Unified Framework for Flexible and Efficient Top-k Retrieval in Peer-to-Peer Networks

William Conner[1], Seung-won Hwang[2], and Klara Nahrstedt[1]

[1] Department of Computer Science,
University of Illinois, Urbana, IL, USA 61801-2302
{wconner,klara}@uiuc.edu

[2] Department of Computer Science and Engineering,
Pohang University of Science and Technology, Pohang, Korea
swhwang@postech.ac.kr

Abstract* — As more and more data from distributed data sources becomes accessible, supporting queries over peer-to-peer networks of such data sources becomes a more convincing application scenario. In such an application scenario, a large scale of accessible data from multiple peers naturally calls for ranked retrieval in order to effectively focus the retrieval on the most relevant, say top-k results. While top-k retrieval has been actively studied lately, existing algorithms are too restrictive due to their assumptions about the predicates and scoring functions used. These restrictive assumptions limit the flexibility of individual users to issue personalized queries. In contrast, we present efficient algorithms that support top-k retrieval customized to the specific predicates and scoring functions desired by the users. Also, unlike existing approaches that only consider a single type of data partitioning, we generalize the application scenario to include peer-to-peer networks of a potentially large number of peers that might partition the data in various ways. More specifically, we develop a unified top-k query processing framework to cover the following types of data partitioning: (1) *vertical partitioning* where each peer stores partial scores of an identical set of data objects, (2) *horizontal partitioning* where each peer stores complete scores of a disjoint set of data objects, and (3) *mixed partitioning* where each peer stores partial scores of a disjoint set of data objects. In particular, we customize queries from users by transforming data synopses on a per-query basis. We also reduce bandwidth consumption by using heuristics to schedule the order in which predicates are evaluated. Our results validate the efficiency and effectiveness of our framework by considering the bandwidth consumption, delay, and correctness of our algorithms.

Keywords: Top-k Query Processing, Ranked Retrieval, Peer-to-Peer, Content Distribution Networks, Multimedia

Technical Areas: Data Management, Peer-to-Peer

Corresponding Author: wconner@uiuc.edu

1. INTRODUCTION

Peer-to-peer systems are becoming a common architecture for sharing large amounts of data [5, 6, 7]. Applications running on top of peer-to-peer architectures range from storage systems [8, 10] to high-bandwidth content distribution [9]. Due to the popularity of peer-to-peer systems and the wide range of potential applications that will be built on top of them, it would be useful to support intelligent data retrieval techniques, which help users identify the most relevant results without suffering from information overload due to searching through large amounts of data.

As a key mechanism to reduce information overload, ranked retrieval [1, 2, 3, 4] has been actively studied. Ranked retrieval, which orders retrieval results according to relevance, helps users focus on the top few results and safely ignore the less relevant results. A *top-k query*, or ranked query, selects the k most relevant answers among a database of n database objects ($d_1, \ldots, d_n$). This selection involves two steps. First, each one of the m fuzzy predicates ($p_1, \ldots, p_m$) on each of those data objects must be evaluated to a partial score in the range [0:1]. Second, those partial scores must be aggregated using some scoring function F (e.g., *min*) to produce an overall relevance score for each database object.

To motivate ranked retrieval in peer-to-peer systems, consider a peer-to-peer content distribution network (CDN), such as [9,22], that streams video files to various clients. Each participating site in the CDN has an administrator that occasionally wants to identify videos matching certain criteria. Based on the video files' metadata, we can build a logical relation *video_meta_data* that stores the following three columns: video identifier (unique key), number of accesses to that particular video (attribute), and the bit rate of that video (attribute). The relation *video_meta_data* can be partitioned and distributed among the various peering sites in the system using one of the data partitioning schemes that we will discuss later. An example query is shown in Example 1.

Example 1
A site administrator may ask a ranked query to find the top-5 videos that are both popular and of a certain quality, as query Q_1 illustrates below (in SQL-like syntax):

select *id* **from** *video_meta_data vmd*
order by *min*(p_1 : *popular*(*vmd.num_accesses*), p_2 : *quality*(*vmd.bitrate*))
stop after 5
(Query Q_1)

Using some interface support, the administrator describes her preferences over the attributes as *fuzzy predicates*. In Example 1, *popular* and *quality* are fuzzy predicates defined over the number of accesses and bit rate, respectively. A fuzzy predicate is a function applied to one or more attributes that returns a score between 0 and 1. For instance, assuming that the maximum bit rate possible for a video in the CDN is MAXBITRATE (e.g., 500 Kbps), an administrator interested in videos encoded at 256 Kbps may describe her preference by defining the following predicate *quality* over the bit rate attribute values.

$$\textit{quality}(\text{bitrate}) = 1 - \frac{|\,\text{bitrate} - 256\,|}{\text{MAXBITRATE}}$$

Based on the predicate *quality*, a video with bit rate equal to 256 Kbps will score the highest possible score 1.0 for that predicate, while videos with significantly higher bit rates or significantly lower bit rates will score much lower. Naturally, this predicate definition is specific to user needs. For instance, another administrator interested in videos downloaded by users with faster Internet connections might define *quality* to favor videos with higher bit rates (e.g., 350 Kbps or 500 Kbps). In addition to user-defined fuzzy predicates, the query specifies a user-specific scoring function *F*, indicating how to combine the predicate scores. For example, if *min* is our user-specific scoring function *F*, then $F(0.9, 0.8) = 0.8$. The highest scoring videos (according to *F*) will then be retrieved and appear in the user's query results.

Besides the *min* function used in the preceding query example, a top-k query algorithm typically supports arbitrary monotonic functions where all attribute scores contribute positively towards the score (e.g., average or weighted average with positive weights). However, although video metadata is logically represented as a single relation *video_meta_data*, it can be stored over multiple data sources in practice. For instance, there can be many peers where each stores the complete information on a different set of objects. We refer to this case as *horizontal partitioning*. For example, consider the case where each peering site stores all of the video metadata for the videos stored locally at some node within that site. Alternatively, peers may store different attributes of the same set of objects. We refer to this case as *vertical partitioning*. For instance, using our query Q_1 from Example 1, some peering site may store the numbers of accesses to each video in the system to evaluate the predicate *popular*, while another peering site stores the bit rate of each video in the system to compute predicate *quality*. Another possibility is the combination of both horizontal and vertical partitioning. We refer to this case as *mixed partitioning*. The following example illustrates these different peer data decompositions.

Example 2

Continuing from Example 1, consider a simple dataset of 5 videos.

OID	*Num_Accesses*	*Bitrate*
vid_1	10367	128
vid_2	9600	56
vid_3	834	256
vid_4	7809	500
vid_5	8200	350

First, this logical relation from Example 2 may be decomposed vertically across two peers, with the first peer p_1 storing the number of accesses to all videos while another peer p_2 stores the bit rates of all videos. Second, this logical relation may be decomposed horizontally across two or more peers. For instance, p_1 may store the number of accesses and bit rate values of the first three videos while p_2 stores the number of accesses and bit rate values of the last two videos. Third, the above two decompositions can be mixed arbitrarily. For instance, p_1 may store the number of accesses of the first three videos, while p_2 stores the bit rate of the first three videos and some other peer p_3 stores the number of accesses and bit rate values of the last two videos.

