An Approach to Detect Corrupted Schema Mappings in XML P2P Databases (Extended Abstract)*

Dario Colazzo[1] and Carlo Sartiani[2]

[1] Laboratoire de Recherche en Informatique (LRI)
Bat 490 Université Paris Sud
91405 Orsay Cedex France
dario.colazzo@lri.fr
[2] Dipartimento di Informatica - Università di Pisa
Via B. Pontecorvo 3 - 56127 - Pisa - Italy
sartiani@di.unipi.it

Abstract. In this paper we present an automatic technique for identifying corrupted as well as imprecise links in XML p2p database systems. Our technique, based on static type analysis of XQuery like queries, employs an enhanced version of the μXQ type system [3], that allows for a precise location of errors in queries wrt schema definitions.

1 Introduction

The last few years have seen the rapid emerging of two new Internet-related technologies. The first one, the eXtensible Markup Language (XML), was designed in an effort to make WWW documents cleaner and machine understandable, and has become the standard format for exchanging data between different data sources. The second technology, the *peer-to-peer* (p2p) computational paradigm, started as a way to easily share files over the Internet, and affirmed as a low-cost, scalable and flexible evolution of client-server and distributed systems.

In the field of database systems, these technologies combine to form to a new family of data management systems, sometimes called PDMSs (Peer Data Management Systems), that allow the user to easily share and query XML data dispersed over multitudes of sites without the complex and heavy administrative tasks that characterize traditional distributed database systems. Most current p2p database systems for XML data [6] [8] [5] [2] are based on a common model, where nodes in the system are connected through sparse *point-to-point* mappings, and queries are executed with decentralized versions of GAV and LAV query reformulation algorithms [10, 11]. PDMSs can be seen as a mixture of data integration an p2p technologies, hence resulting in adaptation of prior data integration techniques to a widely distributed and someway unstable environment.

The presence of mappings among peers, while allowing for a more precise and less expensive query routing than *flood-based* p2p systems (e.g., Gnutella [1]),

* Dario Colazzo was funded by the RNTL-GraphDuce project. Carlo Sartiani was funded by the FIRB GRID.IT project.

poses new problems related to the intrinsically dynamic nature of p2p systems, namely the fact that peers may join and leave the network at any time, as well as locally and independently change their data and schemas: for instance, when a peer changes the structure of the data it is sharing, its mappings with other peers may become corrupted. The introduction of corrupted mappings in the system greatly affects the quality and the quantity of the results that can be retrieved in response to a query. Indeed, queries are usually evaluated by traversing a chain of peers and by exploiting their mappings in the query reformulation process: as a consequence, the presence of a corrupted or imprecise mapping from peer p_i to peer p_{i+1} may make peer p_{i+1} unreachable, and may influence the reachability of other peers.

At this time, both the detection of problems in the current set of mappings and the maintenance of these mappings are performed manually by each site owner/user, and systems have no way to automatically warn the user about emerging issues in the mapping chain.

Our Contribution In this paper we present an automatic technique for identifying corrupted as well as imprecise schema mappings in XML p2p database systems. Our technique, based on the typechecking of XQuery queries, employs an enhanced version of the XQuery type system [3], that allows for a precise location of errors in queries wrt schema definitions. By relying on this technique, the p2p system can promptly warn the user about errors in mapping definitions and give her hints about the location of the errors in the mapping chain, as well as about the schema fragments whose mappings should be improved.

The proposed technique can be safely combined with traditional query processing algorithms based on the repeated applications of GAV and LAV algorithms over the transitive closure of schema mappings.

2 Motivating Scenario

We describe our technique by referring to a sample XML p2p database system inspired by Piazza [6]. The system is composed of a dynamic set of peers, capable of executing queries on XML data, and connected through *sparse point-to-point* schema mappings.

The state of the system is modeled as a dynamic set $\{p_i\}_i$ of peers, where each peer is represented as shown by Definition 1.

Definition 1. *A peer is a tuple* $p_i = (id, db, \mathcal{T}, \mathcal{V}, \{\rho_{ij}\}_j)$*, where id is the unique identity of* p_i*, db are the data published by* p_i*,* $\mathcal{T}$ *is the schema of the data of* p_i*,* $\mathcal{V}$ *is the world view of* p_i*, i.e., the view against which* p_i *queries are posed, and* $\{\rho_{ij}\}_j$ *is a set of* bidirectional *point-to-point mappings from* p_i *view to the views of other peers.*

As pointed out by the definition, each peer publishes some XML data (*db*), that may be empty, in which case the peer only submits queries to the system.

Each peer has two distinct schema descriptions. The first one, $\mathcal{T}$ (the *peer schema*), describes how local data are organized. The second one, $\mathcal{V}$ (the *peer view*), is a view over $\mathcal{T}$, and has a twofold role. First, it works as input interface for the peer, so that queries sent to peer p_i should respect p_i view of the world.

