Fig. 3 presents the pseudo-code of `PRUNE-BRANCH`. We will illustrate the algorithm, as well as Lemma 1, by clarifying the details of its three applications in Example 4.1 (using the servers of Fig. 1).

Example 4.2 (*Pruning details in Example 4.1*). The first application of `PRUNE-BRANCH` occurs after `TOPK`, when executed at server A, obtains $rslt = \{o_3, o_7\}$, which are the two objects in $A.ds \cup A.bc_1 \cup A.bc_2$ that maximize $f_{\{0.5, 0.5\}}$. Here, `PRUNE-BRANCH` is invoked with parameters $A.bc_1$, $\vec{q} = \{0.5, 0.5\}$, $k = 2$, and $rslt = \{o_3, o_7\}$. Statement 1 of Fig. 3 calculates $\lambda = f_{\{0.5, 0.5\}}(o_7) = 0.6$, which is the lowest score of the tuples in $rslt$ for the query vector $\vec{q}$. As in Inequality 4, $\vec{q}$ and λ determine a half-space $\mathscr{H}^+(\vec{q}, \lambda)$ (Statement 2). In Fig. 4a, where

Algorithm PRUNE-BRANCH (bc, $\vec{q}$, k, S)

bc: a branch cache
$\vec{q}$: the query vector of the top-k inquiry
k: as in the previous sentence
S: a set of k tuples

1. λ = the smallest score of the tuples in S
2. $\mathcal{H}^+(\vec{q}, \lambda)$ = the half-space represented by Inequality 4
3. c = the number of items in bc
4. for i = 1 to c //consider the i-th item $\{\vec{q}_i, k_i, result(\vec{q}_i, k_i)\}$
5. λ_i = the smallest score of tuples in $result(\vec{q}_i, k_i)$
6. $\mathcal{H}^+(\vec{q}_i, \lambda_i)$ = the half-space represented by Inequality 3
7. check whether $\mathcal{H}^+(\vec{q}, \lambda)$ falls entirely in the region represented by Formula 5
   /* for this purpose, invoke a simplex algorithm based on Corollary 1 */
8. if yes then return true; otherwise, return false

Fig. 3. The PRUNE-BRANCH algorithm.

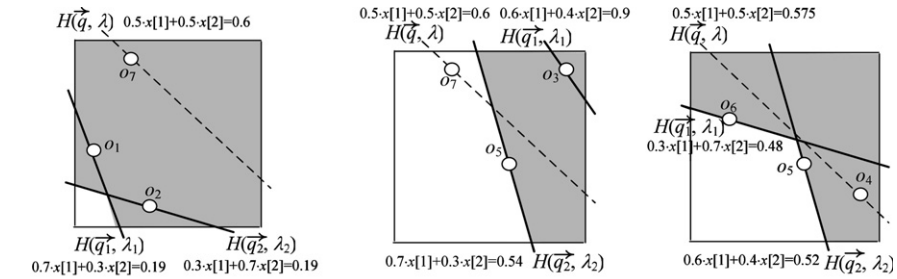


Fig. 4. Illustration of the three applications of PRUNE-BRANCH in Fig. 1.

the square represents the ranking space, the half-space $\mathscr{H}^+(\vec{q}, \lambda)$ is the portion of the space above the dashed line $\mathscr{H}(\vec{q}, \lambda)$. The line crosses o_7, and corresponds to the equation $0.5 \cdot x[1] + 0.5 \cdot x[2] = 0.6$.

At Statement 3, c equals 2, the number of items in $A.bc_1$. Then, Statements 4–6 compute two half-spaces. The first one $\mathscr{H}^+(\vec{q_1}, \lambda_1)$ is decided by Item I_1 of $A.bc_1$ according to Inequality 3, where $\vec{q_1} = \{0.7, 0.3\}$, and λ_1 is the score 0.19 of o_1 for $f_{\{0.7,0.3\}}$. The other half-space $\mathscr{H}^+(\vec{q_2}, \lambda_2)$ is derived from Item I_2 of $A.bc_2$, having $\vec{q_2} = \{0.3, 0.7\}$, and λ_2 equal to the score 0.19 of o_2 for $f_{\{0.3,0.7\}}$. In Fig. 4a, $\mathscr{H}^+(\vec{q_1}, \lambda_1)$ and $\mathscr{H}^+(\vec{q_2}, \lambda_2)$ are the part of ranking space above Lines $\mathscr{H}(\vec{q_1}, \lambda_1)$ and $\mathscr{H}(\vec{q_2}, \lambda_2)$, respectively.

The grey area of Fig. 4a shows the region represented by $\mathscr{H}^+(\vec{q_1}, \lambda_1) \cup \mathscr{H}^+(\vec{q_2}, \lambda_2)$. Since $\mathscr{H}^+(\vec{q}, \lambda)$ falls entirely in this area, Lemma 1 indicates that the subgraph $A.bsg(e_1)$ (see Fig. 1) can be eliminated from further consideration. Hence, PRUNE-BRANCH returns true at Statement 8.

The second execution of PRUNE-BRANCH is invoked with the same parameters as the previous execution, except that the first parameter becomes $A.bc_2$. Following the reasoning of Fig. 4a and b demonstrate the relevant half-spaces. As with the previous execution, $\mathscr{H}^+(\vec{q}, \lambda)$ is again decided by $\vec{q} = \{0.5, 0.5\}$ and λ equals the score 0.6 of o_7. For Line $\mathscr{H}(\vec{q_1}, \lambda_1)$, $\vec{q_1}$ is $\{0.6, 0.4\}$ (in Item I_3 of Fig. 1) and λ_1 is the score 0.9 of o_3 for $f_{\{0.6,0.4\}}$. Similarly, for Line $\mathscr{H}(\vec{q_2}, \lambda_2)$, $\vec{q_2}$ is $\{0.7, 0.3\}$ (in Item I_4) and λ_2 equals the score 0.54 of o_5 for $f_{\{0.7,0.3\}}$. As in Fig. 4b, part of $\mathscr{H}^+(\vec{q}, \lambda)$ lies outside the grey area. As a result, PRUNE-BRANCH returns false (accordingly, A contacts its neighbor B, as described in Example 4.1).

The last application of PRUNE-BRANCH happens at server B, after TOPK has obtained $rslt = \{o_3, o_4\}$ from $B.ds \cup B.dc_2$. This application is invoked with parameters $B.bc_2$, $\vec{q} = \{0.5, 0.5\}$, $k = 2$, and $rslt = \{o_3, o_4\}$. Fig. 4c shows the relevant half-spaces. In particular, for Line $\mathscr{H}(\vec{q}, \lambda)$, although $\vec{q}$ is the same as in the previous two applications, λ changes to the score 0.575 of o_4 for $f_{\vec{q}}$ (because $rslt$ here is different from those in the previous applications). For Line $\mathscr{H}(\vec{q_1}, \lambda_1)$, $\vec{q_1}$ is $\{0.3, 0.7\}$ (in Item I_5) and λ_1 is the score 0.48 of o_6 for $f_{\{0.3,0.7\}}$. For Line $\mathscr{H}(\vec{q_2}, \lambda_2)$, $\vec{q_2}$ is $\{0.6, 0.4\}$ (in Item I_6) and λ_2 is the score 0.52 of o_5 for $f_{\{0.6,0.4\}}$. PRUNE-BRANCH eliminates $B.bsg(e_2)$, because the shaded area of Fig. 4c covers $\mathscr{H}^+(\vec{q}, \lambda)$.

4. Theoretical evidence of the effectiveness of branch caching

As demonstrated in the experiments, *after a number of queries*, *BRANCA is able to solve the subsequent top-k inquiries with little network communication*, *namely*, *the results are found from the branch caches of a very small number of servers*. Next, we provide the theoretical reasoning behind this phenomenon.

We introduce the concept of *query space*, which is a d-dimensional plane passing the maximum points on the d ranking dimensions, respectively. In Fig. 5a ($d = 2$), the query space is the diagonal line as illustrated, while Fig. 5b demonstrates a 3D example where the plane is the triangle.

