The VLDB Journal manuscript No.
(will be inserted by the editor)

A Gossip-Based Approach for Internet-Scale Cardinality Estimation of XPath Queries over Distributed Semistructured Data

Vasil Slavov · Praveen Rao

Accepted on March 24, 2013

Abstract In this paper, we address the problem of cardinality estimation of XPath queries over XML data stored in a distributed, Internet-scale environment such as a large-scale, data sharing system designed to foster innovations in biomedical and health informatics. The cardinality estimate of XPath expressions is useful in XQuery optimization, designing IR-style relevance ranking schemes, and statistical hypothesis testing. We present a novel gossip algorithm called XGossip, which given an XPath query, estimates the number of XML documents in the network that contain a match for the query. XGossip is designed to be scalable, decentralized, and robust to failures – properties that are desirable in a large-scale distributed system. XGossip employs a novel divide-and-conquer strategy for load balancing and reducing the bandwidth consumption. We conduct theoretical analysis of XGossip in terms of accuracy of cardinality estimation, message complexity, and bandwidth consumption. We present a comprehensive performance evaluation of XGossip on Amazon EC2 using a heterogeneous collection of XML documents.

1 Introduction

We have witnessed a huge success of the P2P model of computing in the last decade. This has culminated in the development of Internet-scale applications such as Kazaa, BitTorrent, and Skype. P2P computing has also become popular in ecommerce and ebusiness and has led to the development of many Internet-scale systems. Innovations in P2P computing, most notably the concept of Distributed Hash Table (DHT) (*e.g.*, Chord [78], Pastry [71], CAN [69], Tapestry [88], Kademlia [52]), has been embraced by key-value stores such as Dynamo [24], Cassandra [48], and Voldemort [5].

The overwhelming success of XML, coupled with the popularity of P2P systems, has led to research in indexing and query processing over XML data in a P2P environment [46,31,11,22,67]. One compelling use case for employing XML and P2P technologies is in the design of large-scale data sharing systems for biomedical and healthcare data. This is because of two reasons: First, it is suggested that scalable clinical data sharing systems can be built using a P2P architecture [77]. Second, HL7 version 3, an XML based standard for representation and interchange of healthcare data (*e.g.*, discharge summaries, lab reports), is becoming a standard for enabling semantic interoperability across systems [54]. A large-scale data sharing system can foster unprecedented innovations in areas such as cancer treatment and diagnosis. For instance, the Cancer Biomedical Informatics Grid (caBIG) [26,8] is a real world data sharing system for collaborative e-science and is growing in popularity.

Selectivity/cardinality estimation is a classical task dealt by query optimizers, for example, to decide the best join order for a query. In this work, we address the task of cardinality estimation of XPath queries in a distributed environment, which is formally stated 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.*

The above cardinality estimate is useful for optimizing a distributed XQuery query, for example, to decide the best join order over distributed XML data depending upon how many documents match different XPath expressions in the query. It can also be used to develop IR-style relevance rank-

V. Slavov
University of Missouri-Kansas City
E-mail: vgslavov@mail.umkc.edu

P. Rao
University of Missouri-Kansas City
E-mail: raopr@umkc.edu

ing schemes. Another use case is in the design of a clinical study, where one important criteria is *the number of subjects* (*i.e.*, participants) to be enrolled in the study. Given the thrust towards building distributed data sharing systems [8], biomedical researchers can be quickly notified (through cardinality estimation) whether sufficient samples are available for a study, without querying the network of distributed data sources. Today, there are tools being developed by the medical informatics community to enable investigators to issue cohort discovery queries to know how many patients are available for a clinical trial (*e.g.*, HERON [9]).

Many techniques have been developed for XML selectivity estimation in a local/centralized environment (*e.g.*, XSKETCH [62], StatiX [30], IMAX [63], XCLUSTER [61], XSEED [86]). These techniques estimate the result set size of an XML query expression. Our goal is different in the sense that we aim to estimate the number of XML documents that match an XPath query instead of the query's result set size. Furthermore, we target a distributed environment, where XML documents are stored across a large number of participating peers. While computing statistics over structured data stored in an Internet-scale environment has been addressed in the past [58, 59], none has focused on computing Internet-scale statistics for the XML data model.

One straightforward approach would be to collect all the XML documents in the network at any one peer and then apply existing techniques for XML selectivity estimation [62, 28, 50]. Unfortunately, this approach will be prohibitively expensive and would not scale. Another issue is that the network may be dynamic where peers can join and leave at any time. Under these circumstances, there are important design requirements for an effective cardinality estimation algorithm. First, the algorithm should be scalable and operate on a large number of peers. Second, it should be decentralized and not rely on any central authority. Third, it should consume minimum network bandwidth and be robust to the dynamism of the network. Fourth, it should provide provable guarantee on the quality of estimates.

In this work, we investigate how gossip algorithms can be designed for XPath cardinality estimation. Gossip (or epidemic) algorithms are attractive for large-scale distributed systems due to their simplicity, scalability, decentralized nature, ability to tolerate failures and the dynamism of the network, and ability to provide probabilistic guarantees. We show that designing a gossip algorithm for cardinality estimation over XML data is a non-trivial task and introduces new challenges due to the very nature of the XML data model. While Ntarmos *et al.* [58] argue that gossip algorithms may not be suitable for statistics generation due to high bandwidth requirement and hop-count, we show that efficient gossip algorithms can indeed be designed.

The key contributions of our work are stated below.

- We design a novel gossip algorithm called XGossip for cardinality estimation of XPath queries in an Internet-scale environment. XGossip relies on the principle of gossip and exchanges concise summaries of XML documents among participating peers. The design of XGossip is inspired by the Push-Sum protocol [45].
- For effective load balancing and reducing bandwidth consumption, XGossip employs: (i) a divide-and-conquer strategy by applying locality sensitive hashing [41] and (ii) a compression scheme for compacting document summaries. As a result, a group of peers gossip only a portion of the entire collection of XML document summaries in the network, and this portion tends to contain similar XML document summaries. This results in faster convergence of cardinality estimates to their true values.
- We conduct theoretical analysis of XGossip in terms of the accuracy and confidence of the cardinality estimation, convergence, message complexity, and bandwidth requirement.
- We present a comprehensive performance evaluation of XGossip in an Internet-scale environment using Amazon EC2 [10]. We use a heterogeneous collection of XML documents to evaluate the effectiveness of XGossip. We show that the empirical results are consistent with the theoretical analysis.

The remainder of the paper is organized as follows. We present the related work in Section 2, background and motivations in Section 3, the design of our gossip algorithms in Section 4, the process of cardinality estimation of XPath queries in Section 5, the analysis of our algorithms in Section 6, a discussion on how churn and failures affect our algorithms in Section 7, a comprehensive performance evaluation of our algorithms in Section 8, an extension to our algorithms in Section 9, and our conclusions in Section 10.

This article is an extension of a previous publication in the 6^{th} International Workshop on Networking Meets Databases (NetDB), 2011 [76]. The new additions in this article include (i) a compression scheme to reduce bandwidth consumption (Sections 4.5), (ii) a comprehensive performance evaluation (Section 8) to show that the empirical results are consistent with the theoretical analysis of the proposed gossip algorithms and to study the impact of churn and failures including peer crashes, and (iii) the proofs of theorems along with new examples and figures.

2 Related Work

2.1 Information Exchange and Aggregate Computation Via Gossip Algorithms

Gossip algorithms provide a means for communication, computation, and information spreading [74, 73]. Prior work on

gossip algorithms have mainly focused on information exchange (or rumor spreading) [35,43,60,25,32,15] and computing aggregates (and separable functions) [44,45,20,42, 56]. The essence of these algorithms lies in the exchange of information or aggregates between a pair of peers (picked randomly if the peers form a complete graph or among neighbors using a probability matrix assuming the peers form an arbitrary graph). It has been shown that after a provably finite number of rounds and a provably finite number of message exchanges, the information has reached all the peers or the aggregates (and separable functions) have converged to the true value.

There are real-world systems that use gossip protocols. Amazon S3 data centers use gossip protocols for spreading server state information [2]. Dynamo [24], Cassandra [48], and Redis [6] also utilize gossip protocols for information exchange among server nodes.

2.2 Statistics Computation in a Distributed Environment

In the area of information retrieval, document frequency estimation in a P2P network has received some attention. Bender *et al.* developed an approach for estimating the global document frequencies using hash sketches [29] in a P2P network [14]. This approach leveraged a DHT overlay network. Neumayer *et al.* developed a hybrid aggregation technique based on hierarchical aggregation and gossip-based aggregation for estimating document frequencies in unstructured P2P networks [57].

Recently, methods for statistics generation in large-scale distributed networks have been developed for relational data. Ntarmos *et al.* [58] developed algorithms for aggregates (*e.g.*, SUM, COUNT) and histograms (*e.g.*, Equi-Width, Equi-Depth histograms) by introducing the idea of Distributed Hash Sketches built over a DHT. Pitoura *et al.* [59] adapted several self-join size estimation algorithms (*e.g.*, bifocal sampling, sample-count) designed for a centralized environment to work in a P2P environment. They also developed a new technique using the Gini coefficient.

A few gossip algorithms have been developed for statistics computation in large-scale networks. Lahiri *et al.* [47] developed gossip algorithms for computing frequent elements. The nodes in a network exchanged small-space synopsis (a.k.a. sketches) of their data during gossip. The focus of this work was mainly on the theoretical results. Hu *et al.* [40] developed a distributed non-parametric density estimation algorithm using a gossip protocol. This resource-constrained algorithm achieved high estimation accuracy with small amount of communication and storage overhead. Haridasan *et al.* [38] developed a gossip algorithm where a node could estimate the distribution of values held by other nodes. The messages contained synopsis of data to reduce storage and bandwidth consumption.

None of the above methods can be adapted in a straightforward way to compute cardinality estimates of XPath queries: This is because of the hierarchical nature of XML and the large number of possible XPath patterns that can match a document. (More details are provided in Section 3.2.)

2.3 Statistics Computation over XML Data

Selectivity estimation over XML data in a local environment has been well studied. The following approaches were designed to estimate the result set size of an XML query expression such as a path expression or a twig pattern. Aboulgana *et al.* proposed path trees and Markov tables to estimate the selectivity of simple XML path expressions [12]. Chen *et al.* proposed a summary structure for estimating the selectivity of XML twig queries [21]. Wu *et al.* developed the pH-join algorithm for complex XML patterns using position histograms [83]. Freire *et al.* proposed StatiX [30] for summarizing XML data with schema information using histograms. Later Ramanath *et al.* extended StatiX to support updates to XML repositories [63].

Lim *et al.* used the feedback from the query execution engine to develop an online approach for large XML repositories [49]. Jiang *et al.* proposed Bloom Histograms to summarize XML data for estimating the selectivity of XML path expressions. Subsequently, Polyzotis *et al.* developed the XSKETCH synopsis model and estimation framework to support complex XPath expressions with both branching and value predicates [62]. Later they proposed XCLUSTER [61] to deal with heterogeneous content in XML documents.

Recently, Zhang *et al.* proposed XSEED [86] that built a small kernel of the XML data and incrementally updated the synopsis based on the required space budget. Fischer *et al.* proposed a new synopsis model based on lossy compression of XML documents that could be constructed in one-pass, to support all the XPath axes [28]. Most recently, unlike previous approaches that constructed a structural synopsis of XML data, Lou *et al.* developed a sampling based approach to capture the tree structure and relationships between nodes in XML documents [50].

There has also been some work on cardinality estimation over streaming XML data [65,53]. These approaches leverage sketching techniques to construct synopsis over XML streams.

3 Background and Motivation

A well-formed XML document follows the syntax rules of the XML specification and can be modeled as an ordered, labeled tree. XPath [16] is a query language for navigating and selecting nodes in an XML document. XPath queries

```
<?xml version="1.0" encoding="ISO−8859−1"?>
<books>
  <book>
    <author>Mark Twain</author>
    <title>The Adventures of Tow Sawyer</title>
    <year>1998</year>
  </book>
  <book>
    <author>C.S. Lewis</author>
    <author>Pauline Baynes</author>
    <title>The Chronicles of Narnia</title>
    <year>2004</year>
    <edition>1</edition>
    <price>$32.00</price>
  </book>
</books>
```

(a)

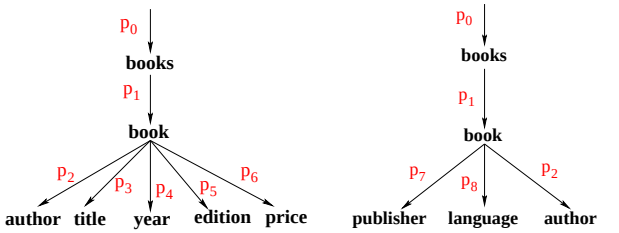
```
<output>
  for $e in doc("....")//books
    where $e/year > 2000
  return $e/title
</output>
```

(b)

```
<output>
<title>The Chronicles of Narnia
</title>
</output>
```

(c)

Fig. 1 (a) XML document d_1. (b) XQuery query q_1. (c) Output obtained by executing q_1 on d_1.



(a) Structural Summary Graph (SSG) of d_1

(b) SSG of another document

Fig. 2 Example

can be represented by *twig patterns*. A twig pattern is essentially a tree-like pattern with nodes representing elements, attributes, and values, and edges representing parent-child or ancestor-descendant relationships. XQuery [18] is a functional query language that subsumes XPath, and allows the creation of new XML content. Figure 1(a) and 1(b) show a well-formed XML document (d_1) and XQuery query (q_1), respectively. Query q_1 when executed on document d_1 will output the title of all books with year greater than 2000. The output is shown in Figure 1(c).

