Processing Top-k Queries in Distributed Hash Tables[1]

Reza Akbarinia, Esther Pacitti, Patrick Valduriez

INRIA and LINA, University of Nantes, France
{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}

Abstract. Distributed Hash Tables (DHTs) provide a scalable solution for data sharing in large scale distributed systems, *e.g.* P2P systems. However, they only provide good support for exact-match queries, and it is hard to support complex queries such as top-k queries. In this paper, we propose a family of algorithms which deal with efficient processing of top-k queries in DHTs. We evaluated the performance of our solution through implementation over a 64-node cluster and simulation. Our performance evaluation shows very good performance, in terms of communication cost and response time.

1 Introduction

Distributed Hash Tables (DHTs), *e.g.* CAN [20] and Chord [23], provide an efficient solution for data location and lookup in large-scale P2P systems. While there are significant implementation differences between DHTs, they all map a given key onto a peer *p* using a hash function and can lookup *p* efficiently, usually in *O(log n)* routing hops where *n* is the number of peers [13]. DHTs typically provide two basic operations [13]: *put(key, data)* stores a pair *(key, data)* in the DHT using some hash function; *get(key)* retrieves the data associated with *key* in the DHT. These operations enable supporting exact-match queries only. Recently, much work has been devoted to supporting more complex queries in DHTs such as range queries [11] and join queries [15]. However, efficient evaluation of more complex queries in DHTs is still an open problem [5].

An important kind of complex queries is top-k queries. Given a dataset *D* and a scoring function *f*, a top-*k* query retrieves the *k* tuples in *D* with the highest scores according to *f*. Top-k queries have attracted much interest in many different areas such as network and system monitoring [2][6], information retrieval [3][16], sensor networks [22][24], multimedia databases [7][12][19], spatial data analysis [14], data streams [18], etc. The main reason for such interest is that they avoid overwhelming the user with large numbers of uninteresting answers which are resource-consuming. Most of the efficient approaches for top-k query processing in centralized and distributed systems, *e.g.*[4][6][8][17], are based on the Threshold Algorithm (TA) [10][12][19]. TA is applicable for queries where the scoring function is monotonic, *i.e.*, any increase in the value of the input does not decrease the value of the output.

In a large-scale P2P system, top-k queries can be very useful [3]. For example assume a community of car dealers who want to take advantage of a DHT to share some data about the used cars which they are willing to sell. Assume they agree on a common Car description in relational format. The Cars relation includes attributes such as car-id, price, mileage, mark, model, picture, etc. Suppose a user wants to submit the following query to obtain the 10 top answers ranked by a scoring function over price and mileage:

[1] Work partially funded by ARA "Massive Data" of the French ministry of research and the European Strep Grid4All project.

SELECT car-id, price, mileage FROM Cars
WHERE price < 3000 AND mileage < 60000
ORDER BY scoring-function(price, mileage) **STOP AFTER** 10

The user specifies the scoring function according to the criteria of interest. For instance, in the query above, the scoring function could be (- (20*price + mileage)).

The problem of top-k queries has been addressed in unstructured P2P networks, *e.g.* in [1], and also super-peer networks, *e.g.* in [3]. However, the specific nature of DHTs, *i.e.* data storage and retrieval based on hash functions, makes it quite challenging to support top-k queries [5]. A simple solution for supporting top-k queries in DHTs is to retrieve all tuples of the relations involved in the query, compute the score of each retrieved tuple, and finally return the *k* tuples whose scores are the highest. However, this solution cannot scale up to a large number of stored tuples. Another solution is to store all tuples of a relation in the DHT by using the same key (*e.g.* relation's name), thus all tuples are stored at the same peer. Then, top-k queries can be processed at the central peer using well-known centralized algorithms. However, the central peer becomes a bottleneck and a single point of failure.

What we need is an efficient solution which can scale up to large numbers of peers and avoids any centralized data storage. In this paper, we propose such a solution for top-k query processing in DHTs. Our main contributions are the followings:

- We propose a data storage mechanism that not only provides good support for exact-match queries, but also enables efficient execution of top-k queries using our algorithms. It stores relational data in the DHT in a fully decentralized way, and avoids skewed distribution of data among peers.
- We propose a family of three algorithms which deal with efficient processing of top-k queries in DHTs. The first algorithm efficiently supports top-k queries with monotonic scoring functions. The second one supports top-k queries with a much larger class of scoring functions. We propose two optimizing strategies that reduce significantly the communication cost of the latter algorithm. We analytically prove that the algorithm finds correctly the *k* highest scored tuples. We propose a third algorithm for the cases where only a small set of relations' attributes are used in scoring functions. At the expense of incurring a small amount of redundancy on the DHT, the third algorithm yields much performance gains in terms of response time and communication cost.
- We evaluated the performance of our algorithms through implementation over a 64-node cluster and simulation using SimJava up to 10,000 peers. The results show the effectiveness of our solution for processing top-k queries in DHTs.

