Answering Similarity Queries in Peer-to-Peer Networks ⋆

Panos Kalnis * Wee Siong Ng Beng Chin Ooi Kian-Lee Tan

Department of Computer Science
National University of Singapore
3 Science Drive 2, 117543 Singapore

Abstract

A variety of Peer-to-Peer (P2P) systems for sharing digital information are currently available and most of them perform searching by exact key matching. In this paper we focus on *similarity* searching and describe *FuzzyPeer*, a generic broadcast-based P2P system which supports a wide range of fuzzy queries. As a case study we present an image retrieval application implemented on top of FuzzyPeer. Users provide sample images whose sets of features are propagated through the peers. The answer consists of the top-k most similar images within the query horizon. In our system the participation of peers is ad-hoc and dynamic, their functionality is symmetric and there is no centralized index.

In order to avoid flooding the network with messages, we develop a technique that takes advantage of the fuzzy nature of the queries. Specifically, some queries are "frozen" inside the network, and are satisfied by the streaming results of similar queries that are already running. We describe several optimization techniques for single and multiple-attribute queries, and study their tradeoffs. We evaluate the performance of our algorithms by a prototype implementation on our P2P platform and a simulated large-scale network. Our results suggest that by reusing the existing streams, the scalability of the system improves both in terms of number of nodes and query throughput.

Key words: Peer-to-Peer, Similarity Searching, Gnutella

⋆ A two-page poster version of this paper will appear in the Thirteenth International World Wide Web Conference, New York, May 2004.

* Corresponding author: Panos Kalnis, Tel: (+65) 68742911, Fax: (+65) 67794580
Email addresses: kalnis@comp.nus.edu.sg (Panos Kalnis), ngws@comp.nus.edu.sg (Wee Siong Ng), ooibc@comp.nus.edu.sg (Beng Chin Ooi), tankl@comp.nus.edu.sg (Kian-Lee Tan).

1 Introduction

Peer-to-Peer (P2P) technology has recently attracted a lot of attention, since it allows the implementation of large distributed repositories of digital information. In a P2P system numerous nodes of equal roles are connected through an arbitrary network and exchange data or services directly with each other. Many P2P systems follow a semi-centralized (or hybrid [23]) architecture (e.g., Napster [3]), where queries are posed to a centralized index although the data and services are distributed. Despite their advantages, hybrid systems inherit the drawbacks of centralized architectures.

As an alternative, several fully distributed (or pure P2P) systems have been proposed. In this case there is no centralized catalogue or functionality; instead, peers are individually contacted and return the results they contain. There are two major subcategories of pure P2P systems: (i) Hash-based systems (e.g. Chord [20], CAN [17] and Pastry [18]), which assign a unique key to each file and forward queries to specific nodes based on a hash function. Although they guarantee the location of content within a bounded number of hops, they require tight control of the data placement and the topology of the network. (ii) Broadcast-based systems (e.g., Gnutella [1]), which use message-flooding to propagate queries. There is no specific destination; hence every neighbor peer is contacted and forwards the message to its own neighbors until the message's lifetime expires. Such systems have been successfully employed in practice to form large-scale ad-hoc networks. Here we assume a pure P2P, broadcast-based architecture.

Most existing systems support only boolean query evaluation. Each file is characterized by its metadata (i.e., a set of keywords) and queries ask for combinations of keywords. Consider for instance a music sharing system. Users ask for a song title, or a combination of an artist and album name. Such queries can be unambiguously evaluated as "found" or "not found" by searching the metadata for matching keywords.

In this paper we investigate a different problem: Users ask fuzzy queries like "find the top-k images which are similar to a given sample". Such queries are common in image retrieval systems (e.g., QBIC [14]) because it is difficult for humans to express precisely an image's content in keywords. Since there is no centralized index, each peer within the query horizon is contacted and returns k results (i.e., the top-k local images) to the initiator, which, in turn, computes the global result. Unfortunately, the extremely low selectivity of such queries floods the network with useless messages. An alternative solution is to set a threshold similarity and accept answers only above this value. The issue in this case is how to select the query-dependant threshold given that the interpretation of an image depends on the user's perception; if the threshold

is too low, there is no benefit in terms of transmitted messages while, if it is too high, there is the risk of wasting the query messages without locating any satisfactory answer. Moreover, this method would not reduce the number of query messages which grows exponentially with the number of hops.

Observe, however, that due to the fuzzy nature of the queries, the answers are always approximations. As a result, if two queries are similar, the top-k answers for the first one may contain (with high probability) some of the answers for the second query. In addition, in P2P networks each peer can examine the messages that pass through it. Motivated by these observations, we developed *FuzzyPeer*, a generic P2P system which supports similarity queries. In FuzzyPeer some of the queries are paused (i.e., they are not propagated further) and stay resident inside a set of peers. We use the term *frozen* for such queries. The frozen queries are answered by the stream of results that passes through the peers, and was initiated by the remaining running queries. By carefully selecting the set of queries that will be answered from the streaming data, the quality of the results and the response time remain at acceptable levels even when the system is overloaded. Additionally, the number of messages drops considerably, thus improving the scalability and the throughput of the network. Moreover, our optimization algorithms do not pose any overhead due to synchronization messages.

Although throughout this paper we focus on image retrieval, our methods are applicable to other domains where similarity searching is performed in P2P networks; as an example, consider the case of text retrieval. Moreover, the network topology does not need to be flat. For instance, given a two-level super-peer organization (e.g., Morpheus [2]) we can apply the same techniques at the upper level which contains the index of its clients, rendering the entire system more scalable.

The rest of the paper is organized as follows: Section 2 contains some essential background. In Section 3 we describe the architecture of our prototype, while in Section 4 we analyze our proposed techniques. Section 5 presents an extensive experimental evaluation of the system and Section 6 summarizes the paper.

2 Related Work

Research in the P2P area was triggered by the apparent success of systems like Napster [3], Gnutella [1] and Freenet [6]. Napster is a hybrid system since it maintains a centralized index which is used for searching; a detailed evaluation of various hybrid architectures can be found in Ref. [23]. Gnutella, on the other hand, is a pure P2P system and performs searching by Breadth-First-Traversal

(*BFT*) of the nodes around the initiator peer. Each peer that receives a query propagates it to all of its neighbors up to a maximum of d hops. The advantage of BFT is that by exploring a significant part of the network, it increases the probability of satisfying the query. The disadvantage is the overloading of the network with unnecessary messages. Alternatively, Freenet uses Depth-First-Traversal (*DFT*) up to depth d. Each node forwards the query to a single neighbor and waits for a response before contacting the next one. DFT saves messages, but increases the response time (i.e., exponential to d in the worst case).

Yang and Garcia-Molina [24] observed that the Gnutella protocol could be modified in order to reduce the number of nodes that receive a query, without compromising the quality of the results. They proposed three techniques: (i) *Iterative Deeping* [1], where multiple BFTs are initiated with successively larger depth d_i. (ii) *Directed BFT*, where queries are propagated only to a beneficial subset of the neighbors of each node. This method is extended in [7] by maintaining summarized information of the neighbors' contents. A similar approach [13], [10] reconfigures dynamically the structure of the network, in order to create clusters of compatible nodes. (iii) *Local Indices*, where each node maintains an index over the data of all peers within r hops of itself. An analogous technique is used in Ref. [25], the only difference being that local indices are kept only in a subset of powerful nodes called super-peers. All three techniques are orthogonal to our methods and can be employed in our system in order to further reduce the query cost.

