A Query Algebra for XML P2P Databases*

Carlo Sartiani[1]

Dipartimento di Informatica - Università di Pisa
Largo B. Pontecorvo 3 - 56127 - Pisa - Italy
sartiani@di.unipi.it

Abstract. One missing point in the current research about p2p XML databases is the definition of a proper query algebra that addresses p2p-specific issues, such as the dissemination and replication of data, the dynamic nature of the system, and the transient nature of data and replicas.
This paper describes a query algebra for queries over XML p2p databases that provides explicit mechanisms for modeling data dissemination and replication constraints.

1 Introduction

Peer-to-peer (p2p) database systems are usually composed by a *dynamic*, *open-ended* network of *autonomous* or semi-autonomous peers, which contribute data to the system and query data exported by other peers. These systems affirmed as an interesting evolution of distributed and integration systems as well as an attempt to overcome their limitations, namely the heavy administration load, the need for centralization points, and their quite limited scalability, as they blur the distinction between clients and servers, each peer being able to submit service requests and (simultaneously) support other peer requests.

Several *ongoing* projects focus on the the design and the implementation of p2p database systems, mostly for XML data and for supporting Semantic Web applications [1–3]. One missing point in the current research about p2p XML databases is the definition of a proper query logical algebra. In this context, a query algebra can be profitably used in the following tasks.

Query distribution Algebraic expressions are a convenient form into which queries and sub-queries can be translated and packaged, so to be distributed across the network; since algebraic expressions are independent from the actual implementation of the query engine in local nodes and from the local data organization (e.g., indexes, etc), they can be locally translated into highly optimized physical query plans, hence allowing for the best exploitation of local data structures and computational capabilities.

* Carlo Sartiani was funded by the FIRB GRID.IT project and by Microsoft Corporation under the BigTop project.

Distributed query rewriting As in distributed database systems, algebraic expressions can be manipulated to perform *global-scale* optimizations, such as query unnesting and replica selections, while pushing most of the optimization load to the local peers.

Existing query algebras for XML data, most notably the *official* algebra by W3C [4], have been defined in the context of static and centralized database systems, and cover issues ranging from query result analysis and query type-checking to the rigorous definition of the statical and dynamic semantics of XML query languages. As a consequence, they lack support for three key issues in p2p database systems:

- data are disseminated in multiple peers, which may appear and disappear unpredictably;
- data are usually *replicated* into multiple peers, and, due to the dynamic nature of the system, the replicas have a limited time validity;
- data distribution and replication may change during query execution.

Our contribution This paper describes a logical query algebra for queries over XML p2p databases. The relevance of the contribution is twofold. First, the algebra provides an abstraction from physical or system-specific issues, hence it can be used for reasoning about p2p query processing (p2p distributed optimization, in particular) without worrying about the peculiar issues of a given system. Second, it provides explicit mechanisms for modeling data dissemination and replication constraints: in particular, the algebra data model incorporates the notion of *locations*, which model peer content, as well as the notion of data freshness; moreover, the algebra provides operators for manipulating locations, and for expressing replication constraints, together with the related rewriting rules.

The proposed query algebra supports a relevant fragment of the XQuery query language [5] (FLWR queries with free nesting), and provides corresponding rewriting rules.

Paper outline The paper is organized as follows. Section 2 identifies some requirements for a p2p logical query algebra. Section 3, then, describes the algebra data model and operators. Next, Section 4 discusses rewriting rules that can be applied to algebraic expressions. Section 5, then, illustrates some related works. Section 6 concludes.

2 Requirements for a p2p Query Algebra

The design of the proposed query algebra has been guided by three main requirements that emerge in XML p2p databases (in addition to the obvious requirement of supporting queries on XML data). These requirements are discussed below.

Data dissemination Since data are dispersed on multiple peers, the algebra should model the notion of peer as well as the distribution of data into peers. Hence, the data model should not be limited to represent XML trees, but also peers with their content. A clear benefit of having explicit peer information inside algebraic expressions is the ability to support routing decisions taken at both the global and the local level.

