On the Correctness of Query Results in XML P2P Databases

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Via F. Buonarroti 2 - 56127 - Pisa - Italy
sartiani@di.unipi.it

Abstract

In XML peer-to-peer *(p2p) database systems, query results are usually assumed to be* incomplete*. Incompleteness issues derive from the unstable and open-ended nature of the network, where new nodes may connect at any time, and existing nodes may suddenly disappear.*

The incompleteness of input data used for query evaluation may also lead to the incorrectness of query results, which greatly affects the usefulness of the whole p2p approach to XML databases.

In this paper we formally deal with the problem of result correctness in the presence of incomplete input data, and identify query classes for which the result correctness can be statically predicted at no extra cost w.r.t. usual syntactical and semantic query analysis.

1. Introduction

The last three years have seen the rapid emerging of the *peer-to-peer* (p2p) computational model. In this model, the system is composed of an *open-ended* and dynamic network of peers, which share data, computational resources, etc. Peers are usually autonomous or semi-autonomous, and may cooperate together in the execution of computational tasks, or in the hosting and querying of data.

In the field of database research, p2p systems affirmed as an interesting evolution of distributed and integration systems. Several projects focus on the design and the implementation of p2p database systems, mostly for XML data. In these systems, query results returned by the p2p query engine are usually *incomplete*, i.e., they are a subset of the result computed by centralizing the whole database in a single site s and then executing the query on s. Incompleteness issues derive from the unstable and open-ended nature of the network, where new nodes may connect at any time, and existing nodes may suddenly disappear. The non-feasibility of the instantaneous propagation of topology and schema change information, together with the clear and present danger of network partitions due to link failures, leads to the assumption that queries are executed on a fraction of input data only, which in turn gives rise to result incompleteness.

The incompleteness of input data used for query evaluation may also lead to the *incorrectness* of query results, i.e., the system may return data that are not part of the result produced by a corresponding centralized system. Consider, for instance, a query retrieving all authors in a bibliographic database having published less than five papers: if input data are incomplete, the result returned by the system may contain also authors with more than five papers.

Thus, the evaluation of an XML query on a p2p database may produce incomplete as well as incorrect results, which greatly affects the quality of results, and the usefulness of the whole p2p approach to XML databases.

Our contribution In this paper we formally deal with the problem of result correctness in the presence of incomplete input data. First, we map a subset of XQuery into a p2p query algebra; then, we formalize our intuitive notion of result correctness, and, by relying on the algebraic mapping, we identify query classes for which the result correctness can be statically predicted. As shown in Section 5, the analysis for result correctness requires no extra cost w.r.t. usual syntactical and semantic query analysis.

Paper outline The paper is organized as follows. Section 2 presents the reference scenario for this work, and Section 3 introduces our notion of completeness and correctness of query results. Section 4 describes the query algebra, while Section 5 formalizes the correctness problem, and shows the main results of the paper. In Sections 6 and 7, we discuss some related works and draw our conclusions.

2. Motivating scenario

The background for this paper is a p2p database management system for XML data [8], composed of an *open-ended* network of *fully autonomous* peers, which share *heterogeneous* data and pose queries on these data; in particular, unlike [5] and [4], no restriction is imposed on the semantic category of data contributed to the system. The system *self-*

organizes its overlay network, and requires no human intervention for its administration.

Peers may connect to the system at any time, and they may disconnect at any time. Hence, the resulting topology is potentially very dynamic, which distinguishes this system from traditional distributed database systems, where network topology is almost static. Furthermore, peers may freely update their local data, the only restriction being the impossibility to relocate subtrees; in particular, peers may perform *schema changing* updates, i.e., updates that trigger schema modifications. As a consequence of the dynamicity of the system, in both the topology and the schemas of the contributed data, and of the zero-administration policy, the system does not use schema mapping and integration techniques, even in their distributed versions [4].

In addition to share data, peers may submit queries to the database system. These queries, expressed in the FLWR core of XQuery [2] without universally quantified predicates, are sent from peers to the *query plan generation layer*, and query plans are sent back to peers for the execution.

