Retrieving subset of result before completing top-k query

Takumi Okazaki

Visiting Scholar (NEC Corporation)
Department of Computer Science
Stanford University

August 25, 2005

Abstract

As the power of PCs is getting better, Peer-to-peer (P2P) architectures are becoming popular. Pure P2P does not have servers, and peers (PCs) share their resource. A P2P strategy can reduce the cost of the system.

One of the basic problems is to search by keyword in a P2P system. To enable the efficiency of P2P, search information should be managed in a distributed fashion not in central servers.

One approach to reducing query cost is to search only the top-$k$ elements, in order not to calculate the complete results.

We focus on the top-$k$ algorithm and propose a method to recognize some objects are certainly among the top-$k$ before completing top-$k$ algorithm.

1 Introduction

As the processing power and storage capacity increases rapidly, it becomes more efficient to interconnect each computer directly, without a central server. Then, a Peer-to-peer (P2P) architecture, which means to communicate from peer (peer can be considered as a PC) to another peer, has been studied recently[4][5]. Therefore, many P2P architectures are used through the Internet[3].

Introducing P2P can reduce communication and storage cost because P2P does not need expensive servers. P2P has high availability because P2P does not rely on some special servers. Therefore, people who do not
have enough money to provide some service to the public with expensive servers can do that with a P2P architecture.

Recently, DHT (Distributed Hash Table) has been studied by many researchers[9][10]. It is a type of P2P architecture which finds some contents users want faster. Therefore, DHT becomes a popular architecture of P2P.

There are some challenges in building P2P systems. One of the problems is to search contents by keywords. P2P manages huge amount of data, and P2P can find some content by querying the name of the content because P2P manages its contents by categorizing the name of the content. If some servers are used for managing the queries by keywords, this query system itself is not a P2P architecture and it does not have the efficiency of P2P architectures, such as reducing the cost.

We focus on the query by keyword in DHT architecture. Given a query, each data object matches the query with some "score". Suppose that we would like to find objects which have the highest scores. When some data for querying are too large, it is not a good idea to calculate the scores of all the documents. The top-k query is a query which finds the k objects that have the highest scores. Top-k query can reduce the cost of the query because it does not calculate the complete results for all the documents.

Many algorithms of top-k queries have been studied. Most of them focus on the efficiency of calculating the complete top-k results. We observe the process of top-k queries and a method to identify objects that are among exact top-k, before completing the calculation of obtaining top-k results. This method uses a new threshold value for determining which objects can be considered as exact top-k. We also describe the application of this query method for a DHT architecture.

In Section 2, we define the problem of top-k and present our idea. In Section 3, we describe the results of our experiments using our idea. In Section 4, we discuss the efficiency of our method. In Section 5, we show related works to our research. Then, in Section 6, we conclude and discuss future work.

2 Top-k Query Description

2.1 Top-k Problem Definition

There are m lists, $L_1, \ldots, L_m$. Each list entry consists of an object and its score. Assume that the score of object $d$ in list $L_i$ is $s_i(d) > 0$. Score
\( s_i(d) = 0 \) if there is no object \( d \) in the list \( L_i \). List \( L_i \) is sorted by scores \( s_i \) in descending order.

Let \( S(d) \) be the total score of object \( d \) among all the lists, that is, 
\[
S(d) = \sum_{i=1}^{m} s_i(d).
\]
The top-\( k \) query retrieval problem is to discover the \( k \) objects \( d_1, d_2, \ldots, d_k \) that have the \( k \) highest scores, in other words, the scores \( S(d_1), S(d_2), \ldots, S(d_k) \) are the \( k \) highest among all the objects.

The top-\( k \) retrieval formulation can be easily mapped to a text search context. A document can be considered as an object, and the list \( L_i \) can represent an inverted list which is the list of document, score pairs for a specific word \( w_i \).

### 2.2 Basic Concept of Top-\( k \) Query and Our Idea

Many top-\( k \) searching techniques have been studied. Consider that lists \( L_1, \ldots, L_m \) are distributed in different storage devices and it is expensive to read whole lists \( L_1, \ldots, L_m \) because of the distance of storages or the amount of the lists. The goal of top-\( k \) searching is to avoid to calculate the whole searching results and get some results faster. Further more, users do not need to have the complete results of top-\( k \) search. If there is some information about top-\( k \) results before getting the complete top-\( k \) results, it is useful for users. For example, if some subset of top-\( k \) are recognized in advance, users can see the part of the results faster. The concept of our idea is to observe the possibility that each object is exact among top-\( k \) while computing the complete top-\( k \) results. Therefore, we can get some information to know the candidates of top-\( k \) faster.