Recently, there have been efforts to support complex queries in Distributed Hash Table systems. In Ref. [19] the authors describe a distributed cache built on top of CAN [17]; the system, however, is limited to range queries over a globally known schema. Closer to our work is the pSearch system [21] which supports similarity search for text files. Documents are characterized by feature vectors which are stored in a CAN network as multi-dimensional points. CAN, however, cannot handle efficiently high dimensional spaces; therefore, pSearch uses several heuristics to decrease the dimensionality. Although hash-based systems minimize bandwidth consumption during query processing, they require tight control on the contents of each peer and exhibit considerable overhead when nodes enter and leave the network frequently. Therefore, they are more suitable for intranets, where peers belong to the same organization and are relatively stable.

Our work is also related to the optimization of continuous queries over streaming data. Similar to the NiagaraCQ [5] and the CACQ [12] system, our goal is to minimize the total cost by sharing the work of multiple concurrent queries. These systems assume that there is an existing infinite data stream and all

[1] In Ref. [24] the term "frozen queries" has different meaning.

the queries are processed at a centralized location. In our case the streams are results from queries which have already been propagated. Therefore, we need to optimize the number of streams we initiate, taking into consideration that the lifetime of the queries is limited therefore the streams are finite. Moreover, the optimization process is distributed over all peers within the query horizon.

In addition to single-feature searching, FuzzyPeer supports multiple feature queries [8]. The intuition of processing such queries is that the answers are constructed by combining sequential access to sorted single-feature streams with random access to the original sources [4]. Here we also employ the idea of random accesses; our problem is different, however, since we cannot assume sorted data streams.

Since we study the special case of image retrieval, this work is also relevant to centralized multimedia database systems like QBIC [14] and Photobook [16], which use feature vectors to represent images. Here, we employ the Daubechies' wavelet feature vector [22], a compact representation of images which is suitable for querying by example. Note that the Daubechies' transformation was chosen for illustration purposes only. Other representations may be more appropriate in different applications and can be easily incorporated in our system.

3 System Description

Figure 1 depicts a typical FuzzyPeer network. It consists of 8 peers which are connected through a set of links. Each link represents an active TCP/IP connection and is independent of the physical network layer's structure. Connections are symmetric, meaning that if a peer P_i is a neighbor of P_j then P_j is also a neighbor of P_i. Ideally, each peer P should be connected to all others since this would maximize its search space. However, there is a tradeoff because connections consume system resources and also cause more messages to be processed by P. Since the system is heterogeneous both in terms of bandwidth and processing power, powerful peers with fast links are typically connected to more neighbors. This is common in most P2P systems. For example, Gnutella implementations allow up to 4 neighbors for peers with slow connections, while powerful peers may have tens of neighbors.

Participation in the network is dynamic; each peer can join or leave the system at any time. When a peer enters the network, it contacts a location independent global name lookup (*LIGLO*) server [13] to get a set of potential neighbors; then it employs a Gnutella-like protocol [1] to connect. Except from LIGLO servers, the system is fully distributed. Furthermore, LIGLO servers are not involved in the query processing and can be completely eliminated if

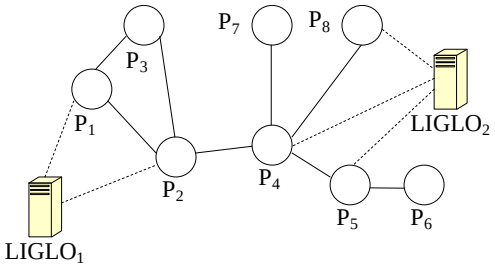


Fig. 1. A Typical FuzzyPeer network

the set of initial neighbors can be otherwise determined; for instance, peers on the same segment of a LAN may connect to each other.

Let the user of P_1 ask a query q: "find the top-10 images which are similar to a given sketch". P_1 will broadcast q to P_2 and P_3. The receiving nodes will search their databases and return the ids of the top-10 most similar images together with a similarity measure to P_1. At the same time they will broadcast q to their neighbors. For example, P_2 will send q to P_3 (which will reject the duplicate message) and P_4. Notice that P_4 will return the results through P_2. Queries can propagate for up to a maximum number of hops d. Assuming that $d = 3$, the query cannot reach P_6. P_1 waits for up to $MaxWaitTime$; during this interval it receives the answers and continuously refines the result. After $MaxWaitTime$ expires, any answer message that reaches P_1 is rejected.

Assume now that soon after P_1, P_3 also submits a query q' which is similar to q. In a traditional P2P system q' propagates through P_2 the same way as q. q' causes messages to pass through P_2 almost simultaneously with the messages generated by q. Therefore, P_2 is overloaded and all messages are delayed. If the delay is long enough, $MaxWaitTime$ expires causing q and q' to terminate before they receive enough useful results. Notice, however, that we can do better: When q passes through P_2 it initiates an answer stream $Stream_q$. All the answers from P_4, P_5, P_7 and P_8 will go through $Stream_q$. When q' reaches P_2 the system can identify that q and q' are similar, so instead of been propagated, q' will freeze inside P_2 and will be attached to $Stream_q$. P_2 will afterwards duplicate and sent to P_3 all answers that reach $Stream_q$. The intuition is that since q and q' are similar, their answers are also similar and it is preferable to get some approximate answer than not getting any answer at all. The rest of the paper presents the freezing technique in details.

3.1 Prototype Implementation

The low level network functionality of FuzzyPeer, such as connecting to other nodes, message handling, etc., is provided by BestPeer [13], our Java based generic P2P platform. A part of BestPeer also provides the LIGLO function-

ality. All the FuzzyPeer-specific code resides in the Query Processing Applet. This module coordinates the entire system and connects to the local database. The details of the database are irrelevant to FuzzyPeer. The only requirement is to support a *Top-K* operator. It is also desirable to provide a cost estimation function. In our case study, the database was flat files. It consisted of the original high-resolution images together with precalculated feature vectors. We used Daubechies' wavelets [22] to represent the visual features of the images. In our experiments, the number of images in each peer was relatively small, allowing us to keep all the feature vectors in memory and find the top-k queries by performing sequential search. In practical situations, where the image database is expected to be larger, we can employ a high-dimensional index for k-Nearest-Neighbor search, like Ref. [26].

Note that for the rest of the paper we only consider the optimization of the searching process. We assume that downloading the image is performed outside the searching network (i.e., as a separate http connection).

4 Query Processing

In this section we analyze our freezing techniques. Although we employ the Euclidian distance of image vectors as a similarity metric, the translation to other domains is straightforward.

Users pose queries by means of a sample image img_{user}. We apply a Daubechies' wavelet transformation $DT(img_{user})$ on the sample image and produce an m-D feature vector $f_1, \ldots, f_m$. This vector is the query q. The size of q is typically much smaller than the size of the image. The similarity $S(q, img)$ between a query q and an image img is defined as the Euclidian distance between the vector q and $DT(img)$. In the same way, we define the similarity $S(q, q')$ between two queries q and q' to be the Euclidian distance of the corresponding vectors. Obviously, when $S(q, q') = 0$ the vectors are identical.

When a peer receives a query q, it computes the k most similar images $img_1, \ldots, img_k$ from the local database, and returns a set of k pairs $(id_i, S(q, img_i))$, $i = 1..k$. Note that the answers do not contain the feature vectors of the images, but only the image id and the similarity measure. This is done in order to reduce the size of the answer messages, since in some applications each feature vector may be several KBytes.