Data replication To increase the robustness of p2p systems as well as their performance, data are usually replicated in high-speed/high-capacity peers. As a consequence, information about replicas (who is replicating what) should be part of the algebraic vision of the database, i.e., the data model should record both the data provenance and the data replication. Moreover, since replicas in a p2p context are usually not up-to-date (2PL/2PC synchronization protocols are too restrictive for this setting), replica information should be enhanced with details about the validity of these replicas, e.g., the period of time during which a replica can be *safely* used in place of the original data.

Data freshness Peer-to-peer database systems are chaotic systems, where some data are very frequently updated and others remain untouched for a long period of time. This chaotic nature, together with the presence of loosely synchronized replicas, makes important the explicit representation of data freshness information into algebraic expressions. This allows for the support of queries where the user can choose between *fresh* data, at the price of a potentially higher evaluation cost, and *older* data, potentially not up-to-date, retrieved much more quickly.

3 Query Algebra

The proposed algebra is based on that of [6]. The most important extensions concern the representation of peer contents and replicas, the introduction of a *data freshness* notion, as well as algebraic operators for manipulating them.

For reasons of space, we focus here on the description of new operators and refer the reader to [6] for more detail about the others.

3.1 Data Model and Term Language

The query algebra represents XML data as unordered forests of node-labeled trees. According to the term grammar shown in Fig. 1, each tree node (n) has a unique *object identifier* (oid) that can be accessed by the special-purpose function *oid*; an algebraic support operator ν is used to generate new oids and to refresh existing ones. Furthermore, each node 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 node was performed ($\perp$ indicates that the freshness is undefined, and it is necessary to ensure the closeness of the model).

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

Fig. 1. Term grammar.

The label, the location, and the freshness of a node can be accessed by means of the auxiliary functions *label*, *loc*, and $freshness$, which are used thorough the whole algebra. For the sake of simplicity, we assume that peers perform leaf updates only (deletions, insertions, and value changes), hence the model satisfies the following parent/child freshness constraint.

Property 1 (Structural freshness constraint). Given a data tree t, it holds that:

$$t = n[t_1] \Rightarrow freshness(n) \leqslant freshness(t_1)$$
$$t = n[t_1, \ldots, t_k] \Rightarrow freshness(n) \leqslant min_i freshness(t_i)$$

Locations model the content of peers, hence they are represented as a pair of partial functions: the first function (loc^1) returns, for each database identifier, the trees contributed to the database by the given peer, if any; the second function (loc^2), instead, describes the replication services offered by a given peer, i.e., it returns, for each database identifier and location, the replicated trees for such database and location, if any. Replicas are further described by a (distributed) set *replicas*, which contains dynamic replication constraints. A replication constraint has the form $(loc_1, loc_2, db, \delta_1, \delta_2)$, and it states that loc_2 replicates the content of loc_1 for the database *db* from time δ_1 to time δ_2 (δ_2 may assume the special value ∞, which indicates that the replica is always kept up to date); given the dynamic nature of the system, we expect replication constraints to evolve over time.

Location content can be accessed through the function *content*, as shown below:

$$content(loc) = \bigcup_{id} loc^1(id)$$
$$AllLocs(id) = \{loc \mid loc^1(id) \neq \emptyset\}$$

The set of locations containing data relevant for a given database *db* is returned by the function *AllLocs*. The way *AllLocs* is computed and updated goes far beyond the scope of this paper, as it depends on the physical organization of the system; we only assume that the system will provide a set supposed to comprise an exhaustive list of locations containing relevant data. As usual in p2p systems, we also expect this set to be incomplete or even incorrect. The same considerations apply to *replicas*.

3.2 Global Time

For the sake of supporting freshness parameters, the data model has a universal constant τ, which denotes the current global time in the system. The hypoth-

esis of the existence of a global time, shared by all peers in the network, even though unrealistic, is not restrictive and does not affect the well-foundness of the algebra. Indeed, as shown in [2], query results are usually incomplete in p2p systems, and their incompleteness implies, in many cases, their incorrectness, so the assumption of the existence of a universal shared time does not significantly affect the quality of query results.