It is well-known [35] that the magnitude of a query vector (i.e., its distance to the origin) does not affect the top-k result, as long as the *direction* of the vector remains fixed. For instance, in Fig. 5a, the line of vector $\vec{q}_1$ crosses the diagonal at point A, and hence, a query at A produces the same top-k tuples as $\vec{q}_1$. In this way, we can convert any query vector to a point in the query space with an equivalent top-k set (e.g., $\vec{q}_2$ is mapped to point B). Note that the points in the query space essentially distribute in a $(d-1)$-dimensional space, that is, a $(d-1)$-dimensional vector is sufficient for determining the top-k set of a relation with d ranking attributes.

To simplify analysis, we first consider that all queries are top-1 search (i.e., $k = 1$), before discussing the general scenario. Assume that, in Fig. 5a, $\vec{q}_1$ and $\vec{q}_2$ are two queries in a branch cache *X.bc*, and both of them return o (i.e., o is the top-1 tuple of $\vec{q}_1$ and $\vec{q}_2$ in the subgraph *X.bsg*, for which *X.bc* is responsible). Consider a query $\vec{q}$ whose converted point C falls on segment AB. As will be proved in Lemma 2, (i) the top-1 tuple of $\vec{q}$ in *X.bsg* is also o, and very importantly, (ii) *$\vec{q}$ can be directly answered using X.bc, without having to search X.bsg*.

Indeed, the shaded area in Fig. 5a shows the pruning region (Formula 5) decided by $\vec{q}_1$ and $\vec{q}_2$ (e.g., $\mathscr{H}(\vec{q}_1, \lambda_1)$ is the line that passes o and is perpendicular to $\vec{q}_1$). The region above the dashed line is the half-space $\mathscr{H}^+(\vec{q}, \lambda)$, where λ is the score of o for $f_{\vec{q}}$. Since $\mathscr{H}^+(\vec{q}, \lambda)$ is entirely covered by the pruning region, by Lemma 1, the subgraph *X.bsg* can be eliminated.

Next, we generalize the above discussion to arbitrary dimensionalities.

Lemma 2. *Assume that X.bc is a branch cache responsible for a subgraph X.bsg. Let $\vec{q}_1$ and $\vec{q}_2$ be two queries* (*in the query space*) *that are cached in X.bc*, *and their top*-1 *tuples are both o. For any top*-1 *query $\vec{q}$ in the segment $\vec{q}_1\vec{q}_2$*, (i) *the result of $\vec{q}$ is also o*, *and* (ii) `TOPK` (*Fig. 2*) *does not need to explore X.bsg in answering $\vec{q}$.*

Proof. Since $\vec{q}$ is in the segment $\vec{q}_1\vec{q}_2$, it can be represented as $\omega \cdot \vec{q}_1 + (1-\omega) \cdot \vec{q}_2$ for some $\omega \in [0,1]$. Let $\lambda_1 = f_{\vec{q}_1}(o), \lambda_2 = f_{\vec{q}_2}(o)$, and $\lambda = f_{\vec{q}}(o)$. Hence

$$\lambda = \vec{q} \cdot \vec{o} = (\omega \cdot \vec{q}_1 + (1-\omega) \cdot \vec{q}_2) \cdot o = \omega \cdot \vec{q}_1 \cdot \vec{o} + (1-\omega) \cdot \vec{q}_2 \cdot \vec{o} = \omega \cdot \lambda_1 + (1-\omega) \cdot \lambda_2$$

Consider any vector $\vec{x}$ in $\mathscr{H}^+(\vec{q}, \lambda)$. Since $\vec{q} \cdot \vec{x} > \lambda$, we have

$$(\omega \cdot \vec{q}_1 + (1-\omega) \cdot \vec{q}_2) \cdot \vec{x} > \omega \cdot \lambda_1 + (1-\omega) \cdot \lambda_2$$

leading to

$$\omega \cdot (\vec{q}_1 \cdot \vec{x} - \lambda_1) + (1-\omega)(\vec{q}_2 \cdot \vec{x} - \lambda_2) > 0$$

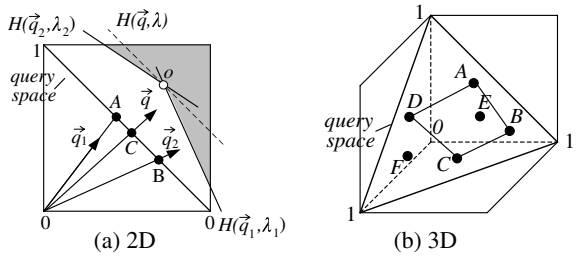


Fig. 5. The theory behind pruning a subgraph.

As both ω and $1-\omega$ are larger than or equal to 0, at least one of the following two inequalities holds: $\vec{q_1}\cdot\vec{x} > \lambda_1$ and $\vec{q_2}\cdot\vec{x} > \lambda_2$. This means that $\vec{x}$ falls in the region of $\mathscr{H}^+(\vec{q_1},\lambda_1)\cup\mathscr{H}^+(\vec{q_2},\lambda_2)$. Note that $\vec{x}$ is an arbitrary point in $\mathscr{H}^+(\vec{q},\lambda)$; therefore:

$$\mathscr{H}^+(\vec{q},\lambda)\subseteq\mathscr{H}^+(\vec{q_1},\lambda_1)\cup\mathscr{H}^+(\vec{q_2},\lambda_2) \tag{7}$$

We are ready to prove statement (i) in the target lemma. Assume, on the contrary, that there exists a tuple o' whose score (with respect to query $\vec{q}$) is larger than the score λ of o. It means that $\vec{q}\cdot o' > \lambda$. By the above reasoning, either $\vec{q_1}\cdot o' > \lambda_1$ or $\vec{q_2}\cdot o' > \lambda_2$, that is, o' is the top-1 tuple of either q_1 or q_2. This contradicts the fact that tuple o is the top-1 tuple for both $\vec{q_1}$ and $\vec{q_2}$.

To establish statement (ii), recall that, before exploring any branch subgraph, our TOPK algorithm first obtains the tuple with the highest score among all branch caches. Let the highest score for $\vec{q}$ be λ', which is at least λ (since o belongs to a branch cache). Hence, $\mathscr{H}^+(\vec{q},\lambda')\subseteq\mathscr{H}^+(\vec{q},\lambda)$, which, together with Formula 7, indicates that

$$\mathscr{H}^+(\vec{q},\lambda')\subseteq\mathscr{H}^+(\vec{q_1},\lambda_1)\cup\mathscr{H}^+(\vec{q_2},\lambda_2)$$

By Lemma 1, if *X.bsg* contains the top-1 tuple for $\vec{q}$, this tuple must exist in *X.bc*; hence, *X.bsg* is not visited by our algorithm. □

For instance, assume that queries *A* and *B* in Fig. 5a return the same top-1 tuple *o*. The above lemma indicates that *o* is also the top-1 tuple of any query $\vec{q}$ that lies on segment *AB* in the query space. Furthermore, if both *A* and *B* appear in a branch cache *X.bc*, in answering $\vec{q}$, our TOPK algorithm does not need to access the branch subgraph that *X.bc* is responsible for.

The next corollary generalizes Lemma 2 to arbitrary dimensional ranking spaces.

Corollary 2. *Consider that all queries are represented in the query space. Let Q be the set of queries* $\{\vec{q_1},\vec{q_2},\ldots\}$ *cached in X.bc that have the same top-*1 *result. Then,* TOPK *does not need to search X.bsg in answering any top-*1 *query* $\vec{q}$ *covered by the convex hull of Q.*

Proof. Consider an arbitrary $\vec{q}$ in the convex hull of *Q*, which is a $(d-1)$-dimensional polyhedron, and *d* is the number of ranking attributes (recall that the query space can be regarded as a $(d-1)$-dimensional space). Let us shoot a ray from $\vec{q_1}$ (or any vector in *Q*) passing $\vec{q}$. The ray crosses the convex hull at a point $\vec{p}_1$. Next, we will show that TOPK does not need to search *X.bsg* in answering a query at $\vec{p}_1$, which establishes the correctness of the corollary according to Lemma 2 (treating $\vec{p}_1$ as an implicit cache item in *X.bc*).

