A STUDY OF GOSSIP ALGORITHMS FOR INTERNET-SCALE CARDINALITY ESTIMATION OF DISTRIBUTED XML DATA

A THESIS IN

Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

by

VASIL GEORGIEV SLAVOV

B.A., William Jewell College, 2005

Kansas City, Missouri
2012

A STUDY OF GOSSIP ALGORITHMS FOR INTERNET-SCALE CARDINALITY ESTIMATION OF DISTRIBUTED XML DATA

Vasil Georgiev Slavov, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2012

ABSTRACT

After more than a decade of active research and development, the peer-to-peer (P2P) computing model continues to be successful. We have witnessed the deployment of commercial P2P applications in large, Internet-scale environments. With the rise and growth of P2P, indexing and querying data stored in large-scale sharing systems has become increasingly difficult. Computing statistics over data stored in Internet-scale P2P systems is an important component of query optimization. Decentralized gossip-based protocols are very popular in networking, and in particular, in sensor networks. The simplicity and scalability of gossip protocols render them perfect for quickly computing accurate estimates of aggregates (sums, averages, etc.) in Internet-scale systems where node and link failures are the norm.

In this thesis, we present the problem of cardinality estimation of XPath queries over XML data stored in a distributed, Internet-scale environment. We focus our work on three objectives: implementing gossip in an Internet-scale environment, conducting a comprehensive performance evaluation in a wide-area network, and an-

alyzing the experimental results.

We implement two gossip-based algorithms (VanillaXGossip and XGossip) which, given an XPath query, estimate the number of XML documents in the network that contain a match for the query. XGossip employs a new, divide-and-conquer strategy for load-balancing and reducing the bandwidth consumption. We conduct a comprehensive performance evaluation of both gossip algorithms on Amazon Elastic Compute Cloud (Amazon EC2) web service using a heterogeneous collection of XML documents. The goal of the performance evaluation is to find if the results we obtain are consistent with the theoretical analysis of VanillaXGossip and XGossip.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "A Study of Gossip Algorithms for Internet-Scale Cardinality Estimation of Distributed XML Data," presented by Vasil Georgiev Slavov, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Praveen R. Rao, Ph.D., Committee Chair

Department of Computer Science Electrical Engineering

Yugyung Lee, Ph.D.

Department of Computer Science Electrical Engineering

Deep Medhi, Ph.D.

Department of Computer Science Electrical Engineering

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Dr. Praveen Rao. He has been very supportive of and involved in my thesis work. My experience in working with him on this and other research projects inspired and motivated me to continue my academic career by pursuing a Ph.D. degree in Computer Science. Dr. Rao supported me financially which allowed me to dedicate all of my time to research. I would also like to thank the National Science Foundation for funding our research (IIS-1115871, 2011-2014). Finally, I want to thank my family for their support throughout the years.

CHAPTER 1

INTRODUCTION

The peer-to-peer (P2P) computing model has been very successful in the last decade. Having gained an incredible popularity for file sharing (Kazaa, BitTorrent) and being widely used in communication networks (Skype) for Internet-scale applications, P2P has now evolved into more specialized, structured peer-to-peer systems. Distributed Hash Tables are one example of a widely popular, structured P2P system. DHTs started primarily as research projects (Chord [82], Pastry [73], CAN [71], Tapestry [91], Kademlia [51]), but eventually turned into profitable and heavily used key-value stores like Amazon's Dynamo [22], Apache Cassandra [47], and Voldemort [4].

By the time P2P started gaining traction, XML had already been standardized and was starting to become the data model of choice. Taken together, XML and P2P provided a new way to implement efficient, large-scale data sharing systems. A lot of research has been devoted to indexing and query processing over XML data in P2P networks [45, 30, 9, 21, 68]. The areas of biomedical research and healthcare provide a unique opportunity for testing the P2P and XML technologies on both a large data and network scale. The healthcare community has already recognized the benefits of the P2P architecture in their information-intensive enterprise [81]. Along with that, the standard for representation and interchange of healthcare data (HL7 version 3) is XML-based. HL7v3 is instrumental in allowing semantic interoperability across

different health care systems [53].

An example of a real-world data sharing system is the Cancer Biomedical Informatics Grid (caBIG) [25, 8]. An initiative of the National Cancer Institute to accelerate scientific discoveries, caBIG enables researchers to collaborate across 124 participating institutions in the US. In order to be able to integrate divergent data types, caBIG is implemented using a semantic Service Oriented Architecture (sSOA) [56]. In addition, caGrid, the underlying network infrastructure of caBIG, provides a XML Data Service interface for querying and retrieving data from XML databases [7, 59]. User queries are translated to their corresponding XPath equivalent and are then executed over the XML database. We believe that a P2P architecture may enhance the scalability of caGrid for large-scale sharing of distributed XML data.

We define the task of cardinality estimation as it is used in this thesis as follows.

> *Given an XPath expression (or query) q, estimate the total number of XML documents in the network that contain a match for q with provable guarantee on the quality of the estimate.*

We would like to note that even though the above estimate does not provide the size of the result set of an XPath query, the cardinality estimate of XPath expressions is useful in XQuery optimization. By knowing what the distribution of the relevant documents in the network is, a query optimizer can select appropriate query plans. For example, given the cardinality estimate, the optimizer may determine the most efficient join ordering. Another useful application of the cardinality estimate is if researchers would like to figure out if there is a sufficient number of samples for a

particular study without having to submit a query over distributed data sources.

Ntarmos *et al.* [58] and Pitoura *et al.* [60] discuss the problem of computing statistics over structured data stored in an Internet-scale environment. Neither one, however, uses the XML data model. In order to apply existing techniques for XML selectivity estimation [63, 28, 49], all the XML documents have to be collected from a large number of participating peers. This requirement is not only inefficient, but also very difficult to implement because peers may join, leave, or fail at any time.

1.1 Thesis Objectives

We focus our work on three objectives: implementing gossip in an Internet-scale environment, conducting a comprehensive performance evaluation in a wide-area network, and analyzing the experimental results.

- First, we implement two gossip-based algorithms (VanillaXGossip and XGossip) which, given an XPath query, estimate the number of XML documents in the network that contain a match for the query. XGossip employs a new, divide-and-conquer strategy for load-balancing and reducing the bandwidth consumption.
- Second, we conduct a comprehensive performance evaluation of both gossip algorithms on Amazon Elastic Compute Cloud (Amazon EC2) web service using a heterogeneous collection of XML documents.
- Third, we analyze the experimental results with the goal of finding if the performance evaluation and the analysis of the results we obtain are consistent with the theoretical analysis of VanillaXGossip and XGossip.

The rest of the thesis is organized as follows. In Chapter 2, we provide the background and motivations behind our work. In Chapter 3, we describe the design of the two gossip algorithms (VanillaXGossip and XGossip). In Chapter 4, we describe the implementation of VanillaXGossip and XGossip. In Chapter 5, we present a comprehensive performance evaluation of both VanillaXGossip and XGossip. Finally, we conclude and present ideas for future work in Chapter 6.

CHAPTER 2

BACKGROUND AND MOTIVATIONS

2.1 XML and XPath

The Extensible Markup Language (XML) [88] has become the standard data model for information representation and interchange on the Internet. An XML document can be represented as an ordered, labeled tree (or a linearization of a tree structure).

The XML Path language (XPath) [15] is a query language for XML data. XPath takes advantage of the tree representation of XML documents and selects nodes from documents based on different criteria specified in the query string. XPath queries resemble tree-like patterns called twig patterns. Nodes in these twig patterns represent elements, attributes, and values. Edges represent ancestor-descendent relationships.

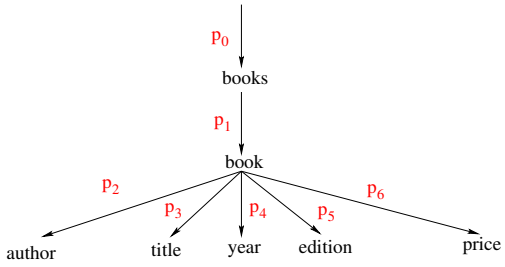
In order to process XML queries efficiently, it is necessary to compute accurate aggregates. While cardinality/selectivity estimation over XML data in local environments (e.g., Path trees/Markov tables [10], Correlated subpath tree [20], pH-join [87], StatiX [29], XPathLearner [48], Bloom Histogram [86], XSKETCH [63], IMAX [64], XCLUSTER [62], XSEED [89], lossy compression based synopsis [28], sampling based technique [49]) and over structured distributed data [**?**] has been well-studied in the past, no prior work is available on computing statistics over XML data in an Internet-scale environment.

```
<books>
  <book>
    <author>John Doe</author>
    <title>The XYZ of XML</title>
    <year>2012</year>
  </book>
  <book>
    <author>Mary Baker</author>
    <author>Robert Doe</author>
    <title>The XYZ of RDF</title>
    <year>2011</year>
    <edition>3</edition>
    <price>$25.00</price>
  </book>
</books>
```

(a) XML document d_1

/books//price

(b) XPath query



(c) Structural Summary Graph (SSG) of d_1

Figure 1. Examples of XML, XPath, and algebraic signatures

2.2 Signatures of XML Documents and XPath Queries

The information retrieval community has studied and applied signature schemes very successfully. Signatures or fingerprints are very attractive because they allow a significant reduction of bandwidth consumption and a small message size (a requirement for gossip-based algorithms [16]). In this thesis, we generate signatures for both documents and queries based on the method proposed by Rao and Moon [68, 67]. Instead of enumerating all possible XPath patterns in each XML document (a computationally expensive task), peers gossip the summaries of XML documents represented as signatures. XML documents are summarized and mapped into algebraic signatures. XPath queries (at query time) are also mapped into algebraic signatures. The signatures are represented as a product of irreducible polynomials. The irreducible polynomials are assigned to the edges connecting ancestor-descendent nodes in an XML document. This design helps preserve a document's structural properties and content. The signature representation satisfies the following necessary condition (divisibility test): *if a document contains a match for a query, then the query signature divides the document signature* [68].

2.3 Gossip Algorithms

There are three types of gossip algorithms [16]. Dissemination protocols use gossip to spread information. This type of gossip protocols either reports events or disseminates background data associated with the nodes [34, 42, 61, 23, 31, 14]. The second type is used for repairing replicated data. The third type of gossip protocols is used to compute aggregates [43, 44, 19, 41, 55]. Most research focuses on information spreading and aggregates computation. Gossip algorithms use an iterative

procedure: each node communicates or exchanges information with a neighbor or a randomly selected peer on every iteration (round). One attractive feature of gossip protocols is their "eventual consistency" [58]. Eventual consistency provides a probabilistic guarantee that despite failures and changes in the P2P environment, the protocol will eventually converge either by spreading the information to all peers or by computing the true value of the aggregates. Another strength of gossip protocols is their simplicity: all nodes run the same code and usually no complex distributed synchronization is needed. Gossip algorithms can avoid the risk of disruptive load surges by producing bounded worst-case loads on participating peers [16]. Finally, due to its decentralized design, gossip exhibits very high-tolerance to temporary network disruptions.

A number of real-world systems use gossip protocols. Gossip is used in Amazon's S3 data centers for information spreading of server states [1]. Information exchange between server nodes is used in Dynamo [22], and Cassandra [47].

2.4 Motivations

No prior is available on XPath cardinality estimation in an Internet-scale environment. Distributed XQuery optimization can benefit from such cardinality estimates. Unfortunately, Distributed Hash Sketches [58] may not be adapted efficiently for XML data. In order to use DHS for XML data, each XPath pattern from the XML document will need to be mapped to one dimensional space. That is, the adaptation will require the enumeration of all possible XPath patterns. This is a computationally expensive task due to the hierarchical nature of XML.

Applying gossip algorithms for XML data introduces new challenges. The

first one is if gossip should start at query time or if it should run in the background continuously. If gossip starts at query time, the cardinality estimate will not be available until after a finite number of rounds. If gossips runs in the background, it will not be possible to gossip every unique XPath pattern because of the large number of distinct patterns. As the number of documents and participating peers grows, scalability may become a challenge. Finally, bandwidth consumption should be minimal. The total amount of data depends on the number of messages transmitted (finite) and their size (small-sized).

