Scalable Query Dissemination in XPeer

Giovanni Conforti[1,3], Giorgio Ghelli[1], Paolo Manghi[2], and Carlo Sartiani[1]

[1] Dipartimento di Informatica - Università di Pisa - Italy
{confor,ghelli,sartiani}@di.unipi.it
[2] ISTI - CNR
paolo.manghi@isti.cnr.it
[3] Fachbereich Informatik
Universität Dortmund

Abstract. This paper presents XPeer, a data sharing system for massively distributed XML data. XPeer allows users to publish and query heterogeneous information without any significant administration efforts. XPeer tries to dispatch any given query to all and only the potentially relevant peers, exploiting a superpeer network to this aim.

1 Introduction

In today's world, the Internet affirmed as a powerful communication *medium*, allowing people from distant places to share and exchange information, as well as to interact. The Internet offers users many ways to communicate, such as forums, blogs, email, instant messages, voip and conference applications. In a similar way, complex global-scale applications can be built by composing services dispatched by single sites (e.g., web services), so to provide new and sophisticated tools to large and geographically distributed organizations.

While much emphasis is posed on the role of the Internet as a communication medium, this vision is someway limiting. Indeed, the Internet can also be seen as a formidable, massively distributed data repository, containing user-supplied information about near all knowledge fields. This repository is characterized by some ground properties, mostly induced by the behavior of data providers (typically, net-users) and by the characteristics of data being provided. These properties can be described as *heterogeneity*, *autonomy*, and *no administration.*

Heterogeneity Data published on the Internet and, more generally, information exchanged on the Net are by nature heterogeneous. Not only they span over completely unrelated domains (e.g., espresso machines reviews and discussions about LaTeX), but also data referring to the same domain are represented in quite different ways.

With only a few notable exceptions, heterogeneity is definitely not avoidable, and it is common to data published by single net-users as well as to data exported by applications. Even though some application fields have very well defined standard data representation formats (e.g., the logs of SMTP servers can be assumed to be homogeneous), the same does not happen for some very popular applications. Consider, for instance, multimedia players like iTunes, WinAmp, and MediaPlayer: although they manage essentially the same kind of information (usually mp3 or AAC music files), they organize their internal *metadata* database in very different and usually incompatible ways.[1]

When building an Internet-scale application, managing heterogeneity is, to some extent, an inevitable (and disturbing) issue. In particular, heterogeneity plays a significant role in any large scale data sharing system, where users are allowed to share information without a superimposed global schema. For instance, in the above example of multimedia players, one can think of a system allowing users to *transparently* share descriptions, comments, and ratings about the multimedia files they legally own: as each kind of player represents its internal database according to a different schema (differences spread from field names to data nesting), some efforts for managing heterogeneity are necessary for making data available to all users.

Autonomy Strictly related to heterogeneity, a second ground property of the Internet as massively distributed repository is data providers' and data sources *autonomy*. As data providers are (almost) free to publish whatever kind of information they want, they are also free to add new contents, as well as to modify and even drop existing contents they already published. For instance, a blogger has usually the full control on the information she is publishing, hence she can add new posts, modify existing posts, or delete old ones whenever she wants.

Of course, autonomy is someway limited by replication and gossiping phenomena that are intrinsic to the nature of the Internet.

No administration A key factor in the success of the Internet as global communication medium and global-scale repository is that publishing new information (or using even sophisticated Internet services) requires no or little administration efforts by a net-user. For instance, setting up a new blog can be done in a few minutes by a non-expert user and requires only a few and non-technical information about the new blog. Furthermore, popular file sharing systems currently being used for sharing large amounts of video and/or music files do not even require significant administration efforts, as they are able to self-manage their distributed architecture and their configuration.

Our Contribution

We emphasized the role of the Internet as a massively distributed, global-scale data repository. Till now, database technology has not been able to replicate the success of the Internet in building large or global-scale databases. As pointed out in [1], the reasons of this failure are mostly related to common features of current database systems, such as ACID transactions, that are not adequate to a global-scale environment (and they are sometimes even an obstacle).