Keeping in mind that the logical relation can be decomposed in one of many different ways as illustrated above, a top-k algorithm must perform accesses on peers to gather enough scores to correctly identify the top-*k* results out of *n* database objects. However, since the user is interested in only a fraction of the data (i.e., $k << n$), the top-k results could potentially be identified by partial evaluation of a fraction of the data objects. To illustrate, consider horizontal decomposition in Example 2 with some peer p_1 storing complete scores for vid_1, vid_2, and vid_3. When the user wants to retrieve the top-1 video with bit rate around 56 Kbps and high popularity with *sum* as the aggregate scoring function *F*, we may terminate after accessing peer p_1 storing vid_2, since the aggregated score for vid_2 already outscores the upper bound score for the rest of the peers (i.e., videos stored at any other peer are less relevant). To take full advantage of these optimization opportunities, many top-k algorithms [1,2,3,4,11,13,14] have been studied lately for both local and distributed scenarios. However, all of these algorithms focus on a single type of data partitioning scheme. Also, many of these algorithms limit the flexibility of users regarding their predicates and scoring functions used. In contrast, our solution proposes a unified top-k query framework supporting arbitrary peer data decompositions (i.e., vertical, horizontal, and mixed) with arbitrary user-defined predicates and arbitrary monotonic scoring functions. To highlight, our main contributions are listed below:

- ***InformedDepth* algorithm**: Since the order in which predicates are evaluated can significantly affect the total number of objects retrieved overall during distributed query processing (as we will show later), we develop a predicate scheduling algorithm based on a greedy heuristic. Unlike another previous greedy predicate scheduling algorithm [4], our algorithm only needs data synopses as input to determine this schedule, which makes it suitable for distributed peer-to-peer environments.
- ***Unified top-k query processing framework***: We develop a unified framework over a wide range of peer-to-peer retrieval scenarios. In particular, in contrast to existing work focusing either on vertical or horizontal decomposition, we develop a unified solution for all cases, which is the first to the best of our knowledge.
- ***Efficiency and effectiveness***: We formally and empirically argue our framework is efficient and effective in terms of bandwidth consumption, delay, and correctness.

To further place our work in context, we review related work in the next section. We then discuss preliminaries on ranked retrieval and peer-to-peer networks in Section 3, as well as present our algorithms that efficiently identify top-k query results over peer-to-peer networks with different data decompositions. Section 4 presents a performance evaluation of our algorithms obtained by both simulation and analysis. Finally, Section 5 concludes our work.

2. RELATED WORK

Much work has been previously done on top-k query processing in centralized data management environments where issues related to network bandwidth and delays were not explicitly considered. Fagin's algorithm (FA) and the threshold algorithm (TA) were two of the earliest top-k algorithms [1,2]. One of the scenarios that initially motivated such top-k algorithms were multimedia database systems consisting of several subsystems that store different attributes of the database objects. For example, an image database might have the following two subsystems: color and texture. Unfortunately, both FA and TA would be prohibitively expensive in distributed scenarios since they require object retrievals at each step and the number of steps is not bounded by a constant. Therefore, the number of message rounds where each message round corresponds to one step could also be potentially high, which leads to large network delays.

Top-k algorithms have also been proposed for scenarios involving the retrieval of the most relevant objects from Web-accessible databases and evaluating top-k queries with expensive predicates [3,4]. Both of these algorithms are very suitable for Web search scenarios where accessing attributes might be restricted by some external Web interface. Unfortunately, neither of these algorithms explicitly considers peer-to-peer networks. However, the idea of greedy predicate scheduling presented in [4] has been significantly modified for suitable use in our algorithms.

Finally, many top-k query processing algorithms suitable for peer-to-peer environments have been recently proposed in the literature. TPUT is a threshold-based algorithm suitable for distributed environments [11]. TPUT limits the number of phases (or message rounds) to three. Using a novel data structure for data synopses, which we borrow in our algorithms, KLEE presents a family of approximate top-k algorithms [13]. Balke et al. present one of the few top-k retrieval algorithms that explicitly considers the underlying peer-to-peer network topology [14]. Our framework also explicitly considers the underlying network topology as we explain in our peer-to-peer network model in the next section. There are two major problems with current top-k query processing algorithms for peer-to-peer networks. The first problem is that many top-k algorithms implicitly assume a single type of data partitioning (i.e., vertical or horizontal). For example, KLEE assumes vertical data partitioning where each peer is responsible for a single attribute. In our framework, we explicitly consider how our algorithms can handle different types of data partitioning (i.e., vertical, horizontal, and mixed). The second problem with some of the existing approaches is that they have limited flexibility with respect to user-defined predicates and scoring functions. For example, the

TPUT algorithm description clearly shows how *sum* can be used as the scoring function in their algorithm, but it is not so clear how some other monotonic scoring function (e.g., *min*) would be plugged in. Our framework is adaptive in order to support user-defined predicates and arbitrary monotonic scoring functions.

3. UNIFIED FRAMEWORK

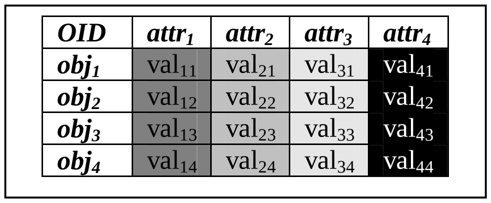
In our unified framework for top-k query processing in peer-to-peer networks, we propose a family of algorithms called the *InformedDepth* algorithms. Before discussing our proposed algorithms, it is important to cover some preliminary background information concerning the following: data model, query model, and peer-to-peer network model. After presenting these preliminaries, we then present the data structures and algorithms used in our framework.

3.1. Data Model

One of the shortcomings present in existing work on top-k query processing in peer-to-peer networks is that the existing algorithms are focused only on horizontal data decomposition or vertical data decomposition [11,13,14]. Unlike existing solutions, one of the main contributions of our unified framework is that it can handle all of the following types of data decomposition: vertical, horizontal, and mixed.

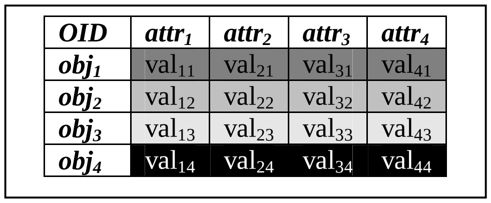
More formally, suppose we have some relation R with a set of attributes A and a set of database objects D on which values for those attributes are defined. In the following definitions, we identify three possible decompositions of a relation R into different peer data sources. Example data decompositions are illustrated in Figure 1 with different shades being assigned to one of four different peers.

1. *vertical decomposition*: each peer stores values from A' on data objects from D where $A' \subseteq A$
2. *horizontal decomposition*: each peer stores values from A on data objects from D' where $D' \subseteq D$
3. *mixed decomposition*: each peer stores values from A' on data objects from D' where $A' \subseteq A$ and $D' \subseteq D$



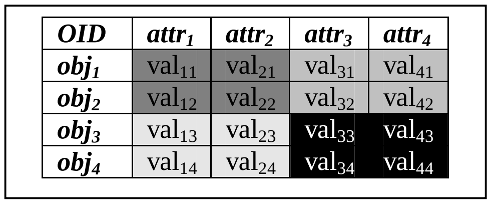
OID	***attr$_1$***	***attr$_2$***	***attr$_3$***	***attr$_4$***
obj$_1$	val$_{11}$	val$_{21}$	val$_{31}$	val$_{41}$
obj$_2$	val$_{12}$	val$_{22}$	val$_{32}$	val$_{42}$
obj$_3$	val$_{13}$	val$_{23}$	val$_{33}$	val$_{43}$
obj$_4$	val$_{14}$	val$_{24}$	val$_{34}$	val$_{44}$

(a) vertical decomposition



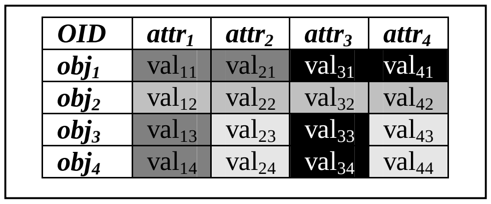
OID	***attr$_1$***	***attr$_2$***	***attr$_3$***	***attr$_4$***
obj$_1$	val$_{11}$	val$_{21}$	val$_{31}$	val$_{41}$
obj$_2$	val$_{12}$	val$_{22}$	val$_{32}$	val$_{42}$
obj$_3$	val$_{13}$	val$_{23}$	val$_{33}$	val$_{43}$
obj$_4$	val$_{14}$	val$_{24}$	val$_{34}$	val$_{44}$

(b) horizontal decomposition



OID	$attr_1$	$attr_2$	$attr_3$	$attr_4$
obj_1	val_{11}	val_{21}	val_{31}	val_{41}
obj_2	val_{12}	val_{22}	val_{32}	val_{42}
obj_3	val_{13}	val_{23}	val_{33}	val_{43}
obj_4	val_{14}	val_{24}	val_{34}	val_{44}