The system returns query results without preserving the document order of XML data: this feature is motivated by the absence of a global order notion for data spanning on multiple sites, which makes the preservation of local document order pretty useless for queries returning results coming from a wide number of sites.

3. Completeness and correctness of query results

In this Section we will provide a basic intuition of the notions of completeness and correctness of query results in a p2p setting; this intuition will help the reader in understanding the formal system described in the next Sections.

Consider the following query, which extracts all authors-title pairs from a bibliographic database:

```
for $p in input()//article
    $a in $b/author,
    $t in $b/title
return <author-title> {$a,$t} </author-title>
```

The result of this query is a forest of trees rooted by author-title elements.

Assume that the data relevant to the query are dispersed over a set $\mathcal{L}$ of four peers, let's say $\mathcal{L} = \{l_2, l_{15}, l_{18}, l_{33}\}$. If the query plan generation layer returns a query plan containing all the four peers, then the p2p query engine will compute (in the absence of network or peer failures) a *complete* result res, i.e., res will contain all author-title pairs in the database. Instead, if a subset $\mathcal{L}'$ of $\mathcal{L}$ is returned to the query engine, then the query result res' will consist of a fraction of the complete result res, so the query engine will produce an *incomplete* but correct result.

Consider now the following query, which returns all authors having published less than five papers in the last year.

```
for $a in input()//author
let $p_list := for $p in input()//article
               where $p/author = $a AND
                     $p/year = 2003
               return {$p}
where count($p_list) < 5
return <subIudice> {$a} </subIudice>
```

As for the previous query, assume that the set of peers $\mathcal{L} = \{l_2, l_{15}, l_{18}, l_{33}\}$ contains all the data relevant for the query and, in particular, that peers l_{15} and l_{33} contain information about three and four papers published by John Doe in the last year, respectively, while l_2 and l_{18} make no mention of John Doe. Then, if the subset $\mathcal{L}'' = \{l_2, l_{15}, l_{18}\}$ is returned to the query engine, then the query engine will produce a result containing an entry for John Doe, which is clearly wrong. As a consequence, the query engine will post an *incorrect* result, comprising data that do not satisfy the requirements of the query.

4. Query algebra

In this Section we introduce the query algebra used for mapping the FLWR core of XQuery. The mapping of XML queries into algebraic expressions is straightforward, and it is shown in [7].

4.1. Data model and term language

Data in the system are represented as unordered forests of node-labeled trees. According to the term grammar shown below, each node is augmented with the indication of the hosting peer (by means of a *logical location* loc) as well as with its *label*, e.g., the tag of the corresponding XML element. The label and the location of a node can be accessed by means of the auxiliary functions $label$ and loc.

$t ::= t_1, \ldots, t_p \quad | \quad n[t] \quad | \quad n$
$n ::= (loc)label$
$loc : dbname \rightarrow t$
where $label \in \Sigma^*$ and loc is a partial function.

Logical locations model the content of peers, hence they are represented as a partial function returning, for each database identifier, the trees contributed to the database by the given peer, if any.

The set of locations containing data relevant for a given database db is returned by the function $AllLocs(db)$. We expect the query plan generation layer to compute, for a database db, a subset of $AllLocs(db)$, or, even worst, a partially overlapping set ls.

4.2. Algebra operators

The query algebra, in the spirit of YAT [3], exploits *relational-like* intermediate structures, called Env structures, to accumulate variable bindings collected during query evaluation. Env structures can be seen as streams of tuples carrying variable bindings, and, to ensure the closure of the algebra, they can be represented as node-labeled trees conforming to the data model.

Env structures are manipulated by quite traditional operators, such as $Selection$, $Projection$, $TupJoin$, and $DJoin$. In addition to these operators, the query algebra features three supplementary operators: $LocUnion$, $path$, and $return$. $LocUnion$ is used for manipulating locations, while $path$ and $return$ perform conversions from data model instances to Env structures, and *vice versa* (the full definition of the operators can be found in [7]). In the following we will survey key algebraic operators such as $path$, $return$, $LocUnion$, and $Selection$.

