Abstract Approximating Aggregation Queries in Peer-to-Peer Networks

Benjamin Arai, *UC Riverside*, *barai@cs.ucr.edu*
Gautam Das, *UT Arlington*, *gdas@cse.uta.edu*
Dimitrios Gunopulos, *UC Riverside*, *dg@cs.ucr.edu*
Vana Kalogeraki, *UC Riverside*, *vana@cs.ucr.edu*

Abstract

Peer-to-peer databases are becoming prevalent on the Internet for distribution and sharing of documents, applications, and other digital media. The problem of answering large scale, ad-hoc analysis queries – e.g., aggregation queries – on these databases poses unique challenges. Exact solutions can be time consuming and difficult to implement given the distributed and dynamic nature of peer-to-peer databases. In this paper we present novel sampling-based techniques for approximate answering of ad-hoc aggregation queries in such databases. Computing a high-quality random sample of the database efficiently in the P2P environment is complicated due to several factors – the data is distributed (usually in uneven quantities) across many peers, within each peer the data is often highly correlated, and moreover, even collecting a random sample of the peers is difficult to accomplish. To counter these problems, we have developed an adaptive two-phase sampling approach, based on random walks of the P2P graph as well as block-level sampling techniques. We present extensive experimental evaluations to demonstrate the feasibility of our proposed solution.

1. Introduction

Peer-to-Peer Databases: The peer-to-peer network model is quickly becoming the preferred medium for file sharing and distributing data over the Internet. A peer-to-peer (P2P) network consists of numerous peer nodes that share data and resources with other peers on an equal basis. Unlike traditional client-server models, no central coordination exists in a P2P system, thus there is no central point of failure. P2P network are scalable, fault tolerant, and dynamic, and nodes can join and depart the network with ease. The most compelling applications on P2P systems to date have been file sharing and retrieval. For example, P2P systems such as Napster [24], Gnutella [14], KaZaA [19] and Freenet [12] are principally known for their file sharing capabilities, e.g., the sharing of songs, music, and so on. Furthermore, researchers have been interested in extending sophisticated IR techniques such as keyword search and relevance retrieval to P2P databases.

Aggregation Queries: In this paper, however, we consider a problem on P2P systems that is different from the typical search and retrieval applications. As P2P systems mature beyond file sharing applications and start getting deployed in increasingly sophisticated e-business and scientific environments, the vast amount of data within P2P databases pose a different challenge that has not been adequately researched thus far – that of how to answer *aggregation queries* on such databases. Aggregation queries have the potential of finding applications in decision support, data analysis and data mining. For example, millions of peers across the world may be cooperating on a grand experiment in astronomy, and astronomers may be interesting in asking decision support queries that require the aggregation of vast amounts of data covering thousands of peers.

We make the problem more precise as follows. Consider a single table T that is distributed over a P2P system; i.e., the peers store horizontal partitions (of varying sizes) of this table. An aggregation query such as the following may be introduced at any peer (this peer is henceforth called the *sink*):

Aggregation Query

```
SELECT Agg-Op(Col) FROM T
WHERE selection-condition
```

In the above query, the *Agg-Op* may be any aggregation operator such as SUM, COUNT, AVG, and so on; *Col* may be any numeric measure column of T, or even an expression involving multiple columns; and the *selection-condition* decides which tuples should be involved in the aggregation. While our main focus is on the above standard SQL aggregation operators, we also briefly discuss other interesting statistical estimators such as medians, quantiles, histograms, and distinct values.

While aggregation queries have been heavily investigated in traditional databases, it is not clear that these techniques will easily adapt to the P2P domain. For example, decision support techniques such as OLAP commonly employ materialized views, however the distribution and management of such views appears difficult in such a dynamic and decentralized domain [18, 10]. In contrast, the alternative of answering aggregation queries at runtime "from scratch" by crawling and scanning the entire P2P repository is prohibitively slow.

Approximate Query Processing: Fortunately, it has been observed that in most typical data analysis and data mining applications, timeliness and interactivity are more important considerations than accuracy - thus data analysts are often willing to overlook small inaccuracies in the answer provided the answer can be obtained fast enough. This observation has been the primary driving force behind recent development of *approximate query processing* (AQP) techniques for aggregation queries in traditional databases and decision support systems [9, 3, 6, 8, 1, 13, 5, 7, 22]. Numerous AQP techniques have been developed, the most popular ones based on random sampling, where a small random sample of the rows of the database is drawn, the query is executed on this small sample, and the results extrapolated to the whole database. In addition to simplicity of implementation, random sampling has the compelling advantage that in addition to an estimate of the aggregate, one can also provide confidence intervals of the error with high probability. Broadly, two types of sampling-based approaches have been investigated: (a) *Pre-computed samples* - where a random sample is pre-computed by scanning the database, and the same sample is reused for several queries, and (b) *Online samples* - where the sample is drawn "on the fly" upon encountering a query.

Goal of Paper: In this paper, we also approach the challenges of decision support and data analysis on P2P databases in the same manner, i.e., we investigate what it takes to enable AQP techniques on such distributed databases.

Goal of Paper: Approximating Aggregation Queries in P2P Networks

Given an aggregation query and a desired error bound at a sink peer, compute with "minimum cost" *an approximate answer to this query that satisfied the error bound.*

The *cost* of query execution in traditional databases is usually a straightforward concept – it is either I/O cost or CPU cost, or a combination of the two. In fact, most AQP approaches simplify this concept even further, by just trying to minimize the number of tuples in the sample; thus making the assumption that the sample size is directly related to the cost of query execution. However, in P2P networks, the cost of query execution is a combination of several quantities, e.g., the number of *participating peers*, the *bandwidth* consumed (i.e., amount of data shipped over the network), the number of *messages* exchanged, the *latency* (the end-to-end time to propagate the query across multiple peers and receive replies), the *I/O cost* of accessing data from participating peers, the *CPU cost* of processing data at participating peers, and so on. In this paper, we shall be concerned with several of these cost metrics.

Challenges: Let us now discuss what it takes for sampling-based AQP techniques to be incorporated into P2P systems. We first observe that two main approaches have emerged for constructing P2P networks today, *structured* and *unstructured*. Structured P2P networks (such as Pastry [26] and Chord [29]) are organized in such a way that data items are located at specific nodes in the network and nodes maintain some state information, to enable efficient retrieval of the data. This organization sacrifices atomicity by mapping data items to particular nodes and assume that all nodes are equal in terms of resources, which can lead to bottlenecks and hot-spots. Our work focuses on unstructured P2P networks, which make no assumption about the location of the data items on the nodes, and nodes are able to join the system at random times and depart without a priori notification. Several recent efforts have demonstrated that unstructured P2P networks can be used efficiently for multicast, distributed object location and information retrieval [23, 30].

