XPeer: A Self-organizing XML P2P Database System*

Carlo Sartiani, Paolo Manghi, Giorgio Ghelli, and Giovanni Conforti

Dipartimento di Informatica - Università di Pisa
Via F. Buonarroti 2 - 56127 - Pisa - Italy
{sartiani, manghi, ghelli, confor}@di.unipi.it

Abstract. This paper describes XPeer, a *zero-administration* system for sharing and querying XML data. The system allows users to share XML data without significant human intervention, and to pose XQuery FLWR queries against them.The proposed system can be used in any application field, being a general purpose XML p2p DBMS, even though its main application is the management of resource descriptions in *GRID* environments.

1 Introduction

The last few years have seen the emerging of the *peer-to-peer* (p2p) computational paradigm. This model extends existing ideas about distributed and client-server computing, blurring the distinction between clients and servers. Systems conforming to this paradigm appear as *open-ended* and dynamic networks of peers willing to share computational resources, ranging from CPU cycles to local data, and even to algorithms (for instance, knowledge discovery algorithms).

The p2p paradigm was recently adopted in the database community to overcome the limitations of distributed database systems, namely the static topology and the heavy administration work, and to exploit the dissemination of data sources over the Internet.

One key factor in the success of p2p systems, mostly in the field of content sharing, is their easy administration. On the contrary, existing distributed database systems require heavy administration efforts, both in the design phase and at run-time: indeed, these systems are based on the presence of global and local schemas, together with their mappings, whose definition and maintenance are a duty of the DBA. Nevertheless, existing p2p systems for XML databases still require significant administration tasks: in Piazza [1], for instance, human intervention is still necessary for defining schema mappings between peers, which implies significant efforts for the DBA, and decreases the dynamicity of the system.

* This work was partly funded by the FIRB GRID.IT project.

Our Contribution This paper describes a *zero-administration* p2p system for sharing and querying XML data (XPeer). The system allows users to share XML data and to pose XQuery FLWR queries against them without any significant human intervention (the user still has to write her own queries). The system, based on a hybrid p2p architecture, *self-organizes* its superpeer network, and allows for arbitrary changes in the network topology.

Paper Outline The paper is organized as follows. Section 2 describes some important issues that emerge in the management of p2p XML databases. Section 3, then, presents an overview of the system, while Section 4 illustrates the system architecture in more detail. Section 5, next, outlines the techniques used in XPeer for processing queries. Section 6, then, discusses some related works. In Section 7, finally, we draw our conclusions and describe some future work.

2 Issues in P2P XML Data Management

The problem of managing p2p XML databases is quite complex. The source of most issues is the dynamic nature of these systems, where both data and topology may suddenly change. Hence, a closer look at these aspects is necessary.

Changing topology Peer-to-peer systems are usually described as *open-ended* networks of peers willing to share resources. Peers are autonomous, in the sense that they are free to choose the data to contribute to the system, to manage local data without external constraints, and to connect and disconnect at any time. As a consequence, the system is formed by a collection of nodes $\mathcal{S} = \{p_1, \ldots, p_n\}$ that can evolve over time.

Topology changes mostly affect the indexing structures used for routing queries. For instance, if a node p_i containing data (let's say a set of XML nodes s) relevant for a query q suddenly becomes unreachable, then any index entry associating p_i to s should be updated to avoid unnecessary messages, or, in the worst case, *run-time* problems.

Local updates Peer autonomy implies that peers have the right to update their data, even if shared, at any time. In particular, peers can perform both *value* and *schema* changing updates (unlike in relational databases, the loose structure of XML data blurs the distinction between value and schema updates).

Value and schema updates influence query mediation and query routing since sudden data changes may invalidate existing query plans or routing structures, hence imposing potentially expensive updates of distributed index structures. Moreover, most *schema-driven* data management approaches (see [1]) are severely affected by local updates, hence requiring human intervention for adapting the system to the new data.

3 XPeer Overview

In this Section we provide a quick overview of the architecture of XPeer, as well as of its data model and query language.

3.1 Basics

XPeer is an XML p2p database system, which manages data dispersed over an *open-ended* network of autonomous peers. In XPeer no constraints are imposed over exported data, i.e., a peer may export whatever kind of data, provided that data are encoded in the XML format, and described by a schema, and it may freely update its local data; moreover, nodes can join and leave the system at any time, so the system has a dynamic topology. Exported data are integrated in a *blind* way, i.e., no *global schema* is defined: this solution allows for a significant decrease in the administration load of the system. Of course, this fundamental choice restricts the applicability of the approach to situations where schema mapping can be avoided, or can be performed out of the p2p system (i.e., by a local schema adapter). We believe the choice is perfectly reasonable in the application field we are targeting first (resource description).