$path$ 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*; according to the grammar shown below, an input 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.

$$\begin{array}{rl} F & ::= F_1, \ldots, F_n \\ & \mid\ (op, var, binder)label[F] \\ & \mid\ \emptyset \\ \text{where} & op \in \{/, //, \_\}, \\ & var \in String \cup \{\_\}, \\ & binder \in \{\_, in, =\} \end{array}$$

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.

The $path$ operator takes as input a data model instance t and an input filter, and it returns an Env structure containing the variable bindings described in the filter. The following example shows a simple input filter and its application to a sample document.

Example 4.1 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:

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

■

As shown by the filter grammar, multiple input filters can be combined to form more complex filters: in this case, the Env structures built by simple filters are joined together, hence imposing a product semantics.

$return$ 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.[1]

Output filters satisfy the following grammar:

$$\begin{array}{l} (1) OF ::= OF_1, \ldots, OF_n \ \mid\ n[OF] \ \mid\ val \\ (2) val ::= n \ \mid\ var \ \mid\ f(var) \\ (3) f \in \{avg, min, max, sum, count, \ldots\} \end{array}$$

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 (val), or a combination of output filters ($OF_1, \ldots, OF_n$). Copied elements (var) are published as they are, i.e., their location information remains untouched, while newly created elements ($OF ::= n[OF]$) and values ($val ::= n$) receive an empty location.

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

Example 4.2 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:

$$return_{entry[\$d,\$p]}(path_{(//,\$b,in)building[(/,\$d,in)desc[\emptyset],(/,\$p,in)price[\emptyset]]}(loc_1(db1)))$$

■

1 Since XQuery lacks the `group-by` operator, aggregation functions can be applied to *let* variables only.

$LocUnion$ $LocUnion$ is an operator used for combining data dispersed over multiple peers. $LocUnion$ ($\bullet$) takes as input two logical locations loc_1 and loc_2, and it returns a new logical location obtained by uniting the content functions of the arguments. The following example shows the use of $LocUnion$.

Example 4.3 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 4.2 can be expressed by the following algebraic expression:

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

■

$Selection$ As in many other query algebras, $Selection$ σ takes as input an Env and a boolean predicate P, and returns a new env structure where binding tuples not satisfying P are missing.

5. Correctness properties

In this paper we study the correctness properties of FLWR queries executed on top of p2p databases. The study is based on two assumptions. First, we assume that trees are *locally complete*, i.e., a tree is fully contained within the same location.

Definition 5.1 (Tree local completeness) *A* non-leaf *data model instance* t *is* locally complete *if:*

- *if* $t = n[t_1, \ldots, t_p]$, *then* $loc(n) = l_1 \Rightarrow loc(t_1) = l_1 \wedge \ldots \wedge loc(t_p) = l_1$, *and* $t_1, \ldots, t_p$ *are locally complete;*
- *if* $t = t_1, \ldots, t_p$, *then* $t_1, \ldots, t_p$ *are locally complete and* $loc(t_1) = loc(t_2) = \ldots = loc(t_p)$.

The second assumption is that the query plan generation layer generates no *false positives*, i.e., it does not fill query plans with locations that have no data matching the for/ let clauses of the query. This implies that, in a way that will be detailed later, the query plan generation layer returns a subset of $AllLocs(db)$.

To define our notion of correctness, we introduce a tree containment relation: this relation, formally defined below, is crucial since input data, query results, as well as Env structures are represented as trees.

Definition 5.2 (Tree containment) *The tree containment relation is inductively defined as follows:*

(1) $n_1 \leqslant n_2 \iff label(n_1) = label(n_2)$
(2) $n_1[t_1] \leqslant n_2[t_2] \iff label(n_1) = label(n_2) \wedge$
$\wedge t_1 \leqslant t_2$
(3) $t_1, \ldots, t_n \leqslant t_p, \ldots, t_{p+m} \iff$
$\exists t_j \; j \in [p, p+m] : t_1 \leqslant t_j \wedge$
$\wedge t_2, \ldots, t_n \leqslant t_p, \ldots, t_{j-1}, t_{j+1}, \ldots, t_{p+m}$
(4) $() \leqslant t$

The following definitions introduce the concepts of *location assignment* and query results. They are based on the representation of a query Q as a *location-free* algebraic expression, i.e., an algebraic expression with *spots* (context holes) in place of locations; the algebraic representation is obtained by mapping Q into corresponding algebraic expressions, and by applying common rewriting rules (push-down of selections, etc).