For approximate query processing in unstructured P2P systems, attempting to adapt the approach of pre-computed samples is impractical for several reasons: (a) it involves scanning the entire P2P repository, which is difficult, (b) since no centralized storage exists, it is not clear where the pre-compute sample should reside, and (c) the very dynamic nature of P2P systems indicates that pre-computed samples will quickly become stale unless they are frequently refreshed.

Thus, the approach taken in this paper is to investigate the feasibility of online sampling techniques for AQP on P2P databases. However, online sampling approaches in P2P databases pose their own set of challenges. To illustrate these challenges, consider the problem of attempting to draw a *uniform random sample* of n tuples from such a P2P database containing a total of N tuples. To ensure a true uniform random sample, our sampling procedure should be such that each subset of n tuples out of N should be equally likely to be drawn. However, this is an extremely challenging problem due to the following two reasons.

- Picking even a set of uniform random peers is a difficult problem, as the sink does not have the IP addresses of all peers in the network. This is a well-known problem that other researchers have tackled (in different contexts) using *random walk* techniques on the P2P graph [13, 20, 4] – i.e., where a Markovian random walk is initiated from the sink that picks adjacent peers to visit with equal probability, and under certain connectivity properties, the random walk is expected to rapidly reach a stationary distribution. If the graph is badly clustered with small cuts, this affects the speed at which the walk converges. Moreover, even after convergence, the stationary distribution is *not uniform*; in fact, it is skewed towards giving higher probabilities to nodes with larger degrees in the P2P graph.
- Even if we could select a peer (or a set of peers) uniformly at random, it does not make the problem of selecting a *uniform random set of tuples* much easier. This is because visiting a peer at random has an associated overhead, thus it makes sense to select multiples tuples at random from this peer during the same visit. However, this may compromise the quality of the final set of tuples retrieved, as the tuples within the same peer are likely to be *correlated* – e.g., if the P2P database contained listings of, say movies, the movies stored on a specific peer are likely to be of the same genre. This correlation can be reduced if we select just one tuple at random from a randomly selected peer; however the overheads associated with such a scheme will be intolerable.

Our Approach: We briefly describe the framework of our approach. Essentially, we abandon trying to pick true uniform random samples of the tuples, as such samples are likely to be extremely impractical to obtain. Instead, we consider an approach where we are willing to work with *skewed samples*, provided we can accurately estimate the skew during the sampling process. To get the accuracy in the query answer desired by the user, our skewed samples can be larger than the size of a corresponding uniform random sample that delivers the same accuracy, however, our samples are much more cost efficient to generate.

Although we do not advocate any significant pre-processing, we assume that certain aspects of the P2P graph are known to all peers, such as the average degree of the nodes, a good estimate of the number of peers in the system, certain topological characteristics of the graph structure, and so on. Estimating these parameters via pre-processing are interesting problems in their own right, however we omit these details from this paper. The main point we make is that these parameters are relatively slow to change and thus do not have to be estimated at query time – it is the data contents of peers that changes more rapidly, hence the random sampling process that picks a representative sample of tuples has to be done at runtime.

Our approach has two major phases. In the first phase, we initiate a fixed-length random walk from the sink. This random walk should be long enough to ensure that the visited peers[1] represent a close sample from the underlying stationary distribution – the appropriate length of such a walk is determined in a pre-processing step. We then retrieve certain information from the visited peers, such as the number of tuples, the aggregate of tuples (e.g., SUM/COUNT/AVG, etc.) that satisfy the selection condition, and send this information back to the sink. This information is then analyzed at the sink to determine the skewed nature of the data that is distributed across the network - such as the variance of the aggregates of the data at peers, the amount of correlation between tuples that exists within the same peers, the variance in the degrees of individual nodes in the P2P graph (recall that the degree has a bearing on the probability that a node will be sampled by the random walk), and so on. Once this data has been analyzed at the sink, an estimation is made on how much more samples are required - and in what way should these samples be collected - so that the original query can be optimally answered within the desired accuracy with high probability. For example, the first phase may recommend that the best way to answer this query is to visit m' more peers, and from each peer, randomly sample t tuples. We mention that the first phase is not overly driven by heuristics – instead it is based on strong underlying theoretical principles, such as theory of random walks [13, 20, 4], as well as statistical techniques such as cluster sampling, block-level sampling and cross-validation [9, 15].

The second phase is then straightforward – a random walk is reinitiated and tuples collected according to the recommendations made by the first phase. Effectively, the first phase is used to "sniff" the network and determine an optimal-cost "query plan", which is then implemented in the second phase. For certain aggregates, such as COUNT and SUM, further optimizations may be achieved by pushing the selections and aggregations to the peers – i.e., the local aggregates instead of raw samples are returned to the sink, which are then composed into a final answer.

Summary of Contributions:

- We introduce the important problem of approximate query processing in P2P databases that is likely to be of increasing significance in the future.
- The problem is analyzed in detail, and its unique challenges are comprehensively discussed.

[1] Actually, only a small fraction of the visited peers are selected for consideration, and the remaining is "jumped over" – this is determined by the *jump size* parameter that is discussed in later sections.

- Adaptive, two-phase sampling-based approaches are proposed, based on well-founded theoretical principles.
- The results of extensive experiments are presented that demonstrate the importance of the problem and the validity of our approaches.

The rest of this paper is organized as follows. In Section 2 we describe related work. We provide the foundation of our approach in Section 3, and the algorithm in Section 4. In Section 5 we present the results of experiments, and conclude in Section 6.

2. Related Work

Peer-to-Peer (P2P) systems are becoming very popular because they provide an efficient mechanism for building large, scalable systems [23]. Most recent work has focused on Distributed Hash Tables (DHTs) [25, 26, 29]. Such techniques provide scalability advantages over unstructured systems (such as Gnutella) however they are not flexible enough for some applications, especially when nodes join or leave the network frequently or change their connections often.

Recent work has proposed different techniques for exact query processing in P2P systems. Most proposals use structured overlay networks (DHTs), such as CAN, Pastry, or Chord. Such techniques include PIER [16], DIM [22], or [27], and since they use DHTs they have a different focus and are not directly applicable to our case. A hybrid system, Mercury [4], using routing hubs to answer range queries, was also recently proposed. This system is also designed to provide exact answers to range queries. Exact solutions to OLAP queries have been considered in [10, 18].

Methods to sample random peers in P2P networks have been proposed in [13, 20, 4]. These techniques use Markov chain random walks to select random peers from the network. Their results show that when certain structural properties of the graph are known or can be estimated (such as the second eigenvalue of the graph) the parameters of the walk can be set so that a representative sample of the stationary distribution can be collected with high probability. In [4] it is shown that if the graph is an expander, a random walk converges to the stationary distribution in O(logM) steps, where M is the number of peers in the network.