CHAPTER 3

THE DESIGN OF VANILLAXGOSSIP AND XGOSSIP

This chapter presents the design of two novel gossip protocols. We draw inspiration from the Push-Sum protocol designed by Kempe *et al.* [44]. The first gossip algorithm for XPath cardinality estimation we present is called VanillaXGossip. Next, we introduce an improved version called XGossip. XGossip is a novel, divide-and-conquer algorithm which does not have the limitations of VanillaXGossip. Table 1 shows the commonly used notation in this thesis.

3.1 Design Requirements

Based on the observations outlined in the Introduction chapter, the key design requirements for an effective cardinality estimation algorithm are the following:

1. Scalability: the algorithm should be scalable to a large number of peers.
2. Decentralized architecture: the algorithm should have no single point of failure.
3. Bandwidth consumption: the algorithm should consume a minimum amount of bandwidth.
4. Fault tolerance: the proposed solution should be robust to failures and be able to function in dynamic networks where node and link failures are the norm.
5. Accuracy: the algorithm should provide provable guarantees on the accuracy of the cardinality estimates.

Gossip-based protocols and algorithms satisfy all the requirements listed above for effective cardinality estimation in large-distributed systems. The simplicity and

scalability of gossip protocols render them perfect for quickly computing accurate estimates of aggregates (sums, averages, etc.). Gossip algorithms are completely decentralized: nodes communicate with their neighbors or with randomly selected peers in a DHT network in an iterative basis. The bandwidth requirements of gossip protocols are very low. Finally, gossip algorithms are very well suited for Internet-scale systems where node and link failures are very common and fault-tolerance is important. However, the nature of the XML data model makes applying gossip algorithms for cardinality estimation somewhat challenging. We aim to show that it is possible to design efficient gossip-based algorithms for computing statistics.

3.2 Push-Sum Protocol

Push-Sum is a gossip-based protocol for computing aggregates. In a network of n peers, each peer p_i has a non-negative value x_i. Let us estimate the average of all the values. According to the Push-Sum protocol [44], in each round t, a peer maintains a sum s_t and a weight w_t. In round 0, each peer p_i sends the tuple $(x_i, 1)$ to itself. In every round $t > 0$, a peer chooses a target uniformly at random and sends half of its sum and weight to the target peer and half to itself. The peer also computes the new sum and weight for the round by adding up all the sums and weights of the received messages. At any round t, the estimate of the average is the ratio of the sum and the weight (s_t/w_t). Push-Sum employs uniform gossip because a peer can contact any other peer in the network and all the peers form a complete graph.

THEOREM 3.1 (Push-Sum Protocol [44]). Suppose there are n peers $p_1, \ldots, p_n$ in a network. Each peer p_i has a value $x_i \geq 0$. With at least probability $1 - \delta$, there is

Table 1. Commonly used notations

Notation	Description
n	number of peers in the network
s, s_i	signature of an XML document
f_s	frequency of a signature s
D	number of distinct document signatures in the network
t	a particular round during gossip
f, f_i	sum maintained by a peer during gossip
w, w_i	weight maintained by a peer during gossip
$\perp$	special multiset used by peers in VanillaXGossip
T	tuple list maintained by a peer during gossip
Δ	number of peers in a team or team size
$\perp_h$	special multiset used by peers of a team in XGossip
δ	confidence parameter
ϵ	accuracy parameter
k, l	tuning parameters for locality sensitive hashing (LSH)
$\overline{h_s}$	vector of values produced by LSH on signature s
α	probability that there is at least one team (of peers) that gossips two given signatures after applying LSH
R	set of distinct document signatures that are divisible by a query signature
r	$\|R\|$
q_{min}	minimum similarity between a query signature and a signature in R
p_{min}	minimum similarity between a proxy signature and a signature in R

a round $t_o = O(log(n) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta}))$, such that in all rounds $t \geq t_o$, at peer p_i, the relative error of the estimate of the average value $\frac{1}{n}\sum_{i=1}^{n} x_i$ is at most ϵ.

For the proof of Push-Sum, Kempe *et al.* rely on an important property which they call *mass conservation*: at any round t, the average of all the sums s_t is the true average and the sum of all the weights w_t is always n [44]. In order to compute

the sum, instead of the average, at $t = 0$, one peer p_1 starts with $w_0 = 1$ and all the other peers start with weights equal to 0 [44].

3.3 VanillaXGossip

The Push-Sum protocol is the basis for both novel gossip algorithms we propose VanillaXGossip and XGossip. Two main reasons make Push-Sum a good model for a gossip algorithm in DHT-based structured overlay networks. First, because Push-Sum uses uniform gossip, it assumes that peers are connected in a complete graph. That is, just like in a DHT, a peer can contact any other peer in $O(log(n))$ hops. Second, even though Push-Sum is a synchronous gossip protocol, peers need not follow the same clock. As long as mass conservation is preserved [44], convergence will hold. The main difference between VanillaXGossip and XGossip is that instead of "sum," we compute "average." The only way to compute the sum is for one peer to start with weight 1 and all the others with weight 0. This would require a sophisticated distributed synchronization to take place.

As discussed in Chapter 2, peers gossip signatures of XML documents rather than XPath patterns. We denote the frequency of a signature s by f_s. VanillaXGossip runs in two consecutive phases. Algorithm 1 in Appendix A shows the pseudo code for the first one, the initialization phase. In this phase, each peer creates a list of signatures and their frequencies. Let T denote the tuple list of a peer. Tuples consist of a signature, its frequency, and its weight: $(s, (f_s, 1))$. Note that it is possible for the same signature to be generated for two different documents [68]. All the tuples in T are sorted on the signature s. The last tuple in the list is $(\perp, (0, 1))$. The symbol $\perp$ denotes the special multiset. The special multiset plays the role of a placeholder

for signatures which are unknown to the peer. In order to preserve the important property of mass conservation, if a peer receives a gossip message with an unknown signature, the peer will add the frequency and the weight of that signature to the special multiset in its tuple list T.

The execution phase follows the initialization phase. Algorithm 2 in Appendix A shows the pseudo code for this phase. The execution phase starts with the *RunGossip*() procedure. Because a peer may receive multiple lists in the same gossip round, after the round ends, the peer merges all the lists (including the one it sent to itself). The merging consists in adding up the frequencies f_s and weights w_s of each tuple. The peer divides the merged frequencies and weights in half and sends the list to a randomly selected peer and to itself. We generate a random Chord identifier and send the list to the successor of that id.

The $MergeLists()$ procedure in Algorithm 2 describes the the merging process of VanillaXGossip. Merging completes in linear time because the lists are sorted by the signature in each tuple. We find the tuple with the minimum multiset and compute the sum and weight across all lists. For lists which do not contain the minimum multiset, we use the sum and weight of the special multiset $\perp$. The estimate of the average of the frequency of s in any round is $\frac{f_s}{w_s}$.

THEOREM 3.2 (VanillaXGossip). Given n peers $p_1, \ldots, p_n$, let a signature s be published by some m peers with frequencies $f_1, \ldots, f_m$, where $m \leq n$. With at least probability $1 - \delta$, there is a round $t_o = O(log(n) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta}))$, such that in all rounds $t \geq t_o$, at peer p_i, the relative error of the estimate of the average frequency of s, i.e., $\frac{1}{n}\sum_{i=1}^{m} f_i$, is at most ϵ.

The proof of Theorem 3.2 is in the technical report [69].

There is an important difference between Push-Sum and VanillaXGossip. If we used Push-Sum for cardinality estimation of XPath queries, we would have to start running Push-Sum at query time, when the XPath query was submitted. The peer which received the query would then compute the cardinality estimate on its local documents and would start gossiping the estimate to other peers. A number of rounds later, the average of the estimate would be available. In contrast to Push-Sum, VanillaXGossip runs in the background continuously. Peers gossip signature frequencies without any knowledge of future queries. The special multiset in VanillaXGossip guarantees convergence like Push-Sum. At query time, a peer only needs to compute the cardinality estimate on its local state. This approach minimizes query time because the query initiator need not wait for a finite number of rounds to receive the cardinality estimate.

3.4 XGossip: A Divide-and-Conquer Approach

The problem with VanillaXGossip is that when a peer learns about all the signatures in the network, the tuple list T will get very large. Because peers have a limited amount of memory to store the lists, VanillaXGossip is not scalable. In addition, the size of the gossip messages grows large because peers send the whole tuple list T in one message. As noted earlier, to be efficient, gossip algorithms require small-sized messages. Next, we present XGossip, a novel divide-and-conquer algorithm which overcomes the limitations of VanillaXGossip. In XGossip, peers gossip a provably finite faction of signatures. There are three main improvements which distinguish XGossip from VanillaXGossip: peers consume less memory, peers

consume less bandwidth, and convergence requires a fewer number of rounds.

XGossip relies on Locality Sensitive Hashing (LSH) for signature distribution and load-balancing. The main idea of LSH is that we can map similar items to the same buckets with high probability by hashing the input items. Since Indyk and Motwani [40] introduced LSH, the concept has been widely used in many domains: indexing high dimensional data and similarity searching [12, 50], similarity searching over web data [38] and P2P networks [38, 36], ranges queries in P2P networks [35], and so on. LSH is also useful when working with similarity of sets based on the Jaccard index [38, 12]. Because the signatures of XML documents we use for gossiping are, in essence, sets of irreducible polynomials, it is possible to use LSH in gossip. There are two parameters which define how LSH behaves: l, which denotes the number of groups of hash functions and k, which denotes the number of hash functions per group. We pick a total of $k \times l$ random linear hash functions. Each function is of the form $h(x) = (ax + b) \mod p$, where p is a prime, and a and b are integers such that $0 < a < p$ and $0 \leq b < p$. The output hash value for a signature s (a set of irreducible polynomials) is $g(s) = min(\{h(x)\})$. Indyk and Motwani establish [40] that the probability that the output hash values of two different sets is equal, is equivalent to the Jaccard index of the sets: $Prob(g(s_1) = g(s_2)) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$. Because each group l of hash values can be hashed again, the output for each set is k hash values. We use SHA-1 for the LSH hash function. This allows us to map the 160-bit output values onto the Chord DHT ring as an identifier. Applying LSH on a signature s produces a vector of hash values which we call $\overline{h_s}$. This vector defines k different teams for one signature s. We use the hash values to denote the

ids of each team. Let Δ denote the size of a team. We can calculate the Chord identifier for each team (the team id) h_{si} by performing the following operations: $\{h_{si}, h_{si} + 1 \times \frac{2^{160}}{\Delta}, h_{si} + 2 \times \frac{2^{160}}{\Delta}, \ldots, h_{si} + (\Delta - 1) \times \frac{2^{160}}{\Delta}\}$. In other words, we split the Chord ring in Δ similar arcs by starting at the team Chord id and assigning successively the rest of the IDs to the other members of the team. The result will wrap around the ring.

The members of the team are the successors of the Chord ids which we generated above. These members only communicate with peers of the teams they belong to. They gossip only signatures whose LSH output was mapped to their ids. The goal of applying LSH is for the same team to gossip similar signatures with high probability. If two signatures have a similarity p, there is at least one team which gossips both signatures with probability $1 - (1 - p^l)^k$. The chances of finding all the signatures required for estimating the cardinality of an XPath query are higher if the same team gossips all the signatures.

Unlike VanillaXGossip, using a single special multiset ($\perp$) is not possible in XGossip. In order for mass conservation to occur, in XGossip, peers need to maintain a different special multiset for each team they belong to. We denote the special multiset for a team h by $\perp_h$.

Like VanillaXGossip, XGossip is comprised of two phases. Algorithm 3 in Appendix A shows the initialization phase. The goal of this phase is the create the teams which gossip a subset of signatures. The $InitGossipSend()$ procedure consists first creating a sorted tuple list T of all the signatures and their frequencies (like in VanillaXGossip). Then, the peer creates k teams by applying LSH on each tuple's

signature. The peer sends the tuple to the successor of that id by picking a random Chord id from the LSH output. The $InitGossipReceive()$ procedure in Algorithm 3 (Appendix A) handles the received messages based on the type of multiset in the message. If the peer receives a regular multiset which it knows about (it can find in the T of the team responsible for gossiping it), the procedure updates the frequency in the tuple list. We handle the possibility for a peer to not receive a message during initialization by using a procedure called $InformTeam()$. In effect, peers inform the next members of the team by sending them the special multiset of the team. This lets each member of the team to initialize its special multiset.