The rest of this paper is organized as follows. In Section 2, we present our mechanism for storing the shared data in a DHT. In Section 3, we present our algorithms for processing top-k queries in DHTs. Section 4 describes a performance evaluation of our algorithms through implementation over a 64-node cluster and simulation using SimJava. Section 5 concludes.

2 Data Storage Mechanism

In this section, we propose a mechanism for storing relational data in the DHT. This mechanism not only provides good support for exact-match queries, it also enables efficient execution of our top-k query processing algorithms. In our data storage mechanism, peers store their relational data in the DHT with two complementary methods: tuple storage and attribute-value storage. In this paper, we assume that the data which are stored in the DHT are highly available by using yet proposed approaches, *e.g.* using multiple hash functions as in [20].

2.1 Tuple Storage

With the tuple storage method, each tuple of a relation is entirely stored in the DHT using its tuple identifier (*e.g.* its primary key) as the storage key. This enables looking up a tuple by its identifier. Let R be a relation name and A be the set of its attributes. Let T be the set of tuples of R and $id(t)$ be a function that denotes the identifier of a tuple $t \in T$. Let h be a hash function that hashes its inputs into a DHT key, *i.e.* a number which can be mapped by the DHT onto a peer. For storing relation R, each tuple $t \in T$ is entirely stored in the DHT where the storage key is $h(R, id(t))$, *i.e.* the hash of the relation name and the tuple identifier. Hereafter, the key by which we store a tuple in the DHT is called *tuple storage key*.

Tuple storage allows us to answer exact-match queries on the tuple identifier. For example, consider relation Car(car-id, price, mileage, …) in which *car-id* is the primary key. If we store the tuples of this relation in the DHT using the tuple storage method, we are able to answer to exact match queries on *car-id* attribute, *e.g.* "Is there any car whose *car-id* is equal to 20?". But, it does not help answering exact-match queries on other attributes, *e.g.* "Is there any car whose price is 2000?". Attribute-value storage helps answering such queries.

A straightforward extension to tuple storage is to partition (fragment) the relation horizontally and store all tuples of each partition with the same key, thereby at the same peer. The key for storing the tuples of each partition can be constructed as for attribute-value storage which we describe in the next section. For very large numbers of tuples, this extension can be much more efficient than storing each tuple in the DHT with a different key.

2.2 Attribute-Value Storage

Attribute-value storage stores individually the attributes that may appear in a query's equality predicate or in a query's scoring function in the DHT. Thus, like database secondary indices, it allows checking for the existence of tuples using attribute values. Our attribute-value storage method has two important properties. 1) after retrieving an attribute value from the DHT, peers can retrieve easily the corresponding tuple of the attribute value; 2) attribute values that are relatively "close" are stored at the same peer. To satisfy the first property, the key used for storing the entire tuple, *i.e.* tuple storage key, is stored along with the attribute value. The second property is satisfied by using the concept of domain partitioning as follows. Consider an attribute a and let D_a be its domain of values. Assume there is a total order $<$ on D_a, *e.g.* D_a is numeric, string, date, etc. D_a is partitioned into n nonempty sub-domains $d_1, d_2, \ldots, d_n$ such that their union is equal to D_a, the intersection of any two different sub-domains is empty, and for each $v_1 \in d_i$ and $v_2 \in d_j$, if $i<j$ then we have $v_1 < v_2$. For example, the attribute "mileage", whose domain is integer values in [0..300K], can be partitioned into 30 sub-domains [0..10K), [10K..20K), …, [290K..300K]. Given a value v, the sub-domain to which v belongs is denoted by $sd(a, v)$. The number of sub-domains of an attribute and the lower bound of each sub-domain are known to all peers of the DHT. Therefore, given an attribute a and a value v, any peer can locally compute $sd(a, v)$.

The key which is used for storing an attribute value in the DHT is constructed as follows. Let R be a relation, a be an attribute of R, and v be the value of a in a tuple t, then the key for storing v in the DHT is $h(R, a, sd(a, v))$, *i.e.* the hash of the relation name, attribute name and the sub-domain to which v belongs. Therefore, the attribute values that belong to the same sub-domain are stored with the same key. Thus, they are maintained at the same peer.

The values of the attributes, for which we do an attribute-value storage, are stored two times in the DHT, *i.e.* once upon tuple storage and once upon attribute-value storage. However, this controlled redundancy allows us to answer exact-match queries on these attributes.