(c) grid mixed decomposition



OID	$attr_1$	$attr_2$	$attr_3$	$attr_4$
obj_1	val_{11}	val_{21}	val_{31}	val_{41}
obj_2	val_{12}	val_{22}	val_{32}	val_{42}
obj_3	val_{13}	val_{23}	val_{33}	val_{43}
obj_4	val_{14}	val_{24}	val_{34}	val_{44}

(d) arbitrary mixed decomposition

Figure 1. Different types of data decomposition

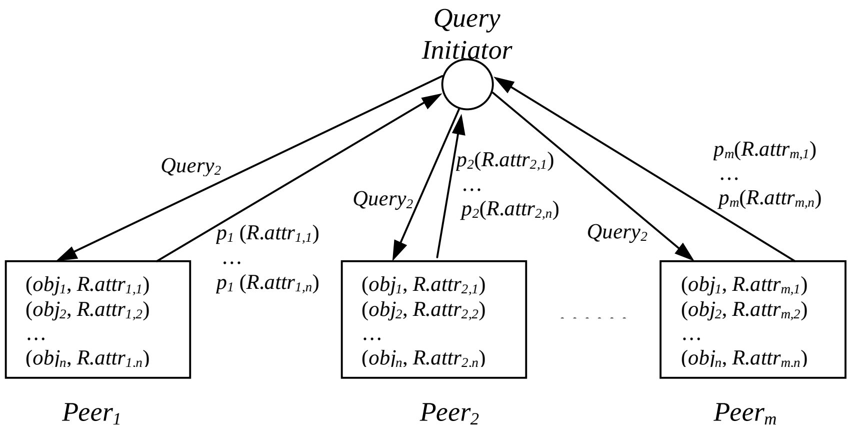
It is important to distinguish between the different types of objects in our data model. First, we have database objects (e.g., obj_1 in Figure 1) that correspond to tuples (i.e., rows) in the logical relation R. We also have attribute value objects (e.g., val_{11} in Figure 1) that correspond to attribute values for a particular database object. More specifically, attribute value objects map database object identifiers to some corresponding attribute value. Finally, when a user-defined predicate *pred* is applied to some attribute value object val_{xy}, $pred(val_{xy})$ evaluates to some partial score object $score_{xy}$.

In our data model, notice that we also make a distinction between two different types of mixed decompositions. When every peer is responsible for exactly m attributes and n objects, we refer to this case as *grid* mixed decomposition as shown in Figure 1(c). When different peers might be responsible for different numbers of attributes and objects, we refer to this case as *arbitrary* mixed decomposition as shown in Figure 1(d).

3.2. Query Model

Our query model assumes that some relation R has its data partitioned either vertically, horizontally, or mixed across several different peers in the network. A user will then define predicates $p_1,\ldots,p_m$ to apply to the various attributes $attr_1,\ldots,attr_m$ to obtain a score from each object for each predicate. To obtain an overall score for each database object *obj*, the user defines a scoring function F, which is applied to combine the various partial scores for that object. To avoid information overload, each user can also specify the maximum number of results to retrieve as k with the objects returned in order starting with the highest scoring objects according to F. Our query model is expressed in SQL-like syntax as Query 2 below. As with traditional databases, top-k queries expressed in declarative languages like SQL must be translated into execution plans. An example plan to retrieve the partial score objects for Query 2 with R partitioned vertically across m peers is illustrated in Figure 2.

select *obj*
from *R*
order by $F(p_1(R.attr_1), \ldots., p_m(R.attr_m))$
stop after *k*
(Query 2)



Query Initiator
Query2
p1 (R.attr1,1) … p1 (R.attr1,n)
Query2
p2(R.attr2,1) … p2(R.attr2,n)
Query2
pm(R.attrm,1) … pm(R.attrm,n)
(obj1, R.attr1,1) (obj2, R.attr1,2) … (objn, R.attr1.n)
(obj1, R.attr2,1) (obj2, R.attr2,2) … (objn, R.attr2.n)
(obj1, R.attrm,1) (obj2, R.attrm,2) … (objn, R.attrm.n)
Peer1
Peer2
Peerm

Figure 2. Example execution plan

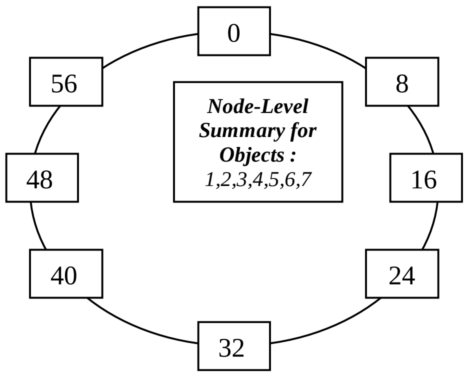
Observe that this class of top-k queries significantly extends existing peer-to-peer queries that focus on Boolean range queries. An example Boolean range query could be the following: select all objects that have an attribute value greater than some threshold. We believe the extension provided by top-k queries in peer-to-peer networks significantly improves on Boolean queries in peer-to-peer networks for several reasons. First, considering the potentially overwhelming amount of data accessible, Boolean queries often return too many results (i.e., information overload), while ranking queries enable effective retrieval of the highest scoring results of manageable size *k*. Second, user predicates can be defined at run time, which enables the retrieval of the most relevant results that best match the user's specific needs. Our query model enables us to support an effective ranked retrieval of desirable size for user-specific ranking criteria. With data and query models defined, we describe our peer-to-peer network model in the next section.

3.3. Peer-to-Peer Network Model

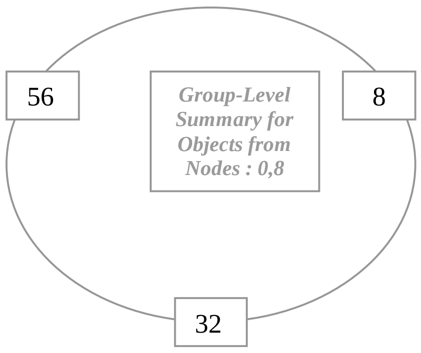
Since our algorithms make certain assumptions about the underlying peer-to-peer network organization, we must first explain our peer-to-peer network model before we describe our algorithms in detail. Our model assumes a peer-to-peer topology based on a ring-like distributed hash table (DHT), such as those described in [5,6,7]. Our algorithms are not limited to ring-like DHTs, but such DHTs have previously been shown to provide scalability and robustness [5,6,7]. Also, since many existing peer-to-peer applications are already built on top of ring-like DHTs (e.g., [8,9,10]), our top-k algorithms can be incorporated with these applications seamlessly without requiring the installation of a new peer-to-peer routing middleware layer.

In our framework, we assume that each node and object has a unique identifier. In our application, an object stored at a node consists of one or more attribute value objects for some corresponding database object (in the case of vertical and mixed decompositions) or consists of all of the attribute value objects for some corresponding database object (in the case of horizontal decomposition). It is important to note that an object identifier in our peer-to-peer network is not necessarily the same as the identifier for the corresponding database object. For example, val_{11} in Figure 1(a) with vertical decomposition, would be represented as an attribute value object that maps database object identifier obj_1 to some attribute value for attribute $attr_1$. Here, the peer-to-peer network object identifier is different from the database object identifier. Given the object's identifier, each object is uniquely assigned to a node using the object assignment rules for that particular ring-like DHT. For example, in Chord, each object is assigned to the node that *succeeds* it in the circular identifier space moving clockwise (see Figure 3 for more details) [5]. Therefore, the type of data partitioning and ring-like DHT used dictate how node and object identifiers will be assigned.

In addition to the underlying DHT that includes all of the nodes in the peer-to-peer network, which we will refer to as the *node-level DHT*, we also organize a subset of nodes, which we will refer to as *group leaders*, from the peer-to-peer network into another DHT on top of the node-level DHT to form a *group-level DHT*. Our group leaders and group-level DHT are somewhat analogous to the supernodes and backbone mentioned in [14], but our approach is unique in that we utilize groups in our top-k algorithms. Also, our underlying peer-to-peer network topology is DHT-based while the underlying topology presented in [14] is HyperCuP-based. All nodes in the node-level DHT are assigned to one of the group leaders using the object assignment rules for the DHT with their node identifiers from the node-level DHT treated as object identifiers in the group-level DHT as shown in Figure 3. A similar peer-to-peer network organization appears in [15].