The execution phase of XGossip consists of peers gossiping the signatures distributed during the initialization phase to the other members of the team by randomly picking a receiver. In a finite number of rounds, all the members of the team will know about all the signatures their team is responsible for.

THEOREM 3.3 (XGossip). Given n peers $p_1, \ldots, p_n$ in a network, let a signature s be published by some m peers with frequencies $f_1, \ldots, f_m$, where $m \leq n$. Suppose p_i belongs to a team that gossips s after applying LSH on s. Let Δ denote the team size. With at least probability $1-\delta$, there is a round $t_o = O(log(\Delta)+log(\frac{1}{\epsilon})+log(\frac{1}{\delta}))$, such that in all rounds $t \geq t_o$, at peer p_i, the relative error of the estimate of the average frequency of s, i.e., $\frac{1}{\Delta}\sum_{i=1}^{m} f_i$, is at most ϵ.

The proof of Theorem 3.2 is beyond the scope of this thesis and can be found in the technical report [69].

Table 2 compares VanillaXGossip and XGossip based on different metrics and

Table 2. Comparison of VanillaXGossip and XGossip

Metric	**VanillaXGossip**	**XGossip**
Accuracy	$r\epsilon$	$r\epsilon$
Confidence	$(1-\delta)$	$(1-\delta)$
Convergence (# of rounds)	$O(log(n)+log(\frac{1}{\epsilon})$ $+log(\frac{1}{\delta}))$	$O(log(\Delta)+log(\frac{1}{\epsilon})$ $+log(\frac{\alpha}{\alpha+\delta-1}))$
Bandwidth	$O(nD)$	$O(log(n)kD\Delta)$
Messages	$O(n\ log(n))$	$O(\frac{log(n)}{n}kD\Delta log(\Delta))$

using the notation in Table 1. Detailed explanation of this comparison is in the technical report [69].

3.5 Cardinality Estimation of XPath Queries

3.5.1 VanillaXGossip

The cardinality estimation of XPath queries in VanillaXGossip is trivial. A peer needs only to check its local state to produce an estimate. For example, if a peer p receives a query q, it searches its merged list T_m to find all tuples $(s,(f_s,w))$ whose signatures are supersets of q's signature. According to Rao [68], the divisibility test (defined in Chapter 2), is equivalent to testing the superset relationship between a document and a query signature. To calculate the final cardinality estimate for all matched tuples, we compute $\sum \frac{f_s}{w}$ and multiply the sum by the total number of peers n. We can obtain a good estimate of n by using Push-Sum. (We have to multiply the "sum" (of all tuple frequencies) by n because the frequency f_s stored in each tuple is the average, not the sum).

3.5.2 XGossip

In XGossip, however, a few peers need to be contacted for a successful cardinality estimate. If a peer p receives a query q, $\overline{h_q}$ would denote the output of LSH. We pick a random team member for each team h_{qi} $(1 \leq i \leq k)$ to send q's signature and h_{qi} to. The team member which receives the query checks its local state for a tuple list for team h_{qi}. It returns all tuples $(s, (f_s, w))$ for which s is a superset of q's signature. For tuples with identical signature returned by different peers, we retain one at random and discard the others. We compute $\sum \frac{f_s}{w}$ for all the tuples received from the k peers. The main difference with VanillaXGossip is that instead of multiplying by n, we multiply by Δ to produce the cardinality estimate of q. Because we contact k peers, the number of hops required is $O(k\ log(n))$.

The probability that at least one team gossips two different signatures with similarity p is $1 - (1 - p^l)^k$. If the similarity between a query signature and a matching document is low, the probability will be low too and we may completely miss some document signatures. This will lead to a poor quality cardinality estimate. We propose the idea of a *proxy signature*. A proxy signature contains the maximum number of distinct elements and attributes of a DTD or an XML Schema to which a document conforms to. Instead of applying LSH on a query signature, we apply it on a proxy signature, but still send the query signature to obtain the cardinality estimate. The proxy signature is more likely to find the teams which gossip the documents matching the query signature.

CHAPTER 4

IMPLEMENTATION OF VANILLAXGOSSIP AND XGOSSIP

We implemented VanillaXGossip and XGossip on top of a peer-to-peer Distributed Hash Table. DHTs store key-value pairs by assigning them to different peers in the network based on the key for which the node is responsible. Among others, DHTs support operations for storing values, assigning keys to nodes, and finding values associated with given keys. We use the Chord DHT [83] as on overlay network. In Chord, peers are mapped to a 160-bit identifier space which forms a ring. To retrieve a key, Chord supports a *lookup* operation. To store key-value pairs, the *insert* operation is available. Chord utilizes consistent hashing for assigning nodes and keys an m-bit identifier called a chordID. The function used for base hashing is SHA-1 [27]. Identifiers for nodes and keys are generated by using a peer's IP address or a key's value respectively. A node is a successor of a key if that key's identifier is equal to or precedes the identifier of the node.

Because we build our gossip algorithm on top of the Chord DHT, we can take advantage of many of the attractive features it provides. For example, all the nodes in Chord form a complete graph. In a network with N nodes, a peer is able to contact any other peer (not just its neighbors) in $O(log(n))$ hops. That is, the DHT provides a very efficient routing mechanism for gossip messages. Peers participating in uniform gossip (VanillaXGossip and XGossip) require connectivity to random nodes in each iteration (round) of gossiping. The gossip algorithm only need to perform

an insert operation on a key and that message will get routed to the successor of that key quickly and efficiently. In order for VanillaXGossip and XGossip to not be susceptible to the dynamism of the network on which are they built, the overlay itself has to be fault-tolerant. Chord provides a highly fault tolerant environment which is ideal for Internet-scale distributed data sharing. Even in the presence of churn, the DHT is able to stabilize (reassign identifiers for keys and nodes which have changed due to nodes joining and leaving the network).

Each peer in the Chord DHT owns a set of XML documents. The peer publishes these documents in order to share them with other peers in the network. Because locating publisher's of a particular document based on the content of the document is the goal of cardinality estimation, peers gossip information about the documents. Per the gossip protocols described in Chapter 3, peers gossip the frequency and weight of the signatures which correspond to the XML documents the peers publish.

4.1 System Architecture

We implemented VanillaXGossip and XGossip in C++. We relied heavily on the Standard Template Library (STL) in C++, and in particular, on the containers it provides. We also used an asynchronous library called libasync. Libasync allowed us to respond to events through callbacks. The Chord DHT uses libasync (in the form of sfslite) [76] to associate function callbacks with readability and writability conditions on sockets.

Each peer involved in gossip runs separate Chord and gossip processes. Figures 2 and 3 show in details the system architecture of VanillaXGossip and XGossip.

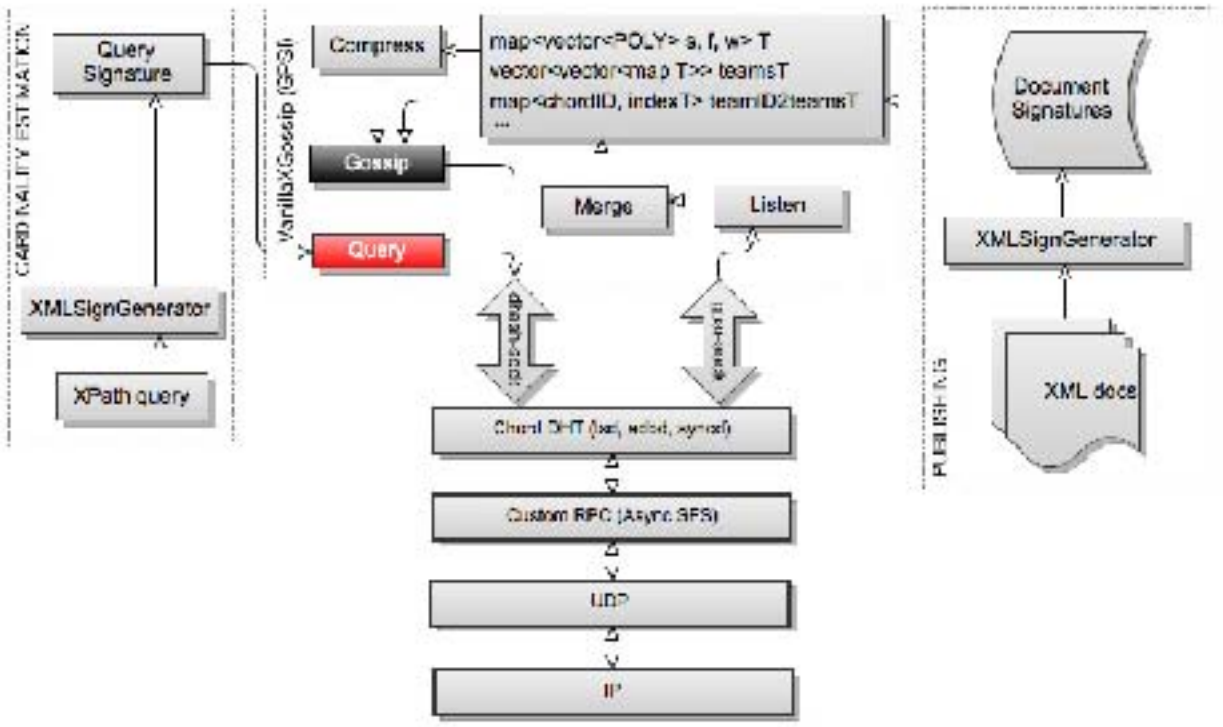


Figure 2. System Architecture of VanillaXGossip

The main process is lsd. It communicates over UNIX sockets with adbd (asynchronous database), and syncd. Only one gossip process runs which is called gpsi. The gossip algorithm (gpsi) and Chord communicate using a UNIX socket. Even though gpsi is a single-threaded program, because of libasync, gpsi is able to both listen for gossip messages and insert key-value pairs in Chord at the same time. Before the initialization phase begins in both VanillaXGossip and XGossip, the gpsi process reads the signatures of XML documents stored locally. The distribution of documents is described in Chapter 5. To minimize I/O, gpsi does not write anything to disk, all

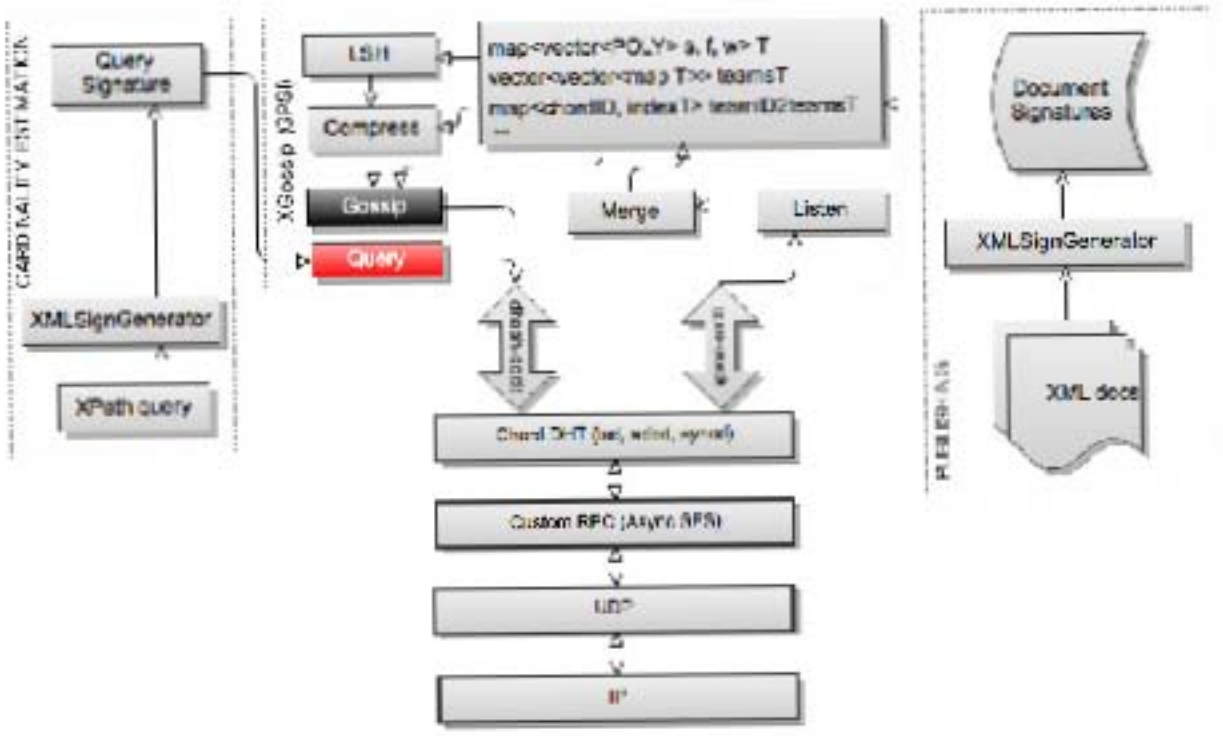


Figure 3. System Architecture of XGossip

signatures and data structures are stored in memory throughout gossip. Only a log file is written to disk. We changed the amount of data logged depending on the need (minimal logging when not debugging). The main containers we used from STL were maps, multi maps, and vectors. VanillaXGossip required a significantly simpler set of data structures. We only had to keep track of the signatures and their frequencies and weights. We stored them in a vector of maps. Because peers could receive multiple messages in a particular round, we had to store each set of lists received in the same round in a separate map. We merged all lists before a new round began.