In fact, the face of the convex hull containing $\vec{p}_1$ is a $(d-2)$-dimensional polyhedron. Let us recursively apply the above reasoning to reduce dimensionality. Specifically, we arbitrarily select a vertex of the polyhedron (the vertex is an element in *Q*), and shoot another ray from it to $\vec{p}_1$, crossing the polyhedron at point $\vec{p}_2$. The face of the polyhedron enclosing $\vec{p}_2$ is a $(d-3)$-dimensional polyhedron. If the procedures are carried out repetitively, eventually we obtain a point $\vec{p}_{d-2}$ on the segment connecting two vectors in *Q*. By Lemma 2, TOPK does not need to search *X.bsg* in answering a query at $\vec{p}_{d-2}$. As a result, TOPK does not need to search *X.bsg* in answering a query at any of $\vec{p}_{d-3},\ldots,\vec{p}_1$, thus completing the proof. □

The corollary indicates that, *whenever a server X needs to explore a subgraph X.bsg for a query* $\vec{q}$, *the pruning power of its branch cache increases* (*after including the result of* $\vec{q}$), because $\vec{q}$ will expand the convex hull of the cached queries.

To illustrate this, consider Fig. 5b again, where *A*, *B*, *C*, *D*, and *E* belong to the query space (the triangle), and they are five items in *X.bc* having the same top-1 tuple *o*. The convex hull of these five points is quadrilateral *ABCD*. Corollary 1 indicates that, for any query in this quadrilateral, our TOPK algorithm can answer it without accessing *X.bsg*.

On the other hand, assume that a user issues a query *F* outside the quadrilateral, whose top-1 result is also *o*. To process *F*, TOPK needs to visit *X.bsg*, after which *F* will be cached in *X.bc*. In the future, all queries in the pentagon *ABCFD* can be directly solved using *X.bc* (without accessing *X.bsg*), that is, the set of prunable queries has increased from quadrilateral *ABCD* to the pentagon.

Notice that in this example E is a "redundant" cache item, since the pruning power remains the same even if it is discarded (E is not a vertex of the convex hull). It will be removed by an algorithm proposed in the next section.

So far we have assumed that a branch cache contains only top-1 queries. If the cached queries have higher values of k, the pruning effect for top-1 search is even stronger, because in general a top-k ($k > 1$) cache item leads to a larger pruning region (Formula 5). Finally, the above discussion can be extended to top-k pruning with arbitrary k. In particular, Lemma 2 and Corollary 1 still hold by replacing "top-1" with "top-k", and o with k objects $o_1, \ldots, o_k$.

5. Cache replacement

Given a branch cache $X.bc$ of X, we denote $vol(X.bc)$ as the volume of its pruning region (given by Formula 5). Lemma 1 indicates that, the effectiveness of $X.bc$ is determined by $vol(X.bc)$. For example, $A.bc_1$ (of Fig. 1) is expected to be more effective than $A.bc_2$ because the shaded region in Fig. 4a has a larger area than that in Fig. 4b.

We say that an item in $X.bc$ is *redundant* if its removal does not affect $vol(X.bc)$. For instance, the first item of $A.bc_2$ in Fig. 1 is redundant since, as shown in Fig. 4b, $\mathscr{H}^+(\vec{q}_1, \lambda_1)$ lies *completely inside* the shaded region. Formally, (following the notations in the previous section), let $X.bc$ contain c items $I_1 = \{\vec{q}_1, k_1, result(\vec{q}_1, k_1)\}, \ldots, I_c = \{\vec{q}_c, k_c, result(\vec{q}_c, k_c)\}$. Then, the jth ($1 \leqslant j \leqslant c$) item I_j is redundant if

$$(\mathscr{H}(\vec{q}_j, \lambda_j) \cup \mathscr{H}^+(\vec{q}_j, \lambda_j)) \cap \bigcap_{\substack{i \in [1, j-1] \\ \cup [j+1, c]}} \mathscr{H}^-(\vec{q}_i, \lambda_i) = \emptyset \tag{8}$$

where line $\mathscr{H}(\vec{q}_j, \lambda_j)$ and half-spaces $\mathscr{H}^+(\vec{q}_j, \lambda_j)$, $\mathscr{H}^-(\vec{q}_i, \lambda_i)$ are as defined in Section 3.2. Examining whether an item is redundant can be cast as a linear programming problem, in the same way as checking the integrity of Eq. (6). The next lemma shows that redundant items are indeed useless in query processing.

Lemma 3. *Let $I = \{\vec{q}, k, result(\vec{q}, k)\}$ be a redundant item in X.bc, which is responsible for the branch subgraph X.bsg. Then,* (i) *all the tuples stored in $result(\vec{q}, k)$ must appear in the other items of X.bc*; (ii) *for any query, if the* TOPK *algorithm (Fig. 2) prunes X.bsg, the subgraph can also be pruned even if I does not exist in X.bc.*

Proof. Without loss of generality, assume that I_1 is the redundant item. Thus, Formula 8 holds with $j = 1$.

Consider any tuple o in $result(\vec{q}_1, k_1)$. Since $o \in \mathscr{H}(\vec{q}_1, \lambda_1) \cup \mathscr{H}^+(\vec{q}_1, \lambda_1)$, by Formula 8, o must be in one of the $c - 1$ half-spaces $\mathscr{H}^+(\vec{q}_2, \lambda_2), \ldots, \mathscr{H}^+(\vec{q}_c, \lambda_c)$. Without loss of generality, suppose that o is in $\mathscr{H}^+(\vec{q}_2, \lambda_2)$, i.e., $f_{q_2}(o) > \lambda_2$. This means that o is in the top-k_2 set for query vector $\vec{q}_2$, i.e., o appears in $X.bc$, establishing statement (i) of the target lemma.

To prove statement (ii), assume that TOPK prunes $X.bsg$ for an arbitrary top-k query $\vec{q}$. According to Lemma 1, we have

$$\mathscr{H}^+(\vec{q}, \lambda) \subseteq \bigcup_{i \in [1,c]} \mathscr{H}^+(\vec{q}_i, \lambda_i)$$

where λ is the kth largest score of the tuples in $X.ds$ and all the branch caches of X. Let us remove I_1. According to statement (i), the value of λ remains unchanged. Furthermore, by Formula 8, $\mathscr{H}^+(\vec{q}_1, \lambda_1)$ is entirely covered by $\bigcup_{i \in [2,c]} \mathscr{H}^+(\vec{q}_i, \lambda_i)$, indicating

$$\bigcup_{i \in [1,c]} \mathscr{H}^+(\vec{q}_i, \lambda_i) = \bigcup_{i \in [2,c]} \mathscr{H}^+(\vec{q}_i, \lambda_i)$$

Hence, $\mathscr{H}^+(\vec{q}, \lambda)$ is fully covered by $\bigcup_{i \in [2,c]} \mathscr{H}^+(\vec{q}_i, \lambda_i)$, and, by Lemma 1, $X.bsg$ is pruned. □

For example, as mentioned earlier, Item I_3 of $A.bc_2$ in Fig. 1 is redundant (I_3 is the first item of $A.bc_2$, as illustrated in the figure). According to the above lemma, the object o_3 in I_3 must appear in other items of $A.bc_2$.

Indeed, o_3 is included in the second item I_4. Lemma 3 also points out that I_3 can be expunged without affecting the performance of any query.

Therefore, in case the permissible cache space of a server X has been exceeded, we should first evict the redundant items from all the branch caches. For the remaining items, our goal is to discard those that would cause the smallest degradation in the cache quality of X. For this purpose, we introduce a metric *bcQuality* for evaluating the quality:

$$bcQuality(X) = \min_{i=1}^{m}\{vol(X.bc_i)\} \tag{9}$$

where m is the number of branch caches in X (also the number of its neighbors), and $X.bc_1, \ldots, X.bc_m$ represent these caches. The intuition behind the definition is that, by maximizing $bcQuality(X)$, we ensure good quality for the worst branch-cache (i.e., the one having the smallest volume $vol(X.bc_i)$), and therefore, minimize the chance that a neighboring server needs to be contacted in answering a query.