Our work also generalizes to the P2P domain previous work on approximate query processing in relational databases. Recent work by [9, 3, 6, 8, 1, 13, 5, 7, 22] has developed powerful techniques for employing sampling in the database engine to approximate aggregation queries and to estimate database statistics. Recent techniques have focused on providing formal foundations and algorithms for block-level sampling and are thus most relevant to our work. The objective in block-level sampling is to derive a representative sample by only randomly selecting a set of disk blocks of a relation [9, 15]. Specifically, [9] presents a technique for histogram estimation that uses cross-validation to identify the amount of sampling required for a desired accuracy level. In addition, the paper [15] considers the problem of deciding what percentage of a disk block should be included in the sample, given a cost model.

3. Foundations of our Approach

In this section we discuss the principles behind our approach for approximate query processing on P2P databases. Our actual algorithm is described in Section 4.

3.1. The Peer-to-Peer Model

We assume an unstructured P2P network represented as a graph $G = (P, E)$, with a vertex set $P=\{p_1, p_2, ..., p_M\}$ and an edge set E. The vertices in P represent the peers in the network and the edges in E represent the connections between the vertices in P. Each peer p is identified by the processor's IP address and a port number (IP_p, $port_p$). The peer p is also characterized by the capabilities of the processor on which it is located, including its CPU speed p_{cpu}, memory bandwidth p_{mem} and disk space p_{disk}. The node also has a limited amount of bandwidth to the network, noted by p_{band}. In unstructured P2P networks, a node becomes a member of the network by establishing a connection with at least one peer currently in the network. Each node maintains a small number of connections with its peers; the number of connections is typically limited by the resources at the peer. We denote the number of connections a peer is maintaining by p_{conn}.

The peers in the network use the Gnutella's P2P protocol to communicate. The Gnutella P2P protocol supports four message types (Ping, Pong, Query, Query_Hit); of which the Ping and Pong messages are used to establish connections with other peers, and the Query and Query_Hit messages are used to search in the P2P network. Gnutella, however, uses a naïve Breadth First Search (BFS) technique in which queries are propagated to all the peers in the network, and thus consumes excessive network and processing resources and results in poor performance. Our approach, on the other hand, uses a probabilistic search algorithm based on random walks. The key idea is that, each node forwards a query message, called *walker*, randomly to one of its adjacent peers. This technique is shown to improve the search efficiency and reduce unnecessary traffic in the P2P network.

3.2. Query Cost Measures

As mentioned in the introduction, the cost of the execution of a query in P2P databases is more complicated that equivalent cost measures in traditional databases. The primary cost measure we consider is

latency, which is the end-to-end time to propagate the query across multiple peers and receive replies.

For the purpose of illustration, we focus in this section on the SUM and COUNT aggregates. For these specific aggregates, latency can be approximated by an even simpler measure: *the number of peers that participate in the algorithm*. This measure is appropriate for these aggregates primarily because the overheads of visiting peers dominate other incurred costs.

To see this, we note that the aggregation operator (as well as the selection filter) can be pushed to each visited peer. Once a peer is visited by the algorithm, the peer can be instructed to simply execute the original query on its local data and send only the aggregate (and its degree) back to the sink, from which the sink can reconstruct the overall answer. Moreover, this information can be sent directly without necessitating any intermediate hops, as the visited peer knows the IP address of the sink from which the query originated. Thus the bandwidth requirement of such an approach is uniformly very small for all visited peers – they are not required to send more voluminous raw data (e.g., all or parts of the local database) back to the sink.

In approximating latency by the number of visited peers, we also make the implicit assumption that the overhead of visiting peers dominates the costs of local computations (such as, execution of the original query on the local database). This is of course true if the local databases are fairly small. To ensure that the local computations remain small even if local databases are large, our approach in such cases is to execute the aggregation query only on a small fixed-sized random sample of the local data – i.e., we *sub-sample* from the peer - scale the result to the entire local database, and send the scaled aggregate back to the sink. This way, we ensure that the local computations are uniformly small across all visited peers.

In summary, for SUM and COUNT aggregates, latency is shown to be proportional to the number of visited peers. Thus, our goal is to minimize the number of peers that must be visited in order to arrive at an approximate answer with the desired accuracy.

We mention that for other types of aggregations – e.g., statistics computations such as medians, quantiles, histograms, and distinct values – the cost model is more complex as the aggregation operator usually cannot be pushed to the peers. In such cases, more voluminous data has to be sub-sampled from the visited peers and sent back to the sink, thus incurring nontrivial bandwidth costs. An appropriate cost model usually has to take into account multiple factors, such as costs of visiting peers, local computations at peers, transportation of data back to the sink, and local computations at the sink. Handling such aggregations is part of ongoing work – e.g., we have interesting results on the median and quantile computations that are presented later in the paper - however we omit complete details of these efforts due to lack of space.

3.3. Random Walk in Graphs

In seeking a random sample of the P2P database, we have to overcome the sub-problem of how to collect a random sample of the peers themselves. Unrepresentative samples of peers can quickly skew results producing erroneous aggregation statistics. Sampling in a non-hierarchical decentralized P2P network presents several obstacles in obtaining near uniform random samples. This is because no peer (including the query sink) knows the IP addresses of all other peers in the network – they are only aware of their immediate neighbors. If this were not the case, clearly the sink could locally generate a random subset of IP addresses from among all the IP addresses, and visit the appropriate peers directly. We note that this problem is not encountered in traditional databases, as even if one has to resort to cluster (or block-level) sampling such as in [9, 15], obtaining an efficient sample of the blocks themselves is straightforward.

This problem has been recognized in other contexts (see [13] and the references therein), and interesting solutions based on *Markov chain random walks* have been proposed. We briefly review such approaches here. A Markov chain random walk is a procedure that is initiated at the sink, and for each visited peer, the next peer to visit is selected with equal probability from among its neighbors (and itself – thus self loops are allowed). It is well known that, if this walk is carried out long enough, the eventual probability of reaching any peer p will reach a stationary distribution. To make this more precise, let $P = \{p_1, p_2, \ldots, p_M\}$ be the entire set of peers, let E be the entire set of edges, and let the degree of a peer p be $\deg(p)$. Then the probability of any peer p in the stationary distribution is:

$$prob(p) = \frac{\deg(p)}{2|E|}$$

It is important to note that the above distribution is *not uniform* – the probability of each peer is proportional to its degree. Thus, even if we can efficiently achieve this distribution, we will have to compensate for the fact that the distribution is skewed as above, if we have to use samples drawn from it for answering aggregation queries.