XGossip required a much more complex set of containers. In addition to keeping track of the signatures, we had to keep track of the teams to which a peer belonged and associate the signatures with the corresponding team. To store this information, we used a vector of a vector of maps. We also maintained a team index with pointers to the right lists, and multiple other maps for associating chordIDs, teamIDs, signatures, DTDs, etc. Because at any time, any peer may perform the task of cardinality estimation on an XPath query, we built it algorithms and data structures for query processing. The heavy use of data structures did not affect adversely the performance of VanillaXGossip and XGossip because the containers we used are very efficient. Vectors allow contestant random access. The asymptotic complexity of all map and multimap operations in STL is $O(log(n))$.

4.2 Challenges

In VanillaXGossip, after a peer learns about all the distinct signatures in the network, the peer starts gossiping messages which contain all those signatures. The size of the message, even when the message is compressed, can increase substantially. Gossip messages contain additional information as well (frequencies, weights, team identifier, etc.). While running VanillaXGossip with the larger dataset D_2 described in Chapter 5, we ran into a limitation of the Chord DHT which we could not overcome. When the message size surpassed 1MB in size, Chord refused to route the message. We were not able to find where the maximum message size was defined. While we initially thought that the limitation was enforced by the operating system in the form of a kernel/system variable, running gossip on different machines and/or Linux distributions did not change the behavior described above. Because the interprocess

communication is performed by sfslite (libasync), most likely, the message size is defined there. We did not investigate this problem further because exchanging large-size messages using gossip is not recommended [16].

Defining and working with multiple complex data structures posed difficulties. Even though we made an effort to anticipate future requirements of the software, we did reorganize some of the main containers halfway through the project. Because only one type of data type may be associated with a key in STL maps, we had to use creative ways to map different data types and to use vectors and indices to other containers.

Working with libasync proved difficult because of the lack of detailed documentation and support. While libasync is very powerful, it's API is not very intuitive and programming with callbacks in general takes time to get used to. We had to read a lot of the Chord source code in order to understand how to correctly use the library.

Running gossip algorithms in a distributed environment requires a great deal of scripting. While cloud services like Amazon EC2 are orders of magnitude more reliable than research-run networks like PlanetLab (which we used in the NetDB paper [80]), automating the deployment of code, datasets, and queries was a challenge.

We analyzed large amounts of data in order to evaluate the gossip algorithms in this study. This analysis required that we perform some tasks on the peers themselves, after gossip was done. Because the log files grew very large, we made some of the computations in a distributed manner and collected the results from each peer.

Finally, debugging code which runs in a distributed environment required a

lot of time because certain problems arose only when the code was scaled to more peers or when the data was scaled. Replicating the peer-to-peer network in a local environment was not always possible and forced us to rely more on verifying code by re-reading it rather than by working with a debugger.

CHAPTER 5

EVALUATION

We conducted a comprehensive evaluation of both VanillaXGossip and XGossip and report the results in this section. We ran the evaluation on the Amazon Elastic Compute Cloud (EC2) [11] using 20 instances. (By default, EC2 allows at most 20 instances per user.) Each instance was a medium instance (2 virtual cores, 1.7GB memory). We ran all instances in the same region (US East - N. Virginia), but we did not favor any particular availability zone.

5.1 Datasets and Queries

We used two different datasets in the evaluation of VanillaXGossip and XGossip. We generated the datasets using the IBM synthetic XML data generator and DTDs published on the Internet [84, 85, 88]. Dataset D_1 contained documents from 11 DTDs, the average number of documents per DTD was 190,809 and the total number of documents was 2,098,900 (Table 3). Dataset D_2 contained documents from 13 DTDs with an average number of 192,223 documents per DTD and a total of 2,498,900 documents. The average size of a document signature was 114 and 127 bytes respectively. The reason for using two different datasets was a limitation of the underlying DHT: Chord did not allow us to transmit the large messages which VanillaXGossip generated during the final rounds. First, we used dataset D_1 to compare VanillaXGossip and XGossip. Then, we used the bigger dataset D_2 to evaulate XGossip by modifying the parameters specified in the Evaluation Metrics section.

Table 3. Datasets

Dataset	**# of DTDs**	**Avg. # of documents per DTD**	**Total # of documents**	**Avg. document signature size**
D_1	11	190,809	2,098,900	114 bytes
D_2	13	192,223	2,498,900	127 bytes

Table 4 shows the network setup and distribution of documents. We ran an equal number of peers on each of the 20 Amazon EC2 instances: 25/instance for 500 peers, 50/instance for 1000 peers, and 100/instance for 2000 peers (Figure 4). We distributed the initial set of documents across the peers running on each instance. This distribution resulted in the frequency of each document to be scaled by the number of instances (20). In order for each peer to start gossiping with approximately equal number of documents, we distributed the documents from each DTD uniformly to a fixed number of peers depending on the total number of peers running. For example, when we ran gossip on a total of 1000 peers, we randomly picked 25 peers for each DTD and distributed the documents for that DTD equally across those peers. When using 500 and 2000 peers, we distributed the documents across 13 and 50 peers respectively. The average number of DTDs per host was 6.

We generated XPath queries for each DTD by using the XPath generator from the YFilter project [24]. The queries contained wildcards '*' and '//' axis. From this generated query set, we selected a subset with a $q_{min} \geq 0.3$. This selection resulted in a total of 753 queries. We further split the queries in 6 different subsets based on their p_{min} value as shown in Table 5.

During both VanillaXGossip and XGossip, each peer followed its local clock. The length of each successive gossip round in all experiments was 120 seconds.

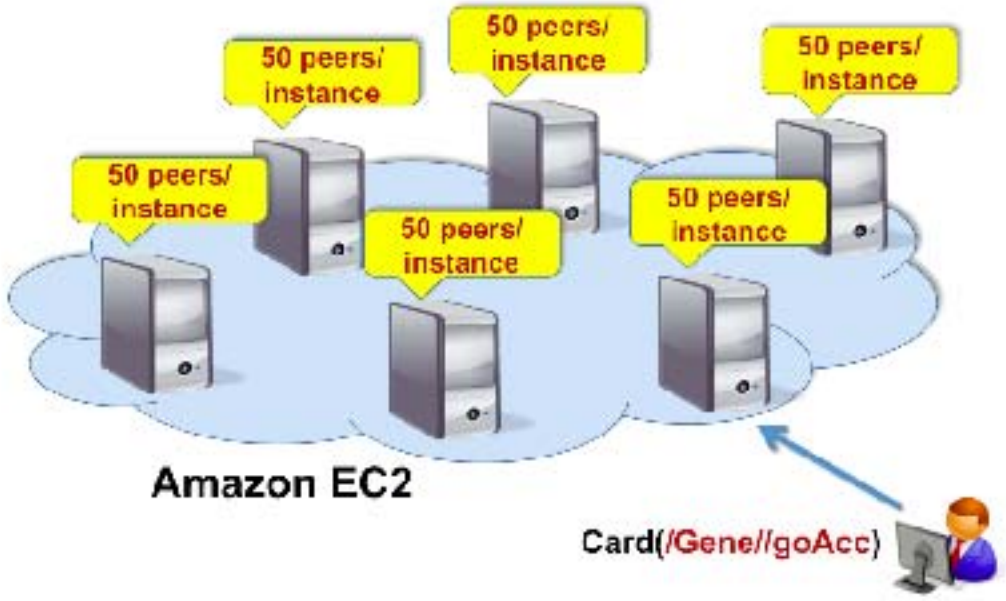


Figure 4. Amazon EC2

Table 4. Network setup and distribution of documents

Total # of peers in the network (n)	**# of peers per EC2 instance**	**# of peers picked per DTD** (z)	**# of documents published by a peer**	
			D_1 (μ, σ)	D_2 (μ, σ)
500	25	250	n/a	4997.8, 257.7
1000	50	500	2098.9, 99.1	2498.9, 99.1
2000	100	1000	n/a	1249.5, 49

5.2 Evaluation Metrics

To compare VanillaXGossip and XGossip, we measured (a) the accuracy of cardinality estimation, (b) the signature convergence speed, and (c) the bandwidth

Table 5. Query Sets

Query Set	**Value of p_{min}**	**# of queries**
Q_0	$[0, 0.5)$	101
Q_1	$[0.5, 1]$	652
Q_2	$[0.6, 1]$	356
Q_3	$[0.7, 1]$	300
Q_4	$[0.8, 1]$	277
Q_5	$[0.9, 1]$	26

consumption.

To calculate the accuracy of cardinality estimation, we computed the mean absolute relative error of the returned query results compared to the true results for each query. For the signature convergence speed, we calculated the mean absolute relative error of the gossiped frequency estimates compared to the true frequency of each signature. For VanillaXGossip only, we measured the diffusion speed of signatures for each peer (how long it takes for each peer to learn about all the unique signatures in the network). Measuring the diffusion speed of signatures for XGossip is not meaningful in comparison to VanillaXGossip because each peer may belong to multiple teams and be responsible for a different number of signatures.

To achieve high accuracy of cardinality estimation and to reduce the bandwidth consumption, we ran XGossip with different values of the LSH parameter k, and different team sizes (Δ). In addition to showing the accuracy of cardinality estimation for all queries, we measured the accuracy for different query subsets based on their p_{min} value (Table 5).

We measured the effectiveness of locality sensitive hashing in reducing the bandwidth by calculating the amount of data transmitted. We measured the signature

Table 6. Time taken to contact k peers during cardinality estimation

LSH parameter k	**Team size** (Δ)	**Average time to contact** k **peers (ms)**
4	8	28.04
4	16	28.31
8	8	53.33
8	16	56.36

distribution of XGossip per team and per peer.

We increased the total number of peers to evaluate the effects of scaling XGossip. For each peer, we measured the average number of teams, signatures, and message size. In addition, we calculated the total number of messages across all rounds. We measured the amount of transmitted data for each set of peers. To evaluate the benefits of compression, we measured the bandwidth consumption with and without compression.

5.3 Comparison of VanillaXGossip and XGossip on Dataset D_1

Figure 5 shows the diffusion speed of signatures in VanillaXGossip. We observe that by round 11, all 3 peers have learned about all the signatures in the network. Note that the fraction of unique signatures is different from the true frequency of a signature. Since different (but similar) documents may generate the same signature, signature frequencies are usually greater than 1. It is possible for a peer to know about all the unique signatures (the fraction of unique signatures to be equal to 1), but for the peer to still have high mean absolute relative error of the frequency estimates. This scenario may occur in the beginning of gossip when the frequency estimates of

individual signatures are still inaccurate.

