Distributed Hash Sketches: Scalable, Efficient, and Accurate Cardinality Estimation for Distributed Multisets

N. NTARMOS and P. TRIANTAFILLOU
R.A. Computer Technology Institute and University of Patras
and
G. WEIKUM
Max-Planck-Institut für Informatik

Counting items in a distributed system, and estimating the cardinality of multisets in particular, is important for a large variety of applications and a fundamental building block for emerging Internet-scale information systems. Examples of such applications range from optimizing query access plans in peer-to-peer data sharing, to computing the significance (rank/score) of data items in distributed information retrieval. The general formal problem addressed in this article is computing the network-wide distinct number of items with some property (e.g., distinct files with file name containing "spiderman") where each node in the network holds an arbitrary subset, possibly overlapping the subsets of other nodes. The key requirements that a viable approach must satisfy are: (1) scalability towards very large network size, (2) efficiency regarding messaging overhead, (3) load balance of storage and access, (4) accuracy of the cardinality estimation, and (5) simplicity and easy integration in applications. This article contributes the DHS (Distributed Hash Sketches) method for this problem setting: a distributed, scalable, efficient, and accurate multiset cardinality estimator. DHS is based on hash sketches for probabilistic counting, but distributes the bits of each counter across network nodes in a judicious manner based on principles of Distributed Hash Tables, paying careful attention to fast access and aggregation as well as update costs. The article discusses various design choices, exhibiting tunable trade-offs between estimation accuracy, hop-count efficiency, and load distribution fairness. We further contribute a full-fledged, publicly available, open-source implementation of all our methods, and a comprehensive experimental evaluation for various settings.

N. Ntarmos was supported by the 03ED719 research project, implemented within the framework of the "Reinforcement Program of Human Research Manpower" (PENED), cofinanced by the National and Community Funds (25% from the Greek Ministry of Development-General Secretariat of Research and Technology and 75% from E.U.-European Social Fund). P. Triantafillou was funded by the 6th Framework Program of the EU through the Integrated Project DELIS (#001907) on Dynamically Evolving Large-scale Information Systems.
Authors' addresses: N. Ntarmos and P. Triantafillou, R.A. Computer Technology Institute and Computer Engineering and Informatics Department, University of Patras, 26500 Rio, Patras, Greece; email: {ntarmos, peter}@ceid.upatras.gr; G. Weikum, Max-Planck-Institut für Informatik, Saarbrücken, Germany; email: weikum@mpi-sb.mpg.de.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Distributed estimation, distributed information systems, distributed cardinality estimation, distributed data summary structures, hash sketches, peer-to-peer networks and systems

ACM Reference Format:

Ntarmos, N., Triantafillou, P., and Weikum, G. 2009. Distributed hash sketches: Scalable, efficient, and accurate cardinality estimation for distributed multisets. ACM Trans. Comput. Syst. 27, 1, Article 2 (February 2009), 53 pages. DOI = 10.1145/1482619.1482621 http://doi.acm.org/10.1145/1482619.1482621

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 [Gnutella 2001; Yang and Garcia-Molina 2001], to the cutting-edge structured data networks of today. The main advance that made this feasible was the introduction of Distributed Hash Tables (or DHTs) [Druschel and Rowstron 2001; Maymouknov and Mazières 2002; Ratnasamy et al. 2001; Stoica et al. 2001]. 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 items, 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 [Gupta et al. 2003; Harren et al. 2002; Huebsch et al. 2005, 2003; Ives et al. 2005; Koloniari and Pitoura 2004; Ng et al. 2003; Papadimos et al. 2003; Triantafillou and Pitoura 2003]. 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.

1.1 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 properties. For example, file-sharing peer-to-peer systems often need to know the total number of distinct documents shared by their users, without counting copies of the same (popular) document multiple times; 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

distinct indexed documents containing each keyword to the total number of distinct 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 execute aggregate queries and to maintain statistics over stored/shared data, en route to selectivity estimation and optimization algorithms for query access plans; etc.

Furthermore, with wide distribution comes the need for completely decentralized methods of performing traditionally centralized operations, and a 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. We believe the key constraints that any acceptable solution must satisfy consist of the following.

(1) *Efficiency*. The number of nodes that need to 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 may need to add elements to multiple multisets, which dictates the need for a highly distributed solution, avoiding single-point-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). This is comprised of, and highly accurate cardinality estimation.
(5) *Simplicity and Ease of Integration*. Special, solution-specific indexing structures, and their required extra (routing) state to be maintained by nodes, should be avoided.
(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.

1.2 Related Work

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

—rendezvous-based protocols;
—gossip-based protocols;
—broadcast/convergecast-type protocols; and
—sampling-based protocols.

The first type of solution is 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 Flajolet and Martin [1985]). Also in this category are 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) or "coordinator"-based solutions (abundant in sensor networks and distributed data stream processing [Cormode and Garofalakis 2005; Hadjieleftheriou et al. 2005], where summaries of data are gathered at a central aggregation point to be processed).

Solutions of this type suffer many shortcomings, most notably 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. This violates constraint (2). Moreover, each of these counting nodes is subjected to a high access and storage load, violating constraint (3), while it can be argued that such highly loaded nodes will exhibit high response times, also violating constraint (1). Using a (fixed) number of rendezvous nodes for each counter does not solve the problem, but merely mitigates the scalability issues to the cost of inserting items to and/or querying the value of such a counter (as these grow linearly with the number of rendezvous nodes engaged in the computation) while also violating constraint (1). Similar arguments hold for the case when multiple rendezvous nodes have to be contacted as the result of simultaneously maintaining multiple counters (i.e., counting multiple quantities at the same time). However, rendezvous-based solutions are quite popular in the literature, mainly due to their simplicity and excellent hop-count performance in the single-counter case.

The second type of solutions [Babaoğlu et al. 2002; Jelasity and Montresor 2004; Kempe et al. 2003; Montresor et al. 2002] usually provide probabilistic semantics of "eventual consistency" for their outcome; gossip-based protocols are based on an iterative procedure, wherein every node exchanges information with a (set of) its neighboring 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. Moreover, the fact that all nodes have to actively participate in a gossip-based computation (even if it is of no interest to them), coupled with the multiround property of these solutions, violates constraint (1), while their semantics violate constraint (4). Of course, in unstructured overlays it is it not clear if it is possible to do better than this.

The third type of solution [Bawa et al. 2003, 2004; Considine et al. 2004; van Renesse et al. 2003; Yalagandula and Dahlin 2004] 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 based on prebuilt tree structures also belong in this group.

Of these works, Astrolabe [van Renesse et al. 2003] was among the first 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 idea was proposed in Yalagandula and Dahlin [2004]. Bawa et al. [2003] proposed 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 multiple simultaneous counters/aggregates. Specifically, similarly to rendezvous-based approaches, the cost of maintaining multiple counters simultaneously grows linearly with the number of such counters (e.g., when having to maintain a different tree per counter). Furthermore, even with just one counter, if the number of nodes containing items to be counted is in $O(N)$, then the counting cost (total hop-count and number of messages) is also in $O(N)$. Moreover, it could be argued that such solutions are a sort of "directed gossip," since the core functionality is very similar to that of gossip-based algorithms. This is with the exception that during phase (ii), nodes only exchange information with their parent and children nodes in the (virtual) tree. Consequently, these solutions violate constraint (1), while most of them (with the exception of Bawa et al. [2004]) also violate constraints (3) and (4).

The core idea of the last type of solution [Bharambe et al. 2004; Manku 2003] is to estimate the value of the counter in question by selectively querying (sampling) a set of nodes in the network. Bharambe et al. [2004] attempt to compute approximate histograms of system statistics by using random sampling of nodes in the network. Manku [2003] estimates the number of nodes in the overlay by also using a random sampling algorithm. There are some obstacles in such approaches. First, there is no obvious way to generalize these techniques to count arbitrary quantities (other than the ones they were designed for). Second, sampling-based techniques are known to suffer from accuracy issues [Chaudhuri et al. 1998], thus violating constraint (4). On the other hand, if the sample is big enough [Chaudhuri et al. 1998] to guarantee a certain level of confidence, then these solutions violate constraint (1). Lastly, sampling-based techniques are usually duplicate-sensitive, violating constraint (6).

Hash sketches [Durand and Flajolet 2003; Flajolet and Martin 1985], to be presented shortly, provide a distributable, duplicate-insensitive method of estimating the cardinality of multisets. All known works that manage to provide duplicate-insensitive counting [Bawa et al. 2003, 2004; Considine et al. 2004] use hash sketches. However, they all fall into the broadcast/convergecast category of counting algorithms, with the disadvantages mentioned earlier. Moreover, in other sketching techniques, such as Alon et al. [1999, 1996]; Bar-Yossef et al. [2002], Cormode and Muthukrishnan [2004], and Beyer et al. [2007], relying on ordering of (hashed) data cannot be implemented in a completely decentralized manner following the DHT paradigm, without resorting to a rendezvous-style solution or gossip-based approach. Assume that each node

constructs such a local synopsis. The question arises as to how to distribute the items (be they bits, hashes, tuples, etc.) of the synopsis across nodes in the overlay, so that this information is readily available to all other nodes without burdening any one of them. Take, for example, the KMV synopsis of Beyer et al. [2007]. Suppose that each node computes a local such synopsis. In order to compute the global KMV synopsis, we have to somehow store this information in the overlay for all nodes to know. Putting aside the choices of gossiping and aggregation trees, for the reasons discussed before, the only other alternative (we can think of) is a rendezvous-based approach where k nodes are each assigned the task of storing the i^{th}, $(i = 1...k)$, smallest value in the KMV synopsis.

1.3 Contributions

In Ntarmos et al. [2006] we proposed Distributed Hash Sketches (or DHS): a novel, fully decentralized mechanism capable of providing estimates on the cardinality of multisets of items in a structured peer-to-peer system. DHS is, to our knowledge, the first truly distributed version of hash sketches, that is, a probabilistic counting mechanism, as first proposed by Flajolet and Martin [1985] and more recently by Durand and Flajolet [2003], along with the accompanying algorithms and protocols for DHTs. Moreover, this is to our knowledge the first distributed counting mechanism satisfying all six requirements presented earlier.

One of the core design considerations and premises of the peer-to-peer paradigm, of which DHT networks are a prime example, is completely decentralized operation. Designing and implementing hash sketches over DTHs in an efficient and scalable way, while abiding the aforesaid constraint, is a formidable task. Salient properties of our solution are:

—balanced storage and access load;
—highly efficient operation, independent of the number of items and logarithmic in the number of nodes in the overlay;
—excellent scalability properties attained through the accomplishment of the previous two goals;
—derivation of bounds on the error added by the distributed operation and examination of its algorithmic implications;
—discussion and examination of the design space and presentation of new methods, techniques, and algorithms to implement hash sketches in a distributed manner; and
—implementation and evaluation of DHS with respect to its estimation error, overhead, and scalability properties.

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 balanced distribution of storage and access load on the DHT nodes; (iii) provides probabilistic guarantees regarding the correctness of the distributed implementation in relation to the original, centralized algorithm, as well as accuracy measures of the produced estimates in terms of their average error and

variance; (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 multisets.

In this work, we discuss the approach in Ntarmos et al. [2006] in more detail. We contribute a number of optimizations and new algorithms which introduce dramatic performance improvements during all operations and in efficiency and load distribution fairness. More specifically, in addition to the vanilla DHS functionality, this work contributes the following.

(1) Improvements in the message routing protocol, in the form of:
   (a) a recursive routing scheme, allowing for lower overall hop counts over lower-latency network links; and
   (b) a "shortcuts" mechanism, harnessing the entries in the nodes' routing tables, that essentially turns most of the DHS messages into single-hop operations.
(2) Improvements in the bit-probing (query) algorithm; more specifically:
   (a) a binary-search bit-probing algorithm as an alternative to the existing sequential probing technique; and
   (b) a bit-caching mechanism, coupled with recursive routing, that allows for even lower query hop-counts and a better load distribution across nodes in the overlay.
(3) Improvements in the data insertion algorithm, consisting of:
   (a) a "bulk" insertion algorithm allowing multiple items to be inserted in a single operation;
   (b) a simple soft-state replication scheme with a low hop-count overhead, allowing for better fault tolerance and an even lower query hop-count cost; and
   (c) an insertion/update algorithm based on piggybacking messages on both DHS-specific and DHT routing table maintenance messages.

Last but not least, we contribute both a full-fledged implementation of the proposed algorithms and protocols on top of FreePastry [FreePastry 2002], a freely available implementation of the Pastry DHT overlay [Druschel and Rowstron 2001]. Further, we provide an in-depth performance evaluation of all aspects of the proposed algorithms and protocols using this substrate, which offers new insights and explores trade-offs. The source code of the implementation (coined *FreeDHS*) is available on the World Wide Web [FreeDHS 2006] for anyone to download and test.

1.4 Outline

The rest of this article proceeds as follows. Section 2 introduces the background behind the design of DHS. Section 3 presents the architecture, algorithms, and protocols of DHS, both describing the basic infrastructure of Ntarmos et al. [2006] and discussing the novel methods, algorithms, and techniques contributed by this work. Section 4 presents the results of our extensive performance evaluation, and Section 5 concludes.

2. BACKGROUND

This section presents the basic building blocks used in the design of DHS. First, we give a brief introduction to Distributed Hash Tables and their functionality and properties. Then, we discuss hash sketches as defined by Flajolet and Martin [1985] and Durand and Flajolet [2003]. The fusion of these two threads forms the fabric underlying DHS, to be presented in the next section.

2.1 Distributed Hash Tables

Distributed Hash Tables comprise a family of structured peer-to-peer network overlays exposing a hash-table-like interface. The main advantage of DHTs over older, unstructured P2P networks lies in their probabilistic (in the presence of node failures and network dynamics) performance guarantees. Prominent examples of DHTs include Pastry [Druschel and Rowstron 2001], Chord [Stoica et al. 2001], CAN [Ratnasamy et al. 2001], Tapestry [Zhao et al. 2001], Kademlia [Maymouknov and Mazières 2002], etc.

DHTs offer two basic primitives: *insert(key, value)* and *lookup(key)*. Nodes are assigned unique identifiers from a circular ID space and arranged according to a predefined geometry and distance function [Gummadi et al. 2003]. Node IDs are computed as either the secure hash of some node-specific piece of information [Stoica et al. 2001; Druschel and Rowstron 2001] (e.g., the concatenation of the IP address of the node and the port number on which the P2P application is operating), or as the outcome of a pseudo-uniform random number generator [Maymouknov and Mazières 2002].[1] This results in a partitioning of the node-ID space among nodes, so that each node is responsible for a well-defined set (arc) of identifiers. Each item is also assigned a unique identifier from the same ID space (usually by simply feeding the item to the same (cryptographic or random) hash function used to generate the node IDs) and stored at the node whose ID is closest to the item's ID, according to the DHT's distance function.

Each node in an N-node DHT maintains direct IP links to $O(\log(N))$ other nodes in appropriate positions in the overlay, as dictated by the DHT's geometry, so that routing between any two nodes takes $O(\log(N))$ hops in the worst case.[2] For any given node, these links are usually classified in two major categories: (i) "fingers", namely links to nodes with IDs far away in the ID space from the ID of the current node; and (ii) "predecessors/successors", namely links to nodes immediately preceding/succeeding the current node in the node-ID space. The former links are used to achieve $O(\log N)$-hop lookups between any two nodes, while the latter keep the DHT overlay connected. This information is maintained through PING/PONG and/or periodic heartbeat messages, and updated either when a node leaves—either gracefully or abruptly (e.g., because the node has failed or gone offline, or there is a network partition or other low-level failure)—or as part of a periodic process called *stabilization*. During the stabilization period DHT nodes perform standard DHT lookups for the IDs corresponding to each of the positions in their routing table, with a total hop-count

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

cost in $O(\log^2 N)$ per node (e.g., $O(\log N)$ hops for each of the $O(\log N)$ routing table entries). Furthermore, for fault tolerance and increased availability, nodes usually maintain a list of candidate links for every position in their routing table, so that if one of these links fails it is replaced by a functional link from the candidate list. Lastly, in several DHTs, nodes also keep a log of contact information for nodes they have contacted as part of normal DHT operation (e.g., data insertion, lookups, routing table maintenance, etc.) but which are stored neither in the routing table nor in the replacement list. These links are usually either hidden behind the big-O notation, as they are either a constant factor times the normal $O(\log N)$-size routing state (as is the case for nodes in the candidate list) or ignored in the cost formula, as they do not contribute to the maintenance cost but are only occasionally updated (as is the case for the contact log links).

Lately, a new family of DHTs, namely locality- (or order)-preserving DHTs, has emerged as an answer to the shortcomings of first-wave DHT networks in dealing with queries other than equality ones. In general, due to the pseudorandom output of cryptographic hash functions, each DHT node will be responsible for storing a (possibly random) subset of values. This holds regardless of the specific DHT employed and has grave implications for the efficiency of several query types other than point (equality) queries. Simple range queries are a good example of how the messaging overhead of traditional DHTs may deteriorate to $O(N)$ in an N-node network: Given a range predicate constraint, the basic (if not only) method of proceeding with such a query is to execute a point query for each and every possible value in the requested range. Thus, for a query for a range consisting of v values, this translates to $O(v \cdot \log(N))$ messages, which is prohibitive in most situations. Locality-preserving DHTs [Aspnes and Shah 2003; Harvey et al. 2003; Triantafillou and Pitoura 2003; Aberer et al. 2005; Pitoura et al. 2006] also possess the same hash-table-like interface as first-wave DHTs, but store consecutive data values on adjacent nodes in the overlay (node IDs are still assigned in a pseudorandom manner). Given the same v-value range query, the querying node just has to locate the node responsible for the lower end of the range and then follow single-hop (successor) pointers until it reaches the node responsible for the upper end of the range, gathering matching tuples in the process. This roughly sums up to $O(\log(N) + v)$ total messages, which is far better than the previous figure for traditional DHTs.

The solutions proposed in this article apply equally well to any of the aforementioned overlays, as long as they offer the basic *insert(key, value)/lookup(key)* DHT API.