To support dynamic replication constraints, we assume that each query has two time parameters: the query issuing time τ', and the maximum replica time $\delta_{\tau'}$, which indicates that replication constraints of the form $(loc_1, loc_2, db, \delta_1, \delta_2)$ with $\delta_2 \geqslant \tau' - \delta_{\tau'}$ can be considered during query compilation.

3.3 *Env* Structures

Most algebraic operators manipulate unordered sequences of tuples, each tuple containing the variable bindings collected during query evaluation. These sequences (called *Env* structures as they mimic an environment) allow one to define algebraic operators that manipulate sequences of tuples, instead of trees; hence, common optimization and execution strategies (which are based on tuples rather than trees) can be easily adapted to XML.

Tuples in a given *Env* structure are *flat*, hence they cannot be nested one another. Each variable binding associates a variable to a collection, possibly a singleton, of node identifiers.

To ensure the closure of the algebra, intermediate structures are themselves represented as node-labeled trees conforming to the algebra data model; this kind of representation also allows one to apply useful optimization properties to border operators.

3.4 *path* and *return*

path and *return* represent the interface between XML data and intermediate *Env* structures. They allow for creating *Env* structures from XML trees (*path*) and for creating new documents from existing *Env* structures.

path The main task of the *path* operator is to extract information from the database, and to build variable bindings. The way information is extracted is described by an *input filter*; a filter is a tree, describing the paths to follow into the database (and the way to traverse these paths), the variables to bind and the binding style, as well as the way to combine results coming from different paths. Input filters, hence, are just a way to describe query twigs, according to the grammar shown in Table 3.1.

A simple filter $(op, var, binder)label[F]$ tells the *path* operator a) to traverse the current context by using the navigational operator *op*, b) to select those elements or attributes having label *label*, c) to perform the binding expressed by *var* and *binder*, and d) to continue the evaluation by using the nested filter F.

An input filter fully describes the behavior of its enclosing *path* operator. In addition to an input filter, the *path* takes as input a data model instance, that

Table 3.1. *Input filters grammar*

$F ::= F_1, \ldots, F_n$	conjunctive filters	$op \in \{/, //, \_\}$	navigational axes
$\mid F_1 \vee \ldots \vee F_n$	disjunctive filters	$var \in String \cup \{\_\}$	variable names
$\mid (op, var, binder)label[F]$	simple input filter	$binder \in \{\_, in, =\}$	binders
$\mid \emptyset$	empty filter		

is browsed according to the specification given by the input filter; hence, a *path* operator has the syntax $path_F(t)$, where F is the input filter and t the data model instance. The result of the evaluation of a *path* operator is a sequence of tuples containing the variable bindings described in the filter. The following examples show the behavior of *path*.

Example 1. Consider a real-estate p2p market database, and consider the following query fragment.

```
for $b in input()//building,
    $d in $b/desc,
```

This clause retrieves descriptions for buildings at any level in the database. Assuming that the query plan generation layer found only one relevant location loc_1, the clause can be translated into the following *path* operation:

$path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset]]}(loc_1^1(db1))$

which returns the following *Env* structure:

$\$b : o_1$	$\$d : o_{11}$
$\$b : o_1$	$\$d : o_{12}$
$\$b : o_3$	$\$d : o_{24}$
$\ldots$	$\ldots$

Example 2. Consider the following XQuery fragment:

```
for $b in input()//building
let $d_list := $b/desc,
```

This query retrieves buildings and building descriptions in the database; unlike the previous example, descriptions of the same building are grouped together in $\$d\_list$. This query fragment can be expressed by the following *path* operation:

$path_{(//,\$b,in)building[(/,\$d\_list,=)desc[\emptyset]]}(loc_1^1(db1))$

which returns the following *Env* structure:

$\$b : o_1$	$\$d\_list : \{o_{11}, o_{12}, \ldots\}$
$\$b : o_3$	$\$d\_list : \{o_{24}, \ldots\}$