2.3 Uniform Distribution of Attribute-Values

The method which we use for partitioning attribute domains should avoid skewed distribution of attribute values within sub-domains, which may yield load unbalance among peers. For instance, simply dividing the domain into n equal-width sub-domains, as we did for attribute "mileage" above, may yield attribute storage skew. Using histogram-based information that describes the distribution of the values of an attribute, we can do a better partitioning that uniformly distributes the values within the sub-domains. Formally, let $p_a(v)$ be the probability density function that describes the probability that attribute a takes a value equal to v. Let $lb(d)$ be a function that denotes the lower bound of a sub-domain d, to obtain a uniform partitioning we choose the sub-domains $d_1, d_2, \ldots, d_n$ such that:

$$\int_{lb(d_i)}^{lb(d_{i+1})} p_a(v)dv = \frac{1}{n} \quad \text{, for } 1 \le i \le n-1 \qquad (1)$$

By these $n-1$ equations, the lower bounds are chosen in such a way that the sub-domains have equal cumulative numbers of values. We know that the lower bound of d_1 is equal to the lower bound of D_a, so $lb(d_1)$ is determined. Thus, we have $n-1$ equations with $n-1$ variables, *i.e.* $lb(d_2),\ldots, lb(d_n)$. By solving the equations, we can determine the value of $lb(d_2),\ldots, lb(d_n)$.

3 Top-k Query Processing Algorithms

In this section, we first propose DHTop1, an algorithm for efficient executions of top-k queries whose scoring function is monotonic. Then, based on DHTop1, we propose the DHTop2 algorithm which supports a much larger group of scoring functions. Finally, we propose the DHTop3 algorithm which exploits a small amount of redundancy in the DHT to yield high performance gains in the execution of top-k queries.

3.1 DHTop1 Algorithm

Let Q be a given top-k query, f be its scoring function, and p_{int} be the peer at which Q is issued. Let *scoring attributes* be the attributes that are used in f. We assume that the values of the scoring attributes are stored in the DHT by attribute-value storage. DHTop1 starts at p_{int} and proceeds in two phases as follows:

1. For each scoring attribute α do
   Create a list L_α and add all sub-domains of α to it;
   Remove from L_α the sub-domains which do not satisfy Q's condition;
   Sort L_α in descending order of its sub-domains;
2. end-condition := false;
   For each scoring attribute α do in parallel
   i := 1;
   n := number of sub-domains in L_α;
   While (end-condition = false) and ($i \le n$) do
   Send Q to the peer p that maintains the α values whose sub-domain is $L_\alpha[i]$. p returns to p_{int} its values of α which satisfy Q's condition, one by one in descending order, along with their corresponding tuple storage key;
   v := the first α value returned by p;
   While ($v \neq$ null) and (end-condition = false) do
   - Retrieve the corresponding tuple of v and compute its score. If it

is one of the k highest scores, then record the tuple in a list Y;

- If there are k tuples in Y whose scores are higher than the threshold then set end-condition to true and return to the user these k tuples;
- If end-condition is false then set v to the next α value returned by p;
- If v is null (*i.e.* all values returned by p have been received) then set $i := i + 1$;

The threshold we use is inspired from the TA algorithm [10] and is computed as follows. Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be the scoring attributes. Let $v_1, v_2, \ldots, v_m$ be the last values received respectively for attributes $\alpha_1, \alpha_2, \ldots, \alpha_m$. The threshold is defined as $\delta = f(v_1, v_2, \ldots, v_m)$. After receiving each attribute value, the threshold value is recomputed.

Let N be the number of stored tuples and m be the number of scoring attributes. In a way similar to [9], we can prove that, in the worst case, the number of tuples which should be retrieved by DHTop1 is $O(N^{(m-1)/m} * k^{1/m})$, which is sub-linear in the number of tuples. Due to space limitations we omit the proof. In average case, the execution time of DHTop1 is much better than an algorithm that retrieves all tuples (see Section 4).

We have proved that the DHTop1 works correctly for top-k queries with monotonic scoring functions. The proof is similar to the proof which we give for the correctness of the DHTop2 algorithm in the following section.

3.2 DHTop2

DHTop1 efficiently executes top-k queries whose scoring function is monotonic, *i.e.* any increase in the value of the input does not decrease the value of the output. For example the function $f(x_1, x_2) = x_1 + x_2$ is monotonic, but $f(x_1, x_2) = x_1 - x_2$ is not monotonic because an increase in the value of x_2 decreases the output of the function. Many of the popular aggregation functions, *e.g.* Min, Max, Average, are monotonic. However, there are many useful functions that are not monotonic including most of linear functions. In this section, we propose the algorithm DHTop2 which works with a super-set of monotonic functions. We call these functions *IODEV scoring functions*.

3.2.1 IODEV Scoring Functions

To define IOD-EV scoring functions, we need the two following definitions.

Definition 1 (Increasing wrt variable x): *A scoring function is increasing wrt variable x if any increase in the value of x does not decrease the output of the scoring function.*

Definition 2 (Decreasing wrt variable x): *A scoring function is decreasing wrt variable x if any increase in the value of x does not increase the output of the scoring function.*

For example the function $f(x_1, x_2) = x_1 - x_2$ is increasing wrt x_1 and decreasing wrt x_2. Now we can define IOD-EV scoring functions.