2.2 Hash Sketches

Durand and Flajolet [2003] presented an algorithm (coined *superLogLog* or *DF03* counting) and accompanying data structures allowing to estimate the number of distinct items in a multiset $\mathcal{D}$ of data in a database, improving on the pioneering work of Flajolet and Martin [1985] which introduced hash sketches under the name of *Probabilistic Counting with Stochastic Averaging* (coined *PCSA* or *FM85*). In brief, Durand and Flajolet [2003] reduced the space

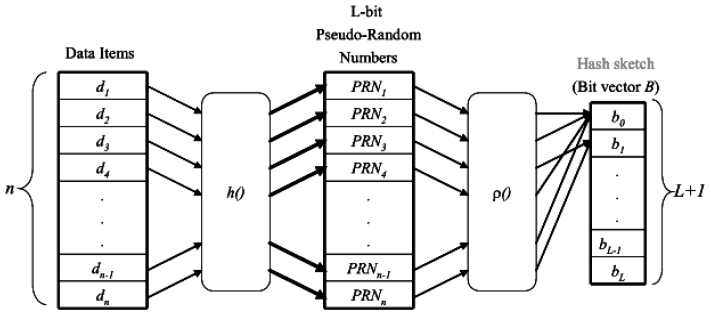


Fig. 1. Inserting items into a hash sketch: single-bitmap case.

complexity and relaxed the assumptions on the statistical properties of the hash function of Flajolet and Martin [1985].[3] The estimate obtained by both algorithms 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 $h() : \mathcal{D} \rightarrow [0, 1, \ldots, 2^{\lambda})$, an assumption also present in most (if not all) P2P-related research. Hash sketches have been used in many application domains where counting distinct elements in multisets is of some importance, such as approximate query answering in very large databases [Krishnan 1995], data mining on the Internet graph [Palmer et al. 2001; Michel et al. 2006], and stream processing [Ganguly et al. 2003; Dobra et al. 2004].

A hash sketch consists of a bit vector $B[\cdot]$ of length λ, with all bits initially set to 0, and a hash function $h()$ as described earlier.

2.2.1 *Super-LogLog Counting.* Let $\rho(y) : [0, 2^{\lambda}) \rightarrow [0, \lambda)$ be the position of the least significant (leftmost) 1-bit in the binary representation of y; that is, $\rho(y) = \{min(i \geq 0) : bit(y, i) \neq 0\}$, $y > 0$, and $\rho(0) = \lambda$, where $bit(y, i)$ denotes the i^{th} bit in the binary representation of y (bit-position 0 corresponds to the least significant bit). In order to estimate the number I of distinct elements in a multiset $\mathcal{D}$, we apply $\rho(h(d))$ to all $d \in \mathcal{D}$ and record the results in the bitmap vector $B[0 \ldots \lambda - 1]$ (see Figure 1). Since $h()$ distributes values uniformly over $[0, 2^{\lambda})$, it follows that

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

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

Then, the quantity $R(\mathcal{D}) = max_{d \in \mathcal{D}} \rho(h(d))$ provides an estimation of the value of $\log(I)$, with an additive bias of 1.33 and a standard deviation of 1.87. Thus, 2^R estimates "logarithmically" I within 1.87 binary orders of magnitude. However, the expectation of 2^R is infinite and thus cannot be used to estimate

[3]The analysis leading to the equations used in this section is well beyond the scope of this article and can be found in Flajolet and Martin [1985] and Durand and Flajolet [2003].

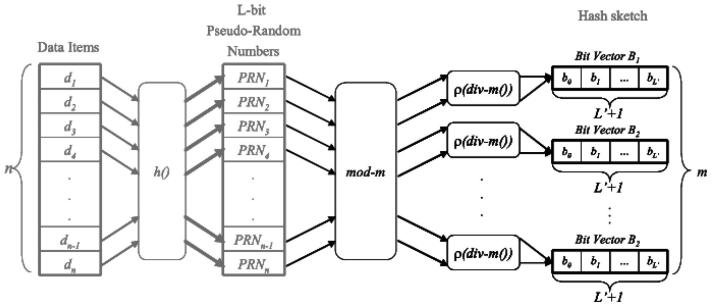


Fig. 2. Inserting items into a hash sketch: multiple-bitmaps case.

I. To this extent, Durand and Flajolet [2003] propose the following technique (similar to the *stochastic averaging* technique in Flajolet and Martin [1985]): (i) Use a set of $m = 2^c$ different $B^{\langle i \rangle}[\cdot]$ vectors (c being a non-negative integer), each resulting in 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 $h(d)$ (see Figure 2).

If $M^{\langle i \rangle}$ is the (random) value of the parameter R for vector i, then the arithmetic mean $\frac{1}{\beta}\sum_{i=1}^{\beta} M^{\langle i \rangle}$ is expected to approximate $\log(\frac{I}{\beta})$ plus an additive bias. The estimate of I is then computed by the formula $E(I) = \alpha_\beta \cdot \beta \cdot 2^{\frac{1}{\beta}\cdot\sum_{i=1}^{\beta} M^{\langle i \rangle}}$, where $\alpha_\beta = (-\beta \cdot \frac{2^{-\frac{1}{\beta}}-1}{\log(2)} \cdot \int_0^\infty e^{-t} \cdot t^{-\frac{1}{\beta}} dt)^{-\beta}$ [Durand and Flajolet 2003]. The authors further propose a *truncation rule*, consisting of taking into account only the $\beta_0 = \lfloor \theta_0 \cdot \beta \rfloor$ 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

$$E(I) = \tilde{\alpha}_\beta \cdot \beta_0 \cdot 2^{\frac{1}{\beta_0}\cdot\sum^{*} M^{\langle i \rangle}}, \tag{2}$$

where $\sum^*$ indicates the truncated sum, and the modified constant $\tilde{\alpha}_\beta$ ensures that the estimate remains unbiased (see Figure 3). The resulting estimate has a standard deviation of $\frac{1.05}{\sqrt{\beta}}$, while the hash function must have a length of at least

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

I_{max} being the maximum cardinality estimated.

2.2.2 *PCSA Counting.* The algorithm in Flajolet and Martin [1985] is based on the same hashing scheme (i.e., using $\rho(\cdot)$) and the same observations as Durand and Flajolet [2003]. The PCSA algorithm differs from the super-LogLog algorithm in the following ways: (i) Flajolet and Martin [1985] rely on the existence of an explicit family of hash functions exhibiting ideal random properties, while Durand and Flajolet [2003] have relaxed this assumption; (ii) Flajolet and Martin [1985] 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 for Durand and Flajolet [2003]; (iii) Durand and Flajolet [2003] use on the order of $\log\log(\textit{max cardinality})$ bits per bitmap while Flajolet and Martin [1985]

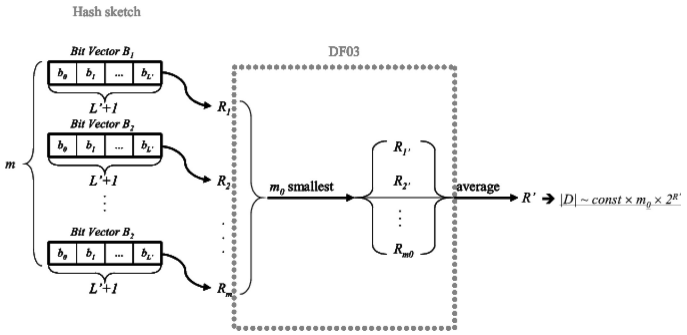


Fig. 3. Counting items using a hash sketch.

need on the order of log(*max cardinality*) bits per bitmap; (iv) the estimation in Flajolet and Martin [1985] is computed as

$$E(I) = \frac{1}{0.77351} \cdot \beta \cdot 2^{\frac{1}{\beta}\sum_0^{\beta-1} M^{(i)}}; \tag{4}$$

and (v) the bias and standard error of Flajolet and Martin [1985] are closely approximated by $1 + 0.31/\beta$ and $0.78/\sqrt{\beta}$, respectively. Note that the data insertion algorithm is the same for both Durand and Flajolet [2003] and Flajolet and Martin [1985] (with the sole difference in the assumptions on the hash function).

Hash sketches exhibit a natural distributivity; the hash sketch of the union of any number of sets can be computed from the hash sketches of these sets by a simple bitwise OR of the corresponding bit vectors, given that all hash sketches have the same number of bit vectors and length and that they have been built using the same set of hash functions. Thus, if an initial set A is spread across several hosts (e.g., across a peer-to-peer network), it is possible to compute the global hash sketch for A from each of the locally computed hash sketches corresponding to the subset of A that each peer is responsible for.

3. DISTRIBUTED HASH SKETCHES

Table I summarizes the notation we shall be using for the rest of this article (small Greek letters denoting DHS-specific parameters, and capital Latin letters being used for DHT-related parameters). We have implemented both the superLogLog and the PCSA estimators within our framework [FreeDHS 2006]. Here we report only on the former for clarity of presentation and due to its better analytical properties and runtime requirements. We shall first discuss hash sketches using only a single $B[\cdot]$ vector (i.e., $\beta = 1$), extending our design to multiple vectors later.

3.1 Mapping Hash Sketch Bit-Positions to DHT Nodes

3.1.1 *The Naive Approach.* Assume that our hash sketch consists of a single λ-bit vector $B[\cdot]$. A naive DHT-based implementation would assign each of these λ positions to a node in the network (a so-called rendezvous approach, also mentioned in Flajolet and Martin [1985]) and use these nodes to store

Table I. Notation Summary

DHT Parameters	
L	: Length (in bits) of DHT keys (typically 160 bits)
N	: Number of nodes in the overlay
I	: Number of distinct items in the distributed multiset (maximum 2^{80} items)
I_n	: Number of distinct items contributed by node n
DHS Parameters	
β	: Number of DHS bitmaps (typically 256 bitmaps)
λ	: Length (in bits) of DHS bitmaps (maximum 80 bits; typically 40 bits)
μ	: Number of DHS metrics/dimensions (maximum 2^{80} metrics)
τ	: Maximum number of retries per DHS probe (typically 5)
α	: Degree of replication of DHS data
σ_{type}	: Size (in bytes) of DHS message type *type*

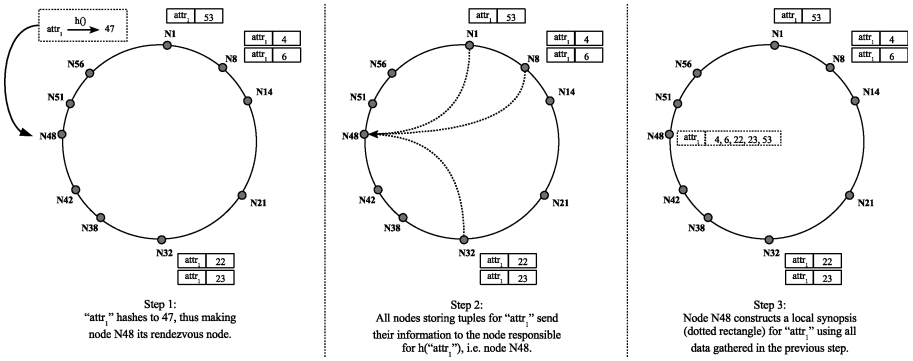


Fig. 4. Example of a rendezvous-based approach: Node 48 is designated as the rendezvous node and all nodes send to it any relevant data they store.

bit values in a distributed manner (Figure 4). However, this design has many serious flaws: (i) Only λ out of N (with $\lambda \ll N$) 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 items spread over λ nodes, the node join/leave operations for any of these nodes would result in moving around information for $\frac{2^L}{\lambda}$ items—a prohibitive cost, regardless of the size of the data maintained per item.

Obviously, the edge case of storing the complete hash sketch (i.e., all bit positions of all bitmaps) on a single rendezvous node for a given estimated quantity is in an even worse state with regard to the aforesaid shortcomings, as a single node will be burdened with all of the load (let alone becoming a single point of failure and defeating the core premise of P2P overlays for decentralized operation). Moreover, in order to compute the hash sketch estimate for multiple quantities in a rendezvous-based scenario, we need to contact equally many rendezvous nodes, thus facing an increase in the overall hop-count cost linear to the number of estimated metrics. Obviously, distributing hash sketches over a DHT needs to be approached from a completely different angle. Enter Distributed Hash Sketches: our highly efficient, scalable, and fully decentralized solution to this problem.

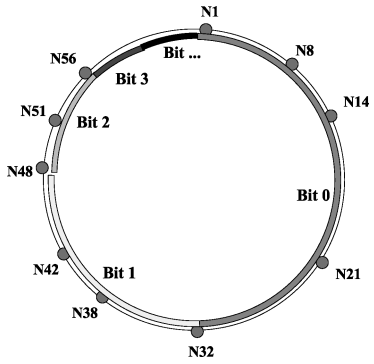


Fig. 5. Mapping of bit positions to nodes in the overlay: single bitmap/metric case.

```
1 Algorithm mapBitToNode(Bit bit) {
2    ID targetID = this.computeRandomIdForBit(bit);
3    Node targetNode = DHT.lookup(targetID);
4    return targetNode;
5 }
```

Fig. 6. Mapping algorithm, randomly selecting an ID in the target bit arc and contacting the node responsible for it.

3.1.2 *The DHS Approach.* We thus propose to partition the node-ID space, $[0, 2^L)$, into λ consecutive, nonoverlapping intervals $\mathcal{I}_i = [thr(i), thr(i-1))$, $i \in [0, \lambda)$, where $thr(i) = 2^{L-i-1}$. Using this partitioning, bit i of $B[\cdot]$ is mapped to node IDs randomly (uniformly) chosen from $\mathcal{I}_i$. In other words, for every DHS operation (insertion, update, query, etc.) related to a specific bit position i of the distributed hash sketch, nodes choose a random ID in $\mathcal{I}_i$ and contact the node responsible for that ID. Figure 5 depicts this mapping and Figure 6 summarizes the basic algorithm. The underlying DHT guarantees that there is at least one node responsible for each possible ID and thus that all bit positions/arcs are covered by at least one node.

Remember (Eq. (1)) that for an I-items multiset, bit i of the bitmap vector is visited $I \cdot 2^{-i-1}$ times. With the λ-bit vectors used in DHS, this translates to a maximum of (roughly) 2^λ distinct items in any possible multiset, or a maximum of $2^{\lambda-i-1}$ items being mapped to position i in the bitmap vector. Now, note that the interval (arc) $\mathcal{I}_i$ for bit position i consists of $|\mathcal{I}_i| = 2^{L-i-1}$ IDs, resulting in a balanced distribution of information across all nodes in the network. Furthermore, this mapping is the same for all estimated metrics and all bitmap vectors of the distributed hash sketch. This allows us to set or check a given bit position for multiple metrics and/or multiple vectors in a single operation, contacting just one node in the corresponding ID-space arc.

3.1.3 *Thresholding.* It is common practice among works in estimation of join sizes [Lipton et al. 1990; Lipton and Naughton 1995; Ganguly et al. 1996] and of frequency moments in general [Alon et al. 1999] to consider a sanity (lower) bound in the cardinality to be estimated. This mainly stems from the

fact that estimating small quantities is much harder than estimating larger ones, and the end result is significantly improved estimation error bounds. We can also consider such a lower bound in our setting, and actually incorporate it in the bit position mapping as follows. Assume we set a lower bound of $\mathcal{C}_{LB}$ on the input multiset cardinality. Based on Eq. (3), we consider only bit positions above $\lambda_0 = \log(\frac{\mathcal{C}_{LB}}{\beta}) - 3$. Thus, we alter the mapping of bit positions to ID-space arcs so that this bit position is mapped to the first half of the ring (as is the case for bit position 0 in the normal mapping) and bits below this position (less significant) are ignored. Specifically, this has the effect of assigning the i^{th} DHT interval to the $(\lambda_0 + i)^{th}$ bit. As less frequent bits are mapped to larger DHT intervals, this technique calls for either a coupling with the replication technique of Section 3.3.5 or a higher retry limit (see Section 3.5.4). On the other hand, there is the added bonus of better load distribution, due to the offloading of higher (albeit less frequently visited) bit positions to larger DHT intervals, and lower insertion and/or query hop-count costs, as less bits have to be set/probed.

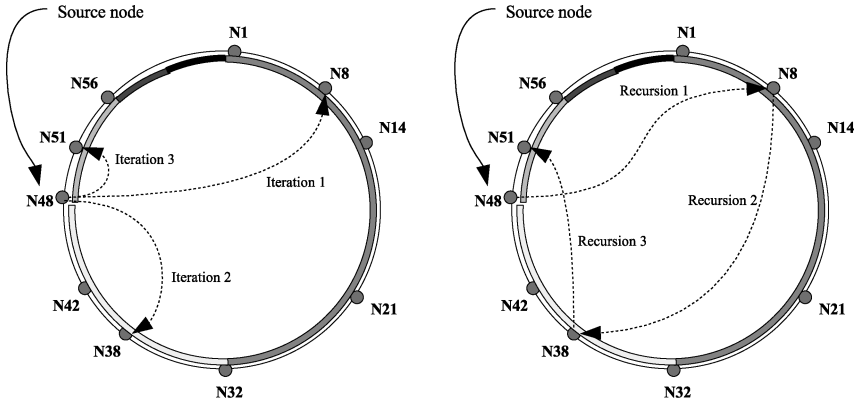
3.2 DHS Message Routing

This section discusses the message routing chores of DHS. First we present the two basic message propagation protocols used when multiple bit positions of the distributed hash sketch need to be contacted. We then describe a routing optimization based on harnessing the entries in the routing tables of DHT nodes to speed-up DHS operations.

3.2.1 *Iterative vs. Recursive Message Routing.* As we shall shortly see, there are several DHS algorithms that require nodes corresponding to multiple bit positions to be contacted as part of a single higher-level DHS operation (such as data insertion or computation of the actual distributed hash sketch estimate). The issue arises then of how to route the messages for such a multi-bit operation: either treat them as multiple singleton operations, iterating over the set of bit positions and accessing appropriate nodes at a time, or as one big operation to be processed in a single sweep around the node-ID circle.

The first scheme, coined *Iterative Bit Routing* (or *IBR*), follows the guidelines of the routing algorithms described in the early DHT-related works, consisting of iteratively visiting nodes closer to the target ID in order to retrieve the address of the next node to be contacted, until the desired node is found. Assume a source node n_s wishes to execute a certain multibit DHS operation, for bit positions stored in a bit vector. The source node then iterates over this vector, and for each bit position it selects a random ID in the corresponding ID-space arc and asks the node responsible for that ID to execute the requested operation. This scheme is depicted in Figure 7(a) and summarized in the pseudocode algorithm of Figure 8.