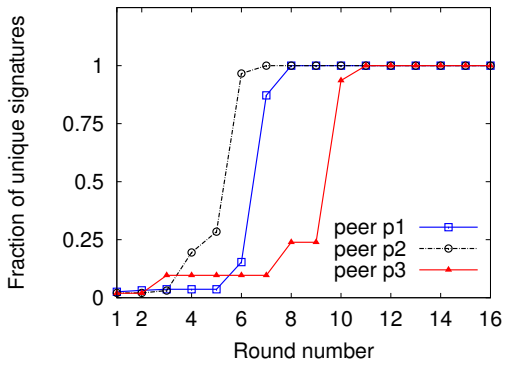
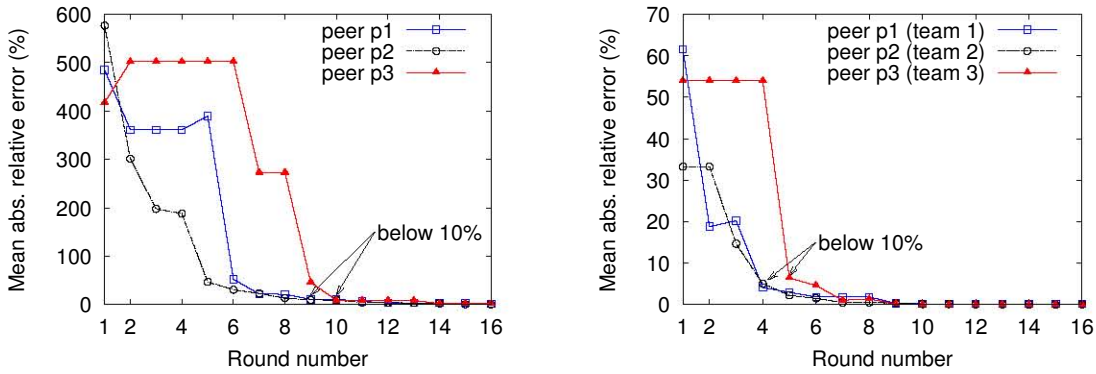


Figure 5. Diffusion speed of signatures in VanillaXGossip, $n = 1000$

Figure 6(a) shows the convergence speed of VanillaXGossip for three randomly selected peers $p1$, $p2$, and $p3$. Beyond round 10, the mean absolute relative error of the average frequency estimate on a subset of signatures remains below 10%. We observe that even though the convergence speed for each peer varies (the error for peers $p1$ and $p2$ drops sharply as early as round 6 while the error for peer $p3$ stays high until round 9), eventually, all peers have high accuracy of their signature frequency estimates.

Figure 6(b) shows the convergence speed of XGossip for three randomly selected peers which belong to three different teams. We observe that beyond round 5, the mean absolute relative error for all three peers remains below 10%. If one runs an XPath query at round 5, XGossip will return an accurate cardinality estimate, while VanillaXGossip will not. After looking at the convergence speed of all the peers, we



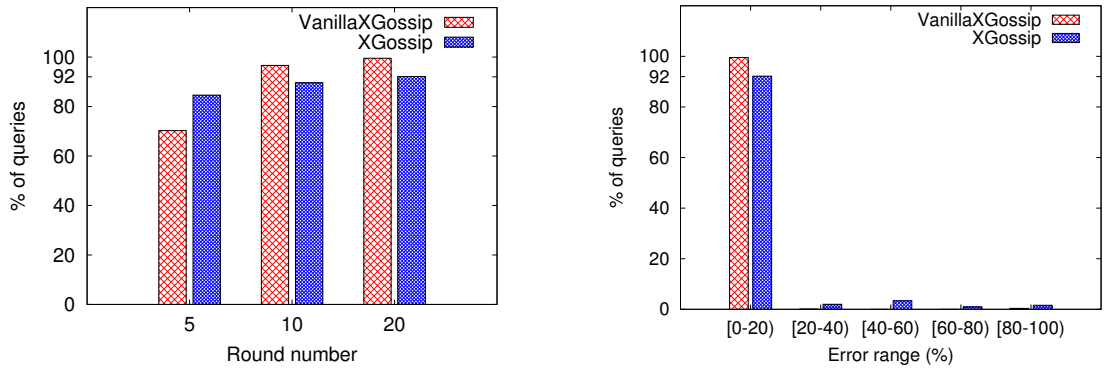
(a) VanillaXGossip (b) XGossip, $\Delta = 8$, $k = 8$, $l = 10$

Figure 6. Comparison of the convergence speed of VanillaXGossip and XGossip, $n = 1000$

observered that, on average, it takes half as many rounds for XGossip to converge than VanillaXGossip. Given this data, we can conclude that XGossip converges faster than VanillaXGossip.

To compare how the accuracy of cardinality estimation changes at different rounds, we ran VanillaXGossip and XGossip for an increasing number of rounds (5, 10, and 20). After all the peers had finished gossiping for the set number of rounds, we ran all 753 XPath queries ($0 \leq p_{min} < 1$). Figure 7(a) compares the accuracy of cardinality estimation for VanillaXGossip and XGossip. The figure shows the percentage of queries with mean absolute relative error below 20% ($\mathrm{r}\epsilon \leq 0.2$) for rounds 5, 10, and 20. We observe that at round 5, the accuracy of XGossip is much higher than the accuracy of VanillaXGossip (84.6% vs. 70.2%). By round 10, VanillaXGossip has surpassed XGossip and by round 20, both have reached the highest accuracy they can achieve: 99.5% for XGossip and 92.2% for XGossip. We ran

both methods for more than 20 rounds, but could not achieve higher accuracy of the cardinality estimation. Figure 7(b) shows the distribution of queries with accuracy above 20% ($r\epsilon \geq 0.2$) after 20 rounds of gossiping.



(a) Accuracy at different rounds, $r\epsilon \leq 0.2$ (b) Accuracy after 20 rounds

Figure 7. Accuracy of cardinality estimation by VanillaXGossip and XGossip, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

Finally, we discuss the bandwidth consumption of VanillaXGossip and XGossip. Figure 8 shows the amount of data transmitted for each of the 20 rounds. Even though we ran both VanillaXGossip and XGossip with compression enabled, by round 20, VanillaXGossip had reached 731 MB per round, while XGossip was using only 25 MB per round (almost 30 times less). The total amount of data transmitted by VanillaXGossip for the duration of 20 rounds was 10,309 MB. XGossip consumed 484 MB across all rounds.

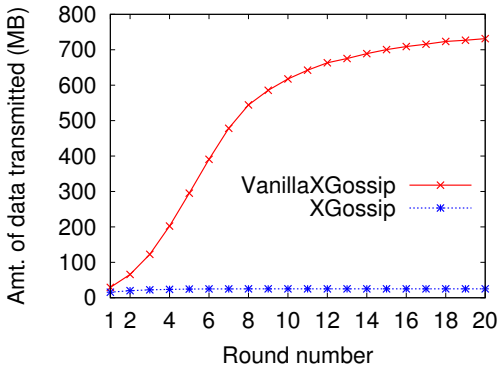


Figure 8. Bandwidth consumption of VanillaXGossip and XGossip, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

# of peers	Avg. # of teams/peer	Avg. # of signatures/peer	Avg. # of signatures/team
500	88.40	4024.25	45.52
1000	44.83	2040.81	45.52
2000	23.09	1051.2	45.52

Table 7. Teams and signatures

# of peers	Avg. message size/peer (bytes)	Total # of messages
500	1,160.18	440,240
1000	1,265.64	440,240
2000	1,244.91	441,750

Table 8. Message complexity

5.4 Evaluation of XGossip on Dataset D_2

The convergence speed of XGossip on the bigger dataset (D_2) showed similar results as on dataset D_1. Figure 9 shows that after round 5, the mean absolute

relative error for all 3 peers remains below 10%.

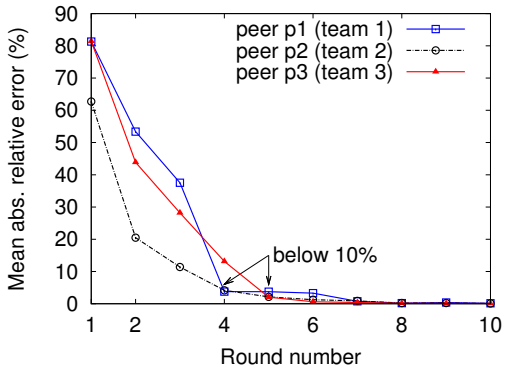


Figure 9. Convergence speed of XGossip, $n = 1000$, $\Delta = 16$, $k = 4$, $l = 10$

In order to achieve high accuracy, we ran XGossip for different values of the LSH parameter k, and the team size Δ (Figure 10). We let XGossip run for 30 rounds and then ran all the 753 XPath queries with $0 \leq p_{min} < 1$. First, we set k to 4 and l to 10 and doubled the team size from 8 to 16. After running XGossip for 20 rounds, the accuracy for both values of Δ was the same: 70.8% of the queries had a relative error below 20%. However, the amount of data transmitted per round had doubled from 30.9 MB to 61.8 MB (Figure 11). Then, we doubled k from 4 to 8 and ran XGossip with Δ set to 8 and 16. Compared to $k = 4$, the accuracy increased significantly (from 70.8% to 92.3%), but it did not change when the team size increased. Again, we observed an increase in the bandwidth consumption for higher values of either k or Δ. We saw the highest bandwidth usage per round (123.9 MB) for $k = 8$ and $\Delta = 16$ (Figure 11).

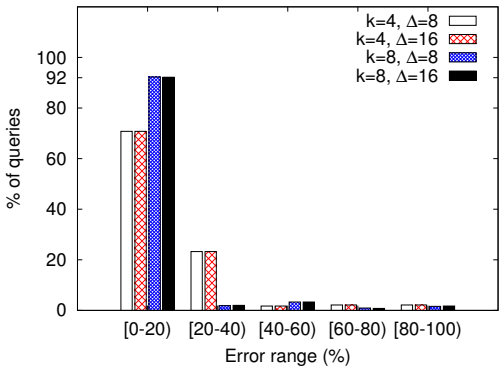


Figure 10. Accuracy of cardinality estimation achieved by XGossip after 30 rounds for different values of k and Δ, $n = 1000$

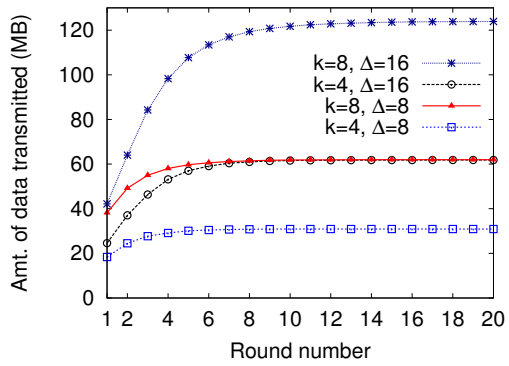
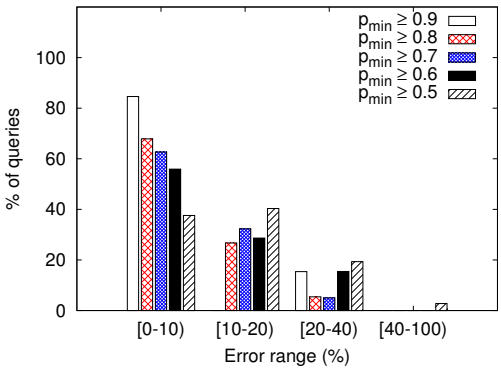


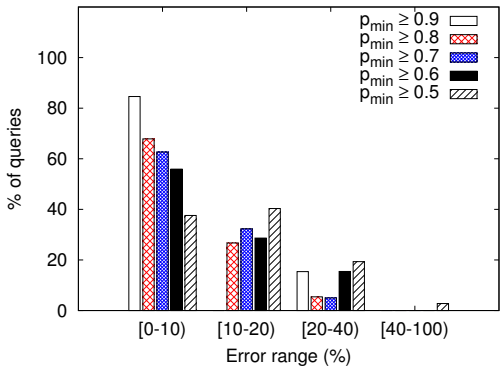
Figure 11. Bandwidth consumption of XGossip for different values of k and Δ, $n = 1000$

Next, we calculated the accuracy of cardinality estimation achieved by XGossip for the five different query sets Q_1 through Q_5 from Table 5. That is, we excluded the 101 queries with minimum similarity between a proxy signature and a signature in the set of signatures which match a query ($p_{min} < 0.5$). Figure 12(a) and (b) show the results for $k = 4$ and Δ equal to 8 and 16. We see relatively high accuracy (84.6%

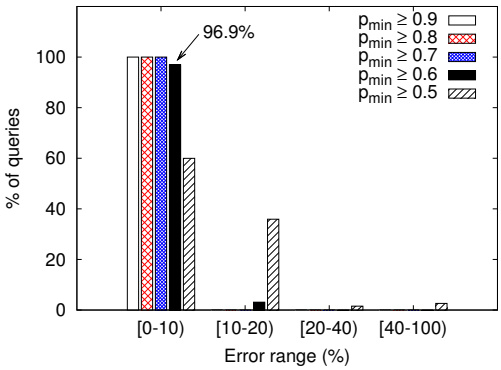
of the queries have relative error below 10%) only for Q_5 ($p_{min} \geq 0.9$). Figure 12(c) and (d) show the results for $k = 8$ and Δ equal to 8 and 16. In contrast to $k = 4$, we observe that only the accuracy for $0.5 \leq p_{min} < 0.6$ is poor. For $p_{min} \geq 0.6$, the accuracy is 96.9%. And for Q_3, Q_4, and Q_5, 100% of the queries had accuracy under 10%.