The main issue that has concerned researchers has been the *speed of convergence*, i.e., how many hops h are necessary before one gets close to the stationary distribution. Most results have pointed to certain broad connectivity properties that the graph should possess for this to happen. In particular, it has been shown that if the transition probabilities that govern the random walk on the P2P graph are modeled as an MxM matrix, the *second eigenvalue* plays an important role in these convergence results. The second eigenvalue describes connectivity

properties of graphs - in particular whether the graph has *small cuts* which would adversely impact the length of the walk necessary to arrive at convergence. For example, Figure 2 describes a clustered graph with a small cut.

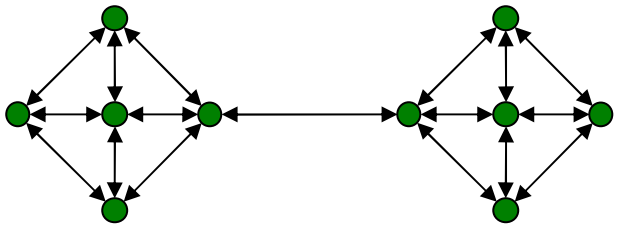


Figure 1: Two clusters with a small cut between each other

As the results in [13] show, if the P2P graph is well connected (i.e., it has a small second eigenvalue, and a minimum degree of the graph is large), then the random walk quickly converges as it "loses memory" rapidly. In fact, under certain specific conditions of connectedness (e.g., expander graphs that are common in P2P networks), convergence can be achieved in O(log$M$) steps.

In our case, recall from the introduction that we assume that we are allowed a certain amount of preprocessing to determine various properties of the P2P graph that will be useful at query time – under the assumption that the graph topology changes less rapidly compared to the data content at the peers. The speed of convergence of a random walk in this graph is determined in this preprocessing step, in addition to other useful properties such as number of nodes M, the number of edges $|E|$, and so on. With respect to speed of convergence, we essentially determine a *jump* parameter j that determines how many peers to skip between selections of peers for the sample. As the jump increases, the correlation between successive peers that are selected for the sample decreases rapidly.

3.4. Sampling Theorems

In this subsection, we shall develop the formal sampling theorems that drive our algorithm. We shall show how the tuples that are retrieved from the first phase of our algorithm can be utilized to recommend how the second phase should be executed, i.e., the "query plan" for answering the query approximately so that a desired error is achieved.

We focus here on the COUNT aggregate for the purpose of illustrating our main ideas (our formal results can be easily extended for the SUM case). Finally, to keep the discussion simple, we assume that all local databases at peers are small, i.e., sub-sampling is not required (our results can be extended for the sub-sampling case, and in fact our algorithm in Section 4 does not make this assumption).

As discussed earlier, our algorithm has two phases. In the first phase, our algorithm will visit a predefined number of peers m using a random walk such that the sample of visited peers will appear as if they have been drawn from the stationary distribution of the graph. The query will be executed locally at each visited peer, and the aggregates will be sent back to the sink, along other information such as the degrees of the visited peers (from which information such as the peers probabilities in the stationary distribution can be computed). The sink analyzes this information, and then determines how many more peers need to be visited in the second phase. The theorems that we develop next provide the foundations on which the decisions in the first phase are made.

Recall that $P = \{p_1, p_2, \ldots, p_M\}$ is the set of peers.

For a tuple u, let $y(u) = 1$ if u satisfies the selection condition, and = 0 otherwise.

Let the aggregate for a peer p be $y(p) = \sum_{u \in p} y(u)$

Let y be the exact answer for the query, i.e. $y = \sum_{p \in P} y(p)$

The query also comes with a desired error threshold Δ_{req}. The implication of this requirement is that, if y' is the estimated count by our algorithm, then

$$\left| y - y' \right| \leq \Delta_{req}$$

Now, consider a fixed-size sample of peers $S = \{s_1, s_2 \ldots s_m\}$ where each s_i is from P. This sample is picked by the random walk in the first phase. We can approximate this process as that of picking peers in m rounds, where in each round a random peer s_i is picked from P with probability $prob(s_i)$. We also assume that peers may be picked *with replacement* – i.e., multiple copies of the same peer may be added to the sample – as this greatly simplifies the statistical derivations below.

Consider the quantity y'' defined as follows

$$y'' = \frac{\sum_{s \in S} \frac{y(s)}{prob(s)}}{m} \qquad (1)$$

Theorem 1: $E[y''] = y$, that is, y'' is an unbiased estimator of y.

Proof: Intuitively, each sampled peer s tries to estimate y as $y(s)/prob(s)$, i.e., by scaling its own aggregate by the inverse of its probability of getting picked. The final estimate y'' is simply the average of the m individual estimates.

To proceed with the proof, consider the simple case of only one sampled peer, i.e., $m = 1$. In this case,

$$E[y''] = \sum_{p \in P} \left(\frac{y(p)}{prob(p)} \right) prob(p) = y$$

To extend to any m, we make use of the *linearity of expectation* formula: $E[X+Y] = E[X] + E[Y]$ for random variables X and Y (that need not even be independent). Thus if the expected estimate of any single random peer is y, then the expected average estimate by m random peers is also y. □

We next need to determine the variance of the random variable y''.

Theorem 2 (Standard Error Theorem):

$$Var[y''] = \frac{\sum_{p \in P} \left(\frac{y(p)}{prob(p)} - y \right)^2 prob(p)}{m}$$

Proof: To easily derive this variance, let us consider the simple case of only one sampled peer, i.e., $m = 1$. In this case, it is easy to see that the variance is defined by the quantity

$$C = \sum_{p \in P} \left(\frac{y(p)}{prob(p)} - y \right)^2 prob(p)$$

To extend to any m, we make use of the following formulas for variance: (a) $Var[aX] = a^2 Var[X]$, and (b) $Var[X+Y] = Var[X] + Var[Y]$, where X and Y are independent random variables and a is a constant. Using these formulas, we can easily show that $Var[y''] = C/m$. □

The above **Standard Error Theorem** shows that the variance varies inversely as the sample size. The quantity C also represents the "badness" of the clustering of the data in the peers – the larger the C, the more the correlation amongst the tuples within peers, and consequently the more peers need to be sampled to keep the variance of the estimator y'' small. Notice also that if we divide the variance by N^2, we will effectively get the square of the error of the relative count aggregate, if y'' was used as an estimator for y.

Our case is actually the reverse, i.e., we are given a desired error threshold Δ_{req}, and the task is to determine the appropriate number of peers to sample that will satisfy this threshold. Of course, we have used a fixed-sized m in the first phase, so unless we are simply lucky, it's unlikely that this particular m will satisfy the desired accuracy. However, we can use the first phase more carefully to determine the appropriate sample size to draw in the second phase, say m'.