[1] We are still wondering why simple information can be modeled in so many different ways.

The main objective of this paper is to present XPeer [2], a data sharing system for massively distributed XML data. XPeer allows users to publish and query *heterogeneous* information without any significant administration efforts. To this end, XPeer is based on a p2p architecture that is able to *self-organize* and *self-manage* its own administrative layers without the intervention of a database or system administrator.

Unlike similar projects like PIER [1], XPeer recognizes that heterogeneity is unavoidable and assists the user in *surviving* heterogeneity, i.e., XPeer assumes that data are potentially heterogeneous and tries to dispatch any given query to any potentially relevant peer, while retaining a good degree of selectivity in query dissemination. The technique used is based on automatic extraction of schematic information from peers and use of a *query-to-schema* matching technique to identify the potentially relevant peers.

The contribution of XPeer is twofold. First, XPeer offers very selective query dissemination solutions, that allow the system to deliver a query only to a small superset of the peers containing relevant data, hence reducing both communication and execution costs. Second, its architectural design is scalable and fault-tolerant, and can be easily adapted to more sophisticated data integration techniques based on schema mappings and query reformulation.

Paper Outline

The paper is structured as follows. Section 2 outlines the design choices on which XPeer architecture is based. Section 3 describes the architecture of the system. Section 4 illustrates the solutions used in XPeer for query dissemination, while Section 5 describes the query execution strategy. Section 6 presents some experimental results validating the XPeer approach. Sections 7 and 8, finally, discuss related work and future research.

2 Architectural Choices

XPeer main objective is the management of global-scale XML data repositories, allowing users to share and query heterogeneous data without the need for any administration activity.

These goals require the adoption of architectural solutions enhancing both the scalability and the reliability/availability properties of the system. In particular, the architectural design of XPeer was inspired by the goal of *distribution scalability*, i.e., the system performance should not deteriorate when the number of nodes in the system increases. As the number of peers increases, both the degree of data distribution and the magnitude of the query workload increases, hence the system must cope with an increasing data tracking and query processing load.

In order to design a scalable data management system for heterogeneous XML data, we made some architectural choices that depart from the tradition of centralized and distributed databases, but can also be regarded as heretic wrt usual p2p design criteria.

Query Compilation vs Query Execution A *selective* and efficient query dissemination is critical for the scalability of the XPeer system. To this end, query processing is split in two distinct phases of query compilation and query execution. During query compilation, a *superpeer network* cooperatively tries to identify the peers containing relevant data, so to provide the issuing peer with a quite precise query plan. During query execution, then, the issuing peer deploys the query at the peers identified in the previous step, and coordinates the execution of the distributed query.

Separation between query compilation and query execution is unusual in p2p systems. Much more frequently, queries are disseminated or routed while being executed [3], hence dissemination choices are taken at query run-time. The reason of this departure from the p2p tradition is twofold. First, we see in this separation a way for improving query optimization: once received the compiled access plan from the superpeer network, the issuing peer can apply distributed query rewriting rules [4] so to decrease query execution costs; this in turn leads to a significant enhancement of system scalability.

Second (and most important), effectiveness and selectivity of query dissemination can be significantly improved by exploiting a wider knowledge of the system properties, managed by a specialized network, while the dissemination of queries on the basis of local information may lead to decisions that are only locally promising. We validate this claim in Section 6, where several experimental results show that XPeer performs query dissemination in a quite selective way.

Non-DHT Superpeer Network Supporting an effective and efficient query compilation phase requires the adoption of an adequate architecture for the superpeer network. In our vision, the superpeer network should track the peers connected to the system and store some form of summary of the content of each peer (describing both the structure and the data distribution); indeed, any peer joining the system submits such a summary of its data to the superpeer network, and should refresh it every time a local update of its data leads to a change in its schema or in the data distribution. By aggregating and manipulating these summaries, the superpeer network is able to identify those peers having potentially relevant data, even if their schemas are not homogeneous.