3.1 Signature Representation of XML Documents and XPath Queries

Recently, Rao and Moon [67,66] developed a method to compactly represent XML documents for indexing and locating XML documents in a P2P network. A document is represented by its signature, which is essentially a product of irreducible polynomials. These irreducible polynomials are carefully assigned to the edges of the Structural Summary Graph (SSG) of the document, so that the signature can capture the document's structural properties and content. An XPath query can also be mapped to its signature [67]. A useful necessary condition of this signature representation is that *if a document contains a match for a query, then the query signature divides the document signature* [67]. Another benefit of these document signatures is that they are much smaller in size than the original XML documents [67] and therefore, by exchanging document signatures instead of actual documents, one can conserve bandwidth – a critical resource in an Internet-scale environment.

```
FOR  $gene IN service
("http://cabio.osu.edu/GeneService.wsdl")/Gene,
$go IN service
("http://cabio.osu.edu/GeneOntologyService.wsdl")/GeneOntology,
$microarray IN service
("http://caarray.duke.edu/caArrayService.wsdl")/Microarray
LET $subject := $microarray/experiment/subject
WHERE
  $go/term='vacuole' AND $gene/goAcc=$go/acc AND
  $gene/gbAcc=$microarray/data/geneId AND
  count($microarray/data[geneId=$gene/$gbAcc]/condition)>50
RETURN
<subject>
  <subjectId>{ $subject/lsid }</subjectId>
  <species>{ $subject/species }</species>
  <microarrayData>
    { $microarray/data }
  </microarrayData>
</subject>
```

Fig. 3 An XQuery query supported by caBIG

Example 1 Consider the document d_1 in Figure 1(a) and its Structural Summary Graph (SSG) in Figure 2(a). (The SSG resembles a backward simulation of the XML document tree [55].) Each edge of the SSG is assigned an irreducible polynomial based on the path from the root of the SSG. As shown in the figure, p_0, p_1, ..., and p_6 are irreducible polynomials assigned to the edges of the SSG. For example, the edge from book to author is assigned the polynomial p_2 after hashing the path /books/book/author into a list irreducible polynomials. The signature of the document is constructed by computing the product of all the irreducible polynomials assigned to the edges of the SSG, *i.e.*, the product of p_0, p_1, ..., and p_6.

Example 2 Two different SSGs may contain common paths starting from their respective roots. Consider the SSG shown in Figure 2(b). It has some common paths starting from the root with the SSG in Figure 2(a). Therefore, the polynomials p_0, p_1, and p_2 are assigned to edges in both SSGs.

The presence of recursive element names in an XML document causes cycles in its SSG and there can be multiple occurrences of an irreducible polynomial in the signature [67]. In essence, a document signature can also be viewed as a multiset of irreducible polynomials if the product is avoided. For example, the signature of the XML document d_1 in Figure 1(a) is equivalent to the multiset $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$.

3.2 Key Motivations

The Cancer Biomedical Informatics Grid (caBIG) [26,8], an initiative of the National Cancer Institute, exemplifies a real world data sharing system for collaborative e-science.

It has about 120 participating institutions across the US. Consider the distributed XQuery query in Figure 3, which is supported by caBIG. The query *finds all the expression data where there are at least 50 conditions for genes found in the vacuole* [3]. It performs joins across data exposed by three data services, namely, Gene, GeneOntology, and Microarray. A more powerful query can be constructed wherein the locations of the documents in the network are not explicitly specified. If a P2P architecture is used to make this system scalable, then the above query can be processed in two steps: First, we locate relevant XML documents based on XPath expressions in the query using prior techniques (*e.g.*, XP2P [19], XPeer [72], multi-level bloom filters [34], inverted index using document paths [31], hierarchy of indexes using query-to-query mappings [33], path-based index [75], KadoP [11], *psi*X [67,66], XTreeNet [22]). Next, we apply existing distributed XQuery processing techniques (*e.g.*, XQueryD [70], DXQ [27], DXQP [4], XRPC [87]).

One may wonder if a P2P architecture is suitable for sharing sensitive biomedical and healthcare data (e.g., patient data). Due to legal constraints (e.g., HIPAA [7]) a federated model is typically used so that a data provider has complete ownership of its data and can employ local access control policies [1]. A P2P architecture can provide similar benefits as shown by CDN [64,68]. In CDN, the actual clinical documents (in XML) are never exchanged or transferred across the network. Only authorized peers are allowed to join CDN, unlike open P2P systems such as Kazaa where membership is not controlled.

In this work, we aim to estimate the number of XML documents in a network that contain a match for an XPath query. Though this estimate does not provide the size of the result set of an XPath query, a query optimizer can select appropriate query plans based on how the relevant documents are distributed in the network. For instance, consider the query in Figure 3. If we know the cardinality estimate of XPath expressions such as /Gene/goAcc, /Gene[term= 'vacuole'], /Microarray/data[geneId]/condition, etc., a particular join ordering can be chosen; other applications include IR-style ranking schemes and tools for identifying if sufficient subjects are available for clinical trials.

To the best of our knowledge, we believe our work is the first to address the problem of cardinality estimation of XPath queries in an Internet-scale environment. One may wonder if a technique such as Distributed Hash Sketches [58], designed for structured data, can be adapted for XML data. This would require us to first map each XPath pattern that appears in an XML document into one dimensional space. However, enumerating all possible XPath patterns is computationally expensive and can result in a very large number of patterns due to the hierarchical nature of XML, the presence of many different element and attribute names in a document, and the presence of axes such as '//' (ancestor-descendant) in the queries.

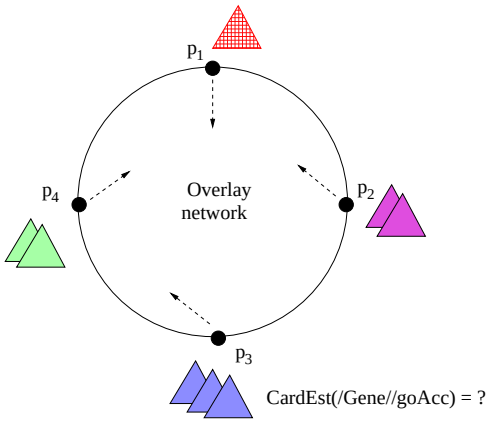


Fig. 4 The system model is shown here. Peers in the network can publish their XML documents. They continuously gossip with each other. At any time, any peer can perform the task of cardinality estimation on an XPath query.

Although gossip algorithms seem simple[1], dealing with XML introduces several challenges. First, if our gossip algorithm begins to execute when a query is posed, like in Push-Sum [45], then we will have to wait for a finite number of rounds before the cardinality estimate is available. On the other hand, if gossip is continuously run in the background, then it is infeasible to gossip all XPath patterns due to their very large number – we expect a heterogeneous collection of XML documents in a distributed environment. Second, our algorithm should scale with increasing number of XML documents and peers in the network and yield effective load balancing. Third, network bandwidth is a critical resource in an Internet-scale environment. Therefore, our gossip algorithm should rely on exchanging a finite number of small sized messages – an essential property of a good gossip algorithm [17].

4 Our Proposed Approach

In this section, we present the Push-Sum protocol introduced by Kempe *et al.* [45]. We draw inspiration from Push-Sum and present a gossip algorithm called VanillaXGossip for XPath cardinality estimation. Subsequently, we employ a novel divide-and-conquer approach to overcome limitations of VanillaXGossip and present an improved algorithm called XGossip. We also provide the theoretical analysis of VanillaXGossip and XGossip. For convenience, notations commonly used in subsequent discussions are listed in Table 1.

[1] The proofs and analyses, however, are mathematically rigorous.

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 (or result set of a query)
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

Table 1 Commonly used notations

4.1 System Model

We assume that peers are connected using a DHT overlay network such as Chord [78]. (See Figure 4.) As in a typical P2P network, a peer owns a set of XML documents. A peer is said to "publish" those documents that it wishes to share with others in the network. The original documents reside at the publishing peer's end. Peers continuously gossip with each other. At any time, any peer can perform the task of cardinality estimation on an XPath query. During this process, a peer will lookup its local state or contact a few other peers to compute the cardinality estimate.

4.2 Push-Sum Protocol

Suppose a P2P network has n peers and each peer p_i has a non-negative value x_i. Suppose we want to estimate the "average" *i.e.,* $\frac{1}{n}\sum_{i=1}^{n} x_i$. In the Push-Sum protocol [45], each peer maintains a sum s_t and weight w_t in round t. In round 0, each peer p_i sends $(x_i, 1)$ to itself. In any round $t > 0$, a peer computes the new sum (or weight) by adding the sum (or weights) of the messages it receives. It sends half of the sum and half of the weight to a randomly selected peer and the remaining half of the sum and weight to itself. In a particular round, the ratio of the current sum and weight is the estimated average. Push-Sum employs uniform gossip where a peer can contact any other peer during a gossip round – in terms of connectivity, the peers form a complete graph.

Theorem 1 (Push-Sum Protocol [45]) *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 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.*

The proof is based on an important property of *mass conservation* [45]. What this means is that in any round, the average of the sums on all the peers is the true average, and the sum of the weights on all the peers is always n. To compute the "sum", *i.e.,* $\sum_{i=1}^{n} x_i$, Push-Sum is run with only one peer starting with a weight of 1 and the rest of the peers starting with a weight of 0 [45]. Push-Sum is able to preserve mass conservation in certain cases of failure and churn. These are discussed in Section 7.

4.3 VanillaXGossip

We draw inspiration from Push-Sum to develop our gossip algorithms VanillaXGossip and XGossip. We select Push-Sum as the basis due to several reasons. Push-Sum relies on uniform gossip where peers form a complete graph with respect to connectivity. Because we assume that peers are connected through a DHT-based structured overlay network, any peer can contact any other peer (in $O(log(n))$ hops). In addition, Push-Sum is synchronous, but peers can follow their local clocks and the convergence holds as long as mass conservation is preserved [45]. (The analysis of a synchronous model is simpler than that of an asynchronous model [45].) In both VanillaXGossip and XGossip, we also compute "average" instead of "sum" because these algorithms run in the background and to guarantee that only one peer will set its weight to 1 and the rest of them to 0, will require sophisticated distributed synchronization.

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 (⊥, (0, 1)) into T
 end
```

Next, we describe VanillaXGossip. Rather than gossiping XPath patterns in XML documents, peers gossip signatures of XML documents. In subsequent discussions, we use the terms "multiset" and "signature" interchangeably. Let f_s denote the frequency of a signature s. VanillaXGossip has

two phases: initialization and execution phases. In the initialization phase as shown in Algorithm 1, each peer creates a sorted list of tuples T using only its local state. Each tuple is of the form $(s, (f_s, 1))$, where s is a signature of an XML document published by the peer and f_s is the number of XML documents having the same signature s and 1 is the initial weight like in Push-Sum. (Two different XML documents can have the same signature [67].) A tuple $(\perp, (0, 1))$ is also added to T, where $\perp$ is a special multiset. The list T is kept sorted by the first item of the tuple (*i.e.*, s), which serves as a primary key. The special multiset $\perp$ is considered to be the largest of all possible signatures when ordered. This means that in any T, $(\perp, (0, 1))$ will appear as the last tuple.

We use the following notations: T.begin(), T.end(), and T.next() are used to iterate over T. For a tuple $c \in T$, $c.s$, $c.f$ and $c.w$ refer to individual elements in the tuple. $T[s]$ denotes the tuple whose signature is s.

Remark 1 A tuple with multiset $\perp$ plays the role of a placeholder in VanillaXGossip for multisets (or signatures) that are not yet known to a peer during a gossip round. This preserves the important property of mass conservation like in Push-Sum.

After initialization, peers begin the execution phase and perform the steps in Algorithm 2 by invoking the procedure $RunGossip()$. During a gossip round, a peer first collects the lists received during that round including the one that it sent to itself. It then merges the lists to update (f_s, w) of each tuple. After merging, the peer sends the merged list with halved frequencies and weights to a randomly selected peer, and sends another copy of that list to itself. (We select a random peer by picking a random Chord id and routing the message to the successor[2] of that id.)

The merging process is unique to VanillaXGossip and is described by procedure $MergeLists()$ in Algorithm 2. Because the lists are sorted by the primary key, the merging phase can be completed in linear time. The minimum key/multiset is selected and its updated sum and weight are computed across all the received lists. If a list does not contain the key, then the sum and weight of $\perp$ are used. (The sum value for $\perp$ is always 0.) In any round, for a tuple $(s, (f, w))$ in the merged list T_m, an estimate of the average of the frequency of s in the network is $\frac{f}{w}$.

Example 3 Figure 5 shows an example of how the merging of three lists T_1, T_2, and T_3 is done using $MergeLists()$. Consider the signature s_1. It is found in T_1, T_2, and T_3. We can compute the new sum and weight for s_1 as follows: $sum_{f_1} = \frac{f_1+f_3+f_4}{2}$ and $sum_{w_1} = \frac{w_1+w_3+w_4}{2}$. Consider the signature s_2. It is not found in T_2 and T_3. Therefore, (f_b, w_b) and (f_c, w_c) are used from T_2 and T_3, respectively, to compute the new sum and weight for s_2 and their values are: $sum_{f_2} = \frac{f_2+f_b+f_c}{2}$ and $sum_{w_2} = \frac{w_2+w_b+w_c}{2}$. □

[2] A *successor of a key* in Chord is a peer mapped to a Chord ID that is the closest to the key (greater than or equal to) in the clockwise direction [78].

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
```

Theorem 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.*

Proof. See Appendix A for the proof. □

Discussion. VanillaXGossip differs from Push-Sum in a few aspects. To illustrate these, suppose Push-Sum is used for cardinality estimation of XPath queries. Push-Sum will be initiated when an XPath query is posed at a peer. This peer will inform other peers about the query. Once a peer becomes aware of the query, it will compute the cardinality estimate of the query on its local documents and gossip this estimate with other peers. After a finite number of rounds, the average of the cardinality estimate is available. On the other hand, VanillaXGossip runs continuously in the background and peers gossip the aggregate values of all the signatures that they are aware of and are oblivious to the queries

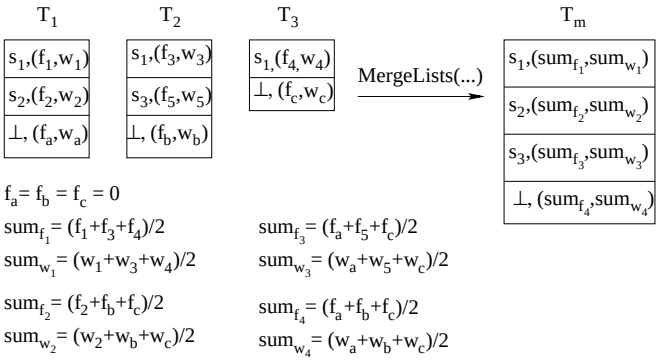
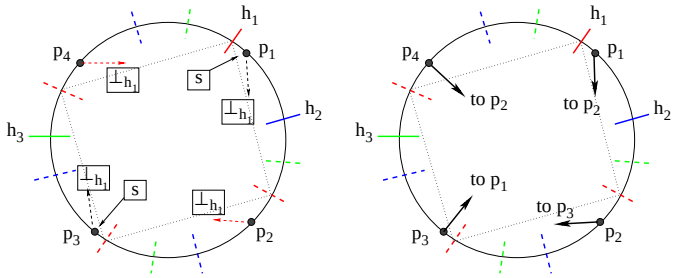


Fig. 5 Merging of lists during VanillaXGossip



(a) Initialization phase (b) Execution phase

Fig. 6 Example for XGossip

that may be posed. VanillaXGossip requires a placeholder to ensure mass conservation and to guarantee convergence similar to Push-Sum. When a query is posed, a peer will lookup its local state and compute the cardinality estimate. We defer the discussion on cardinality estimation until Section 5.