XPeer adopts a hybrid p2p architecture [2], where peer nodes may also perform administrative tasks. System nodes, hence, may act both as peers and as *superpeers*.

Databases hosted by XPeer can be queried with a proper subset of XQuery. Due to the complexity of the system, and, in particular, to the changing topology of the system, no guarantee about the completeness of query results can be provided; this, in turn, implies that queries containing set conditions or aggregation functions may be incorrectly answered [3].

XPeer is a general purpose XML p2p database system, so it can be used in any application field. Still, its main application is the management of resource descriptions in a *GRID-like* environment: in particular, XPeer should form the basic infrastructure for extending (and, eventually, replacing) the *LDAP-based* resource discovery layer of existing GRID systems.

3.2 Data model and Query Language

Data in the system are represented as in most XML database systems, i.e., as unordered forests of node-labeled trees. Each tree is augmented with the indication of the hosting peer (*location* in the following) as well as with a freshness parameter fr, which indicates when the last update on the tree was performed ($\perp$ indicates that the freshness is undefined, and it is necessary to ensure that the model is closed). To support freshness parameters, the data model has a universal constant τ, which denotes the current global time in the system: since query results are assumed to be incomplete, the assumption of the existence of a global time is feasible.

The query language of choice is the FLWR subset of XQuery [4] without universally quantified predicates and sorting operations (the orderby clause). The

choice of the FLWR core of XQuery distinguishes XPeer from most existing p2p systems, which are limited to simple *key-lookup* queries, or to linear path queries, and which require significant modifications to support *full* database queries [5].

4 XPeer Architecture

XPeer is a *hybrid* p2p system composed by a dynamic set $\mathcal{S} = \{p_1, \ldots, p_n\}$ of autonomous peers, which share data and execute global queries on the database. Some nodes in $\mathcal{S}$ (in most cases, those with adequate computational power and/or network bandwidth) perform administration tasks too: these nodes, called *superpeers*, form a set $\mathcal{SP} \subseteq \mathcal{S}$. Peers become superpeers on a voluntary basis, and retain their peer role. We favor a hybrid p2p architecture wrt a hierarchical one (e.g., the GRID GRIS/GIIS system) since it offers more robustness to failures and it can adapt more easily to network changes.

4.1 Peer Network

Peers share XML data and execute queries on top of these data. Peers export a description of the data being shared in the form of a tree-shaped DataGuide [6], called *tree-guide*, which is automatically inferred from the data by means of a tree search algorithm. Leaf nodes in the schema are endowed with statistical information about value ranges, to allow the system to better identify relevant data sources during query compilation. The following Example shows a sample XML document and its tree-guide.

Example 1. Consider the following document, hosted by a peer p_1, describing buildings in a real-estate market database.

```
<market>
   <buildings>
      <building>
         <desc> Marvelous luxury house in the Hamptons </desc>
         <location> Hamptons </location>
         <price> 1600000 </price>
      </building>
      <building>
         <desc> Very nice flat in the Upper East Side </desc>
         <location> Upper East Side, Manhattan </location>
         <price> 1350000 </price>
         <type> comdo </type>
      </building>
      <building>
         <desc> Elegant luxury house in the countryside </desc>
         <location> Greensboro </location>
         <price> 1700000 </price>
      </building>
   </buildings>
</market>
```

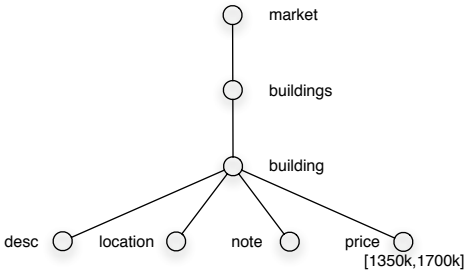


Fig. 1. A sample tree-guide.

The corresponding tree-guide contains each distinct path in the document, endowed with statistical information about value ranges (e.g., the range 1350000−1700000 for price elements), as shown in Figure 1.