(a) Node-level DHT



(b) Group-level DHT

Figure 3. Example peer-to-peer Network

Our reason for choosing this particular peer-to-peer network model is that we want to reduce the amount of statistical summary information that must be retrieved when queries are issued as explained in more detail in Section 3.4. In our peer-to-peer network, each individual node will store a statistical summary of all the objects for which it is responsible (i.e., summary of objects stored at that node). In addition to storing a local per-node statistical summary, the group leaders in the group-level DHT also store statistical summaries derived from the statistical summaries of all the nodes assigned to its group. We assume that the number of nodes *M* in the group-level DHT is much less than the number of nodes *N* in the node-level DHT. Therefore, based on previous routing analysis of ring-like DHTs, messages in the group-level DHT will traverse O(log *M*) hops while messages in the node-level DHT will traverse O(log *N*) hops [5,6,7]. However, we also assume that per-group summaries are approximately the same size as per-node summaries, which means per-group summaries have coarser details since the per-node summaries must be aggregated. Since we assume *M* << *N* with both summaries having roughly the same size, then routing messages over the group-level DHT is less expensive in terms of communication cost, but it also retrieves summaries with less detail. Consider the peer-to-peer network shown in Figure 3 with eight nodes in the node-level DHT and three of those eight nodes in the group-level DHT. Example summaries are only shown for node 8, which is also a group leader (remember that the corresponding per-node summaries are at each individual node and the corresponding per-group summaries are at each group leader).

3.4. InformedDepth Algorithms

We now complete our explanation of our framework by proposing top-k query processing algorithms for peer-to-peer networks with vertical, horizontal, or mixed data partitioning. We first discuss the HistogramBlooms data structure that we use for per-node and per-group summaries as well as how we adapt those summaries for the particular query issued. Next, we discuss the notion of selectivity that we use in our algorithm. Finally, we explain our proposed algorithms. Unless otherwise stated, our discussion is in the context of vertical partitioning. In Section 3.4.6, we show how other data decompositions can be reduced to the case of vertical partitioning.

3.4.1. HistogramBloom Structure

A variant of the HistogramBloom, introduced in [13], is the data structure used to provide per-node and per-group statistical summaries about the attribute values stored at a particular node or within a particular group, respectively. As the name suggests, HistogramBlooms are a combination of both histograms and Bloom filters (with Bloom filters being introduced in [18]). A histogram is an array of cells where each cell has a frequency count for the number of values that fall within the range of that cell. Each histogram cell's range is specified by a lower bound and an upper bound. A Bloom filter is a compact representation of set membership. Bloom filters consist of x hash functions and y bits initially set to 0. The x hash functions are independent and hash inputs into the range $[0,y\text{-}1]$. To insert an element e into the Bloom filter, we set each bit indexed by hash function $hash_i(e)$ for $i = 1,\ldots,x$ to the value 1. An element e is considered to be a member of a Bloom filter if each bit indexed by $hash_i(e)$ for $i = 1,\ldots,x$ is equal to 1. As described, the possibility of false positives exists in Bloom filters. Specifically, a Bloom filter with x hash functions and y bits that stores z elements has a false positive probability of $(1 - e^{xz/y})^x$ during a membership test [19].

A HistogramBloom is a histogram where the cells of the histogram, in addition to specifying a range and frequency count, also contain membership information about the members of each particular cell. The membership information for each cell is represented compactly by a Bloom filter. Figure 4 depicts a simple example of a HistogramBloom.

low	*upper*	*freq*	*BF*
0	5	6	1110……………….11
5	10	17	1011……………….11
10	15	12	1111……………….10
15	20	8	1000……………….01

Figure 4. Example HistogramBloom

In the above example, forty-three objects are distributed amongst the various HistogramBloom cells based on the object's value. The lower bound range for a cell is indicated by *low* and the upper bound is *upper*. The frequency distribution of object values into the cells is indicated by *freq* and the corresponding Bloom filter for each cell is represented by *BF*.

3.4.1.1. Estimating HistogramBloom Cell's Frequency Count

Our algorithm, presented in Section 3.4.5, requires us to determine the size of the membership (i.e., frequency count) of HistogramBloom cells at various steps. These HistogramBloom cells might be the result of combining two or more other cells. Such a frequency count can be stored with the Bloom filter as an additional field, or it can be determined by looking at the full, uncompressed membership vector for the corresponding HistogramBloom cell. Using a separate field for the frequency count of a cell in a HistogramBloom could be problematic when that cell is the result of combining two or more other cells (e.g., union or intersection). In many cases when Bloom filters are combined, we cannot determine the exact frequency count field of the resulting Bloom filter without full membership information. However, retrieving full membership information is extremely costly and defeats the initial purpose of compactly representing set membership with Bloom filters.

Therefore, we take the approach of estimating the frequency count for HistogramBloom cells that have been combined by taking the bitwise AND (i.e., intersection) or bitwise OR (i.e., union) of all of their corresponding bit positions. To estimate the frequency count, we add up the number of bits set to value 1 and divide that sum by the number of hash functions used in the cell's Bloom filter. If the number of collisions is small, then this estimate should give us an approximate frequency count. However, due to collisions, the possibility exists that our frequency count estimation could potentially underestimate or overestimate. For our simulations in Section 4, this estimation method performed well enough to get the correct results.

Reynolds and Vahdat propose a more expensive alternative for remote set intersection using Bloom filters that could be applied to get an accurate frequency count of the intersection of two membership lists represented by Bloom filters [16]. Rather than intersecting two Bloom filters directly (as we do), a peer P_1 sends its Bloom filter b_1 compactly representing its membership list L_1 to peer P_2. P_2 will then test each member of its own list L_2 for membership in b_1 and send the elements L_2' from L_2 thought to be in b_1 back to P_1. P_1 can then verify which elements of L_2' actually belong to L_1 to compute the exact intersection of the two membership lists. Of course, this alternative is much more expensive because it requires objects to be sent in addition to a Bloom filter.

3.4.2. Node-Level to Group-Level Summary

As mentioned earlier, each peer in our network maintains a per-node summary of the information stored at that particular node, which is then combined into a per-group summary stored at the group leader for that node. These group-level summaries provide a synopsis of all the node-level summaries from nodes that comprise that group. In order to build the group-level summary, we must assign the objects represented in the node-level summary into their corresponding place in the group-level summary.

First, we assume that each node-level HistogramBloom cell from a peer has a range corresponding to a subinterval of some group-level HistogramBloom cell's range. More formally, suppose that we have node-level HistogramBloom cell c_i from some peer p_i with range lower bound lb_i and range upper bound ub_i for c_i. Given any such c_i, we can find some group-level HistogramBloom cell c_j where $lb_i \geq lb_j$ and $ub_i \leq ub_j$. Based on this assumption, all members of a node-level HistogramBloom cell will be assigned to the same group-level HistogramBloom cell (i.e., members of same node-level cell are not split between two or more group-level cells).

At this point, adding a node-level summary into a group-level summary becomes straightforward. We simply take each node-level HistogramBloom cell and find the corresponding group-level HistogramBloom cell that completely covers its range. We then perform a bitwise OR operation on the corresponding Bloom filters of the two cells to add the node-level HistogramBloom cell to the group-level HistogramBloom cell. We can repeat this process for each cell of each HistogramBloom of each node belonging to a group.

3.4.3. Summary Adaptation

As mentioned in our introduction, one of our contributions is that our framework is flexible, which allows the user to submit personalized queries with arbitrary monotonic scoring functions and arbitrary user-defined predicates. This flexibility is partially achieved by adapting the statistical summaries on-demand, which are represented by HistogramBlooms, for the specific query.

Unlike previous work that assumes fixed predicates and fixed scoring functions that combine those predicates, we want to support user-defined predicates and scoring functions on-demand when a query is issued. For example, a user u_1 could define a predicate $quality_1$ (shown in Example 3) mapping attribute *bitrate* into a score indicating that u_1 prefers videos encoded at the highest possible bit rate. Similarly, user u_2 might define its own predicate $quality_2$ if u_2 prefers videos encoded as close as possible to 256 Kbps.