In a traditional broadcast-based P2P system, when a query q is initiated at peer P, it is propagated through all its neighbors until a maximum number of hops d is reached. The peers that received the query, send their answer back to P via the reverse path. We call this the *Non-Freezing* algorithm (nf).

When a query q is propagated through a peer P, it creates an answer stream $Stream_q$. All the subsequent answers for q will go through $Stream_q$. If P is the peer which initiated q, the answers that arrive in $Stream_q$ are forwarded to the User Interface module; else they are propagated to the previous peer (i.e. the peer from where the query arrived). Obviously, there can be multiple streams simultaneously active at P. For every stream, P maintains a data structure containing the feature vector q together with various statistics.

There are several sources of delays in a message's path, including the network cost and the processing time. To facilitate our study, we use a simplified [2] model (Figure 2) which encapsulates all possible costs: Each peer P has a Processing Unit with a FIFO queue attached to it. All incoming messages $M_0, M_1, M_2, \ldots$ enter the queue. When the processing unit is ready, it removes the message M_0 from the head of the queue and processes it for time $T(M_0, P)$. After processing is over, the message is transmitted to the next peer. The total time a message M_i spends at P is:

$$T_{total}(M_i, P) = \sum_{j=0}^{i-1} T(M_j, P) + T(M_i, P) \quad (1)$$

where the first factor of the equation is the waiting time in the queue and the second factor is the actual processing time.

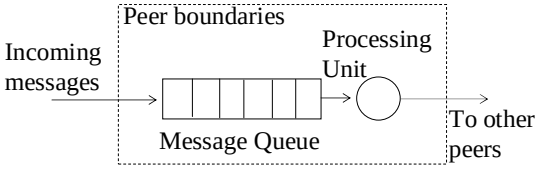


Fig. 2. Message propagation model

For a given number of on-line peers, assume that the query rate is low. Then the queues in all peers are empty, and from Equation 1, it follows that the only delay of a query message is its own processing time. However, when the query rate increases, the message queues become longer; therefore the delays for the messages also increase due to the queue waiting time. Recall that users abort the queries after $MaxWaitTime$. If the delays are long, there is not enough time for the query messages to reach many nodes before $MaxWaitTime$ expires. The number of answers that arrive at the initiating node decreases rapidly; therefore, the probability of obtaining useful answers (i.e. the precision of the results) also drops. This resembles the thrashing effect in time-sharing systems. The freezing algorithm we describe below, attempts to minimize the problem by decreasing the number of concurrent messages in the system.

[2] This abstraction is only used to simplify the presentation of the algorithms; our experimental results consider both the network congestion and the processing time in the nodes.

4.1 Static Query Freezing (SQF)

The intuition behind the Static Query Freezing algorithm is simple: some queries are frozen (i.e. paused) inside the system, in order to reduce the total workload. The result is that the waiting time in the queues decreases for the remaining running queries, so they can retrieve enough answers before $MaxWaitTime$ expires. The frozen queries attach to the streams of similar running queries and receive the same results. There are several benefits of this approach: (i) Thrashing is avoided (if enough queries are frozen). Instead of not answering any query at all, with SQF a considerable percentage of the queries can locate accurate answers. (ii) Excess queries are frozen instead of aborted. Since all answers are approximations, there is a high probability for a frozen query to receive accurate results if it attaches to a similar stream. This is different from other systems (e.g. web search engines) where the probability of finding a concurrent similar query is low. In such systems queries ask for certain keywords and run in the server for a few msec, while in P2P systems queries run for around 3 orders of magnitude more time (i.e., 100's of sec) (iii) Even if the results for the frozen queries are not accurate, users can utilize them to refine their original query.

Algorithm 1 Static Query Freezing

```
On UserQuery(q)
    With probability p_f set q.frozen = true, q.f_hops = h_f
    Initiate an answer stream A_q
    Broadcast q
On QueryReceived(q) // query received from neighbor
    q.traveled_hops++
    If q.frozen==true and q.f_hops == q.traveled_hops then
        Freeze q
        Attach q to a beneficial answer stream, if such stream exists
    Else
        if q.frozen == false then calculate answer and send it back
        if q.traveled_hops < max_Hops then broadcast q
On ResultReceived(r_q) // result received from neighbor
    For each query q' attached to q do
        If there is no cycle due to frozen queries then
            Duplicate r_q to generate r_q'
            Propagate r_q' backwards as a result for q'
    If current peer is the initiator of q then
        Add r_q to answer stream A_q
    Else propagate r_q backwards
```

Algorithm 1 presents SQF which takes two parameters: the probability p_f to freeze a query and the number of hops f_{hops} that q must travel before it freezes. The initiator peer decides with probability p_f if the new query is going to freeze. Then q is propagated as usual. Assume that q was selected to freeze and after f_{hops} reaches peer P'. SQF pauses q (i.e., q will not propagate further through that path), checks all the active streams at P' and attaches q to the most beneficial one. If no stream exists at P', q is just paused. Observe that q will freeze in all peers which are f_{hops} hops away from P. Also notice that the Non-Freezing algorithm (nf) is a special case of SQF where $p_f = 0$.

When an answer comes for a stream, SQF searches whether there are any attached queries to it. For every attached query q, a duplicate answer is generated and returned to the query initiator. Notice that since the answer messages do not contain any feature vectors, we cannot perform any filtering for the attached queries.

The freezing technique has the additional benefit of increasing the query horizon of the frozen queries. Consider again the example of Figure 1 assuming that there is no link between P_2 and P_3. P_1 initiates a query q which propagates to all nodes except P_6 (recall that the maximum number of hops $d = 3$). P_3 also initiates q' which, if not frozen, will reach only P_1, P_2 and P_4. On the other hand, if q' freezes in P_1 and attaches to q, the answers from P_5, P_7 and P_8 will also be forwarded to P_3. Therefore, the probability of locating an accurate result for q' increases.

The method of selecting a beneficial stream needs further clarification. Each stream st has a lifetime lt which is the same as the expiration time of the query that created it. A query queue will benefit by attaching to st only if there is enough time left for many results to propagate through it. Therefore, recent streams are more beneficial. Also the benefit is proportional to the similarity of q with the query that initiated st. Notice that we can calculate this similarity, since we have the feature vectors for both queries. In our implementation we use a combined benefit, giving more weight to the similarity criterion.

In the experimental section we show that SQF improves significantly the throughput and the scalability of the system. Its applicability however is limited in practice, since the user must provide for each query an appropriate set of parameters for the current condition of the network. Below we describe an alternative freezing algorithm which adapts dynamically to the workload of the system.

4.2 Adaptive Query Freezing (AQF)

The drawback of SQF is the need to set accurately the parameters. If the freezing probability p_f is too low the system will enter the thrashing region. On the other hand, if p_f is too high there are not enough running queries; consequently, the probability of a frozen query to locate a similar answer stream will decrease and the precision of the results will drop.

p_f depends on two major factors [3]: (i) The number $|Q|$ of concurrently active queries in the system. Obviously, the load of the system is proportional to

[3] Other factors include the transfer rate of the links, the speed of the peer computers, the indexing structures, etc.

$|Q|$, therefore when there are more active queries p_f must increase. $|Q|$ can be analyzed as $|Q| = Q_{us} \cdot |P|$, where Q_{us} is the number of queries per user per second and $|P|$ is the number of active peers (i.e., users). (ii) The topology of the network. P2P systems typically exhibit power-law topology. Some hub nodes (e.g., P_4 in Figure 1) receive more messages and become the bottleneck, even if the average load of the system is moderate. Since the system is distributed and dynamic, it is difficult to gather information about these factors. Notice, however, that the effect of varying $|Q|$ or altering the topology is that the waiting time in the queues changes.