Peers are logically organized into clusters of nodes, where each cluster contains one superpeer, which is in charge with the management of the cluster: the compilation of user queries and the management of peer information. Peer clustering allows the system to decrease the efforts required for compiling queries. To this aim, clusters are formed, whenever it is possible, on a *schema-similarity* basis, i.e., peers exporting data with similar schemas are clustered together (the system still works even if nodes in the same cluster have very different schemas).

Inside any cluster, some peer may (partially or totally) replicate the content of other peers in the cluster. Replicas are built to balance the workload in the cluster and to exploit peers with huge computational resources, and are valid up to a given time. The replication process, as many other processes in XPeer, happens on a voluntary basis.

4.2 SuperPeer Network

Superpeers have the duties of tracking topology changes, managing schema information, and compiling user queries. Superpeers are organized to form a tree, where each node hosts schema information about its children; superpeers having the same father form a *group* (which is very close to a peer cluster). The resulting logical topology is shown in Figure 2.

Superpeers host two kinds of schema information about their children: the list of the schemas of their children (the *schema list*); and the union of these schemas (the *superpeer schema*). The schema list is used during query compilation for identifying relevant data sources, or superpeers whose descendants can contain relevant data; the superpeer schema, instead, is passed to the father as schema of the superpeer, and it is built without any schema integration activity, so that no human assistance is required. Since tree-guides may have, in the worst case, the same size as the documents they are representing, the schema of the root super-peer may have, in the worst scenario, the same dimension as the whole

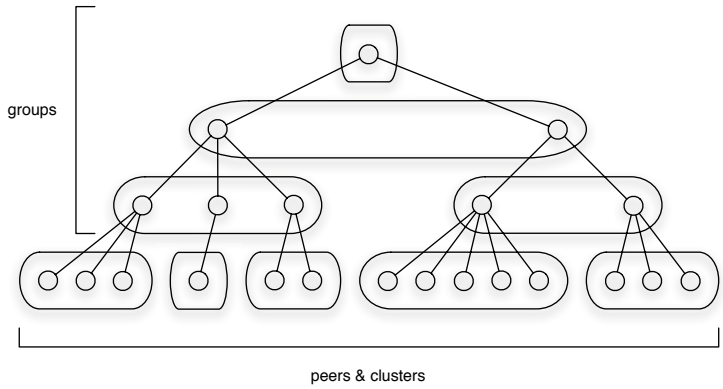


Fig. 2. Overall logical system architecture.

p2p database. However, as shown in [6], this may happen only when a) each local database is formed by non-overlapping rooted paths, and b) there are no local databases having some common rooted path; this scenario is so infrequent that we can safely use tree-guides as document schemas. The following Example shows a sample superpeer schema.

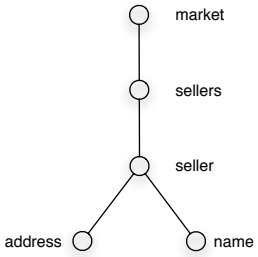
Example 2. Consider the following XML document, hosted by a peer p_2, describing seller information in the real-estate market.

```
<market>
   <sellers>
      <seller>
         <name> Patrick Bateman </name>
         <address> 25, Park Avenue </address>
         <phone> ... </phone>
      </seller>
      <seller>
         <name> Tim Price </name>
      </seller>
   </sellers>
</market>
```

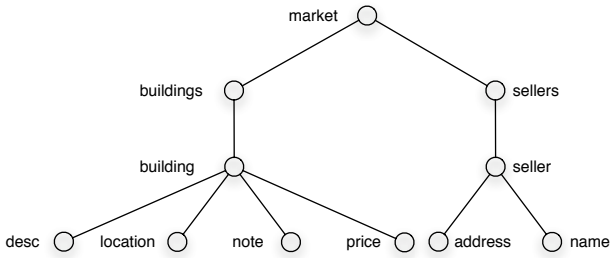
This document can be represented by the tree-guide shown in Figure 3(a). Assuming that both peers p_1 (see Example 1) and p_2 have the same superpeer sp, then the superpeer schema of sp is depicted in Figure 3(b).

4.3 Network Evolution

The topology of the network can evolve over time. To adapt the organization of the superpeer hierarchy to changes in the network, superpeers may split clusters and groups, and may ask for new superpeers. In particular, when the workload



(a) Another sample tree-guide.



(b) A superpeer schema.