Later research in the DHT routing field [Dabek et al. 2004; Rhea et al. 2004] showed that iterative routing faces several performance and fault-tolerance issues in real-world applications, and instead advocated recursive routing. We thus have also implemented a *Recursive Bit Routing* scheme (*RBR*). In this



(a) iterative bit routing (b) recursive bit routing

Fig. 7. Example of DHS routing for multibit operations. Node 48 executes an operation accessing bit positions 0, 1, and 2.

```
1 Algorithm IterativeBitRouting(Operation op) {
2     while (op.getBits().hasMoreBits()) {
3         Bit nextBit = op.getBits().getNextBit();
4         Node targetNode = this.mapBitToNode(nextBit);
5         Result partialResult = targetNode.executeOperation(op);
6         this.processResult(partialResult);
7     }
8 }
```

Fig. 8. Algorithm *IBR*: Iteratively execute a DHS operation *op* for multiple bit positions.

scheme, depicted in Figure 7(b) and summarized in Figure 9, First n_s sorts, the bits in ascending order and stores them in a bit vector. It then chooses a random ID in the ID arc for the lower of these bits, and requests the node responsible for that ID to execute the desired operation. The target node executes locally the operation and recursively forwards its local result, along with the operation and any other input data received so far, to a random ID in the arc for the next bit in the bit vector. When all bit positions in the bit vector have been processed, the last node in the recursion chain can either respond directly back to n_s or return the final result to the previous node in the recursion chain, thus (in essence) doing a backward recursion all the way back to n_s. The recursive message routing scheme is more resilient to network partitions and other such low-level problems, while it also allows routing to take advantage of possible optimizations in the local routing tables of nodes (e.g., Pastry and Kademlia choose among candidate nodes for their routing tables based on link-level statistics, such as minimum network latency and/or round-trip time). Moreover, it paves the way for our caching-based optimization, to be presented later.

3.2.2 *Routing Shortcuts.* An interesting side-effect of the random selection step in the mapping of DHS bits to target node IDs (and in all of the aforementioned routing algorithms), is that, for a given bit position, we do

```
1  Algorithm RecursiveBitRouting(Operation op, Result result) {
2     Result myResult = this.executeOperation(op);
3     myResult.merge(result);
4     if (op.getBits().hasMoreBits()) { // Forward recursion
5        Bit nextBit = op.getBits().getNextBit();
6        Node nextNode = this.mapBitToNode(nextBit);
7        op.addNodeToPath(this);
8        nextNode.recursiveBitRouting(op, myResult, nextBit);
9     } else if (op.respondDirectlyToSender) {
10       op.getSender().processResult(myResult);
11    } else if (op.respondThroughBackwardRecursion) {
12       this.recurseBack(op, myResult);
13    }
14 }
15 recurseBack(Operation op, Result result) {
16    Node prevNode = op.getPrevNodeInPath();
17    // Do any extra work (e.g. result caching) here, then...
18    if (prevNode == null) // Backward recursion reached n_s
19       this.processResult(result);
20    else // Continue with backward recursion
21       prevNode.recurseBack(op, result);
22 }
```

Fig. 9. Algorithm *RBR*: Recursively execute a DHS operation *op* for multiple bit positions. *result* stores the partial result on each recursion.

not care about the exact node we will actually contact, as long as the ID-space interval for which it is responsible overlaps or is contained in the ID-space arc corresponding to the target bit. This allows us to harness the links to quasirandom nodes in the overlay maintained in the DHT routing table, candidate replacement list, and possible contact log data structures. Thus, instead of selecting a random ID in the target bit arc and then doing a standard $O(\log N)$-hops DHT lookup, we can first find all nodes in the aforesaid data structures whose ID is within the target arc and randomly select one of them as the *shortcut* target node. If no such node exists, we revert to the normal routing procedure of random ID selection and standard DHT lookup for the next hop, and repeat the shortcut attempt on the target node. This allows us to convert most DHT lookups to single- or two-hop operations without any observable effect on the distribution of accesses to nodes in the overlay.

3.3 DHS Insertion

We turn now to the algorithm used to insert items into the DHS. We shall first discuss the single-item insertion procedure (or *SII*), used when items arrive one at a time and we are interested in online computation (as is the case for streaming applications). Then, we will present a "bulk" insertion scheme (or *BII*), best suited for data-sharing and data management scenarios, as well as an efficient, soft-state replication scheme. Last, we present a routing scheme that piggybacks DHS messages on DHT maintenance traffic, thus almost completely hiding the (already low) network cost for the DHS.

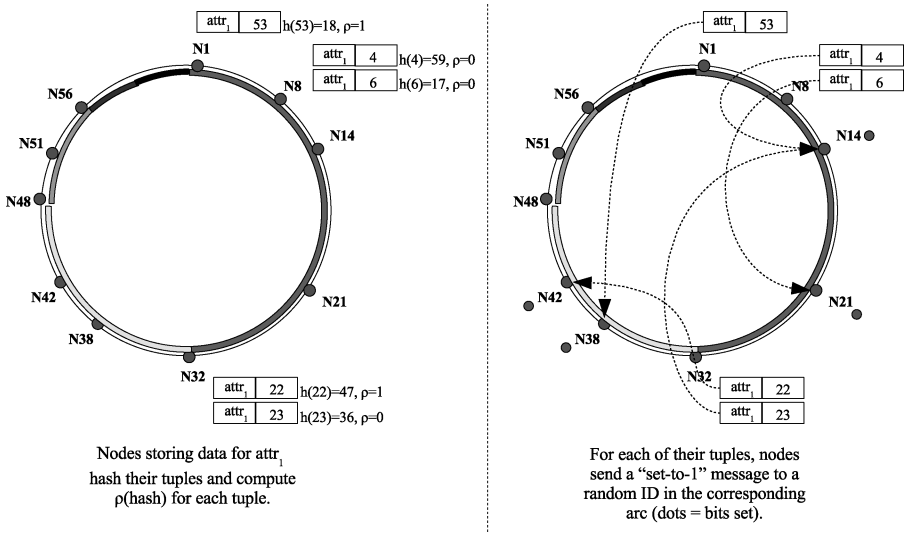


Fig. 10. Example of a DHS-based approach: Nodes insert their tuples in the DHS and the load is spread across the overlay.

3.3.1 *Single-Item Insertion.* Let o be an item to be inserted into the DHS. We proceed as follows:

(1) Compute the checksum $h(o)$ of o using the hash function of choice;
(2) feed the hash function output to the $\rho(\cdot)$ function to compute the bit position i to be set;
(3) choose a random ID in the interval $[thr(i), thr(i-1))$; and
(4) send a "single-set-bit" (SSB) message to the node responsible for that ID.

Figure 10 outlines this procedure. The SSB message consists of the duplet $<metric\_id, bit>$, where $metric\_id$ is an identifier uniquely identifying the metric to be estimated (e.g., part of the checksum of the name of the processed multiset—say, "number of unique files in the overlay"—as computed by a cryptographic hash function), and $bit = i$ denotes the position in the distributed vector of the bit that is to be set. As an example, for the worst case of 160 bits for the $metric\_id$ field (e.g., computed using SHA-1) and $\lambda = 80$ bits in the hash sketch bitmap vector (thus 7 bits or $\approx$1 byte for the bit field), the size σ_{SSB} of the payload of an SSB message is less than 22 bytes. In the average case, using 64-bit metric IDs and 40 bits per bitmap (i.e., $\lambda = 40$), thus allowing for up to $\approx 10^{12}$ distinct items per metric and up to $\approx 10^{19}$ different $metric\_id$'s, this figure drops to 5 to 9 bytes per message.

The hop-count cost to insert an item in an N-node DHT/DHS is in $O(\log N)$ ($\frac{\log N}{2}$ expected [Stoica et al. 2001]) as guaranteed by the underlying DHT, translating to an overall $O(\sigma_{SSB} \cdot \log N)$ bandwidth consumption. For a one-million-node overlay and the aforementioned worst-case numbers, this translates approximately to a worst-case network overhead of 20 hops (10 expected) and 440 overall bytes (220 bytes expected). Furthermore, 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 also takes $O(\log N)$ hops, but requires a more-or-less larger data transfer (should this be due to transferring of the whole item inserted or just of a (set of) index tuple(s) or pointers to the source node).

On the other end, a node receiving such a message:

(1) checks whether there is a local hash sketch stored for the given *metric_id*;
(2) if not, it creates a new, empty local hash sketch instance for this *metric_id*;
(3) sets the given bit position to 1; and
(4) resets the *time_out* field of the given bit position.

3.3.2 *Bulk Insertions.* Single-item insertions are well suited for streaming applications. However, they are a bad match for data-sharing/data management applications. In such a setting, each node usually stores hundreds or thousands of items, inserted all at once, and updated either seldomly or in batches. A prominent example of this type of application is file sharing, where nodes share all of their files at once, the value of each item (i.e., the contents of each file) is almost never changed throughout its lifecycle, and updates (e.g., file additions or deletions) are done in batches (usually by the P2P client, periodically checking the shared directories for new or deleted files).

In such settings, sending out a separate message for each of a node's items makes no sense. For this type of application we advocate bulk insertions. Note that the mapping scheme from bit positions to node IDs is the same for all metrics; this has the very beneficial side-effect that any given bit position maps to the same node-ID space interval for all metrics (see Figure 11), thus allowing us to group together multiple SSB messages for different metrics, if they all set the same bit position. Under this scenario, a node wishing to insert (a set of) its items to the DHS will:

(1) create a local hash sketch instance for each metric in the input multiset;
(2) insert all items in the input multiset to the corresponding hash sketch;
(3) if using iterative bit routing:
    (a) For each bit-position i set to 1 in any of the resulting sketches:
        i. choose a random ID in the interval $[thr(i), thr(i-1))$, and
        ii. send a "bulk-set-bit" (*BSB*) message to the node responsible for that ID.
(4) Else if using recursive bit routing:
    (a) Choose a random ID in the interval corresponding to bit 0; and
    (b) send a bulk-set-bit (*BSB*) message to the node responsible for that ID. That node will then recursively forward the BSB message until all arcs corresponding to set bit positions have been visited.

A BSB message for a given bit position i consists of a set of $<$*metric_id*,*bit*$>$ tuples, one for each distinct metric in the input data whose hash sketch after step 2 in the previous outline has a 1 at bit position i. For μ metrics and any given bit position, the size of this type of message is μ times the size of a

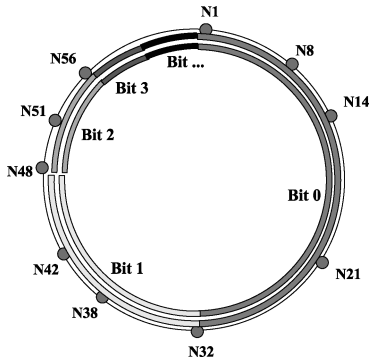


Fig. 11. Mapping of bit positions to nodes in the overlay: multiple bitmaps/metrics case. Bit positions for all metrics and all bitmaps are mapped to the same node-ID arcs as in the single-bitmap/single-metric case.

SSB message in the worst case (i.e., all μ metrics have a 1-bit at the given bit position). Thus, overall, in order for a node to insert all of its I_n items using one big bulk insertion, it will need to send out at most λ BSB messages (one for every bit position), each being the equivalent in bytes of μ SSB messages in the worst case (i.e., all λ bits of all μ metrics are set). This results in a worst-case hop-count cost in $O(\lambda \cdot \log N)$; on the other hand, the expected hop-count cost is $\log I_n \cdot \frac{\log N}{2}$ since, with I_n items per node, only $\log I_n < \lambda$ bits should be set on average. Nodes receiving such messages do the same procedure outlined in the single-item case, only now updating their local augmented hash sketches for all of the metrics included in the BSB message.

The two insertion schemes described before can naturally coexist in the same DHS instance. Specifically, each node can insert items both ways and even interchange between them. If, for example, DHS is used in a data-sharing setting but a node wishes to insert or update just a handful of items, it may choose to do so by multiple single-item operations, if the sum of the SSB message sizes is less than that of the corresponding BSB message. Conversely, in a streaming setting, a node may choose to group together multiple insertions/updates and use bulk operations, if input data creation is bursty.

3.3.3 *On Duplicate (In-)Sensitivity.* Both of the preceding algorithms start by computing the hash $h(o)$ of (each) object o to be inserted into the DHS, then feeding its outcome to the $\rho(\cdot)$ operator. This is in essence the step during which we can define the quantity to be estimated later on. For sake of presentation, assume that each object o consists of an identifier $o.id$, uniquely identifying it, and a set of attributes $o.attr_i$ (the identifier can also be computed from the values of a set of the object's attribute values). For example, in a music file-sharing application the object identifiers could be the hash of each file, with its attributes being its file name, the artist name, album, song title, year of issue, etc. Further remember that each node n is identified by DHT ID $n.id$.

The input to the hash function in the first step of the insertion algorithm defines what is being counted. If we use $o.id$ as the input, we in essence estimate the number of distinct $o.id$'s in the overlay, thus computing the distinct

number of files. On the other hand, we can use the concatenation of $o.id$ and $n.id$ as the input to the hash function, thus also counting files that have been replicated across nodes in the overlay. Furthermore, in order to account for multiple copies of each object on each node, a random suffix can be added to the previous concatenation, thus essentially making each object in the overlay a unique one. This technique can be extended to arbitrary combinations of the objects' attributes depending on what is the quantity of interest, allowing for both duplicate-sensitive and duplicate-insensitive counting.

3.3.4 *DHS Updates/Deletions.* We choose a soft-state approach for the maintenance of DHS data: A *time_out* field is attached to each bit position set to 1, defining a time-to-live interval for the given bit, and nodes periodically reinsert their items in the DHS. Deletion of data stored in a DHS is implicit: A bit position not refreshed within the *time_out* period is reset to 0. As a consequence, the hash sketch instance mentioned in step 2 before is an augmented version of the data structure mentioned in Section 2.2, consisting of the *metric_id* plus a set of β vectors. Each such vector is comprised of λ 32-bit integers.[4] each denoting the *time_out* value of the corresponding bit position, with bit positions with a *time_out* value of 0 considered to be unset. We shall call this data structure an *augmented hash sketch* for the rest of this article, to discern it from *normal hash sketches* as described in Section 2.2. As an example, for the worst case of 160-bit *metric_id*'s and 80 bits per bitmap vector, such a hash sketch instance will require 340 bytes of storage. Also, note that a node that has not been visited during item insertion (e.g., because it was not probed during the random ID choosing step) will not store any data for the given metric.

The *time_out* parameter can be viewed as the reciprocal of the aggregation window in streaming settings, while for a data management system, it can be selected depending on the ratio of insertions/updates to queries. The actual value of this parameter is largely application-dependent and poses an interesting trade-off.

Obviously, larger timeout 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 network resources are concerned. In any case, as depicted by the aforementioned worst-case examples, 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.

An interesting alternative is to allow nodes to set the time-to-live of their own items. In order to accomplish this, we add an extra *time_out* field to the SSB/BSB messages. Each node can then autonomously choose a value for this field on every item insertion. Target nodes will then "set" the requested bit in their appropriate local hash sketch instance, by setting its *time_out* value to the maximum between its current value and the value supplied by the incoming message. This addition increases σ_{SSB} to 26 bytes in the worst-case scenario mentioned earlier, or to 9 to 13 bytes in the average case.

[4]32-bits (=4 bytes) wide, following the *time_t* data type of the POSIX.1 standard.

3.3.5 *Replicating in the Target Neighborhood.* There are application domains were fault tolerance and/or data availability is of greater importance than sheer hop-count or bandwidth efficiency. Such applications definitely call for replication of the DHS data. 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. In the latter case, when inserting (or refreshing) a DHS bit, accessing a particular node in the bit's DHT interval, we also send the same SSB/BSB message to a number of α successors/predecessors of this node within the appropriate bit arc, using one-hop messages. For a replication factor of α, the worst-case hop-count cost for the single-item insertion case is in $O(\log N + \alpha)$ (expected $\frac{\log N}{2} + \alpha$); for the bulk insertion case, this formula becomes $O(\lambda \cdot (\log N + \alpha))$ (expected $\log I_n \cdot (\frac{\log N}{2} + \alpha)$). As we shall also see in the performance evaluation section, such a replication scheme is also beneficial during the DHS querying phase, for reasons to become clear shortly.

3.3.6 *Piggybacking DHS Messages on DHT Maintenance Traffic.* Remember that DHT nodes use PING/PONG messages and heartbeats to detect broken links in their routing state information, and that they periodically rebuild their routing tables as part of the stabilization protocol. In typical, widely deployed real-world implementations of DHT overlays, these heartbeat messages are sent out every 20 to 30 seconds per link, while the stabilization protocol is executed every 15 to 20 minutes. Building on the same observation as earlier, DHS maintenance traffic piggybacking (*MTP*) "hides" DHS-related messages inside DHT maintenance traffic. Depending on the usage pattern of a given DHS deployment, DHS operations may take advantage of these communications and send their data along with such DHT maintenance messages. As we have seen, DHS insertion/updates messages are of negligible size, so we expect that such piggybacking on DHT maintenance packets will eventually make DHS maintenance-free, without any significant increase in the overall network usage.

3.4 Increasing DHS Accuracy

So far we have been dealing with single-bitmap hash sketches. However, the estimates acquired by such simple sketches can typically be off the actual value by a binary order of magnitude [Flajolet and Martin 1985; Durand and Flajolet 2003]. As mentioned in Section 2.2, the accuracy of the estimations acquired from hash sketches improves with multiple bitmap vectors. Extending the previous algorithms for $\beta > 1$ (β being a power of 2) is straightforward. Insertion of an item o is done by selecting one out of the β vectors using the lower $\log \beta$ bits of $h(o)$, then using the remaining $\lambda - \log \beta$ bits as the input to the $\rho(\cdot)$ function to select the bit position i to be set in that specific bitmap. Note that we still do not change the mapping scheme from bit positions to node IDs (see Figure 11); this allows us to further group together SSB/BSB messages for different bitmaps if they all set the same bit position.