4.4 XGossip: A Divide-and-Conquer Approach

One may notice that in VanillaXGossip, the tuple list T at each peer eventually contains all distinct signatures/multisets in the network. This is inefficient in practice due to limited amount of main memory available at each peer. Also the size of messages can grow very large during gossip. To overcome this limitation of VanillaXGossip, we employ a novel divide-and-conquer strategy using *locality sensitive hashing* (LSH). We call this improved algorithm XGossip. In XGossip, each peer will gossip only a provably finite fraction of distinct multisets in the network. The benefit of XGossip over VanillaXGossip is three-fold: Firstly, each peer will consume less memory. Secondly, each peer will consume less bandwidth during gossip. Thirdly, the convergence of XGossip will require fewer number of rounds.

The concept of LSH, introduced by Indyk and Motwani [41], has been employed in many domains, including indexing high dimensional data and similarity searching [13, 51], similarity searching over web data [39] and in P2P networks [39, 37], ranges queries in P2P networks [36], and so forth. For similarity on sets based on Jaccard index, LSH on a set s can be performed as follows [39, 13]: Pick $k \times l$ random linear hash functions 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 $0 \leq b < p$. Compute $g(s) = min(\{h(x)\})$ over all items in the set as the output hash value for s. It is established that given two sets s_1 and s_2, $Prob(g(s_1) = g(s_2)) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$. Each group of l hash values can be hashed again using another hash function $f(\cdot)$. Thus k hash values are output for a set.

In XGossip, we apply LSH on a document signature. We select $f(\cdot)$ to be the SHA-1 hash function. This way the hash values output by LSH for a signature are 160 bits and map onto the Chord DHT ring. We use the notation $\overline{h_s}$ to denote the vector of hash values produced by LSH on s. We say that $\overline{h_s} = (h_{s1}, \ldots, h_{sk})$ defines k teams for s. Each hash value denotes the id of a team. Suppose Δ denotes the size of each team. Then for any team (with id) h_{si}, we calculate the Chord ids describing that team to be $\{h_{si}, h_{si} + 1 \times \frac{2^{160}}{\Delta}, h_{si} + 2 \times \frac{2^{160}}{\Delta}, \ldots, h_{si} + (\Delta - 1) \times \frac{2^{160}}{\Delta}\}$. (The addition operation will cause the result to wrap around the ring.)

The peers that are successors of the Chord ids defining a team, constitute the members of the team. These peers gossip only a fraction of the distinct signatures in the network. Also, they will exchange messages with only the members of their team during a gossip round. Given two signatures with similarity p, the probability that there is at least one team that gossips both signatures is $1 - (1 - p^l)^k$. (We use α to denote this expression in later sections.) This is an important property of LSH that XGossip builds on. Thus similar signatures are gossiped by the same team with high probability. This increases the chances of finding all the signatures that are required to estimate the cardinality of an XPath query.

Example 4 Consider the DHT ring shown in Figure 6(a). Suppose $k = 3$ and a signature s produces hash values h_1 (red), h_2 (blue), and h_3 (green) after applying LSH. Each team is of size 4. The team h_1 is shown by a dotted square box. In this example, peers p_1, p_2, p_3, and p_4 are members of h_1. □

The divide-and-conquer approach in XGossip raises an interesting issue. Recall that in VanillaXGossip, a single special multiset $\perp$, required for mass conservation, is used by all peers in the network and is sent to a peer picked at random during a gossip round. But in XGossip, a peer cannot maintain one special multiset $\perp$. Rather a peer maintains one special multiset per team to which it belongs to. It sends that special multiset in gossip messages to those peers that are members of that team. In fact, a peer may belong to more than one team. For a team h, its special multiset is denoted by $\perp_h$.

XGossip also has two phases. The first phase is the initialization phase and each peer invokes the procedure

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
```

$InitGossipSend()$ shown in Algorithm 3. Each peer creates the sorted list of tuples T based on the signatures of the XML documents it has published similar to the initialization phase of VanillaXGossip. For each tuple, the peer applies LSH on the tuple's signature and creates k teams. For each team, the peer randomly picks one of its Chord ids and sends the tuple to the successor of that id along with the team id.

When a peer receives a message during initialization via $InitGossipReceive()$ in Algorithm 3, it checks if the signature in the message is a regular multiset, *i.e.,* a document signature. If so, it updates its list along with the special multiset for that team. (Note that a peer maintains a separate tuple list for each team that it belongs to.) But if a peer does not receive any message during initialization, then it does not know which teams it belongs to. *Then how can it initialize its special multiset?* We propose the following: When a peer receives a signature and a team id, it initializes the tuple list for that team with the corresponding special multiset. In addition, it contacts the next peer of the team (in clock-wise direction along the DHT ring) and sends only the special multiset, along with its initial sum and weight *i.e.*, (0,1). A peer on receiving a special multiset for a team forwards it to the next member of the team similarly. (The procedure $InformTeam()$ in Algorithm 3 performs this task.) Note that the special multiset is only forwarded when a peer learns about a team it belongs to for the first time.

Algorithm 4: Execution phase of XGossip