Fig. 3. Another tree-guide and a super-peer schema.

for a given superpeer *sp* becomes too hard, *sp* first tries to relocate some of its children in other clusters/groups (*network balancing*); if the problem persists, *sp* then asks the system for new superpeers, and delegates them part of its workload (*network extension*); if the workload is still too heavy, *sp* can finally disconnect some of its children (*peer de-gnoming*). On the other hand, when the workload for a given superpeer *sp* becomes too light, *sp* may decide to import some children from busy superpeers, or it may decide to relocate its children to another superpeer, and then to exit the superpeer network (*network contraction*).

5 XPeer Query Processing

XPeer supports the FLWR core of XQuery, the standard query language for XML data being developed by W3C [4]. Since data are usually dispersed among many peers, XPeer does not support the preservation of document order in query results.

FLWR queries are translated into algebraic expressions, and are executed on the system by relying on *data-integration-like* techniques. To speed up query execution and to decrease peer and superpeer workload, the system exploits mechanisms for replicating peer content, and for caching query plans and query results; these mechanisms can be ignored on an explicit request by the user.

5.1 XPeer Query Algebra

The query algebra of XPeer, further described in Appendix A and in [7], is an evolution of the query algebra for centralized XML data described in [8]. The query algebra consists of three classes of operators. The first class contains operators that navigate unordered forests of node-labeled trees, binding nodes to variables, and that build new XML trees from existing variables bindings (*path* and *return*); the second class, instead, contains operators for manipulating tuples of variable bindings, as in standard OO query algebras [9] [10] (σ, π, $\bowtie$, $DJoin$, etc); the third class, finally, is formed by operators for managing *locations* (the algebraic counterpart of peers), and, in particular, for uniting their content ($LocUnion$) and for inserting replication constraints into query plans ($Choice$). Since location choices are guarded by temporal parameters, the query algebra data model has been enriched with the universal constant τ, describing the system global time, and with time labels for locations.

The following example shows a sample algebraic expression.

Example 3. Consider the following XQuery query:

```
for $b in input()//building,
    $d in $b/desc,
    $p in $b/price
return <entry> {$d, $p} </entry>
```

This query returns the description and the price of each building in the *real-estate* market database. Assume that the database ($db1$) is formed by data

dispersed over locations loc_1, loc_{11}, loc_{13}, and loc_{17}, and that $loc_{11}(db1)$ is replicated at loc_{17} till time δ; furthermore, assume that the query was submitted at time τ' so that $\tau' < \delta$. Then, the query can be expressed by the following algebraic expression:

$$
\begin{array}{l}
return_{entry[\nu\$d,\nu\$p]}(\\
\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\
\qquad (loc_1 \bullet (loc_{11} \mid^{\delta}_{db1} loc_{17}) \bullet loc_{13} \bullet loc_{17})^{1}(db1)))
\end{array}
$$

5.2 Query Compilation

Query compilation is performed in two phases. In the first step, a query is submitted by the user to a peer p_i. p_i translates the query into a triple $Q = (q, \tau', \delta_{\tau'})$, where q is a *location-free* algebraic expression, i.e., an algebraic expression with "holes" (called *spots*) in place of locations, τ' is the query submission time, and $\delta_{\tau'}$ is a user-defined freshness parameter; in particular, $\delta_{\tau'}$ indicates that the system may use replicas and caches synced after time $\tau' - \delta_{\tau'}$, and allows the user to specify freshness and quality requirements for the result of the query (e.g., $\delta_{\tau'} = 0$ means that only up-to-date caches and replicas can be used, while $\delta_{\tau'} = \infty$ means that any existing cache or replica can be used).

In the second phase, p_i sends the query Q to the superpeer network, via the superpeer of its own cluster, for the compilation of a *location assignment* ρ, i.e., a function assigning unions ($\bullet$) and choices ($|$) of locations to location spots. This compilation is performed in a hierarchical way by matching the twigs of q with schema information, and by traversing the superpeer hierarchy till any interesting location has been detected. In particular, the super-peer responsible for the cluster of p_i matches the twigs of the query with the schemas of its children peers, hence finding all relevant locations in the cluster; then, the super-peer sends the query to the super-peer responsible for its group, which in turn matches the query twigs against the schemas of its children, and resend the query to clusters that may contain relevant data. The query is also propagated up in the hierarchy to find all relevant locations. The query compilation process, hence, requires the system to propagate the query till the root of the super-peer network, but still limits the exploration of the network to a fraction of the hierarchy.