The main task is to use the sample drawn in the first phase to try and estimate C; because once we estimate C, we can determine m' using Theorem 2. We suggest a simple *cross-validation* procedure as described below to estimate C (this procedure is inspired by previous work in a different context, see [9]).

Consider two random sample of peers of size m each drawn from the stationary distribution. Let y_1'' and y_2'' be the two estimates of y by these samples respectively according to Equation 1. We define the *cross-validation* error as: $CVError = |y_1'' - y_2''|$

Theorem 3: $E[CVError^2] = 2E[(y'' - y)^2]$

Proof:

$$E[CVError^2] = E[(y_1'' - y_2'')^2] =$$
$$E[(y_1'' - y)^2] + E[(y_2'' - y)^2] = 2E[(y'' - y)^2]$$

□

This theorem says that the expected value of the square of the cross-validation error is 2 times the expected value of the square of the actual error.

This cross-validation error can be estimated in the first phase by the following procedure. Randomly divide the m samples into two halves, and compute the cross-validation error (for sample size $m/2$). We can then determine C by fitting this computed error and the sample size $m/2$ into the equation in Theorem 2. To get a somewhat more robust estimation for C, we can repeat the random halving of the sample collected in the first phase several times and take the average value of C. We also note that since the cross-validation error is larger than the true error, the value of C is conservatively overestimated.

Once C is determined (i.e., the "badness" of the clustering of data in the peers), we can determine the right number of peers to sample in the second phase, m', to achieve the desired accuracy.

4. Our Algorithm

In this section we present details of our two-phase algorithm for approximating answering of aggregate queries. For the sake of illustration, we focus on approximating COUNT queries – it can be easily extended to SUM queries. The pseudo code of the algorithm is presented below.

Algorithm: COUNT queries

Predefined Values

```
M       : Total number of peers in network
E       : Total number of edges in network
m       : Number of peers to visit in Phase I
j       : Jump size for random walk
t       : Max #tuples to be sub-sampled per peer
```

Inputs

```
Q        : COUNT query with selection condition
Sink     : Peer where query is initiated
```
Δ_{req} : Desired max error

Phase I

```
// Perform Random Walk
1. Curr = Sink; Hops = 1;
2. while (Hops < j * m) {
3.      if (Hops % j)
4.              Visit(Curr);
5.      Hops++;
6.      Curr = random adjacent peer
7. }


// Visit Peer
1. Visit(Curr) {
2.      if (#tuples of Curr) <= t) {
3.              Execute Q on all tuples
4.      else
5.              Execute Q on t randomly sampled
6.              tuples
7.      }
```
8. $y(Curr) = \left(\frac{\#tuples}{\#processedTuples} \right) * (result\_of\_Q)$
```
10.     Return (y(Curr), deg(Curr)) to Sink
11.     }


// Cross-Validate at Sink
1. Let S = {s1, s2, …, sm} be the visited peers
2. Partition S randomly into two halves: S1 & S2
3. Compute
```
$$y_1'' = \frac{\sum_{s \in S_1} y(s)/prob(s)}{m/2} \qquad y_2'' = \frac{\sum_{s \in S_2} y(s)/prob(s)}{m/2}$$

where $prob(s) = \frac{\deg(s)}{2E}$

4. Compute $CVError = |y_1'' - y_2''|$

5. Return $m' = (m/2) * \left(\frac{CVError^2}{\Delta_{req}^2} \right)$

Phase II

1. Visit m' peers using random walk
2. Let S' = {s_1, s_2, …, $s_{m'}$} be the visited peers
3. Return $y' = \frac{\sum_{s \in S'} y(s)/prob(s)}{m'}$

Our approach in the first phase is broken up into the following main components. First, we perform a random walk on the peer-to-peer network, attempting to avoid skewing due to graph clustering and vertices of high degree. Our walk skips *j* nodes between each selection to reduce the dependency between consecutive selected peers. As the jump size increases, our method increases overall bandwidth requirements within the database but for most cases small jump sizes suffice for obtaining random samples.

Second, we compute aggregates of the data at the peers and send these back to the sink. Note that in the previous section, we had not formally discussed the issue of sub-sampling at peers – this was primarily done to keep the previous discussion simple. In reality, the local databases at some peers can be quite large, and aggregating them in their entirety may not be negligible compared to the overhead of visiting the peer – in other words, the simplistic cost model of only counting the number of visited peers is inappropriate. In such cases, it is preferable to randomly sub-sample a small portion of the local database, and apply the aggregation only to this sub-sample. Thus, the ideal approach for this problem is to develop a cost model that takes into account cost of visiting peers as well as local processing costs; and for such cost models, an ideal two-phase algorithm should determine various parameters in the first phase, such as how many peers to visit in the second phase, and how many tuples to sub-sample from each visited peer. In this paper we taken a somewhat simpler approach, in which we fix a constant *t* (determined at preprocessing time via experiments), such that if a peer has at most *t* tuples, its database is aggregated in its entirety, whereas if the peer has more than *t* tuples, then *t* tuples are randomly selected and aggregated. Sub-sampling can be more efficient than scanning the entire local database – e.g., by *block-level sampling* in which only a small number of disk blocks are retrieved. If the data in the disk blocks are highly correlated, it will simply mean that the number of peers to be visited will increase, as determined by our cross-validation approach at query time.

Third, we estimate the cross-validation error of the collected sample, and use that to estimate the additional number of peers that need to be visited in the second phase. For improving robustness, steps 2-4 in the cross-validation procedure can be repeated a few times and the average squared CVError computed.

Once the first phase has completed, the second phase is then straightforward – we simply initiate a second random walk based on the recommendations of the first phase, and compute the final aggregate.

Although the algorithm has been presented for the case of COUNT, it can be easily extended for SUM. Finally, we re-emphasize that for more complex aggregates, such as estimation of medians, quantiles, and distinct values, more sophisticated algorithms are required. This is part of ongoing work, and we mention some preliminary results in the experimental section.

5. Experimental Evaluation

In this section, we have provided experimental justification for our methods. We have implemented our

algorithms on simulated and real-world topologies using various degrees of data clustering and topology structures.

5.1. Implementation

Our algorithms and peer-to-peer topologies are implemented in Java 5.0 with the graph generation tool Jung [14] version 1.6. Our implementation includes both sampled and real-world Gnutella topology samples. All of our experiments were run on AMD dual Opteron 2.0 GHz processors with 2GB of RAM.

