Counting at Large: Efficient Cardinality Estimation in Internet-Scale Data Networks

Nikos Ntarmos
R.A.C.T.I. and C.E.I.D.
University of Patras
Rio, Greece
ntarmos@ceid.upatras.gr

Peter Triantafillou
R.A.C.T.I. and C.E.I.D.
University of Patras
Rio, Greece
peter@ceid.upatras.gr

Gerhard Weikum
M.P.I.I.
Saarbrücken, Germany
weikum@mpi-sb.mpg.de

Abstract

Counting in general, and estimating the cardinality of (multi-) sets in particular, is highly desirable for a large variety of applications, representing a foundational block for the efficient deployment and access of emerging internet-scale information systems. Examples of such applications range from optimizing query access plans in internet-scale databases, to evaluating the significance (rank/score) of various data items in information retrieval applications. The key constraints that any acceptable solution must satisfy are: (i) efficiency: the number of nodes that need be contacted for counting purposes must be small in order to enjoy small latency and bandwidth requirements; (ii) scalability, seemingly contradicting the efficiency goal: arbitrarily large numbers of nodes nay need to add elements to a (multi-) set, which dictates the need for a highly distributed solution, avoiding server-based scalability, bottleneck, and availability problems; (iii) access and storage load balancing: counting and related overhead chores should be distributed fairly to the nodes of the network; (iv) accuracy: tunable, robust (in the presence of dynamics and failures) and highly accurate cardinality estimation; (v) simplicity and ease of integration: special, solution-specific indexing structures should be avoided. In this paper, first we contribute a highly-distributed, scalable, efficient, and accurate (multi-) set cardinality estimator. Subsequently, we show how to use our solution to build and maintain histograms, which have been a basic building block for query optimization for centralized databases, facilitating their porting into the realm of internet-scale data networks.

1 Introduction

Peer-to-peer (P2P) networks came into existence as a means of sharing files and/or CPU cycles among end-users. Over time, they evolved from the anarchy of the early small-world architectures to the cutting-edge structured data networks of today. The main advance that made this feasible was the introduction of the Distributed Hash Tables (or DHTs)[10, 26, 31, 34]. The common denominator of all these systems is their ability to scale to large numbers of nodes and to manage an even larger amount of data objects, while providing probabilistic (under node failures and skewed data/access distributions) guarantees for the attained degree of efficiency, fault tolerance, and availability.

As a natural evolution of the widespread adoption of P2P technologies by end-users and the enterprise alike, and the much desirable properties of structured P2P overlays, the academic community has started considering the possibility of using such networks as the substrate for widely distributed database and data integration systems[15, 16, 18, 19, 23, 28, 30]. Thus, peer-to-peer networks have departed from their file/CPU-sharing origins and are rapidly evolving into a powerful infrastructure, capable of supporting data management systems of huge scale.

Motivation

In this new era of internet-scale peer-to-peer data networks, the need for a distributed counting mechanism arises in many occasions. More often than not, the quantity to be counted contains duplicates and the candidate algorithm must provide duplicate insensitivity: file-sharing peer-to-peer systems often need to know the total number of (unique) documents shared by their users; widely distributed peer-to-peer search engines need a method to evaluate the significance of various keywords, expressed as the ratio of the number of unique indexed documents containing each keyword to the total number of unique indexed documents; conversely, internet-scale information retrieval systems need a method to deduce the rank/score of various data items; sensor networks need methods to compute aggregates in a duplicate-insensitive manner since multiple sensors may be sensing and reporting the same event; internet-scale database systems can harness such distributed counting mechanisms to build histograms over stored/shared data, en route to selectivity estimation and optimization algorithms for query access plans; etc.

Especially for the latter case, traditionally query optimizers heavily rely on histograms over stored data, in order to estimate the size of intermediate results and the cost of candidate access plans, en route to choosing the optimal query evaluation strategy[6, 20, 33]. The issue of minimizing the size of the intermediate result-sets is further aggravated when moving to the peer-to-peer world.

Furthermore, with wide distribution comes the need for completely decentralized methods of performing traditionally centralized operations, and the lack of knowledge with regard to overall/global system properties. As a consequence, computing such metrics (e.g. number of documents in the network, sizes of database relations, distributions of data values) in peer-to-peer systems, in a scalable, efficient, and accurate manner, has long been neglected. The key constraints that any acceptable solution must satisfy are:

1. Efficiency: the number of nodes that need be contacted for counting purposes must be small in order to enjoy small latency and bandwidth requirements;
2. Scalability and availability, seemingly contradicting the efficiency goal: arbitrarily large numbers of nodes nay need to add elements to a (multi-) set, which dictates the need for a highly distributed solution, avoiding server-based scalability, bottleneck, and availability problems;
3. Access and storage load balancing: counting and related overheads should be distributed fairly across all nodes;
4. Accuracy: tunable, robust (in the presence of dynamics and failures) and highly accurate cardinality estimation;
5. Simplicity and ease of integration: special, solution-specific indexing structures should be avoided; and
6. Duplicate (in)sensitivity: the proposed solution must be able to count both the total number of items as well as the number of unique items in multisets, as outlined earlier.

Related Work

Distributed counting/aggregation solutions proposed by the peer-to-peer research corpus so far, can be categorized in one of the following groups:

- One-node-per-counter protocols.
- Gossip-based protocols.
- Broadcast/convergecast-type protocols.
- Sampling-based protocols.

The first type of solutions is also the first that comes to mind when using a structured overlay (DHT): select a node in the overlay (e.g. by using the hash function(s) of the DHT overlay) and use it to maintain the counter value (e.g. see the distributed counting mechanism outlined in [12]). Hash-partitioned counters, where the counting space is partitioned into disjoint intervals, with each such interval mapped to a (set of) node(s) in the overlay, also fall in this category.

Solutions of this type suffer from many shortcomings, more notable of which is their very poor scalability; having one node per counter means that this node will be contacted on every update of, and on every query for, the current value of the counter, resembling more of a centralized system, violating constraint (2). Moreover, each of these counting nodes withstands a high access and storage load, violating constraint (3), while one can argue that such highly-loaded nodes will exhibit high response times, also violating constraint (1). Using a fixed small number of nodes for each counter does not solve the problem. On the other hand, using a large number of nodes for each counter merely mitigates the problem of scalability to the cost of gathering the value of such a counter, while also violating constraint (1).

The second type of solutions [2, 21, 22, 27] usually provide weak probabilistic semantics of "eventual consistency" for their outcome; gossip-based protocols are based on an iterative procedure, according to which every node exchanges information with a (set of) node(s) on every iteration. Eventual consistency means that, in the presence of failures and dynamicity in the P2P overlay, the algorithm will eventually converge to a stable state after the overlay has itself stabilized. Although the bandwidth requirements of these approaches are low when amortized over all nodes, the overall bandwidth consumption and hop-count are usually very high. The multi-round property of these solutions violates constraint (1), while their semantics violate constraint (4).