The SSB message must now be extended with a *vector_id* field, identifying the index of the vector being updated for the given metric. For a maximum

of 1024 vectors per hash sketch, the *vector_id* field will be 10-bits wide, thus σ_{SSB} will become ≈24 bytes (6 to 10 bytes in the average case with 256 bitmaps). Conversely, a BSB message for bit position i now consists of a set of $<metric\_id, bit, <bit\_vector>>$ tuples. The latter is β-bits wide and contains a 1 bit for every bitmap whose i'th bit is set. The worst-case size of this type of message will then be 148 bytes per metric, if all 1024 bitmaps for the given metric have a 1 at the given bit position. In the average case (i.e., 256 32- to 64-bit bitmaps), on the other hand, σ_{BSB} will be ≈28bytes per metric. Lastly, the storage requirement per estimated metric is multiplied by β, that is, ≈ 320kbytes per augmented hash sketch in the worst case (160-bit metric IDs, 1024 80-bit bitmaps), ≈65kbytes in the average case (32- to 64-bit metric IDs, 256 32 to 64-bit bitmaps).

Note that, since each item sets only a single bit, even with multiple bitmaps per hash sketch, due to the stochastic averaging technique of Flajolet and Martin [1985] and Durand and Flajolet [2003], the hop-count insertion cost remains the same for both the single-item and bulk insertion cases, irrespective of the number of bitmaps and/or metrics used. On the other hand, the bandwidth cost is again independent of the number of bitmaps and/or metrics for the single-item insertion case, and only slightly increases for the bulk insertion case.

3.5 Counting with DHS

With this infrastructure in place we will now focus on the design and implementation of the DHS estimation subsystem. First, we present algorithms for gathering DHS bit data from nodes in the overlay, in order to reconstruct the distributed hash sketch and compute the desired estimate. We then discuss multidimensional counting (i.e., simultaneously estimating multiple cardinalities) and present a simple bit-caching scheme, greatly reducing both the query hop-count cost and the load on overlay nodes. We will also discuss the issues arising from the random selection steps in the insertion algorithm and in bit probing (to be presented shortly) and attempt a simple balls-and-bins theoretical analysis.

Remember that estimating the number of distinct items in a multiset using hash sketches consists of: (i) finding the positions $R^{\langle i \rangle}$ of the rightmost 1-bits for super-LogLog, or of the leftmost 0-bits for PCSA, in $B[\cdot]$; (ii) computing their (truncated) arithmetic mean; and (iii) using Eq. (2) (super-LogLog) or Eq. (4) (PCSA) to compute an estimate of the cardinality of the multiset in question. The second and third steps are straightforward, provided we have dealt with the first step.

3.5.1 *Basic Bit Probing.* For any given metric, checking for the status of the bit positions of the corresponding distributed hash sketch follows the same concept as item insertions; that is, a node wishing to check whether a bit is set for a specific metric first selects a random ID in the ID-space arc corresponding to that bit position, and then sends a "single-check-bit(s)" (*SCB*) message to the node responsible for that ID. An SCB message consists of a $<metric\_id, bit>$ duplet, just like SSB messages. Nodes may choose to omit the *bit* field, in which case they request any and all bit data for the given metric stored by the target

```
1 Algorithm processSCB(SCBMessage msg) {
2    HashSketch augmentedHashSketch = storage.getHashSketch(msg.
         getMetricId());
3    if (localHashSketch != null) {
4       if (msg.hasBit())
5          return augmentedHashSketch.isBitSet(msg.getBit());
6       else
7          return augmentedHashSketch.getStandardHashSketch();
8    }
9 }
```

Fig. 12. Algorithm for processing single-check-bit(s) messages.

node. The latter, on receiving the SCB message, executes the steps outlined in Figure 12. In brief, the response either consists of a single bit, denoting the status of the requested bit position for the first case, or includes a complete hash sketch (computed by mapping *time_out* values in the augmented hash sketches back to bit values) for the PCSA estimator, or the positions of the most significant 1-bits in the bitmap vectors for the superLogLog estimator. Furthermore, the responding node includes in the answer the IDs corresponding to the lower and upper limits of its area of responsibility. This information is then used for the lifetime of the current query so as to avoid revisiting the same nodes during the random selection step. A null response denotes that the target node stores no information for the requested metric and/or bit. Nodes may also check for the status of multiple metrics with a single bulk-check-bit(s) (*BCB*) message, just like with bulk insertions and BSB messages. A BCB message consists of a set of SCB messages, one for every metric of interest. Nodes may selectively omit the *bit* field in any of the included SCB messages, with the same semantics as earlier. Again, the target node executes the procedure outlined in Figure 12 for every metric in the BCB message.

3.5.2 *Sequential vs. Binary-Search Bit Probing.* The simplest algorithm to find the $R^{\langle i \rangle}$ bit positions required for the computation of the hash sketch estimate would be to visit each of the intervals corresponding to the λ bit positions of the β distributed $B^{\langle i \rangle}[\cdot]$ bitmap vectors and check whether there is any item recorded there. In a P2P environment, network cost (in terms of both hop count and bandwidth consumption) is by far the major overhead factor, so such an algorithm would be an overkill. It turns out, however, that we can save several of these iterations by properly selecting the order in which bit positions are probed. To this extent, we have identified and implemented two algorithms, namely *Sequential Bit Probing* (or *SBP*) and *Binary-search Bit Probing* (or *BBP*). As the names imply, in SBP bit positions are probed in numerical order, while BBP follows a binary-search pattern.

For the PCSA estimator, $R^{\langle i \rangle}, i \in [0, \beta)$ stores the positions of the lower nonset bit for every bitmap, so an intuitive algorithm would be to sequentially probe bit positions in ascending order, starting from bit 0 (least significant bit) and proceeding towards bit position $\lambda - 1$ (most significant bit), until there is at least one 0-bit on every bitmap. On the other hand, for the superLogLog estimator, we are interested in the positions of the higher 1-bits, so intuitively we would

```
1  Algorithm estimateSBP(Metric metric_id) {
2      HashSketch hashSketch;
3      for (int bit = 0; bit < λ; bit++) {
4          Node targetNode = this.mapBitToNode(bit);
5          Result bitData = targetNode.processSCB(new SCBMessage(
               metric_id));
6          hashSketch.merge(bitData);
7          if (hashSketch.bitIsZeroForAllVectors(bit))
8              break;
9      }
10     int R[β] = hashSketch.computeRValues();
11     return α̃_β · β_0 · 2^{(1/β_0)·Σ*(R[·])}
12 }
```

Fig. 13. Algorithm *estimateSBP*: Estimate the number of distinct elements in a multiset using a DHS and sequential bit probing. R[β] stores the bit positions of the least significant 0-bits for PCSA or of the most significant 1-bits for superLogLog counting, respectively.

proceed in the opposite direction: probing bit positions in descending order starting from position $\lambda - 1$, until we either reach bit 0 or there is at least one 1-bit on every bitmap. Both algorithms work perfectly well in practice. However, the latter one has a subtle flaw: It starts probing from the smaller of the bit arcs (corresponding to bit position $\lambda - 1$), which typically consists of no more than a handful of nodes (one or two in the worst case). This leads to a load imbalance with regard to query processing (insertions and storage load balancing are not affected), with the most loaded nodes sharing almost the same cumulative load as would a rendezvous node. The PCSA estimator does not suffer from this, as it first probes the larger of the arcs, thus spreading the load on much more nodes (half of the total population), and terminates as early as possible.

To tackle this problem, we suggest changing the bit-probing algorithm so that it starts from bit position 0 moving to higher bit positions for both estimators, terminating when a bit position is found for which all vectors have a 0 bit (the final Sequential Bit Probing algorithm for both the PCSA and superLogLog estimators is summarized in Figure 13). The intuition behind this lies in the following logical empirical observation: For a given bit position i and a hash sketch implementation with β vectors $B^{\langle i \rangle}[\cdot]$, the probability that there is a 0-bit at position k of all β bitmaps (i.e. $B^{\langle m \rangle}[k] = 0, \forall m \in [0, \delta)$) *and* that there is a 1-bit at some position $l > k$ is negligible for $\beta \gg 1$ and becomes even lower for higher k (up to statistical fluctuations in the $R^{\langle i \rangle}$ values).[5] Moreover, specifically for the superLogLog estimator, even if such extremely rare cases do exist (we have yet to find one after several millions of runs of the algorithms over networks of various sizes and with input datasets of varying size, value domain, and skewness), they should be discarded during the truncation step.

The second bit-probing algorithm, Binary-search Bit Probing, builds on the same observation as before. The algorithm proceeds in three stages, outlined in Figure 14. First, it probes bit positions in $[0, \lambda)$ in a binary-search-like fashion, until it finds the lowest bit position p_h for which all vectors have a 0 bit. Then, if the PCSA estimator is in use, it probes bit positions in $[0, p_h)$

[5]A similar conclusion is also implied by the analysis in Durand and Flajolet [2003].

for the highest bit position p_l for which all vectors have a 1 bit. Finally, it reverts to a Sequential Bit Probing for bit positions (p_l, p_h) for the PCSA and $(p_l, 0]$ for the superLogLog estimator. The algorithm records the bit positions it probes during these phases so that bit positions that have already been visited during some stage are skipped in subsequent stages. The net result is a reduction in the overall estimation hop-count cost and a slightly better load distribution.

Take, for example, a multiset consisting of 2,500,000 unique items and assume we are using 512 32-bit bitmaps in the hash sketch. The highest bit position on average set in the hash sketch with this dataset is ≈16, as computed by Eq. (3). With 32 bits per hash sketch bitmap, we can expect BBP to probe on average 5 to 6 bit positions in the first stage of the algorithm (until it finds the lowest all-0 bit position, e.g., position 17), and as many bits in the third stage as necessary until it has found at least one 1-bit for every bitmap (the second stage is active only for the PCSA estimator). This can translate to anywhere between 12 to 21 bit positions being probed during estimation, while the standard sequential probing algorithm would probe 18 bit positions (bit positions from 0 up to 17). We expect BBP to outperform SBP for larger datasets and longer bitmaps, and to be at least as good as SBP in the smaller cases.

Note that the worst-case hop-count performance of both algorithms is independent of the number of bitmaps and number of items in the overlay: $O(\lambda \cdot \log N)$ hops in the worst case, with a $O(\log \frac{I}{\beta} \cdot \log N)$ expected figure,[6] for both bit-probing algorithms. This observation constitutes a core property of DHS and one of its great strengths: The counting hop-count cost formula is the same, independent of the number of bitmaps and dimensions (metrics). This is because the mapping of bit positions to ID-space intervals is the same for all bitmaps and all dimensions. Thus, by visiting a single 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, hence making multimetric estimation practically free (hop-countwise).

3.5.3 *Bit Caching.* By design, the choice of bit-probing scheme is orthogonal to that of the message routing mechanism; that means that we can have both iterative and recursive variants of both the sequential and the binary-search bit-probing algorithms, possibly also adding routing shortcuts to the picture, with different characteristics and trade-offs for each combination (more on this in the performance evaluation section). A particularly interesting combination is Sequential Bit Probing coupled with Recursive Bit Routing, as introduced in Section 3.2. This scheme lends DHS to a simple but powerful addition: caching of query result bits on the backward recursion path (coined Bit Caching, or BC). Low-level implementations issues, such as the cache replacement strategy, object time-to-live, etc., are orthogonal to our work and are thus not discussed in this article. What is of the essence, though, is the fact that over time, less and less counting operations will have to go all the way from bit 0 to the bit containing the last interesting bit position for the estimator used at the time.

[6]Note that $\log \frac{I}{\beta} < \log I \leq \lambda \leq L = \log N$.

```
1  Algorithm estimateBBP(Metric metric_id) {
2     HashSketch hashSketch;
3     int l = 0, overallLow = 0, h = λ, overallHigh = λ;
4     // Search for the lowest all-0 bit in [0,λ)
5     for (int bit = (l + h) / 2; l < h; bit = (l + h) / 2) {
6        Node targetNode = this.mapBitToNode(bit);
7        Result bitData = targetNode.processSCB(new SCBMessage(
             metric_id));
8        hashSketch.merge(bitData);
9        if (hashSketch.bitIsZeroForAllVectors(bit))
10          h = bit;
11       else
12          l = bit + 1;
13    }
14    overallHigh = l; // Bits above this are all 0
15    // Search for the highest all-1 bit in [0,overallHigh)
16    l = 0; h = overallHigh;
17    for (int bit = (l + h) / 2; l < h; bit = (l + h) / 2) {
18       if (hashSketch.hasInfoForBit(bit) == false) { // Skip the
             probe if we have already done it
19          Node targetNode = this.mapBitToNode(bit);
20          Result bitData = targetNode.processSCB(new SCBMessage(
                metric_id));
21          hashSketch.merge(bitData);
22       }
23       if (hashSketch.bitIsOneForAllVectors(bit))
24          l = bit + 1;
25       else
26          h = bit;
27    }
28    overallLow = l; // Bits below this are all 1
29    // Revert to the standard procedure for [overallLow,overallHigh)
30    for (int bit = overallLow + 1; bit < overallHigh; bit++) {
31       if (hashSketch.hasInfoForBit(bit) == false) { // Skip the
             probe if we have already done it
32          Node targetNode = this.mapBitToNode(bit);
33          Result bitData = targetNode.processSCB(new SCBMessage(
                metric_id));
34          hashSketch.merge(bitData);
35       }
36    }
37    int R[β] = hashSketch.computeRValues();
38    return α̃_β · β_0 · 2^((1/β_0)·Σ*(R[·]))
39 }
```

Fig. 14. Algorithm *estimateBBP*: Estimate the number of distinct elements in a multiset using a DHS and binary-search bit probing.

Moreover, since under the SBP+RBR scheme all queries commence with bit position 0 and proceed from one bit arc to the next in a recursive fashion, the probability of a cache hit on every step increases as the bit-arc size decreases. This add-on is beneficial for query hop-count cost, but also for the balancing of load across nodes; with caching, the query load is mitigated from the few nodes populating the higher bit positions to the rest of the node population. The performance evaluation section will showcase this behavior and its totally positive effect on the system characteristics.

3.5.4 *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, Flajolet and Martin [1985] and Durand and Flajolet [2003] feature a rigorous analysis of their statistical properties. Reciting the proofs found in these works is surely beyond the scope of this article. We refer interested readers to Flajolet and Martin [1985] and Durand and Flajolet [2003] and just mention here that the standard deviation is closely approximated by $1.05/\sqrt{\beta}$ for Durand and Flajolet [2003] and by $0.78/\sqrt{\beta}$ for Flajolet and Martin [1985].

As far as the latter cause is concerned, 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 item (corresponding to the bit being set). As a consequence, the DHS counting algorithm first selects a random node in the target ID-space interval, and probes it for any relevant tuple. If no such information is available at that node, the algorithm must proceed 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 N or more nodes, 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?

We turn thus to the computation of the upper limit of nodes to contact per-bit position of the DHS bitmaps. Assume that I' 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

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

PROOF (SKETCH). 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 I' items, a bin will be empty with probability $(\frac{N'-1}{N'})^{I'}$. 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 $(\frac{N'-2}{N'-1})^{I'}$, 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 $(\frac{N'-t}{N'-t-1})^{I'}$. Since each probe is independent of the others, the probability of choosing t empty bins in the first t probes equals

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

□

By solving Eq. (5) for t, we get that in order to choose a nonempty bin with probability of at least p, we have to visit at least $t \leq \tau = \lceil N' \cdot (1 - p^{\frac{1}{I'}}) \rceil$ bins/nodes. By setting $\xi = \frac{I'}{N'}$, we get $\tau = \lceil N' \cdot (1 - p^{\frac{1}{\xi \cdot N'}}) \rceil$. When using multiple ($\beta$) bitmap vectors, items are partitioned among the vectors, thus $\frac{I'}{\beta}$ items are inserted in N' bins, so the latter formula becomes $\tau_\beta = \lceil N' \cdot (1 - p^{\frac{\beta}{\xi \cdot N'}}) \rceil$. Finally, by taking replication into consideration, and assuming a replication degree of α, we get

$$\tau_\beta^\alpha = \lceil N' \cdot (1 - p^{\frac{\beta}{\alpha \cdot \xi \cdot N'}}) \rceil. \tag{6}$$

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

The default value of τ_β used in DHS for all intervals is 5 (constant), which suffices to guarantee that if a nonempty node exists in any given interval, it will be found with probability of at least 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., $I \geq \beta \cdot N$). Obviously, the default value of 5 also suffices when counting sets with a larger cardinality than the one dictated by the preceding. However, when counting smaller-cardinality sets, we may choose to either: (i) increase τ_β, according to Eq. (6); (ii) use a smaller DHT/DHS overlay for the specific operation via super nodes in hybrid P2P networks, building on a network architecture like the one presented in Ntarmos and Triantafillou [2004] (a trend that has lately started gaining supporters in the DHT world, too); or (iii) use explicit replication of DHS bits as outlined in the DHS insertion subsection. The worst-case

counting hop-count complexity thus becomes $O(\lambda \cdot (\log N + \tau))$, where τ is the upper bound of the number of iterations of the probing phase. For constant τ, as used in DHS, the hop-count complexity becomes again $O(\lambda \cdot \log N)$. Note that the cost of counting is, in any case, independent of the number of bitmaps.

3.6 Choice of Bitmap Length and Hash Function

As mentioned earlier (Section 2.2), hash sketches consist of a hash function and of a bitmap vector recording the output of the former, processed with the $\rho(\cdot)$ function, for each input element. For the first part, we can either use one of the many cryptographic hash functions in the literature (e.g., MD-4, MD-5, SHA-1, RMD-160, etc.) or directly employ whatever hash function is used by the underlying DHT to generate node and item IDs. Assume the output of the hash function is L bits long. Then, all bits in the output of the hash function are pseudorandom, since this is a core design principle of any randomizing/cryptographic hash function. This also fulfills the basic requirement of Durand and Flajolet [2003] and (practically) Flajolet and Martin [1985] with regard to hash functions used to insert items to the hash sketch. Given that the length of the hash sketch bitmap vector λ must be less than L bits, we can use part of the output of the hash function of choice to insert items to the Distributed Hash Sketch. For the remainder of this article, we shall assume that we are using the lower λ bits of the output of the hash function provided by the underlying DHT implementation.