Definition 3 (Increasing Or Decreasing wrt Each Variable (IOD-EV)): *A scoring function f is IOD-EV if for each variable x, f is increasing wrt x or decreasing wrt x.*

For example, the functions $f(x_1, x_2) = x_1 - x_2$ and $f(x_1, x_2) = (x_1)^3 - (x_2)^3$ are IOD-EV. The set of monotonic scoring functions is a subset of IOD-EV functions because a monotonic scoring function is increasing wrt every variable, thus it is *IOD-EV*. The linear functions are also IOD-EV. This can be demonstrated as follows. A linear function can be written as $f(x_1, x_2, \ldots, x_m) = a_0 + a_1x_1 + a_2x_2 + \ldots + a_mx_m$ where $a_0, a_1, \ldots, a_m$ are constant values. For each variable x_i if its coefficient a_i is positive then f is increasing wrt x_i, otherwise it is decreasing wrt x_i. Thus, all linear functions are IOD-EV.

3.2.2 Algorithm

Let Q be a given top-k query, and f be its scoring function. Assume f is IOD-EV, then DHTop2 algorithm is obtained from DHTop1 by performing the following modifications:

- In Phase 1, the sorting of L_α is done as follows. If the scoring function f is increasing wrt α then L_α is sorted in descending order of the lower bound of its sub-domains. Otherwise L_α is sorted in ascending order. Let $lb(d)$ denote the lower bound of a sub-domain d, and $L_\alpha[i]$ denote the ith sub-domain of L_α. At the end of this step, L_α is as follows. If f is increasing wrt α then $lb(L_\alpha[i]) \geq lb(L_\alpha[i+1])$ for $i \geq 1$. And if f is decreasing wrt α then $lb(L_\alpha[i]) \leq lb(L_\alpha[i+1])$ for $i \geq 1$.
- In Phase 2, the peer p, which maintains the α values whose sub-domain is $L_\alpha[i]$, returns the attribute values to p_{int} in the following order. If f is increasing wrt α then p returns the α values in descending order. Otherwise, it returns them in ascending order.

Example. Consider a relation $R(a, b, c)$ such that the domain of the attributes a and b is the real values in [0..10]. Assume that the domain of attribute a is partitioned into 4 sub-domains d_1=[0..3), d_2=[3..5), d_3=[5..8) and d_4=[8..10]. Assume the domain of attribute b is partitioned into 5 sub-domains d'_1=[0..3), d'_2=[3..5), d'_3=[5..6), d'_4=[6..8) and d'_5=[8..10). Consider the following query Q which is issued at p_{int}:

```
SELECT  *  FROM  R
WHERE  a < 6  AND b < 5
ORDER BY (a-b)    STOP AFTER  5
```

In the above query, the scoring attributes are a and b. The scoring function, *i.e.* $f=a-b$, is increasing wrt a and decreasing wrt b. Thus, in Phase 1, L_a is sorted in descending order and L_b in ascending order. At the end of Phase 1, we have $L_a=<d_3, d_2, d_1>$ and $L_b=<d'_1, d'_2>$. Notice that some of the sub-domains of a (and b) are removed from L_a (and L_b) due to Q's condition, *e.g.* d_4 is removed from L_a because of a<6 in Q's condition.

3.2.3 Proof of Correctness

Let us now prove the correctness of the DHTop2 algorithm for top-k queries with IOD-EV scoring functions. For this, we prove the following lemma.

Lemma 1: *Let f be an IOD-EV scoring function, v_i and v'_i be two values of a scoring attribute a_i such that v_i is retrieved by DHTop2, and v'_i is retrieved after v_i or it is not retrieved by DHTop2, then we have $f(x_1, x_2,\ldots, x_{i-1}, v'_i, x_{i+1},\ldots, x_m) \leq f(x_1, x_2,\ldots, x_{i-1}, v_i, x_{i+1},\ldots, x_m)$ for any value x_j ,$1 \leq j \leq m$ and $j \neq i$. In other words, if we only change the value of a_i by replacing v_i with v'_i, the output of the scoring function decreases or does not change.*