Once the location assignment ρ is computed, ρ is passed to the issuing peer p_i for query execution; by making p_i responsible for the execution of its query, the system minimizes the load of the superpeer network.

The following Example shows how query compilation is performed.

Example 4. Consider the query of Example 3, and assume that the system has the structure shown in Figure 4, where p_1 and p_2 contain the documents described in Examples 1 and 2 respectively, while p_3, p_4, and p_5 contain data about loans and mortgages.

Suppose that a user submits the query of Example 3 at peer p_2. p_2 builds the following *location-free* algebraic expression

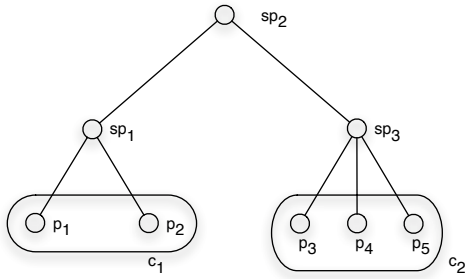


Fig. 4. Another system topology.

$$return_{entry[\nu\$d,\nu\$p]}($$
$$path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\mathbf{spot_1}))$$

and then sends this expression to sp_1. sp_1 matches the query twig against the list of tree-guides of its peers, hence finding p_1 relevant for the query, and then propagates the query to sp_2; sp_2, in turn, matches the query twig over its schema list, hence excluding the descendants of sp_3 from the query plan. The final query is shown below.

$$return_{entry[\nu\$d,\nu\$p]}($$
$$path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_1))$$

5.3 Query Execution

Once the issuing peer p_i has received the location assignment ρ for a query Q, it applies common algebraic rewriting to the fully specified algebraic expression, such as selection push-down and distribution of unions, and then starts executing the query, which is split into *single-location* sub-queries that are sent to the corresponding peers; p_i waits for query results, and then executes operations, such as joins, involving data coming from multiple sources. Query subexpressions are locally optimized and executed by system peers, hence allowing each peer to choose the best execution strategy for any given algebraic expression.

Query decomposition is performed by exploiting an algorithm close to that of YAT [11]: the algorithm just browses the algebraic tree in the search of maximal single-location subexpressions, which correspond to peer sub-queries.

6 Related Works

In this Section we briefly review existing works on XML p2p databases.

Maier's System In [12] authors describe a *coordinator-free* architecture for distributed XML query processing in the context of p2p systems. The proposed architecture is tailored for the needs of *bio-informatics* applications, but it can be safely adopted in other p2p applications.

The proposed architecture is based on two key ideas: *mutant query plans* (MQP) [13], and *multi-hierarchic* namespaces. An MQP is a logical query plan, where leaf nodes may consist of URN/URL references, or of materialized XML data. MQPs are themselves serialized as XML elements, and are exchanged among the nodes of the system. When a node S receives an MQP P, S can resolve URN references, materialize URL references, evaluate MQP sub-plans, re-optimize MQP sub-plans, or just route P to another server; when P is reduced to XML code only, it is sent to the *target* node, i.e., the node originating the query. As a consequence, an MQP traverses the system, carrying partial results and unevaluated sub-plans, until it is fully evaluated, i.e., it becomes a constant XML fragment.

MQPs are routed in the system according to information derived from multi-hierarchic namespaces. Indeed, authors assume that data contributed by peers are semantically connected, i.e., they are part of the same namespace. A namespace is formed by several category hierarchies, e.g., a hierarchy for geographical information and one for item features in a garage-sale p2p application. As in OLAP systems, single hierarchies form the dimensions of an hypercube; contiguous portions of the hypercube are called *interest areas*. Interest areas are generated to match the data provided by peers, i.e., a peer P can provide data belonging to a single interest area.

The proposed system heavily exploits the semantic homogeneity of the data supplied by peers, hence it appears not adequate when data are semantically heterogeneous. Moreover, while MQPs allow the system to avoid centralization points, the correctness of their routing algorithms in the presence of network problems is far from being clear.

DBGlobe In [14] authors describe DBGlobe, a p2p system for global computing. The key points of the project are the management of mobile peers, which may relocate over time, the use of services for dealing with heterogeneity and mismatching problems, as well as the use of Active XML [15] as the paradigm for service invocation/execution and data exchange.