Example 3
$quality_1 = 1 - (bitrate / \text{MAXBITRATE})$
$quality_2 = 1 - ((bitrate - 256)^2 / \text{MAXBITRATE}^2)$

Supporting such user-defined predicates on-demand makes pre-materialized summaries infeasible because the system cannot anticipate which predicates the user will use until the query is issued. However, the other extreme of building a completely new HistogramBloom from scratch whenever a new query is issued is also infeasible due to the prohibitive computational costs of accessing every attribute value object for every database object to determine its predicate score. Therefore, we take a hybrid approach where we adapt the attribute-based HistogramBloom (this HistogramBloom is *predicate oblivious* and only needs to be computed once initially) by reordering its cells in decreasing order of the maximum predicate score possible for an object in that cell. For instance, for $quality_1$, we can order cells in the decreasing order of the upper bound of their respective *bitrate* ranges. Finding such ordering can be trickier for $quality_2$. Assume that the lower bound and upper bound for each cell *i*'s range is denoted by lb_i and ub_i, respectively. For $quality_2$, the maximum predicate score for each cell that does not contain bitrate 256 Kbps is given by $max(quality_2(lb_i), quality_2(ub_i))$, while the maximum score for the cell that contains bitrate 256 Kbps is $quality_2(256) = 1.0$.

Our summary adaptation process for any differentiable scoring function *DF* is to rearrange the HistogramBloom cells in decreasing order of their maximum predicate score possible, which is defined below.

Definition (Cell's Maximum Predicate Score Possible): Points on *DF* where its derivative is equal to zero are called *critical points*. The maximum predicate score possible of some predicate *p* is $max(p(lb_i),p(ub_i))$ for the attribute value range $r_i = [lb_i,ub_i]$ without any critical point and $max_j\{p(lb_i), p(ub_i), p(c_j)\}$ for r_i containing critical points $c_1, . . . , c_x$.

We can similarly support non-differentiable ranking functions by plugging in the function-specific routine for computing each cell's maximum predicate score possible. For example, such a routine for *min* is simply $p(ub_i)$.

3.4.4. Selectivity

Since executing top-k queries in distributed environments requires the retrieval of partial score objects corresponding to candidates for the final top-k result set, we want to try to minimize the number of irrelevant objects retrieved in an effort to reduce bandwidth. To achieve our goal, we rely heavily on the concept of selectivity, which has previously been discussed in the context of both traditional relational database queries and top-k queries [4,17]. Selectivity for a predicate basically refers to the number of objects that we expect to retrieve by evaluating that predicate.

Notice that, unlike relational queries that use Boolean predicates, the order in which fuzzy predicates are evaluated in a scoring function for a top-k query can significantly affect the number of objects retrieved overall [4]. Consider Example 4 with the following predicates p_1 and p_2 with four objects sorted from highest to lowest based on their normalized scores in the range [0:1] for each predicate. Our scoring function F in this example is $F = p_1 + p_2$ (i.e., *sum*). Also, suppose we know that the minimum overall score required to appear in the top-k results where k = 1 is $F(o) \geq 1.4$ for each object o. In this example, notice that our predicate evaluation schedule S could be either $S_1 = (p_1,p_2)$ or $S_2 = (p_2,p_1)$.

Example 4

p_1	p_2
o_3 : 0.7	o_3 : 0.7
o_2 : 0.6	o_1 : 0.3
o_1 : 0.5	o_2 : 0.2
o_4 : 0.4	o_4 : 0.1

$F(o)$
o_3 : 1.4
o_2 : 0.8
o_1 : 0.8
o_4 : 0.5

Remember that our goal is to reduce the number of irrelevant objects retrieved. In the case of S_1, after evaluating each object's score from p_1 and before evaluating each object's score from p_2, we realize that each object could potentially appear in the top-k results. The reason that each object is still a candidate is that each object's score under p_2 is unknown and therefore we must assume that each object receives the best possible score under p_2, which is 1.0 in this scenario. If we know the score for each object according to p_1 and assume that the score for each object under p_2 is 1.0, then potentially $F(o) \geq 1.4$ for every object o. Therefore, each object must still be considered a candidate and thus evaluated according to p_2. After evaluating each object according to p_2 in S_1, we eventually determine that only $F(o_3) \geq 1.4$. However, using similar reasoning for S_2, we see that o_3 emerges as the only possible candidate after evaluating p_2 first in S_2 before evaluating p_1. Therefore, considering the selectivity of p_2 relative to the selectivity of p_1, schedule S_2 has higher *aggregate selectivity*. Aggregate selectivity for fuzzy predicates, introduced in [4], is the summation of the number of partial score objects that we expect to retrieve after evaluating each predicate in a schedule or sub-schedule. Similar to [4], in order to reduce the costs of retrieving objects, we attempt to have the highest aggregate selectivity possible in our algorithms by determining an appropriate predicate evaluation schedule for each top-k query. However, rather than minimizing our per-object probing as done in [4], we instead attempt to minimize the depth of cells visited at each peer in an adapted, predicate-specific HistogramBloom from which partial score objects must be retrieved. Using the data synopses for the various predicates, which represent global information in a summarized format, we can determine the depth of cells that must be visited for each predicate at each peer. Hence, we refer to our algorithms as *InformedDepth*.

3.4.5. Vertical Data Decomposition

Now that we have explained how we adapt our summaries for user-defined predicates and scoring functions as well as the notion of aggregate selectivity, we can present our algorithm *InformedDepth* for the case of vertical data decomposition where each group stores all of the attribute value objects for a single attribute for all database objects. In such a group, each node stores a subset of the attribute value objects for a single attribute and the corresponding per-node summary represented as a HistogramBloom. In our algorithm, we assume that all the metadata from database objects has been extracted with the corresponding attribute value objects stored at the appropriate nodes. In addition to its own per-node summary, we assume that group leaders have already collected all of the per-node summaries from other group members to construct a per-group summary, also represented as a HistogramBloom (combining node-level summaries into group-level summaries is explained in Section 3.4.2).

Ideally, using the notion of aggregate selectivity, we want to find the optimal predicate evaluation schedule that is cost-minimal based on achieving the highest aggregate selectivity possible in order to minimize the number of irrelevant objects retrieved, which leads to bandwidth savings. Unfortunately, finding such cost-minimal scheduling is NP-hard as proven in [23]. Therefore, we develop a greedy algorithm for predicate evaluation scheduling where the next predicate that we choose to schedule at each step is always the predicate that provides the highest aggregate selectivity considering the preceding predicates scheduled thus far. Although the idea of greedy predicate scheduling based on aggregate selectivity was introduced in [4], our work is unique in that we determine this greedy schedule using only HistogramBloom summaries for each predicate rather than having the ability to probe on a per-object basis. Greedy predicate scheduling based only on data synopses, as is done in *InformedDepth*, is ideal for distributed environments where retrieving objects on a per-object basis is too expensive. Before going over the steps in detail, we should mention that *InformedDepth* assumes that each HistogramBloom cell for both per-node and per-group summaries uses Bloom filters with the same number of bits and identical hash functions. This allows us to perform intersection and union operations on our Bloom filters. Of course, individual cell memberships and cell ranges will differ from summary to summary.

Any node can issue a top-k query, which is executed by the following algorithm at any peer *P* wishing to do so. The *InformedDepth* algorithm is illustrated in Figures 5 and 6 with non-group leader peer P_{24} issuing query Q_{24} via its group leader peer G_{32}.