Fig. 6 formally presents the pseudo-code of CACHE-REPLACE for handling cache-space overflows. CACHE-REPLACE takes a parameter $size_{tar}$, which is a system constant smaller than the maximum cache size of a server. The algorithm evicts enough cache items, such that the space consumption of the remaining items is no more than $size_{tar}$. In the sequel, we elaborate the details using an example.

Example 4.3. Assume that, in Fig. 1, server A incurs a space overflow, and CACHE-REPLACE is invoked with a $space_{tar}$ that allows retaining only two cache items. Statements 1–4 of Fig. 6 first eliminate the redundant items in A. Since only I_3 is redundant, Statement 5 inserts the remaining items into $S_{cache} = \{I_1, I_2, I_4\}$. The next statement initializes an empty set S_{rmv}, which will contain the items to be removed at the end of CACHE-REPLACE.

Then, Statements 7–9 perform greedy iterations until enough items have been evicted from S_{cache} to meet the requirement of $space_{tar}$. In each iteration, the algorithm picks the item (among the remaining elements in S_{cache}) whose removal results in the smallest decrease in *bcQuality*. To continue our example, among the three items I_1, I_2, I_4 in S_{cache}, discarding either of the first two does not reduce *bcQuality* (which is decided by the shaded area in Fig. 4b). Eliminating I_4, however, brings *bcQuality* down to 0, because $A.bc_2$ would be empty. Hence, assuming that item I at Statement 8 is set to I_2 (breaking the tie with a random choice), the algorithm terminates by retaining I_1 and I_4 in $A.bc_1$ and $A.bc_2$, respectively.

It is natural to wonder whether Statements 1–4 can be omitted, since a redundant item leads to zero decrease in *bcQuality*, and hence, will be selected by Statement 7 anyway. To see why the removal of the redundant items is beneficial, consider a scenario where there are 100 such items before CACHE-REPLACE starts, whereas the value of $size_{tar}$ requires eliminating only 20 items to fix the space overflow. Without Statements 1–4, after CACHE-REPLACE finishes, there would be still 80 redundant items in the branch caches. Our algorithm in Fig. 6, on the other hand, will discard all these 100 redundant items, thus postponing the next overflow without harming the pruning effectiveness.

Algorithm CACHE-REPLACE (X, $size_{tar}$)
X: the server whose cache space is exceeded
$size_{tar}$: maximum amount of remaining space after the algorithm finishes
1. for each branch cache bc in X
2. for each cache item I in bc
3. if I is redundant (it satisfies Equation 8)
4. remove I from bc
5. S_{cache} = the set of remaining items in all the branch caches in X
6. $S_{rmv} = \emptyset$ //the cache items for removal
7. while the space occupied by S_{cache} is larger than $size_{tar}$
8. I = the item in S_{cache} whose removal results in the smallest decrease of $bcQuality$, calculated with Equation 9 by excluding the items in S_{rmv}
9. $S_{rmv} = S_{rmv} \cup \{I\}$; $S_{cache} = S_{cache} - \{I\}$
10. remove the items in S_{rmv} from the corresponding cache branches

Fig. 6. The CACHE-REPLACE algorithm.

It remains to clarify the computation of $vol(X.bc)$ for a branch cache $X.bc$. The region corresponding to Formula 5 can be a complex concave hyper-polygon in high-dimensional space, whose exact volume calculation requires expensive CPU-overhead. Fortunately, CACHE-REPLACE does not demand precise volumes; it performs well also with accurate estimates of *bcQuality*. Therefore, we estimate $vol(X.bc)$ using the following Monte-Carlo approach. First, α points are randomly generated in the ranking space. Then, we count the number β of these points $\vec{x}$ that are in the region of Formula 5. After this, $vol(X.bc)$ is approximated as $\frac{\beta}{\alpha}$.

Although so far we have assumed that the underlying relation R is static, our technique can be easily extended to support a dynamic R that may be updated (by each individual server) with insertions and deletions. As with the solution of [1], we may associate each cache item with an expiry time, and discard the entry after it expires. Obviously, if updates are frequent, the quality of the query results may be compromised. Nevertheless, this is a well-known tradeoff that can be tackled using various heuristics [12]. For example, a possible solution is to require all updates to be applied only at regular intervals (e.g., at the midnight of a day). In this case, all caches are cleared at the boundary time of two intervals, and guaranteed to be valid during each interval.

6. Related work

Top-k retrieval has been extensively investigated in the database and information retrieval areas. In this section, we review the existing results, and clarify how they are relevant/different to/from our work. We classify the previous methods in three categories. Sections 6.1 and 6.2 survey the solutions on distributed relations that are vertically and horizontally partitioned, respectively. Then, Section 6.3 discusses the methods in centralized databases.

6.1. Solutions on vertically partitioned data

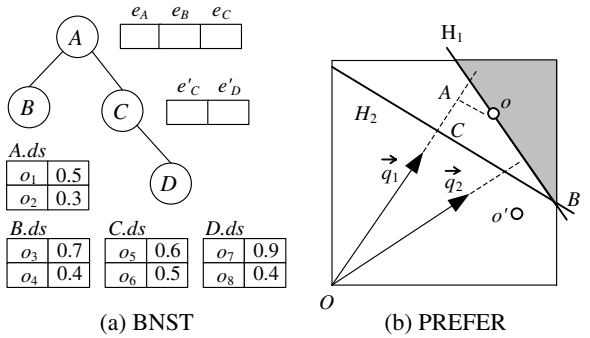
The solutions of this category address the following scenario. A relation R is vertically partitioned, such that each of its attributes is stored at one server (i.e., each tuple on a server has the form {object id, attribute value}). The objects on each server are sorted in descending order of their corresponding attribute values. A top-k query can be issued by any server, and the goal is to minimize the communication across different servers in computing the query result. Clearly, this scenario is entirely different from our problem settings, where R is horizontally partitioned across a significantly larger number of servers.

The existing algorithms [4–6,8,10,14,16,17,19,22,23,27,28,30,34,37] for vertically partitioned relations adopt the "threshold-based" technique originally proposed by Fagin [15]. They differ, however, in the types of accesses allowed at each server (e.g., sequential or random accesses), and the types of objects that can be handled in the underlying applications. Since these algorithms cannot be applied to our problem, we do not discuss them in detail.

6.2. Solutions on horizontally partitioned data

The previous work most related to ours is due to Balke et al. [1], and referred to as BNST in the sequel following the author's initials. They aim at reducing the number of *objects* transmitted among servers, as opposed to our approach that minimizes the number of *messages*. Their objective is reasonable for applications where an object (such as a document, an image, or a video) has a large size, such that forwarding an object is significantly more expensive than sending multiple regular messages. Thus, it pays off to allow servers to exchange frequently information about the "status" of query execution, as long as the communication avoids unnecessary object transmission. In our settings, however, each object is extremely small, because it is merely a simple tuple in a relation, containing an alphanumeric value for each attribute. Therefore, transmission of a tuple entails (almost) the same overhead as an empty message.

We explain the idea of BNST using the example of Fig. 7a, where the topology contains four servers A, B, C, and D. Their local data sets are demonstrated in tables $A.ds$, $B.ds$, $C.ds$ and $D.ds$, respectively. Assume that A issues a top-2 query q. The right column of each table contains the objects' scores with respect to q (e.g., the score of o_1 is 0.5). Server A initializes a *top-result list* containing empty entries e_A, e_B and e_C. During query processing, e_A keeps the next best result in $A.ds$, and e_B (e_C) maintains the next best result in the subtree rooted at server B (C).



(a) BNST (b) PREFER

Fig. 7. Illustration of BNST and PREFER.

A first computes the top-1 result in its local dataset $A.ds$ (i.e., o_1), and fills entry e_A with o_1. Then, A requests the next best objects from servers B and C, respectively. Since B is a leaf server in the topology, it directly returns the best object o_3 in its local database. After receiving o_3, A records it in e_B.

On the other hand, when server C receives the request from A, it also initializes a top result list containing empty entries e'_C and e'_D, which maintain the next best objects in C and D, respectively. To continue our example, C fills e'_C with the top-1 object o_5 in $C.ds$, and forwards a request to D. The leaf server D directly returns the top-1 object o_7 in its local database. C stores o_7 in e'_D. Since o_7 has a larger score than o_5 kept in e'_C, o_7 is returned to A, after which C re-sets e'_D to $\emptyset$.