```
 proc RunGossip(p)
1 Let T_1, T_2, ..., T_R denote the lists received in the current
  round by peer p
2 Group the lists based on their teams by checking their special
  multisets. Suppose each group is denoted by G_i.
3 foreach group G_i do
4     Merge the lists in G_i according to MergeLists(·)
5     Let T_m denote the merged list
6     Compact T_m to save bandwidth /* Optimization */
7     Let h_1, ··· , h_Δ denote the Chord ids of the team
8     Pick an index j ∈ [1, Δ] at random such that p is not the
      successor of h_j
9     Send T_m to the peer that is the successor of h_j and to p
  end
 end
```

Example 5 Consider Figure 6(a). Suppose p_1 and p_3 receive a signature s during the initialization phase (solid black arrows). Each informs the next peer in the team with $\perp_{h_1}$ (black dotted arrows). When p_2 and p_4 learn for the first time about team h_1, they forward $\perp_{h_1}$ to their next peers in team h_1 (red dotted arrows). □

During the execution phase of XGossip, a peer groups the messages based on the teams from which they arrive. These are exactly the teams that the peer became aware of during initialization. For each group, $MergeLists()$ is invoked. The merged list for a team is then sent to a randomly selected peer belonging to that team. The steps involved in the execution phase are described in Algorithm 4.

Example 6 Consider Figure 6(b). Because peers p_1, p_2, p_3, and p_4 are the members of the team h_1, they exchange messages belonging to that team during the gossip phase. In a particular round, peers may exchange messages as shown by solid black arrows. □

Theorem 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.*

Proof. See Appendix A for the proof. □

In Theorem 2, the average of the frequency of a signature is computed over the total number of peers, whereas in Theorem 3, the average is computed over the team size.

4.5 Optimizing Bandwidth Usage in XGossip

Because network bandwidth is a critical resource in an Internet-scale environment, we aim to reduce the size of messages exchanged during gossip. The application of LSH enables similar signatures to be gossiped by the same team with high probability. Thus it is more likely that signatures contained in a tuple list (sent to a particular peer during a gossip round) have high similarity. We design a scheme to compact signatures, which is more effective when signatures have high similarity.

Definition 1 (Compressed Signature) Given a set of signatures $s_1, \ldots, s_W$, its compressed signature is a multiset $\{(u_1, B_1), \ldots, (u_\aleph, B_\aleph)\}$, where $\{u_1, \ldots, u_\aleph\} = s_1 \cup s_2 \cup \cdots \cup s_W$, *i.e.*, the union of the multisets, and each B_i is a bitmap of size W. The following properties hold.

a) Consider any (u_i, B_i). If the j^{th} bit of B_i is 1, then $u_i \in s_j$ although there may exist a u_k such that $k \neq i$ and $u_k = u_i$ and the j^{th} bit of B_k equals 0.
b) For any k, $1 \leq k \leq W$, suppose we construct a multiset M by examining the k^{th} bit of $B_1, \ldots, B_\aleph$, where $M = \{u_i \mid k^{th}$ bit of B_i equals $1\}$. Then $M = s_k$.

Example 7 Suppose there are 3 signatures to compress: s_1 = {a, b, c, d, d, d, e, f, g, h, h, h, h}, s_2 = {b, c, c, c, d, d, e, f, f, f, h, h}, and s_3 = {a, b, c, d, d, e, f, g, h, h, h}. The compressed signature is denoted by {(a, `101`), (b, `111`), (c, `111`), (c, `010`), (c, `010`), (d, `111`), (d, `111`), (d, `100`), (e, `111`), (f, `111`), (f, `010`), (f, `010`), (g, `101`), (h, `111`), (h, `111`), (h, `101`), (h, `100`)}. Consider (a, `101`) in the compressed signature. The bitmap `101` indicates that only the first and third signatures contain 'a'. Suppose we construct a multiset using the 2^{nd} bit of every bitmap (as described in Definition 1). This multiset will be {b, c, c, c, d, d, e, f, f, f, h, h} and is identical to s_2. □

Next, we present the algorithms for compression and decompression of signatures. Each signature is sorted so that compression can be done by reading the input signatures just once. When a compressed signature is produced, the pairs are kept sorted by the key u_i. Let $s[i]$ denote the i^{th} element in the sorted signature s. Algorithm 5 describes the steps involved during compression. Because the input signatures are sorted, the union of the signatures can be computed efficiently. During this process, the bitmaps are also generated. The time complexity of our compression algorithm is $O(W \cdot \sum_{i=1}^{W} |s_i|)$, where W denotes the number of signatures that were compressed. Algorithm 6 describes the steps involved during decompression of a compressed signature to obtain the original uncompressed signatures. The time complexity is $O(\aleph \cdot W)$, where $\aleph$ denotes the cardinality of the union of the uncompressed signatures (or multisets).

Algorithm 5: Compression of signatures

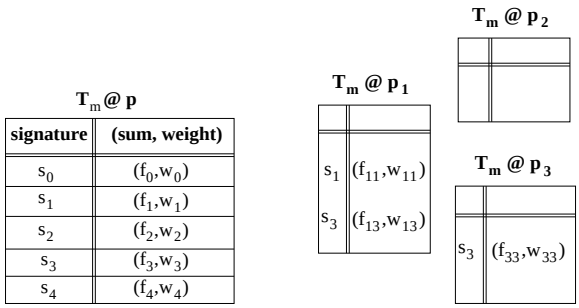
```
Input: list of signatures; each signature is sorted
Output: A compressed signature
proc CompressSignatures((s_1, ..., s_W))
1  j ← 1
2  for i = 1 to W do
3      idx[i] ← 0
   end
4  while end of every signature is not reached do
5      minVal ← min{s_1[idx[1]], ..., s_n[idx[W]]}
6      u_j ← minVal
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), ..., (u_ℵ, B_ℵ)})
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
```

5 Cardinality Estimation of XPath Queries

In this section, we describe the process of cardinality estimation of XPath queries using VanillaXGossip and XGossip. While in VanillaXGossip, the local state at a peer is sufficient to produce the cardinality estimate, in XGossip, a few peers are contacted during cardinality estimation.



(a) VanillaXGossip (b) XGossip

Fig. 7 Tuple lists at peers used for cardinality estimation

5.1 VanillaXGossip

Let us begin with VanillaXGossip. Suppose a query q is issued at peer a p. We search the merged list T_m at p to find every tuple $(s, (f_s, w))$, such that s is a superset of q's signature. (Note that testing for the superset relationship between a document signature and a query signature, when the signatures are viewed as multisets, is equivalent to the divisibility test between them [67].) We compute $\sum \frac{f_s}{w}$ over all such tuples and multiply the sum by n to produce the desired cardinality estimate of q.[3] We can assume that a good estimate of n is known via Push-Sum. By solely looking at the local state of p, we have computed the cardinality estimate of q.

Example 8 Suppose there are two document signatures, namely, s_1 and s_3, in the network that are supersets of q's signature. Consider the tuple list at p shown in Figure 7(a). The qualifying tuples from the tuple list are $(s_1, (f_1, w_1))$ and $(s_3, (f_3, w_3))$. The cardinality estimate of q is given by $n \times (\frac{f_1}{w_1} + \frac{f_3}{w_3})$. □

5.2 XGossip

Now let us focus on XGossip. Suppose q is posed at a peer p. Let $\overline{h_q}$ denote the output of LSH on q's signature. For each team h_{qi} ($1 \leq i \leq k$), we pick a team member at random and send q's signature and h_{qi} to it. The selected team member (or peer) scans its sorted tuple list for team h_{qi} and returns every $(s, (f_s, w))$ such that s is a superset of q's signature. Two or more tuples, each returned by a different peer, may have identical signatures. When this happens, we retain one of the tuples (selected at random) and discard the rest. Finally, we compute $\sum \frac{f_s}{w}$ over the tuples received from k peers (after discarding tuples as needed) and multiply by Δ to produce the desired cardinality estimate of q. In XGossip, we contact k peers during cardinality estimation and this requires $O(k\ log(n))$ hops.

[3] We compute "average" instead of "sum" and therefore, we multiply by n, which is the number of peers in the network.

Example 9 As in Example 8, let s_1 and s_3 be the signatures that are supersets of q's signature in the network. Suppose $k = 3$. We apply LSH on q's signature to obtain 3 team ids, say $\{h_1, h_2, h_3\}$. Let p_1, p_2, and p_3 denote a randomly selected peer from team h_1, h_2, and h_3, respectively. We send q's signature to peers p_1, p_2, and p_3. Figure 7(b) shows the tuple lists maintained by these peers. Peer p_1 returns tuples $(s_1, (f_{11}, w_{11}))$ and $(s_3, (f_{13}, w_{13}))$. Peer p_2 does not return any tuple. Peer p_3 returns the tuple $(s_3, (f_{33}, w_{33}))$. Because there are two tuples with identical signatures, *i.e.*, s_3, we discard one of them. Suppose $(s_3, (f_{13}, w_{13}))$ is discarded. The cardinality estimate of q is given by $\Delta \times (\frac{f_{11}}{w_{11}} + \frac{f_{33}}{w_{33}})$. □

Based on the property of LSH, we know that with probability $\alpha = 1-(1-p^l)^k$ there is at least one team that gossips two different signatures with similarity p. During cardinality estimation, we use the output of LSH on a query signature to find, for each signature that is a superset of a query signature, at least one team that gossips it. It is possible that the similarity between the query signature and a matching document signature is low. For example, if the query has one or two location steps but the matching document contains many different elements and attributes. In such a situation, we may miss some document signatures completely, and obtain a poor quality cardinality estimate. To overcome this situation, we propose the idea of a *proxy signature*.

Definition 2 Suppose a query is posed over documents conforming to an XML Schema S (or a DTD). A proxy signature is the signature of any document that conforms to S and contains the maximum number of distinct elements and attributes in S.

During cardinality estimation of a query, LSH is applied on a proxy signature to identify the teams, instead of on the query signature. The intuition is that the proxy signature will have higher similarity with the matching document signatures for the query. So it is more likely to find all the necessary signatures by contacting the teams generated from the proxy signature. As before, the query signature is sent to a randomly selected member of each team.

6 Analysis of VanillaXGossip and XGossip

In this section, we present the asymptotic analysis of VanillaXGossip and XGossip and compare their accuracy, confidence, convergence, message complexity, and bandwidth consumption. The results are summarized in Table 2.

6.1 Accuracy, Confidence, and Convergence

Suppose R denotes the set of document signatures that are divisible by a query signature. Let $r = |R|$. To fairly com-

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	$O(n\,log(n))$	$O(\frac{log(n)}{n}kD\Delta log(\Delta))$

Table 2 Comparison of VanillaXGossip and XGossip

pare VanillaXGossip and XGossip, we set the desired accuracy and confidence of cardinality estimation to $r\epsilon$ and $(1-\delta)$ for both the algorithms. We state the following theorems and corollary. (See Appendix A for the proofs.)

Theorem 4 *Given an XPath query q, VanillaXGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $(1-\delta)$ in $O(log(n)+log(\frac{1}{\epsilon})+log(\frac{1}{\delta}))$ rounds.*

Theorem 5 *Given an XPath query q, suppose q_{min} denotes the minimum similarity between q's signature and a signature in R. XGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $\alpha\cdot(1-\delta^{'})$ in $O(log(\Delta)+log(\frac{1}{\epsilon})+log(\frac{1}{\delta'}))$ rounds, where $\alpha=1-(1-q_{min}^{l})^{k}$, and k and l denote the parameters of LSH.*

Corollary 1 *XGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $(1-\delta)$ in $O(log(\Delta)+log(\frac{1}{\epsilon})+log(\frac{\alpha}{\alpha+\delta-1}))$ rounds.*

6.2 Message Complexity and Bandwidth Consumption

In VanillaXGossip, eventually all peers gossip every unique signature in the network. Suppose D denotes the number of unique signatures. Therefore, the worst case bandwidth consumed by each peer in a round is $O(D)$, assuming that the size of the longest signature is a small constant. Table 2 shows the worst case bandwidth per round considering all n peers. Similar to Push-Sum, the message complexity of VanillaXGossip is $O(n\,log(n))$.

To analyze, XGossip, let us first discuss the property of consistent hashing in Chord [78]. Suppose there are n peers and K keys. Chord guarantees that with high probability each peer receives at most $\frac{(1+\rho)K}{n}$ keys, where ρ is bound by $O(log(n))$ [78]. In XGossip, we have at most kD team ids (or Chord ids). So each peer becomes the successor for at most $O(\frac{log(n)}{n}kD)$ teams. As there are Δ members per team, there will be at most $O(\frac{log(n)}{n}kD\Delta)$ distinct signatures per team, which denotes the worst case bandwidth consumption of a peer per round. Table 2 shows the worst case bandwidth per round considering all n peers. Given that each team in XGossip exchanges $O(\Delta log(\Delta))$ messages, the overall message complexity is shown in Table 2.

7 Churn and Failures

Kempe *et al.* have discussed a few failure scenarios in Push-Sum [45]. When a message is not delivered successfully to a peer during a gossip round, either because it was lost or because the receiving peer crashed in the initial round, then the sending peer will simply consume the message as if it was never sent and update its local sum and weight to preserve mass conservation. Note that if a peer crashes in the initial round, it is assumed to have not contributed any sum and weight to the network.

If a peer decides to leave the network during gossip, it should do so in an orderly fashion by sending its sum and weight to another peer to preserve mass conservation. If at most 50% of the peers decide to leave in an orderly fashion, then one extra round is needed for convergence [45]. In this case, the average (or sum) computed by Push-Sum would converge to the true average (or sum) before the peers left the network. If a new peer joins the network during gossip, it may receive a gossip message from another peer. It is better to exclude the new peer from participating in the current gossip phase, because it may not know what type of aggregate is being computed just by looking at the sum and weight in the message. So this new peer should discard the message and the sending peer must consume the message. By design, if a peer crashes unexpectedly during gossip, then mass conservation will not be preserved.

A few other scenarios can arise. The gossip interval may be shorter than the network delay between two peers. Messages may be delayed. (Push-Sum does not require the rounds to be synchronous.) In such a situation, mass conservation is preserved after the messages have reached their destination. So the convergence will be delayed. A peer does not send a duplicate message on a failure; it simply consumes the message to preserve mass conservation. The above scenarios apply to both VanillaXGossip and XGossip.