1. If *P* is not a group leader, *P* sends query *Q* to its group leader *G*. Otherwise, skip this step (Figure 5a).
2. *G* floods *Q* on the group-level DHT to the other remote group leaders (Figure 5b) .
3. *G* and other group leaders adapt predicate-oblivious summary *H* to create a predicate-specific summary *H'* (as described in Section 3.4.3) for the query *Q*.
4. *G* collects all of the adapted summaries, including its own (Figure 5b).
5. Using the adapted summaries, which correspond to predicates, the procedure *top(k,F,Hset)* is run by *G* to determine the top-k results where *k* are the desired number of results, *F* is the scoring function, and *Hset* is the set containing the adapted summaries. The procedure *top(k,F,Hset)* is briefly described below at a high-level. Detailed pseudocode for *top(k,F,Hset)* appears in the Appendix.
    a. Estimate lower bound top-k score by finding minimum depth *d* of HistogramBloom cells that must be visited in each adapted summary (where *d* is the same for all summaries) until the intersection of the union of cells visited in each summary indicates that at least *k* objects have been seen in each summary. The lower bound top-k estimate equals the scoring function *F* applied to the lower bound of the d^{th} cell from each summary.
    b. Based on aggregate selectivity and this lower bound top-k score, build a predicate evaluation schedule *S* by greedily choosing the predicate with highest selectivity at each step when a predicate is added to *S*. When each predicate *p* is chosen for the schedule, save the depth of cells that should be visited in that predicate as *p.depth*.
    c. For each *p* in *S*, retrieve the partial score objects only for those partial score objects stored in the top *p.depth* cells in the adapted summary for *p* (these scores are found amongst peers in the same group for the vertical case, so *p.depth* only has to be sent to one group) (Figure 6b).
    d. With the necessary partial score objects retrieved, compute the top-k result *TopK*.
    e. Send *TopK* to *P* (Figure 6a).

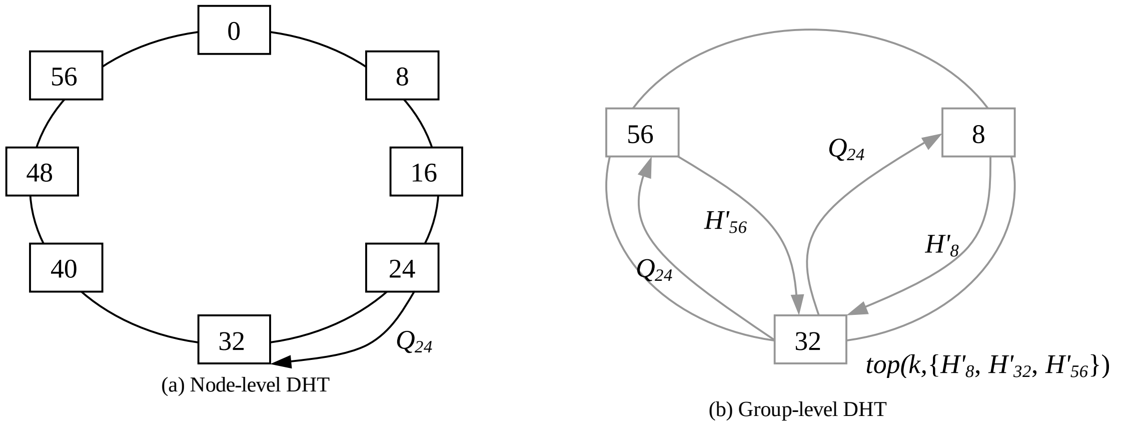
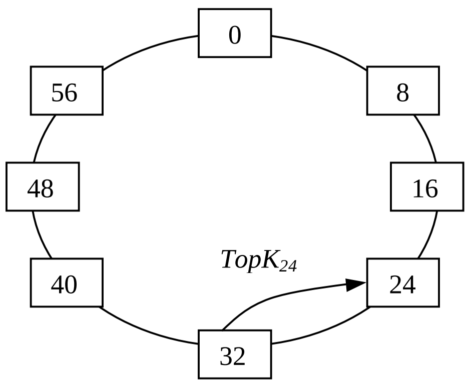
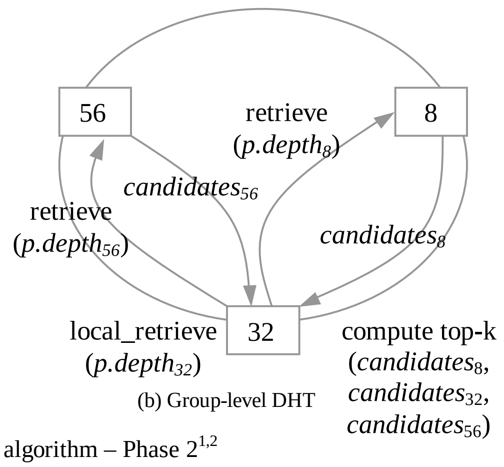


Figure 5. *InformedDepth* algorithm – Phase 1



(a) Node-level DHT


56
8
32
retrieve (p.depth8)
candidates56
retrieve (p.depth56)
candidates8
local_retrieve (p.depth32)
compute top-k (candidates8, candidates32, candidates56)

(b) Group-level DHT

Figure 6. *InformedDepth* algorithm – Phase 2[1,2]

3.4.6. Horizontal and Mixed Data Decomposition

Both the cases of horizontal and mixed data decompositions can easily be reduced to the case of vertical data decomposition. In horizontal and mixed data decompositions, each node might maintain one or more HistogramBlooms for one or more attributes rather than a single HistogramBloom for a single attribute. However, in addition to assuming that each HistogramBloom cell has a Bloom filter with the same number of bits and identical hash functions, we also assume that each HistogramBloom for a particular attribute is *consistent*. By consistent, we mean that the same number of cells must be used as well as the same cell ranges (i.e., lower bounds and upper bounds) for each corresponding cell. Although two HistogramBlooms for the same attribute must be consistent, two HistogramBlooms for two different attributes do not need to be consistent. We use the following algorithm for peer *P* issuing a top-k query in the cases of horizontal or mixed data decompositions.

1. If *P* is not a group leader, *P* sends query *Q* to its group leader *G*. Otherwise, skip this step.
2. *G* floods *Q* on the group-level DHT to the other remote group leaders.
3. *G* and other group leaders adapt each one of their predicate-oblivious summaries H_i to create a corresponding predicate-specific summary H_i' (as described in Section 3.4.3) for the query *Q*.
4. *G* collects all of the adapted summaries (including its own).
5. *G* collects these adapted summaries and combines the corresponding cells of each respective summary to build a single predicate-specific summary *H* for each attribute
6. *Hset*, which contains all of the predicate-specific summaries, can then be used to compute *top(k,F,Hset)* and proceed as in the vertical case. However, note that partial scores for a single predicate might be retrieved from several groups of peers rather than a single group of peers as is done in the vertical case. Therefore, *p.depth* from Section 3.4.5 might need to be sent to multiple groups rather than one group.

[1] Upon receiving retrieve($p.depth_x$), group leader G_x will collect the necessary candidates from the non-group leaders in its group to construct the set $candidates_x$. Group leader G_{24} will also construct candidates based on local_retrieve($p.depth_{24}$) in a similar fashion.
[2] By candidates, we mean the partial scores for all candidate objects that could potentially belong to the top-k result.

3.4.7 Algorithm Correctness

Our algorithm is provably correct if the exact size of the membership can be determined for all Bloom filters. Basically, using our summaries and greedy predicate scheduling algorithm, we identify all candidates that possibly could score above our lower bound top-k score, which is estimated in Step 5(a) from Section 3.4.5. We then evaluate the complete scores for all of these candidates to determine the top-k results. The only way that our algorithm might produce incorrect results is when database objects in the true top-k results are missed because we determined that they do not have potential scores higher than the lower bound top-k score estimated in Step 5(a). However, we prove by contradiction that no such candidates could be part of the true top-k result.

> **Proof.** Suppose some object *obj* in the true top-k result has an overall score less than our estimated lower bound top-k score. When we estimate our lower bound top-k score, we know that at least *k* candidates have appeared in the intersection of the union of cells from each HistogramBloom up to some depth *d*. Since we apply the scoring function *F* to the lower bound of the d^{th} cell from each HistogramBloom to compute the lower bound top-k score, we know that at least *k* candidates have overall scores above the lower bound top-k score. This contradicts *obj* being in the true top-k result because at least *k* elements must have higher scores.