As shown by the filter grammar, multiple input filters can be combined to form more complex filters. The algebra allows filters to be combined in a *conjunctive* way, or in a *disjunctive* way. In the first case, the *Env* structures built by simple filters are joined together, hence imposing a product semantics; in the second case, partial results are combined by using an *outer union* operation. Therefore, disjunctive filters can be used to map occurrences of $op : union$ inside

paths into input filters, as well as more sophisticated queries; the use of outer union ensures that the resulting *Env* has a uniform structure, i.e., all binding tuples have the same fields.

return While the *path* operator extracts information from existing XML documents, the *return* operator uses the variable bindings of an *Env* to produce new XML documents. *return* takes as input an *Env* structure and an *output filter*, i.e., a skeleton of the XML document being produced, and returns a data model instance (i.e., a well-formed XML document) conforming to the filter. This instance is built up by filling the XML skeleton with variable values taken from the *Env* structure: this substitution is performed once per each tuple contained in the *Env*, hence producing one skeleton instance per tuple.

Output filters satisfy the following grammar:

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

An output filter may be an *element constructor* ($n[OF]$), which produces an element tagged n and whose content is given by OF, a value constructor (n), or a combination of output filters ($OF_1, \ldots, OF_n$). The production describing values (val) needs further comments. The algebra offers two ways to publish information contained in an *Env* structure: by copy (νvar) and by reference (var). Referenced elements are published as they are in query results; in particular, their object ids are not changed, as well as their location and freshness information. Copied elements, instead, are published with *fresh* oids, while their location and freshness information remains untouched. Finally, newly created elements ($OF ::= n[OF]$) and values ($val ::= n$) are managed as copied nodes with undefined freshness, so they have fresh oids, empty location, and are marked with the undefined time information ($\perp$).[1]

The following example shows the use of the *return* operator.

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 market, and it can be represented by the following algebraic expression:

$$\begin{array}{l}return_{entry[\nu\$d,\nu\$p]}(\\ \quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\ \quad\quad loc_1^1(db1)))\end{array}$$

[1] Any freshness comparison w.r.t $\perp$ is true, so the freshness structural constraint still holds.

3.5 Operators on Locations

Operators on locations are crucial for retrieving data coming from multiple peers, and for exploiting, if necessary, replicas of the content of some location. The query algebra offers two location operators: *LocUnion* and *Choice*.

LocUnion ($\bullet$) takes as input two locations loc_1 and loc_2, and it returns a new location obtained by uniting the content and the replica functions of the arguments, as shown in Table 3.2.

Table 3.2. *Formal definition of LocUnion*

$$
\begin{aligned}
loc_1 \bullet loc_2 &= ((loc_1^1 \oplus loc_2^1), (loc_1^2 \cup loc_2^2)) \\
loc_1^1 \oplus loc_2^1 &= \{(dbname, t) \mid (dbname, t) \in loc_1^1 \wedge \nexists t' : (dbname, t') \in loc_2^1\} \cup \\
&\quad \{(dbname, t) \mid (dbname, t) \in loc_2^1 \wedge \nexists t' : (dbname, t') \in loc_1^1\} \cup \\
&\quad \{(dbname, (t_1, t_2)) \mid (dbname, t_1) \in loc_1^1 \wedge (dbname, t_2) \in loc_2^1\}
\end{aligned}
$$

LocUnion is primarily used for expressing queries retrieving data from multiple peers. The following example shows the use of *LocUnion*.

Example 4. Consider our real-estate market database, and assume that new locations (loc_{11}, loc_{13}, and loc_{17}) contribute data about buildings. Then, the query of Example 3 can be expressed by the following algebraic expression:

$$
\begin{aligned}
&return_{entry[\nu\$d, \nu\$p]}(\\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\
&\qquad (\bullet_{i=1,11,13,17} loc_i)^1(db1)))
\end{aligned}
$$

As shown in Section 4.1, *LocUnion* operations can be extruded from *path* operators, hence the previous expression can be rewritten as follows:

$$
\begin{aligned}
&return_{entry[\nu\$d, \nu\$p]}(\cup_{i=1,11,13,17} \\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\
&\qquad loc_i^1(db1)))
\end{aligned}
$$

The *Choice* ($|_{db}^{\delta}$) operator is a convenient way to encapsulate replication constraints into query plans. $loc_1 \mid_{db}^{\delta} loc_2$ indicates that loc_2 replicates $loc_1^1(db)$ until time δ, so, if permitted, it can serve requests for data in $loc_1^1(db)$. As a consequence, $loc_1 \mid_{db}^{\delta} loc_2$ can be rewritten (in *path* operations concerning db) as loc_1 or as $loc_2^2(loc_1)$.