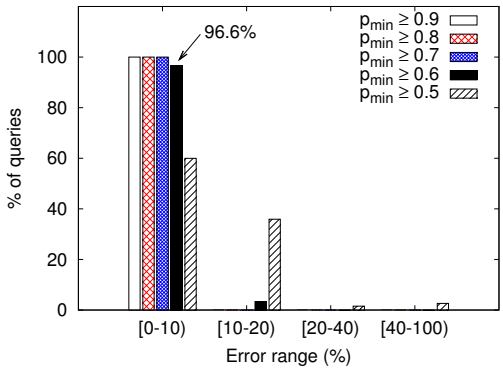
(a) $n = 1000, \Delta = 8, k = 4$, after 30 rounds



(b) $n = 1000, \Delta = 16, k = 4$, after 30 rounds



(c) $n = 1000, \Delta = 8, k = 8$, after 30 rounds



(d) $n = 1000, \Delta = 16, k = 8$, after 30 rounds

Figure 12. Accuracy of cardinality estimation achieved by XGossip for different values of k and Δ, $n = 1000$

We conclude that lower values of k consume less bandwidth, but give lower accuracy and vice versa. Higher values of Δ only increase the amount of data, but do not improve the accuracy after convergence. In order to achieve high accuracy and to consume less bandwidth, we set Δ to 8, k to 8, and l to 10 for the rest of the experiments.

We measured how the accuracy of cardinality estimation improved with increasing the number of rounds for 1000 peers, $\Delta = 8$, and $k = 8$ (Figure 13). At round 5, 83.5% of the queries had a relative error below 20%. The accuracy increased to 88.8% at round 10 and 92.3% at round 20.

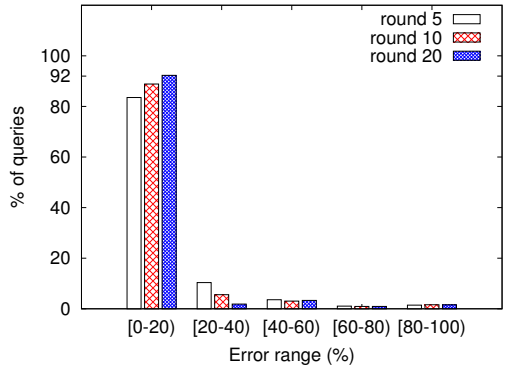


Figure 13. Improvement in the accuracy of cardinality estimation with increasing # of rounds, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

Next, we scaled the number of peers gossiping. We ran XGossip with 500, 1000, and 2000 peers. Figure 14 shows the accuracy of cardinality estimation at different rounds for a varying number of peers. At round 5, we observe the highest accuracy (89.9%) for 2000 peers. The accuracy for 500 and 1000 peers is almost the

same (approximately 84%). By round 10, the difference between 2000 peers and the rest is smaller: 92% vs. 88.8%. And at round 20, all sets of peers reach approximately 92%.

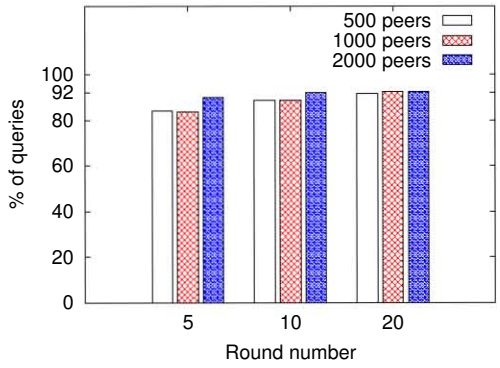


Figure 14. Accuracy of cardinality estimation by varying the # of peers, r$\epsilon \leq 0.2$, $\Delta = 8$, $k = 8$, $l = 10$

Next, we look at the accuracy of each query set from Table 5 for a varying number of peers at different rounds. Figure 15 shows the accuracy of cardinality estimation of query set Q_3 ($p_{min} \geq 0.7$) for 500, 1000, and 2000 peers. The percentage of queries with relative error below 10% was highest for 1000 peers throughout all the rounds: 91.67% at round 5, 97.67% at round 10, and 100% at round 20. The accuracy for all 3 sets of peers reached 100% at the last round. Figures 18, 19, and 20 show the individual percentages for each p_{min} value. The accuracy for all peers and query sets showed similar trends. After 20 rounds of gossiping, all 3 sets of peers achieved high accuracy for Q_3, Q_4, and Q_5 (100%) and Q_2 (approximately 96%) and average accuracy for Q_1 (approximately 60%).

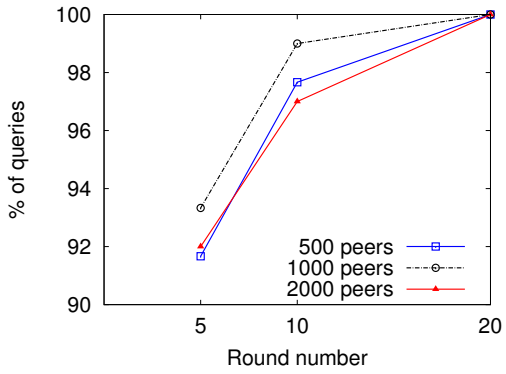


Figure 15. Accuracy of cardinality estimation of query set Q_3 ($p_{min} \geq 0.7$) by varying the # of peers, r$\epsilon \leq 0.1$, $n = 1000$, $\Delta = 8$, $k = 8$

The bandwidth consumption of XGossip for a different number of peers (but the same k and Δ) is almost identical. Figure 16 shows the amount of data transmitted for 500, 1000, and 2000 peers. The only difference is that the amount of data for 500 peers in the first 9 rounds is lower than for 1000 and 2000. In order to explain the similar bandwidth consumption, we measured the average message size per peer for the 3 sets of peers (Table 8). The average message sizes for all 3 are very close, but for 500 peers, the message size is the smallest (1,160.18 bytes). The smaller message size for 500 peers explains why the amount of data transmitted in the first 9 rounds is smaller. We also measured the total number of messages across the 20 rounds (Table 8). The number of messages is almost identical (440,240). The amount of data transmitted is the product of these two numbers (the average message size per peer and total number of messages) which explains why the bandwidth consumption is so similar.

Finally, we look at the bandwidth savings in XGossip through signature com-

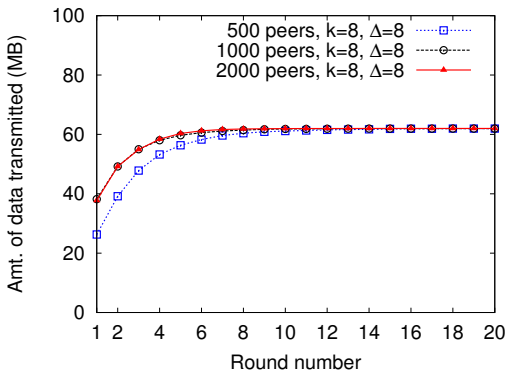


Figure 16. Bandwidth consumption of XGossip by varying the # of peers, $\Delta = 8$, $k = 8$, $l = 10$

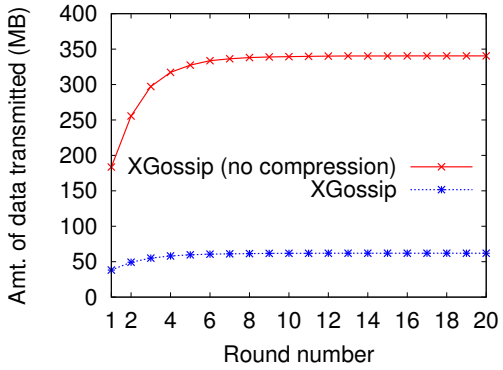
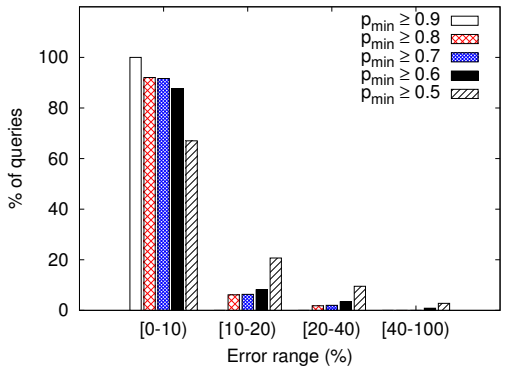


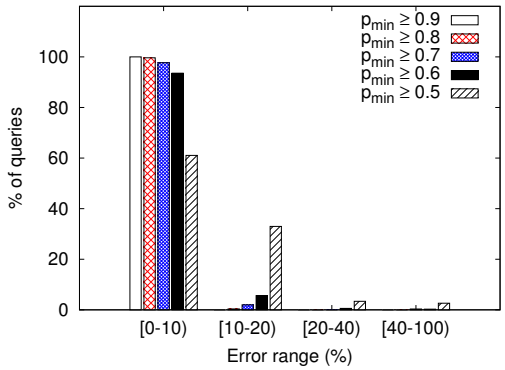
Figure 17. Bandwidth savings in XGossip through signature compression, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

pression. Figure 17 compares the amount of data tranmistted by XGossip with identical settings (k, l, Δ, and total number of peers) when run with and without compression. The maximum amount of data per round transmitted without compression was 340.4 MB. When run with compression, XGossip used more than 5 times less data per round (61.97 MB). This difference is even more significant for the total amount

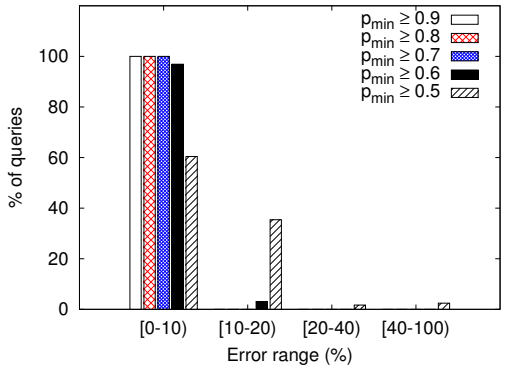
of data across all rounds. XGossip without compression consumed 9,874.2 MB for 20 rounds. With compression, XGossip transmitted only 1,805.9 MB.