Since retrieving full membership information is prohibitively expensive, we use Bloom filters to compactly represent this membership information. Bloom filters introduce the possibility of false positives and frequency count errors causing our algorithm to be correct with a certain probability. Specifically, when estimating the lower bound top-k score in Step 5(a) of the algorithm, if we overestimate that we have seen at least *k* elements when we have actually seen less than *k* elements, our lower bound top-k score might be too high (i.e., overly optimistic). If the lower bound top-k score is too high, we might exclude some candidates of the true top-k result from being considered (i.e., candidates in the true top-k results are missed). Underestimation errors might affect performance by being overly pessimistic and thus retrieving additional candidates, but they will not affect the correctness of the algorithm. Of course, since Bloom filters have tunable parameters, the number of hash functions and number of bits can be chosen in such a way that the probability of false positives or overestimation errors is very low. Since user-defined predicates are fuzzy, many peer-to-peer applications can tolerate a tunable amount of error. In our simulations, we were able to determine the correct top-k results in each simulation run.

4. PERFORMANCE EVALUATION

We evaluated *InformedDepth* by building on Examples 1 and 2. In our simulation scenarios, we have a logical relation *Video_Meta_Data* containing metadata about video files stored in a peer-to-peer CDN. The schema for the logical relation *Video_Meta_Data* is shown below.

***Video_Meta_Data*(*fileId*, *numAccesses*, *avgDuration*, *runtime*, *bitrate*, *size*)**

In *Video_Meta_Data*, *fileId* is the key for some video file that uniquely identifies it. *numAccesses* is the number of times that the file has been streamed to clients of the CDN. *avgDuration* is the average viewing time of the video file by clients of the CDN. *runtime* is the length of the video file in minutes. *bitrate* indicates the encoding bit rate for the video file and *size* indicates the size of the file in bytes. For our predicates, we normalize the attribute values by dividing each attribute value by the maximum value for that particular attribute in the relation *Video_Meta_Data*. We simulated a site administrator issuing the following query (shown below) where $attr_i$ indicates the normalized value of the i^{th} attribute (e.g., $attr_1$ indicates the normalized value for *numAccesses*) and w_i indicates the weight of that attribute's value to the overall score. Each predicate $p_i = w_i(attr_i)$ implicitly indicates the importance of each attribute to that particular query.

select *fileId*
from *Video_Meta_Data*
order by *scoring function* $F = w_1(attr_1) + w_2(attr_2) + w_3(attr_3) + w_4(attr_4) + w_5(attr_5)$
stop after k

Our logical relation *Video_Meta_Data* contained 1000 database objects with 975 of those objects derived from a synthetic streaming media workload generator and 25 of those objects created manually. Medisyn, the synthetic streaming media workload generator used, can create synthetic request logs for media servers based on statistical properties of request logs from real media servers [24]. We use this media server workload to model the client access patterns (along with video file metadata) in our peer-to-peer CDN. We also added 25 manually created video files that would be potential best candidates for our query. The goal of our simulations was to evaluate the performance of *InformedDepth* according to its ability to retrieve these best candidates in a database containing hundreds of other synthetically generated candidates. The five different types of attribute value objects were assigned to five different groups (i.e., vertical partitioning where each attribute is assigned to a different group). Each group has 10 peers with one of those peers assigned to be the group leader. Each HistogramBloom summary had five cells with Bloom filters for each cell using three hash functions and 2048 bits. Of course, cell ranges for each HistogramBloom differed based on the predicate.

We evaluated *InformedDepth* by comparing it to a version of *InformedDepth* with random predicate scheduling (as opposed to the greedy predicate scheduling actually used in *InformedDepth*) and the basic version of TPUT, which is described below. Since the basic version of TPUT, as described below, is limited in its support of user-defined predicates and scoring functions, we used equal weights in our scoring function F for a fair comparison to TPUT.

4.1. Three-Phase Uniform Threshold (TPUT)

The TPUT algorithm is another top-k algorithm designed specifically for distributed environments [11]. It considers only vertical partitioning. Our running example of a peer-to-peer CDN streaming videos is actually motivated by the application scenario presented by the authors of TPUT, which involves running simple top-k queries over a CDN containing cached Web documents. As its name suggests, TPUT consists of three phases, which are listed below (taken from [11]).

1. *Establish a lower bound on the top-k result set*. First, retrieve the top-k partial scores from each peer. Based on these partial scores, compute the partial sums of the objects. Set the estimated lower bound τ_1 to the k^{th} highest partial sum.
2. *Prune away ineligible objects.* Set threshold $T = (\tau_1 / m)$ where m is the number of predicates. Retrieve all partial scores whose values are greater than or equal to T. At this point, all possible members of the top-k result set have had at least one partial score retrieved, as proven in [11]. With these additional partial scores, re-compute the partial sums of the objects to get a refined lower bound τ_2, which is equal to the k^{th} highest partial sum. Also, compute upper bounds for each object assuming the maximum possible score for all unknown partial scores. Eliminate all objects whose upper bounds are less than τ_2. Assign the remaining objects to candidate set S.
3. *Identify top-k objects*. Retrieve all unknown partial scores for each candidate in S from all peers. Compute the complete overall scores for the candidates in S and return the top-k candidates based on overall scores.

4.2. Bandwidth Consumption

To show the benefit gained by greedy predicate scheduling in *InformedDepth*, we show the number of objects retrieved when greedy predicate scheduling is used versus the number of objects retrieved when the predicates are scheduled randomly. If greedy predicate scheduling (as described in Sections 3.4.4 and 3.4.5) is indeed effective, then *InformedDepth* should retrieve significantly fewer objects compared to when predicates are scheduled at random. Figure 7 shows our results for $k = 1,\ldots,10$.

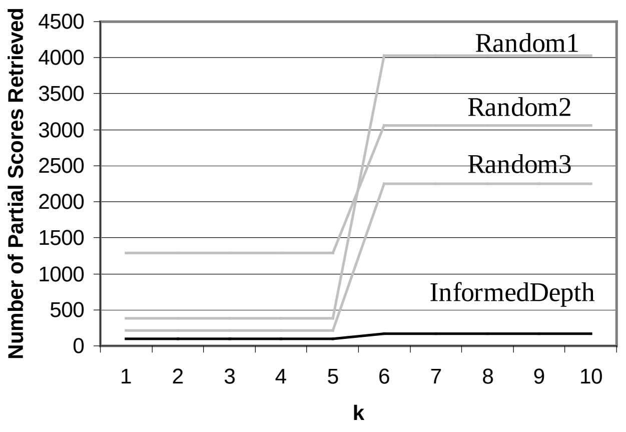


Figure 7. Bandwidth consumption (*InformedDepth* versus Random Predicate Scheduling)

As shown above, using *InformedDepth* significantly reduces the number of candidate objects retrieved compared to the three random schedules shown in the figure. For k ≥ 6, we see an order of magnitude reduction when using greedy predicate scheduling compared to random predicate scheduling. We simulated a total of 10 random schedules (with *InformedDepth* consuming significantly less bandwidth than each one of those random schedules), but decided to include only three random schedules in Figure 7 for clarity. The random schedules shown in Figure 7 range from best to worst as far as the performance of the ten different random schedules simulated.

In Figure 8, we compare the number of objects retrieved using our algorithm compared with the basic version of TPUT described in Section 4.1.

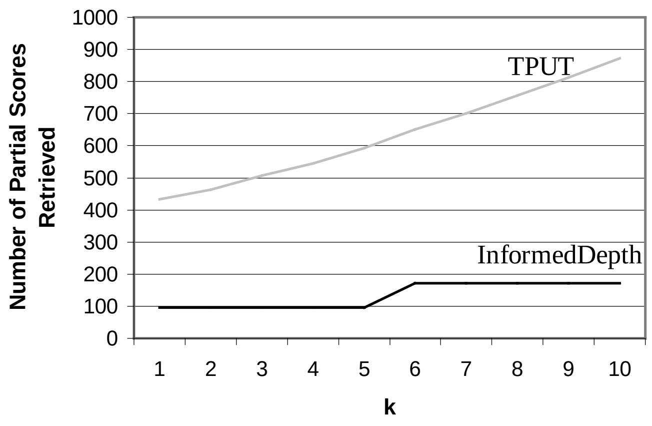


Figure 8. Bandwidth consumption (*InformedDepth* versus TPUT)

As shown above, using *InformedDepth* also reduces the number of candidate objects retrieved compared to TPUT. TPUT performs better than random predicate scheduling, but *InformedDepth* with greedy predicate scheduling still retrieves fewer objects than TPUT. More specifically, between 77% and 80% fewer objects are retrieved using *InformedDepth* with greedy predicate scheduling compared to TPUT.