A key point of this vision is that complex schema manipulations are necessary for surviving heterogeneity and identifying query-relevant peers. To lower both communication and processing costs during schema manipulation, schemas and synopses should be stored in a very *localized* way, i.e., without any fragmentation. Hence, we decided to drop architectures based on *Distributed Hash Tables* (DHTs) in favor of a *dynamically hierarchical* organization of the superpeer network, allowing the superpeer network to store schemas without fragmentation. We understand that this choice is somewhat unusual and, to some extent, heretic. Despite the very good scalability properties of DHTs, we believe that DHTs are not adequate to our aims. Indeed, DHTs fragment data in a very fine way, so to maximize load balancing and parallelism; unfortunately, this aspect significantly increases communication and processing costs of any activity with limited parallelism, including schema manipulation.

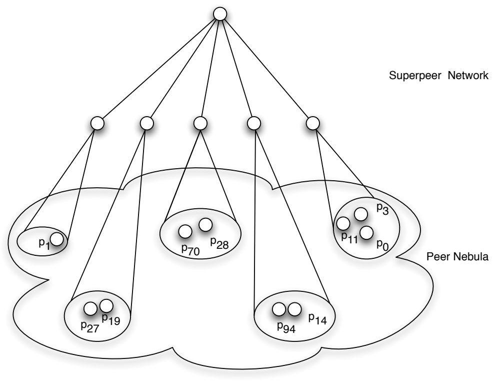


Fig. 1. Peer and superpeer network.

3 System Architecture

The starting point of our architecture is a set of autonomous peers. To make this nebula act as a distributed repository and query system, XPeer builds an *hierarchical tree-shaped* superpeer network over it, as shown in Figure 1.

On the leaves of the superpeer tree, each node (called superpeer) can *spot* a distinct fragment of the nebula, i.e., it keeps track of the schemas and synopses of peers in that fragment, so that the whole nebula is covered by leaf superpeers. The summaries managed by these nodes are then tracked by superpeer nodes in the upper level; in this way, nebula spots are aggregated to allow superpeers to have a wider view of the peer network. This aggregation introduces some form of approximation in the combined summaries, to balance the larger set of information to be managed. This aggregation process is repeated recursively until the root of the tree hierarchy is reached. The superpeer network root, hence, has a high level vision of the whole peer network.

An important issue in this hierarchical process is given by summary (schemas and synopses) representation and aggregation. The current implementation of XPeer uses a DataGuide-like structure (called *treeguide*) [5] for representing peer schemas, which are automatically inferred by the system. Data distribution inside XML simple elements is captured by means of *equi-depth* histograms and Bloom filters: histograms are used for numeric (integer/floats) element only, while Bloom filters come into play for string-valued elements. Histograms, Bloom filters, and treeguides are integrated by endowing each treeguide leaf with the corresponding histogram or Bloom filter, if any.

Based on these representations, summary aggregation is quite straightforward: treeguides are merged, while histograms are combined to lower the space requirements. In particular, buckets are reshaped to preserve equi-depth. As usual, Bloom filters are combined through disjunction.

Hierarchical structures in p2p systems, with only a few exceptions [6], are known to be prone to scalability, reliability, and adaptivity issues. To overcome these problems and make XPeer scale with the size of the peer network, XPeer adopts a technique called *cloning*, which is essentially farm-like replication, with weak synchronization requirements.

The basic idea of cloning is that a superpeer is not a physical entity, but, instead, a *virtual* entity consisting of several nodes cooperating in managing queries and updates. Each of these nodes is called a *clone*, as it replicates the information managed by the virtual superpeer where it resides. It is important to notice that a clone is not a dedicated machine or supercomputer; instead, any peer in the system may become a clone, once it has expressed its willing to perform administrative tasks. A superpeer, hence, is a virtual entity consisting of heterogeneous and geographically distributed clones.

From outside a superpeer, clones are invisible and each of them is able to handle both query compilation requests and summary update requests. Each request to a superpeer is routed to a randomly chosen clone by an enhanced communication and transport layer, which is *clone-aware*, and which is an essential part of the communication infrastructure on top of which XPeer is built.