The following example shows the use of *Choice*.

Example 5. Consider the query of the previous example, and assume 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:

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

4 Optimization Properties

Four main classes of algebraic rewriting rules can be applied to the query algebra. The first class contains *classical* equivalences inherited from relational and OO algebras (e.g., *push-down* of *Selection* operations and commutativity of joins); the second class consists of path decomposition rules, which allows the query optimizer to break complex input filters into simpler ones; the third class contains equivalences used for unnesting nested queries; the last class, finally, contains rewriting rules for location operators. For the sake of brevity, we focus here on location rewritings (the reader can see [6] for a detailed list of equivalence rules for the core of this algebra).

4.1 Location Rewriting Rules

Operators on locations represent a crucial fragment of a query algebra for p2p databases; as a consequence, rules for simplifying location operators as well as for splitting complex location unions are a *must*. The algebra offers three main rewriting rules for location operators: extrusion of *LocUnion* operations from *path* operations; simplification of *Choice* operators; and introduction of *Choice* operations.

Proposition 1 (Extrusion of *LocUnion* operations).
Given a database db disseminated on loc_1 and loc_2, it holds that:

$$path_f((loc_1 \bullet loc_2)^1(db)) = path_f((loc_1)^1(db)) \cup path_f((loc_2)^1(db))$$

This property states that *LocUnion* operations inside path operations can be split and distributed across the query; this, in turn, allows the system to more easily decompose a query in single-location subqueries.

Proposition 2 (Rewriting of location choices).
Given a database db hosted at loc_1 and replicated at loc_2, it holds that:

$$path_f((loc_1 \mid^{\delta}_{db} loc_2)^1(db)) = path_f(loc_1^1(db))$$
$$path_f((loc_1 \mid^{\delta}_{db} loc_2)^1(db)) = path_f(loc_2^2(loc_1)(db))$$

This property shows how a *Choice* operation inside a *path* operation can be rewritten; we expect that this rewriting will be guided by additional information about network conditions, peer computing power, etc.

Proposition 3 (Choice introduction).
Given a database db, if $(loc_1, loc_2, db, \delta_1, \delta_2) \in replicas$, *and* $\delta_2 \geqslant \tau' - \delta_{\tau'}$, *then* $loc_1^1(db) \rightarrow (loc_1 \mid_{db}^{\delta_2} loc_2)^1(db)$

Corollary 1 (Guarded choice introduction). *Given a database db disseminated on* $loc_1, \ldots, loc_m$, *if* $(loc_i, loc_j, db, \delta_1, \delta_2) \in replicas$, *and* $\delta_2 \geqslant \tau' - \delta_{\tau'}$, *then*

$path_f((loc_1 \bullet loc_i)^1(db)) \rightarrow path_f((loc_1 \bullet (loc_i \mid_{db}^{\delta_2} loc_j))^1(db))$

These properties back the introduction of *Choice* operations in query plans.

The following example illustrates how these properties can be used during query compilation.

Example 6. Consider the real-estate database of Section 3, and assume that peer p_i submits the query of Example 4 to the system (we report it below for the sake of clarity).