The third type of solutions [3, 4, 8, 32, 36] is based on a two-round procedure: (i) a *broadcast* phase, during which the querying node broadcasts a query through the network, creating a (virtual) tree of nodes as the query propagates in the overlay; and (ii) a *convergecast* phase, during which each node sends its local part of the answer, along with answers received from nodes deeper down the tree, to its "parent" node. Solutions that are based on pre-built tree structures also belong in this group.

Astrolabe[32] was among the first works to talk of aggregation in the peer-to-peer landscape; the authors proposed the creation and maintenance of a hierarchical, tree-like overlay, used to propagate complex queries and their results through the peer-to-peer overlay. A similar work has been proposed in [36]. [3] propose building a (set of) multicast overlay tree(s) to propagate queries and results back and forth, while using flood-like methods to send messages around the network. Although these structures have nice properties and are capable of computing aggregates in a wide scale, they are not fit for the creation and maintenance of histograms and there seems to be no efficient/easy way in which their functionality could be extended to such tasks as selectivity estimation and histogram construction. One could argue that such solutions are a sort of "directed gossip", since the core functionality is very similar to that of gossip-based algorithms, with the exception that during phase (ii) nodes only exchange information with their parent and children nodes in the (virtual) tree. As of this, these solutions violate constraint (1), while most of them (with the exception of [4]) also violate constraints (3) and (4).

The core idea of the last type of solutions [5, 25] is to estimate the value of the counter in question, by selectively querying (sampling) a set of nodes in the network. [5] attempt to compute approximate histograms of system statistics by using random sampling of nodes in the network. [25] estimate the number of nodes in the overlay by also using a random sampling algorithm. Sampling-based techniques are known to suffer from accuracy issues[7], thus violating constraint (4). On the other hand, if the sample is big enough[7] to guarantee a certain level of confidence then these solutions violate constraint (1). Last but not least, sampling-based techniques are usually duplicate-sensitive, thus violating constraint (6).

As a matter of fact, the duplicate insensitivity constraint seems like the most difficult to abide to. Hash sketches[11, 12], to be presented shortly, provide a distributable, duplicate-insensitive method of estimating the cardinality of (multi-)sets. All known works that manage to provide duplicate-insensitive counting[3, 4, 8] use hash sketches. However, they all fall into the broadcast/convergecast type of counting algorithms, thus having the disadvantages mentioned earlier.

Contributions

With this work we propose Distributed Hash Sketches (or DHS); a novel, fully decentralized mechanism, capable of providing estimates on the cardinality of multi-sets of objects in a peer-to-peer system. DHS is, to our knowledge, the first truly distributed version of hash sketches – a probabilistic counting mechanism, as proposed by either Flajolet and Martin[12] or more recently by Durand and Flajolet[11] – along with the accompanying algorithms, and protocols. Moreover, it is the first distributed counting mechanism satisfying all six constraints presented earlier.

Designing and implementing hash sketches over DHTs in an efficient and scalable way, while maintaining the, implicit in the peer-to-peer world, invariant of completely decentralized operation, is a formidable task. The goals we wish to pursue include: (i) balanced access load, (ii) highly efficient and scalable operation, independent of the number of items and logarithmic in the number of nodes in the overlay, (iii) derivation of bounds on the error added by the distributed operation and examination of its algorithmic implications, (iv) alternative ways to implement hash sketches in a distributed manner, (v) implementation and evaluation of both [12] and [11] within our framework, and (vi) implementation and evaluation of DHS with respect to its estimation error and overhead and with respect to utilizing DHS-based histograms for query optimization.

The proposed design: (i) is DHT-agnostic, in the sense that it can be deployed over any peer-to-peer overlay conforming to the DHT abstraction, (ii) imposes a totally balanced distribution of access load on the DHT nodes, (iii) provides probabilistic guarantees regarding the correctness and accuracy of the produced estimates, (iv) allows for a trade-off between accuracy and cost of maintenance, and (v) incurs low bandwidth, storage, and processing overheads, when used for counting the cardinality of widely distributed item (multi-)sets. We implemented and evaluated DHS, using both PCSA[12] and super-LogLog[11] estimators. Finally, we show how using DHS as infrastructure, we can build and maintain efficient histogram-based selectivity estimators for widely distributed data.

2 Background

2.1 Distributed Hash Tables

Distributed Hash Tables are a family of structured peer-to-peer network overlays exposing a hash-table-like interface. The main advantage of DHTs over unstructured P2P networks, lies in the probabilistic (in the presence of node failures and network dynamics) performance guarantees offered by the former. Prominent examples of DHTs include Chord[34], CAN[31], Kademlia[26], Pastry[10], etc.

DHTs offer two basic primitives: *insert(key, value)* and *lookup(key)*. Nodes are assigned unique identifiers and arranged according to a predefined geometry and distance function[14]. This results in a partitioning of the node-ID space among nodes, so that each node is responsible for a well-defined set of identifiers. Each item is also assigned a unique identifier from the same ID space, and is stored at the node responsible for the set containing its ID. Each node in an N-node DHT maintains direct IP links (aka fingers) to $O(\log \mathcal{M})$[1] other nodes in appropriate positions in the overlay ($\mathcal{M}$ is the maximum allowable number of nodes/items in the DHT), so that routing between any two nodes takes $O(\log N)$ hops in the worst-case.

DHTs are highly efficient for point queries; they are designed and optimized for locating single items corresponding to a predefined key. For DHTs to accommodate RDBMS-class applications, support must be added for more rich and complex types of queries. There have been several proposals in the direction of supporting RDBMS functionality over P2P infrastructures in the last few years[1, 15, 18, 30, 32, 35, 36]. The main focus of these systems is on simple select-project-join (SPJ) or single-attribute range queries, mainly due to performance limitations; without a distributed query optimization mechanism, the efficiency of multi-attribute and multi-join queries deteriorates rapidly with the number of attributes/joined relations.

2.2 Hash Sketches

Hash sketches were first proposed by Flajolet and Martin[12] (coined *Probabilistic Counting with Stochastic Averaging* or *PCSA*), as a means of estimating the cardinality of a multiset $\mathcal{D}$ of data in a database (i.e. the number of distinct items in the multiset). The estimate obtained is (virtually) unbiased, while the authors also provide upper bounds on its standard deviation. The only assumption underlying hash sketches is the existence of a pseudo-uniform hash function $h() : \mathcal{D} \rightarrow [0, 1, \ldots, 2^L)$ – an assumption also present in most (if not all) P2P-related research. Durand and Flajolet presented a similar algorithm[11] (*super-LogLog counting*) which reduced the space complexity and relaxed the assumptions on the statistical properties of the hash function of [12][2]. Hash sketches have been used in many application domains where counting distinct elements in multi-sets is of some importance, such as approximate query answering in very large databases[24], data mining on the internet graph[29], and stream processing[9, 13].