Proof: With respect to the possible values v_i *and* v'_i, there are two cases to consider. In the first case, v_i *and* v'_i belong to two different sub-domains, *e.g.* d_1 and d_2 respectively. Thus, d_1 is before d_2 in L_{ai}. If f is increasing wrt a_i, then considering the first phase of the algorithm, we have $lb(d_1) \geq lb(d_2)$. Thus, we have $v_i \geq v'_i$ and since f is increasing wrt a_i, we have $f(x_1, x_2,\ldots, x_{i-1}, v'_i, x_{i+1},\ldots, x_m) \leq f(x_1, x_2,\ldots, x_{i-1}, v_i, x_{i+1},\ldots, x_m)$, *i.e.* increasing the value of a_i does not decrease the output of the function. Now, if f is decreasing wrt a_i, then considering the first phase of the algorithm, we have $lb(d_1) \leq lb(d_2)$. Thus $v_i \leq v'_i$ and since f is decreasing wrt a_i, we have $f(x_1, x_2,\ldots, x_{i-1}, v'_i, x_{i+1},\ldots, x_m) \leq f(x_1, x_2,\ldots, x_{i-1}, v_i, x_{i+1},\ldots, x_m)$. The second case is when v_i *and* v'_i belong to the same sub-domain. If f is increasing wrt a_i, then considering the second phase of the algorithm, we have $v_i \geq v'_i$ and thus $f(x_1, x_2,\ldots, x_{i-1}, v'_i, x_{i+1},\ldots, x_m) \leq f(x_1, x_2,\ldots, x_{i-1}, v_i, x_{i+1},\ldots, x_m)$. If f is decreasing wrt a_i then we have $v_i \leq v'_i$ and thus $f(x_1, x_2,\ldots, x_{i-1}, v'_i, x_{i+1},\ldots, x_m) \leq f(x_1, x_2,\ldots, x_{i-1}, v_i, x_{i+1},\ldots, x_m)$.□

The following theorem provides the correctness of our algorithm.

Theorem 1: *If f is an IOD-EV scoring function, then DHTop2 finds the k top tuples correctly.*

Proof: the proof is by contradiction. Let Y be the set of k top tuples obtained by DHTop2, and t' be the tuple in Y whose score is the lowest. We assume there is a tuple $t'' \notin Y$ such that its score is greater than t', and we show that this assumption yields to a contradiction. Let $a_1, a_2, \ldots, a_m$ be the scoring attributes. Let $v_1, v_2, \ldots, v_m$ be the last values, *i.e.* before ending the algorithm, retrieved respectively for attributes $a_1, a_2, \ldots, a_m$. Let $v'_1, v'_2, \ldots, v'_m$ be the values of the attributes $a_1, a_2, \ldots, a_m$ in t', respectively. Let $v''_1, v''_2, \ldots, v''_m$ be the values of attributes $a_1, a_2, \ldots, a_m$ in t'', respectively. Since t'' is not in Y, it was not retrieved during the execution of our algorithm. Thus, none of its values, *i.e.* $v''_1, v''_2, \ldots, v''_m$, was retrieved by p_{int}, because if the value of any attribute of a tuple was retrieved, the entire tuple would have been retrieved by the algorithm. By applying Lemma 1 on attribute a_1 we have $f(v_1, v_2, \ldots, v_m) \geq f(v''_1, v_2, \ldots, v_m)$. By applying Lemma 1 on attribute a_2, we have $f(v''_1, v_2, v_3, \ldots, v_m) \geq f(v''_1, v''_2, v_3, \ldots, v_m)$. By continuing the application of Lemma 1 on attributes $a_3, \ldots, a_m$, we have $f(v_1, v_2, \ldots, v_m) \geq f(v''_1, v_2, \ldots, v_m) \geq f(v''_1, v''_2, \ldots, v_m) \geq \ldots \geq f(v''_1, v''_2, \ldots, v''_{m-1}, v_m) \geq f(v''_1, v''_2, \ldots, v''_{m-1}, v''_m)$. Therefore, we have $f(v_1, v_2, \ldots, v_m) \geq f(v''_1, v''_2, \ldots, v''_m)$. According to the end condition of the algorithm, we have $f(v'_1, v'_2, \ldots, v'_m) \geq f(v_1, v_2, \ldots, v_m)$, and by comparing this inequality with the former one, we have $f(v'_1, v'_2, \ldots, v'_m) \geq f(v''_1, v''_2, \ldots, v''_m)$. In other words, the score of tuple t' is greater than that of t'', which yields to a contradiction.□

3.2.4 Optimizations

In order to further reduce the communication cost of the DHTop2 algorithm, we propose two optimizing strategies: batch retrieval of attribute values and retrieving each tuple at most once.

Batch retrieval of attribute values (BRAV). In Phase 2 of the basic version of DHTop2, the values of the scoring attributes are returned to p_{int} one by one, *i.e.* each value in a message. Since each message has its own overhead, *e.g.* latency, returning only one value per message is very costly. To reduce such overhead, we modify Phase 2 of the algorithm such that the peer, which maintains the values of a sub-domain, sends the attribute values to p_{int} in a batch fashion, *e.g.* k values per message.