After receiving o_7, A keeps it in e_C. Since o_7 has a larger score than objects o_1, o_3 kept in e_A, e_B, respectively, o_7 is reported as the final top-1 object. Accordingly, e_C is re-set to $\emptyset$. The process of retrieving the top-2 object is similar. Specifically, since e_C is empty, A issues another request to C. As e'_D is also empty, C contacts D again. D returns o_8 (i.e., the second best object in $D.ds$) to C, which keeps it in e'_D and compares its score with that of o_5 in e'_C. As o_5 has a higher score, it is sent to A. A keeps o_5 in e_C, and reports the object o_3 in its top-result list with the highest score as the second query result.

The above algorithm contacts all the servers at least once. To remedy the drawback, BNST adopts an index which may terminate a query without sending messages to the entire network. However, as mentioned in Section 1.2, early terminate is possible only if the same query has been processed before. Therefore, the index is not useful in our problem, where there are an infinite number of potential queries differing in their weights.

6.3. Solutions in centralized databases

PREFER [20] is an efficient system for top-k retrieval. It sorts all the tuples in descending order of their scores, with respect to a certain query vector. Then, given any other query, PREFER processes it by scanning the tuples in their sorted order, and terminating the scan as soon as no tuple in the unscanned part of the dataset can be in the query result. In the sequel, we illustrate its idea for $k = 1$. Since, as mentioned in Section 4, the magnitude of a query vector does not affect its result (only the direction matters), we consider that all queries are normalized to have unit magnitude.

In Fig. 7b, assume that the tuples have been sorted according to $\vec{q}_1$, and o is the first tuple in the sorted list (i.e., o is the top-1 tuple for $\vec{q}_1$). Based on the score definition as in Eq. 1, the score of a tuple equals the length of the segment connecting its projection on the query vector and the origin. For example, the score of o for $\vec{q}_1$ equals the length of segment OA, where A is the projection of o on $\vec{q}_1$.

Assume that a user issues a query $\vec{q}_2$. Let us draw a line H_1 that passes o and is perpendicular to vector $\vec{q}_2$. The line crosses the right boundary of the data space at point B. Let us shoot another line H_2 passing B that is perpendicular to $\vec{q}_1$. H_2 intersects $\vec{q}_1$ at point C. A crucial observation is that if the score of a point o' for $\vec{q}_1$ is lower than the length of segment OC, then o' cannot be the top-1 result of $\vec{q}_2$. This is because o' lies below H_2, whereas the top-1 result of $\vec{q}_2$ must fall in the shaded triangle. Hence, PREFER simply scans the objects in descending order of their scores for $\vec{q}_1$, and stops as soon as an object below H_2 is encountered. Among the visited objects, the one with the highest score for $\vec{q}_2$ is reported.

Obviously, the effectiveness depends on how similar $\vec{q_1}$ and $\vec{q_2}$ are. Therefore, PREFER duplicates the underlying dataset multiple times, and sorts each copy (called a "materialized view") according to a different query. Given a user query q, PREFER answers it using the view whose sorting vector is the most similar to q. Yi et al. [36] discuss the dynamic maintenance of these views. Recently, Das et al. [13] extend the heuristics of PREFER to answer a top-k query by leveraging multiple views simultaneously.

Among other work, the "STOP AFTER" operator is defined in [7] for formulating top-k queries with SQL. Chang et al. [9] propose the Onion technique, which was mentioned at the end of Section 3.2. Tsaparas et al. [35] present a ranked index based on the concept of "k-dominating set". They also develop a branch-and-bound algorithm based on R-trees [2]. Chaudhuri and Gravano [11] suggest a method that converts a top-k query to range search using data statistics. Nastev et al. [29] introduce an algorithm for processing complex joins over ranked inputs. In [21,24,26,11], the authors discuss integration of the ranking operator into relational databases, and the relevant optimization issues.

The above solutions are not directly applicable to the problem addressed in this paper, since they demand all servers to have a copy of the entire dataset. This is not realistic, as it necessitates the transmission of a massive number of tuples.

7. Experiments

This section experimentally evaluates the effectiveness of the proposed algorithms. We simulate a distributed environment using a machine with a 2.8 Ghz Pentium 4 processor. Specifically, the network organizes servers into a binary tree. The number of servers (i.e., network size) varies from 100 to 10,000. Each server stores the same number of tuples (the overall distributed relation is the union of the data at all servers). The relation has d attributes, where d ranges from 2 to 4. Each tuple can be regarded as a point in a d-dimensional space with domain $[0, 1]$ on each axis.

We test two distributions: Gaussian and Zipf. In a Gaussian dataset, each coordinate of a point follows a Gaussian distribution with mean 0.5 and variance 0.25. In a Zipf dataset, data is skewed towards the "maximal corner" of the data space (the corner has 1 as the coordinate on all axes). The skewness coefficient equals 0.8 (if the skewness equals 1, all the data degenerates into a single point; if it is 0, the distribution becomes uniform). All the dimensions are mutually independent.

Each top-k query is generated as follows. The query vector contains d positive weights, each of which randomly distributes in $[0, 1]$. k is another random value in $[1, k_{\max}]$, where $k_{\max}$ is an experiment parameter varied from 10 to 50. Each query is issued by a random server in the network. The query cost is dominated by the communication overhead. In practice, the overhead is determined by the number of network messages, which in turn is proportional to the number of servers contacted in query processing.[1] Hence, we measure the cost as the number of contacted servers.

We adopt the simplex implementation in the GNU Linear Programming Kit,[2] for solving the linear programming problems in checking Eqs. (6) and (8). As mentioned in Section 5, our cache-replacement algorithm requires the Monte-Carlo approach to calculate the volumes of pruning regions. For this purpose, we fix the parameter α to 10^5, which guarantees the precision of numerical evaluation. Every time a cache overflow occurs, 20% of the cache is emptied, i.e., the parameter $size_{tar}$ (see Section 5) equals 80% of the cache size.

We present the experimental results in two parts. In the next section, we study the characteristics of BRANCA with respect to several data and query parameters. Then, Section 7.2 compares BRANCA with the only existing solution BNST (reviewed in Section 6.2) for solving top-k queries in large-scaled distributed environments.

[1] If x servers are contacted, the total number of messages equals $4x$. For example, if server A sends a query to server B, B should first acknowledge the receipt of the message, before relaying the query to its neighbors. Similarly, when B returns the result to A, A sends another message to acknowledge receiving the result. Hence, totally four messages are required.

[2] Available at http://www.gnu.org/software/glpk/glpk.html.

7.1. Characteristics of BRANCA

In this section, each server stores 10,000 tuples. Unless specifically stated, we use a network with 1000 servers, i.e., the entire distributed relation has a cardinality of 10 million. The first set of experiments evaluates the improvement of query cost as time evolves, using a relation with $d = 3$ attributes. Each server is allowed the same amount of cache space, measured in the largest number of tuples that can be accommodated. We examine three cache sizes 500, 1000, and 2000. The parameter k_{max} equals 30.

Focusing on Gaussian data, Fig. 8a shows the cost of BRANCA (in terms of how many servers are contacted) for the first 10,000 queries in the system. Initially, when all the caches are empty, processing a query requires contacting all the servers in the network. However, the cost decreases drastically after only a small number of queries. This observation has two implications. First, it indicates that the proposed heuristics have good pruning power even with small caches. Second, the pruning effectiveness increases continuously as the caches are filled with more items, confirming the analysis of Section 4. As time progresses, the query cost eventually stabilizes when the caches of all servers are full. In particular, with cache size 2000, eventually a query needs to contact only three servers (out of 1000).

Fig. 8b shows the same results for the Zipf dataset, revealing similar observations, except that the query overhead is even lower. This is due to the fact that a significant part of a Zipf dataset lies very close to the maximal corner; hence, the number of different query results is smaller. Therefore, given the same amount of cache space, the pruning effect is stronger than in Fig. 8a.