(a) $n = 500, \Delta = 8, k = 8$, after 5 rounds

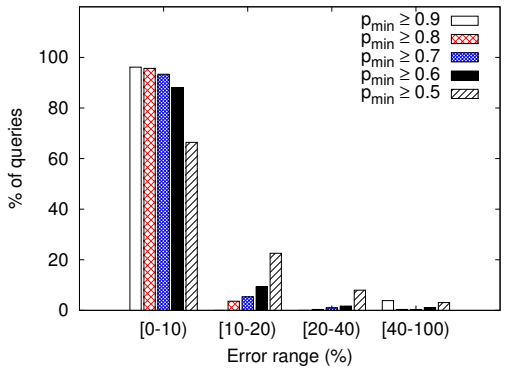


(b) $n = 500, \Delta = 8, k = 8$, after 10 rounds

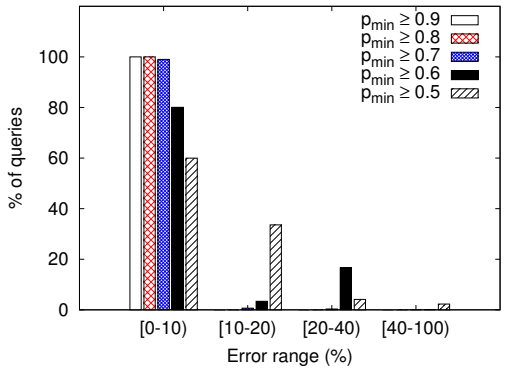


(c) $n = 500, \Delta = 8, k = 8$, after 20 rounds

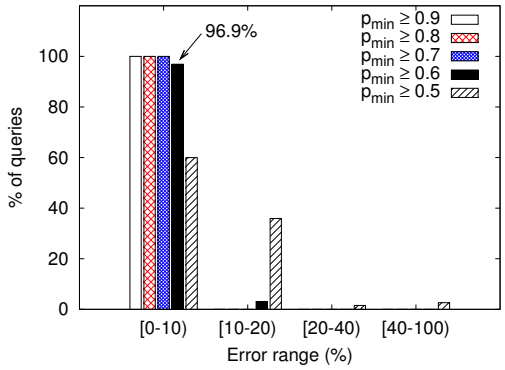
Figure 18. Accuracy of cardinality estimation achieved by XGossip for 500 peers



(a) $n = 1000, \Delta = 8, k = 8$, after 5 rounds

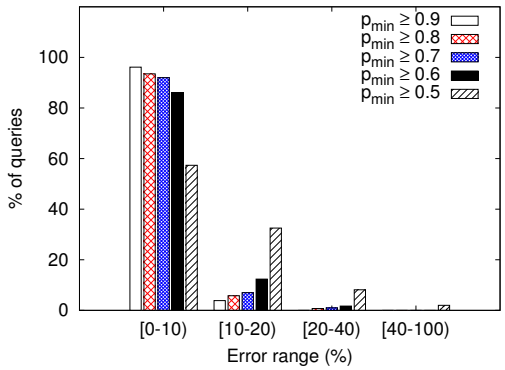


(b) $n = 1000, \Delta = 8, k = 8$, after 10 rounds

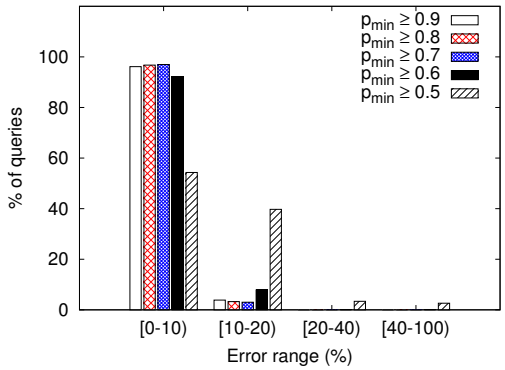


(c) $n = 1000, \Delta = 8, k = 8$, after 20 rounds

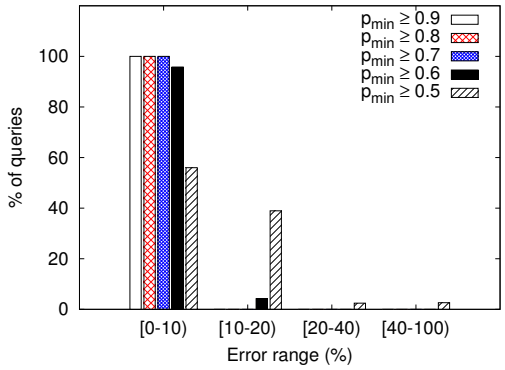
Figure 19. Accuracy of cardinality estimation achieved by XGossip for 1000 peers



(a) $n = 2000, \Delta = 8, k = 8$, after 5 rounds



(b) $n = 2000, \Delta = 8, k = 8$, after 10 rounds



(c) $n = 2000, \Delta = 8, k = 8$, after 20 rounds

Figure 20. Accuracy of cardinality estimation achieved by XGossip for 2000 peers

CHAPTER 6

CONCLUSION AND FUTURE WORK

We implemented two novel gossip-based algorithms (VanillaXGossip and XGossip) which, given an XPath query, estimate the number of XML documents in the network that contain a match for the query. We conducted a comprehensive performance evaluation of both algorithms on Amazon EC2 using a heterogeneous collection of XML documents. Finally, we analyzed the experimental results. The results we obtained were consistent with the theoretical analysis of VanillaXGossip and XGossip. For future work, we plan to extend the XPath query grammar we support to allow the use of value predicates.

APPENDIX A

ALGORITHMS

Algorithm 1: Initialization phase in VanillaXGossip

global: T - sorted tuple list

proc $InitGossip(p)$

1 Let $s_1, ..., s_n$ denote the distinct signatures published by peer p

2 Compute the frequency f_i of each s_i published by p

3 **foreach** s_i **do**

4 | Insert $(s_i, (f_i, 1))$ into T

end

5 Insert $(\perp, (0, 1))$ into T

end

Algorithm 2: Execution phase of VanillaXGossip

```
proc RunGossip(p)
1  Let T_1, T_2, ..., T_R denote the lists received in the current round by peer p
2  T_m ← MergeLists(T_1, T_2, ..., T_R)
3  Send T_m to a random peer p_r and the participating peer p
end

proc MergeLists(T_1, T_2, ..., T_R)
4  T_m ← ∅
5  for r=1 to R do
6  |   c_r ← T_r.begin()
   end
7  while end of every list is not reached do
8  |   s_min ← min{c_1.s, ..., c_R.s}
9  |   sum_f ← 0; sum_w ← 0;
10 |   for r=1 to R do
11 |   |   if c_r.s = s_min then
12 |   |   |   sum_f ← sum_f + c_r.f
13 |   |   |   sum_w ← sum_w + c_r.w
14 |   |   |   c_r ← T_r.next()
   |   |   else
15 |   |   |   sum_f ← sum_f + T_r[⊥].f
16 |   |   |   sum_w ← sum_w + T_r[⊥].w
   |   |   end
   |   end
17 |   Insert (s_min, (sum_f/2, sum_w/2)) into T_m
   end
18 return T_m
end
```

Algorithm 3: Initialization phase of XGossip

```
   global: T - tuple list
  proc InitGossipSend(p)
 1  Let T be initialized as in VanillaXGossip
 2  foreach c ∈ T and c.s ≠⊥ do
 3  |  h_s ← LSH(c.s)
 4  |  foreach h_si ∈ h_s do
 5  |  |  Create a team h_si and pick one id say q for the team at random and
    |  |  send (c.s, (c.f, c.w)) and h_si to the peer responsible for q according to
    |  |  the DHT protocol
    |  end
    end
  end

  proc InitGossipReceive(p, (s, (f, w)), h)
    /* Keep one tuple list per team while receiving */
    /* p is the peer that receives the message */
 6  if T_h does not exists then create T_h
 7  if s is a regular multiset and T_h[s] exists then
 8  |  Update the frequency in the tuple by adding f
    end
    else if s is a regular multiset and T_h[s] does not exist then
 9  |  Insert (s, (f, w)) into T_h
10  |  if ⊥_h does not exist in T_h then
11  |  |  Insert (⊥_h, (0, 1)) into T_h;
12  |  |  InformTeam(p, ⊥_h)
    |  end
    end
13  else if T_h[s] does not exist then
14  |  Insert (s, (f, w)) into T_h;
15  |  InformTeam(p, s)
    end
  end
  proc InformTeam(p, ⊥_h1)
    /* p is the peer executing InitGossipReceive */
16  Suppose h_2, ··· , h_Δ denote the other Chord ids for the team h_1
17  Let peer p be the successor of h_i
18  Send (⊥_h1, (0, 1)) to the successor of h_((i mod Δ)+1
  end
```

Algorithm 4: Execution phase of XGossip

proc *RunGossip*(p)

Let $T_1, T_2, ..., T_R$ denote the lists received in the current round by peer p

Group the lists based on their teams by checking their special multisets. Suppose each group is denoted by G_i.

foreach *group* G_i **do**

    Merge the lists in G_i according to $MergeLists(\cdot)$

    Let T_m denote the merged list

    Compact T_m to save bandwidth /* Optimization */

    Let $h_1, \cdots, h_\Delta$ denote the Chord ids of the team

    Pick an index $j \in [1, \Delta]$ at random such that p is not the successor of h_j

    Send T_m to the peer that is the successor of h_j and to p

end

end

Algorithm 5: Compression of signatures

```
Input: list of signatures; each signature is sorted
Output: A compressed signature
proc CompressSignatures(< s_1, ..., s_W >)
1   j ← 1
2   for i = 1 to W do
3   |   idx[i] ← 0
    end
4   while end of every signature is not reached do
5   |   minVal ← min{s_1[idx[1]], ..., s_n[idx[W]]}
6   |   u_j ← minVal
7   |   for i = 1 to W do
8   |   |   if s_i[idx[i]] = minVal then
9   |   |   |   Set the i^th bit of B_j to 1
10  |   |   |   idx[i] ← idx[i] + 1
    |   |   else
11  |   |   |   Set the i^th bit of B_j to 0
    |   |   end
    |   end
12  |   j ← j + 1
    end
13  return {(u_1, B_1), ..., (u_{j-1}, B_{j-1})}
    end
```

Algorithm 6: Decompression of signatures

Input: a compressed signature
Output: original uncompressed signatures

proc *DecompressSignatures*$(\{(u_1, B_1), \ldots, (u_{\aleph}, B_{\aleph})\})$

```
1  for i = 1 to W do
2  |   s_i ← ∅
   end
3  for i = 1 to ℵ do
4  |   for j = 1 to W do
5  |   |   if j^th bit of B_i equals 1 then
6  |   |   |   Append u_i to the end of s_j so that s_j is sorted
   |   |   end
   |   end
   end
7  return < s_1, ..., s_W >
end
```

APPENDIX B

XPATH GRAMMAR

```
locationPath := step locationPath | step

step := ’//’nodetest | ’/’nodetest

        | step[pred]

pred := @nodetest op1 "nval"

        | pstep op1 "nval"

        | @nodetest op2 "tval"

        | pstep op2 "tval"

pstep := nodetest | nodetest//pstep

         | nodetest/pstep

nodetest := string | *

op1 := = | < | > | <= | >=

op2 := =

nval := number

tval := string
```

REFERENCES

1. Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.
2. caBIG Architecture Workspace: Common Query Language SIG, Summary and Initial Recommendations, March 16, 2005. https://cabig.nci.nih.gov/archive/SIGs/Common%20Query%20Language/ArchWSQuery%20SIG_Recomd_F2F_%20March05.ppt.
3. DXQP - Distributed XQuery Processor: December 13, 2010. http://sig.biostr.washington.edu/projects/dxqp/.
4. Project Voldemort: Reliable Distributed Storage. Invited talk at ICDE 2011, http://project-voldemort.com/.
5. Redis: June 26, 2012. http://redis.io.
6. The caGrid Portal: July 31, 2011. http://cagrid-portal.nci.nih.gov/web/guest.
7. The caGrid xService: March 22, 2011. https://web.cci.emory.edu/confluence/display/xmlds/caGrid+xService.
8. The Cancer Biomedical Informatics Grid: June 21, 2011. https://cabig.nci.nih.gov/.
9. Serge Abiteboul, Ioana Manolescu, Neoklis Polyzotis, Nicoleta Preda, and Chong Sun. XML Processing in DHT Networks. In *Proc. of the 24th IEEE Intl. Conference on Data Engineering*, pages 606–615, Cancun, Mexico, April 2008.
10. Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating

the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. of the 27th International Conference on Very Large Data Bases*, pages 591–600, San Francisco, CA, 2001.

11. Amazon AWS. Amazon Elastic Compute Cloud (EC2), 2010. http://aws.amazon.com/ec2/.
12. Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH Forest: Self-tuning Indexes for Similarity Search. In *Proceedings of the 14th International Conference on World Wide Web*, pages 651–660, 2005.
13. Matthias Bender, Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Global Document Frequency Estimation in Peer-to-Peer Web Search. In *Proceedings of WebDB*, 2006.
14. Noam Berger, Christian Borgs, Jennifer T. Chayes, and Amin Saberi. On the Spread of Viruses on the Internet. In *Proc. of the 16th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 301–310, 2005.
15. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, August 2002.
16. Ken Birman. The Promise, and Limitations, of Gossip Protocols. *Operating Systems Review*, 41(5):8–13, 2007.
17. Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language W3C working draft 16. Technical Report WD-xquery-20020816, World Wide Web Consortium,

August 2002.