Retrieving each tuple at most once (RTO). In Phase 2 of the basic version of DHTop2, after retrieving each value of a scoring attribute, the corresponding tuple of that value is retrieved. Since there may be several scoring attributes, a tuple may be retrieved several times. However, after the first retrieval of the tuple and comparing its score with the k highest scores, there is no need to retrieve it again because either the tuple is in the set Y or its score cannot be one of the k highest scores. Thus, to optimize our algorithm, we change Phase 2 such that p_{int} maintains in a list the identifiers of all tuples which have yet been retrieved. Before retrieving a tuple, p_{int} checks the list, and if the identifier of the tuple is in the list, it does not retrieve the tuple.

3.3 DHTop3

By analyzing many scoring functions in useful queries, we observed that only a small set of a relation's attributes are likely to be used for scoring. For instance, the attributes typically used for scoring used cars are price and mileage. In addition to their small number, these attributes typically have small size, *e.g.* numerical. Based on this observation, we modify our storage mechanism such that the value of the attributes typically used for scoring are stored upon each attribute-value storage, *e.g.* upon storing the price value, we store the value of both price and mileage in the DHT.

Formally, let R be a relation, and A_{sf} be the set of attributes of R which are typically used for scoring. Let t be a tuple of R, and V be the set of values of attributes of A_{sf} in tuple t. Let $v \in V$ be the value of $a \in A_{sf}$ in tuple t. Upon storing v by the modified attribute-value storage, we store all values involved in V in the DHT. Like with the basic attribute-value storage, we also store the storage key of t along with V.

With the modified attribute-value storage, the values of the attributes involved in A_{sf} are stored $/A_{sf}/$ times in the DHT. At the expense of this redundancy, which is usually low because $/A_{sf}/$ is small, we can now compute a tuple score at peers that maintain the attribute values without having to retrieve the tuple.

Relying on the modified attribute-value storage, we develop a top-k query processing algorithm, called DHTop3, based on DHTop2 by performing the following modifications in the second phase. The peer p, which maintains the attribute values of sub-domain $L_a[i]$, computes the scores of the corresponding tuple of each attribute value, and returns the scores along with the attribute values to p_{int}. Thus, p_{int} no longer needs to retrieve the corresponding tuple of the received values; it only keeps the storage keys of the k highest scored tuples in the set Y and continues until the end condition holds. Finally, p_{int} retrieves from the DHT the k tuples whose storage keys are maintained in Y.

4 Performance Evaluation

We evaluated the performance of the algorithms, which we proposed in the pervious section, through implementation and simulation. The implementation over a 64-node cluster was useful to validate our algorithms and calibrate the simulator. The simulator allows us to study scale up to high numbers of peers (up to 10,000 peers).

In this section, we first describe our experimental setup. Then, we investigate the scalability of our algorithms by increasing the number of peers and also by increasing the number of tuples which, we store for each relation, in the DHT. Finally, we evaluate the performance of our algorithm by varying other parameters such as the number of requested tuples, *i.e.* k, the distribution of attribute values, and the number of sub-domains of each attribute.

4.1 Experimental Setup

Our implementation and simulation are based on Chord [23] which is an efficient DHT. We tested our algorithms over a cluster of 64 nodes connected by a 1-Gbps network. Each node has two Intel Xeon 2.4 GHz processors, and runs the Linux operating system. We make each node act as a peer in the DHT.

To study the scalability of our algorithm far beyond 64 peers, we also implemented a simulator using SimJava. After calibration of the simulator, we obtained simulation results similar to the implementation results up to 64 peers. Thus, since the simulator allows us to study larger systems and due to space limitations, we only report simulation results for most of our tests.

Our default settings for different experimental parameters are shown in Table 1. Most of these settings are the same as in [7]. In our tests, we use a synthetically generated relation with six attributes a_i, $1 \le i \le 6$ and the domain of the attributes is numeric. The default number of tuples of the relation is 100,000 and they are randomly generated in two different ways: (1) Uniform data set, and (2) Gaussian data set. With (1), the values of attributes are independent of each other, and the distribution of the values of each attribute is uniform. This is our default setting. With (2), the values of different attributes are independent of each other, and the values for each attribute are generated via overlapping multidimensional Gaussian belles.

In our tests, the top-k query Q is delivered to a randomly selected peer. The selectivity of Q over the generated data is 10% and the scoring function specified in Q is the linear function $f(a_1, a_2, a_3, a_4, a_5, a_6) = a_1 + a_2 + a_3 + a_4 + a_5 + a_6$. Typically,

users are interested in a small number of top answers, thus we set *k=10*. In our storage mechanism, the domain of each attribute is uniformly partitioned into *n* sub-domains and the default value for *n* is 100. The network parameters of the simulator are shown in Table 2. We use parameter values which are typical of P2P systems [21]. The simulator allows us to perform tests up to 10,000 peers, after which the simulation data no longer fit in RAM. This is quite sufficient for our tests.

To evaluate the performance, we measure the following metrics. 1) Response time: the time elapsed between the delivery of Q to p_{int} and the end of the algorithm. 2) Communication cost: the total number of bytes which are transferred over the network for executing a given top-k query.