The Adaptive Query Freezing (AQF) algorithm controls the waiting time in the queues. Intuitively, if the waiting time is such that there is no time for a query to be forwarded and the answer to come back before the query is aborted, there is no benefit of propagating the query message. Instead there is a penalty, since the message will put additional load to the subsequent peers. In such case, it is better to freeze the query in order to prevent thrashing. On the other hand, if the queues are short it is beneficial to propagate the query in order to improve accuracy. Algorithm 2 presents the details; the OnResultReceived method is the same as in SQF. In contrast to SQF, however, the initiator peer does not need to decide if the new query will freeze. The query is propagated inside the network and each receiving peer decides independently. Notice that AQF is general and versatile since it does not depend on any application-specific criterion.

Algorithm 2 Adaptive Query Freezing

```
On UserQuery(q)
    Initiate an answer stream A_q
    Broadcast q
On QueryReceived(q) // query received from neighbor
    q.traveljd_hops++
    calculate answer and send it to the previous peer
    If q.traveled_hops < max_Hops then
        If checkFreezeCriteria(q) == true then
            q' is a running query which is similar to q
            // checkFreezeCriteria ensures that such q' exists
            Freeze q
            Attach q to q'
        Else broadcast q
```

For an incoming query message MQ at P, the checkFreezeCriteria() function returns true, if:

$$T_{total}(KQ, P) > a_q \cdot MaxWaitTime \tag{2}$$

where T_{total} is defined by Equation 1 and a_q is a system-wide parameter. The accurate evaluation of T_{total} is difficult. The reason is that $T(\cdot)$ depends on the message type. For example, a query message needs more processing time than an answer, since the former requires an expensive search in the local database, while the latter just needs to be propagated. Even if we have an

accurate estimation for each message type we still cannot evaluate T_{total}; the exact processing time will be known when all messages in the queue enter the Processing Unit, since some of them might get frozen.

$MaxWaitTime$ is also an estimation, since each user decides independently when to abort a query. In practice, we expect to get a quite accurate estimation for this parameter by observing the behavior of users over a period of time (i.e., most users would wait for a couple of minutes before aborting and refining their query).

The value of the parameter a_q should be such that it allows enough time for query processing and for answer messages to return to the initiator before the query is aborted. Formally, a_q depends on the query path:

$$a_q(P_0, \ldots, P_i) = \frac{\sum_{j=0}^{i} T_{total}(MQ, P_j)}{MaxWaitTime} \tag{3}$$

where P_0 is the initiating peer and P_i is the current peer. Nevertheless, this formula assumes that every peer has knowledge about the queue waiting time at the other peers in the query path, which is unrealistic. In practice, the exact value of a_q is not critical, as long as it prevents the queues of growing exponentially. By gathering statistics of the queue lengths over a period of time, a_q can be set as a system-wide constant. For our settings we found that $a_q = 1$ provided the best results; however varying it for almost an order of magnitude did not affect considerably the performance, indicating the robustness of AQF.

4.3 Similarity Query Freezing (simQF)

AQF bases its decisions solely on the size of the message queues. While it is a general algorithm, it does not employ any application-specific knowledge to enhance its performance. Since our queries ask for similar images, we developed the simQF freezing algorithm which uses the query similarity as the freezing criterion. The algorithm is the same as AQF, except that a query q is frozen at a peer P if there is an answer stream whose distance to q is less than a threshold ϱ.

Our experiments revealed that simQF produces good results if the threshold ϱ is set correctly; else the behavior resembles the Static Query Freezing technique.

4.4 Multiple-feature queries

One of the major challenges in multimedia retrieval systems is that the similarity between objects cannot be defined precisely with a simple description. For instance, an orange image can be best described by a combination of a color feature similar to yellowish, round shape and possible contents text description like "fruit or orange". Multimedia repositories combining multiple atomic subqueries are called Multiple-feature query systems [8].

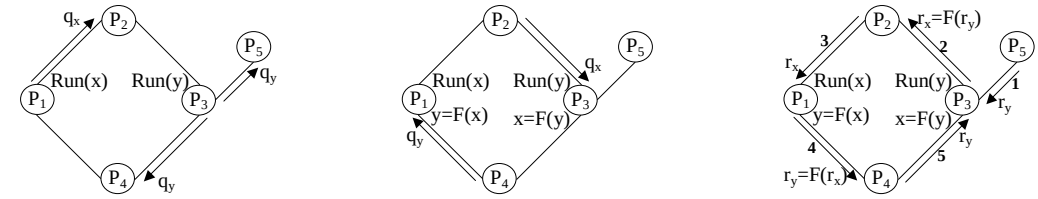
FuzzyPeer supports multiple-feature queries and integrates them into the query freezing framework. Assume that an object is characterized by n feature vectors $v_1, \ldots, v_n$. Then there are $2^n - 1$ possible query types for every combination of features. We support two methods for processing such queries: serial and random.

4.4.1 Multiple-attribute Random processing (maR)

This algorithm is inspired by the Fagin's Algorithm (FA) [8]. Queries that have some common subset of features are considered compatible. For example the query "Color = yellowish and Shape = round" is compatible with "Color = orange and Type = fruit" because they contain the same feature "Color". The algorithm works in two phases. In the first phase, it runs exactly like AQF with the additional characteristic that compatible queries can attach to each other. Because of this, some of the results that arrive from frozen queries are not complete (i.e., they answer only some of the features). In the second phase, the algorithm sorts the incomplete results and selects the top-k according to the similarity metric. For these k objects, it performs direct access to the remote peers that contain them, and receives the complete answers. As we show in the experimental section, this algorithm is beneficial when the data are clustered.

4.4.2 Multi-attribute Sequential processing (maS)

In contrast yo the previous method, this algorithm considers different query types as incompatible, even if they share some features. Therefore, it does never attach a query to an answer stream of a different type. The result is that there are no incomplete answers so there is no need for a second phase. Our experiments revealed that this algorithm is more suitable for uniform datasets.



(a) 1st hop of query propagation (b) 2nd hop of query propagation (c) Duplicate answers

Fig. 3. Cycles due to frozen queries. q_x is attached to q_y while q_y is attached to q_x.

4.5 Dealing with Cycles

Connections among peers are arbitrary, resulting to network graphs that contain cycles. The existence of cycles generates unnecessary messages both during query propagation and in the process of answering frozen queries. The first case is easier to manage: If the cycles are longer than the maximum number of query hops d, there is no overhead. Else, if a peer receives a message that has passed before [4], it simply drops it. The overhead is one extra message per cycle.

The effect of cycles on frozen queries is more complex. To illustrate this, assume the network topology of Figure 3.a, where P_1 initiates query q_x and P_3 initiates q_y. Let q_x reach P_3 faster through P_2 and q_y reach P_1 through P_4, as shown in Figure 3.b (the exact route is not important). P_3 realizes that q_x is similar to q_y, which is already running, so it freezes q_x and attaches it to q_y. In the same way, P_1 freezes q_y since it is similar to q_x. Now, assume that P_3 receives a result r_y from P_5 that answers q_y. Since q_x is attached to q_y, P_3 labels the result as $r_x = F(r_y)$ and propagates it to P_1. There r_x answers q_x that has q_y attached, so it changes again the label to $r_y = F(r_x)$ and sends it to P_3 (Figure 3.c). Obviously, P_3 detects the duplicate answer and rejects it. This kind of cycle, however, creates considerable overhead; up to $O((c-1)\cdot N^d)$ unnecessary messages are propagated, where c is the length of the cycle and N is the maximum degree of the nodes that receive q_y.