Definition 5.3 (Location assignment) *Given a query* $Q = (q)$ *on a database* db *and a set of locations* ls, *a location assignment for* Q *on* ls *is a function* ρ *mapping location spots in* q *into unions of locations in* ls: $\rho : spot \rightarrow loc$.

Definition 5.4 (Query result) *Given a query* $Q = (q)$, *a set of locations* ls, *and a location assignment* ρ *for* Q *on* ls, $Res_\rho(Q)$ *is the result of the evaluation of* Q *on* ρ.

By these definitions, a query is represented as an algebraic expression without locations and location operators, which are then introduced by location assignments, hence allowing one to parametrize a query w.r.t. input data. By relying on these definitions, we can formalize our notions of completeness and correctness of query results.

Definition 5.5 (Query result completeness) *Let* Q *be a query* $Q = (q)$, *let* ls *be the set of locations computed for* Q *by the plan generation layer, and let* ρ_1 *and* ρ_2 *be location assignments for* Q *on* ls *and on* $AllLocs(db)$, *respectively. Then, the result of the evaluation of* Q *on the system is* complete *if* $Res_{\rho_2}(Q) \leqslant Res_{\rho_1}(Q)$.

Definition 5.6 (Query result correctness) *Let* Q *be a query* $Q = (q)$, *let* ls *be the set of locations computed for* Q *by the plan generation layer, and let* ρ_1 *and* ρ_2 *be location assignments for* Q *on* ls *and on* $AllLocs(db)$, *respectively. Then, the result of the evaluation of* Q *on the system is* correct *if* $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$.

These notions are independent from the assumption of absence of false positives, hence they can be used also when this assumption is relaxed. To incorporate this assumption in our study, we need the following definition.

Definition 5.7 (Assignment compatibility) *Given a query* $Q = (q)$ *on a database* db, *two location sets* ls_1 *and* ls_2 ($ls_1 \subseteq ls_2$), *and a location assignment* ρ_1 *for* Q *on* ls_1, *then a location assignment* ρ_2 *for* Q *on* ls_2 *is* ***compatible*** *with* ρ_1 ($\rho_1 \propto \rho_2$) *if* $\forall spot \; s \; in \; q : \rho_2(s) = \rho_1(s)[f(l_i)/l_i, \ldots, f(l_j)/l_j]$, *where* $f(l_i) = l_i$ *or* $f(l_i) = l_i \bullet l_{k_1} \bullet \ldots \bullet l_{k_p}$ ($\{l_{k_1,\ldots,l_{k_p}}\} \subseteq ls_2$).

This definition says that ρ_2 extends ρ_1 in a *conservative* way, hence the following lemma holds.

Lemma 5.8 *Given a query* $Q = (q)$ *on a database* db, *two location sets* ls_1 *and* ls_2 ($ls_1 \subseteq ls_2$), *and two compatible location assignments* ρ_1 *and* ρ_2 *for* Q *on* ls_1 *and* ls_2 ($\rho_1 \propto \rho_2$) *respectively, then* $\forall spot \; s \; in \; q : \rho_1(s)(db) \leqslant \rho_2(s)(db)$.

Proof. By the definition of $LocUnion$.

The previous lemma states that location assignments are extended in a way that satisfies the tree containment relation, hence allowing one to reduce the problem of result correctness to the problem of checking whether the algebra operators in the query plan are monotone (or, even better, linear).