5.2. Generation of P2P Networks and Databases

5.2.1. P2P Networks

Synthetic Topology: The power-laws [11] offer insight to the structure of Internet topologies; and [2] confirms that the power-laws extend to peer-to-peer networks. Our synthetic topology is created through the process of connecting sub-graphs using the graph generation tool Jung [14]. It consists of 10,000 peers and 100,000 edges. The parameters during graph creation are:

- **Sub-graphs [*s*]:** *s* sub-graphs are created that follow the power-laws topology [11].
- **Edges between sub-graphs [*e*]:** The size of *e* determines the cut size between sub-graphs. As the cut size decreases, number of edges between sub-graphs decreases.

Real-World Topology: We also experimented with 2001 Gnutella topology data containing 22,556 peers and 52,321 edges, acquired from the group of M. Ripeanu at the University of Chicago.

5.2.2. P2P Databases

Both types of networks were populated with data generated by a synthetic data generator. We use single attribute tuples. The attribute values have a range between 1 and 100. The values follow the Zipf-distribution. The parameters that define the main characteristics of our synthetic data sets are as follows:

- **Cluster Level [CL]:** If the cluster level is equal to 0, then the dataset is perfectly clustered, i.e., it is sorted and then partitioned across the peers. If the cluster level is set to 1, then the dataset is randomly permuted, then partitioned across the peers. In-between values correspond to in-between scenarios.
- **Skew [Z]:** The skew determines the slant in frequency distribution of distinct values the data. Low skew values give the dataset an even distribution of frequencies per value, conversely high skew values distort the distribution of frequencies.

We populated the synthetic network with 1,000,000 tuples and the Gnutella network with 2,200,000 tuples. It is well-known that peer-to-peer databases have strong clustering properties, e.g., large networks such as Gnutella contain sub-graphs of peers, containing similar music genre, movies, software, or documents [21]. Thus, while populating the peers of both networks, we distributed the data in a breadth-first method, in order to obtain reasonable clustering of synthetic data within the topologies. I.e., when loading a peer, the adjacent peers are also loaded with similarly clustered data.

5.2.3. Aggregation Queries

In our experiments we use SUM and COUNT range queries with different selectivity of the form: "SELECT COUNT(A) FROM T WHERE A BETWEEN A1 AND A2" (i.e. find the number of tuples with values in the range [A1, A2]).

5.3. Input Parameters

We evaluate the accuracy, use of network resources, the size of sample acquired, and total number of tuples sampled from the network. We define each of the user defined inputs as follows:

1. **Required Accuracy [Δ_{req}]:** This parameter defines is the maximum allowed error for the estimated answer.
2. **Tuples Sampled per Peer [*t*]:** This parameter defines the number of tuples to sample from each selected peer.
3. **Jump Size [*j*]:** This parameter defines the Number of peers to pass over before selecting the next peer for sampling.
4. **Initial Sample Size [r_{orig}]**: This parameter defines the initial number of tuples to acquire from the database to execute the first phase. (Thus, r_{orig} / t = *m* where *m* is the number of peers visited in the first phase. In our experiments, the local databases are always large enough to ensure that sub-sampling always takes place.)

Parameter 1 is provided by the user for each query. Parameters 2-4 may be provided by the user, or may be set via a pre-processing step. In the end of the experimental section we provide a user guide for setting parameters 2-4.

5.4. Evaluation Metrics – Cost and Accuracy

Our algorithms are evaluated based on the cost of execution as well as how close they get to the desired accuracy. As discussed earlier, we use latency as a measure of our cost, noting that in our case that it is proportional to the number of peers visited. In fact, if the number of tuples to be sampled is the same for all peers - which is true in our experiments - latency is also proportional to the total number of sample tuples drawn by the overall algorithm. Thus we use the number of

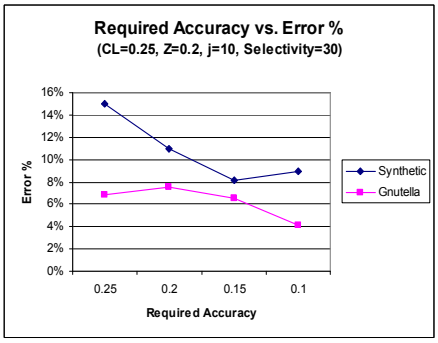


Figure 2: Effects of required accuracy on the error percentage for the *COUNT* technique

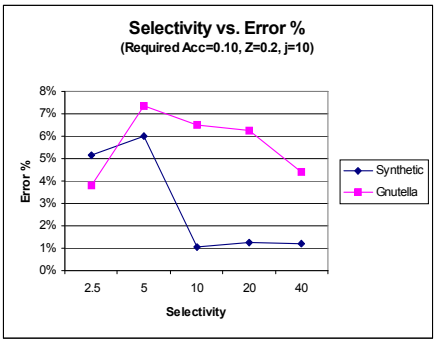
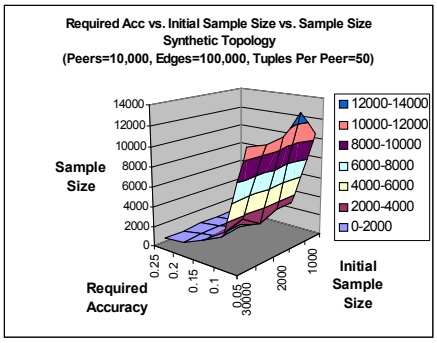


Figure 3: Effects of selectivity on the error percentage for the *COUNT* technique



Figures 4: Effects of the sample size collected for given required accuracies and initial sample sizes for the *COUNT* technique

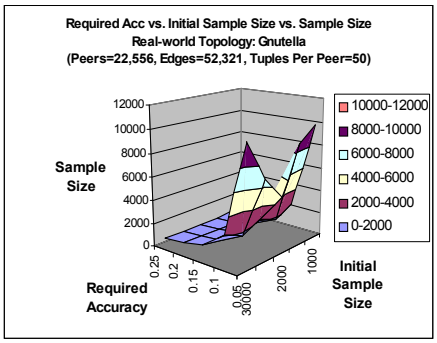


Figure 5: Effects of the sample size collected for given required accuracies and initial sample sizes for the *COUNT* technique

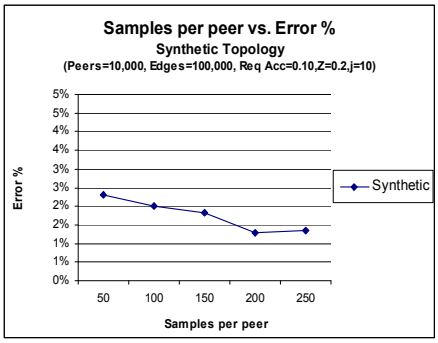


Figure 6: The figure shows the number of peers does not make a vast difference in accuracy

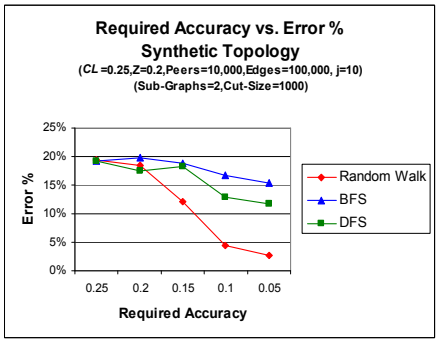


Figure 7: The figure shows random walks perform better then BFS and DFS

sample tuples used as a surrogate for latency in describing our results.