Next, we study the amount of workload on individual servers, using the settings in the previous experiments. After 10,000 queries, we measure the number of times that a server needs to contact a neighbor in processing the next 1000 queries. Fig. 9a shows the results over the Gaussian dataset, for cache sizes 500, 1000,

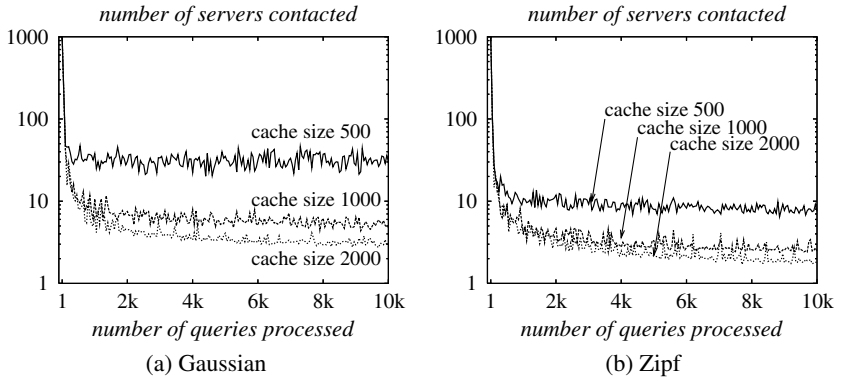


Fig. 8. Query cost vs. time (network size = 1000, $d = 3$, $k_{max} = 30$).

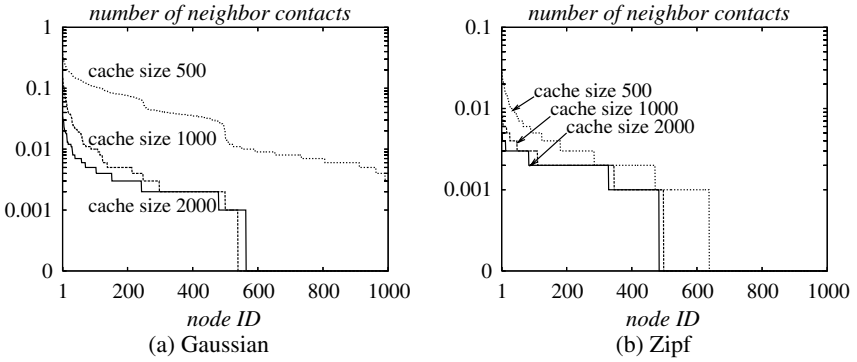


Fig. 9. Workload of individual servers (network size = 1000, $d = 3$, $k_{max} = 30$).

and 2000, respectively. The y-axis captures the number of neighbor contacts by each server, averaged over 1000 queries. The x-axis represents the server IDs (from 1 to 1000), sorted in descending order of their contact times.

It is clear that all the servers have light workload. In particular, for cache sizes at least 1000, 95% of the servers contact their neighbors less than 0.01 times in a query (i.e., totally 10 times for 1000 queries). In any case, even the "busiest" server needs no more than 0.2 contacts per query. Fig. 9b demonstrates the results for Zipf data. The cost is lower than the Gaussian case, due to the reasons analyzed earlier for Fig. 8b. For example, for cache size 2000, all servers perform less than 0.01 neighbor contacts per query, and half of the servers do not need to contact any neighbor at all.

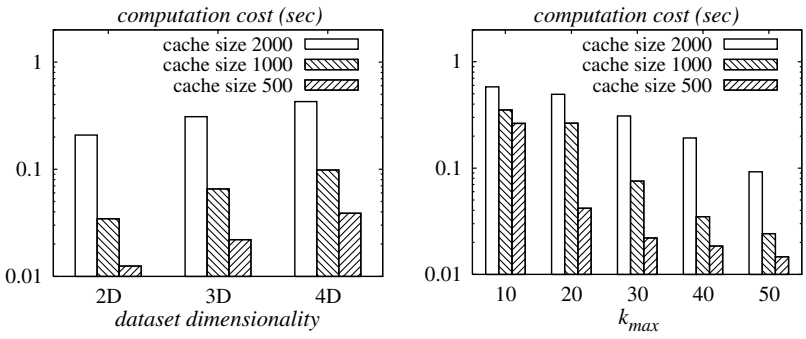
We proceed to examine the efficiency of our cache-replacement strategy. The efficiency depends on two factors: the cost of handling one cache overflow, and the overflow frequency. In particular, the first factor is affected by the dimensionality d of the dataset, and the value of $k_{\max}$ for the queries. Fig. 10 evaluates their effect using Gaussian data (the results for Zipf data are similar and omitted). Fixing $k_{\max}$ to 30, Fig. 10a plots the average cost of resolving an overflow as a function of d. The cost increases with d for two reasons. First, solving the linear programming problem (for removal of redundant cache items) is more expensive in high-dimensional space. Second, the overhead of checking whether a point lies in a half-space is linear to d. Fig. 10b exhibits the computation cost for different $k_{\max}$, setting d to 3. The cost decreases with $k_{\max}$. To understand this, note that the space occupied by the cache item of a top-k query is proportional to k. Hence, for a large $k_{\max}$, the average space consumption of an item is higher; given the same cache size, the number of items that can be retained is smaller, leading to faster cache replacement. All cache overflows are handled in less than 1 s.

To study the overflow frequency, we set d and $k_{\max}$ to 3 and 30, respectively. After the system has stabilized (i.e., after 10,000 queries, as shown in Fig. 8), we issue 5×10^5 queries, and record, for each server, the average number of queries between its two consecutive overflows as its "overflow interval". Fig. 11a and b present the average overflow interval of all servers as the network size varies from 100 to 10,000, for Gaussian and Zipf distributions, respectively. The interval is longer for a larger network size. For the same number of queries, the chance that a server is contacted becomes smaller when the number of servers increases. As a result, the cache size of a server grows more slowly, thus postponing the next cache overflow. In particular, for the cache size 2000 and network sizes at least 1000, on average a server has an overflow every more than 10^4 queries.

To summarize, after initialization, the query cost of BRANCA rapidly stabilizes at a very low level, even if the cache is relatively small compared to the dataset. This phenomenon confirms the effectiveness of our pruning mechanism. Furthermore, management of the cache content entails little overhead.

7.2. Comparison with BNST

Having demonstrated the characteristics of BRANCA, we proceed to compare its efficiency against BNST. The parameters inspected include the network size, $k_{\max}$, dimensionality d, and the number of tuples stored in



(a) Cost vs. d (k_{max} = 30) (b) Cost vs. k_{max} (d = 3)

Fig. 10. Cost of cache replacement (network size = 1000).

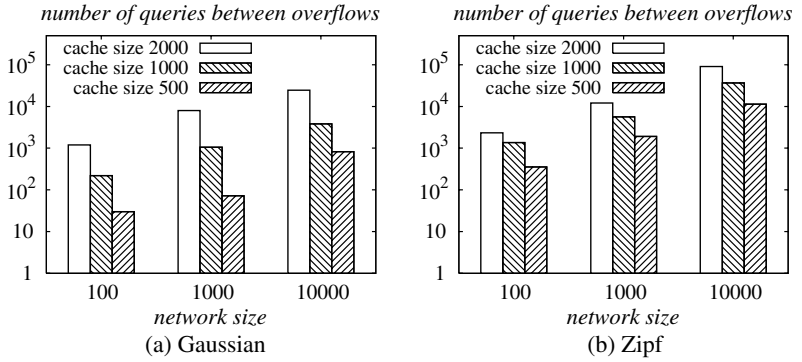


Fig. 11. Cache overflow frequency of a server (network size = 1000, $d = 3$, $k_{max} = 30$).

a server, whose default values equal 1000, 30, 3, and 10,000, respectively. We measure the query cost (i.e., the number of servers contacted) of each method after each server has issued 10 queries on average (e.g., if the network size is 100, totally 1000 queries have been processed), so that the cost represents the method's stabilized performance. Each result in the following diagrams is the average overhead of 10,000 queries.

Fig. 12a and b illustrate the results for Gaussian and Zipf data respectively, when the network size varies from 100 to 10,000, with the other parameters set to their default values. In all cases, BRANCA outperforms its competitor significantly, by a factor more than an order of magnitude. As discussed in Section 6.2, BNST needs to contact all the servers to answer a query, and hence, its cost is linear to the network size. On the other hand, with cache size at least 1000, BRANCA contacts less than 10 servers per query even in the largest network.