In order to break such cycles, we append each answer with information about the transformations that have taken place in the return path. In our example, r_x carries a tag indicating that its original label was r_y; therefore, P_1 will not use r_x to answer q_y. Notice that there may exist cycles with up to l transformations, where l is not bounded by d. Nevertheless, in practice l is expected to be moderate. Long cycles are rare, because the original queries expire causing their attached queries to expire too, before the messages manage to travel all the hops around the cycle. In our experiments, for example, queries

[4] Peers maintain a list with the IDs of messages that have passed recently through them. Each message has a unique ID, consisting of the IP address of the initiating peer and a unique local key.

were expiring within 60sec in networks with 1000 nodes; the resulting cycles contained at most 3 transformations.

More sophisticated solutions are also possible. For instance, the system can implement a cycle avoidance algorithm when propagating the queries, in order to prevent the forming of cycles. Alternatively, we could run a cycle detection and recovery algorithm to un-freeze some of the queries after a cycle is formed. However, both methods would add complexity to the system and introduce overhead due to control messages, while the simple solution we described above works acceptably well.

5 Experimental Evaluation

We employed two implementations to evaluate our methods. The first one is a JAVA prototype based on our BestPeer platform which runs on Pentium III PCs with 256MB RAM and Windows 2000; it was used to derive the basic parameters of the system (see the full paper for details [11]). The parameters were used in our simulator which run on a 2-CPU Ultra-SRARC III server with 4GB RAM. We employed a simulator since it was otherwise impractical to set up large networks, while the benefits of our methods become significant when there are many participating nodes.

Since there is no similar system in use, we did not have any information about the network topology or the users' behavior. As an approximation, we adopted the parameters of existing broadcast-based content-sharing systems which are presented in Ref. [24]. We generated two network topologies for the simulation: (i) Uniform, where the average number of neighbors per node il 3.5, and (ii) Power Law [9]; we employed the PLOD algorithm [15] and set $\alpha \in [0.85, 0.99]$, $\beta \in [96, 355]$, resulting also to 3.2 neighbors per node in average. Nodes are connected either through a slow (WAN [5]) or a fast (LAN) line to the network. A node which is connected through a slow link can support up to 4 concurrent neighbors (this is the default value in Gnutella). The number of nodes which were simultaneously active varied from 100 to 1000. Since in practice only around 5% of the users are active at any given time [23], our results are representative for populations of up to 20000 users.

We used three image datasets in our experiments. The first one, $REAL_{48}$ consists of a library of 10504 high resolution images. We used a vector of 48 coefficients from the Daubechies' wavelet transformation [22] to represent the

[5] In the prototype, LAN nodes were physically connected to a 10Mbps local network, while WAN means that one node was in Singapore and the other in Hong Kong.

visual features of each image. The second dataset, $REAL_{191}$, consists of the same set of images but for each one we extracted two feature vectors. The first is a 32-D vector which is the prefix of the $REAL_{48}$ values. The second vector has 159 dimensions and encodes information about the texture. Our third dataset is $SYNTH_{200}$. It consists of 10000 pairs of feature vectors, one with 32 and the other with 168 dimensions, and was generated synthetically. There are 100 clusters of vectors around 100 random points. The vectors inside each cluster follow a Gaussian distributions, where $\sigma = 0.2$.

5.1 Static Query Freezing

In the first set of experiments we compare a broadcast-based system that does not support frozen queries (denoted as nf) against our static freezing algorithm. There are 100 peers simultaneously on-line and for each setting we calculate the average results over 1000 queries; these are images selected from our dataset with uniform distribution. The peer which initiates each query is also randomly chosen. For every possible query we precalculated the *Global-top-k* which is the set of the top-k images from the entire dataset. k was fixed to 10 for all the experiments in this section. The performance results of our algorithms are presented in Figure 4. In the first row, we draw the $FirstDelay$ which is the average delay in seconds for obtaining the first image which belongs to the Global-Top-k. Intuitively, this measure indicates how long the user must wait until the arrival of the first useful result. If no useful result has arrived before the query is aborted, $FirstDelay$ is set to $MaxWaitTime$. The second row of Figure 4 presents the average precision of the results. $Precision$ is defined as the percentage of results that belong to Global-top-k.

Q_{us} is the number of queries that each user initiates per second. It follows a Gaussian distribution, where $\mu \in [4 \cdot 10^{-3}, 16 \cdot 10^{-3}]$ and $\sigma = 5\% \cdot \mu$. The x-axis in the graphs represents the mean value of Q_{us}; the maximum and minimum value correspond to one query per user every 60 to 250sec, respectively. For the static freezing algorithms 10, 30, 50 and 70% of the queries are selected randomly to freeze 1 hop away from their origin.