A relevant issue for the proper work of clones is summary synchronization. As superpeer information is fully replicated among clones, and, as each clone can independently handle summary update requests, a synchronization algorithm among clones, ensuring (some sort of) consistency and scalability, is necessary. We chose to favor scalability on consistency by adopting a synchronization algorithm based on a best-effort linear synchronization scheme, where, every time a clone receives an update request, it forwards the request to all its sibling clones. Retransmission mechanisms based on version numbers and hash signatures are employed for dealing with synchronization message failure.

We can now see how cloning helps both reliability and scalability. For what concerns reliability, a superpeer failure requires that *all* clones fail, which is quite infrequent.

For what concerns scalability, the number of clones in a superpeer is continuously modified to match the load of incoming and outgoing messages. In particular, each clone in a superpeer periodically monitors the length of its message queues: when they exceed a given threshold the clone tries to recruit a new peer to join the superpeer.[2] This process is performed by the system with no human intervention.

4 Query Compilation

Query compilation in XPeer aims at transforming a query in an algebraic expression where each leaf is a *location operator*, identifying a data source that is found in a specific peer. The core step of this process is *distributed compilation*,

[2] Clone failures are managed essentially in the same way, as they impact the messaging load of the remaining clones.

where the superpeer network tries to identify interesting peers on the basis of their summaries, so that only such peers appear in the compiled query.

4.1 Query Language and Query Algebra

XPeer supports a significant fragment of XQuery [7], roughly equivalent to the FLWR core of the language. The XPeer query algebra [4] contains operators for evaluating path and twigs, for filtering variable bindings according to predicates, for building new XML fragments, and for incorporating peer information inside query plans. This algebra is an extension of that described in [8], and is close to other algebras for semistructured and XML data [9].

4.2 Local Compilation

Local compilation translates a query into an *incomplete* algebraic expression, i.e., an algebraic expression without location operators. The query is first translated, by the issuing peer, into an *intermediate* representation based on *query blocks*; on this representation standard rewriting techniques are applied, like, for instance, the factorization of common subexpressions. Finally, the intermediate representation is used for generating the algebraic expression corresponding to the query.

4.3 Distributed Compilation

Once a peer p_x issuing a query q has locally compiled the query into an *incomplete* algebraic expression, it submits the algebraic representation of q to the superpeer network. The superpeer network, hence, fills the holes in the algebraic expression by identifying a set of peers containing potentially relevant data; this process is guided by the summary information stored in the superpeer network.

For an example, we will refer to the query shown below, assuming that the query is satisfied by peers p_1, p_{13}, and p_{17}, and that the superpeer network has the structure shown in Figure 2. This query retrieves the titles of all undergraduate courses from a university courses database.

```
for $c in $db//course,
    $l in $c/level
where $l = "U"
return $c/title
```

The distributed compilation step is then organized in two phases: the *ascending* phase, and the *descending* phase.

Ascending phase p_x submits the query q, by sending its compiled version to its superpeer (sp_1, in this case). Once received the query, this superpeer i) *propagates* the query to its father in the hierarchy, and ii) *matches* the query against schemas and synopses locally hosted, as shown in Figure 2(a). Any positive match (p_1) is then directly communicated to p_x.

As the father of sp_1 (sp_{19}) receives the query from sp_1, it recursively forwards the query to its father (the *root* of the hierarchy, in this case) and matches the query against its schemas and synopses. Unlike the match performed by leaf superpeers, which can directly identify interesting peers, the match at an intermediate level serves the purpose of finding hierarchy subtrees that may contain information about peers with relevant data. In the case of our example, sp_{19} finds that sp_{27} children may comprise interesting peers, hence sp_{19} forwards the query to sp_{27} (see Figure 2(b)). sp_{27} will perform exactly the same actions as sp_1, with the only difference that it will not resend the query to its own father.