In Fig. 12a, the cost of BRANCA grows with the network size, while it is almost unaffected in Fig. 12b. When the network size increases, each branch cache is responsible for a subgraph containing a larger number of data points. As a result, for Gaussian distribution, the number of possible top-k results becomes higher, which weakens the pruning effect of our heuristics, and leads to higher query cost. For Zipf data, the number of result combinations does not change significantly (recall that, in a Zipf dataset, points are skewed towards the maximum corner), resulting in steady query performance as the network size grows.

The next set of experiments evaluates the influence of k_{max}. Using default values for the other parameters, Fig. 13a plots the query cost as a function of k_{max} for Gaussian data. BRANCA is again considerably faster. As expected, its cost increases with k_{max}. As mentioned in Section 7.1, a large k_{max} reduces the number of items in a cache, and hence, negatively affects the pruning effectiveness. Interestingly, if caches are sufficiently large, the impact of k_{max} is limited. For example, when the cache size equals 2000, the average number of servers contacted in a query increases by less than 1 as k_{max} varies from 10 to 50, indicating that a cache with this size is able to capture most top-k results for $k \leqslant 50$. Fig. 13b presents the results for Zipf data, also

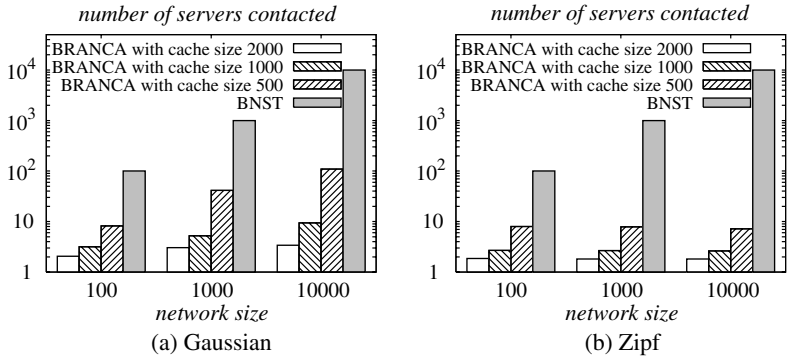


Fig. 12. Query cost vs. network size ($k_{max} = 30$, $d = 3$).

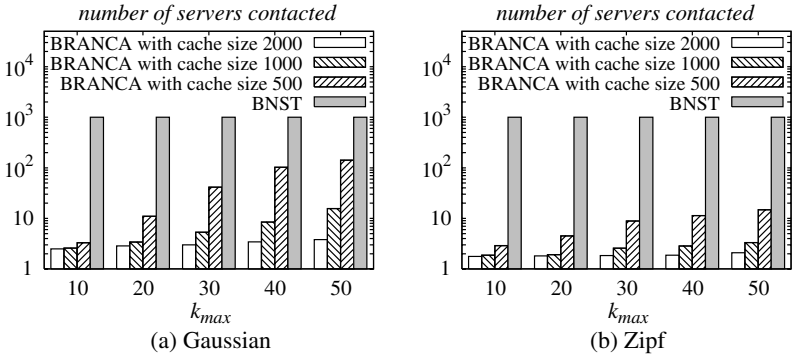


Fig. 13. Query cost vs. $k_{\max}$ (network size = 1000, $d = 3$).

demonstrating the above phenomena, except that the difference between BRANCA and BNST becomes even more obvious.

Fig. 14 shows the results when d changes from 2 to 4. The efficiency of BRANCA is lower as the dimensionality increases. This is not surprising because, in high dimensional space, there are more possible result combinations, and hence, a larger cache is necessary to achieve the same query cost. Note that BRANCA is extremely effective for 2D data. For example, even with the smallest cache size, each query requires contacting less than two servers, for both Gaussian and Zipf data. In all experiments, when the cache size reaches 2000, the query cost is limited to below 20 server contacts.

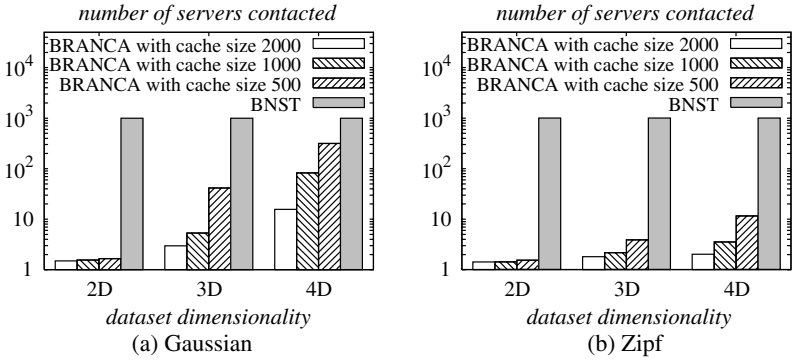


Fig. 14. Query cost vs. dataset dimensionality (network size = 1000, $k_{\max} = 30$).

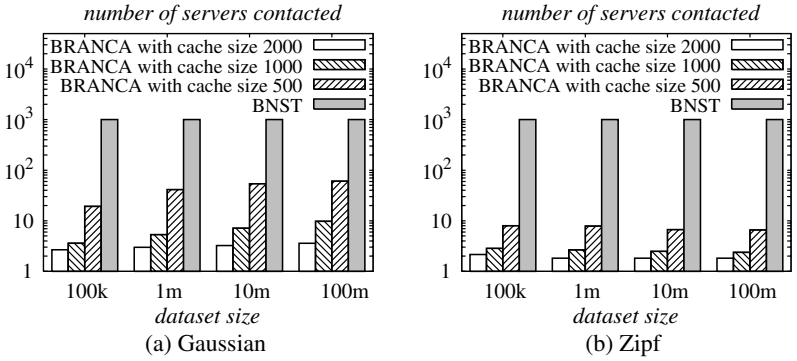


Fig. 15. Query cost vs. dataset cardinality (network size = 1000, $k_{\max} = 30$, $d = 3$).

Finally, we fix the network size, k_{max} and d to their default values, but vary the number of tuples stored at each server from 100 to 100,000, generating distributed relations with cardinalities from 100k to 100m. Fig. 15 plots the query overhead as a function of dataset cardinality. For Gaussian data, BRANCA incurs higher cost for larger datasets, while the impact of cardinality is not significant for Zipf distribution. This is consistent with the results in Fig. 12 (note that increasing the network size also leads to larger datasets). BRANCA again outperforms BNST by orders of magnitude.

In summary, our technique can process top-k queries very efficiently in large-scaled environments. Except in one experiment (involving 4D data as in Fig. 14), when the cache size equals 2000, each query requires the cooperation of less than five servers, even in a network with 10,000 servers. Furthermore, BRANCA also demonstrates good scalability with respect to the dataset distribution, cardinality, and number k of results requested.

8. Conclusions and future work

In this paper, we tackle top-k processing in the scenario where a relation is horizontally partitioned across a very large number of servers. Traditional distributed top-k algorithms are inadequate because they either assume vertical partitioning, or (in most cases) must contact all the servers at least once in answering a query. We propose an alternative approach BRANCA, which combines semantic caching with routing indexes to significantly reduce query cost. Our solutions have solid theoretical foundation, and their effectiveness is verified with extensive experiments.

For future work, we plan to adapt BRANCA for supporting other query types in both relational and non-relational domains. For example, it is interesting to study "skyline" retrieval [3], which is highly related to the top-k problem studied in this paper, because a skyline contains all the tuples that may be the top-1 result of any monotone preference function. Another promising topic is nearest neighbor search, where each server contains a set of multi-dimensional points, and the objective is to find the point (among the data of all servers) closest to a query point.

Acknowledgements

Yufei Tao was supported by Grant CUHK 1202/06 from the Research Grant Council of the HKSAR government. Shuigeng Zhou was supported by the National Natural Science Foundation of China (grant number 90612007) and the Shuguang Scholar Program of Shanghai Education Development Foundation.