VanillaXGossip and XGossip use Chord's `insert` API to send a message to a peer during a gossip round. [4] Only when a peer receives a message successfully, the insert call at the sender returns with a success status. On failure, the sending peer consumes the *undelivered message* to preserve mass conservation. Similar to Push-Sum, VanillaXGossip and XGossip cannot preserve mass conservation if a participating peer crashes during gossip.

A few new issues arise in our gossip algorithms. The DHT's routing stabilization mechanism runs periodically in the background to cope with changes in the network (*e.g.*, failures, network partitioning, joining and leaving of peers).

[4] This API takes 2 arguments: a key and a value. When invoked, it uses remote procedure calls (RPCs) to send the value to the peer that is the successor of the key. The underlying transport protocol is reliable with features similar to TCP but optimized for high throughput and low latency [23].

It is possible for the successor of any of the Chord ids defining a team to change temporarily or permanently during the execution of XGossip. Then a message sent by a peer of the team may be received by a peer who is actively gossiping but does not belong to the same team. We will call such a message *a wrong-team message*. The receiving peer should reject the message and notify the sending peer, which can then consume the message to preserve mass conservation.

If a new peer joins the network during the execution of XGossip (or VanillaXGossip), it may receive a gossip message from another peer. We will call such a message *a donot-care message*. Because this new peer did not participate in the initialization phase of the gossip algorithm, we exclude it from participating in the current execution phase[5]. This peer should reject the message and notify the sending peer, which can then consume the message to preserve mass conservation.

Scenario	Can (sum, wt) pairs be consumed?		Can disturbance be avoided?	
	XGossip	Vanilla-XGossip	XGossip	Vanilla-XGossip
Undelivered msg.	yes	yes	yes	yes
Wrong-team msg.	yes	–	yes	–
Donot-care msg.	yes	yes	yes	yes
A peer crashes	no	no	no	no

Table 3 Scenarios causing disturbance to mass conservation in XGossip and VanillaXGossip

Solely for the purpose of exposition, we introduce the term "disturbance to mass conservation" to indicate the difference between the average of the sums held by the peers (considering all signatures) and the true average. The theorems stated in earlier sections assume that mass conservation is preserved, and therefore, there is no disturbance to mass conservation. Any scenario that does not preserve mass conservation (*e.g.*, when peers involved in gossiping crash) causes disturbance to mass conservation. It follows that the higher the disturbance, the lower the accuracy of frequency estimates of signatures, and therefore, the lower the accuracy of cardinality estimation achieved by XGossip and VanillaXGossip.

In Table 3, we summarize the different scenarios that cause disturbance to mass conservation in XGossip and VanillaXGossip. In XGossip, disturbances due to undelivered, wrong-team, and donot-care messages can be avoided if the senders can consume the (sum, weight) pairs of signatures in those messages and the receivers can reject those messages (except for undelivered messages). Similar is the case with VanillaXGossip except that wrong-team messages do not arise.

[5] The new peer can participate the next time the gossip algorithm runs.

8 Performance Evaluation

Dataset	# of DTDs	Avg. # of documents per DTD	Total # of documents	Avg. document signature size (bytes)	Avg. document size (bytes)	Max. document size (KB)
D_1	11	190,809	2,098,900	114	1343	39.6
D_2	13	192,223	2,498,900	127	1330	39.6

Table 4 Datasets

We conducted a comprehensive performance evaluation of both VanillaXGossip and XGossip and report the results in this section. We show that the results are consistent with the theoretical analysis presented in Section 6. To highlight why gossip algorithms are a better choice, we implemented an approach called Broadcast and compared it with our gossip algorithms. Broadcast is described in Section 8.7. We also report the behavior of XGossip under churn and failures, including peer crashes, in Section 8.9.

8.1 Implementation

We implemented VanillaXGossip and XGossip in C++ using the Chord package [80] and compiled the code using the GNU g++ compiler (version 4.0.2). In the implementation of VanillaXGossip and XGossip, we followed the steps described in Section 7 to avoid disturbance to mass conservation due to undelivered and donot-care messages. However, for a wrong-team message in XGossip, although the receiving peer discarded the message, the sending peer did not consume the message. One may wonder if this caused substantial disturbance to mass conservation in XGossip: Gladly this was not the case, because in our experiments, the number of wrong-team messages was a tiny fraction (*i.e.*, under 0.55%) of the total number of gossip messages. [6]

8.2 Datasets and Queries

We used two different datasets in the evaluation of VanillaXGossip and XGossip. We generated the datasets using a synthetic XML data generator from IBM and DTDs published on the Internet [81,82,84]. The characteristics of the datasets and document signatures are summarized in Table 4. Note that dataset D_1 is a subset of dataset D_2. We used D_1 to compare VanillaXGossip, XGossip, and Broadcast. We used D_2 to demonstrate an inherent limitation of VanillaXGossip – it suffers from large message sizes during gossip.

[6] If the number of wrong-team messages becomes large, then not consuming them would cause higher disturbance to mass conservation. To avoid this, we can modify the implementation of XGossip to consume these messages.

We generated XPath queries for each DTD by using the XPath generator from the YFilter project [85]. The queries contained the wildcard '*' and the '//' axis. The total number of queries was 753. For each query, q_{min} was at least 0.3. We created 6 different query sets by selecting queries from the original set that had their p_{min} value in a particular range as shown in Table 5. Recall that q_{min} (or p_{min}) is the minimum similarity between a query signature (or a proxy signature) and a document signature in R, *i.e.*, the result set of the query.

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

Table 5 Query sets

8.3 Network Setup and Distribution of Documents

We ran VanillaXGossip and XGossip in an Internet-scale environment using the Amazon Elastic Compute Cloud (EC2) [10]. VanillaXGossip and XGossip were run on 20 EC2 instances or virtual machines. (By default, EC2 allows at most 20 instances per user.) Each instance was a medium instance with 2 virtual cores, 1.7 GB of RAM, 350 GB of disk space, and had moderate I/O performance. We ran all instances in the US East availability zone.

We used two separate setups for the evaluation of VanillaXGossip and XGossip. We used the first setup to show XGossip's superiority over VanillaXGossip and Broadcast. We used the second setup to conduct an in-depth evaluation of XGossip.

In each setup, we conducted the evaluation by establishing a DHT overlay network with n peers. We ran an equal number of peers on each EC2 instance. We distributed the documents in a dataset as follows: We randomly picked z peers per DTD. We distributed the documents conforming to each DTD uniformly across those z peers. Each peer published all the assigned documents.

Table 6 shows the values of n, z, and the number of EC2 instanced used in the first setup. The mean and standard deviation of the number of documents published by a peer for dataset D_1 is also shown. Table 7 shows the values of n, z, and the number of EC2 instanced used in the second setup. The mean and standard deviation of the number of documents published by a peer for dataset D_2 is also shown. Note that dataset D_1 was not used in the second setup because it was a subset of D_2.

Each peer followed its local clock during the execution of VanillaXGossip and XGossip and the interval between successive gossip rounds was fixed at 120 secs.

Total # of peers in the network (n)	# of EC2 instances	# of peers per instance	# of peers picked per DTD (z)	# of documents published by a peer D_1 (μ, σ)
1000	20	50	500	2098.9, 99.1
2000	20	100	1000	1049.5, 49.0

Table 6 Network setup and distribution of documents in D_1 for comparing Broadcast, VanillaXGossip, and XGossip

Total # of peers in the network (n)	# of EC2 instances	# of peers per instance	# of peers picked per DTD (z)	# of documents published by a peer D_2 (μ, σ)
500	20	25	250	4997.8, 257.7
1000	20	50	500	2498.9, 99.1
2000	20	100	1000	1249.5, 49
4000	20	200	2000	624.7, 24.5
8000	20	400	4000	312.3, 12.2

Table 7 Network setup and distribution of documents in D_2 for in-depth evaluation of XGossip

8.4 Evaluation Metrics

We compared VanillaXGossip and XGossip on three metrics: (a) the accuracy of cardinality estimation, (b) the convergence speed of the frequency of signatures, and (c) the bandwidth consumption during gossip. Unless otherwise stated, VanillaXGossip and XGossip were run with compression enabled to minimize the bandwidth consumption.

For the accuracy of cardinality estimation, we calculated the relative error of the cardinality estimate of queries. For the convergence speed, we calculated the mean absolute relative error (MARE) of the frequency estimate of document signatures. For the bandwidth consumption, we calculated the amount of data transmitted per round by all peers.

We also evaluated XGossip by varying the total number of peers in the network and choosing different values for the LSH parameter k and the team size (Δ). We fixed the LSH parameter l at 10 so that $\alpha \approx 0$ when two signatures have similarity less than 0.5. Recall that α is the probability that there is at least one team that gossips two given signatures. We measured the average number of teams that a peer belonged to, the average number of signatures gossiped by a peer and by a team, and the average size of messages exchanged during gossip. In addition, we calculated the total number of messages exchanged across all rounds. We measured the amount of transmitted data per round and also

evaluated the benefit of our compression scheme to reduce the bandwidth consumption of XGossip.

8.5 Diffusion Speed of Signatures During Gossip

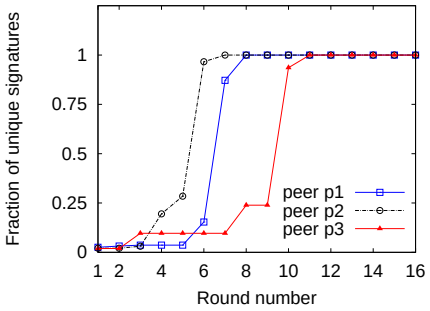


Fig. 8 Diffusion speed of signatures in VanillaXGossip, $n = 1000$

One may wonder how quickly do the signatures diffuse through the network during gossip. Just for the purpose of illustration, the diffusion speed of signatures in VanillaXGossip is shown in Figure 8 by computing the fraction of the unique signatures in the network maintained by a peer (in its tuple list) in each round. We observed that by round 11, three randomly selected peers had learned about all the signatures in the network. The trend would be similar for XGossip when we observe a particular team, but the peers in the team would learn about all their respective signatures in fewer rounds. Although a peer may learn about all the unique signatures in the network, the frequency estimates of these signatures may have high relative error and more rounds may be needed for convergence.

8.6 Comparison of VanillaXGossip and XGossip on Dataset D_1

We compared the convergence speed of the frequency of signatures of VanillaXGossip and XGossip on dataset D_1. Figure 9(a) shows the convergence speed of VanillaXGossip for three randomly selected peers p_1, p_2, and p_3. Beyond round 10, the mean absolute relative error of the frequency estimate of a subset of signatures remained below 10%. Figure 9(b) shows the convergence speed of XGossip for three randomly selected peers which belonged to three different teams. We observed that beyond round 5, the mean absolute relative error of the frequency estimate of signatures for three randomly selected peers remained below 10%. Thus, XGossip converged faster than VanillaXGossip because only Δ peers gossiped a particular signature instead of all the peers in the network. Recall that while the convergence speed of VanillaXGossip depends on $log(n)$

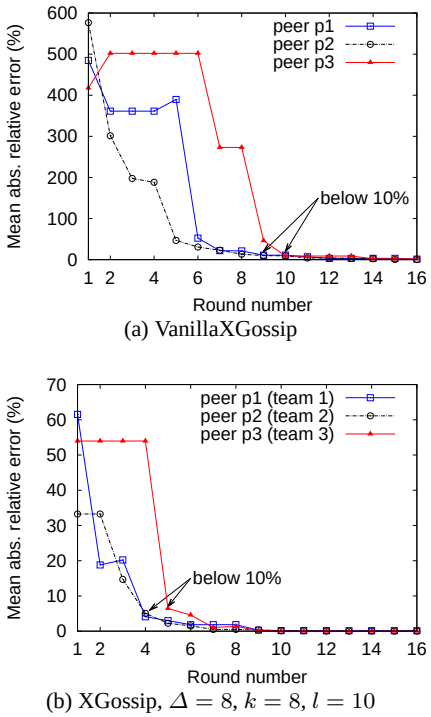


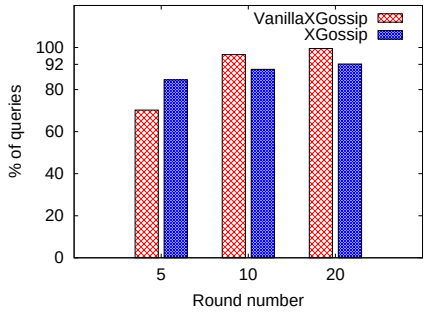
Fig. 9 Comparison of the convergence speed of VanillaXGossip and XGossip, $n = 1000$

(Theorem 2), the convergence speed of XGossip depends on $log(\Delta)$ (Theorem3).

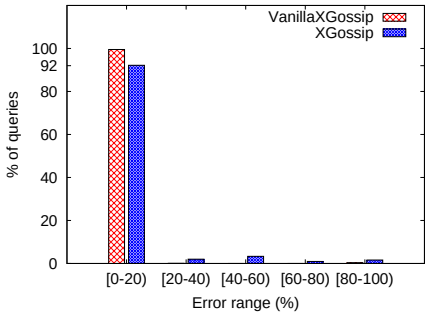
We also compared the accuracy of cardinality estimation of VanillaXGossip and XGossip at different rounds, namely, 5, 10, and 20, for all the 753 XPath queries. (The p_{min} value was in the range [0,1]). Figure 10(a) compares the accuracy of cardinality estimation for VanillaXGossip and XGossip and shows the percentage of queries that had a relative error below 20% (r$\epsilon \leq 0.2$). As expected, both VanillaXGossip and XGossip yielded better accuracy with increasing number of rounds – they were able to estimate more queries under 20% relative error. (This was because the relative error of the frequency estimate of each signature that was a superset of a query signature decreased as the number of rounds increased.)

We observed that at round 5, the accuracy of XGossip was higher than that of VanillaXGossip: XGossip estimated 84.6% of the queries under 20% relative error, but VanillaXGossip could estimate only 70.2% of the queries with the same level of accuracy. We made an interesting observation starting from round 10. VanillaXGossip had better accuracy than XGossip and by round 20, both reached their highest accuracy: 99.5% for VanillaXGossip and 92.2% for XGossip.[7] The reason why XGossip had lower accuracy than VanillaXGossip is straightforward: VanillaXGossip can find the

[7] We ran VanillaXGossip and XGossip for more than 20 rounds, but the accuracy did not improve further.



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



(b) Accuracy by round 20

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

complete result set for a query locally during cardinality estimation, but XGossip may miss signatures in the result set due to the application of LSH. Figure 10(b) shows the distribution of the estimation accuracy for queries by round 20.