The architecture of the DBGlobe system is a 3-level hierarchical architecture. The first level contains the system peers (called Primary Mobile Objects). The second level, instead, contains Cell Administration Servers, which have the duty to manage the underlying network. The network itself is divided into contiguous cells, just like the network for mobile phones, each cell being managed by a CAS.

On top of the first two levels of DBGlobe, which form the infrastructure of the system, the third level contains the components devoted to offer *application-oriented* services, ranging from the management of *communities* (i.e., groups of peers whose contents and services are semantically correlated) to the collection of user query results when a given user goes offline. As for network cells, communities are managed by Community Administration Servers (CoaSs).

The location of nodes containing interesting services is performed by relying on an index structure called *Multi-Level Bloom Filter*.

Although very interesting, the DBGlobe system requires heavy administration activities for its build-up and its maintenance; moreover, it is not clear how the system can react to failures in its administrative layers.

Piazza In [1] authors give an overview of Piazza, a peer data management system for XML data. The Piazza project focuses on the use of schemata, and, in particular, on the definition of schema integration and mapping techniques for p2p systems.

The architecture of Piazza is basically a hierarchical p2p architecture, where peers are fully autonomous, and may contribute data with schemas, while a central node hosts an index structure structure for query routing and performs query reformulation. Each peer has a schema, the *peer schema*, which describes how the given peer views the data offered by the system; while the Piazza approach is based on the assumption that all peers share similar views of the world, these visions are usually different, so the need for peer schema reconciliation techniques emerges. Moreover, the peer schema is somehow independent from the schema of the data the peer may store, so a second class of mappings is required.

Peers contributing data also have a second schema, the *storage schema*, which describes the structure of the data; both the peer schema and the storage schema conform to the XML Schema specification [16].

Peer schemas represent the peer vision of the world. As a consequence, each query submitted by a given peer P is posed against the peer schema of P, and it must be reformulated to work against the storage schema of the relevant peers in the system. To this purpose, Piazza supports two kinds of schema mappings: *peer descriptions*, which relate two or more peer schemas, and *storage descriptions*, which map the data stored at one peer into the peer's view of the world.

Unlike common integration systems, no centralized mediated schema exists, query reformulation being executed by solely using peer descriptions and schema descriptions. Moreover, the set of peer descriptions of a given system is *sparse*, i.e., a peer schema is mapped into a few other schemas, in order to decrease the efforts required for schema integration, and to simplify the extension of the system with new peers.

Peer and schema descriptions are expressed by means of inclusion relations with *virtual* documents (*views*). This choice allows the system to exploit techniques for query reformulation over views, and, in particular, GAV and LAV techniques.

Peer and schema descriptions are stored in a centralized node, which also hosts a data indexing structure. This index is used for filtering the list of relevant peer identified during query reformulation, hence it allows the system to decrease the number of false positives in a query plan. The index structure hosts data summaries as well as other information about data and peers.

While very promising, the Piazza approach still requires human intervention for the definition of schema mappings.

7 Conclusions and Future Work

This paper describes the architecture of XPeer, a p2p XML data management system. The system supports the FLWR core of XQuery, and allows the user to execute queries on the global database; global queries are translated into algebraic expressions, which are then decomposed and sent to the relevant peers.

The architecture of the system is *self-organizing*, in that the superpeer network can adapt its structure to changes in the system network topology and in the query workload. Furthermore, the system requires no human intervention for its administration, hence being a *zero-administration* DBMS.

XPeer is a general purpose XML p2p database system, so it can be used in any application field. Still, its main application is the management of resource descriptions in a *GRID-like* environment: in particular, XPeer should form the basic infrastructure for extending (and, eventually, replacing) the *LDAP-based* resource discovery layer of existing GRID systems.

XPeer is currently being implemented on top of an existing persistent XML query engine, so at this time no remarks about its performance and scalability properties can be done. In addition to implementing XPeer, we are currently investigating the problem of correctness of query results in the presence of incomplete query plans.