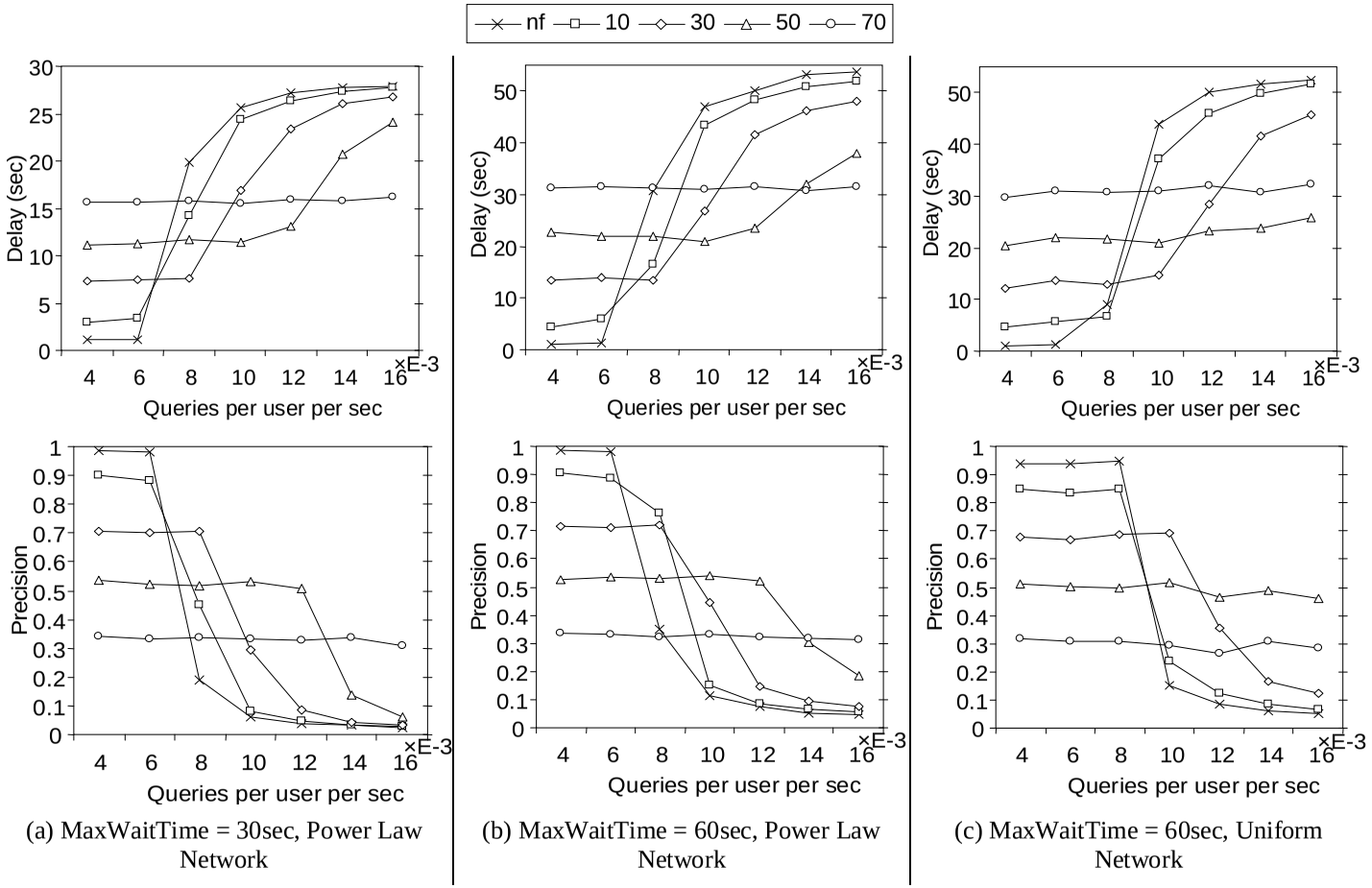


Fig. 4. Non-frozen (nf) vs. 10, 30, 50, 70% statically frozen queries. The first row presents the $FirstDelay$ and the second row the $Precision$ as a function of Q_{us} (Queries per user per second).

In Figure 4.a we set $MaxWaitTime$ = 30sec in a power-law network and allow each query to propagate for up to 7 hops. When Q_{us} is low there is no congestion in the network. The best results both in terms of $FirstDelay$ and $Precision$ are achieved by nf, since a large number of peers is exploited. If there is an attempt to freeze a query q at a peer P, there is only a low probability that q' exists or q and q' are similar enough, so most of the results will be useless.

When the query rate Q_{us} increases, however, the performance of nf deteriorates rapidly. Due to congestions, messages take longer to propagate; therefore $FirstDelay$ increases. Moreover, since there is not enough time to contact many nodes before the queries expire, $Precision$ decreases. Assume now that we freeze 10% of the queries, causing the number of concurrent messages inside the network to decrease. Although both $FirstDelay$ and $Precision$ deteriorate, this occurs at a slower rate than the nf case. The result is that for large values of Q_{us} (greater than $8 \cdot 10^{-3}$ for this setting) $Frozen_{10\%}$ performs better than nf. Performance can be further improved by freezing more queries. For example, freezing 70% of the queries produces better results than $Frozen_{10\%}$ for $Q_{us} \geq 10^{-2}$. The tradeoff is that for smaller values of Q_{us} the $Frozen_{70\%}$ case performs considerably worse.

Figure 4.b depicts the results for the same settings except that $MaxWaitTime$ is fixed to 60sec [6]. Observe that while the trend is the same as before, the absolute values are higher. This is due to the fact that queries are allowed more time to propagate; therefore, they explore a larger part of the network and the relative performance of the algorithms changes. For example, $Frozen_{10\%}$ is now better than $Frozen_{30\%}$ and $Frozen_{50\%}$ for $Q_{us} = 10^{-2}$, because even if there are congested points in the network, there is enough time for the messages to pass through.

In the experiment of Figure 4.c we set again $MaxWaitTime$ to 60sec but we changed the network structure from Power-Law to Uniform. Although the behavior is similar to the previous cases, the performance of $Frozen_{70\%}$ is always worse. This is justified as follows: In a power-law network there are some nodes that receive exponentially more messages that the others. By freezing more queries, such nodes are benefited because they become less congested. In a uniform network, on the other hand, the variation of the peers' workload is not so significant. By freezing 50% of the queries, almost no node is overloaded anymore. If more queries are frozen, there do not remain enough running queries to provide answer streams; therefore, the performance does not improve.

We also tested the behavior of the system for larger user populations. In

[6] $FirstDelay$ is not comparable for different values of $MaxWaitTime$ because of the way it is calculated

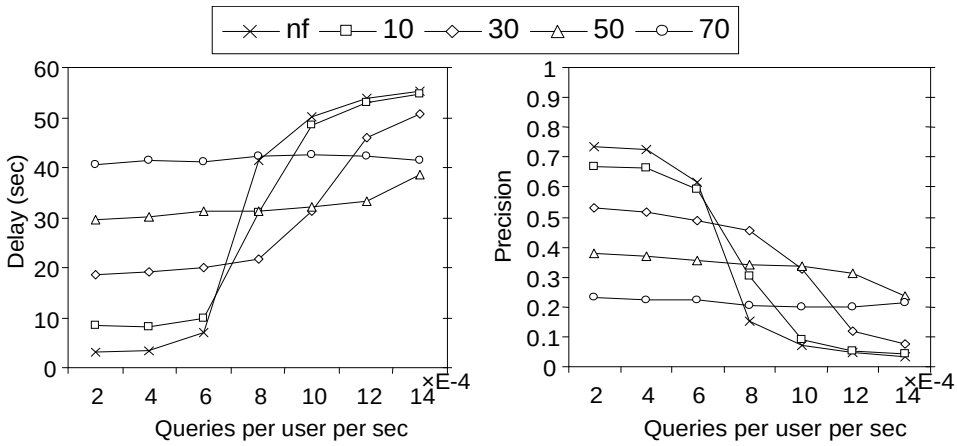


Fig. 5. Non-frozen vs. statically frozen queries. 1000 peers, $MaxWaitTime$ = 60sec, Power Law Network

Figure 5 we show the results for a power-law network with 1000 simultaneously active users. The trend of the algorithms is the same, although the results were obtained for lower values of Q_{us}. The maximum and minimum value of the x-axis correspond to a range of one query per user every 12 to 83min. Notice that the absolute number of concurrent queries that the network supports, remains roughly the same as the previous experiments (i.e., the number of nodes is increased and Q_{us} decreased by one order of magnitude). Also observe that the best value for $Precision$ drops to 0.75 compared to 0.98 for the previous experiments. This happens because the radius of the query horizon (i.e., the maximum number of hops) does not cover the entire network.

The effect of the active users' population size was further investigated by running experiments with variable number of peers. The results verified that increasing the number of active peers has the same effect as increasing the query rate. Therefore, our freezing technique can be successfully employed to improve the scalability of the system both in terms of throughput and number of active users. Also note that in all cases so far, the frozen query was attached to the most beneficial stream. We also tried attaching queries to multiple streams, but there were no significant differences.