Descending phase When the root of the tree hierarchy receives a query from one of its children, the ascending phase for this query ends, and the descending phase starts. The purpose of this phase is to explore the fragment of the hierarchy that has not been touched by the ascending phase. This exploration is performed on a *summary-driven* basis, hence only the subtrees that track potentially relevant peers are actually explored. In the case of our example, the root finds that the subtree rooted by sp_{90} is worth a further exploration, hence it propagates the query to sp_{90} (Figure 2(c)). sp_{90}, in turn, discovers that sp_{99} may have information on interesting peers, while sp_{94} does not match the query; hence, sp_{90} sends the query to sp_{99} (Figure 2(d)). sp_{99}, finally, identifies p_{17} as a potential data supplier, and sends this information directly to p_x.

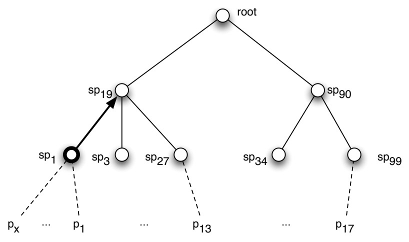
Two important things must be noted about distributed compilation. First, each query must reach the root of the hierarchy, as the root only has some knowledge about the whole peer network. This, in turn, means that the superpeer load increases while moving from the bottom to the top of the hierarchy. This is not a problem, since each superpeer just recruits as many clones as necessary to perform its task.

Second, the superpeer network sends the information about any interesting peer as soon as it is found. This allows for an improvement in the compilation response time.

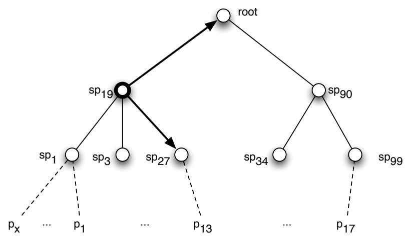
4.4 Query-to-schema Match

A key aspect of the query compilation approach of XPeer is represented by the algorithm being used for *matching* a query against a schema and a set of synopses. This algorithm guides the compilation process and allows the system to identify peers that may satisfy the query as well as to avoid the traversal of fragments of the superpeer network that do not contribute to the compilation result.

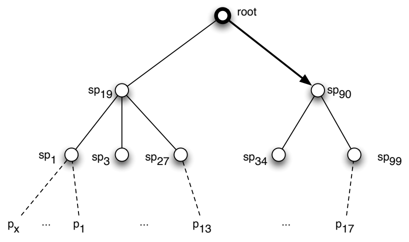
The *query-to-schema match* algorithm works in two main steps. During the first step, the algorithm extracts the twigs (i.e., the “tree patterns”) from a given query (after clause normalization), and “evaluates” the twigs over the schema: if the result of this evaluation is not empty, there may exist some instance of the schema that contains data satisfying the structural requirements of the query. To improve the selectivity of the matching process, the algorithm then compares the



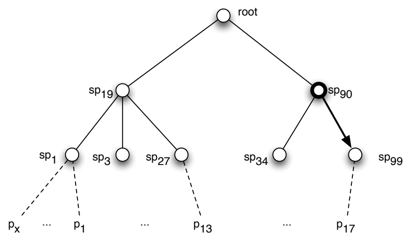
(a) Ascending phase: leaf superpeers



(b) Ascending phase: intermediate superpeers



(c) Descending phase: root



(d) Descending phase: intermediate superpeers

Fig. 2. Distributed compilation.

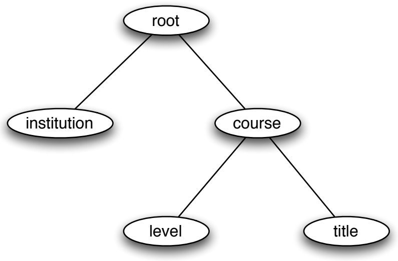


Fig. 3. p_1 treeguide.

predicates specified in the `where` clauses of the query with the statistical synopses associated with the schema, so to discard those peers that give no contribution to the query result. The following Example illustrates the algorithm and the use of synopses.