5.5. Experiments

All of our results were generated from five independent experiments and averaged for each individual parameter configurations. Errors are normalized between 0 and 1.

Accuracy: Figure 2 and 3 shows representative accuracy results for COUNT using synthetic and real datasets. In this case we have a query with selectivity 30%, CL=0.2, and Z=0.2. In Figure 2 we vary the required accuracy. The figure shows that the algorithm's result is always within the required accuracy. In Figure 3 we set required accuracy to 0.1 and show the resulting accuracy for each query with different selectivity's.

Sample Size: Figures 4 and 5 show that the required sample size increases with $1/\Delta^2_{req}$. They also, show that the required sample size does not vary much when the initial sample is ranged from 1000 to 3000. The selectivity of the query in this experiment was 30%, and the algorithm gave an answer within the required accuracy in all cases. We note that the result of our algorithm specifies the number of peers to be sampled. In the experiments we convert it to the number of samples by taking 25 samples per peer. Figure 6 shows that the improvement by getting more tuples per peer is small. To minimize the cost of sampling in each peer we take 25 samples in each peer.

Comparison with naïve techniques: Figure 7 compares our approach with DFS, where we collect our sample using a random walk with j=0, and BFS, where we collect our sample from the peers in the neighborhood of the querying peer. Note that our method always meets the required accuracy. Our technique clearly outperforms both techniques.

Effects of data clustering and skew: Figures 8, 9, 10, and 11 show the effects of different degrees of data clustering (8, 9) and different degrees of skew (10, 11). Figures 7 to 12 simulate a peer-to-peer database with two sub-graphs, each containing similar data within individual sub-graphs but different from others. The results show that with clustering closer to 0 (data are more clustered) we need to collect more samples, while with clustering close to 1 (data are less clustered) we need less samples; since each peer contains a better sample of the entire dataset. Regarding skew, the results show that when skew increases, we need fewer samples. The reason is that some values become much more frequent in the dataset and therefore easier to estimate their count.

Figure 8: Effects of clustering on the error percentage for the *COUNT* technique

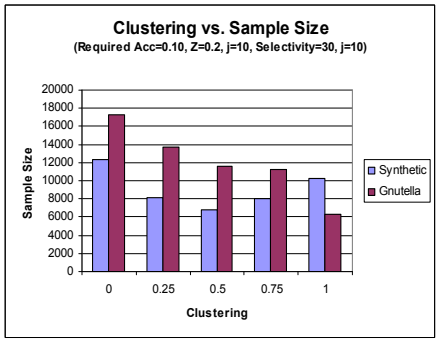


Figure 9: Effects of clustering on the sample size for the *COUNT* technique

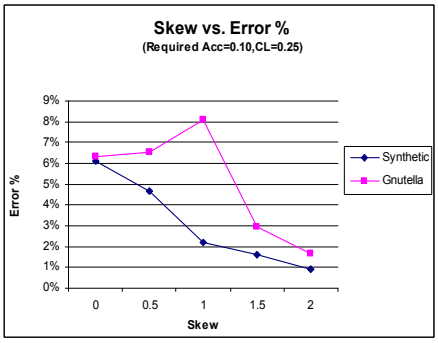


Figure 10: Effects of skew on the error percentage for the *COUNT* technique

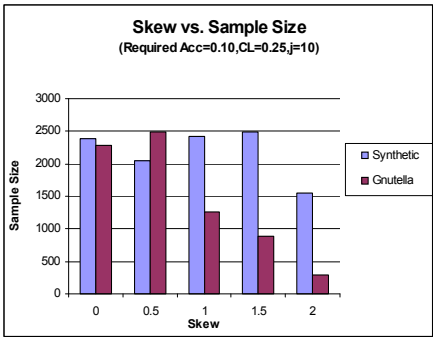


Figure 11: Effects of skew on the sample size for the *COUNT* technique

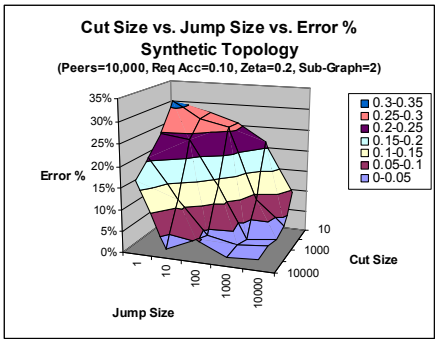


Figure 12: Effects of cut size with jump size on error percentage for *SUM* technique

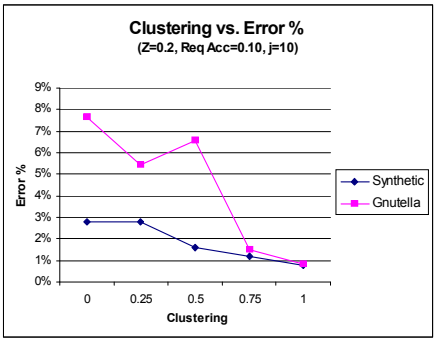


Figure 13: Effects of clustering on the error percentage for the *SUM* technique

Graph size vs. jump size: Figure 12 illustrate the relationship between jump size and size of cuts in a peer-to-peer database. As the number of edges connecting sub-graphs or the jump size increase, the accuracy of the sample increase. The relationship between number of edges connecting sub-graphs and the jump size are inversely proportional in determining the quality of the sample acquired.

Evaluating the SUM query: Figures 13 and 14 show that our technique shows similar accuracy results for SUM. Here we estimate the SUM of all tuples in the database. (i.e. selectivity=1).

5.6. Estimating the Median

Figure 15 and 16 shows that our technique can be extended to accurately estimate the median. Our algorithm for computing the median is given below:

1. Select m peers at random.
2. Each peer computes its median and sends it to the sink.
3. The sink randomly partitions the m medians into two groups of $m/2$ medians.
4. Find the median of the first group.
5. Find the rank i of this median in the second group, let $C = (i - m/4)$
6. Select additional c^2/Δ_{req}^2 peers using random walk.
7. Find and return the median of the medians of the additional peers.