5.2 Adaptive Query Freezing

In the following, we evaluate our adaptive freezing algorithm AQF. Figure 6.a presents the results for a power-law network with 100 peers and $MaxWaitTime$ = 30sec. For every value of Q_{us}, we compute the maximum (Upper Bound UB) and the minimum (Lower Bound LB) of the metrics from all the algorithms presented in Figure 4.a (including nf). The adaptive algorithm is compared against the best and worst performance achieved by the static methods. Recall that AQF has only one system-wide parameter a_q (see Equation 2). We

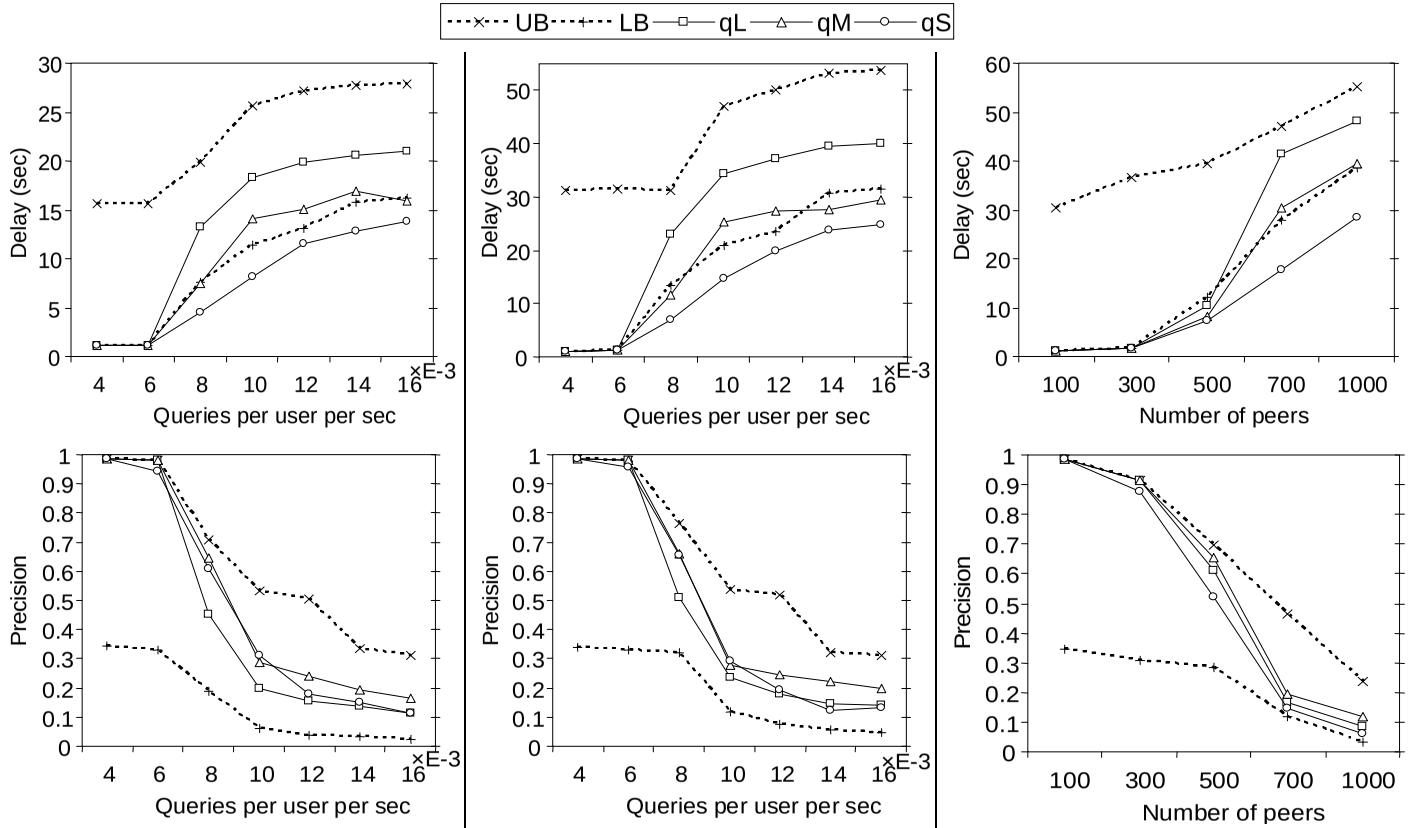
present the results for three values of a_q: (i) $a_q = 4$, which produces long message queues qL, (ii) $a_q = 1$, which corresponds to medium queues qM and (iii) $a_q = \frac{1}{16}$ for short queues qS.

Consider the qL case first; for $Q_{us} \leq 6 \cdot 10^{-3}$ there are relatively a few queries propagated simultaneously in the network. Consequently the waiting time at the message queues at most peers is less than $4 \cdot MaxWaitTime$, so AQF does not freeze any queries; thus AQF behaves like nf. When Q_{us} increases, longer message queues appear. AQF starts freezing some queries and outperforms nf. Nevertheless, the results are still worse than the best static freezing algorithm.

The qM case, on the other hand, prohibits the generation of long queues by freezing more queries. Notice that qM also behaves like nf for $Q_{us} \leq 6 \cdot 10^{-3}$ and beyond this it follows closely the best static result in terms of $FirstDelay$. $Precision$ is also improved compared to qL, but still it is not as good as UB. We investigated further this issue and observed that for uniform networks qM was closer to the best static results. The problem with the power-law network is that the length of the queues may be much larger in some peers (i.e., the may differ up to 2 orders of magnitude for our settings). In such peers most of the queries are frozen even if the similarity with the attached streams is low, leading to low $Precision$. Improving this aspect of AQF is part of our on-going work.

We also tested the qS case which results to shorter message queues. An interesting observation is that the results for $FirstDelay$ are better than the best case of all static alternatives. This illustrates that the static algorithms are not optimal for any percentage of frozen queries, since they do not consider the different conditions at each peer. Notice that qS outperforms qM in terms of $FirstDelay$ because the waiting time of the messages in the queues is shorter. However, qS freezes too many queries so the number of useful answers that reach the initiating peers, drops; therefore, qS is worse than qM in terms of $Precision$. We also experimented with smaller values of a_q. In those cases both $FirstDelay$ and $Precision$ deteriorated since many queries were expiring prior to receiving any useful answer. The above results were also verified by the experiment of Figure 6.b, where $MaxWaitTime$ is set to 60sec.

Finally, we tested AQF for larger user populations. In Figure 6.c we set $Q_{us} = 14 \cdot 10^{-4}$ and varied the number of users from 100 to 1000 in a power-law network; the results support our previous observations. Summarizing, AQF is a practically useful algorithm since it achieves good performance with minimal parameter tuning. The best results were obtained for $a_q = 1$; our experiments, however, revealed that AQF is robust so the exact value of a_q is not critical.



(a) 100 peers, MaxWaitTime = 30sec, Power Law Network

(b) 100 peers, MaxWaitTime = 60sec, Power Law Network

(c) $Q_{us} = 14 \cdot 10^{-4}$, MaxWaitTime = 60sec, Power Law Network

Fig. 6. Adaptive Freezing Algorithm. The first row presents the *FirstDelay* and the second row the *Precision*.

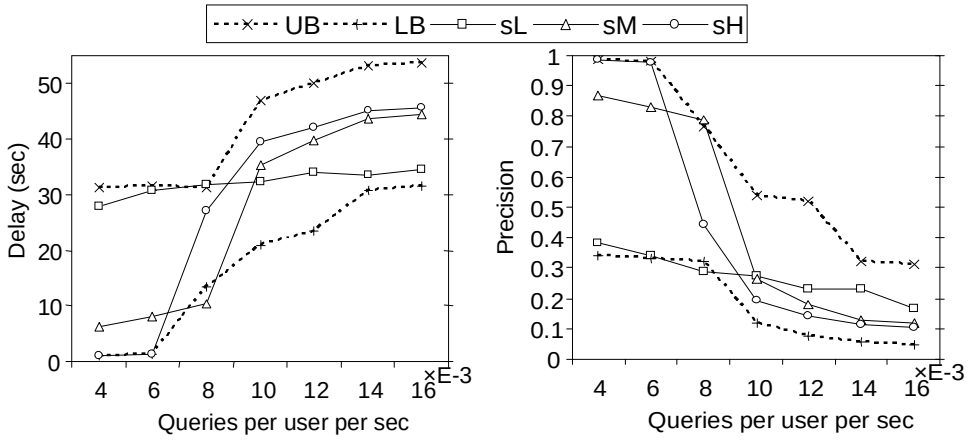


Fig. 7. Similarity Query Freezing. 100 peers, $MaxWaitTime$ = 60sec, Power Law Network.