2.2.1 *Super-LogLog* counting

Let $\rho(y) : [0, 2^L) \rightarrow [0, L)$ be the position of the least significant (leftmost) 1-bit in the binary representation of y;

[1]All $\log(\cdot)$ notation refers to base-2 logarithms.

[2]The analysis leading to the equations used in this section is well beyond the scope of this paper. Interested readers are referred to [12, 11].

that is, $\rho(y) = min_{k\geq 0}\, bit(y,k) \neq 0,\ y > 0$, and $\rho(0) = L$. $bit(y,k)$ denotes the kth bit in the binary representation of y (bit-position 0 corresponds to the least significant bit).

In order to estimate the number n of distinct elements in a multiset $\mathcal{D}$ we apply $\rho(h(d))$ to all $d \in \mathcal{D}$ and record the results in a bitmap vector $B[0 \ldots L-1]$. Since $h()$ distributes values uniformly over $[0, 2^L)$, it follows that

$$P(\rho(h(d)) = k) = 2^{-k-1} \tag{1}$$

Thus, when counting elements in an n-item multiset, $B[0]$ will be set to 1 approximately $\frac{n}{2}$ times, $B[1]$ approximately $\frac{n}{4}$ times, etc. This fact is rather intuitive: imagine all n possible L-bit numbers; the least significant bit (bit 0) will be 1 for half of them (odd numbers); of the remaining $\frac{n}{2}$ numbers, half will have bit 1 set, or $\frac{n}{4}$ overall, and so on.

Then, the quantity $R(\mathcal{D}) = max_{d\in\mathcal{D}}\rho(d)$ provides an estimation of the value of $\log n$, with an additive bias of 1.33 and a standard deviation of 1.87. Thus, 2^R estimates "logarithmically" n within 1.87 binary orders of magnitude. However, the expectation of 2^R is infinite and, thus cannot be used to estimate n. To this extent, [11] propose the following technique (similar to the *stochastic averaging* technique in [12]): (i) use a set of $m = 2^c$ different $B^{\langle i\rangle}[\cdot]$ vectors (also called *buckets*), each resulting to a different $R^{\langle i\rangle}$ estimate, (ii) for each element d, select one of these using the first c bits of $h(d)$, and (iii) update the selected vector and compute $R^{\langle i\rangle}$ using the remaining bits of $h(d)$.

If $M^{\langle i\rangle}$ is the (random) value of the parameter R for vector i, then the arithmetic mean $\frac{1}{m}\sum_{i=1}^{m} M^{\langle i\rangle}$ is expected to approximate $\log\frac{n}{m}$ plus an additive bias. The estimate of n is then computed by the formula: $E(n) = \alpha_m \cdot m \cdot 2^{\frac{1}{m}\cdot\sum_{i=1}^{m} M^{\langle i\rangle}}$, where the constant α_m is computed by ([11]): $\alpha_m = (-m \cdot \frac{2^{-\frac{1}{m}}-1}{\log 2} \cdot \int_0^\infty e^{-t} \cdot t^{-\frac{1}{m}} dt)^{-m}$.

The authors further propose a *truncation rule*, consisting of taking into account only the $m_0 = \lfloor \theta_0 \cdot m \rfloor$ smallest M values. θ_0 is a real number between 0 and 1, with $\theta_0 = 0.7$ producing near-optimal results. With this modification, the estimate formula becomes:

$$E(n) = \tilde{\alpha}_m \cdot m_0 \cdot 2^{\frac{1}{m_0}\cdot\sum^{*} M^{\langle i\rangle}}, \tag{2}$$

where $\sum^{*}$ indicates the truncated sum, and the modified constant $\tilde{\alpha}_m$ ensures that the estimate remains unbiased. The resulting estimate has a standard deviation of $\frac{1.05}{\sqrt{m}}$, while the hash function must have a length of at least

$$H_0 = \log m + \lceil \log\left(\frac{n_{max}}{m}\right) + 3 \rceil, \tag{3}$$

n_{max} being the maximum cardinality estimated.

2.2.2 *PCSA* counting

The algorithm in [12] is based on the same hashing scheme (i.e. using $\rho(\cdot)$) and the same observations (i.e. eq. 1) as [11]. The PCSA algorithm differs from the super-LogLog algorithm in the following: (i) [12] rely on the existence of an explicit family of hash functions exhibiting ideal random properties, while [11] have relaxed this assumption, (ii) [12] set R to be the position of the leftmost 0-bit in the bitmap $B[\cdot]$, as opposed to the position of the rightmost 1-bit in the bitmap with [11], (iii) [12] use $\log(max\ cardinality)$ bits per bitmap, while [11] need in the order of $\log\log(max\ cardinality)$ bits per bitmap, (iv) the estimation in [12] is computed as:

$$E(n) = \frac{1}{0.77351} \cdot m \cdot 2^{\frac{1}{m}\sum_0^{m-1} M^{\langle i\rangle}}. \tag{4}$$

and (v) the bias and standard error of [12] are closely approximated by $1+0.31/m$ and $0.78/\sqrt{m}$ respectively. Note that data insertion is the same for both algorithms (with the sole difference of the assumptions on the hash function).

Symbol	Quantity
b	size of a data item
N	number of nodes
n	number of items
$\mathcal{M}$	maximum number of items/nodes
L	length (in bits) of DHT keys ($= \log\mathcal{M}$)
k	length of DHS bitmaps/keys ($\leq L$)
m	number of DHS bitmaps
l	number of DHS dimensions/metrics
$\mathcal{R}$	degree of replication of DHS data

Table 1. Notation Summary

3 DHS: Distributed Hash Sketches

Table 1 summarizes the notation we shall be using for the rest of this paper. DHTs already feature a pseudo-uniform hash function; object (node/document) IDs are (usually) computed as either the secure hash of some object-specific piece of information[34, 10] (e.g. the IP address and port of nodes, the content for files, etc.), or as the outcome of a pseudo-uniform random number generator[26][3]. In both cases, the resulting ID is an L-bit pseudo-uniform number (for some fixed, system-specific L), thus satisfying the main assumption of hash sketches.

We note here that an $L = 160$ (as is the case in many current DHTs) is too long a bit vector for any application, since the bit vector must only be (at most) as long as the base-2 logarithm of the estimated metric, give-or-take a few bits (see eq. 3 and [12]). We denote by $k \leq L$ the length of the DHS bitmap vectors and assume that items are added to the DHS using the k lower-order bits of their corresponding DHT keys. The minimum value of k is dictated by eq. 3[4].