References

1. 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
2. Yang, B., Garcia-Molina, H.: Designing a Super-peer Network. In: Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, 5-8 March 2003, IEEE Computer Society (2003)
3. Sartiani, C.: On the Correctness of Query Results in XML P2P Databases (2003) Manuscript draft. Available at `http://www.di.unipi.it/~sartiani/papers/rocketman.pdf`.
4. 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 (2003) W3C Working Draft.
5. Harren, M., Hellerstein, J. M., Huebsch, R., Thau Loo, B., Shenker, S., Stoica, I.: Complex Queries in DHT-based Peer-to-Peer Networks. In: IPTPS 2002, pages 242-259
6. 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
7. Sartiani, C.: A Query Algebra for XML P2P Databases (2003) Manuscript draft. Available at `http://www.di.unipi.it/~sartiani/papers/eve.pdf`.
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. A. M. Alashqur, Stanley Y. W. Su, and Herman Lam. Oql: A query language for manipulating object-oriented databases. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 433–442. Morgan Kaufmann, 1989.
10. Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bases. Technical report, University of Karlsruhe, 1994.
11. Siméon, J.: Intégration de sources de données hétérogènes. PhD thesis, Université Paris XI (1999)
12. 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)
13. Papadimos, V., Maier, D.: Mutant query plans. Information & Software Technology **44** (2002) 197–206
14. Pitoura, E., Abiteboul, S., Pfoser, D., Samaras, G., Vazirgiannis, M.: DBGlobe: a service-oriented P2P system for global computing. Sigmod Record **32** (2003) 77–82
15. Abiteboul, S., Benjelloun, O., Manolescu, I., Milo, T., Weber, R.: Active XML: Peer-to-Peer Data and Web Services Integration. In: 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, August 20-23, 2002, Proceedings, Morgan Kaufmann (2002) 1087–1090
16. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures. Technical report, World Wide Web Consortium (2002) W3C Recommendation.

A Query Algebra

A.1 Data Model and Term Language

$$
\begin{array}{l}
t ::= t_1, \ldots, t_n \quad | \quad n[t] \quad | \quad n \\
n ::= (oid, loc, fr)label \\
loc : (dbname \rightarrow t, (dbname, loc) \rightarrow t) \\
\text{where } label \in \Sigma^*,\ fr \in \mathbb{N} \cup \{\perp\}, \text{ and} \\
loc^1 \text{ and } loc^2 \text{ are partial functions.}
\end{array}
$$

Node Functions

$$
\begin{array}{l}
label(n) = label \\
oid(n) = oid \\
loc(n) = loc \\
freshness(n) = fr
\end{array}
$$

A.2 *Env* and Tuples Operations

Four basic operations are defined on *Env* structures and tuples.

1. $t.A = t_j$ where $A = label_j$ (where t is a tuple) (field extraction)

2. $t.\vec{A} = \{t_{i1}, \ldots, t_{ip}\}$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$ (repeated field extraction)
3. $t \downarrow \vec{A} = [label_{i1} : t_{i1}, \ldots, label_{ip} : t_{ip}]$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$
4. $\bullet$, a concatenation operator between tuples (known as *tup_concat* in other algebras).

A.3 Support Operators

1. $e[x] = \{[x : t] \quad | \quad t \in e\}$
2. $child(t) =$
   (a) if $t = v_B$, then $child(t) = \{\}$
   (b) if $t = (oid)\_[t_1, \ldots, t_n]$, then $child(t) = \{t_i \quad | \quad i \in 1, \ldots, n\}$
3. $descendant(t) = child(t) \cup \bigcup_{t_i \in child(t)} descendant(t_i)$
4. $self(t) = \{t_1, \ldots, t_n \quad | \quad t = t_1, \ldots, t_n\}$
5. $self - descendant(t) = self(t) \cup descendant(t)$
6. $nav(op)(label)(t) =$
   (a) if $op = (\_)$, then $nav(op)(label)(t) = \{t_i \quad | \quad t = t_1, \ldots, t_n \wedge label(t_i) = label\}$
   (b) if $op = (/)$, then $nav(op)(label)(t) = \{t_j^{'} \quad | \quad t = t_1, \ldots, t_n \wedge \exists i \in 1, \ldots, n : t_j^{'} \in child(t_i) \wedge label(t_j^{'}) = label\}$
   (c) if $op = (//)$, then $nav(op)(label)(t) = \{t_j^{'} \quad | \quad t = t_1, \ldots, t_n, \exists i \in 1, \ldots, n : t_j^{'} \in self - descendant(t_i) \wedge label(t_j^{'}) = label\}$

A.4 Basic Operators

Map $\chi_f(e) = \{f(t) \quad | \quad t \in e\}$

TupJoin $e_1 \bowtie_{Pred} e_2 = \{t_1 \bullet t_2 \quad | \quad t_1 \in e_1 \wedge t_2 \in e_2 \wedge Pred(t_1, t_2)\}$