### 2.3 TA-Sorted Algorithm

For solving top-\( k \) queries, the Threshold Algorithm (TA) \[6\] is well-known. TA seeks from the top of each list \( L_1, \ldots, L_m \) and calculates \( S(d) \) when document \( d \) is found at first in some list. That is, when document \( d \) is found in some list, TA tries to find document \( d \) in other lists immediately. Therefore, TA needs random access to the data list for document \( d \). Therefore, it does not work if the data structure of lists does not allow random access. To address this issue, the TA-Sorted Algorithm was developed\[12\]. It computes the upper bound score and the lower bound score of all the objects.

TA-Sorted Algorithm keeps the list of the objects which have been found and the range of their scores. TA-Sorted works well when the data of the lists can searched only sequentially from the top, that is, when that data cannot be scanned randomly.
Table 1: Example of data lists

<table>
<thead>
<tr>
<th></th>
<th>L₁</th>
<th></th>
<th>L₂</th>
<th></th>
<th>L₃</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
<td>Score</td>
<td>ID</td>
<td>Score</td>
<td>ID</td>
<td>Score</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1 shows an example. Each word \((w₁, w₂, w₃)\) has its own list \((L₁, L₂, L₃)\) of object IDs and scores. All the elements in each list are ordered by scores in descending order.

In the example Table 1, suppose that for each word, only first object has been scanned, in other words, only object 1 in \(L₁\), object 2 in \(L₂\) and object 3 in \(L₃\) have been scanned. In \(L₁\), object 1 has the score 10. In \(L₂\), the score of object 1 cannot be more than 10 because \(L₂\) is sorted by score and the largest score is 10 and scores are not negative by definition. Similarly, the algorithm knows that objects 1, 2, and 3 have scores between 10 and 30. Any other objects at this time are known to have a score between 0 and 30. We refer to objects that have not been scanned by the algorithm as “unknown” objects.

At some point, let set \(E(d)\) have the list element which has already been scanned for object \(d\). And let \(t_i\) be the value which is the score of the object that has just been scanned, in other words, an “unknown” object may have the score \(t_i\) as its maximum in \(L_i\). By using these definitions, the range of the score objects may have at this specific time can be written as follows;

\[
\sum_{i \in E(d)} s_i(d) \leq S(d) \leq \sum_{i \in E(d)} s_i(d) + \sum_{i \notin E(d)} t_i
\]

Suppose that the data in the Table 1 have been scanned, but scanning has not been completed yet, in other words, there may be more elements below Table 1. Then, each document may have the range of the score \(S(d)\) as shown in 2.

For example, object 1 is in \(L₁\) with score 10 and \(L₂\) with score 8. Therefore, the minimum score of \(L₁\) is 18. And there may be object 1 just below object 4 in \(L₃\) with score 5. Then, the maximum score is 23.

Table 2 illustrates a simple way to implement TA-Sorted. From the range expression, objects can be categorized by set \(E(d)\) [12]. If object \(d₁\) and \(d₂\)
Table 2: Range of score of Table 1 example

<table>
<thead>
<tr>
<th>Object ID</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>φ</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

have the same set $E(d)$, the value of the difference between maximum and minimum is the same. Then, we can make $2^m$ queues to store the data of Table 2 instead of managing huge number of elements in each list. Assume that $m$ is much smaller than the number of elements in any list.

Table 3 shows the image of the above-mentioned queues for Table 1. The difference is a gap from the maximum score to the minimum score. In each queue, objects are stored with their ID and the minimum scores and sorted by their minimum scores. For example, for category $w_1, w_2$, object 1 and 6 have the same range of difference, briefly 5 because scores in $L_1$ and $L_2$ have been recognized and $L_3$ has no element of either of them and the last score of object in $L_3$ is 5. We can consider a virtual queue for category φ representing objects that have not been seen yet. In this example, those object have scores between 0 and 12.

Then, consider that objects are sorted by the value of the above minimum score in descending order. Let the $k$-th value of that be the threshold value. If some object has no possibility to have the score larger than that value, that object is certainly not among top-$k$ because at least $k$ objects at the lists ordered in the above condition have larger scores than that threshold value. We say this threshold as a lower threshold in order to distinguish this threshold from another threshold which we are proposing.