Definition 5.9 *An algebraic operator op is* monotone *if $e_1 \leqslant e_2 \Rightarrow op(e_1) \leqslant op(e_2)$, where e_1 and e_2 are Env structures and $Att(e_1) = Att(e_2)$ ($Att(e)$ is the set of variable names in e).*

Lemma 5.10 *Given a query $Q = (q)$ on a database db, two location sets ls_1 and ls_2 ($ls_1 \subseteq ls_2$), and two compatible location assignments ρ_1 and ρ_2 for Q on ls_1 and ls_2 ($\rho_1 \propto \rho_2$) respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if each operator in q is monotone.*

Proof. The thesis follows from Lemma 5.8, and from the definition of monotone operator.

5.1. Monotonicity properties of algebraic operators

Lemma 5.10 states that the result of the evaluation of a query, in the absence of false positives, is correct if each operator in the query plan is monotone. All operators in the algebra are monotone (the proof is trivial, except for $path$), with the only exceptions of $return$ and σ. The *non-monotonicity* of $return$ comes from the presence of aggregation functions in output filters, hence the following lemmas hold.

Lemma 5.11 *Let of be an output filter without aggregation functions. Then, $return_{of}$ is monotone.*

Proof. By the definition of $return$ and by the tree structural containment relation.

Lemma 5.12 *Let $of = f(\$x)$ be an output filter applying an aggregation function f to a set variable $\$x$. Then, $return_{f(\$x)}$ is **monotone** if the set bound to $\$x$ in any tuple is guaranteed to be complete.*

Proof. Let $e_1 \leqslant e_2$ and $Att(e_1) = Att(e_2)$. If the set bound to $\$x$ is guaranteed to be complete in any tuple, then $\forall s_1 \in e_1 \ \exists s_2 \in e_2$ such that $s_1 \leqslant s_2 \wedge s_1.\$x = s_2.\$x$. Let g the function mapping e_1 tuples into e_2 tuples. Then:

$$\begin{aligned} return_{f(\$x)}(e_1) &= \bigcup_{s_i \in e_1} f(s_i.\$x) = \\ &= \bigcup_{s_i \in e_1} f(g(s_i.\$x)) \leqslant return_{f(\$x)}(e_2) \end{aligned}$$

Observation 5.13 *Let $of = f(\$x)$ an output filter applying an aggregation function f to a set variable $\$x$. Then, $return_{f(\$x)}$ is not monotone if $\$x$ is not guaranteed to be bound, in any tuple, to a complete set.*

These lemmas shows that the presence of aggregation functions in query plans may lead to incorrect results. Similar considerations apply to σ, as shown by the following results.

Lemma 5.14 *Let $P(\$x)$ a predicate on the variable $\$x$. Then, $\sigma_{P(\$x)}()$ is monotone if $\$x$ is bound by the iterative binder (in).*

Proof. Let $e_1 \leqslant e_2$ and $Att(e_1) = Att(e_2)$. By the definition of tree containment, it follows that $e_1 \subseteq e_2$, or $\exists tuple_1 \in e_1$, $\exists tuple_2 \in e_2$ such that $tuple_1$ and $tuple_2$ differ for the set bound to a let variable $\$v$.

If $e_1 \subseteq e_2$, then $\sigma_{P(\$x)}(e_2) = \sigma_{P(\$x)}(e_1) \bigcup \sigma_{P(\$x)}(e_2/e_1)$; otherwise, $\pi^{\rightarrow}_{\$x}(e_1) \subseteq \pi^{\rightarrow}_{\$x}(e_2)$, so $\sigma_{P(\$x)}(e_1) \leqslant \sigma_{P(\$x)}(e_1)$.

Observation 5.15 *Let $P(\$x)$ a predicate on the variable $\$x$. Then, $\sigma_{P(\$x)}()$ is **not** monotone if $\$x$ is bound by the let binder to an incomplete set.*

Proof. The proof is based on a simple counterexample. Let P be a set predicate of the form $P(\$x) \equiv \$x = \{o_1\}$. Let $e_1 \leqslant e_2$ and $Att(e_1) = Att(e_2)$, where $e_1.\$x = \{o_1\}$ and $e_2.\$x = \{o_1, o_2\}$. Then $P(e_1.\$x)$ is **true**, while $P(e_2.\$x)$ is **false**.