Table 1. Default setting of experimental parameters

Parameter	Default values
Number of tuples	100,000
K	10
Number of attributes	6
Data set	Uniform
Data selectivity	10 %
Number of attribute's sub-domains	100

Table 2. Network parameters of the simulator

Parameter	Default values
Bandwidth	Normally distributed random, Mean = 56 Kbps, Variance = 32
Latency	Normally distributed random, Mean = 150 ms, Variance = 100
Number of peers	10,000 peers

We evaluated the performance of our three algorithms DHTop1, DHTop2, and DHTop3. We also compared our algorithms with the simple algorithm which we introduced in the introduction of this paper. The simple algorithm, which we denote as Simple_DHT, retrieves all tuples of the relations involved in the query, computes the score of each retrieved tuple, and finally returns the *k* tuples whose scores are the highest.

4.2 Scale up

In this section, we investigate the scalability of our algorithm. For this, we study the effect of the number of peers and also the number of stored tuples on performance.

Effect of the Number of Peers

We used both our implementation and our simulator to study the response time while varying the number of peers. Using our implementation over the cluster, we ran experiments to study how response time increases with the addition of peers. Figure 1 shows the response time of our three algorithms and the Simple-DHT algorithm with the addition of peers up to 64. In all four algorithms, the response time grows logarithmically with the number of peers. However, the response time of our algorithms is much better than Simple_DHT.

Using simulation, Figure 2 shows the response times of the four algorithms with the number of peers increasing up to 10000 and the other parameters set as in Table 1 and Table 2. Overall, the experimental results correspond qualitatively with the simulation results. However, we observed that the response time gained from our experiments over the cluster is slightly better than that of simulation, simply because of faster communication in the cluster.

Effect of the Number of Tuples

We used our simulator to study the response time while varying the number of tuples which we store for each relation in the DHT. Figure 3 shows how the response time of our three algorithms increases with the number of tuples, using our simulator with the

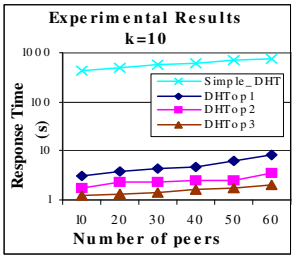

Fig. 1. Response time vs. number of peers

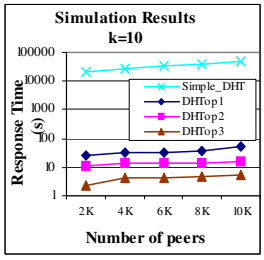

Fig. 2. Response time vs. number of peers

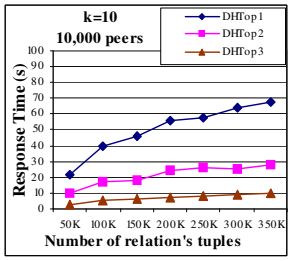

Fig. 3. Response time vs. number of relation's tuples

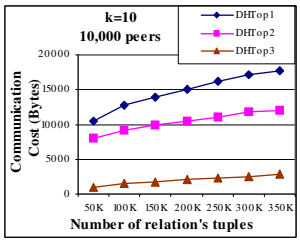

Fig. 4. Communication cost vs. number of relation's tuples

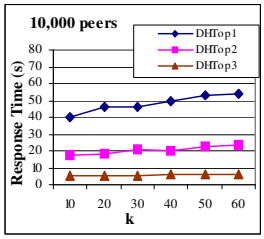

Fig. 5. Response time vs. *k*

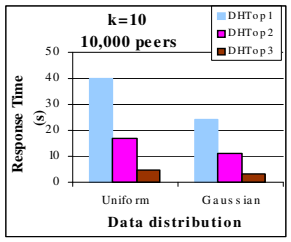

Fig. 6. Response time over uniform & Gaussian data sets

other parameters set as in Table 1 and Table 2. The number of tuples has a very small impact on DHTop3, because it does not need to retrieve stored tuples for computing the scores. The impact of the number of tuples on the response time of DHTop2 is less than DHTop1 due to the optimizations which we proposed for DHTop2. We also tested the communication cost of our algorithms. Using the simulator, Figure 4 depicts the number of bytes with increasing numbers of tuples, with the other parameters set as in Table 1 and Table 2.

4.3 Effect of other Parameters

We studied the effect of *k*, *i.e.* the number of top tuples requested, on response time. Figures 5 shows how the response time increases with *k*, using our simulator with the other parameters set as in Table 1 and Table 2. As expected, the response time of our algorithms increases with *k* because more tuples and attribute values are needed to be retrieved in order to obtain *k* top tuples. However, the increase is very small.