Example 1. Consider the query of the previous Section, and assume that p_1 data are described by the schema shown in Figure 3.

Suppose that sp_1 checks whether p_1 may contain relevant data. To this purpose, sp_1 interprets p_1 treeguide as a data tree, and executes the binding fragment of the query on it. In our case, sp_1 evaluates `for $c in $db//course, $l in $c/level` on the treeguide, and returns a non-empty tuple where $\$c$ and $\$l$ are bound to the matching schema nodes.

This tuple tells sp_1 that p_1 data satisfy the structural requirements of the query; however, p_1 data may not comprise level elements whose content is "U". As a consequence, a supplementary check is needed to enforce the satisfaction of the `where` clause predicate. To this end, sp_1 accesses the Bloom filter associated to the level node of the schema, and just verifies if the set of strings described by the filter includes "U".

5 Query Execution

Query execution strategy of XPeer is quite simple, as it involves no other nodes but the query issuer and the data sources, and it exploits the simplest communication pattern. The input of this phase is an algebraic expression, where every data source has been identified during the distributed compilation phase. The peer issuing the query decomposes this algebraic expression into subexpressions, called *pipes*, to be delivered to and executed by remote peers, and coordinates remote peer execution. As usual, XPeer tries to minimize network traffic by pushing selections and twig evaluation down the tree, exploiting canonical algebraic rewriting rules [4].

The decomposed query is then formed by several *pipes.* Each given pipe contains an algebraic expression describing the query fragment that a given remote peer must execute. Any operation involving data coming from multiple peers is executed by the issuing peer: the corresponding pipe, which essentially *coordinates* the execution of the whole query, is called the *host* pipe.

After query decomposition, pipes are sent to remote peers and executed. When a remote peer receives a pipe $\mathcal{P}$, it compiles the algebraic expression inside $\mathcal{P}$ into a physical plan and waits for a *start-execution* message.

Execution inside a given pipe follows the iterative model, where new results are requested by means of `next` messages. Interaction among pipes, instead, follows an *asynchronous buffered* iterative model. To avoid deadlocks and to be resilient to node and network failures, interactions are asynchronous and subject to *time-outs*: indeed, the operator used for managing communications with children pipes sends asynchronous `next` requests to the children pipes; if no response is given after a certain period of time, it assumes that the corresponding pipe is blocked or that no more data are available, hence it stops contacting the remote peer.

6 Experimental Results

A crucial feature of XPeer is its ability to dispatch a given query to a relatively small superset of the peers with relevant data, avoiding dissemination policies based on broadcast or flooding. In this Section we will present experimental results validating this claim; these results show that, by relying on the query-to-schema match algorithm, the superpeer network of XPeer is able to discard a significant fraction of peers containing irrelevant data.

6.1 Experimental Setup

Experiments were performed on a 100-node peer network, running on a cluster of Linux machines. We simulate an application where a set of universities publish information on their courses, and the data of each university is structured according to one among three different schemas. To this aim, we started from the three files in the University Courses XML dataset, available at `http://www.cs.washington.edu/ research/xmldatasets/`; these XML documents have been randomly fragmented and each fragment has been assigned to a peer. This setup exemplifies a typical situation where a number of sites publish information with a limited degree of dishomogeneity.

The behavior of the system was controlled and observed through XOrch, a global orchestration tool that we developed inside the XPeer project. XOrch can control the behavior of peers and superpeers by means of *scores*, i.e., scripts represented in a CSP-based language and describing the actions a single peer or superpeer must execute; scores are sent by the *Orc* (i.e., the central monitoring component) to local *proxies*, which execute them by interacting with the attached peers.

6.2 Query Workload

The test query workload is formed by 10 XQuery queries, divided into four classes on the basis of a qualitative selectivity estimation. The first class contains queries with rather selective twigs, while the second class focuses on selective predicates; the third class comprises queries where both twigs and predicates are deemed as selective; the fourth class, finally, contains negative queries only, i.e. queries with no answer.

We assembled our test workload with the aim of mimicking "real life" workloads, hence the test queries contain both / and // operations, union paths, and multiple predicates connected by *and*/*or* logical connectors.