[3]A PRNG can be used as a hash function, by using the hash function input value as the seed to the PRNG, and (part of) the random sequence produced as the hash function output.

[4]The Birthday Paradox limits the number of items in a DHT namespace to 2^{80} with 160-bit keys, while cryptographic invariants related to the hash functions used may further limit the number of "useful" bits; thus, in the absence of a notion of "maximum cardinality", 80 bits seem like a good (maximum) value for the length of DHS keys.

A naive DHT-based implementation would assign each of the k positions of the $B[\cdot]$ vector to a node in the network and use these nodes to store bit values in a distributed manner. However, this design has many serious flaws: (i) only $k \leq L$ out of 2^L (maximum) nodes in the network are burdened with the task of maintaining the values of the vector positions, leading to a severe load imbalance for these nodes; (ii) due to eq. 1, there is a severe load imbalance even among these very nodes; and (iii) with (maximum) 2^L objects spread over k nodes, the node join/leave operations for any of these nodes would result in moving around information for $\frac{2^L}{k} \geq 2^{L-1}$ objects – a prohibitive cost, regardless of the size of the data maintained per object.

We have implemented and evaluated both [12] (DHS-PCSA) and [11] (DHS-sLL) within DHS. As noted earlier, the data insertion procedure is the same for both algorithms. We'll first discuss the PCSA case when $m = 1$ (i.e. hash sketches implemented using one $B[\cdot]$ vector only), extending our design for multiple vectors and super-LogLog later.

3.1 Mapping DHS bits to DHT nodes

We partition the node ID space, $[0, 2^L)$, into k consecutive, non-overlapping intervals $\mathcal{I}_r = [thr(r), thr(r-1))$, $r \in [0, k)$, where: $thr(r) = 2^{L-r-1}$. Using this partitioning, bit r of $B[\cdot]$ is mapped to node IDs randomly (uniformly) chosen from $\mathcal{I}_r$ (bit k is mapped to the interval $[0, thr(k-1))$).

Remember (eq. 1) that when counting distinct items in an n-object multiset, bit r of the bitmap vector is "visited" $n \cdot 2^{-r-1}$ times. With the k-bit IDs used in DHS, this translates to a maximum of 2^k distinct objects in any possible multiset, or to a maximum of 2^{k-r-1} objects being mapped to position r in the bitmap vector. Now, note that intervals $\mathcal{I}_r$ have exponentially decreasing sizes $|\mathcal{I}_r| = 2^{L-r-1}$. The above result in a distribution of information across all nodes in the network, as uniform as the hash function used.

3.2 DHS Insertion

For an object o with ID $o.id$ to be recorded in the DHS, we need to compute $r = \rho(lsb_k(o.id))$, where $lsb_k(\cdot)$ returns the k lower-order bits of its argument, and store an appropriate tuple on the underlying DHT using a key uniformly chosen from the interval $[thr(r), thr(r-1))$.

Each DHS tuple is of the form <$metric\_id$, bit, $time\_out$>, where $metric\_id$ is an identifier uniquely identifying the metric to be estimated, $bit = r$ denotes the position in the distributed vector of the bit that is to be set, and $time\_out$ defines a time-to-live interval for the current tuple, reset at every updates of the tuple, allowing for aging out of DHS entries. Estimated metrics may range from basic network parameters as the cardinality of the node population or the number of distinct data objects shared in a P2P overlay, to more elaborate quantities such as the cardinality of relations in an RDBMS-like P2P setting, or the number of tuples/object satisfying some predefined condition, etc. Unless stated otherwise, we will assume that there is only one metric estimated in the overlay; counting multiple metrics at once (also called "multi-dimensional counting") will be discussed in sect. 4.2.

Moreover, if a node desires to record multiple items in the DHS, it can first compute the r values for these items, group the results by r, and perform a "bulk" insertion of all of each items. Thus, every node will need to contact at most $k \leq L = \log \mathcal{M}$ nodes in order to record *all* of its items in the DHS. Obviously, the node may choose a different set of k nodes on each update round, as dictated by the ID-space partitioning outlined earlier in this section and the random selection of target nodes in these intervals.

Cost Analysis

The hop-count cost to insert an object in an N-node DHT/DHS is in $O(\log N)$, as guaranteed by the underlying DHT, translating to an overall $O(b \cdot \log N)$ bandwidth consumption if the size of the datum stored is b bytes. As far as storage overhead is concerned, note that each node will store information for at most one DHS bit; if multiple items set the bit stored on a given node, the storing node will only maintain data for one bit and update its timestamp field accordingly. Thus, the storage overhead per node per metric is in $O(b)$.

We shall quantify the above with the following simple example: assuming we use $160 (= L = \log N)$ bits for the $metric\_id$, 5 bits for the *bit* (i.e. DHS keys are $k = 2^5 = 32$ bits long, counting up to $2^{32} \approx 4$ billion items), and 32 bits for the $time\_out$ field, a DHS-based estimator would require on average ~ 200 bits, or (b =)25 bytes, per node per estimated metric, in the worst case!

Further note that, compared to the cost of actually inserting a data item in the DHT, the cost of a DHS insertion is negligible. Inserting an item in a DHT requires $O(\log N)$ hops by default, but requires a more-or-less large data transfer – should that be due to transferring of the whole object inserted or just of a (set of) index tuple(s); on the other hand, setting a bit in the DHS can be as cheap as a PING-PONG/heartbeat message, and could actually be piggy-backed on such DHT-related messages.

3.3 DHS Deletion and Maintenance

Deletion of data stored in a DHS is implicit, following a soft-state approach. Remember that a time-to-live value is stored on the DHT/DHS along with every piece of information; data items are then deleted if not updated within this time period, so deleting an item incurs no extra cost.

The computation of this $time\_out$ field poses an interesting trade-off. Larger time-out values will result in less updates per time unit needed to keep the DHS up-to-date. On the other hand, a smaller value will allow for faster adaptation to abrupt fluctuations in the value of the metric estimated, but will incur a higher maintenance cost as far as (primarily) network resources are concerned. However, we have to point out once again that the per-node bandwidth and storage requirements of DHS are very low, thus even a high update rate might translate to a negligible bandwidth consumption.

3.4 Increasing DHS Accuracy

As mentioned earlier, the accuracy of estimations of hash sketches improves with multiple bitmap vectors. Extending the above algorithms for $m > 1$ (m being a power of 2) is straightforward. Insertion of an item o with an ID of $o.id$ is done by selecting one out of m vectors using $lsb_k(o.id) \; mod \; m$, and then using $r = \rho(lsb_k(o.id) \; div \; m)$ as the position of the bit to be set.