DJoin $e_1 < e_2 >= \{y \bullet x \quad | \quad y \in e_1, x \in e_2(y)\}$

Selection $\sigma_{Pred}(e) = \{t \quad | \quad t \in e, Pred(t)\}$

Projection $\pi_{\vec{A}}(e) = \{t \downarrow \vec{A} \quad | \quad t \in e\}$

A.5 Operators on Locations

LocUnion $loc_1 \bullet loc_2 = ((loc_1^1 \oplus loc_2^1), (loc_1^2 \cup loc_2^2))$ where:

$$
\begin{aligned}
loc_1^1 \oplus loc_2^1 = \{&(dbname, t) \mid (dbname, t) \in loc_1^1 \wedge \\
&\wedge \nexists t' : (dbname, t') \in loc_2^1\} \cup \\
&\{(dbname, t) \mid (dbname, t) \in loc_2^1 \wedge \\
&\wedge \nexists t' : (dbname, t') \in loc_1^1\} \cup \\
&\{(dbname, (t_1, t_2)) \mid (dbname, t_1) \in loc_1^1 \wedge \\
&\wedge (dbname, t_2) \in loc_2^1\}
\end{aligned}
$$

Choice

$$loc_1 \mid^{\delta}_{db} loc_2 = loc_1$$
$$loc_1 \mid^{\delta}_{db} loc_2 = loc_2$$

A.6 Path

Input filters grammar

$$\begin{array}{l}(1)F ::= F_1, \ldots, F_n \quad | \\ \quad | \quad (op, var, binder)label[F] \quad | \quad \emptyset \\ (2)op \in \{/, //, \_\} \\ (4)var \in label \cup \{\_\} \\ (5)binder \in \{\_, in, =\}\end{array}$$

$path_f(t) =$

1. if $f = f_1, \ldots, f_m$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_{f_1}(t)\ TupJoin(true)$ $\ldots TupJoin(true)\ path_{f_m}(t)$
2. if $f = (\_, \_, binder)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
3. if $f = (\_, \_, binder)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(\_)(label)(t))$;
4. if $f = (op, l, in)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = nav(op)(label)(t)[l]$;
5. if $f = (op, l, =)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}$;
6. if $f = (op, l, in)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \bigcup_{t_i \in nav(op)(label)(t)} \{[l : t_i]\} TupJoin(true) path_F(t_i)$;
7. if $f = (op, l, =)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}$ $TupJoin(true)\ path_F(nav(op)(label)(t))\}$;
8. if $f = (op, \_, \_)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
9. if $f = (op, \_, \_)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(op)(label)(t))$;
10. $path_\emptyset(t) = \{\}$;

A.7 Return

Output filters grammar

$$\begin{array}{l}(1)OF ::= OF_1, \ldots, OF_n \quad | \quad n[OF] \quad | \quad val \\ (2)val ::= n \quad | \quad var \quad | \quad f(var)\end{array}$$

$return_{of}(e) =$

1. if $of = n$, then $return_{of}(e) = \bigcup_{i=1}^{p} n$ where $e = \{t_1, \ldots, t_p\}$, $oid(n) = \nu oid$, $loc(n) = (\emptyset, \emptyset)$, and $freshness(n) = \tau$;
2. if $of = var$, then $return_{of}(e) = \{refresh\_oid(t.var) \quad | \quad t \in e\}$;
3. if $of = label[of']$, then $return_{of}(e) = \bigcup_{i=1}^{p} n[return_{of'}(\{t_i\})]$ where $e = \{t_1, \ldots, t_p\}$, $oid(n) = \nu oid$, $loc(n) = (\emptyset, \emptyset)$, and $freshness(n) = \tau$;
4. if $of = of_1, \ldots, of_k$, then $return_{of}(e) = \bigcup_{i=1}^{p} return_{of_1}(\{t_i\}), \ldots, return_{of_k}(\{t_i\})$ where $e = \{t_1, \ldots, t_p\}$.

5. if $of = f(var)$, then $return_{of}(e) = f(return_{var}(e))$;

where

1. $refresh\_oid(t_1, \ldots, t_n) = refresh\_oid(t_1), \ldots, refresh\_oid(t_n)$
2. $refresh\_oid((oid, loc, fr)label[t]) = (\nu(oid), loc, fr)label[refresh\_oid(t)]$
3. $refresh\_oid((oid, loc, fr)label) = (\nu(oid), loc, fr)label$