On the other hand, the length λ of the bitmap vectors is a matter of debate. Since we are dealing with distributed, autonomous systems, it should be fixed when deploying a Distributed Hash Sketch instance. However, λ is an application-specific parameter whose value depends on the hash function used and the maximum cardinality to be estimated. More specifically, λ must be less than the output of the hash function used minus the logarithm of the number of bitmap vectors in the hash sketch, and bigger than the rounded-up sum of 3 plus the logarithm of the cardinality I to be estimated over the number of bitmaps in the hash sketch (see Eq. (3)). As an example, in order to calculate the cardinality of a multiset containing one million unique items with a single-vector hash sketch, we will need at least 23 bits, while one trillion unique items will take at least 43 bits per vector; for 512 bitmaps in the hash sketch, these figures drop to 14 and 34 bits, respectively. Moreover, note that an λ value (even much) larger than the minimum given by Eq. (3) has no negative effect on the accuracy of the computed estimate (it will merely result in the upper bits never being set to 1). Also note that, due to the mapping of bit positions to DHT nodes, we do not need to set a value of λ close to the aforesaid. Furthermore, with L-bit DHT keys and 2^L possible ID values, the Birthday Paradox limits the number of items in any DHT namespace (and hence the maximum estimated cardinality) to $2^{\frac{L}{2}}$. For example, most DHTs use a 160-bit hash function (specifically SHA-1), so λ will never have to be larger than 83 bits, dropping to 76 bits for 512 bitmaps in the hash sketch.

The previous analysis leads to the following simple algorithm: If on application design/deployment time there is some, even vague, notion of the maximum

Table II. Design Space Taxonomy: List of Configuration Parameters for DHS

Hash Sketch Parameters	⇒ Hash Function	($h(\cdot)$)
	⇒ Number of Bitmaps	(β)
	⇒ Number of Bits per Bitmap	(λ)
	⇒ PCSA *vs.* superLogLog Estimator	
Message Routing	⇒ Iterative *vs.* Recursive Bit Routing	(IBR/RBR)
	⇒ Routing Shortcuts	(RS)
DHS Insertions/Updates	⇒ Single-item *vs.* Bulk Item Insertions	(SII/BII)
	⇒ Target Neighborhood Replication	(TNR)
	⇒ Maintenance Traffic Piggybacking	(MTP)
Counting with DHS	⇒ Sequential *vs.* Binary-search Bit Probing	(SBP/BBP)
	⇒ Bit Caching	(BC)
	⇒ Retry Limit	(τ)

cardinality to be estimated by the given distributed hash sketch instance (even within a few binary orders of magnitude), and we fix the number of bitmaps in the hash sketch, we can compute λ by Eq. (3) as in the first example given before. Otherwise, we can use the upper bound on I implied by the Birthday Paradox and compute λ as in the second example.

3.7 Design Space Taxonomy—Putting It All Together

This section described Distributed Hash Sketches, our approach for distributing the data and computation chores of hash sketches over a DHT P2P overlay in a scalable and efficient manner. Our architecture offers a wealth of algorithms, techniques, and methods for a system designer to harness to her needs when deploying a DHS instance. First, there are parameters related to the base hash sketch: the hash function, the number and bit-length of bitmaps, and the actual estimator algorithm (superLogLog versus PCSA). Then, it is possible to choose among two bit-routing protocols, namely, iterative and recursive bit routing, and whether to use routing shortcuts. As far as item insertions and updates are concerned, there is the choice of single-item versus bulk insertions, while we can selectively replicate items in the target neighborhood and use message piggybacking to further lower the overall maintenance cost. For the counting part, there are again two candidate bit-probing algorithms: sequential and binary-search bit probing; there is also the option of caching estimation result bits (in tandem with recursive routing), and the retry limit parameter. Table II summarizes the resulting design space.

4. PERFORMANCE EVALUATION

We shall now present an in-depth evaluation of the performance, efficiency, and scalability of the algorithms and protocols discussed in this article. We first outline the experimental setup and methodology of the evaluation procedure, then discuss our results and findings.

4.1 Methodology

In Ntarmos et al. [2006] we had implemented a basic Chord-like DHT and DHS in an event-driven simulator coded in C++ from scratch. The simulator

supported all basic DHT primitives, that is, node joins and leaves/failures and data addition and deletion, and all the functionality of DHS and subsequent constructions. We used this simulator for preliminary performance evaluation purposes and sanity checks, and in order to observe how our solutions behave and scale in a fully controlled environment. We now implement DHS over FreePastry [FreePastry 2002], a publicly available implementation of the Pastry overlay [Druschel and Rowstron 2001] in the Java programming language. In Pastry (and FreePastry) lookups have a hop-count cost in $O(\log_{2^b} N)$, with b a positive integer. We configured FreePastry so that its routing cost is in $O(\log_2 N)$ (i.e., $b = 1$), so that hop-count results are easily comparable to other DHTs. Both implementations gave similar results, so we chose to show only results computed using the latter, to showcase the performance of our solutions as implemented in a real-world system. The source code of our implementation consists of approximately 5,500 lines of Java code[7] and is publicly available on the World Wide Web [FreeDHS 2006].

In both cases, the performance evaluation was carried out in the following steps.

(1) We populated the network with peers and allowed enough time for the DHT to stabilize. We ran P2P overlays consisting of 1000, 2500, 5000, and 10000 nodes in order to showcase the scalability properties of our approach and as examples of mid-range distributed systems, but larger networks are naturally supported by both our solutions and the code base.

(2) We generated the data items to be stored on nodes. As is standard practice in the relevant literature, we synthetically generated datasets that: (i) can test our system for various value distributions ranging from near uniform to highly skewed, and (ii) correspond to value distributions also observed in real-world systems [Saroiu et al. 2002]. Specifically, data item values were drawn from either a skewed Zipfian distribution with parameter $\theta = 1.2$ or a uniform distribution. Note that the value distribution skewness translates to multiple occurrences of highly popular values across items in the overlay, and to few or no occurrences for values in the tail of the distribution. The number of total data items was set to 250× the number of nodes in the overlay (i.e., 250,000 items for the 1000-node overlay, 625,000 items for the 2500-node overlay, etc.), so as to also showcase the scalability of our approach with regard to the size of the distributed multiset.

(3) We randomly assigned data items to nodes in the overlay with a uniform distribution, so that each node on average stores 250 items. This corresponds to the data contributed by each overlay node.

(4) We then had all nodes insert their items into the DHS.

(5) Finally, we selected random nodes and had them estimate the number of distinct items stored on the overlay. Without loss of generality, we have that: (i) In the configurations employing bit caching, there is no cache replacement, and (ii) these queries are executed one at a time, so that there is at

[7]Generated using David A. Wheeler's SLOCCount.

most one query active at any time in the system. Unless stated otherwise, the number of retries was set to 5 hops.

We used 128 and 512 bitmaps per hash sketch, and 32 bits per hash sketch bitmap. Note that the expected performance figures are directly affected by the bitmap length, and that with 32-bit bitmaps we are able to count over *trillions* of items. The results we shall present shortly were averaged over multiple runs for every case, to avoid statistical artifacts.

In brief, we focus on and present results for the following fields:

(1) hop-count cost for inserting items to the DHS;
(2) insertion load per input item and overall insertion load distribution;
(3) hop-count cost for queries (i.e., computing the estimate);
(4) node load per query and overall query load distribution; and
(5) accuracy of the computed estimate, as expressed by the average error versus a centralized estimate.

More specifically, our main focus is on distribution transparency. Thus, we mainly want to prove that the proposed solutions: (i) are as accurate as their centralized counterparts, (ii) impose low runtime overhead, and (iii) scale well with the size of the network and of the overall data collection of peers. We thus measure the accuracy and (message count) performance of the proposed solutions, attempting to showcase their applicability under the wide-scale distribution setting of a P2P overlay. More specifically, we are primarily interested in the number of hops required to do the estimation, the accuracy of the estimation itself, as well as the fairness of the load distribution across nodes in the network. We first instrumented FreePastry with the appropriate hooks to allow us to measure the number of hops each message needs to reach its destination node. We present hop-count results for both inserting items to the DHS and for doing the actual estimation. Moreover, we report on the mean error of the estimation, computed as the percentage by which the distributed estimation differed from the value that would be estimated by a hash sketch with the same parameters if all items were stored on a single machine (centralized case).

Another vital metric for distributed systems such as peer-to-peer data networks is the distribution of load across participating hosts. We measure the load on a given node as the insertion and query/probe "hits" on this node; that is, the number of times this node is the target of an insertion or query/probe operation. Conversely, we can think of the load imposed by an insertion/query in the system as the number of nodes visited during processing of the insertion/query. We thus further instrumented FreePastry to report on the aforesaid metrics; namely node insertion hits and node query hits. In order to visualize the impact of the different approaches on load distribution, we employed the Gini Coefficient [Damgaard and Weiner 2000], as advocated in Pitoura and Triantafillou [2007]. In brief, Pitoura and Triantafillou [2007] compare the appropriateness and performance of nine of the most well-known distribution fairness metrics (including the standard deviation, skewness, kurtosis, coefficient of variation,

maximum-to-minimum load ratio, fairness index, Lorenz curves, and the Gini Coefficient) for P2P applications. The Gini Coefficient emerged the winner, being intuitive, predictable, not oversensitive to changes in the input data skew, providing a global perspective on the underlying distribution, and exhibiting the smallest estimation error among its contenders.

The Gini Coefficient, GC, is the mean of the absolute difference of every possible pair of load values, taking values in the interval [0, 1), where a GC value of 0.0 is the best possible state, with 1.0 being the worst. The Gini Coefficient reflects the amount of imbalance in the system, so that, for example, a GC value of 0.25 translates to ≈75% of the total load being equally distributed across all nodes in the system. More specifically, if l_i, $1 \leq i \leq N$ is the load on the i^{th} node and μ_l is the mean of these loads, then

$$GC = \frac{1}{2 \cdot N^2 \cdot \mu_l} \sum_{i=1}^{N} \sum_{j=1}^{N} (|l_i - l_j|).$$

As a yardstick: (i) In a rendezvous-based approach in which nodes locally compute a hash sketch for their items and store the result on a handful (or just one) rendezvous node(s), the GC scores for the insertion and query load distributions are almost equal to 1; (ii) when (quasi-)uniformly selecting numbers in a fixed interval, as when assigning IDs to nodes placed on a DHT, the GC score of the distribution of arc lengths between consecutive DHT nodes (i.e., the distance of any two consecutive values) is ≈0.5; and (iii) if we subsequently (quasi-)uniformly select item values from the same interval as node IDs, and assign each new item to the node at the upper end of the arc the item value falls in, in a balls-and-bins scenario (as with the classic consistent hashing approach of DHTs [Karger et al. 1997]), then the distribution of items to nodes results in a GC score of ≈0.7. This means that, given that objects are assigned to nodes in the DHT using consistent hashing, the expected GC score for item insertion and access under uniform value and popularity assumptions (in essence, a best-case scenario) for a standard DHT configuration and mapping of items to nodes is ≈0.7.

4.2 Evaluated Configurations

Due to the breadth of the design space, we have identified the most interesting design choice combinations and present results only for them. We commence with the configuration of Ntarmos et al. [2006] and then gradually introduce our novel techniques contributed in this article one at a time, showing how each of them improves over the previous configuration. We will briefly describe each of these configurations, summarized in Table III, then discuss their effect on the various performance metrics. All of the configurations employ the superLogLog estimator.

The first configuration (case 1), coined DHS_0, corresponds to the settings in Ntarmos et al. [2006] and serves as the starting point on which this work builds. In brief, DHS_0 uses iterative bit routing without routing shortcuts, single-item insertions without replication, and sequential bit probing without bit caching and with a retry limit of 5 hops. In a second step, we turn routing shortcuts on

Table III. Evaluated System Configurations

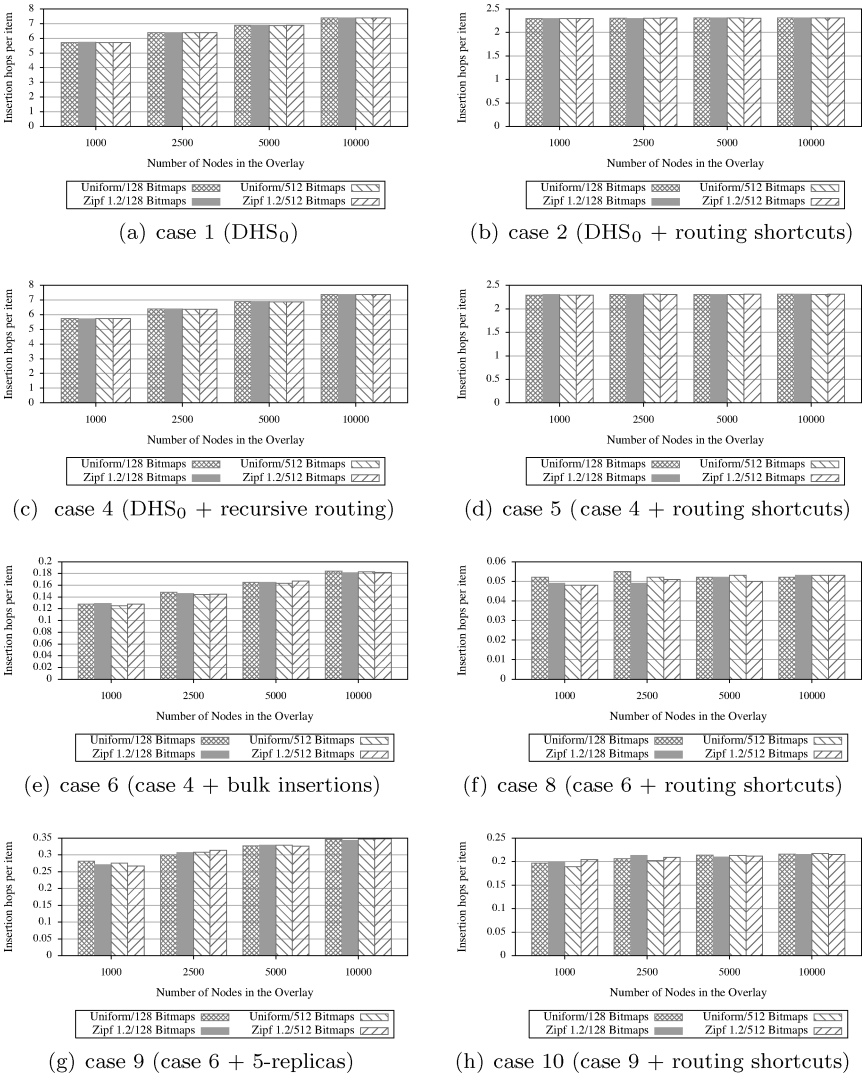
	Routing	Maintenance		Querying		Affects
Case No.	Bit Routing Algorithm	Items per Operation	No. of Replicas	Bit Probing Algorithm	Retry Limit	I: Insertions Q: Queries
Streaming Scenarios						
1	IBR	Single	0	SBP	5	–
2	IBR+RS	Single	0	SBP	5	I+Q
3	IBR	Single	0	BBP	5	Q
4	RBR	Single	0	SBP	5	I+Q
5	RBR+RS	Single	0	SBP	5	I+Q
Data Management Scenarios						
6	RBR	Bulk	0	SBP	5	I
7	RBR	Bulk	0	SBP+BC	5	Q
8	RBR+RS	Bulk	0	SBP+BC	5	I+Q
9	RBR	Bulk	5	SBP	0	I+Q
10	RBR+RS	Bulk	5	SBP	5	I+Q

(case 2) and repeat our measurements, to examine the effect of this optimization in an otherwise plain configuration of DHS. We report on both the insertion and query characteristics of these setups. Next (case 3) we add binary-search bit probing into the mix, substituting the sequential bit-probing algorithm of DHS_0. As this change only affects bit probing, we will show results only for queries. Case 4 builds again on the DHS_0 configuration, adding Recursive Bit Routing (RBR). We further add routing shortcuts (case 5) on top of this, in order to examine the synergy of recursive routing and routing shortcuts. We then turn to configurations using bulk insertions. First, case 6 examines the effect of bulk insertions on the hop-count cost and load distribution of DHS. In brief, this case is the same as case 4, only now dropping single-item insertions in favor of bulk ones. We only report on the insertion-related metrics for this case, as its behavior with regard to queries is the same as that of case 4. The great advantage of RBR is that it paves the way for bit caching on the query part, to be examined in case 7. We further examine the effect of routing shortcuts on this setup, in case 8. Next, we examine the effect of target neighbor replication on both the hop-count cost and load balancing of insertions and queries. Case 9 builds on the configuration of case 6, by setting the retry limit to 0, creating instead 5 replicas per bulk insertion, as explained in Section 3.3.5. As a last step (case 10), we enable routing shortcuts on this setup and set the retry limit back to its original value of 5 hops, so as to examine the effect of routing shortcuts in a configuration with replication and retries.

4.3 Insertion Characteristics

The first set of results pertains to the data insertion algorithms and protocols of DHS. First we examine the per-item insertion hop-count cost, then look into the distribution of insertion load across nodes in the overlay.

4.3.1 *Insertion Hop-Count Cost.* Figures 15(a) and 15(c) plot the per-item insertion hop-count cost for cases 1 and 4, respectively. Since in these



(a) case 1 (DHS_0) (b) case 2 (DHS_0 + routing shortcuts)

(c) case 4 (DHS_0 + recursive routing) (d) case 5 (case 4 + routing shortcuts)

(e) case 6 (case 4 + bulk insertions) (f) case 8 (case 6 + routing shortcuts)

(g) case 9 (case 6 + 5-replicas) (h) case 10 (case 9 + routing shortcuts)

Fig. 15. Average insertion hops per input item (lower is better).

configurations, each item insertion results in a (possibly) separate target node storing the related DHS bit information, the hop count closely follows the expected average path length of the underlying DHT (i.e., $\approx 0.5 \cdot \log N$ in an N-node DHT overlay with a diameter of $O(\log N)$[Stoica et al. 2001]). Figures 15(b) and 15(d), on the other hand, depict the hop-count cost for cases 2 and 5, respectively, corresponding to the configurations of cases 1 and 4 with routing shortcuts enabled. As we can see, it now takes on average ≈2.3 hops to insert an item into the DHS, irrespective of the number of nodes in the overlay. This roughly means that when routing a message to a random ID in a target bit arc, a shortcut to that arc is found at most on the second hop towards the target ID.