In this case, item insertion is performed in the exact same manner as in the single-bitmap case, only with m and r computed as above. The DHS tuple data must now be extended to $< metric\_id, vector\_id, bit, time\_out >$, where $vector\_id$ is the ID of the vector being updated. With another 10 bits for the $vector\_id$ field (i.e. maximum 1024 bitmaps), the DHS tuple size increases from 25 to 26 bytes, while the per-metric update cost increases to 4160 bytes per update round – still a negligible quantity.

The worst-case hop-count cost and bandwidth consumption for a node to insert an item in such a DHS are again in $O(k \cdot \log N)$ and $O(b \cdot k \cdot \log N)$ respectively, while the worst-case per-node storage overhead now becomes $O(m \cdot b)$. Note that the hop-count cost and bandwidth consumption are independent of the number of bitmaps, since insertions/updates touch a single bitmap on every insertion/update.

3.5 DHS Fault Tolerance and Robustness

For a chosen degree of replication of DHS data, the probability of not being able to locate DHS information for some bit can be made arbitrarily small. For example, with $\log N$ replicas, if p_f is the probability of any node in the system failing, the probability of missing DHS bit information is equal to $p_f^{\log N}$, which for any practical purpose is adequately small. Assuming for example $p_f = 0.10$ and with $N = 1024$, we get a DHS fault probability equal to 10^{-10}. Thus, in practice, the replication degree $\mathcal{R}$ will be smaller than $\log N$.

At this stage we have a number of options: we can either rely on the underlying vanilla replication functionality, offered now by most DHTs, or we can implement such a replication strategy of our own. An appealing option for a the latter case is: when inserting (or refreshing) a DHS bit, accessing a particular node in the bit's DHT interval, replicate this set bit to a number of $\mathcal{R}$ successors/predecessors of this node. Given a node failure observed during counting, the counting algorithm can visit a number of successor nodes (up to $\mathcal{R}$) until the missing information is found. This replication strategy incurs an additional absolute cost of at most $\mathcal{R}$ hops for insertion and refresh operations ($O(\log N)$ total hops per insertion, with constant $\mathcal{R}$).

Note that such replication is only needed for the DHS bits stored in the smaller DHT intervals, since DHS bits in the larger intervals already have better fault tolerance, given that a larger number of nodes are responsible for storing the corresponding DHS bit. Alternative techniques for ensuring fault tolerance and counting robustness could try to leverage this built-in fault tolerance feature of DHS. For example, slightly changing the way DHS bits are mapped to DHT intervals, by disregarding the first few least significant bits of each item being inserted. If the first b bits are disregarded, this has the effect of assigning the i^{th} DHT interval to the $(i+b)^{th}$ bit. The assumption that only sizes beyond some threshold given by 2^b are being measured with DHS, (e.g., with $b = 10$, greater than a one thousand) is certainly justified. This ensures that more DHS bits are assigned to larger DHT intervals. This approach for fault tolerance comes at no extra 'replication' cost during insertions and refreshes. The above sketches the fault tolerance possibilities with DHS. Detailed investigation of the various approaches and their trade-offs is beyond the scope of this paper.

Algorithm 1 Estimate the number of distinct elements in a multiset using a DHS

```
1: $R[0,\ldots,m-1] = \{-1,\ldots,-1\}$; $all\_bitmaps\_set$ = **false**;
2: **for all** bit positions $r = L-1,\ldots,0$ **and** $all\_bitmaps\_set ==$ **false do**
3:    Select a random node ID $id \in [thr(r), thr(r-1))$;
4:    $target = id$; $go\_to\_succ = true$; $counter = 0$;
5:    Compute $lim$ for position $r$ ($lim = 5$ by default);
6:    Execute: DHT_lookup($id$);
7:    **while** $all\_bitmaps\_set ==$ **false and** $counter < lim$ **do**
8:       $counter = counter + 1$;
9:       **for all** $j = \{0,\ldots,m-1\}$ **do**
10:         **if** $R[j] == 0$ **and** bit $r$ is set for bitmap $j$ at $target$ **then**
11:            $R[j] = r$;
12:      **if** all $R[\cdot] \geq 0$ **then**
13:         $all\_bitmaps\_set$ = **true**;
14:      **else if** $go\_to\_succ$ == **true and** $id < thr(r-1)$ **then**
15:         $target = target.successor$;
16:      **else**
17:         $target = id.predecessor$; $go\_to\_succ$ = **false**;
18: **return** $E(n) = \tilde{\alpha}_m \cdot m_0 \cdot 2^{\frac{1}{m_0} \cdot \sum^* R[\cdot]}$
```

4 Counting with DHS

Remember that estimating the number of distinct items in a multiset using hash sketches consists of (i) finding the (truncated) arithmetic mean of the positions $R^{\langle i \rangle}$ of the rightmost 1-bit for super-LogLog, or of the leftmost 0-bit for PCSA, in $B[\cdot]$, and (ii) using eq. 2 (super-LogLog) or eq. 4 (PCSA) to compute an estimate of the cardinality of the multiset in question. Applying this algorithm in the DHS setting is rather straightforward: what we need to do is visit each of the intervals corresponding to the various bit positions of the m distributed $B^{\langle i \rangle}[\cdot]$ bitmap vectors and check whether there is any object recorded there. Alg. 1 outlines the algorithm with super-LogLog counting (the algorithm for PCSA is omitted due to space limitations).

Since every bit position of the bitmaps is uniformly mapped to an interval on the node ID space, we may have to visit multiple nodes in every interval until we find one storing information for an object (corresponding to the bit being set). The DHS counting algorithm first selects a random node in every ID-space interval and probes it for any relevant tuple. If no such information is available at that node, the algorithm proceeds by visiting the target node's immediate successors/predecessors within the specific ID-space interval until either some tuple is located or an upper limit of such retries is reached.

This iterative phase exists to compensate for the following issue: when recording i items in an ID-space interval mapping to i or more nodes, then there will exist nodes which will store no relevant information; even when the target interval consists of less than i nodes, some of them may store no DHS-related information, due to the randomness in choosing the target nodes (both when storing and when retrieving DHS information). For our algorithms this means that when we randomly visit a node holding a DHS bit, if it is zero, we are still not certain, so we have to retry until we find a set bit. The question is how many times before we stop, while with a controllable probability we do not err.

4.1 Errors and Retries

Errors in the estimate returned by the DHS counting algorithm are caused by: (i) statistical deviation on behalf of the underlying hash sketch theory, and (ii) bits not being set during the node probe step in the counting algorithm.