Lemma 5.16 *Let $P(\$x)$ a predicate on the variable $\$x$. Then, $\sigma_{P(\$x)}()$ is monotone if $\$x$ is bound by the let binder and the set bound to $\$x$ in any tuple is guaranteed to be complete.*

Proof. Let $e_1 \leqslant e_2$ and $Att(e_1) = Att(e_2)$. If the set bound to $\$x$ is guaranteed to be complete in any tuple, then $\forall s_1 \in e_1 \ \exists s_2 \in e_2$ such that $s_1 \leqslant s_2 \wedge s_1.\$x = s_2.\$x$. Let f be the function mapping e_1 tuples into e_2 tuples. Then:

$$\begin{aligned} \sigma_{P(\$x)}(e_1) &= \bigcup_{s_i \in e_1} \sigma_{P(\$x)}(\{s_i\}) = \\ &\quad \bigcup_{s_i \in e_1} \sigma_{P(\$x)}(\{f(s_i)\}) \leqslant \sigma_{P(\$x)}(e_2) \end{aligned}$$

5.2. Main properties

In the previous Sections we see how $return$ and σ may lead, in particular circumstances, to incorrect query results. Lemmas for both $return$ and σ tie the incorrectness of query results to the presence of incomplete sets. Hence, before examining the main correctness results of the paper, it is necessary to investigate the sources of incomplete sets. Incomplete sets may be introduced as a consequence of the evaluation of an *unguarded* path expression, i.e., a path expression evaluated starting from the roots of the database

(e.g., $input()//book$), and as a consequence of the evaluation of a nested query. Nested queries may also lead to the binding of incorrect values to variables, whenever their results are flagged as (potentially) incorrect, so our theorems must take into account this issue[2].

Theorem 5.17 (Complete sets) *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and set variables in Q are bound to complete sets.*

Proof. By Observation 5.15 and Lemma 5.16.

This theorem is a straightforward application of the results of the previous Sections. We can go a step further with the following theorem, which extends the class of queries with correct results.

Theorem 5.18 *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and it does not contain set predicates or aggregation functions applied to variables bound to incomplete sets.*

The previous theorems identify a large class of queries whose results can be considered correct. Unfortunately, these theorems do not define query classes for which correctness can be statically enforced, since they depend on the property of set completeness, which in turn depends on the behavior of the query plan generation layer.

Classes of queries whose result correctness can be statically checked are identified by the following theorems.

Theorem 5.19 (For/no-let queries) *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and it does not contain set variables.*

Proof. By Observation 5.15 and Lemma 5.16.

Theorem 5.20 *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and it does not contain set predicates and aggregation functions.*

Proof. By Observation 5.15 and Lemma 5.16.

2 In the following we will use the expression *incorrect nested query* for indicating a nested query returning a result flagged as incorrect.

Corollary 5.21 *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and it does not bind set variables to the result of the evaluation of nested queries, or to the result of the evaluation of an* unguarded *path expression.*

Corollary 5.22 *Given a query Q, given $ls \subseteq AllLocs(db)$, and given two compatible location assignments ρ_1 and ρ_2 for Q on ls and $AllLocs(db)$ respectively, then $Res_{\rho_1}(Q) \leqslant Res_{\rho_2}(Q)$ if Q does not contain incorrect nested queries, and it does not contain set predicates or aggregation functions applied to variables bound to the result of the evaluation of nested queries, or to the result of the evaluation of an* unguarded *path expression.*

These theorems identify syntactical conditions guaranteeing the correctness of query results. The corresponding query classes are related to the classes of Theorems 5.17 and 5.18 by the following relations:

- Theorem 5.19 $\subseteq$ Corollary 5.21;
- Theorem 5.20 $\subseteq$ Theorem 5.18;
- Theorem 5.20 $\subseteq$ Corollary 5.22;
- Theorem 5.20 $\cap$ Corollary 5.21 $\neq \emptyset$, but no containment relation exists;
- Corollary 5.21 $\subseteq$ Theorem 5.17;
- Theorem 5.17 $\subseteq$ Theorem 5.18;
- Theorem 5.17 $\cap$ Corollary 5.22 $\neq \emptyset$, but no containment relation exists.