More information about the test workload as well as supplementary test results can be found at `http://datatop2.di.unipi.it/experiments`.

6.3 Experiments

In our experiments, we measured *precision* and *recall* of compilation for positive queries, as well as the absolute compilation error for negative queries, i.e., the difference between the number of peers deemed as interesting by the compilation process and the number of peers actually hosting relevant data; we performed these evaluations for four different compilation configurations. In the first configuration, we did not use synopses (Bloom filters and histograms) at all, hence limiting the query-to-schema match to a purely structural one; in the remaining configurations we used synopses of increasing dimensions: 96-bit, 480-bit, and 4800-bit Bloom filters, and histograms containing up to 15, 30, and 70 values, respectively.

Results of our experiments are shown in Figures 4 and 5. Since both schemas and synopses are upper approximations of the actual peer data, the *recall* should always be 100% when data is neither updated nor lost; the experiments confirm this, hence we only plot *precision*. As it can be noted, compilation is quite precise even in the absence of synopses; only query Q6 showed poor results, probably related to some issues concerning partial match string predicate.

Synopses really come into play on negative queries; while the no-synopses configuration generates significant compilation errors, the use of synopses allows the system to correctly recognize negative queries.

7 Related Works

Several projects focus on the problem of evaluating structured and complex queries in p2p systems. Among these projects, Pier [1] is definitely the closest system to XPeer. Pier is based on the use of a DHT, where relational, homogeneous data are cached. Pier queries are executed by adapting techniques from parallel databases to a DHT-based storage, hence exploiting the inner parallelism of DHTs. As Pier is focused on homogeneous data and parallelizable queries, it should be regarded as complementary wrt XPeer, which, instead, focuses on heterogeneous, hierarchical data and mostly hierarchical query operators. The only

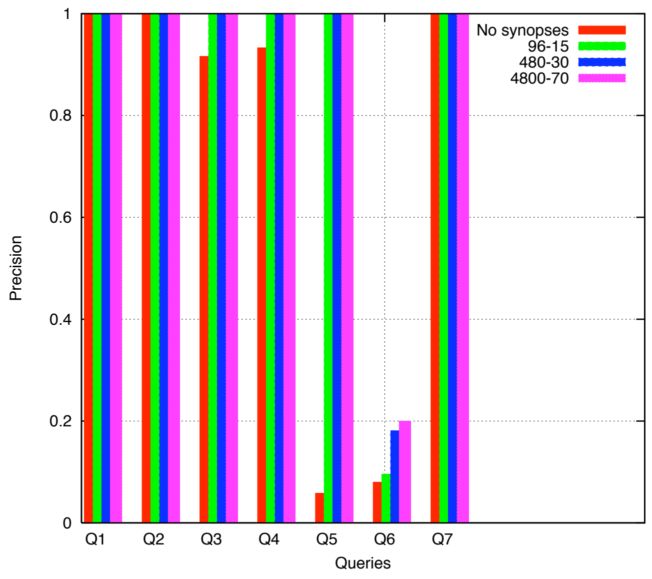


Fig. 4. Precision of compilation.

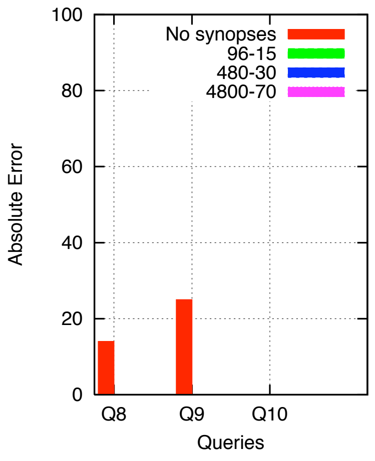


Fig. 5. Absolute compilation error on negative queries.

significant limitation of Pier wrt XPeer is query dissemination, which, in most cases, requires a broadcast on the whole peer network.