Finally, we discuss the bandwidth consumption of VanillaXGossip and XGossip. Figure 11 shows the amount of data transmitted during each of the 20 rounds. (Compression was enabled for both approaches.) The bandwidth consumed by VanillaXGossip grew quickly with increasing number of rounds and reached 731 MB by round 20 when most of the signatures were learned. By design, peers in XGossip gossip only a finite fraction of the signatures in the network. Therefore, XGossip transmitted a meager 25 MB per round – almost 30 times less than VanillaXGossip. The total amount of data transmitted by VanillaXGossip and XGossip in 20 rounds was 10,309 MB and 484 MB, respectively.

8.7 Comparison of Broadcast with Gossip Algorithms

We compared our gossip algorithms with an approach called Broadcast. In Broadcast, each peer computes its tuple list based on the documents that it wishes to publish. It then sends its tuple list to all other peers in the network. When a peer receives tuple lists from other peers, it merges these lists with its own list and updates the frequencies of signatures. In the end, each peer learns about all the distinct signatures in the network along with their frequencies. Note

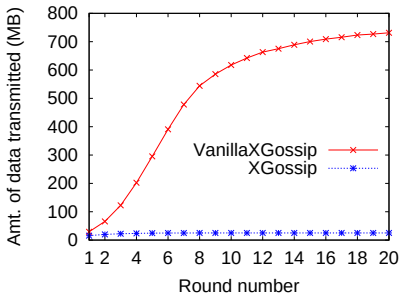


Fig. 11 Bandwidth consumption of VanillaXGossip and XGossip, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

that we expect any peer to be capable of estimating the cardinality of an XPath expression.

We compared the bandwidth consumption of Broadcast, VanillaXGossip, and XGossip for dataset D_1. Figure 12 shows the total amount of data transmitted by peers using the three approaches for $1,000$ and $2,000$ peers. (We used compression in Broadcast, to get the best result.) For VanillaXGossip and XGossip, we report the total bandwidth consumed in 20 rounds. Broadcast consumed significantly higher bandwidth than our gossip algorithms. XGossip consumed the least bandwidth and was better than VanillaXGossip. Broadcast consumed 50 times and 131 times more bandwidth than XGossip on 1000 and 2000 peers, respectively. We conclude that Broadcast will yield poor scalability with increasing number of peers. Note that in Broadcast, we require $O(n^2)$ messages to be exchanged, which is asymptotically higher than our gossip algorithms. (See Table 2.)

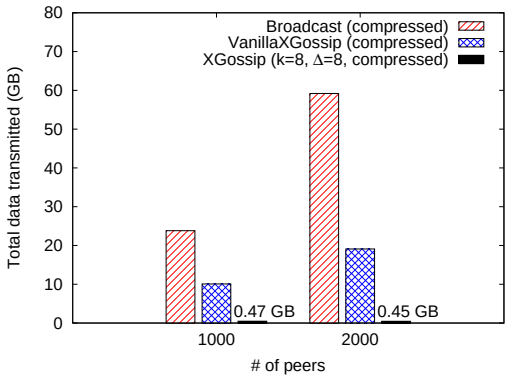


Fig. 12 Bandwidth usage: Broadcast vs VanillaXGossip vs XGossip

8.8 Evaluation of XGossip on Dataset D_2

While XGossip had to compromise on the accuracy of cardinality estimation for dataset D_1, we show that it can scale better than VanillaXGossip for larger datasets by consuming less amount of bandwidth. VanillaXGossip has an inherent limitation: it suffers from large message sizes during gossip.

Dataset D_2 had more unique signatures than dataset D_1. Towards the later rounds in VanillaXGossip, peers had learned about most of the signatures in the network. However, the tuple list at a peer became so large that the peer could not transmit its gossip message through the underlying DHT. This caused VanillaXGossip to fail during the execution phase on D_2. In contrast, XGossip completed successfully by virtue of its scalable design.

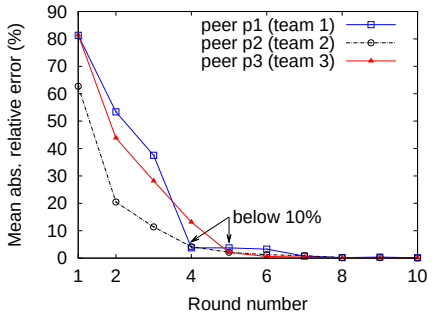


Fig. 13 Convergence speed of XGossip, $n = 1000$, $\Delta = 16$, $k = 4$, $l = 10$

Hereinafter, we focus on the evaluation of XGossip on D_2. We measured the convergence speed of the frequency of signatures in XGossip, and the trend, as shown in Figure 13, was similar to what we observed for D_1. Starting from round 5, the mean absolute relative error of the frequency estimate of signatures for 3 randomly selected peers (from different teams) remained below 10%.

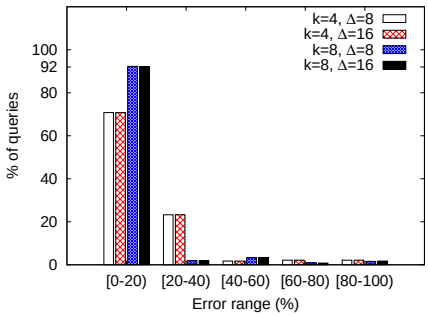


Fig. 14 Accuracy of cardinality estimation achieved by XGossip after 20 rounds for different values of k and Δ, $n = 1000$

8.8.1 Impact of k and Δ on the Performance of XGossip

We also evaluated the effect of the LSH parameter k and team size Δ on the accuracy of cardinality estimation of VanillaXGossip. XGossip ran for 20 rounds and we performed cardinality estimation on all the 753 XPath queries. The results are shown in Figure 14. With $l = 10$, when k was higher, α increased quickly and therefore, the probability of finding at least one team that gossiped two signatures with similarity at least 0.5 increased. Therefore, the accuracy of cardinality estimation was much higher when $k = 8$: 92.3% of the queries were estimated under 20% relative error as compared with 70.8% for $k = 4$. Because XGossip ran for 20 rounds, we did not observe any change in the accuracy of cardinality estimation when Δ was increased from 8 to 16; however, the total time to contact k peers during cardinality estimation increased. (See Table 8.)

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

Table 8 Time taken to contact k peers during cardinality estimation

As reported in Table 2, the bandwidth consumption of XGossip depends on k and Δ. The results are shown in Figure 15. When k was increased from 4 to 8 (and Δ was set to 8), the amount of data transmitted per round (in the later rounds) doubled from 30.9 MB to 61.8 MB. This was because each signature was gossiped by k teams. Again, when Δ was increased from 8 to 16 (and k was set to 8), the bandwidth consumption doubled in the later rounds and reached a maximum of 123.9 MB per round. One may argue that if $\Delta = 1$, the bandwidth consumption will be the least. (This implies no gossip.) However, this is not suitable because if a peer fails, then all the signatures that it is responsible for (after applying LSH) will be lost. Moreover, if a particular signature is frequently accessed during cardinality estimation, then a single peer will be overloaded. When $\Delta > 1$, failures can be tolerated, and the load during cardinality estimation can be distributed across members of a team.

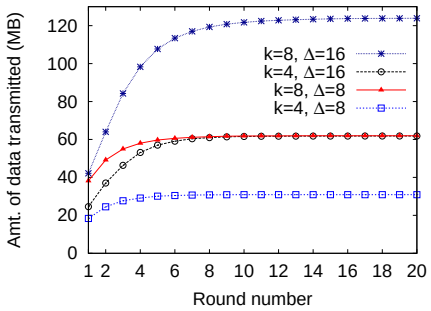
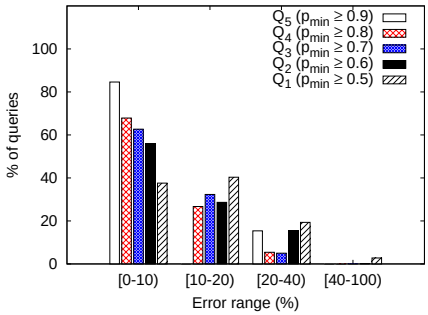
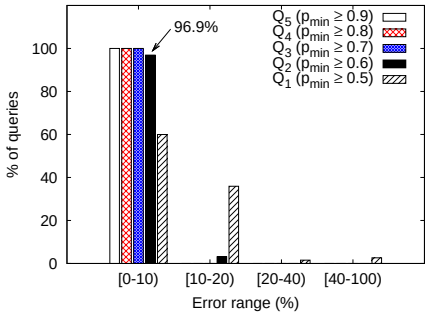


Fig. 15 Bandwidth consumption of XGossip for different values of k and Δ, $n = 1000$

Next, we calculated the accuracy of cardinality estimation achieved by XGossip for query sets Q_1 through Q_5 listed in Table 5. Note that each query set had a different range of p_{min}. In Theorem 3, we showed that the accuracy of cardinality estimation depends on q_{min} (or p_{min} if a proxy signature is used). The higher the value of q_{min} (or

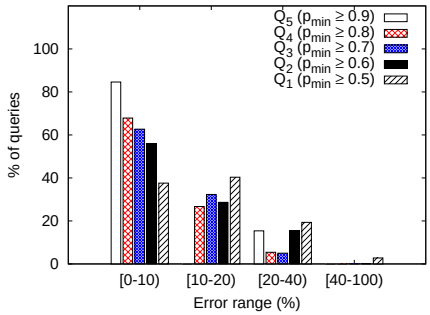


(a) $n = 1000$, $\Delta = 8$, $k = 4$, after 20 rounds

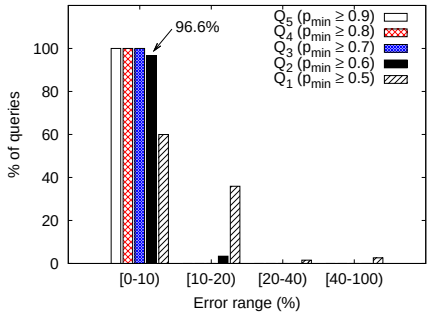


(b) $n = 1000$, $\Delta = 8$, $k = 8$, after 20 rounds

Fig. 16 Accuracy of cardinality estimation achieved by XGossip for $\Delta = 8$



(a) $n = 1000$, $\Delta = 16$, $k = 4$, after 20 rounds



(b) $n = 1000$, $\Delta = 16$, $k = 8$, after 20 rounds

Fig. 17 Accuracy of cardinality estimation achieved by XGossip for $\Delta = 16$

p_{min}), the higher is the probability of having a more accurate estimate. Thus we expect XGossip to achieve the highest level of accuracy for Q_5 and the lowest for Q_1. This is precisely what we observed in our evaluation. Figures 16(a) shows the results for $k = 4$ and $\Delta = 8$. For under 10% relative error, XGossip correctly estimated 84.6% of the queries in Q_5, but could do the same for only 37.6% of the queries in Q_1. Figure 16(b) shows the results for $k = 8$ and $\Delta = 8$. The accuracy of cardinality estimation improved significantly. The reason was by increasing k (and $l = 10$), α increased quickly and therefore, the probability of finding at least one team that gossiped two signatures with similarity at least 0.5 increased. This, in turn, increased the probability of finding all the signatures in the result set of a query, thereby yielding higher accuracy. For all the query sets except Q_1, less than 4% of the queries were estimated with relative error higher than 10%.

When the team size was set to 16, the trends were similar, because we performed the cardinality estimation task after 20 rounds, and by then the frequency estimate of every document signature had converged for both $\Delta = 8$ and $\Delta = 16$. The results are shown in Figures 17(a) and 17(b).

While a higher value of k improved the accuracy of cardinality estimation of XGossip, it also increased the bandwidth consumption. (See Figure 15.) As XGossip achieved good accuracy and bandwidth efficiency for $k = 8$ and $\Delta = 8$, we used these values for the rest of the experiments.

We measured how the accuracy of cardinality estimation improved with increasing number of rounds for 1000 peers, $\Delta = 8$, and $k = 8$ (Figure 18). At round 5, XGossip estimated 83.5% of the queries under 20% relative error; this increased to 88.8% at round 10 and 92.3% at round 20.

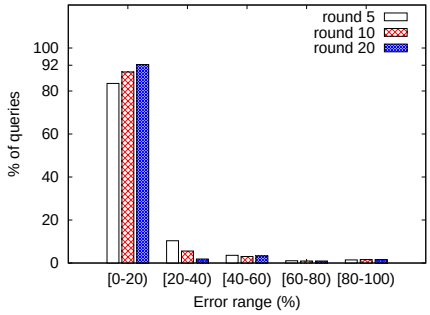


Fig. 18 Improvement in the accuracy of cardinality estimation with increasing # of rounds, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

8.8.2 Evaluation of Compression in XGossip

In Section 4.5, we proposed a compression scheme for XGossip to reduce the size of messages exchanged during gossip rounds. We evaluated the bandwidth savings achieved by our scheme on dataset D_2 with 1000 peers in the network, and $k = 8$ and $\Delta = 8$. Figure 19 compares the bandwidth consumed by XGossip in each round with and without compres-

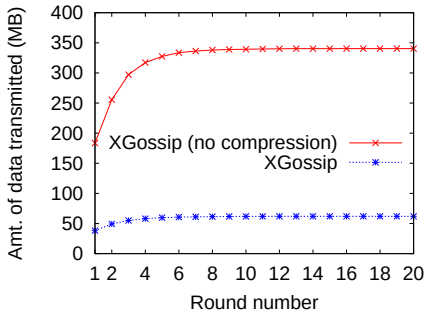


Fig. 19 Bandwidth savings in XGossip through signature compression, $n = 1000$, $\Delta = 8$, $k = 8$, $l = 10$

sion. Interestingly, with compression, XGossip consumed about 5 times less bandwidth than without compression, in the later rounds: 62 MB with compression vs 340 MB without compression. We also computed the total amount of data transmitted in 20 rounds. While with compression, XGossip transmitted 1,806 MB, without compression it transmitted 9,874 MB.

8.8.3 Scalability of XGossip

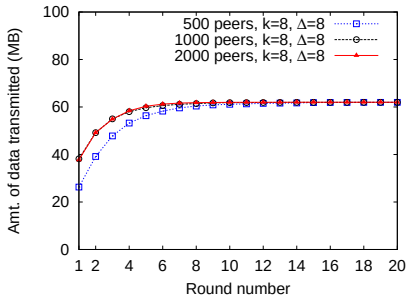


Fig. 20 Bandwidth consumption of XGossip by varying the # of peers, $\Delta = 8$, $k = 8$, $l = 10$

We studied the scalability of XGossip by varying the number of peers in the network (n) from 500 to 8000. In each setting, the same dataset D_2 was used with $k = 8$ and $\Delta = 8$; thus, the total number of teams was identical in each case. Compression was also enabled.

Results for 500, 1000, and 2000 peers. We begin by reporting the results for $n = 500$, $n = 1000$, and $n = 2000$. Figure 20 shows the amount of data transmitted by XGossip in each round. Eventually, the bandwidth consumption of XGossip per round reached 62 MB in each case. (For 500 peers, the bandwidth consumed in the first 9 rounds was lower than that for 1000 and 2000 peers.) Overall, the bandwidth consumption trends were very similar despite different number of peers in the network.

# of peers	Avg. # of teams/ peer	Avg. # of signatures/ peer	Avg. # of signatures/ team	Avg. message size/peer (bytes)	Total # of messages
500	88.40	4024.25	45.52	1,160.18	880,480
1000	44.83	2040.81	45.52	1,265.64	880,480
2000	23.09	1051.2	45.52	1,244.91	883,500

Table 9 Teams, signatures, and messages

In order to explain the observed trends, we measured the average number of teams a peer belonged to, average number of signatures gossiped by a peer eventually, and average number of signatures per team. The results are shown in Table 9. We observed that when the number of peers was doubled in the network, a peer became a member of almost half the number of teams and gossiped almost half the number of signatures. The total number of teams was the same for each case because the value of k was fixed at 8 and peers gossiped the same dataset D_2. Therefore, the average number of signatures assigned to a team was identical.

We also measured the average size of a gossip message transmitted by a peer and total number of messages exchanged in 20 rounds (Table 9). For 500 peers, the average message size was slightly smaller – this was due to the lower bandwidth consumption in the initial rounds of gossiping (Figure 20). The total number of messages was almost identical, because, in a round, each peer sent one message for each team it belonged to. The total amount of data transmitted is the product of the average message size and total number of messages. This explains why the bandwidth consumption of XGossip was very similar for 500, 1000, and 2000 peers.

Next, we present the results for the accuracy of cardinality estimation. Figure 21 shows the percentage of queries estimated by XGossip under 20% relative error at different rounds for $n = 500$, $n = 1000$, and $n = 2000$. We tested all the 753 XPath queries. In each case, the accuracy of estimation improved as the number of rounds increased and reached a maximum of 92% by round 20. Essentially, the quality of estimates produced by XGossip was tolerant to the increase in the number of peers in the network. This is because, in XGossip, the convergence speed of the frequency estimate of document signatures depends on Δ instead of n. (See Theorem 3.) However, the time taken to send a message during a gossip round will increase as n increases – the DHT requires $O(log(n))$ hops to route a message to a peer.

A curious reader may ask why the accuracy of cardinality estimation after 5 rounds was noticeably better for $n = 2000$ than for $n = 500$ or $n = 1000$, although Δ was the same. (Also for $n = 2000$, the accuracy after 10 rounds was slightly better than the others.) This happened because of the way we set up the peers to publish the documents in D_2. (See Section 8.3.) The total frequency of a signature in D_2 was partitioned differently in each setting: When n was higher, more peers published a particular sig-

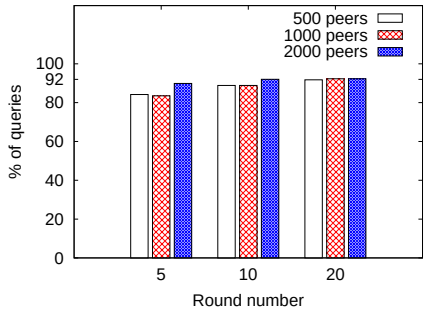
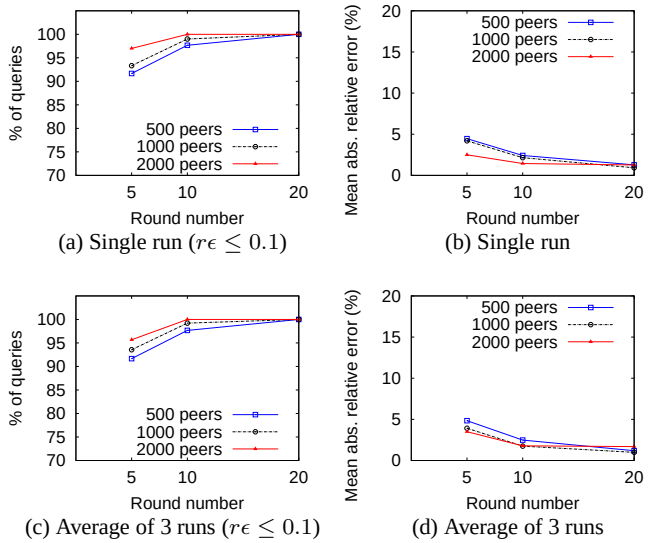


Fig. 21 Accuracy of cardinality estimation by XGossip for different # of peers in the network, $r\epsilon \leq 0.2$, $\Delta = 8$, $k = 8$, $l = 10$



(a) Single run ($r\epsilon \leq 0.1$)
(b) Single run
(c) Average of 3 runs ($r\epsilon \leq 0.1$)
(d) Average of 3 runs

Fig. 22 Accuracy of cardinality estimation by XGossip for Q_3 by varying the # of peers ($\Delta = 8$, $k = 8$)

nature and therefore, held smaller fractions of the total frequency before the beginning of the initialization phase of XGossip. This led to a more even partitioning of the signature's frequency among team members during the initialization phase (Algorithm 3) and therefore, more team members were likely to have that signature in their lists before the beginning of the execution phase of XGossip.

Recall that in XGossip, each peer can be a member of multiple teams and therefore, will maintain a separate list for each team that it belongs to. For the results in Figure 21, we calculated the percentage of lists across all peers that did not contain any signature at the end of the initialization phase. The values were 19.85%, 15.84%, and 9.91% for $n = 500$, $n = 1000$, and $n = 2000$, respectively. This validates our argument that when n was higher, the total frequency of a signature was more evenly partitioned across team members.