4.3. Delay

Based on the algorithm description for *InformedDepth*, it is clear that determining the top-k results takes two message rounds (i.e., two round-trips). The adapted summaries are retrieved during the first round-trip and all of the partial scores for all candidate objects are retrieved during the second round-trip. As shown in Section 4.1, TPUT requires three round-trips with each round-trip corresponding to one phase of the algorithm. In scenarios with potential network bottlenecks, this extra round-trip might significantly delay results. Part of our future work involves simulating such scenarios and also running both *InformedDepth* and TPUT on the PlanetLab network testbed [12].

4.4. Correctness of Top-k Results

Since the cell membership of each cell in a HistogramBloom is represented by a Bloom filter, the possibility of false positives and overestimation errors is present as discussed in Section 3.4.7. Therefore, we checked our results to make sure that each simulation run of *InformedDepth* would output the correct top-k results. Not only did *InformedDepth* output the correct top-k results in the correct order during each simulation, but *InformedDepth* also computed the correct top-k score each time.

5. CONCLUSION

InformedDepth can flexibly and efficiently calculate the top-k results of a query executed in a peer-to-peer network. Unlike previous approaches that only focus on a single type of data partitioning, *InformedDepth* includes algorithms for handling different types of data decompositions. To execute top-k queries effectively, *InformedDepth* uses data synopses to make greedy predicate scheduling feasible in distributed peer-to-peer environments. Our simulation results indicate that *InformedDepth* exhibits good performance in terms of bandwith consumed and correctness compared to current top-k query processing algorithms for distributed environments. Our simple analysis also demonstrates that *InformedDepth* has fewer message rounds than TPUT. In the future, we plan to fully implement *InformedDepth* and run it on the PlanetLab testbed to get actual delay measurements.

REFERENCES

[1] R. Fagin. Combining fuzzy information from multiple systems. *Proceedings of the 15th ACM Symposium on Principles of Database Systems (PODS)*, 1996.

[2] R. Fagin, Amnon Lote, and Moni Naor. Optimal aggregation algorithms for middleware. *Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS)*, 2001.
[3] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.
[4] K. C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.
[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2001.
[6] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proceedings of the 3rd International Middleware Conference (Middleware)*, 2001.
[7] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, Vol. 22, No. 1, January 2004.
[8] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.
[9] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh. SplitStream: high-bandwidth multicast in a cooperative environment. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.
[11] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.
[12] PlanetLab. *http://www.planet-lab.org*.
[13] S. Michel, P. Triantafillou, and G.Weikum. KLEE: a framework for distributed top-k query algorithms. *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*, 2005.
[14] W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.
[15] W. Conner, K. Nahrstedt, and I. Gupta. Preventing DoS attacks in peer-to-peer media streaming systems. *Proceedings of the 13th Conference on Multimedia Computing and Networking (MMCN)*, 2006.
[16] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. *Proceedings of 4th International Middleware Conference* (*Middleware*), 2003.
[17] J.M. Hellerstein and M. Stonebreaker. Predicate migration: optimizing queries with expensive predicates. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1993.
[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol. 13, Issue 7, July 1970.
[19] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: a scalable wide-area web cache sharing protocol. *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 1998.
[20] Los Alamos National Laboratory. *http://www.lanl.gov*.
[21] DOE Joint Genome Institute. *http://www.jgi.doe.gov*.
[22] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation* (*NSDI*), 2004.
[23] K. C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. Technical Report UIUCDCS-R-2001-2258, University of Illinois, December 2001.
[24] W. Tang, Y. Fu, L. Cherkasova, A. Vahdat. MediSyn: synthetic streaming media service workload generator. *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Audio and Video* (*NOSSDAV*), 2003.

APPENDIX

```
**Top-k-results top(int k, ScoringFunction F, HistogramBloomSet Hset)**{
    PredicateSet $P$ = {};
    for each HistogramBloom $hb$ in $Hset$
            add Predicate $p$ to $P$ with $p.hb = hb$;
    Schedule $S$ = findGreedySchedule($k$,$F$,$P$);
    for each Predicate $p$ in $S$
            Retrieve partial scores from remote peers
            corresponding to $p.hb$'s cells up to $p.depth$;
    Use $F$ to compute overall scores from partial scores retrieved;
    return top-k results based on overall scores;
}

**Schedule findGreedySchedule(int k, ScoringFunction F, PredicateSet P)**{
    float $lowerBoundTopkScore$ = estimateLowerBoundTopkScore($k$,$F$,$P$);
    Schedule $S$ = {};
    while $P$ is not empty
            Choose $p$ from $P$ where
            selectivity($lowerBoundTopkScore$, $F$, $S \cup p$, $P - p$)
            is minimal;
            Set $p.depth$ to the second integer returned by
            selectivity( );
            Add $p$ to $S$;
            Remove $p$ from $P$;
    return $S$;
}

**float estimateLowerBoundTopkScore(int k, ScoringFunction F, PredicateSet P)**{
    BloomFilterSet $B$ = {};
    for $i$ = 1 to $m$ where $m$ is the number of predicates in $P$
            Create BloomFilter $cumulative_i$ with all bits initialized
            to 0;
            Add $cumulative_i$ to $B$;
    for $i$ = 1 to $numcells$ where $numcells$ are the number of cells in
    each predicate in $P$
            for $j$ = 1 to $m$ where $m$ is the number of predicates in
            $P$
                    Predicate $p$ = $j$th predicate in $P$;
                    HistogramBloomCell $c_i$ = $i$th cell from
                    $p.hb$;
                    BloomFilter $bf_i$ = BloomFilter for $c_i$;
                    $cumulative_j$ = $j$th BloomFilter from $B$;
                    Update $cumulative_j = cumulative_j \cup bf_i$;
            BloomFilter $intersection_i = \bigcap_j cumulative_j$ for all $j$
            where $cumulative_j$ is $j$th BloomFilter from $B$
            if estimated number of members in $intersection_i \geq k$)
                    return $F$ applied to lower bound of $i$th cell
                    from each predicate $p$ in $P$;
    return $F$ applied to lower bound of last cell from each predicate $p$
    in $P$;
}

**struct Predicate**{
    int $depth$;
    HistogramBloom $hb$;
}

**struct Candidate**{
    int $depth$;
    float $upperbound$;
    BloomFilter $bf$;
}

**(int,int) selectivity(float lowerBoundTopKScore, ScoringFunction F, Schedule S, PredicateSet UnknownPset)**{
    CandidateSet $C$ = findCandidates($F$,$S$,$UnknownPset$);
    Remove all candidates $c$ from $C$ whose maximum upper bound
    scores are less than $lowerBoundTopkScore$;
    Return the number of bits set to 1 in all Bloom filters of all
    candidates in $C$ and maximum $c.depth$ among all candidates $c$;
}

**CandidateSet findCandidates(ScoringFunction F, Schedule S, PredicateSet UnknownPset)**{[3]
    CandidateSet $C$ = {};
    1.  Find all possible combinations of cells (i.e., cross product
        of cells) from the predicates $p_1,\ldots,p_n$ that comprise all
        subsets of $S$ and add a corresponding Candidate to $C$;
    2.  For each Candidate $c$ in $C$, figure out the maximum depth
        of a cell in the combination and set $c.depth$ to this
        maximum depth;
    3.  For each Candidate $c$ in $C$, figure out the upperbound score
        for $c$ given $F$, $UnknownPset$, and its constituent cells;
            a.  Assume maximum possible predicate scores
                from predicates in $UnknownPset$;
            b.  Assume cell upperbound scores for predicates in
                $S$ where it has a score and the lowest possible
                score otherwise (i.e., unseen candidate);
            c.  Set $c.upperbound$ to refined upperbound
                estimate;
    4.  For each Candidate $c$ in $C$, set $c.bf$ to the intersection of all
        the Bloom filters for constituent cells;
    return $C$;
}
```

[3] The algorithm, as described, is very inefficient. The actual implementation is more efficient, but also more difficult to explain.