Heterogeneity management is, instead, the focus of the Piazza project [10]. Piazza is essentially a p2p-structured data integration system for XML data, where the data integration task is dispersed across the whole network. Peers (which may represent data sources) are connected through schema mappings, and query processing is based on a *flooding* algorithm that broadcasts queries among peers. Piazza, hence, suffers from severe scalability and query dissemination problems, that greatly limit its applicability.

A different way of supporting heterogeneity is adopted in the system described in [3], where a *coordinator-free* architecture for distributed XML query processing is presented. By assuming that peers contain semantically related data, the system uses a multi-hierarchic organization of the domain space to route queries (in the form of *mutant query plans*, i.e., query plans containing materialized data too). This approach does not seem adequate when data are semantically heterogeneous, and it makes query dissemination quite expensive.

An interesting hybrid between structured and unstructured p2p systems is represented by KadoP [11]. In KadoP a DHT is used for storing a distributed full-text index about documents and web services; this index, together with ontologies, is used during query processing for locating interesting data and services. Resource location, hence, requires the system to perform several key lookups in the DHT index, with a significant messaging cost, which depends on the dimension of the query, as well as on the data and services involved by the query.

8 Conclusions

The wide diffusion of the Internet has greatly increased the number of data-sources publicly available as well as the amount of available data. As a consequence, a need for query systems able to match the scalability properties of the Internet emerged. This paper presented XPeer, a query system for massively distributed and heterogeneous XML data. XPeer allows users to perform structured queries on globally distributed data, without the hassles of any administration activity and while preserving full control on their own data. We described the principles on which XPeer is based, as well as its architectural paradigm, relying on a tree-shaped superpeer network; we also illustrated how a virtualization technique called *cloning* can be used for ensuring both robustness and scalability of the architecture.

We showed in detail the techniques used for disseminating queries inside the network: indeed, we illustrated how a query-to-schema matching technique can be exploited for making query dissemination smart and precise, as confirmed by the experimental results we provided.

In the near future, we plan to perform extensive experiments about compilation, so to evaluate the selectivity of query dissemination in the presence of massive network or node failures. We also plan to enhance the class of supported predicates at compilation time, as well as to study the behavior of the system on very large networks.

References

1. Huebsch, R., Chun, B.N., Hellerstein, J.M., Loo, B.T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., Yumerefendi, A.R.: The architecture of pier: an internet-scale query processor. In: CIDR. (2005) 28–43
2. Sartiani, C., Ghelli, G., Manghi, P., Conforti, G.: XPeer: A self-organizing XML P2P database system. In: Proceedings of the First EDBT Workshop on P2P and Databases (P2P&DB 2004), 2004. (2004)
3. Papadimos, V., Maier, D., Tufte, K.: Distributed Query Processing and Catalogs for Peer-to-Peer Systems. In: CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003. (2003)
4. Sartiani, C.: A query algebra for xml p2p databases. In: Proceedings of the 11^{th} International Workshop on Foundations of Models and Languages for Data and Objects (FMLDO). In conjunction with the 10th Int. Conference on Extending Database Technology (EDBT 2006). (2006)
5. Goldman, R., Widom, J.: DataGuides: Enabling query formulation and optimization in semistructured databases. In: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, Morgan Kaufmann (1997) 436–445
6. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: VLDB. (2005) 661–672
7. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium (2005) W3C Candidate Recommendation.
8. Sartiani, C., Albano, A.: Yet Another Query Algebra For XML Data. In Nascimento, M.A., Özsu, M.T., Zaïane, O., eds.: Proceedings of the 6th International Database Engineering and Applications Symposium (IDEAS 2002), Edmonton, Canada, July 17-19, 2002. (2002)
9. Paparizos, S., Jagadish, H.V.: Pattern tree algebras: Sets or sequences? In: VLDB. (2005) 349–360
10. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: data management infrastructure for semantic web applications. In: Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003, ACM (2003) 556–567
11. Abiteboul, S., Manolescu, I., Preda, N.: Constructing and querying peer-to-peer warehouses of xml resources. In: ICDE. (2005) 1122–1123