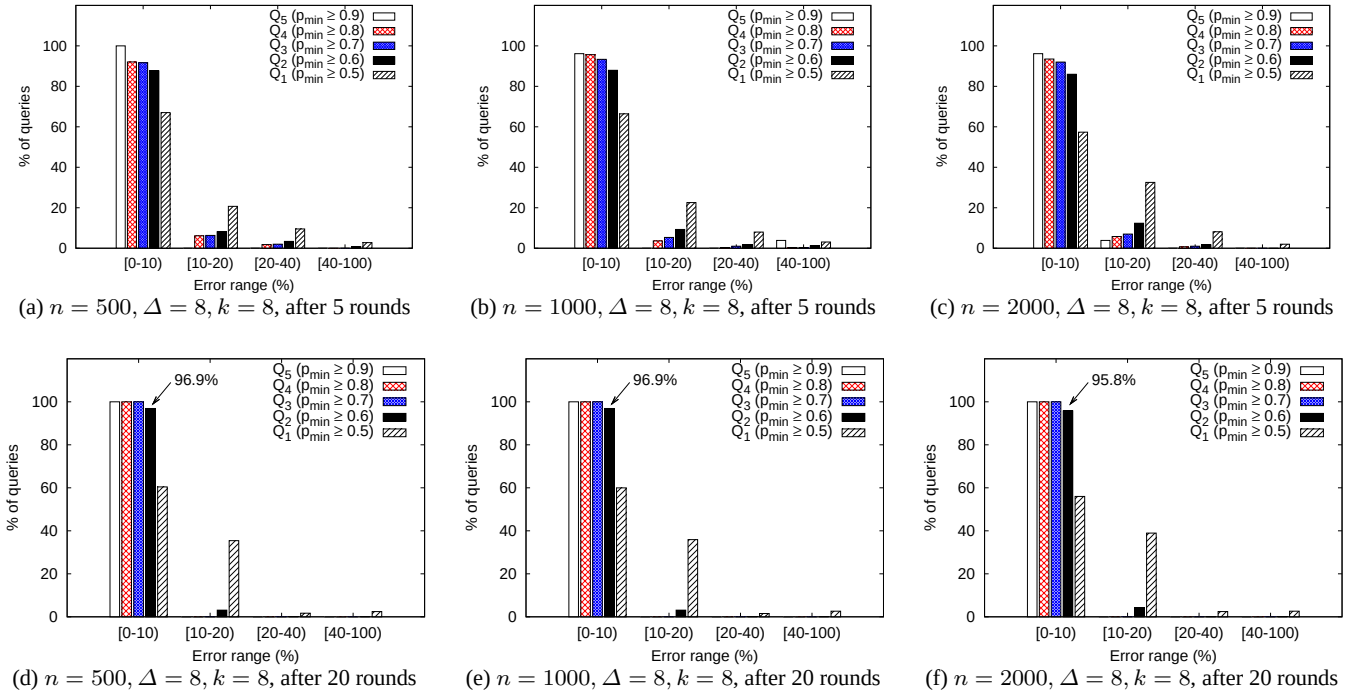
The variation in the initial distribution of signatures affected how quickly the signatures and their frequencies diffused through team members during the execution phase. When $n = 2000$, peers were closer to convergence after 5 rounds than when $n = 500$ or $n = 1000$. This resulted in noticeably better accuracy of cardinality estimation after 5 rounds for $n = 2000$.

Next, we report the accuracy of cardinality estimation for different query sets at different rounds. In Figure 22(a), we first show the trend for query set Q_3. Despite different number of peers in the network, the percentage of queries in Q_3 estimated with relative error under 10% was more than 91% after round 5 and reached 100% after round 20. Figure 22(b) shows the average of the mean absolute relative error for queries in Q_3 after 5, 10, and 20 rounds. Notice that the trends in Figure 22 are consistent with those in Figure 21. We also averaged the accuracy of cardinality estimation over three runs and the results are shown in Figures 22(c) and 22(d).

Figure 23 shows the accuracy of cardinality estimation of XGossip on query sets Q_1 through Q_5 for $n = 500$, $n = 1000$, and $n = 2000$, respectively. Figures 23(a), 23(b), and 23(c) show the accuracy after 5 rounds. Figures 23(d), 23(e), and 23(f) show the accuracy after 20 rounds. (The accuracy after 10 rounds is shown in Appendix B.) As before, XGossip estimated a higher percentage of queries with a relative error of under 10% as the number of rounds increased. It estimated query sets with higher value of p_{min} with higher accuracy. For all the three cases, *i.e.*, 500, 1000, and 2000 peers, after 20 rounds, XGossip estimated 100% of the queries in Q_3, Q_4, and Q_5, and about 96% of the queries in Q_2, and about 60% of the queries in Q_1, with a relative error of under 10%.

One may wonder if the accuracy of cardinality estimation obtained by XGossip is statistically significant and is not obtained by chance. To verify this, we ran XGossip four times with 2000 peers on dataset D_2. Each time, the DHT overlay network was set up on a different set of EC2 instances. (This changed the Chord ids that peers mapped to.) Table 10 shows the accuracy of cardinality estimation of XGossip after 20 rounds, measured as the percentage of queries estimated with relative error under 10%, on each query set for four different runs. The mean (μ) and standard deviation (σ) of the accuracy are also shown in the table. For Q_1 and Q_2, there was a slight variation in percentages across different runs, but for the other query sets there was no variation. This indicates that convergence of XGossip is statistically significant and does not happen by chance. Note that in earlier rounds, we can expect more variation across runs when the convergence has not occurred yet.

Results for 4000 and 8000 peers. Now we report the results of XGossip for $n = 4000$ and $n = 8000$. Table 11 shows how the signatures and teams were distributed across peers, the number of messages, and the average message size ex-



(a) $n = 500$, $\Delta = 8$, $k = 8$, after 5 rounds (b) $n = 1000$, $\Delta = 8$, $k = 8$, after 5 rounds (c) $n = 2000$, $\Delta = 8$, $k = 8$, after 5 rounds

(d) $n = 500$, $\Delta = 8$, $k = 8$, after 20 rounds (e) $n = 1000$, $\Delta = 8$, $k = 8$, after 20 rounds (f) $n = 2000$, $\Delta = 8$, $k = 8$, after 20 rounds

Fig. 23 Accuracy of cardinality estimation achieved by XGossip for 500, 1000, and 2000 peers

Query set	% of queries with $r\epsilon < 0.1$					
	Run 1	Run 2	Run 3	Run 4	μ	σ
Q_1	59.97	55.98	59.97	59.82	58.93	1.97
Q_2	96.91	95.79	96.91	96.63	96.56	0.53
Q_3	100	100	100	100	100	0.00
Q_4	100	100	100	100	100	0.00
Q_5	100	100	100	100	100	0.00

Table 10 Statistical significance of the accuracy of cardinality estimation by XGossip ($n = 2000$, $\Delta = 8$, $k = 8$)

# of peers	Avg. # of teams/ peer	Avg. # of signatures/ peer	Avg. # of signatures/ team	Avg. message size/peer (bytes)	Total # of messages
4,000	13.14	598.24	45.52	1,283.09	885,220
8,000	6.50	295.78	45.52	1,424.40	880,800

Table 11 Teams, signatures, and messages

changed during gossip. As expected, by doubling the number of peers, the load on each peer was almost halved.

Figures 24(a) and 24(b) show the accuracy of cardinality estimation of XGossip on query sets Q_1 through Q_5 with 4000 and 8000 peers, respectively, after 20 rounds. As before, XGossip estimated query sets with higher value of p_{min} with higher accuracy. For both 4000 and 8000 peers, XGossip estimated 100% of the queries in Q_3, Q_4, and Q_5, and about 97% of the queries in Q_2, and about 59% of the queries in Q_1, with a relative error of under 10%.

Figure 24(c) shows the amount of data transmitted by XGossip in each round. For 8000 peers, the bandwidth consumed was slightly higher; we attribute this to the fact that fewer signatures are compressed in each round, thereby reducing the compression ratio. The average message size for 8000 peers was higher than that for 4000 peers and this supports our reasoning. (See Table 11.)

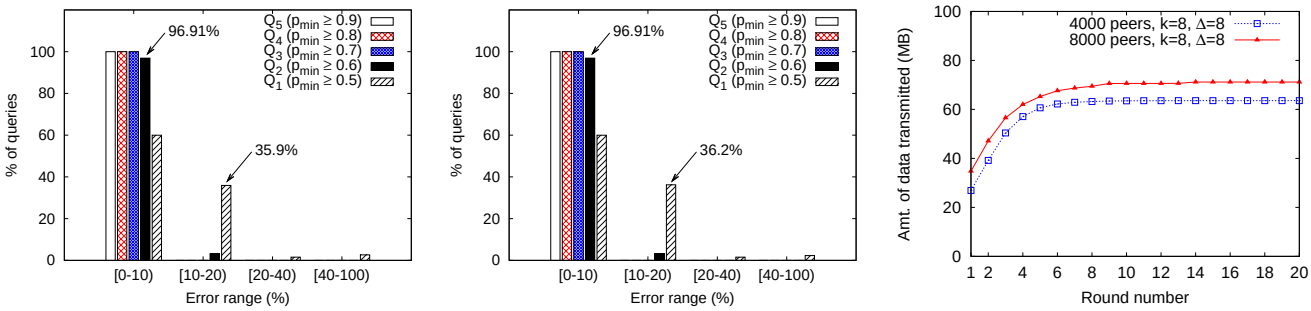
8.9 Churn and Failures

In this section, we report how XGossip performed in the presence of churn and failures, including peer crashes.

8.9.1 Varying the Degree of Churn

A recent study by Stutzbach *et al.* [79] showed that the majority of peers in a P2P network are long-lived peers and the remaining peers are short-lived and join and leave the network at a high rate. It also showed that the session lengths of peers fitted well into log-normal and Weibull distributions. We designed an experiment based on these observations to study the behavior of XGossip under varying degrees of churn.

We set up the network with 8,000 (long-lived) peers. During the execution of XGossip, we varied the number of short-lived peers that would join and leave the network. We set the number of short-lived peers to 0%, 5%, 10%, and 15% of the network size (*i.e.*, the number of long-lived peers). The session lengths of short-lived peers followed a log-normal distribution with $\mu = 0$ and $\sigma = 0.25$ and the



(a) Accuracy after 20 rounds ($n = 4000$) (b) Accuracy after 20 rounds ($n = 8000$) (c) Bandwidth consumption

Fig. 24 Accuracy of cardinality estimation and bandwidth consumption by XGossip for $n = 4000$ and $n = 8000$ ($\Delta = 8$, $k = 8$, $l = 10$)

Degree of churn # of short-lived peers (%)	Undelivered + donot-care messages (%)	Wrong-team messages (%)
0 (0%)	1,641 (0.18%)	4,182 (0.47%)
400 (5%)	4,393 (0.49%)	3,500 (0.39%)
800 (10%)	8,061 (0.91%)	3,542 (0.40%)
1200 (15%)	12,818 (1.45%)	4,654 (0.53%)

Table 12 Number of undelivered, donot-care, and wrong-team messages in XGossip under varying degrees of churn ($n = 8000$, $k = 8$, $\Delta = 8$, after 20 rounds)

round when a short-lived peer joined the network was picked randomly between 1 and 20.

We calculated (a) the number of undelivered and donot-care messages, (b) the number of wrong-team messages, and (c) the accuracy of cardinality estimation. Table 12 shows the sum of the number of undelivered and donot-care messages under varying degrees of churn. As expected, when more short-lived peers joined the network, more donot-care messages were observed. Note that in the implementation of XGossip, we avoided the disturbance to mass conservation due to undelivered and donot-care messages, but did not do so for wrong-team messages. Still, XGossip achieved high accuracy of cardinality estimation after 20 rounds. The results are shown in Figure 25. Table 12 also shows the number of wrong-team messages. Although wrong-team messages were not consumed, they caused negligible disturbance to mass conservation as their count was under 0.55% of the total number of gossip messages.

8.9.2 Varying the Number of Peer Crashes

We studied the behavior of XGossip by allowing peers to crash during gossip. Note that XGossip cannot preserve mass conservation when peers crash, because the tuple lists of these peers will be permanently lost. We set up the network with 8000 peers and picked a subset from these peers at random, containing 5%, 10%, and 20% of the network size, respectively. In one scenario, which we call S_1, each peer (in the selected subset) crashed in a round between 1 and 10, picked randomly. In another scenario, which we call S_2, each peer (in the selected subset) crashed in a round between 11 and 20, picked randomly. In Table 13, we report the accuracy of cardinality estimation by XGossip after 20 rounds for settings S_1 and S_2. This table shows the percentage of queries in the entire query set (containing 753 queries) that were estimated with relative error below 20%. It also shows the accuracy achieved by XGossip when none of the peers crashed (92.56%). As expected, the accuracy dropped when more peers crashed because of higher disturbance to mass conservation. The accuracy was slightly higher when peers crashed in later rounds as this caused lower disturbance to mass conservation.

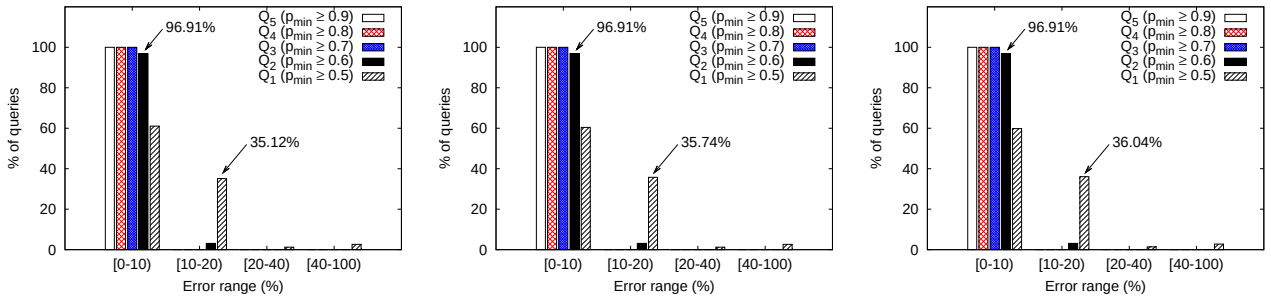
# of peers that crashed (%)	% of queries ($r\epsilon \leq 0.2$)	
	setting S_1	setting S_2
0 (0%)	92.56%	
400 (5%)	89.38%	90.44%
800 (10%)	81.41%	82.20%
1600 (20%)	80.35%	81.27%

Table 13 Accuracy of cardinality estimation achieved by XGossip in the presence of peer crashes ($n = 8000$, $k = 8$, $\Delta = 8$, after 20 rounds)

8.10 Summary of Main Results

We summarize the main results obtained from our performance evaluation.

- XGossip was superior to VanillaXGossip: it converged faster, consumed significantly less bandwidth, and scaled on larger datasets than VanillaXGossip. This is because of the divide-and-conquer strategy in XGossip.
- XGossip obtained high accuracy of cardinality estimation on large number of peers. LSH enabled XGossip to do effective load balancing among peers, and the experimental results were consistent with the theoretical analysis. As expected, query sets with higher value of p_{min} were estimated with higher accuracy.
- Compression yielded significant reduction in the bandwidth consumption of XGossip, as it was able to compress similar signatures effectively.



(a) Degree of churn = 5% (b) Degree of churn = 10% (c) Degree of churn = 15%

Fig. 25 Accuracy of cardinality estimation achieved by XGossip under varying degrees of churn ($n = 8000$, $k = 8$, $\Delta = 8$, after 20 rounds)

– Finally, XGossip tolerated reasonable degrees of churn during execution and still achieved high accuracy of cardinality estimation. However, when peers crashed during gossip, there was higher disturbance to mass conservation and this lowered the accuracy of cardinality estimation achieved by XGossip.

9 Handling Value Predicates in XPath Queries

XGossip (and VanillaXGossip) can be extended to handle XPath queries with value predicates. Consider the document d_1 shown in Figure 1(a) and its SSG in Figure 2(a). The signature scheme proposed in *psi*X [67] captures values in a document by summarizing them under the corresponding element name in the SSG. For example, if a book had multiple prices, then the price values are summarized using a histogram and the histogram is stored with an id p_6 corresponding to `price`. The histograms are stored along with the signature of the document.

During gossip, the tuple list will contain signatures along with their histograms. While merging tuple lists, we merge the histograms of identical signatures when updating their frequencies. (A special multiset does not contain any histogram.) For example, consider signature s_1 in Figure 5. When computing the new sum and weight for s_1, we will also merge the histograms of s_1 from T_1, T_2, and T_3, and store it along with s_1 in T_m.

Consider an XPath query `//book[price > 10]`. To estimate its cardinality, we first determine the set R, which contains the signatures in the network that are supersets of the signature of `//book/price`. For each signature in R, we use its histogram to estimate the frequency of the value predicate `price > 10`. We compute the sum of the frequency of the value predicate on all the signatures in R to obtain the desired cardinality estimate.

10 Conclusions

We have developed a novel gossip algorithm called XGossip for Internet-scale cardinality estimation of XPath queries over distributed semistructured data. The design of XGossip is inspired by the Push-Sum protocol. For effective load balancing and reducing bandwidth consumption, XGossip employs: (i) a divide-and-conquer strategy by applying locality sensitive hashing and (ii) a compression scheme for compacting document summaries. XGossip was evaluated on Amazon EC2 using a large heterogeneous collection of XML documents. XGossip produced high quality cardinality estimates and was efficient in bandwidth usage. The empirical results were consistent with the theoretical analysis of XGossip. We also reported the behavior of XGossip in the presence of churn and failures, including peer crashes.

Acknowledgements This work is supported in part by the National Science Foundation Grant No. 1115871, a grant from University of Missouri Research Board, and Amazon Web Services (AWS) Education Research Grant.

References

1. U.S. National Health Information Network (NHIN) and Open Source Health Information Exchange (HIE) Solutions. http://www.hoise.com/vmw/07/articles/vmw/LV-VM-01-07-29.html.
2. Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.
3. caBIG Architecture Workspace: Common Query Language SIG, Summary and Initial Recommendations. https://cabig.nci.nih.gov/archive/SIGs/Common%20Query%20Language/ArchWSQuery%20SIG_Recomd_F2F_%20March05.ppt.
4. DXQP - Distributed XQuery Processor. http://sig.biostr.washington.edu/projects/dxqp/.
5. Project Voldemort: Reliable Distributed Storage. Invited talk at ICDE 2011, http://project-voldemort.com/.
6. Redis. http://redis.io.
7. Summary of the HIPAA Privacy Rule. http://www.hhs.gov/ocr/privacy/hipaa/understanding/summary/index.html.
8. The Cancer Biomedical Informatics Grid. https://cabig.nci.nih.gov/.
9. The Healthcare Enterprise Repository for Ontological Narration (HERON). Available from http://informatics.kumc.edu/work/wiki/HERON.
10. Amazon Elastic Compute Cloud (EC2), 2010. http://aws.amazon.com/ec2/.
11. S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML Processing in DHT Networks. In *Proc. of the 24th IEEE Intl. Conference on Data Engineering*, Cancun, Mexico, Apr. 2008.

12. A. Aboulnaga, A. R. Alameldeen, and J. 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.