We investigated the response time of our algorithms over two Uniform and Gaussian data sets that we have generated synthetically, as described in Section 4.1. Using our simulator, Figure 6 shows the response time of our algorithms over Uniform and Gaussian data sets, with the other parameters set as in Table 1 and 2. The response time of our algorithms over the Gaussian data set is much better than their response time over the Uniform data set. The reason stems from that, in the Gaussian distribution, a high percentage of generated values are around the mean value and a very small percentage of the values are in the extremes. This characteristic of the Gaussian distribution makes the end condition of our algorithms hold sooner over the Gaussian data set than over the Uniform data set.

We studied the effect of the number of attributes' sub-domains, *i.e. n*, on performance (due to space limitations we do not show the figure). The results show that for the case of issuing a lot of simultaneous queries, increasing *n* reduces the average response time because it increases the number of peers that are responsible for maintaining the values of an attribute, so the load of each peer decreases.

5 Conclusion

In this paper, we addressed the problem of efficient top-k query processing in DHTs. We first proposed a mechanism for data storage in DHTs which provides good

support for exact-match queries and enables efficient execution of our top-k query processing algorithm. Then, we proposed a family of algorithms which deal with efficient processing of top-k queries in DHTs. We evaluated the performance of our algorithms through implementation over a 64-node cluster and simulation using SimJava. The results showed the effectiveness of our solution for processing top-k queries in DHTs.

6 References

[1] R. Akbarinia, E. Pacitti and P. Valduriez. Reducing Network Traffic in Unstructured P2P Systems Using Top-k Queries. *Distributed and Parallel Databases 19(2)*, 2006.

[2] B. Babcock and C. Olston. Distributed Top-K Monitoring. *SIGMOD Conf.*, 2003.

[3] W.-T. Balke, W. Nejdl, W. Siberski and U. Thaden. Progressive Distributed Top k Retrieval in Peer-to-Peer Networks. *ICDE Conf.*, 2005.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald and G. Weikum. IO-Top-k: Index-access Optimized Top-k Query Processing. *VLDB Conf.*, 2006.

[5] R. Blanco, N. Ahmed, D. Hadaller, L.G.A. Sung, H. Li and M.A. Soliman. A Survey of Data Management in Peer-to-Peer Systems. Technical Report CS-2006-18, University of Waterloo, 2006.

[6] P. Cao and Z. Wang. Efficient Top-K Query Calculation in Distributed Networks. *PODC Conf.*, 2004.

[7] S. Chaudhuri, L. Gravano and A. Marian. Optimizing Top-K Selection Queries over Multimedia Repositories, *IEEE Trans. on Knowledge and Data Engineering 16(8)*, 2004.

[8] G. Das, D. Gunopulos, N. Koudas and D. Tsirogiannis. Answering Top-k Queries Using Views. *VLDB Conf.*, 2006.

[9] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. System Sci. 58 (1)*, 1999.

[10] R. Fagin, J. Lotem and M. Naor. Optimal aggregation algorithms for middleware. *J. of Computer and System Sciences 66(4)*, 2003.

[11] J. Gao and P. Steenkiste. An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems. *IEEE Int. Conf. on Network Protocols (ICNP)*, 2004.

[12] U. Güntzer, W. Kießling and W.-T Balke. Optimizing Multi-Feature Queries for Image Databases. *VLDB Conf.*, 2000.

[13] M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. *Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[14] G.R. Hjaltason H. and Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS), 28(4)*, 2003.

[15] R. Huebsch, J. Hellerstein, N. Lanham, B.T. Loo, S. Shenker and I. Stoica. Querying the Internet with PIER. *VLDB Conf.*, 2003.

[16] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. *PODS Conf.*, 2006.

[17] S. Michel, P. Triantafillou and G. Weikum. KLEE: A Framework for Distributed Top-k Query Algorithms. *VLDB Conf.*, 2005.

[18] K. Mouratidis, S. Bakiras and D. Papadias. Continuous monitoring of top-k queries over sliding windows. *SIGMOD Conf.*, 2006.

[19] S. Nepal and M.V. Ramakrishna. Query Processing Issues in Image (Multimedia) Databases. *ICDE Conf.*, 1999.

[20] S. Ratnasamy, P. Francis, M. Handley, R.M. Karp and S. Shenker. A scalable content-addressable network. *SIGCOMM Conf.*, 2001.

[21] S. Saroiu, P.K. Gummadi and S.D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc. of Multimedia Computing and Networking (MMCN)*, 2002.

[22] A. Silberstein, R. Braynard, C.S. Ellis, K. Munagala and J. Yang. A Sampling-Based Approach to Optimizing Top-k Queries in Sensor Networks. *ICDE Conf.*, 2006.

[23] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proc. of SIGCOMM*, 2001.

[24] M. Wu, J. Xu, X. Tang and W-C Lee. Monitoring Top-k Query in Wireless Sensor Networks. *ICDE Conf.*, 2006.