As far as hash sketches are concerned, [12, 11] feature a rigorous analysis of their statistical properties. Reciting the proofs found in these works is surely beyond the scope of this paper. We refer interested readers to [12, 11] and just mention here that the standard deviation is closely approximated by $1.05/\sqrt{m}$ for [11] and by $0.78/\sqrt{m}$ for [12].

We turn now to the computation of the upper limit of nodes to contact per bit position of the DHS bitmap(s). Assume that n' items have been uniformly distributed to N' bins (i.e. mapped to an N'-node *interval* in the DHS). The counting process of the previous section corresponds to uniformly and independently picking a bin from the set of bins without replacement, and checking for whether there is any item stored in it. The probability $P(X = t)$ that t empty bins are selected in the first t probes, equals:

$$P(X = t) = \left(\frac{N' - t}{N'}\right)^{n'}. \qquad (5)$$

Sketch of proof: *When uniformly placing a single item in one of N' bins, the probability of selecting a particular bin is $\frac{1}{N'}$ and the probability of not selecting it is $\frac{N'-1}{N'}$. Thus, after placing n' items, a bin will be empty with probability $\left(\frac{N'-1}{N'}\right)^{n'}$. This also equals the probability of choosing an empty bin at our first probe. Now, the probability of one of the remaining $N' - 1$ bins being empty (and the probability of choosing an empty bin in our second probe), is $\left(\frac{N'-2}{N'-1}\right)^{n'}$, given our first probe resulted in an empty bin being chosen. Note that choosing the next-in-line bin after the one we selected in the previous step is equivalent to choosing one of the $N' - 1$ bins uniformly at random, since items are put into bins in a uniform manner. In our t^{th} probe, the probability of choosing an empty bin will be $\left(\frac{N'-t}{N'-t-1}\right)^{n'}$. Since each probe is independent of the others, the probability of choosing t empty bins in the first t probes equals:*

$$\left(\frac{N'-1}{N'}\right)^{n'} \cdot \left(\frac{N'-2}{N-1}\right)^{n'} \cdots \left(\frac{N'-t+1}{N'-t+2}\right)^{n'} \cdot \left(\frac{N'-t}{N'-t+1}\right)^{n'} = \left(\frac{1}{N'}\right)^{n'} \cdot \left(\frac{(N'-1)\cdot(N'-2)\cdots(N'-t+1)}{(N'-1)\cdot(N'-2)\cdots(N'-t+1)}\right)^{n'} \cdot (N' - t)^{n'} = \left(\frac{N'-t}{N'}\right)^{n'} = P(X = t). \qquad \diamond$$

By solving eq. 5 for t, we get that, in order to choose a non-empty bin with probability of at least p, one has to visit at least: $t \leq lim = \lceil N' \cdot (1 - p^{\frac{1}{n'}})\rceil$ bins/nodes. By setting $\alpha = \frac{n'}{N'}$, we get: $lim = \lceil N' \cdot (1 - p^{\frac{1}{\alpha \cdot N'}})\rceil$. When using multiple ($m$) bitmap vectors, items are partitioned among the vectors, thus $\frac{n'}{m}$ items are inserted in N' bins, so the latter formula becomes: $lim_m = \lceil N' \cdot (1 - p^{\frac{m}{\alpha \cdot N'}})\rceil$. Finally, by taking replication into consideration, and assuming a replication degree of $\mathcal{R}$, we get:

$$lim_m^{\mathcal{R}} = \lceil N' \cdot (1 - p^{\frac{m}{\mathcal{R} \cdot \alpha \cdot N'}})\rceil. \qquad (6)$$

Note again that N' is the number of nodes responsible for a single-bitmap single-bit position (i.e. belonging to the same ID-space interval), n' is the number of items mapping to this interval, and α is their ratio. This means that there is a different optimal lim_m for every ID-space interval, with smaller-sized intervals (given a total n items being inserted in an N-node DHS) having lower values for lim_m (i.e. the interval(s) responsible for the least significant bit of the bitmap(s) will have the largest lim_m value(s)).

The default value of lim_m used in DHS for all intervals is 5 (constant), which suffices to guarantee that a non-empty node will be found with probability of at least 0.99 when the number of items mapped to any ID-space interval is greater or equal to the number of nodes in the interval (i.e. $n \geq m \cdot N$). Obviously, the default value of 5 also suffices for when counting sets with a larger cardinality then the one dictated by the above. However, when counting smaller-cardinality sets, we may choose to either (i) increase lim_m, according to eq. 6, (ii) use a smaller DHT/DHS overlay for the specific operation a la super-nodes in hybrid P2P networks – a trend that has lately started gaining supporters in the DHT world too, or (iii) use explicit replication of DHS bits.

Thus, hop-count complexity is in $O(k \cdot (\log N + lim))$, where lim is the upper bound of the number of iterations of the probing phase. For constant lim, as used in DHS, the hop-count complexity becomes $O(k \cdot \log N)$. Note that the cost of counting is independent of the number of bitmaps.

4.2 Counting in Multiple Dimensions

The latter observation constitutes a nice property of DHS and one of its great strengths: counting hop-count cost is independent of the number of bitmaps or in dimensions (metrics). This is so because counting consists of finding the rightmost 1-bit (or leftmost 0-bit) for all bitmaps of the distributed hash sketch. For the single-bitmap single-dimension case, the hop-count cost is in $O(k \cdot \log N)$ ($O(\log N)$ hops for each of the k bits of the bitmap). For multiple bitmaps and/or dimensions, the hop-count cost remains the same; the mapping of bit positions to ID-space intervals is the same for all bitmaps and all dimensions; thus, by visiting a given node in such an interval, the counting algorithm is able to probe for the status of the corresponding bit position in all bitmaps for all dimensions/metrics.

4.3 Histograms over DHS

Note that DHS is able to estimate the number of distinct items satisfying a predefined condition; all we need to do is have nodes record in the DHS all items satisfying this condition, and then execute the DHS counting algorithm. This provides us with the necessary tools to create and maintain equi-width histograms over data stored on the P2P overlay[5].

Assume we have a peer-to-peer system exposing an RDBMS-like functionality (e.g. [1, 18], etc.); data are stored in relations over the P2P overlay, following some predefined (relational) schema, and are usually replicated across nodes in the overlay. Now suppose we want to build a histogram over some attribute a in one of the relations. We proceed as follows: we create a partitioning of the domain $\mathcal{D} : [a_{min}, a_{max}]$ of values of attribute a into I equally-sized intervals/buckets $\mathcal{B}_i$, with $S = |\mathcal{B}_i| = \frac{a_{max} - a_{min} + 1}{I}$, such that $\mathcal{B}_i = [a_{min} + i \cdot S, a_{min} + (i+1) \cdot S)$, for all i in $[0, I-1)$. Thus each $\mathcal{B}_i$ will be assigned tuples satisfying the condition: $a_{min} + i \cdot S \leq a < a_{min} + (i+1) \cdot S$. We then create a $metric\_id$ for each bucket, and have nodes record all items they store to the corresponding metrics.