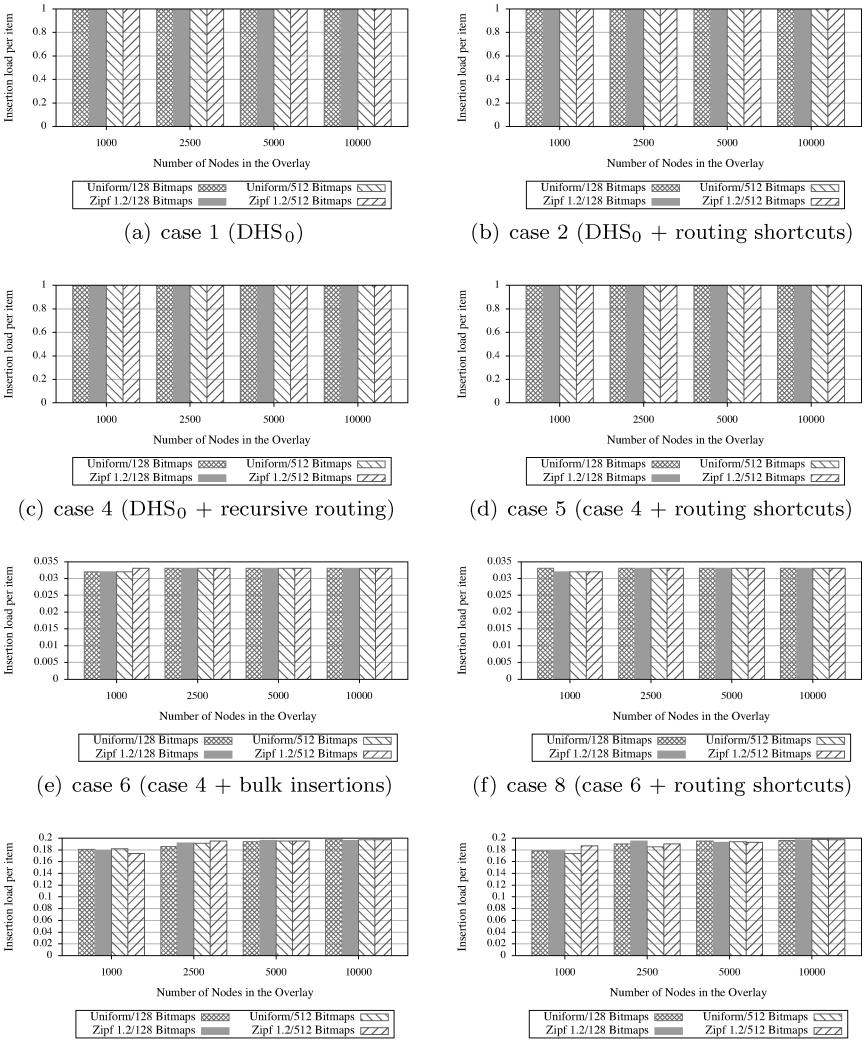
This results in huge savings with regard to hop-count costs, but has a definite negative effect on load balancing, as we shall see shortly.

Figure 15(e) (case 6) showcases why bulk insertions are the best choice when we can/need to insert multiple items at a time (as is the case for data management scenarios). As we can see, our expectations for a low insertion hop-count cost are met with flying colors; the average per-item insertion hop count has dropped to 0.12 to 0.18 hops: a reduction of a factor of approximately 40 to 45 compared to DHS_0, and of 13 to 18 compared even to the routing-shortcuts-enabled cases! This roughly translates to 30 to 45 hops (or 6 to 7 DHT lookups) needed by each node to insert all of its (on average) 250 data items into the DHS. Case 8 then adds routing shortcuts on top of this, and brings the insertion hop-count cost down to an average of $\approx$0.05 hops per item, a reduction of a factor of $\approx$2.5 to 3.5$\times$. This means that, for a random node to insert all of its items into the DHS, it will take $\approx$12.50 hops, or equivalently 2 to 3 DHT lookups.

We next examine the effect of replication on bulk insertion configurations. With a replication factor of 5, using the target neighborhood replication scheme outlined in Section 3.3.5, the per-item insertion hop count roughly doubles, as can be seen in Figure 15(g) for case 9. The end result is $\approx$60 to 90 hops (or 12 to 14 DHT lookups) to be paid by each node to insert all of its data items to the DHS. Adding routing shortcuts into the mix (case 10) reduces these numbers to $\approx$0.2 hops per item, or 50 hops (10 DHT lookups) to insert all of a node's item collection. This smaller decrease (factor of 1.2 to 1.8$\times$) compared to the corresponding decrease in the previous cases is due to the fact that routing shortcuts only save hops from long-distance routes (i.e., routing from the node inserting an item to a node in the target bit arc), but successor hops during replica creation cannot be avoided.

The hop-count figures reported in this section are actually very low, thus showcasing the low insertion overhead of our solutions. As a reference point, keep in mind that the sheer periodic maintenance of the node routing tables in most popular DHT implementations requires on the order of 160 lookups every few (e.g., 15) minutes, plus the (much more frequent) cost for maintaining the node's connections to its immediate successors/predecessors. This fact further allows us to apply the piggybacking technique of Section 3.3.6 to completely hide DHS insertion/update messages inside DHT maintenance traffic.

4.3.2 *Insertion Load Characteristics.* Remember that for cases 1, 2, 4, and 5, utilizing SII, each item insertion results in a single (possibly different) target node storing DHS bit info, so the per-item insertion load equals exactly 1. This is clearly depicted in Figures 16(a), 16(b), 16(c), and 16(d), respectively. Bulk insertions (cases 6 and 8) bring this figure down to $\approx$0.033 (Figures 16(e) and 16(f)): a reduction of a factor of $\approx$30$\times$; this is a direct result of the fact that only one bulk insertion message is needed for a node to insert all of its items, so the more the items in the bulk, the lower the per-item inflicted load. These numbers translate to a total load of $\approx$8 per node; that is, each node stored data on roughly 8 other nodes, or conversely each node was visited by 8 other nodes during the insertion phase. Last, with a replication factor of 5, the insertion load is naturally multiplied by $\approx$5, as we can see in Figures 16(g) and 16(h) for



(a) case 1 (DHS_0) (b) case 2 (DHS_0 + routing shortcuts)

(c) case 4 (DHS_0 + recursive routing) (d) case 5 (case 4 + routing shortcuts)

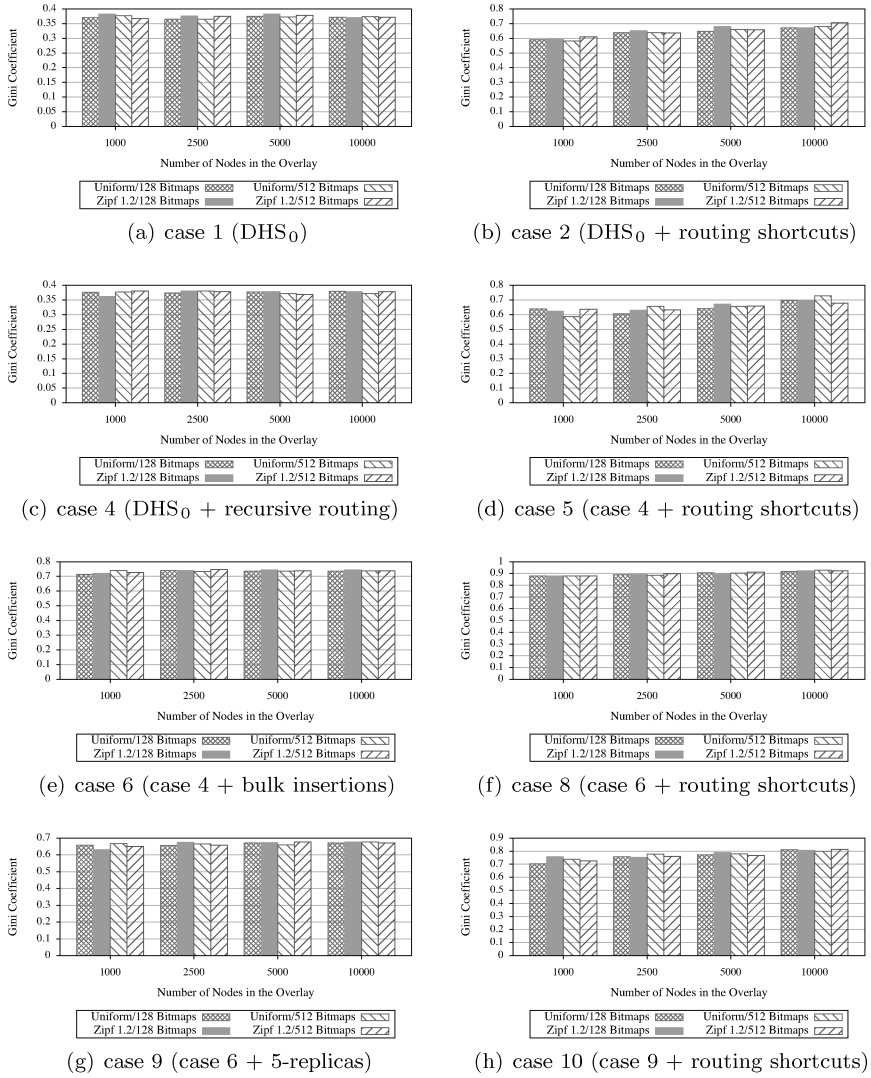
(e) case 6 (case 4 + bulk insertions) (f) case 8 (case 6 + routing shortcuts)

(g) case 9 (case 6 + 5-replicas) (h) case 10 (case 9 + routing shortcuts)

Fig. 16. Average insertion load per input item (lower is better).

cases 9 and 10, respectively, but still remains very low: ≈45 node hits in the 1000-node overlay, and ≈50 node hits in the 10000-node overlay).

Lastly we examine the distribution of this insertion load across nodes in the overlay. To this extent, we use the Gini Coefficient, as advocated by Pitoura and Triantafillou [2007] and described earlier. In cases 1 and 4, the insertion load distribution is excellent, achieving on average a 0.4 GC score (Figures 17(a) and 17(c)), rising to ≈0.6–0.7 when routing shortcuts are enabled (Figures 17(b) and 17(d)). Case 6, using bulk insertions, achieves a slightly higher GC score (Figure 17(e)), soaring just over 0.73. Case 8, using routing shortcuts, produces a slightly worse average GC score of ≈0.9 (Figure 17(f)). Replication improves



(a) case 1 (DHS_0) (b) case 2 (DHS_0 + routing shortcuts)

(c) case 4 (DHS_0 + recursive routing) (d) case 5 (case 4 + routing shortcuts)

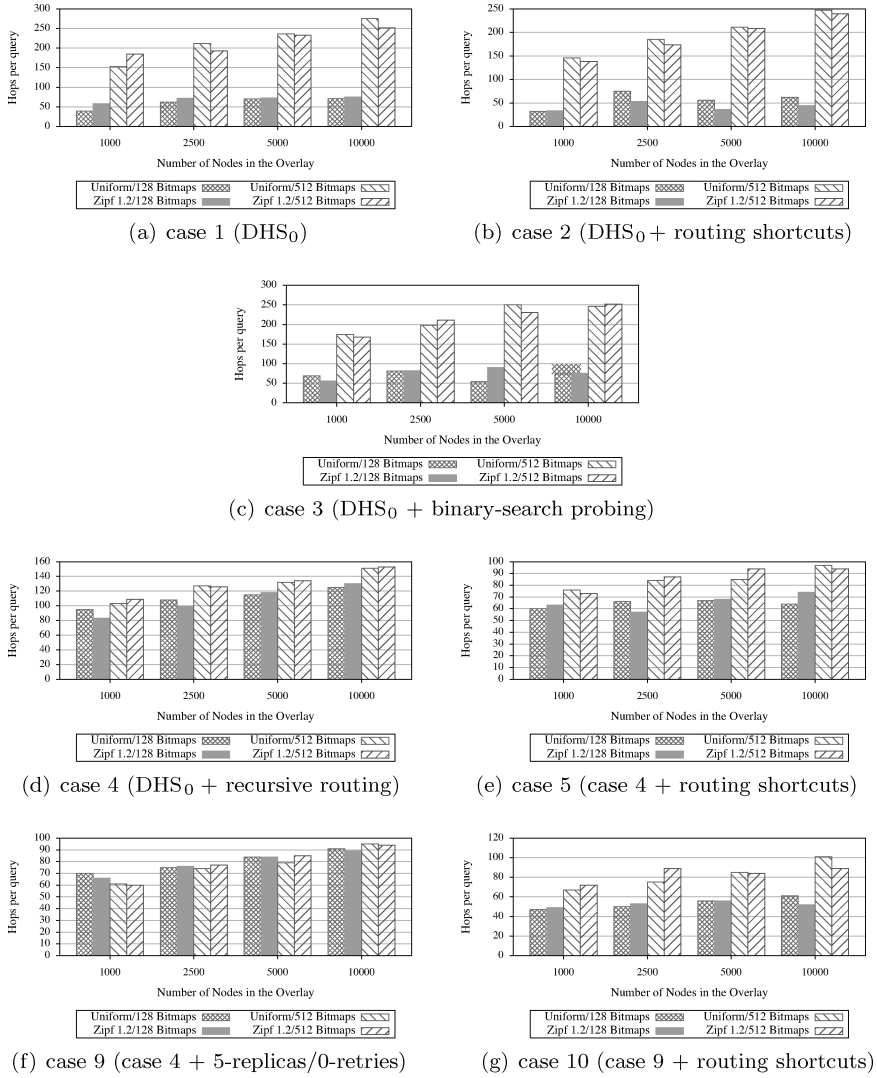
(e) case 6 (case 4 + bulk insertions) (f) case 8 (case 6 + routing shortcuts)

(g) case 9 (case 6 + 5-replicas) (h) case 10 (case 9 + routing shortcuts)

Fig. 17. Insertion load distribution: Gini Coefficient (lower is better).

the GC score to ≈0.69 on average, becoming 0.7 to 0.8 with routing shortcuts enabled (Figures 17(g) and 17(h), respectively). We would like to point out that all of these GC scores are quite good in practice. As a reference point, a DHT with pseudorandom IDs assigned to nodes and data items, and with items mapped to nodes using the standard consistent hashing technique [Karger et al. 1997] (a setup assumed to spread the storage load evenly across nodes in the overlay) achieves a storage GC score of just ≈0.7 on average.[8]

[8]Computed empirically using the PAST P2P application on top of FreePastry.



(a) case 1 (DHS_0) (b) case 2 (DHS_0 + routing shortcuts)

(c) case 3 (DHS_0 + binary-search probing)

(d) case 4 (DHS_0 + recursive routing) (e) case 5 (case 4 + routing shortcuts)

(f) case 9 (case 4 + 5-replicas/0-retries) (g) case 10 (case 9 + routing shortcuts)

Fig. 18. Average hops per query: noncaching cases (lower is better).

4.4 Query Characteristics

The next set of results are related to the actual estimation (query) algorithms and protocols of DHS. As in the insertion case, first we examine the query hop-count cost, then look into the distribution of query load across nodes in the overlay. Last, we examine the accuracy of the acquired estimates compared to the respective estimates computed in a centralized manner (i.e., if all data items were stored on a single host).

4.4.1 *Query Hop-Count Cost.* On the query front, cases 1 and 3 take on average ≈50 to 70 hops (i.e., the equivalent of approximately 10 DHT lookups)

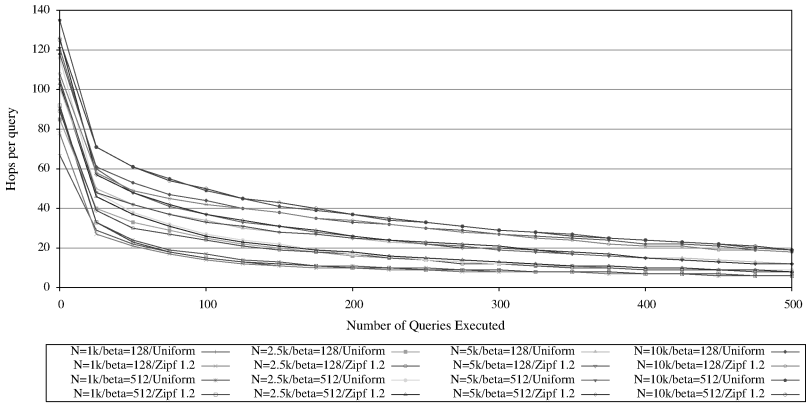
to compute the estimate with 128 bitmaps, and ≈160 to 250 hops (or 32 to 40 DHT lookups) with 512 bitmaps (Figures 18(a) and 18(c)), with the higher hop count for the latter being due to more retries. With more bitmaps in the DHS, less bits are set per bitmap and thus less DHS bit data items are spread across the network, so the chance that we randomly select a node not storing any data for the queried metric increases. Routing shortcuts (case 2) lower these numbers to 40 to 50 hops for 128 bitmaps and to 150 to 250 hops for 512 bitmaps. These figures verify our earlier claim for an $O(\lambda \log N)$ query hop-count cost in the worst case. Note that we expect case 3, featuring the binary-search bit-probing algorithm, to outperform the plain DHS instances for larger test cases.

Figure 18(d) plots the query hop-count cost for case 4. Now estimations take on average ≈80 to 120 hops with 128 bitmaps (or the equivalent of 16 to 18 DHT lookups), with the 512-bitmap case dropping to ≈100 to 160 hops (i.e., 20 to 32 DHT lookups). Routing shortcuts (case 5) further reduce these numbers to 40 to 60 hops for 128 bitmaps and to 65 to 90 hops for 512 bitmaps (Figure 18(e)).