13. M. Bawa, T. Condie, and P. Ganesan. LSH Forest: Self-tuning Indexes for Similarity Search. In *Proceedings of the 14th International Conference on World Wide Web*, pages 651–660, 2005.
14. M. Bender, S. Michel, P. Triantafillou, and G. Weikum. Global Document Frequency Estimation in Peer-to-Peer Web Search. In *Proceedings of WebDB*, 2006.
15. N. Berger, C. Borgs, J. T. Chayes, and A. 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.
16. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, Aug. 2002.
17. K. Birman. The Promise, and Limitations, of Gossip Protocols. *Operating Systems Review*, 41(5):8–13, 2007.
18. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language W3C working draft 16. Technical Report WD-xquery-20020816, World Wide Web Consortium, Aug. 2002.
19. A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. 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, Nov. 2004.
20. S. P. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip Algorithms: Design, Analysis and Applications. In *INFOCOM 2005*, pages 1653–1664, 2005.
21. Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *Proc. of the 17th International Conference on Data Engineering*, pages 595–604, Heidelberg, Germany, 2001.
22. E. Curtmola, A. Deutsch, D. Logothetis, K. K. Ramakrishnan, D. Srivastava, and K. Yocum. XTreeNet: Democratic Community Search. In *Proc. of the 34th VLDB Conference*, pages 1448–1451, Auckland, New Zealand, 2008.
23. F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proc. of the 1st Conference on Symposium on Networked Systems Design and Implementation*, San Francisco, California, 2004.
24. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. 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.
25. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.
26. 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.
27. 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.
28. D. K. Fisher and S. 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. P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.*, 31(2):182–209, Sept. 1985.
30. J. Freire, J. R. Harista, M. Ramanath, P. Roy, and J. Simone. StatiX: Making XML Count. In *Proc. of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.
31. L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *Proc. of the 29th VLDB Conference*, Berlin, 2003.
32. A. Ganesh, L. Massoulie, and D. Towsley. The Effect of Network Topology on the Spread of Epidemics. In *INFOCOM 2005*, pages 1455–1466, 2005.
33. L. Garces-Erice, P. A. Felber, E. W. Biersack, G. Urvoy-Keller, and K. 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, Mar. 2004.
34. 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.
35. C. Georgiou, S. Gilbert, R. Guerraoui, and D. 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.
36. A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.
37. P. Haghani, S. Michel, and K. 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.
38. M. Haridasan and R. 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.
39. T. H. Haveliwala, A. Gionis, D. Klein, and P. 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.
40. Y. Hu, J. G. Lou, H. Chen, and J. Li. Distributed Density Estimation Using Non-parametric Statistics. In *Proc. of 27th International Conference on Distributed Computing Systems (ICDCS)*, pages 28–36, June 2007.
41. P. Indyk and R. 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.
42. M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-Based Aggregation in Large Dynamic Networks. *ACM Transactions on Computer Systems*, 23:219–252, August 2005.
43. R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized Rumor Spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.
44. S. Kashyap, S. Deb, K. Naidu, R. Rastogi, and A. Srinivasan. Efficient Gossip-Based Aggregate Computation. In *Proc. of the 35th ACM Principles of Database Systems*, Chicago, IL, 2006.
45. D. Kempe, A. Dobra, and J. Gehrke. Gossip-Based Computation of Aggregate Information. In *Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science*, Cambridge, MA, Oct 2003.
46. G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Record*, 34(2):6–17, June 2005.
47. B. Lahiri and S. Tirthapura. Identifying Frequent Items in a Network using Gossip. *Journal of Parallel and Distributed Computing*, 70(12):1241–1253, 2010.
48. A. Lakshman and P. Malik. Cassandra: A Structured Storage System on a P2P network. In *Proc. of the 2008 ACM-SIGMOD Conference*, Vancouver, Canada, 2008.

49. L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. 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.
50. C. Luo, Z. Jiang, W.-C. Hou, F. Yu, and Q. 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.
51. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-Probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *Proc. of the 33rd VLDB Conference*, pages 950–961, Vienna, Austria, 2007.
52. P. Maymounkov and D. 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.
53. 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, Apr. 2009.
54. C. 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.
55. T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of the 7th Intl. Conference on Database Theory*, pages 277–295, Jerusalem, Israel, Jan. 1999.
56. D. Mosk-Aoyama and D. Shah. Fast Distributed Algorithms for Computing Separable Functions. *IEEE Transactions on Information Theory*, 54(7):2997–3007, July 2008.
57. R. Neumayer, C. Doulkeridis, and K. Nørvåg. A Hybrid Approach for Estimating Document Frequencies in Unstructured P2P Networks. *Information Systems*, 36(3):579–595, 2011.
58. N. Ntarmos, P. Triantafillou, and G. Weikum. Statistical Structures for Internet-Scale Data Management. *The VLDB Journal*, 18(6):1279–1312, 2009.
59. T. Pitoura and P. 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.
60. B. Pittel. On Spreading a Rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.
61. 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, Apr. 2006.
62. N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity Estimation for XML Twigs. In *Proc. of the 20th IEEE Intl. Conference on Data Engineering*, Boston, MA, March 2004.
63. M. Ramanath, L. Zhang, J. Freire, and J. 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.
64. P. Rao, S. Edlavitch, J. Hackman, T. Hickman, D. McNair, and D. 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.
65. P. Rao and B. 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, Apr. 2005.
66. P. Rao and B. 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.
67. P. Rao and B. 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.
68. P. Rao, T. K. Swami, D. Rao, M. Barnes, S. Thorve, and P. 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.
69. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. 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.
70. 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.
71. A. Rowstron and P. 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, Nov. 2001.
72. C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-Organizing XML P2P Database System. In *Intl. Workshop on Peer-to-Peer Computing and Databases*, Greece, 2004.
73. D. Shah. Gossip Algorithms. *Foundations and Trends in Networking*, 3(1):1–125, 2009.
74. D. Shah. Network Gossip Algorithms. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3673–3676, 2009.
75. G. Skobeltsyn, M. Hauswirth, and K. 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, Oct. 2005.
76. V. Slavov and P. 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.
77. W. W. Stead and H. S. Lin. Computational Technology for Effective Health Care: Immediate Steps and Strategic Directions. *The National Academies Press, Washington D.C.*, 2009.
78. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the 2001 ACM-SIGCOMM Conference*, pages 149–160, San Diego, CA, Aug. 2001.
79. D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-peer Networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, Rio de Janeriro, Brazil, 2006.
80. The Chord/DHash Project. Available from http://pdos.csail.mit.edu/chord/.
81. The Niagara Project. http://www.cs.wisc.edu/niagara/.
82. UW XML Repository. www.cs.washington.edu/research/xmldatasets.
83. Y. Wu, J. 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.
84. XML.org. Available from http://www.xml.org/xml.
85. Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
86. N. Zhang, M. T. Ozsu, A. Aboulnaga, and I. 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.
87. Y. Zhang and P. A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.
88. B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

Appendix A

Theorem 2

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.*

Proof.

In this proof, we show that VanillaXGossip does not violate "mass conservation" and therefore the proof for Push-Sum holds for VanillaXGossip. Without loss of generality, suppose only one signature s exists in the network and a peer p is observed. Then p has either published s or has not.

Case 1: Suppose p has not published s. Suppose p knows this fact, and starts with a sum and weight (0,1). Suppose in round t, p receives (f_s, w) where $f_s > 0$ such that in all the previous rounds p has only received messages from those peers that have not published s. Suppose $(0, w^{'})$ be the sum and weight in round $t-1$ at p. Then p will compute its new sum to be $f_s + 0$ and weight $w + w^{'}$ and sends $\frac{f_s+0}{2}$ and $\frac{w+w^{'}}{2}$ to another peer. Now VanillaXGossip resembles Push-Sum and mass conservation is preserved and the proof of Push-Sum holds.

But if p does not know the fact that s exists, and uses the placeholder $\perp$ and starts with (0,1) as the sum and weight for this placeholder signature. Suppose we replay the actions up to round t. Now in round t, p receives (f_s, w) where $f_s > 0$. Now the sum and weight based on $\perp$ will be $(0, w^{'})$ in round $t-1$ at p. Peer p will compute its new sum to be $f_s + 0$ and weight $w + w^{'}$ and sends $\frac{f_s+0}{2}$ and $\frac{w+w^{'}}{2}$ to another peer. So even when p does not know about the existence of s, it can arrive at the right sum and weight in round t to guarantee mass conservation.

Case 2: Suppose p has published s. Without loss of generality, suppose in round t, p receives the placeholder signature with $(0, w^{'})$ from some peer q. This means that so far q has received messages from peers that do not know about s to begin with. Then p computes the sum and weight as $f_s + 0$ and $w + w^{'}$. This would be the same if the peers that have sent a message to q (including q) until round $t-1$, started with (0,1) for signature s if they assumed that s existed in the network. Then mass conservation is guaranteed and the proof of Push-Sum holds.

From cases 1 and 2, we can conclude that for any s, mass conservation holds in VanillaXGossip and therefore, the proof of Push-Sum holds.

□

Theorem 3

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.

Proof. Suppose $\overline{h_s} = (h_{s1}, ..., h_{sk})$ denotes the output of LSH on s. Without loss of generality, consider the team h_{si}. During initialization, any peer that published an XML document whose signature is s, will send $(s, (f, w))$ to a member of h_{si}. At the end of the initialization phase, mass conservation holds for s w.r.t. team h_{si}. This is because the average of the frequency of s across all the members of team h_{si} is the true average, and the sum of weights for s is Δ. During the execution phase, s is gossiped by the members of team h_{si} and mass conservation is preserved like in VanillaXGossip due to the use of the special multiset $\perp_{h_{si}}$. Now the situation is identical to VanillaXGossip except that the number of peers involved in computing the average is Δ. Hence the above theorem holds. □

Theorem 4

Given an XPath query q, VanillaXGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $(1-\delta)$ in $O(log(n) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta}))$ rounds.

Proof.

From Theorem 2, we know that the frequency of a signature in R can be estimated with a relative error of at most ϵ and confidence $(1-\delta)$ in $O(log(n) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta}))$, where n denotes the number of peers in the network. In VanillaXGossip, the frequencies of r signatures in R is used to compute the cardinality estimate of q, and therefore, the total relative error is at most $r\epsilon$.

□

Theorem 5

Given an XPath query q, suppose q_{min} denotes the minimum similarity between q's signature and a signature in R. XGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $\alpha \cdot (1-\delta^{'})$ in $O(log(\Delta) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta^{'}}))$ rounds, where $\alpha = 1-(1-q_{min}^{l})^{k}$, and k and l denote the parameters of LSH.

Proof.

From Theorem 2, we know that the frequency of a signature in R can be estimated with a relative error of at most ϵ and confidence $(1-\delta^{'})$ in $O(log(\Delta) + log(\frac{1}{\epsilon}) + log(\frac{1}{\delta^{'}}))$, where Δ denotes the team size. In XGossip, the frequencies of r signatures in R is used to compute the cardinality estimate of q, and therefore, the total relative error is at most $r\epsilon$.

The confidence of the estimate also depends on the properties of LSH. Because q_{min} denotes the minimum similarity between q's signature and a signature in R, the probability of finding all signatures in R by contacting k teams at query time is at least $\alpha = 1-(1-q_{min}^{l})^{k}$, where k and l are the parameters of LSH. (Suppose a proxy signature is used such that the minimum similarity between it and a signature in R is p_{min}. Then $\alpha = 1-(1-p_{min}^{l})^{k}$.) Therefore, the net confidence is at least $\alpha \cdot (1-\delta^{'})$.

□

Corollary 1

XGossip can estimate the cardinality of q with a relative error of at most $r\epsilon$ and a probability of at least $(1-\delta)$ in $O(log(\Delta) + log(\frac{1}{\epsilon}) + log(\frac{\alpha}{\alpha+\delta-1}))$ rounds.

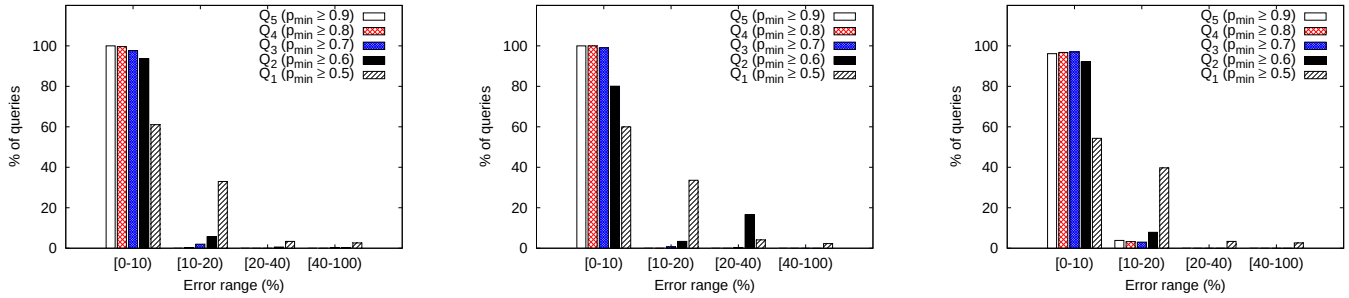
Proof.

To achieve the same confidence as VanillaXGossip, *i.e.*, $(1-\delta)$, the following equations must hold:

$$(1-\delta) = \alpha \cdot (1-\delta^{'})$$
$$\therefore \delta^{'} = \frac{\alpha+\delta-1}{\alpha} \quad (1)$$

Therefore, XGossip achieves a relative error of at most $r\epsilon$ with a probability of at least $(1-\delta)$ in $O(log(\Delta) + log(\frac{1}{\epsilon}) + log(\frac{\alpha}{\alpha+\delta-1}))$ rounds.

□



(a) $n = 500, \Delta = 8, k = 8$ (b) $n = 1000, \Delta = 8, k = 8$ (c) $n = 2000, \Delta = 8, k = 8$

Fig. 26 Accuracy of cardinality estimation achieved by XGossip after 10 rounds

Appendix B

Accuracy of Cardinality Estimation

Figure 26 shows the accuracy obtained by XGossip after 10 rounds for 500, 1000, and 2000 peers.