References

[1] W.-T. Balke, W. Nejdl, W. Siberski, U. Thaden, Progressive distributed peer-to-peer top-k retrieval in peer-to-peer networks, in: Proc. of International Conference on Data Engineering (ICDE), 2005, pp. 174–185.
[2] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, in: Proc. of ACM Management of Data (SIGMOD), 1990, pp. 322–331.
[3] S. Borzsonyi, D. Kossmann, K. Stocker, The skyline operator, in: Proc. of International Conference on Data Engineering (ICDE), 2001, pp. 421–430.
[4] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: Mapping strategies and performance evaluation, ACM Transactions on Database Systems (TODS) 27 (2) (2002) 153–187.
[5] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over web-accessible databases, in: Proc. of International Conference on Data Engineering (ICDE), 2002, pp. 369–380.
[6] P. Cao, Z. Wang, Efficient top-k query calculation in distributed networks, in: Proc. of ACM Symposium on Principles of Distributed Computing (PODC), 2004, pp. 206–215.
[7] M.J. Carey, D. Kossmann, On saying "enough already!" in SQL, in: Proc. of ACM Management of Data (SIGMOD), 1997, pp. 219–230.
[8] K.C. Chang, S. Hwang, Minimal probing: Supporting expensive predicates for top-k queries, in: Proc. of ACM Management of Data (SIGMOD), 2002, pp. 346–357.
[9] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, J.R. Smith, The Onion Technique: Indexing for linear optimization queries, in: Proc. of ACM Management of Data (SIGMOD), 2000, pp. 391–402.
[10] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum, Probabilistic ranking of database query results, in: Proc. of Very Large Data Bases (VLDB), 2004, pp. 888–899.

[11] S. Chaudhuri, L. Gravano, Evaluating top-k selection queries, in: Proc. of Very Large Data Bases (VLDB), 1999, pp. 397–410.
[12] A. Crespo, H. Garcia-Molina, Routing indices for peer-to-peer systems, in: Proc. of International Conference on Distributed Computing Systems (ICDCS), 2002.
[13] G. Das, D. Gunopulos, N. Koudas, D. Tsirogiannis, Answering top-k queries using views, in: Proc. of Very Large Data Bases (VLDB), 2006, pp. 451–462.
[14] R. Fagin, Fuzzy queries in multimedia database systems, in: Proc. of ACM Symposium on Principles of Database Systems (PODS), 1998, pp. 1–10.
[15] R. Fagin, Combining fuzzy information from multiple systems, J. Comput. Syst. Sci. 58 (1999) 83–99.
[16] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66 (2003) 614–656.
[17] R. Fagin, E.L. Wimmers, Incorporating user preferences in multimedia queries, in: Proc. of International Conference on Database Theory (ICDT), 1997, pp. 247–261.
[18] S.I. Gass, Linear Programming: Methods and Applications, fifth ed., McGraw-Hill, Inc., 1984.
[19] U. Guntzer, W.-T. Balke, W. Kiebling, Optimizing multi-feature queries for image databases, in: Proc. of Very Large Data Bases (VLDB), 2000, pp. 419–428.
[20] V. Hristidis, N. Koudas, Y. Papakonstantinou, Prefer: a system for the efficient execution of multi-parametric ranked queries, in: Proc. of ACM Management of Data (SIGMOD), 2001, pp. 259–270.
[21] F. Ilyas, G. Aref, K. Elmagarmid, Supporting top-k join queries in relational databases, The VLDB Journal 13 (3) (2004) 207–221.
[22] I. Ilyas, W. Aref, A. Elmagarmid, Joining ranked inputs in practice, in: Proc. of Very Large Data Bases (VLDB), 2002, pp. 950–961.
[23] I. Ilyas, W. Aref, A. Elmagarmid, Supporting top-k join queries in relational databases, in: Proc. of Very Large Data Bases (VLDB), 2003, pp. 754–765.
[24] I.F. Ilyas, R. Shah, W.G. Aref, J.S. Vitter, A.K. Elmagarmid, Rank-aware query optimization, in: Proc. of ACM Management of Data (SIGMOD), 2004, pp. 203–214.
[25] H.V. Jagadish, B.C. Ooi, Q.H. Vu, Baton: a balanced tree structure for peer-to-peer networks, in: Proc. of Very Large Data Bases (VLDB), 2005, pp. 661–672.
[26] C. Li, K.C. Chang, I.F. Ilyas, S. Song, Ranksql: Query algebra and optimization for relational top-k queries, in: Proc. of ACM Management of Data (SIGMOD), 2005, pp. 131–142.
[27] A. Marian, N. Bruno, L. Gravano, Evaluating top-k queries over web-accessible databases, ACM Transactions on Database Systems (TODS) 29 (2) (2004) 319–362.
[28] S. Michel, P. Triantafillou, G. Weikum, Klee: A framework for distributed top-k query algorithms, in: Proc. of Very Large Data Bases (VLDB), 2005, pp. 637–648.
[29] A. Natsev, Y.-C. Chang, J.R. Smith, C.-S. Li, J.S. Vitter, Supporting incremental join queries on ranked inputs, in: Proc. of Very Large Data Bases (VLDB), 2001, pp. 281–290.
[30] S. Nepal, M.V. Ramakrishna, Query processing issues in image (multimedia) databases, in: Proc. of International Conference on Data Engineering (ICDE), 1999, pp. 22–29.
[31] S. Ratnasamy, P. Francis, M. Handley, R.M. Karp, S. Shenker, A scalable content-addressable network, in: Proc. of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), 2001, pp. 161–172.
[32] Q. Ren, M.H. Dunham, V. Kumar, Semantic caching and query processing, IEEE Transactions on Knowledge and Data Engineering 15 (1) (2003) 192–210.
[33] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proc. of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), 2001, pp. 149–160.
[34] M. Theobald, G. Weikum, R. Schenkel, Top-k query evaluation with probabilistic guarantees, in: Proc. of Very Large Data Bases (VLDB), 2004, pp. 648–659.
[35] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, D. Srivastava, Ranked join indices, in: Proc. of International Conference on Data Engineering (ICDE), 2003, pp. 277–288.
[36] K. Yi, H. Yu, J. Yang, G. Xia, Y. Chen, Efficient maintenance of materialized top-k views, in: Proc. of International Conference on Data Engineering (ICDE), 2003, pp. 189–200.
[37] H. Yu, H.-G. Li, P. Wu, D. Agrawal, A.E. Abbadi, Efficient processing of distributed top- queries, in: Proc. of Database and Expert Systems Applications (DEXA), 2005, pp. 65–74.

Keping Zhao obtained his master degree in Computer Science from Fudan University. His research interests focus on complex query optimization in Peer-to-Peer networks. After graduation, he worked as a Research Assistant in the City University of Hong Kong. Currently, he is a Software Design Engineer of Microsoft China.

Yufei Tao holds a Ph.D. degree in Computer Science from the Hong Kong University of Science and Technology, and did his post doc as a visiting scientist in the Computer Science Department of the Carnegie Mellon University, during September 2002–August, 2003. In the next 3 years, he was an Assistant Professor at the City University of Hong Kong. Since September 2006, he has been an Assistant Professor at the Department of Computer Science and Engineering, the Chinese University of Hong Kong. He is the winner of the Hong Kong Young Scientist Award 2002, conferred by the Hong Kong Institution of Science. His research interests include database query optimization, and data privacy protection.

Shuigeng Zhou is a Professor in Department of Computer Science and Engineering, Fudan University, Shanghai, China. He received his Bachelor degree of Electronic Engineering from Huazhong University of Science and Technology (HUST) in 1988, his Master degree of Electronic Engineering from University of Electronic Science and Technology of China (UESTC) in 1991, and his Ph.D. of Computer Science from Fudan University in 2000. He served in Shanghai Academy of Spaceflight Technology from 1991 to 1997, as an Engineer and a Senior Engineer (since August 1995), respectively. He was a post-doctoral researcher in State Key Lab of Software Engineering, Wuhan University from 2000 to 2002. His research interests include data management in P2P and sensor networks, data mining and information retrieval. He has published more than 100 papers in domestic and international journals and conferences. Currently he is member of IEEE and IEICE.