### 2.4 Our Idea: Upper Threshold

Then we propose another threshold value. Consider that objects are sorted by the value of the above “maximum” score in descending order. Let the $k$-th value be a new threshold value called upper threshold. If some object
Table 3: Image of Queues for data Table 1

<table>
<thead>
<tr>
<th>Category of words</th>
<th>Difference</th>
<th>Queue of objects (ID,score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1, w_2, w_3$</td>
<td>0</td>
<td>(4,20)</td>
</tr>
<tr>
<td>$w_1, w_2$</td>
<td>5</td>
<td>(1,18) (6,9)</td>
</tr>
<tr>
<td>$w_2, w_3$</td>
<td>4</td>
<td>$\phi$</td>
</tr>
<tr>
<td>$w_1, w_3$</td>
<td>3</td>
<td>(3,14)</td>
</tr>
<tr>
<td>$w_1$</td>
<td>8</td>
<td>$\phi$</td>
</tr>
<tr>
<td>$w_2$</td>
<td>9</td>
<td>(2,10)</td>
</tr>
<tr>
<td>$w_3$</td>
<td>7</td>
<td>(5,9) (7,7)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>12</td>
<td>All other unknown objects</td>
</tr>
</tbody>
</table>

has no possibility to have the score smaller than that value, that object is certainly among top-$k$ because at most $k$ objects at the lists ordered in the above condition have larger scores than this threshold value.

By using the same queue architecture as that of TA-Sorted algorithm, that threshold can be calculated because those queues are sorted by minimum values of their scores, and simultaneously by maximum values due to the categorization.

For example, the query of top-2 is done and the present situation is shown as Table 1, 2 and 3. From Table 2, the second largest maximum value is 20. Therefore, the upper threshold is 20. Then, we see the minimum scores and we can see that the object 4 is equal to (or more than) 20. We can say that object 4 is among exact top-2. Object 4 can be found before calculating complete top-2 result. And after recognizing object 4 is among top-$k$, object 4 can be pruned.

And the second largest maximum value can be calculated from Table 3. The first object in each queue is a candidate of the maximum value. Then to compare the values of scores with the distances which is the difference from the minimum and the maximum values and we can find the object with the maximum score. Then, the pointer for the queue where the maximum score has been found increases. And do the similar procedure and we can find the $k$-th score.

Then, we consider another query of top-10 for the same lists shown as Table 3. In that situation, only 7 objects have been found. Therefore, we cannot find the 10th score of object. But, we can set the maximum of unknown objects, that is 12, as the upper threshold. We can say that
objects which have the larger minimum score than 12 are among top-7 at the same reason mentioned above. And objects among top-7 are also among top-10. In this example, object 1, 3 and 4 have larger minimum score than 12. Therefore they are certainly among top-10.

The algorithm to find exact top-$k$ algorithm costs some amount. That depends on $2^m$ and $k$. (By a greedy algorithm, it costs $O(k2^m)$.) But, we can run this algorithm at any time we want. If the cost of the algorithm is sensitive, we can reduce the frequency of running the algorithm to decrease the total cost.

2.5 Application for DHT

Our idea can be applied for DHT architecture of P2P. For DHT, how to manage inverted lists is the problem of introducing top-$k$ query. As well as the idea of ODISSEA[11], there are two main methods to manage inverted lists.

One is to manage lists locally, which means each peer has to manage the lists for objects that belong to that peer. The benefit of this approach is to reduce the cost when objects are added/removed because only one peer has to manage that change. But, if inverted lists are distributed locally as well, one inverted list is scattered and we have to access almost the all peers in order to know the whole inverted list.

The other is to manage the data globally, lists are divided by words and those words can be searched by DHT architecture. This approach can reduce the cost of top-$k$ querying because at most $m$ peers ($m$ is the number of query words) have to be scanned for recognizing inverted lists.

The answer of the question: which option is better, depends on the whole system that would like to be developed, for example, which feature is important. But, for optimizing the cost of the query, the larger DHT system is, the better the latter method is because the former method has to access almost all the nodes in order to collect the data of lists.

3 Experimental Results

3.1 Environment

In order to evaluate our algorithm, we developed a simulator to output inverted lists. This simulator produces a random inverted list from the number $x$ of objects in the inverted list, and the total number $y$. By running this simulator $n$ times, $n$ inverted lists can be produced. In each inverted
list, $x$ objects are randomly selected among $y$ objects and random scores are assigned to them. Then, each inverted list is sorted by object score. For each query, we assume $m = 3$, in other words, the number of query words is 3.