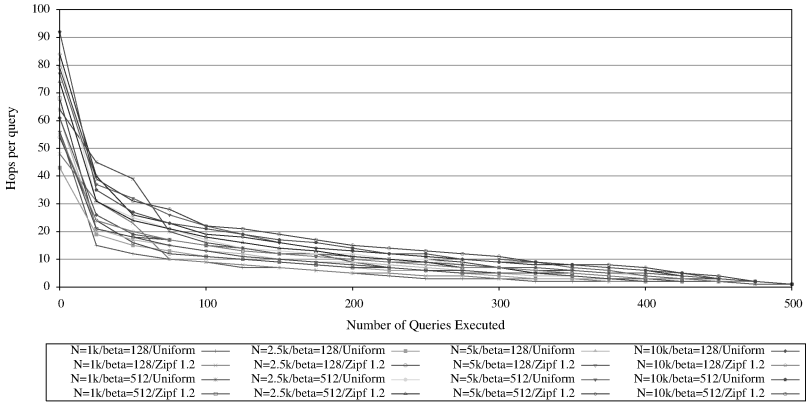
Next in line are cases 9 and 10 (Figures 18(f) and 18(g)), featuring our replication technique, coupled with routing shortcuts for the latter. As we can see, replication is also beneficial for the query hop-count cost, as there is a 30%–40% decrease for case 9 compared case 4 (the equivalent case without replication). On the other hand, the hop-count cost of case 5 (i.e., case 4 plus routing shortcuts) is already as low as we can achieve with our arsenal; thus, this replication on top of this has virtually no effect on the average hop-count cost, as can be seen in Figure 18(g) compared to Figure 18(e).

We have left the caching-enabled cases (i.e., cases 7 and 8) for last. Figures 19(a) and 19(b), respectively, plot the average query hop-count cost for these two cases over time. As we can see, there is a sharp drop in the average hop count for the first 25 to 50 queries as the cache is getting warm, evening out at approximately 5 to 20 hops for case 7 and 1 hop for case 8 after 500 queries. We would like to note that there is no cache invalidation in these experiments. This was an explicit choice made in order to showcase the effect of caching on the hop-count cost (and on query load distribution, to be presented shortly). Assuming a time-to-live period for cached results and given a query arrival rate, it is possible to compute how many queries on average will be answered from the cache and directly find out the respective hop-count cost earnings from these figures.

4.4.2 *Query Load Distribution.* Figures 20 and 21 plot the total load imposed by each query in the system for the various configurations, averaged in 25-query windows. A common denominator of all noncaching configurations except case 9 (i.e., Figures 20(a) through 20(e) and 20(g)) is that the total load is higher for 512-bitmaps cases than it is with 128 bitmaps. The reason why this happens for the nonreplicating configurations is also the cause of the higher hop-count cost for these cases: More bitmaps result in less items being inserted per bitmap, in turn resulting in a higher possibility of visiting nodes that do not store bit data for all bitmaps, and thus forcing the algorithm to retry the probing. This is also the reason why case 9 in Figure 20(f), having a retry limit of 0



(a) case 7 (case 4 + bit caching)

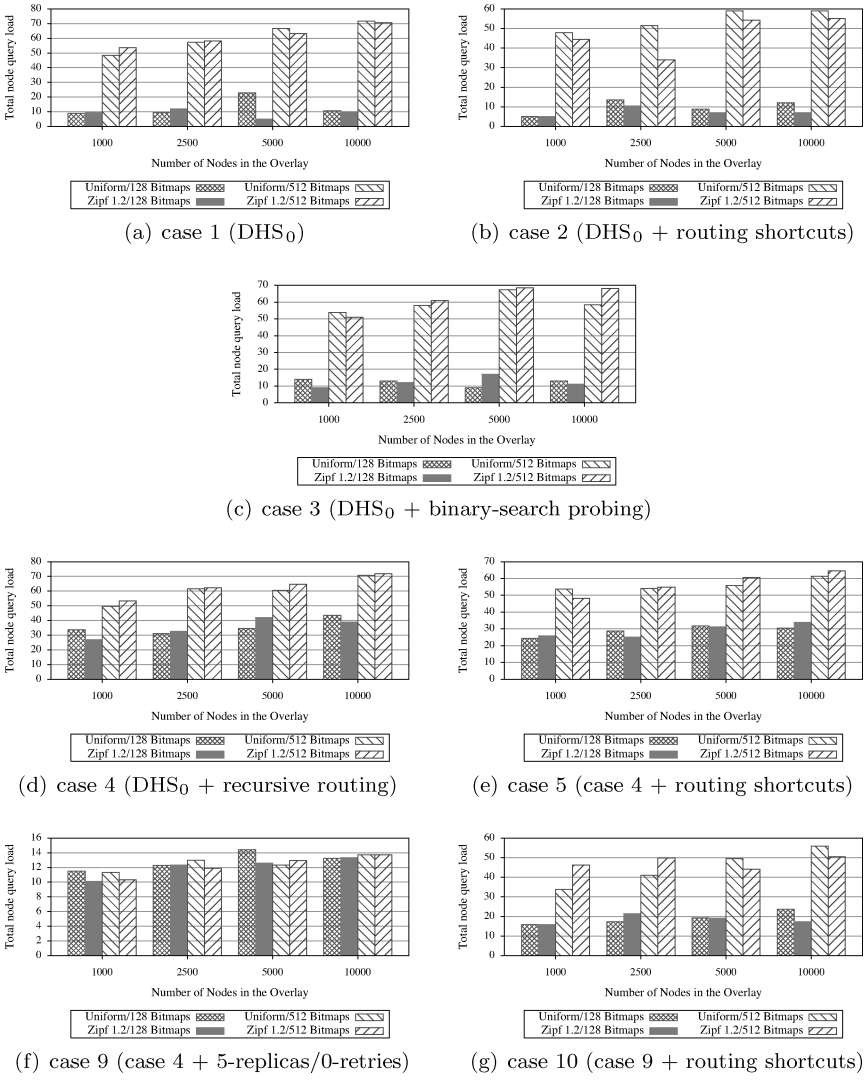


(b) case 8 (case 7 + routing shortcuts)

Fig. 19. Average hops per Query: caching cases (lower is better).

hops, does not suffer this setback, achieving on the contrary a much lower total load per query, but slightly losing in accuracy, as we shall see shortly. Lastly, Figures 21(a) and 21(b) show the per-query load for the caching cases (cases 7 and 8, respectively). The preceding conclusions on the effect of the number of bitmaps and routing shortcuts on the per-query load still hold, but now we also see how caching decreases the per-query load in the long run. We can also see that routing shortcuts do not affect the performance of DHS with regard to this metric; although they cut down the hop-count cost for long-distance lookups, they have no effect on the number of nodes that have to be visited during query evaluation.

We will now examine the effect of the various techniques and algorithms contributed in this article on the fairness of the distribution of query load across nodes in the overlay. The first three figures plot the evolution of the GC score of the query load distribution as queries are executed in the system, for the configurations of cases 1, 2, and 3 (Figures 22(a), 22(b), and 22(c), respectively). It takes on average 100 to 200 queries for the GC score to settle



(a) case 1 (DHS_0) (b) case 2 (DHS_0 + routing shortcuts)

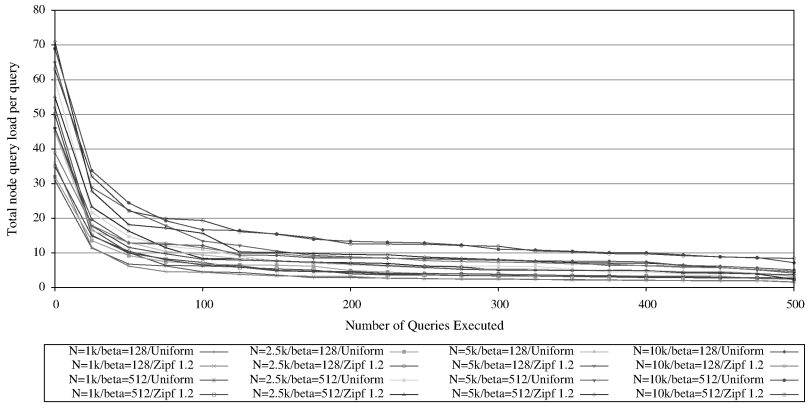
(c) case 3 (DHS_0 + binary-search probing)

(d) case 4 (DHS_0 + recursive routing) (e) case 5 (case 4 + routing shortcuts)

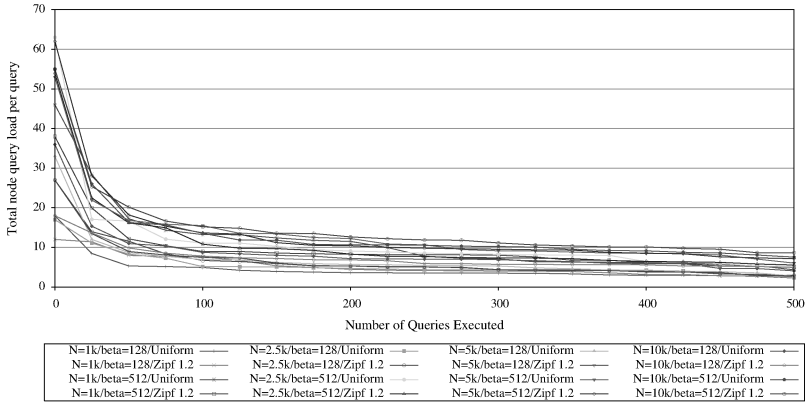
(f) case 9 (case 4 + 5-replicas/0-retries) (g) case 10 (case 9 + routing shortcuts)

Fig. 20. Average node load per query: noncaching cases (lower is better).

for the configurations using 512 bitmaps (the curves for 128 bitmaps are close to 1, but drop to ≈0.9 to 0.95 after approximately 2000 queries, not shown on these figures for reasons of presentation). The reason why those cases with 512 bitmaps fair better than those with 128 bitmaps is due to the fact that less bits are set per DHS bitmap, thus bit probing terminates earlier and in larger bit arcs; note that the last few bit positions are mapped to a handful of nodes (with the most significant bit being mapped to a single node in the worst case). However, this situation can easily be alleviated with thresholding (see Section 3.1.3). Also note that: (i) Routing shortcuts have a negative effect on load distribution fairness, and that (ii) the two cases not using shortcuts achieve a GC



(a) case 7 (case 4 + bit caching)



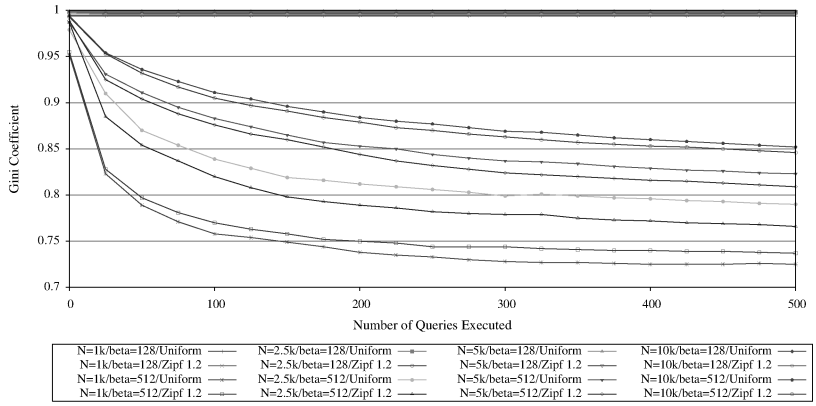
(b) case 8 (case 7 + routing shortcuts)

Fig. 21. Average node load per query: caching cases (lower is better).

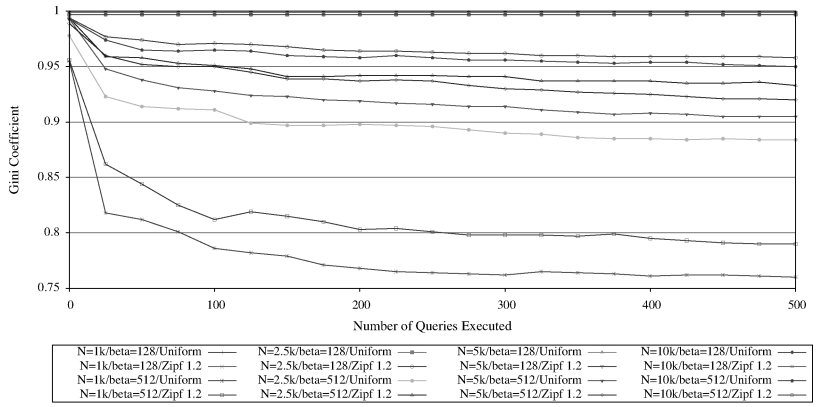
score of 0.73 to 0.85, which is on par with the GC of the load distribution of popular DHTs under uniform value and item popularity distributions ($\approx$0.7).

The load distribution fairness for cases 4 and 5 is plotted in Figures 23(a) and 23(b). Compared to the previous cases, we can see that recursive routing greatly improves the load balancing: The system now achieves a 0.87 GC score in the worst case (even for 128-bitmap cases), and a 0.68 GC score in the best case (Figure 23(a)). Moreover, note that the slope of the 128-bitmaps cases (especially for case 4) is still somewhat steep at the 500-queries point, meaning that the GC score has not yet stabilized. We have further tested these configurations for several thousands of queries (not shown for reasons of presentation), and found that the GC score settles around 0.65 for the 512-bitmap cases, and around 0.75 for the 128-bitmap cases.

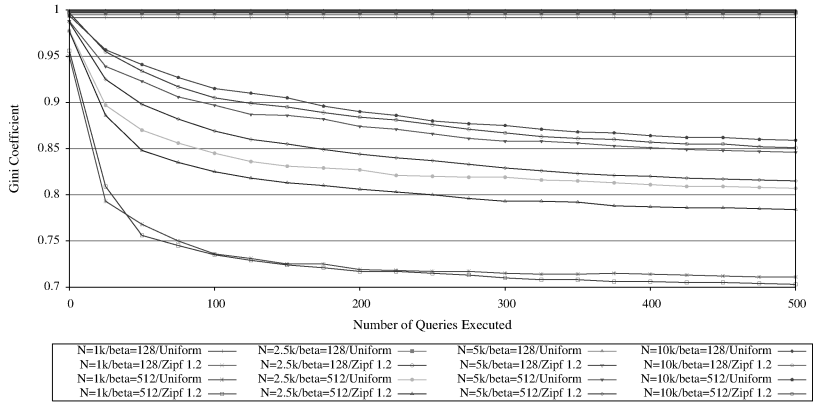
The effect of replication on query load distribution is depicted in Figures 24(a) and 24(b), plotting the GC score for cases 9 and 10, respectively. For the latter case, the achieved GC score for the query load distribution ranges from 0.82 to 0.91 for 512-bitmaps, and from 0.93 to 0.96 for the 128-bitmaps configurations.



(a) case 1 (DHS_0)

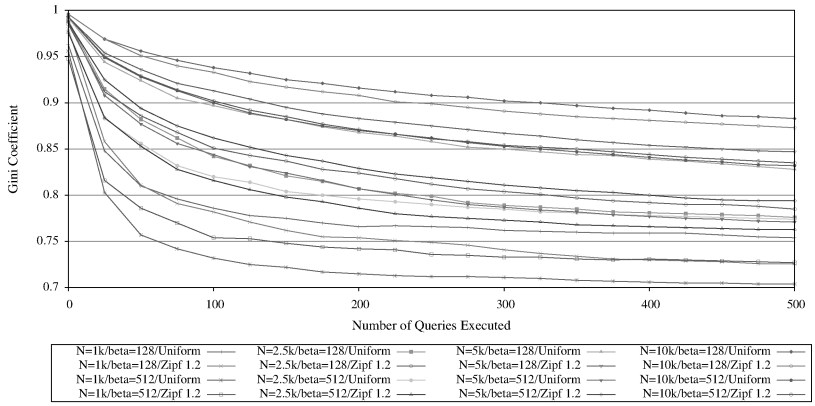


(b) case 2 (DHS_0 + ruting shortcuts)

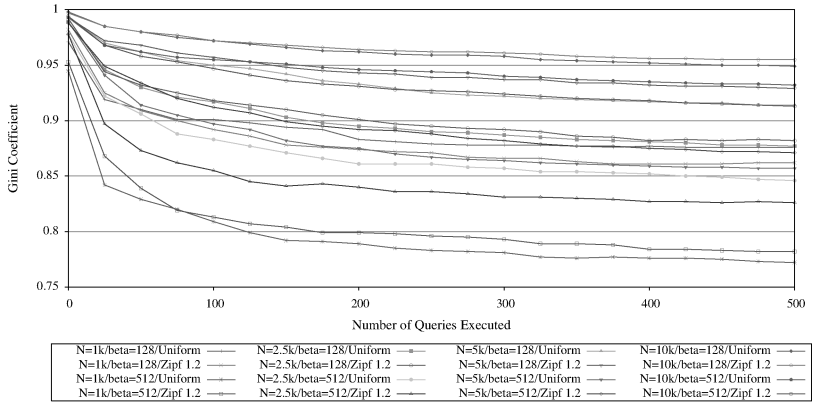


(c) case 3 (DHS_0 + binary-search probing)

Fig. 22. Query load distribution (iterative routing): Gini Coefficient (lower is better).



(a) case 4 (DHS_0 + recursive routing)

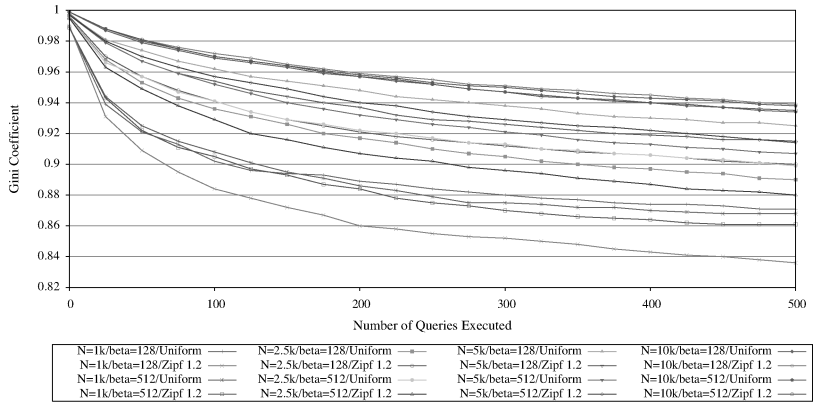


(b) case 5 (case 4 + routing shortcuts)

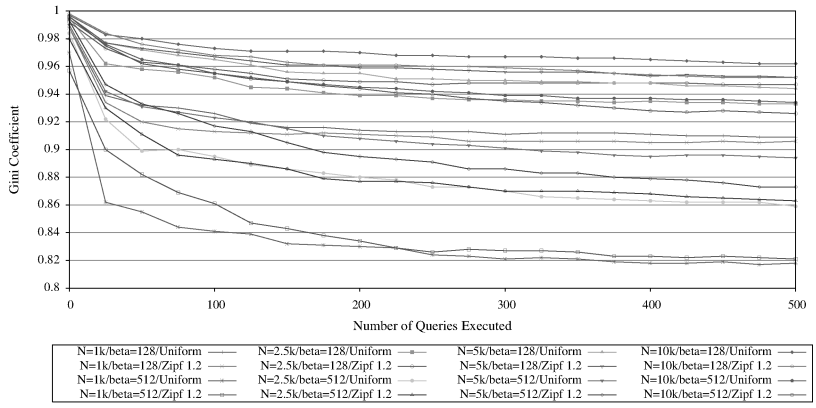
Fig. 23. Query load distribution (recursive routing): Gini Coefficient (lower is better).