The per-node cost of creating the histogram equals the per-node cost for inserting all items for i metrics into the DHS – i.e. $O(k \cdot \log N)$ hops and $O(I \cdot b \cdot k \cdot \log N)$ bytes. The cost for a node to reconstruct the histogram from the data stored on the DHS is in $O(k \cdot \log N)$ hops and $O(I \cdot m \cdot b \cdot k \cdot \log N)$ bytes. Note that the hop-count cost is independent of the number of buckets and of tuples in the relation, and even independent of the number of bitmaps. The above design can be used to create and maintain arbitrary (non-equi-width) histograms, provided that the bucket boundaries are constant and known in advance.

5 Performance Evaluation

In this section we present an experimental evaluation of the performance and efficiency of DHS. First, we evaluate the cost of building and using a DHS, with a focus on network and storage requirements of the insertion algorithm, and network overhead of the counting algorithms. A second property we want to measure is the accuracy of the acquired estimates. Finally, we evaluate the appropriateness of DHS for use within a query optimizer for internet-scale P2P-based query processing systems such as [18].

5.1 Methodology

We assume we have a network consisting of 1024 nodes, arranged on a Chord[34]-like DHT. Node and item IDs are $(L =)64$ bits, created using MD4[6]. DHS keys are $(k =)$ 24 bits long (i.e. DHS bitmaps are 24 bits long). We tested configurations with various numbers of DHS bitmaps and report on the relative costs and accuracy. Unless stated otherwise, DHS is using 512 bitmaps. With 24-bit DHS keys, 8 bits should be more than enough for the bit field. With another 32 bits for the $time\_out$ field, plus 8 bits for the $metric\_id$ field, plus another 16 bits for the $vector\_id$ field (i.e. maximum 65536 vectors), the DHS tuple has a total size of 64 bits or 8 bytes. The value of the lim parameter was set to its default of 5 hops maximum.

The system hosts four relations – Q, R, S, and T – of size equal to 10, 20, 40, and 80 GBytes respectively. We assume a tuple size of 1kByte, so that relations contain 10, 20, 40, and 80 million tuples respectively. Tuples in the relations consist of a single integer attribute each, receiving values according to a Zipf distribution with $\theta = 0.7$. Tuples are randomly (uniformly) assigned to nodes. After all items have been assigned to nodes, we have the latter insert their items into the DHS one at a time, and record (i) the overall and average routing hops and bandwidth requirements, and (ii) the storage requirements on a per-node basis.

After the DHS has been populated, we select nodes at random and have them estimate the cardinalities of the four relations, using the DHS. During this phase we measure both routing hops and bandwidth consumption on a per-estimation basis, and the accuracy of the estimation relative to the actual cardinalities of the relations. Finally, we create 100-bucket equi-width histograms over DHS for all four relations and measure again the cost for populating the DHS and for reconstructing the histograms.

5.2 Results

Insertions and Maintenance DHS insertions and updates took on average 3.4 hops in our 1024-node network, for an average overall bandwidth consumption of ~ 27 bytes per insertion/update (excluding possible DHT protocol overheads and TCP/IP routing header information). This is in agreement with the expected $O(\log N)$ hop count and $O(b \cdot \log N)$ byte count per insertion (sect. 3.2).

As far as storage overhead is concerned, the experimental results again agree with the analysis of sect. 3.2: the average storage per node was measured to be ~ 384kBytes per relation, for a total of ~ 1.5MBytes per node for all tuples of all four relations. Note that this figure corresponds to the per-node storage requirements for maintaining information for 100 histogram buckets per relation with 512 DHS bitmap vectors per bucket, for a theoretical storage overhead of $\sim 400kBytes$ per node per relation.

[5]We are currently investigating methods to construct other, more complicated types of histograms (e.g. compressed, v-optimal, maxdiff, etc.)

[6]MD4 was selected due to its speed on 32-bit CPUs; a real-world system would probably use a stronger algorithm, such as MD5 or SHA1.

m	**nodes visited**	**hops**	**BW (kBytes)**	**error (%)**
128	68 / 65	86 / 69	11.0 / 8.8	5.0 / 5.8
256	73 / 69	92 / 77	11.8 / 9.6	3.5 / 4.3
512	81 / 80	120 / 114	15.4 / 15.9	1.8 / 2.7
1024	96 / 91	139 / 128	17.8 / 16.0	1.1 / 7.5

Table 2. Counting costs (sLL/PCSA)

m	**nodes visited**	**hops**	**BW (MBytes)**
128	69 / 67	89 / 72	1.1 / 0.9
256	73 / 70	94 / 80	1.2 / 1.0
512	79 / 81	118 / 108	1.5 / 1.4
1024	94 / 89	142 / 131	1.8 / 1.7

Table 3. Histogram building costs (sLL/PCSA)

Counting The counting results are summarized in table 2. Each estimation visited on average 80 nodes and required on average 109 hops for DHS-sLL algorithm, while the relevant numbers for the DHS-PCSA case were 76 nodes and 97 hops, depending on the number of bitmaps used.

This might seem counter-intuitive, since (i) sect. 4 states that the cost of counting using a DHS is independent of the number of bitmaps, and (ii) with an average of $5(= 0.5 \cdot \log 1024)$ hops per DHS lookup one would expect a larger hop-count for the given number of visited nodes. Note however that with multiple bitmaps, the per-bitmap inserted items are much less than in the single-bitmap case. Thus the probability of choosing an “empty” node increases with the number of bitmaps, and so does the hop-count cost due to retries (sect. 4.1). The node count mentioned above is largely dominated by one-hop visits to neighboring nodes due to retries; for example, of the 96 (average) nodes visited by DHS-sLL in the 1024-bitmaps case, only ~ 12 nodes were visited via DHT lookups, while the remaining 84 nodes were visited through one-hop retries.

Scalability We also tested our system with different numbers of nodes. As discussed in sect. 3.2 and 4, all operations in DHS are logarithmic to the number of nodes in the system. The experimental results confirm this analysis, with the average hop-count growing from $109/97$ hops for 1024 nodes, to $\sim 112/103$ for 10240 nodes for the DHS-sLL and DHS-PCSA cases respectively (figure omitted due to space limitations).