In these experiments we use both the Gnutella and synthetic graph, vary the clustering factor, and set $\Delta_{req} = 0.1$. The error that we show in the graph is the difference of the true rank of the median that the algorithm returns and $N/2$.

6. Conclusion & Future Work

In this paper we present adaptive sampling-based techniques for the novel problem of approximate answering of ad-hoc aggregation queries in P2P databases. We present extensive experimental evaluations to demonstrate the feasibility of our solutions.

Several intriguing open problems remain. Is it possible to build hybrid solutions that do some amount of pre-computations of samples, in addition to "on-the-fly" sampling such as ours? Is it possible for sampling-based algorithms to perform "biased sampling", i.e., focus the samples from regions of the database where tuples that satisfy the query are likely to exist? More generally, decision support and data analysis in P2P databases

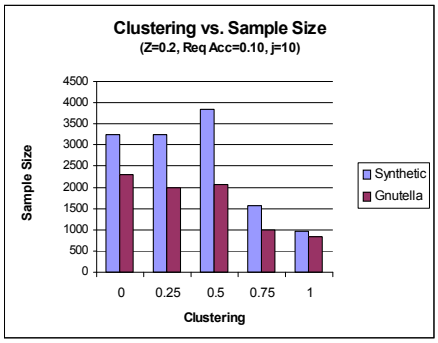


Figure 14: Effects of clustering on the sample size for the *SUM* technique

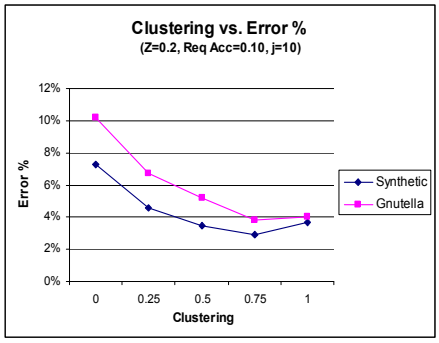


Figure 15: Effects of clustering on the error percentage for the *median* technique

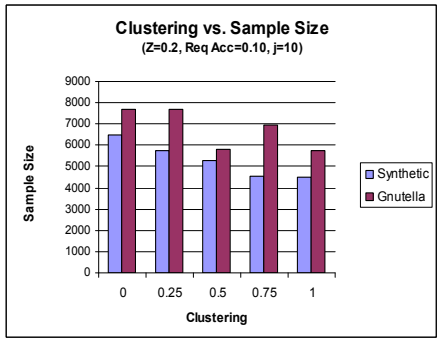


Figure 16: Effects of clustering on the sample size for the *median* technique

appears to be an important area of research with emerging applications, and we hope our work will encourage further research in this field.

7. Acknowledgements

Thanks to M. Ripeanu at the University of Chicago for providing us with the Gnutella topologies samples. The work of Kalogeraki and Gunopulos was supported by NSF 0330481.

8. References

[1] S. Acharya, P. B. Gibbons and V. Poosala. *Aqua: A Fast Decision Support System Using Approximate Query Answers*. Demo in Intl. Conf. on Very Large Databases (VLDB '99).

[2] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. *Search in Power-Law Networks*. Phys. Rev. E, 2001.

[3] B. Babcock, S. Chaudhuri, and G. Das. *Dynamic Sample Selection for Approximate Query Processing*. SIGMOD Conference 2003: 539-550.

[4] A.R. Bharambe, M. Agrawal, and S. Seshan. *Mercury: Supporting Scalable Multi-Attribute Range Queries*. SIGCOMM 2004.

[5] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. *Towards estimation error guarantees for distinct values*. In Proceedings of the ACM Symp. On Principles of Database Systems, 2000.

[6] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. *Overcoming Limitations of Sampling for Aggregation Queries*. ICDE 2001: 534-542.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. *Random sampling for histogram construction: How much is enough?* IN Proceedings. Of the 1998 ACM SIGMOD Intl. Conf. on Management of Data, pages 436-447, 1998.

[8] S. Chaudhuri, G. Das, and V. Narasayya. *A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries*. SIGMOD Conference 2001.

[9] S. Chaudhuri, G. Das, and U. Srivastava. *Effective Use of Block-Level Sampling in Statistics Estimation*. SIGMOD 2004.

[10] Mauricio Minuto Espil and Alejandro A. Vaisman. *Aggregate queries in peer-to-peer OLAP*. DOLAP '04.

[11] C. Faloutsos, P. Faloutsos, and M. Faloutsos. *On Power-Law Relationships of the Internet Topology*. SIGCOMM 1999.

[12] Freenet Homepage, http://freenet.sourceforge.net

[13] C. Gkantsidis, M. Mihail, and A. Saberi. *Random Walks in Peer-to-Peer Networks*. IEEE Infocom 2004.

[14] Gnutella Homepage, http://rfc-gnutella.sourceforge.net

[15] P. Haas, and C. Kőnig. *A Bi-Level Bernoulli Scheme for Database Sampling*. SIGMOD 2004.

[16] R. Heubsch, J. Hellerstein, N. Lanhan, B. T. Loo, S. Shenker, and I. Stoica. *Querying the Internet with PIER*. VLDB 2003.

[17] JUNG website. http://jung.sourceforge.net.

[18] P. Kalnis, W. S. Ng, B. C. Ooi and D. Papadias and K-L. Tan. *An adaptive peer-to-peer network for distributed caching of OLAP results*. SIGMOD 2002.

[19] KaZaA Homepage, http://www.kazaa.com

[20] V. King and J. Saia. *Choosing a Random Peer*. PODC 2004.

[21] F. Le Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulié. *Clustering in Peer-to-Peer File Sharing Workloads*. 3rd Intl. Workshop on Peer-to-Peer Systems IPTPS 2004.

[22] X. Li, Y.J. Kim, R. Govindan, and W. Hong. *Multi-dimensional range queries in sensor networks*. SENSYS 2003.

[23] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. *Peer-to-Peer Computing*. HP Technical Report, *HPL-2002-57*.

[24] Napster Hompage, http://www.napster.com

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A Scalable Content-Addressable Network*. SIGCOMM 2001.

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

[27] O.D. Sahin, A. Gupta, D. Aggrawal, and A. El Abbadi. *A Peer-to-peer Framework for Caching Range Queries*. ICDE 2004.

[28] Julian L. Simon. Resampling: *The New Statistics*. Second Edition published October 1997.

[29] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. *Chord: A scalable Peer-to-peer Lookup Service for Internet Applications*. SIGCOMM 2001.

[30] D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos. *Exploiting locality for scalable information retrieval in peer-to-peer networks*. Inf. Syst. 30(4): 277-298 (2005).