18. Angela Bonifati, Ugo Matrangolo, Alfredo Cuzzocrea, and Mayank Jain. XPath Lookup Queries in P2P Networks. In *the 6th annual ACM Intl. Workshop on Web Information and Data Management (WIDM'04)*, pages 48–55, Washington, DC, November 2004.
19. Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip Algorithms: Design, Analysis and Applications. In *INFOCOM 2005*, pages 1653–1664, 2005.
20. Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *Proc. of the 17th International Conference on Data Engineering*, pages 595–604, Heidelberg, Germany, 2001.
21. Emiran Curtmola, Alin Deutsch, Dionysios Logothetis, K. K. Ramakrishnan, Divesh Srivastava, and Kenneth Yocum. XTreeNet: Democratic Community Search. In *Proc. of the 34th VLDB Conference*, pages 1448–1451, Auckland, New Zealand, 2008.
22. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, Stevenson, Washington, 2007.
23. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for

Replicated Database Maintenance. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

24. Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. In *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467 516, 2003.
25. D. Fenstermacher, C. Street, T. McSherry, V. Nayak, C. Overby, and M. Feldman. The Cancer Biomedical Informatics Grid (caBIG). In *Proc. of IEEE Engineering in Medicine and Biology Society*, pages 743–746, Shanghai, China, 2005.
26. M. Fernandez, T. Jim, K. Morton, N. Onose, and J. Simeon. DXQ: A Distributed XQuery Scripting Language. In *4th International Workshop on XQuery Implementation Experience and Perspectives*, 2007.
27. FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.
28. Damien K. Fisher and Sebastian Maneth. Structural Selectivity Estimation for XML Documents. In *Proc. of the 23th IEEE Intl. Conference on Data Engineering*, pages 626–635, Istanbul, Turkey, 2007.
29. Juliana Freire, Jayant R. Harista, Maya Ramanath, Prasan Roy, and Jerome Simone. StatiX: Making XML Count. In *Proc. of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.
30. Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating Data Sources in Large Distributed Systems. In *Proc. of the 29th VLDB Conference*, Berlin, 2003.

31. A. Ganesh, L. Massoulie, and D. Towsley. The Effect of Network Topology on the Spread of Epidemics. In *INFOCOM 2005*, pages 1455–1466, 2005.

32. Luis Garces-Erice, Pascal A. Felber, Ernst W. Biersack, Guillaume Urvoy-Keller, and Keith W. Ross. Data Indexing in Peer-to-peer DHT Networks. In *Proc. of the 24th IEEE Intl. Conference on Distributed Computing Systems*, pages 200–208, Tokyo, March 2004.

33. Georgia Koloniari and Evaggelia Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proc. of the 9th Intl. Conference on Extending Database Technology*, pages 29–47, Crete, Greece, 2004.

34. Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz Kowalski. On the Complexity of Asynchronous Gossip. In *Proc. of the 27th ACM Symposium on Principles of Distributed Computing*, pages 135–144, Toronto, Canada, 2008.

35. Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

36. Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed Similarity Search in High Dimensions using Locality Sensitive Hashing. In *Proc. of the 12th International Conference on Extending Database Technology*, pages 744–755, 2009.

37. Maya Haridasan and Robbert van Renesse. Gossip-Based Distribution Estimation in Peer-to-Peer Networks. In *Proc. of the 7th International Conference on Peer-to-Peer Systems*, Tampa Bay, Florida, 2008.

38. Taher H. Haveliwala, Aristides Gionis, Dan Klein, and Piotr Indyk. Evaluating

Strategies for Similarity Search on the Web. In *Proc. of the 11th international conference on World Wide Web*, pages 432–442, Honolulu, Hawaii, 2002.

39. Yusuo Hu, Jian Guang Lou, Hua Chen, and Jiang Li. Distributed Density Estimation Using Non-parametric Statistics. In *Proc. of 27th International Conference on Distributed Computing Systems (ICDCS)*, pages 28–36, June 2007.
40. Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc. of the 13th ACM Symposium on Theory of Computing*, pages 604–613, Dallas, Texas, 1998.
41. Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-Based Aggregation in Large Dynamic Networks. *ACM Transactions on Computer Systems*, 23:219–252, August 2005.
42. R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized Rumor Spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.
43. Srinivas Kashyap, Supratim Deb, K.V.M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient Gossip-Based Aggregate Computation. In *Proc. of the 35th ACM Principles of Database Systems*, Chicago, IL, 2006.
44. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science*, Cambridge, MA, Oct 2003.
45. Georgia Koloniari and Evaggelia Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Record*, 34(2):6–17, June 2005.
46. Bibudh Lahiri and Srikanta Tirthapura. Identifying Frequent Items in a Network

using Gossip. *Journal of Parallel and Distributed Computing*, 70(12):1241–1253, 2010.

47. Avinash Lakshman and Prashant Malik. Cassandra: A Structured Storage System on a P2P network. In *Proc. of the 2008 ACM-SIGMOD Conference*, Vancouver, Canada, 2008.
48. Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ronald Parr. XPathLearner: An On-line Self-Tuning Markov Histogram for XML Path Selectivity Estimation. In *Proc. of the 28th International Conference on Very Large Data Bases*, pages 442–453, Hong Kong, China, 2002.
49. Cheng Luo, Zhewei Jiang, Wen-Chi Hou, Feng Yu, and Qiang Zhu. A Sampling Approach for XML Query Selectivity Estimation. In *Proc. of the 12th International Conference on Extending Database Technology*, pages 335–344, Saint Petersburg, Russia, 2009.
50. Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-Probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *Proc. of the 33rd VLDB Conference*, pages 950–961, Vienna, Austria, 2007.
51. Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proc. of 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, London, 2002.
52. V. Mayorga and N. Polyzotis. Sketch-Based Summarization of Ordered XML Streams. In *Proc. of the 25th IEEE Intl. Conference on Data Engineering*, pages 541 –552, April 2009.
53. Charles N. Mead. Data Interchange Standards in Healthcare IT – Computable

Semantic Interoperability: Now Possible but Still Difficult, Do We Really Need a Better Mousetrap? *Journal of Healthcare Information Management*, 20(1):71–78, 2006.

54. Tova Milo and Dan Suciu. Index structures for path expressions. In *Proc. of the 7th Intl. Conference on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999.
55. D. Mosk-Aoyama and D. Shah. Fast Distributed Algorithms for Computing Separable Functions. *IEEE Transactions on Information Theory*, 54(7):2997–3007, July 2008.
56. NCI Service Oriented Architecture Strategy. http://cabig.cancer.gov/perspectives/biomedicine/health20/soa/.
57. Robert Neumayer, Christos Doulkeridis, and Kjetil Nørvåg. A Hybrid Approach for Estimating Document Frequencies in Unstructured P2P Networks. *Information Systems*, 36(3):579–595, 2011.
58. Nikos Ntarmos, Peter Triantafillou, and Gerhard Weikum. Statistical Structures for Internet-Scale Data Management. *The VLDB Journal*, 18(6):1279–1312, 2009.
59. Tony C. Pan, Justin D. Permar, Ashish Sharma, David W. Ervin, Tahsin M. Kurc, and Joel H. Saltz. Virtualizing XML Resources As caGrid Data Services. In *Proc. of AMIA Translational Bioinformatics Summit*, San Francisco, CA, 2009.
60. Theoni Pitoura and Peter Triantafillou. Self-Join Size Estimation in Large-scale Distributed Data Systems. In *Proc. of the 24th IEEE Intl. Conference on Data*

Engineering, Cancun, Mexico, April 2008.

61. Boris Pittel. On Spreading a Rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.
62. N. Polyzotis and M. Garofalakis. XCluster Synopses for Structured XML Content. In *Proc. of the 22th IEEE Intl. Conference on Data Engineering*, page 63, Atlanta, GA, April 2006.
63. Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Selectivity Estimation for XML Twigs. In *Proc. of the 20th IEEE Intl. Conference on Data Engineering*, Boston, MA, March 2004.
64. Maya Ramanath, Lingzhi Zhang, Juliana Freire, and Jayant R. Haritsa. IMAX: Incremental Maintenance of Schema-Based XML Statistics. In *Proc. of the 21st International Conference on Data Engineering*, pages 273–284, Tokyo, Japan, 2005.
65. Praveen Rao, Stanley Edlavitch, Jeffery Hackman, Timothy Hickman, Douglas McNair, and Deepthi Rao. Towards Large-scale Sharing of Electronic Health Records of Cancer Patients. In *Proc. of 1st ACM International Health Informatics Symposium*, pages 545–549, Arlington, VA, 2010.
66. Praveen Rao and Bongki Moon. SketchTree: Approximate Tree Pattern Counts over Streaming Labeled Trees. In *Proc. of the 22th IEEE Intl. Conference on Data Engineering*, pages 80–91, Atlanta, Georgia, April 2005.
67. Praveen Rao and Bongki Moon. An Internet-Scale Service for Publishing and Locating XML Documents. In *Proc. of the 25th IEEE Intl. Conference on Data Engineering*, pages 1459–1462, Shanghai, China, March 2009.

68. Praveen Rao and Bongki Moon. Locating XML Documents in a Peer-to-Peer Network using Distributed Hash Tables. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1737–1752, December 2009.

69. Praveen Rao and Vasil Slavov. Towards Internet-Scale Cardinality Estimation of XPath Queries over Distributed XML Data. Technical Report TR-DB-2011-01, http://r.faculty.umkc.edu/raopr/TR-DB-2011-01.pdf, University of Missouri-Kansas City, Kansas City, MO 64110, June 2011.

70. Praveen Rao, Tivakar Komara Swami, Deepthi Rao, Michael Barnes, Swati Thorve, and Prasad Natoo. A Software Tool for Large-Scale Sharing and Querying of Clinical Documents Modeled Using HL7 Version 3 Standard. In *Proc. of 2nd ACM International Health Informatics Symposium*, Miami, FL, 2012.

71. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proc. of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, San Diego, CA, 2001.

72. C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *Proc. of the Workshop on Information Integration on the Web*, pages 116–121, 2004.

73. Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the IFIP/ACM Intl. Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

74. Joel Saltz, Scott Oster, Shannon Hastings, Stephen Langella, Tahsin Kurc, William Sanchez, Manav Kher, Arumani Manisundaram, Krishnakant Shanbhag,

and Peter Covitz. caGrid: Design and Implementation of the Core Architecture of the Cancer Biomedical Informatics Grid . *Bioinformatics*, 22(15):1910–1916, 2006.

75. Carlo Sartiani, Paolo Manghi, Giorgio Ghelli, and Giovanni Conforti. XPeer: A Self-Organizing XML P2P Database System. In *Intl. Workshop on Peer-to-Peer Computing and Databases*, Greece, 2004.
76. The SFS Programming Libraries. https://github.com/okws/sfslite/wiki.
77. Devavrat Shah. Gossip Algorithms. *Foundations and Trends in Networking*, 3(1):1–125, 2009.
78. Devavrat Shah. Network Gossip Algorithms. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3673–3676, 2009.
79. Gleb Skobeltsyn, Manfred Hauswirth, and Karl Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *The 4th Intl. Conference on Ontologies, DataBases, and Applications of Semantics*, Aiga Napa, Cyprus, October 2005.
80. Vasil Slavov and Praveen Rao. Towards Internet-Scale Cardinality Estimation of XPath Queries over Distributed XML Data. In *Proc. of the 6th International Workshop on Networking Meets Databases (NetDB)*, Athens, Greece, 2011.
81. William W. Stead and Herbert S. Lin. Computational Technology for Effective Health Care: Immediate Steps and Strategic Directions. *The National Academies Press, Washington D.C.*, 2009.
82. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.

In *Proc. of the 2001 ACM-SIGCOMM Conference*, pages 149–160, San Diego, CA, August 2001.

83. The Chord/DHash Project. Available from http://pdos.csail.mit.edu/chord/.
84. The Niagara Project. http://www.cs.wisc.edu/niagara/.
85. UW XML Repository. www.cs.washington.edu/research/xmldatasets.
86. Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proc. of the 30th VLDB Conference*, pages 240–251, Toronto, Canada, 2004.
87. Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. of the 8th International Conference on Extending Database Technology*, pages 590–608, Prague, 2002.
88. XML.org. Available from http://www.xml.org/xml.
89. Ning Zhang, M. Tamer Ozsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proc. of the 22th IEEE Intl. Conference on Data Engineering*, page 61, Atlanta, GA, 2006.
90. Ying Zhang and Peter A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.
91. B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

VITA

Vasil Slavov is currently a Ph.D. student in the Department of Computer Science Electrical Engineering at the University of Missouri-Kansas City. He received a B.A. in Computer Science and Mathematics from William Jewell College in 2005 and worked as a Network Administrator at the Kansas City Art Institute until 2011. He joined the Ph.D. program full-time in 2011. His research interests are in the area of XML and P2P networks, and large-scale RDF query processing.