Second, it describes the peer *view* of the world, i.e., the virtual view against which queries are posed. $\mathcal{V}$ can be seen as the schema that the peer assumes to be adopted by the rest of the world; so, each peer poses queries against its peer view, since it assumes that the outer world adopts this schema.

The peer schema and the peer view are connected through a schema mapping, that shows how to translate a query expressed on the peer view into a query against the peer schema (in the following we will use the "schema mapping" to denote any mapping between types). The mapping can be defined according to the *Global As View* approach, or to the *Local As View* approach. The pros and cons of the GAV and LAV approach are extensively discussed in the literature; our distributed typechecking algorithm is independent from the view definition approach, so we can abstract from the specific view definition policy.

Since $\mathcal{V}$ plays the role of view of the world too, it can be loosely connected to $\mathcal{T}$: in particular, when $\mathcal{T}$ is empty, $\mathcal{V}$ is fully independent from $\mathcal{T}$:

The presence of two distinct schema descriptions has been introduced in the GLAV data integration approach [**?**], and it allows one to use both the LAV and GAV query rewriting approaches. $\mathcal{T}$ and $\mathcal{V}$ are systems of type equations conforming to the type system we will describe in the next Section.

In addition to (possibly empty) data and schema information, each peer contains a set, possibly a singleton, of *peer mappings* $\{\rho_{ij}\}_j$. A peer mapping ρ_{ij} from peer p_i to peer p_j is a pair of type correspondences $(\mathcal{V}_j \leftarrow q_{1j}, \mathcal{V}_i \leftarrow q_{2j})$ that map the view of p_i $(\mathcal{V}_i)$ into the view of p_j $(\mathcal{V}_j)$, and vice versa. More precisely, the mapping shows how to transform the extension of $\mathcal{V}_i$ into the extension of $\mathcal{V}_j$, the transformation being expressed through *queries*. These queries are then used to reformulate user queries against $\mathcal{V}_j$ into queries over $\mathcal{V}_j$.

Queries are expressed in the same query language used for posing general queries. These mappings link peers together, and form a sparse graph; queries are then executed by exploring the transitive closure of such mappings.[3]

For the sake of simplicity, we use bidirectional mappings, while real systems use mostly unidirectional mappings, the reverse mapping being obtained at run-time by applying LAV processing techniques, as described in [9].

Queries are expressed in an XQuery-like language, called μXQ, that is roughly equivalent to the FLWR core of XQuery, with two exceptions: first, we forbid the navigation of the result of a nested query by the outer query; second, we restrict the predicate language to the conjunction, disjunction, or negation of variable comparisons. These restrictions allow for a better handling of errors at the price of a modest decrease in the expressive power of the language: indeed, most nested queries are used without any further navigation of their results; moreover, a comparison between a variable and a constant can be simulated by binding the constant to a *let* variable in the binding section of the query.

The following Example illustrates the basic concepts of μXQ [4]

Example 1. Consider the following query Q posed against the Pisa view.

```
nuovaBib[
for $aut in $bib//autore
```

[3] We assume that ρ_{i0} specifies how to map the peer schema of p_i into its peer view, and vice versa.

[4] For the sake of simplicity, in the rest of he paper we will use / and // in place of *child* :: and *dos* ::, respectively.

```
let $pap := for $a in $bib/articolo
            let $a_list := $a/autore
            where $aut isin $a_list
            return lavoro[$a/titolo, $a/anno]
return item[$aut, $pap]]
```

posed against the following schema:

```
PisaBib = bib[(Articolo)*]
Articolo = articolo[Autore*,Titolo,Anno]
Autore = autore[String]
Titolo = titolo[String]
Anno = anno[Integer]
```

The query Q returns the list of authors in the database, together with the basic information about the papers they wrote. The clause `for $aut ... autore` iterates over the set of autore elements and binds the $\$aut$ variable to each autore node, hence returning a set of variable-to-node bindings. The `let` clause evaluates a nested query, whose result is a sequence of XML nodes, and binds the whole sequence to the $\$pap$ variable, hence producing a single variable binding. The nested query contains a `where` clause that checks whether the node bound to $\$aut$ in the outer query is in the set returned by the evaluation of the path expression $\$a/autore$.

Systems conforming to this architecture rely on schema mappings to process and execute queries; in particular, the cost of query execution, together with the quality of the results returned by the system, are deeply connected to the quality of schema mappings. Unfortunately, the evolution of the system, namely the connection of new nodes and the disconnection of existing nodes, as well as the changes in peer data and schemas, can dramatically affect the quality of schema mappings. In particular, the dynamicity of both the topology and the schema/content of nodes can lead to the corruption of existing mappings, which can significantly affect the quality of results returned by the system. Furthermore, existing optimization techniques for p2p systems, such as the mapping composition approach described in [9], can be vanished by mapping changes; this, in turn, reflects on the result quality as well as on the query processing cost (as shown in [9] and [7], mapping composition algorithms have unbounded space complexity).