Accuracy For up to 2048 bitmaps (1024 for DHS-PCSA), the accuracy achieved is really good; we measured an average error of $\sim 2.9\%$ for DHS-PCSA, and $\sim 5\%$ for DHS-sLL. Both DHS-sLL and DHS-PCSA lose in accuracy for more than 4096 bitmaps; with 4096 bitmaps or more, the retry limit of 5 does not suffice to guarantee that we actually find a node storing information for a bit in question. DHS-sLL seems more tolerant in this direction, giving an estimation error of $\sim 15\%$ for 4096 vectors (as opposed to $\sim 44\%$ for DHS-PCSA), mainly due to the fact that DHS-sLL probes higher-order bits first, thus dealing with less sparse intervals. In any case, the accuracy achieved with 64 to 1024 bitmaps is already good, so using a larger number of vectors is not necessary. In the following experiments we are using 512 bitmaps for both DHS-PCSA and DHS-sLL.

Histograms and Query Processing The hop-count cost for a node to reconstruct the histogram for a relation stored in the P2P overlay was measured to be (on average) $\sim 111/98$ hops, while the bandwidth consumed during the histogram reconstruction phase was ~ 1.4MB$/1.0$MB for the DHS-sLL and the DHS-PCSA cases respectively. The results are summarized in table 3. We note again that the hop-count cost to estimate a complete histogram – i.e. estimating the cardinalities of multiple multisets/buckets at once – is equivalent to the cost of estimating the cardinality of a single multiset, as outlined in sect. 4.2. Moreover, the attained accuracy was also very good: we measured an average estimation error of $\sim 8.6\%$ per histogram cell for a DHS with 64 bitmap vectors, dropping to $\sim 7.7\%$ for 128 and $\sim 6.8\%$ for 256 bitmap vectors.

The bandwidth-consumption figures are very small, compared to the data transfers required during the actual query processing. Note that: (i) after a node has paid the $1MByte$ cost to reconstruct the histograms, choosing the right join ordering is a local operation, and (ii) this is the cost to reconstruct the whole histogram; query processing may require estimation of the cardinality of only specific buckets, depending on the query predicate constraints.

Now assume that a query optimizer, armed with the above histograms, is (at least) able of selecting the optimal join tree. In this case, the savings in bandwidth and response time will be considerable[5, 17]. For example, in [17] the authors consider multi-way joins in a much smaller setting than the one discussed above (256 nodes and four relations with $256,000$ tuples each or 100 tuples per node). The optimal join strategy in the three-way join case results in a data transfer of 47MB, as opposed to 71MB transferred by FREddies, both of which are orders of magnitude larger than the ~ 1MB required to reconstruct the histograms using DHS. Thus, if DHS-based histograms were added to the PIER query processing logic, PIER would select the optimal join plan at a negligible additional cost, resulting in major bandwidth and latency savings. This shows the impact that DHS can have in internet-scale query engines.

6 Conclusions

In this paper we presented Distributed Hash Sketches (or DHS); a novel, fully decentralized, scalable, and efficient mechanism, capable of providing estimates on the cardinality of multi-sets in internet-scale information systems. DHS is, to our knowledge, the first to simultaneously satisfy the central goals of efficiency, scalability, access and storage load balancing, high accuracy, and duplicate (in)sensitivity, all without additional explicit indexing structures. These characteristics make it suitable to become the counting technique of preference for internet-scale data networks.

We have shown how to build DHS utilizing either the PCSA or the super-LogLog hash sketches. We have analytically estimated the additional estimation errors introduced by the wide-scale distribution inherent in our technique. We have implemented DHS and evaluated it both in terms of its error and its performance characteristics. The experimental results substantiate our claims for small errors and related storage and bandwidth overheads, while showing the efficiency of the counting operation. Furthermore, our implementation has shown that DHS-based histograms can introduce great performance savings during query optimization.

We have illustrated a number of applications where DHS can play a catalyst's role. In particular, with DHS-based histograms, most – if not all – histogram-based techniques for query optimization can now be ported into internet-scale environments, making a stride towards leveraging existing know-how in the database systems community to facilitate internet-scale query processing optimizations.

Acknowledgments

Peter Triantafillou was partly funded by FP6 of the EU through IST DELIS (#001907). Nikos Ntarmos was funded by the PENED 2003 Programme of the EU and the General Secretariat for Research and Technology of the Hellenic State.

References

[1] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. In *Proc. PODC '04*.

[2] O. Babaoğlu, H. Meling, and A. Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proc. ICDCS '02*.

[3] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. Estimating aggregates on a peer-to-peer network. Submitted (http://dbpubs.stanford.edu/pub/2003-24).

[4] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proc. SIGMOD '04*.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proc. SIGCOMM '04*.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS '98*.

[7] S. Chaudhuri, R. Motwani, and R. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. SIGMOD '98*.

[8] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proc. ICDE '04*.

[9] A. Dobra, M. Garofalakis, J.Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Proc. EDBT '04*.

[10] P. Druschel and A. Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware '01*.

[11] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *Proc. ESA '03*.

[12] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2), October 1985.

[13] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *Proc. SIGMOD '03*.

[14] K. Gummadi, et al. The impact of DHT routing geometry on resilience and proximity. In *Proc. SIGCOMM '03*.

[15] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proc. CIDR '03*.

[16] M. Harren, et al. Complex queries in DHT-based peer-to-peer networks. In *Proc. IPTPS '02*.

[17] R. Huebsch and S. Jeffery. FREddies:DHT-based adaptive query processing via FedeRated Eddies. Technical Report UCB/CSD-4-1339, U. of Californa at Berkeley, 2004.

[18] R. Huebsch, et al. The architecture of PIER: an internet-scale query processor. In *Proc. CIDR '05*.

[19] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *Proc. CIDR '05*.

[20] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2), June 1984.

[21] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proc. ICDCS '04*.

[22] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. FOCS '03*.

[23] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proc. EDBT '04*.

[24] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis, April 1995.

[25] G. Manku. Routing networks for distributed hash tables. In *Proc. PODC '03*.

[26] P. Maymouknov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS '02*.

[27] A. Motresor, H. Meling, and O. Babaoğlu. Messor: Load-balancing through a swarm of autonomous agents. In *Proc. Workshop on Agent and Peer-to-Peer Systems '02*.

[28] W. Ng, B. C. Ooi, K. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. ICDE '03*.

[29] C. Palmer, et al. The connectivity and fault-tolerance of the internet topology. In *Proc. NRDM '01*.

[30] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *Proc. CIDR '03*.

[31] S. Ratnasamy, et al. A scalable content-addressable network. In *Proc. ACM SIGCOMM '01*.

[32] R. v. Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[33] P. Selinger, et al. Access path selection in a relational database management system. In *Proc. SIGMOD '88*.

[34] I. Stoica, et al. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. ACM SIGCOMM '01*.

[35] P. Triantafi llou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *Proc. VLDB DBISP2P '03*.

[36] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. SIGCOMM '04*.