We used a Linux machine with an Intel® Pentium® 4 2.80GHZ CPU in order to run the simulator and our algorithm.

### 3.2 Results

Figure 1 shows the result of Top-$k$ for three different data sets. The $x$-axis represents the position of inverted lists where our algorithm is working. The $y$-axis represents the number of objects that have been found as exact top-$k$ objects. Each data set has three lists, and the number of words for the query is three. And each list has $x = 10,000$ objects which are chosen among $y = 50,000$ objects at random.

Each line in Figure 1 shows a similar trend. It means our algorithm works similarly for random data.

All of the lines start at around position 2000, which is about one-fourth of
the position the complete top-\(k\) algorithm ends. In order to say some objects are among top-\(k\), those score must be larger than the upper threshold and it needs some amount of seeking. Before seeking position 2000, scores while seeking are too high that any lower bound scores cannot be larger than the upper threshold. But, after finding one top-\(k\) element, another elements can be found gradually. Some top-\(k\) elements can be found in advance.

Figure 2 shows the difference when the number \(k\) is changed to 50, 100, 150 and 200. The data 1 which is used in the experiment of Figure 2 is used. Therefore, for this data, \(x = 10000\) and \(y = 50000\).

When \(k\) increases, the position where objects are found first also increases. The threshold determining exact top-\(k\) objects increases when \(k\) increases because the threshold is the \(k\) highest maximum score. For \(k = 100, 150, 200\), all top-\(k\) elements cannot be found until almost all the inverted lists have been scanned. Even in such cases, some objects can be also found as top-\(k\) in advance.

And then, Figure 3 shows the difference of changing the number \(y\) to 10,000, 20,000 and 50,000. The number \(x\), the number of objects in each inverted list is 10,000. The graph says the fewer \(y\) is, the faster the complete
Figure 3: Top-50 search for changing the area of objects
result and the first found result are found. That is because $y$ is small, it is easier for an object to be found in all the inverted lists. Therefore, at the initial step, some objects have been found in all the inverted lists and their minimum score is fixed at the large rate.

In any case, some objects can be found before completing top-$k$ algorithm.

4 Discussion

We have to say that the experiments are based on random data. Therefore, if data are based on another distribution or some inverted list depends on another one, the behavior of the results may be changed.

In any experiment, the upper threshold algorithm finds top-$k$ elements gradually. Therefore, reducing the frequency of evaluating upper threshold works. After pausing the upper threshold algorithm for a while, it is expectable to have another top-$k$ elements.

And in any experiments, some objects can be found as top-$k$ elements as three or more times faster, compared to the complete top-$k$ results, from the view of positions. Therefore, if actual users just want to have some subset of the top-$k$ result, this algorithm works well.

5 Related Works

Many DHT ideas which manage the whole P2P system have been studied. Among these, CAN [9] and CHORD [10] are famous.

There is some work about queries in DHTs. J. Li et al [7] discusses the feasibility of querying in P2P search and they think the cost of querying is too expensive to use normally. Therefore, there are some challenges for solving the cost of querying. ODISSEA [11] is one of these to propose some basic idea about querying in DHT. One of the ideas to narrow down the area of seeking is to introduce Bloom Filters [1] [8].

The top-$k$ problems have also been studied. P. Cao and Z. Wang propose the method of defining proper thresholds for retrieving top-$k$ efficiently from distributed network [2]. M. Theobald et al show the probabilistic view to retrieve top-$k$ results [12].
6 Conclusion

We proposed another threshold called upper threshold for recognizing that some objects are certainly among top-\(k\) before completing top-\(k\) query. From random inverted lists, the upper threshold algorithm can retrieve some subset of top-\(k\) result. Especially for the initial objects, they can be found three or more times faster than completing top-\(k\) query.

Future work is to implement the upper threshold algorithm in a real DHT and do some experiments, to combine the probabilistic top-\(k\) algorithm and the upper threshold algorithm, and to examine some inverted lists based on another distributions.

Acknowledgment

I would like to thank Prof. Hector Garcia-Molina for his continuous support of my research activities at Stanford University. I also appreciate Prof. Pei Cao for her help of my initial stage of my research and Daishi Kato for giving me many useful comments about my research. I thank Mayank Bawa and Andy Kacsmar for the help with the computing environment. I would also like to thank the other InfoLab members at Stanford and many members of NEC Corporation who supported me.

References