These relations induce the query class hierarchy shown in Figure 1. From this hierarchy it follows that the class of queries described by Corollary 5.22 is, at this time, the maximal class of queries for which we can statically enforce result correctness.

5.3. Extensions

The results described in the previous Sections are based on the key hypothesis of local completeness of input trees. It is worth to see what happens when this assumption is relaxed.

By relaxing the tree local completeness properties, we assume that a single tree can be fragmented among multiple sites, e.g., if $t = n[t_1, t_2]$, then $loc(n) = l \not\Rightarrow loc(t_1) = l \vee loc(t_2) = l$. In this case the evaluation of a guarded path expression, e.g., $\$b/desc$, may lead to incomplete sets, since nodes at any level in the tree can be dispersed in multiple locations. As a consequence, Corollaries 5.21 and 5.22 do not hold under this relaxed hypothesis, hence the maximal class of queries for which result correctness can be statically checked is described by Theorem 5.20.

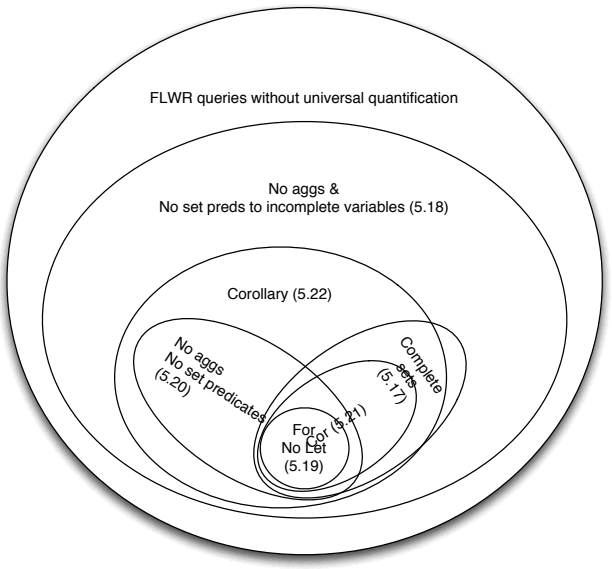

Figure 1. Overall query class hierarchy.

6. Related works

To our knowledge, no previous work addressed the problem of query result correctness in XML p2p databases. As a consequence, in this Section we briefly review existing works on XML p2p databases.

In [5] authors describe a *coordinator-free* architecture for distributed XML query processing in the context of p2p systems. The proposed architecture is based on two key ideas: *mutant query plans* (MQP), 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: MQPs traverse the system, carrying partial results and unevaluated sub-plans, until they are fully evaluated, i.e., they become 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.

In [6] 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 matching mismatch problems, as well as the use of Active XML [1] as the paradigm for service invocation/execution and data exchange.

In [4] 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.

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.

7. Conclusions

This paper studied the problem of query result correctness in XML p2p database. The paper identified classes of queries, whose correctness can be statically checked, the check being a simple syntactical analysis. These classes are large enough to cover most common practical cases.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. 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*, pages 1087–1090. Morgan Kaufmann, 2002.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, May 2003. W3C Working Draft.

[3] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[4] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. 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*, pages 556–567. ACM, 2003.

[5] V. Papadimos, D. Maier, and K. Tufte. 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.

[6] E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, and M. Vazirgiannis. DBGlobe: a service-oriented P2P system for global computing. *Sigmod Record*, 32(3):77–82, 2003.

[7] C. Sartiani. A Query Algebra for XML P2P Databases, 2003. Manuscript draft. Available at `http://www.di.unipi.it/~sartiani/papers/eve.pdf`.

[8] C. Sartiani, G. Ghelli, P. Manghi, and G. Conforti. XPeer: A self-organizing XML P2P database system. In *Proceedings of the First EDBT Workshop on P2P and Databases (P2P&DB 2004), 2004*, 2004.