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

Assuming, as in Example 4, that relevant data for the query are hosted at locations loc_1, loc_{11}, loc_{13}, and loc_{17}, then the system compiles this query into the following algebraic expression.

$$return_{entry[\nu\$d,\nu\$p]}(\\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\ (\bullet_{i=1,11,13,17} loc_i)^1(db1)))$$

In the case that no suitable replicas are available, the system can just apply Proposition 1, so to push down *path* operations and to maximize the query fragments to be delivered to remote peers, as shown below.

$$return_{entry[\nu\$d,\nu\$p]}(\cup_{i=1,11,13,17} \\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(\\ loc_i^1(db1)))$$

Assume now that *replicas* contains two relevant replication constraints for the query: $(loc_{11}, loc_1, db1, \tau_1, \tau_2)$ and $(loc_{11}, loc_{24}, db1, \tau_3, \tau_4)$. If loc_{24} models a very reliable and fast peer, the system may decide to apply Corollary 1 and Proposition 2, so to speed up query execution, as shown below.

$$return_{entry[\nu\$d,\nu\$p]}(\\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_1^1(db1)) \cup \\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_{24}^2(loc_{11})(db1)) \cup \\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_{13}^1(db1)) \cup \\ path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_{17}^1(db1)))$$

Alternatively, the system may decide, on the basis of network traffic conditions and other parameters, to exploit the first replication constraint, so to concentrate the query load to the peer corresponding to loc_1. In this case, the application of Corollary 1 and Proposition 2 leads to the following algebraic expression.

$$
\begin{aligned}
&return_{entry[\nu\$d,\nu\$p]}(\\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_1^1(db1)) \cup \\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_1^2(loc_{11})(db1)) \cup \\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_{13}^1(db1)) \cup \\
&\quad path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_{17}^1(db1)))
\end{aligned}
$$

It should be noted that, unlike Proposition 1, which is always applicable, the application of Corollary 1 and Proposition 2 is subject to conditions that may change over time, hence, after a certain period of time, the algebraic expressions reported above may become invalid.

5 Related Works

Current research about the algebraic treatment of queries in p2p database systems mostly focuses on *physical* query algebras for relational databases. [7] presents a physical query algebra for a DHT-based relational p2p database system. This algebra provides low-level operators for supporting relational queries on a DHT, which cannot be generalized to other contexts. In particular, no abstract notions of locations and replicas are provided, and all operators strictly depend on the presence of a DHT.

In [8] a more sophisticated approach is described. Queries are posed against *virtual* tables by means of a standard relational algebra, and are then translated into relational *concrete* queries over *local* and *distributed* tables; concrete queries are expressed through a concrete query algebra, which contains operators inspired by traditional distributed systems. Distributed tables are attached to peers, so they can be used to implicitly denote locations. While this approach is more general than that of [7], the lack of explicit modeling of locations and replicas is a significant difference with our algebra.

The explicit modeling of locations is not new. For instance, [9] contains a formalization of *static* p2p systems in terms π-calculus expressions. The main limitation of the work is the lack of support for dynamic topologies.

6 Conclusions

This paper describes a query algebra for XML p2p database systems. The algebra features mechanisms for dealing with p2p-specific issues, namely the dissemination and replication of data across an unstable network, as well as for incorporating replication constraints into query plans.

Even though designed for a specific class of systems (XML databases), the key ideas of the proposed algebra can be generalized to p2p systems with different data models.

The proposed algebra is now being used in the XPeer [10] p2p system: XPeer is a scalable and self-organizing p2p system for XML data designed for resource discovery applications.

The proposed algebra represents the first step in the development of a p2p query optimization systems. Its definition will be the starting point for the further investigation of suitable rewriting rules and for the design of a proper query optimizer.

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. 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)
3. 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
4. Draper, D., Fankhauser, P., Fernandez, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium (2005) W3C Working Draft.
5. 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.
6. 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)
7. Sattler, K.U., Rösch, P., Buchmann, E., Böhm, K.: A physical query algebra for dht-based p2p systems. In: Proceedings of 6th Workshop on Distributed Data and Structures (WDAS'2004), Lausanne, Switzerland, July 8-9, 2004. (2004)
8. Boncz, P., Treijtel, C.: Ambientdb: relational query processing in a p2p network. Technical report, CWI - INS (2003)
9. Gardner, P., Maffeis, S.: Modelling dynamic web data. In Lausen, G., Suciu, D., eds.: DBPL. Volume 2921 of Lecture Notes in Computer Science., Springer (2003) 130–146
10. 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)