For the former case, we can clearly see that the GC curve is still quite steep at the 500-query point; after some more thousands of queries, the GC score settles at around the same level as in case 4 previously. Again, it is obvious that recursive routing is beneficial (compare to case 1, for example), and that routing shortcuts result in a worse load distribution.

Again, we have left the configurations utilizing bit caching for last. Bit caching is (along with thresholding) the best way for spreading the load across nodes. Figure 25(a) plots the GC scores for case 7. We can see that the 512-bitmaps configurations achieve a GC score of $\approx$0.45 after 500 queries, with the 128-bitmap setups having reached 0.55 to 0.75 scores by that time, and both sets of curves dropping to $\approx$0.4 after a few thousands of queries. Even with routing shortcuts on (Figure 25(b)), the achieved GC score is approximately 0.55 to 0.66 at the 500-query point for 512 bitmaps, and 0.75 to 0.85 for 128 bitmaps (both dropping to around 0.55 after a few thousands of queries).



(a) case 9 (case 4 + 5-replicas/0-retries)

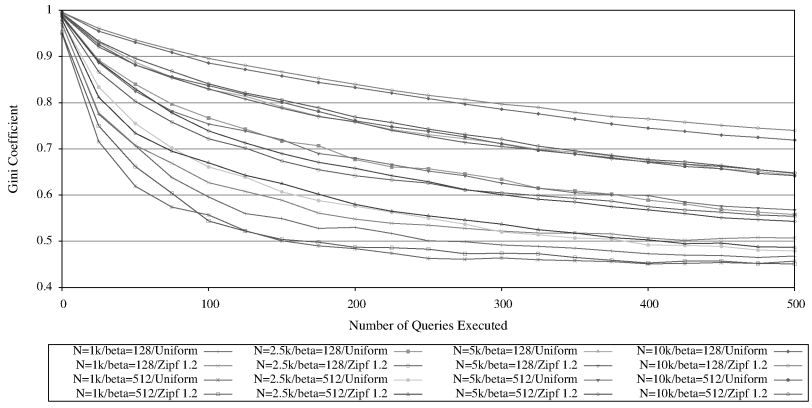


(b) case 10 (case 9 + routing shortcuts)

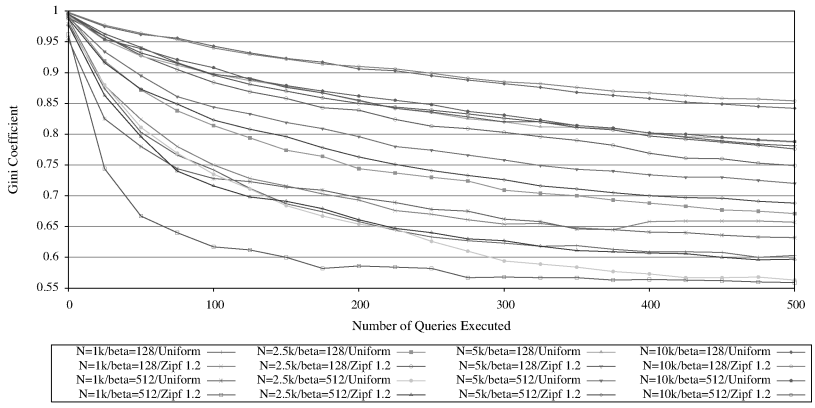
Fig. 24. Query load distribution (replication): Gini Coefficient (lower is better).

4.4.3 *Estimation Accuracy.* Lastly, we examine the accuracy of the computed estimates versus a centralized estimator. In all but the last three cases, the average error is below 1%, climbing to just 3.5% in the worst case for the last three configurations, thus also proving our claim for distribution transparency. Note that these figures roughly equal the average error of the centralized sketch algorithms, which are already deemed accurate by the research community. Table IV summarizes the average error for the various configurations. Note the negative effect of routing shortcuts on accuracy (numbers for RS-featuring cases shown in italics), and the slightly larger error for 512 bitmaps compared to 128 bitmaps caused by the lower-than-needed number of retries.[9]

[9]With a higher number of bitmaps in the distributed hash sketch, less items are assigned to each bitmap and thus a higher number of retries are needed to attain the same accuracy as when using less bitmaps; see Section 3.5.4 for the details.



(a) case 7 (case 4 + bit caching)



(b) case 8 (case 7 + routing shortcuts)

Fig. 25. Query load distribution (bit caching): Gini Coefficient (lower is better).

4.5 Discussion

The basic DHS instance (DHS_0), first presented in Ntarmos et al. [2006], already enjoyed a fairly low insertion and query hop-count cost, achieving excellent accuracy and storage load balancing while also enjoying better query load balancing over the standard rendezvous-based approach. In this work we have contributed several additional methods which introduce significant performance improvements during query execution and data insertion and for hop-count costs as well as for load balancing. The following summarizes the key performance results from our experimental study.

First came binary search bit probing. For the dataset of our experiment, binary-search bit probing resulted in a slightly worse hop-count cost (Figure 18(c)), as this algorithm may end up probing more bits than SBP. It will surely probe some unset bits during the binary search phases, and it still has to do a sequential partial pass after that to complete the resulting hash sketch. For the same reason, we observed an artificially slightly better load distribution (Figures 20(c) and 22(c)), as load is spread across more nodes. We

Table IV. Average Estimation Error (%)

Case No.	$\beta = 128$		$\beta = 512$	
	Uniform	Zipf 1.2	Uniform	Zipf 1.2
Case 1	0.00	0.00	0.00	0.00
Case 2	*0.01*	*0.00*	*0.26*	*0.25*
Case 3	0.03	0.04	0.24	0.23
Case 4	0.00	0.00	0.00	0.00
Case 5	*0.17*	*0.12*	*0.54*	*0.52*
Case 6	0.00	0.02	0.04	0.05
Case 7	0.05	0.01	0.01	0.08
Case 8	*1.62*	*1.54*	*2.33*	*2.41*
Case 9	0.87	0.92	2.54	2.48
Case 10	*2.11*	*2.56*	*3.27*	*3.49*

expect BBP to outperform SBP for larger datasets and longer bitmaps, and to be at least as good as SBP in the smaller cases. Next we examined recursive bit routing. RBR had the same hop-count cost (Figures 15(c) and 18(d)) and load distribution properties (Figures 17(c) and 23(a)) as the plain DHS setup, as we still have to contact one (random) node per input item during insertions, and still have to sequentially locate the highest all-1 bit position. However, keep in mind that RBR has the less obvious advantages of better resilience to network-level errors and lower per-hop latency [Dabek et al. 2004; Rhea et al. 2004] if using a network proximity-aware DHT (such as Pastry and FreePastry).

All three configurations presented so far use single-item insertions. This setting is well suited for streaming scenarios, where nodes wish to record their observations in the DHS as data flies by. In a data management scenario, on the other hand, grouping multiple item insertions into a single insertion operation is very beneficial with regard to the hop-count cost and node load. We thus turn to bulk insertion configurations. Remember that with bulk insertions, each node first populates a local hash sketch with its items and, coupled with recursive routing, it then sends a single bulk insertion message around the DHT overlay. We saw huge improvements in the insertion hop-count cost, and a slight degradation in insertion load balancing in case 6 (Figures 15(e), 16(e), and 17(e)). Bit caching, the next step in our configurations, in addition to lowering the query hop-count cost over time, has the added positive effect of spreading the query load across all nodes in the overlay (Figures 19(a), 21(a), and 25(a)). Most notably, since query results are cached all the way back to the bit arc corresponding to bit 0, there is a good chance that queries may get answered on the first bit probe. Moreover, the probability of a cache miss reduces by roughly 50% on every jump to the next bit arc, since the sizes of subsequent bit arcs differ by a factor of 2. Target neighborhood replication was next; the intuition behind the configuration of case 9 is trading off insertion hops for query hops, and gaining, some more DHS bit data redundancy in the process. We observed a higher insertion hop-count (Figure 15(g)) and a better insertion load distribution (Figure 17(g)) compared to case 6, and a lower query hop-count (Figure 18(f)) and possibly worse query load distribution (Figure 24(a)).

Finally, in those cases making use of routing shortcuts, most operations were resolved by single-hopping to nodes registered in the routing tables of the inserting/querying node, resulting in a dramatic cut on the hop-count costs. On the other hand, the pool of random nodes that can possibly be contacted by a given node is limited to those in the node's routing table, thus also affecting negatively the load balancing and probably the accuracy of the estimate. This is the main reason why we bumped the retry limit back to its original value in case 10; if always selecting a target among those few nodes in the routing table, then the possibility of choosing a node storing bit data for all bitmaps in the DHS is quite low.

5. CONCLUSIONS

In this article we presented Distributed Hash Sketches (or DHS): a novel, fully decentralized, scalable, and efficient mechanism capable of providing estimates on the cardinality of multisets in Internet-scale information systems. DHS is, to our knowledge, the first method 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 PCSA or the superLogLog 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 and scalability of the counting operation. In brief, the asymptotic performance of item insertion and distributed computation of the estimate is logarithmic in the number of nodes and independent of the number of bitmaps, items, and metrics. Moreover, our optimizations yield a query/insertion hop count equivalent to a very small number of DHT lookups. The insertion/storage load distribution is also very good, on par with the storage load balancing achieved in DHT overlays. The query/estimation load distribution can be adjusted, even approaching a uniform distribution in the long run when bit caching is enabled. Last, the accuracy is also excellent, with the average error under 3.5% in the worst case.

The design space of our DHS framework is quite large, providing a rich set of methods and mechanisms, offering trade-offs between hop-count efficiency, estimation accuracy, and fair load distribution. A real-world deployment of DHS would probably choose some amalgam of the clearcut configurations presented in the performance evaluation section. For example, an interesting configuration for a streaming scenario could consist of a fusion of iterative routing, single-item insertions, replication, and routing shortcuts to insert items, and plain recursive routing without shortcuts plus binary-search bit probing for the query part. On the other hand, for a data management setting, a configuration with bulk insertions piggybacked on DHT traffic, along with recursive

sequential probing with caching, seems like a winning synthesis. We intend to further examine such combinations in the future.

REFERENCES

ABERER, K., DATTA, A., HAUSWIRTH, M., AND SCHMIDT, R. 2005. Indexing data-oriented overlay networks. In *Proceedings of the International Conference on Very Large Databases (VLDB)*.

ALON, N., GIBBONS, P. B., MATIAS, Y., AND SZEGEDY, M. 1999. Tracking join and self-join sizes in limited storage. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*.

ASPNES, J. AND SHAH, G. 2003. Skip Graphs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

BABAOĞLU, Ö., MELING, H., AND MONTRESOR, A. 2002. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the IEEE International Conference on Distributed Computing and Systems (ICDCS)*.

BAR-YOSSEF, Z., JAYRAM, T., KUMAR, R., SIVAKUMAR, D., AND TREVISAN, L. 2002. Counting distinct elements in a data stream. In *Proceedings of the International Workshop on Randomization and Approximation Techniques (RANDOM)*.

BAWA, M., GARCIA-MOLINA, H., GIONIS, A., AND MOTWANI, R. 2003. Estimating aggregates on a peer-to-peer network. Tech. rep., Computer Science Department, Stanford University.

BAWA, M., GIONIS, A., GARCIA-MOLINA, H., AND MOTWANI, R. 2004. The price of validity in dynamic networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

BEYER, K., HAAS, P. J., REINWALD, B., SISMANIS, Y., AND GEMULLA, R. 2007. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. 2004. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*.

CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, R. 1998. Random sampling for histogram construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

CONSIDINE, J., LI, F., KOLLIOS, G., AND BYERS, J. 2004. Approximate aggregation techniques for sensor databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

CORMODE, G. AND GAROFALAKIS, M. N. 2005. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the International Conference on Very Large Databases (VLDB)*.

CORMODE, G. AND MUTHUKRISHNAN, S. 2004. An improved data stream summary: The count-min sketch and its applications. In *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*.

DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. 2004. Designing a DHT for low latency and high throughput. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

DAMGAARD, C. AND WEINER, J. 2000. Describing inequality in plant size or fecundity. *Ecology 81*, 1139–1142.

DOBRA, A., GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. 2004. Sketch-Based multi-query processing over data streams. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

DRUSCHEL, P. AND ROWSTRON, A. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*.

DURAND, M. AND FLAJOLET, P. 2003. Loglog counting of large cardinalities. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*.

FLAJOLET, P. AND MARTIN, G. N. 1985. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31,* 2, 182–209.

FREEDHS. 2006. Homepage. `http://netcins.ceid.upatras.gr/DHS.php`.

FREEPASTRY. 2002. Homepage. `http://freepastry.org/FreePastry/`.

GANGULY, S., GAROFALAKIS, M., AND RASTOGI, R. 2003. Processing set expressions over continuous update streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

GANGULY, S., GIBBONS, P. B., MATIAS, Y., AND SILBERSCHATZ, A. 1996. Bifocal sampling for skew-resistant join size estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

GNUTELLA. 2001. Homepage. `http://gnutella.wego.com/`.

GUMMADI, K., GUMMADI, R., GRIBBLE, S., RATNASAMY, S., SHENKER, S., AND STOICA, I. 2003. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*.

GUPTA, A., AGRAWAL, D., AND EL ABBADI, A. 2003. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the ACM SIGMOD/VLDB Biennial Conference on Innovative Data Systems Research (CIDR)*.

HADJIELEFTHERIOU, M., BYERS, J. W., AND KOLLIOS, G. 2005. Robust sketching and aggregation of distributed data streams. Tech. rep. 2005-011, Computer Science Department, Boston University.

HARREN, M., HELLERSTEIN, J. M., HUEBSCH, R., LOO, B. T., SHENKER, S., AND STOICA, I. 2002. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*.

HARVEY, N., JONES, M., SAROIU, S., THEIMER, M., AND WOLMAN, A. 2003. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*.

HUEBSCH, R., CHUN, B. N., HELLERSTEIN, J. M., LOO, B. T., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., AND YUMEREFENDI, A. R. 2005. The architecture of PIER: An Internet-scale query processor. In *Proceedings of the ACM SIGMOD/VLDB Biennial Conference on Innovative Data Systems Research (CIDR)*.

HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. 2003. Querying the Internet with PIER. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

IVES, Z., KHANDELWAL, N., KAPUR, A., AND CAKIR, M. 2005. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *Proceedings of the ACM SIGMOD/VLDB Biennial Conference on Innovative Data Systems Research (CIDR)*.

JELASITY, M. AND MONTRESOR, A. 2004. Epidemic-Style proactive aggregation in large overlay networks. In *Proceedings of the IEEE International Conference on Distributed Computing and Systems (ICDCS)*.

KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'97)*.

KEMPE, D., DOBRA, A., AND GEHRKE, J. 2003. Computing aggregate information using gossip. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS)*.

KOLONIARI, G. AND PITOURA, E. 2004. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

KRISHNAN, P. 1995. Online prediction algorithms for databases and operating systems. Ph.D. thesis, Brown University.

LIPTON, R. AND NAUGHTON, J. F. 1995. Query size estimation by adaptive sampling. *J. Comput. Syst. Sci. 51*, 1, 18–25.

LIPTON, R. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1990. Practical selectivity estimation through adaptive sampling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

MANKU, G. 2003. Routing networks for distributed hash tables. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*.

MAYMOUKNOV, P. AND MAZIÈRES, D. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*.

MICHEL, S., BENDER, M., NTARMOS, N., TRIANTAFILLOU, P., WEIKUM, G., AND ZIMMER, C. 2006. Discovering and exploiting keyword and attribute-value co-occurrences to improve P2P routing indices. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*.

MONTRESOR, A., MELING, H., AND BABAOĞLU, Ö. 2002. Messor: Load-Balancing through a swarm of autonomous agents. In *Proceedings of the Workshop on Agent and Peer-to-Peer Systems*.

NG, W. S., OOI, B. C., TAN, K. L., AND ZHOU, A. 2003. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

NTARMOS, N. AND TRIANTAFILLOU, P. 2004. AESOP: Altruism-Endowed self-organizing peers. In *Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*.

NTARMOS, N., TRIANTAFILLOU, P., AND WEIKUM, G. 2006. Counting at large: Efficient cardinality estimation in Internet-scale data networks. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

PALMER, C. R., SIGANOS, G., FALOUTSOS, M., FALOUTSOS, C., AND GIBBONS, P. B. 2001. The connectivity and fault-tolerance of the Internet topology. In *Proceedings of the Workshop on Network-Related Data Management (NRDM)*.

PAPADIMOS, V., MAIER, D., AND TUFTE, K. 2003. Distributed query processing and catalogs for peer-to-peer systems. In *Proceedings of the ACM SIGMOD/VLDB Biennial Conference on Innovative Data Systems Research (CIDR)*.

PITOURA, T., NTARMOS, N., AND TRIANTAFILLOU, P. 2006. Replication, load balancing, and efficient range query processing in DHT data networks. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

PITOURA, T. AND TRIANTAFILLOU, P. 2007. Load distribution fairness in p2p data management systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*.

RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. 2004. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*.

SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. 2002. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking Conference (MMCN)*.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*.

TRIANTAFILLOU, P. AND PITOURA, T. 2003. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*.

VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21,* 2, 164–206.

YALAGANDULA, P. AND DAHLIN, M. 2004. A scalable distributed information management system. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*.

YANG, B. AND GARCIA-MOLINA, H. 2001. Comparing hybrid peer-to-peer systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. rep. UCB/CSD-01-1141, University of California at Berkeley, Computer Science Department.

Received February 2008; accepted December 2008