5.3 Similarity Query Freezing Algorithm

This section evaluates the performance of our alternative freezing algorithm *simQF*. In Figure 7 we present the results for three similarity thresholds: (i) a low similarity case *sL* where $\varrho = 4$, (ii) medium similarity *sM* with $\varrho = 3$ and (iii) high similarity *sH*, where $\varrho = 1$. There are 100 peers in a power-law network; *UB* and *LB* are copied from Figure 6.b. For low query rates, the *sH* case performs better since it does not freeze many queries due to the tight similarity threshold. For the same reason however, *sH* deteriorates fast as Q_{us} increases. Notice that this behavior is similar to nf. Also observe that *sM* and *sL* freeze gradually fewer queries; therefore their relative performance compared to *sH* is low for slow query rates and improves as Q_{us} increases. This behavior is identical to static query freezing, where a tight similarity threshold ϱ corresponds to low percentage of frozen queries. Since the performance of simQF is tightly coupled to ϱ, the algorithm is useful in practice only if there is prior application-specific knowledge for the expected similarity.

5.4 Multiple-feature Queries

In our final set of experiments, we test the performance of AQF for multiple-feature queries. We employed the $SYNTH_{200}$ dataset in a power-law network with 100 peers. Since $SYNTH_{200}$ has 2 feature vectors, there are 3 possible types for every query (i.e., q_{f1}, q_{f2} and $q_{f1,f2}$). The query type was selected with uniform distribution.

The results are presented in Figure 8. nf is the traditional non-freezing broadcast-based algorithm. It does not consider the similarities among query types; q_{f1} and $q_{f1,f2}$ for instance, are treated as different queries. Our multiple-feature

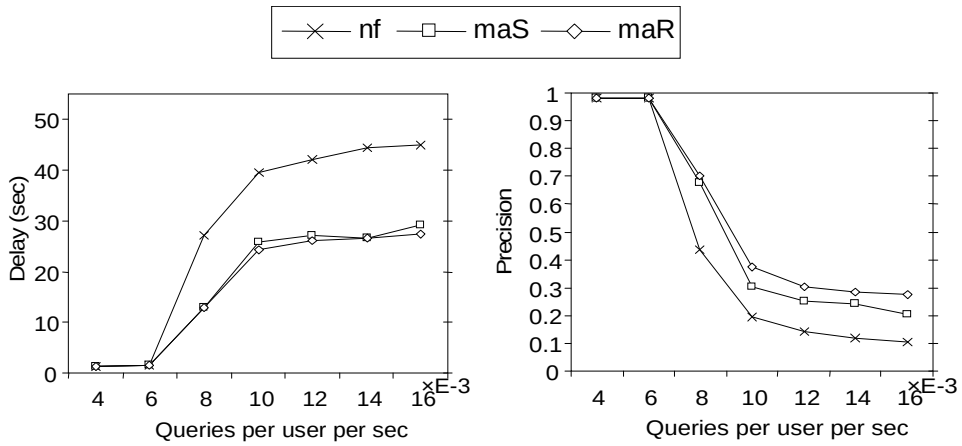


Fig. 8. Multiple-feature Queries. 100 peers, $MaxWaitTime$ = 60sec, Power Law Network, $a_q = 1$, $SYNTH_{200}$ dataset.

serial algorithm *maS* also considers such queries as different. Therefore the improved performance of *maS* is due to adaptive query freezing. The trends of the results are the same as these presented in Figure 6.b, although the absolute values fluctuate due to the different dataset. The results of the multiple-feature random algorithm *maR* are more interesting. Compared to *maS*, $FirstDelay$ does not improve significantly. This is expected since the refinement step of *maR* is initiated at the second step. On the other hand, there is a more obvious improvement in terms of $Precision$, since direct accesses can fetch useful results which would be otherwise inaccessible.

The good performance of *maR* is partially due to the fact that the $SYNTH_{200}$ is clustered. We also run the same experiments with the unclustered $REAL_{181}$. In this case, *maR* performed almost identically to *maS*. Actually, for some settings *maR* was slightly worse that maS due to the overhead of initiating new connections. Nevertheless, in practice this overhead is not significant and since we do not assume any knowledge about the dataset's properties, the possible benefits of *maR* justify its employment.

6 Conclusion

In this paper we dealt with the problem of retrieving information from large repositories built on top of ad-hoc P2P networks. While most existing approaches are limited to exact key matching, we developed FuzzyPeer which supports content based similarity queries. Due to the absence of centralized indexing, such queries are challenging; the difficultly of defining an application independent terminating criterion in addition to their extremely low selectivity, overload the system with useless messages and cause thrashing. We addressed this issue by introducing the freezing technique: some queries

are paused and attached to answer streams from similar concurrently running ones, since the answer for both queries is expected to overlap. We proposed AQF, a simple yet efficient distributed optimization algorithm which improves the scalability and the throughput of the system. Numerous applications, including full-text searching in large archives or fuzzy queries in distributed multimedia repositories, can benefit from our techniques as we demonstrated by an image retrieval case study.

References

[1] Gnutella home page. http://gnutella.wego.com.

[2] Morpheus home page. http://www.musiccity.com.

[3] Napster home page. http://www.napster.com.

[4] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proc. of ACM-SIGMOD*, pages 91–102, 1996.

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of ACM-SIGMOD*, pages 379–390, 2000.

[6] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, January/February 2002.

[7] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proc. of ICDCS*, 2002.

[8] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of ACM-PODS*, pages 216–226, 1996.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proc. of ACM-SIGCOMM*, pages 251–262, 1999.

[10] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K. L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *Proc. of ACM-SIGMOD*, pages 25–36, 2002.

[11] P. Kalnis, W. S. Ng, B. C. Ooi, and K. L. Tan. Answering similarity queries in peer-to-peer networks. http://www.comp.nus.edu.sg/-kalnis/ASQ.pdf.

[12] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of ACM-SIGMOD*, pages 49–60, 2002.

[13] W. Ng, B. Ooi, and K. Tan. Bestpeer: A self-configurable peer-to-peer system (poster). In *ICDE*, 2002.

[14] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The qbic project: Querying images by content using color, texture and shape. In *Storage and Retrieval for Image and Video Databases, SPIE*, pages 173–187, 1993.

[15] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *Proc. of GLOBE-COM*, 2000.

[16] A. Pentland, R. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. In *SPIE, Storage and Retrieval Image and Video Databases II*, 1995.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of ACM-SIGCOMM*, pages 161–172, 2001.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM ICDSP*, 2001.

[19] O. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *Proc. of ICDE*, 2004.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM-SIGCOMM*, pages 149–160, 2001.

[21] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. of ACM-SIGCOMM*, 2003.

[22] J. Wang, G. Wiederhold, O. Firschein, and S. Wei. Content-based image indexing and searching using daubechies' wavelets. *Int. Journal on Digital Libraries*, 1:311–328, 1997.

[23] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proc. of VLDB*, pages 561–570, 2001.

[24] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *Proc. of ICDCS*, pages 5–14, 2002.

[25] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. of ICDE*, 2003.

[26] C. Yu, B. Ooi, K. Tan, and H. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proc. of VLDB*, pages 421–430, 2001.