As a consequence , the ability to detect corrupted mappings between alive nodes as soon as possible becomes very important, as shown in the following Example.

Example 2. Consider a bibliographic data sharing system, whose topology is shown in Figure 1.

Assume that Pisa and New York use the views of Figures **??** and **??**. Suppose now Pisa uses the query of Figure 4 to map its view into the view of New York. Pisa also uses the query of Figure 5 to map its view into its schema.

Consider now the query shown in Figure 6. This query, submitted by a user in Pisa, asks for all articles written by Mary F. Fernandez. The query is first executed locally in Pisa; since it expressed in terms of the Pisa view, the system rewrites the query by using the mapping of Figure 5, so to obtain a query posed against the Pisa schema. In particular, as the mapping of Figure 5 describes the

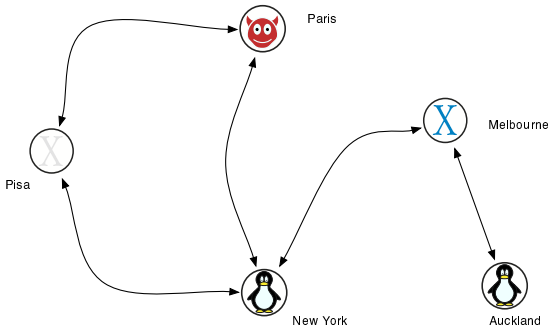

Fig. 1. Bibliographic p2p network.

```
PisaBib = bib[(Articolo)*]
Articolo = articolo[Autore*,Titolo,Anno]
Autore = autore[String]
Titolo = titolo[String]
Anno = anno[Integer]
```

Fig. 2. Pisa view.

```
NYBib = bib[(Article|Book)*]
Article = article[Author*,Title,Year, RefCode]
Author = author[String]
Title = title[String]
Year = year[Integer]
Book = book[Author*,Title,Year,Publisher, RefCode]
Publisher = publisher[String]
RefCode = refCode[String]
```

Fig. 3. New York view.

```
NYBibliography <-
Q1($input): for $ed in $input/editore
            return publisher[$ed/text()]
Q2($input): for $an in $input/anno
            return year[$an/data()]
Q3($input): for $t in $input/titolo
            return title[$t/text()]
Q4($input): for $aut in $input/autore
            return author[$aut/text()]
Q5($input): for $art in $input/articolo
            return article[Q4($art), Q3($art),Q2($art),Q6($art)]
Q6($input): refCode['xxx-Pisa-xxx']
Q7($input): for $bib in /pisaBib
            return bib[Q5($bib)]
```

Fig. 4. Pisa → New York mapping.

```
PisaBib <-
Q1($input): for $ed in $input/editore
            return editore[$ed/text()]
Q2($input): for $an in $input/anno
            return anno[$an/data()]
Q3($input): for $t in $input/titolo
            return titolo[$t/text()]
Q4($input): for $aut in $input/autore
            let $nome := $aut/nome
            return autore[$nome/text()]
Q5($input): for $art in $input/articolo
            return articolo[Q4($art), Q3($art),Q2($art)]
Q6($input): for $bib in /pisaBib
            return bib[Q5($bib)]
```

Fig. 5. Mapping from the Pisa schema into the Pisa view.

Pisa view in terms of the Pisa schema, the system has to *invert* this mapping, so to obtain a mapping from the view into the schema, and then to compose the newly obtained mapping with the query. This rewriting is performed by relying on standard algorithms for rewriting query over views [7, 9].

Once the query is locally executed, the system reformulates the query so to match New York view; this reformulation is performed by directly composing the query with the mapping from Pisa to New York, relying again on standard algorithms for query unfolding [7, 9]. To illustrate how these algorithms work, consider the first `for` clause of the query; the clause searches for articolo elements nested into elements bound to $*bib*. To reformulate this clause, the system matches the path with the mapping definition, hence discovering that the path is mapped by query Q_5. By looking into the chosen mapping fragment, the system knows that the path must be rewritten as `$bib/article`, and that the remaining paths must be reformulated by using queries Q_4, Q_3, Q_2, and Q_6.

At the end of the reformulation process, the reformulated query, shown in Figure 7, is then sent to the New York site; when this query arrives at New York, it is successfully executed since it is compatible with New York peer view.

```
articoli_Fernandez[
for $a in $bib/articolo,
    $aut in $a/autore
let $mf := ''Mary F. Fernandez''
where $aut() = $mf
return $a]
```

Fig. 6. Pisa user query.

Assume now that New York slightly changes the way author names are represented, and that this schema change reflects on the peer view; instead of a simple

```
articoli_Fernandez[
for $a in $bib/article,
    $aut in $a/author
let $mf := ‘‘Mary F. Fernandez’’
where $aut() = $mf
return $a]
```

Fig. 7. Transformed Pisa user query.

author element, detailed information about author's first name and second name are represented: `Author = author[first[String],second[String]]`.

Now, when the Pisa user runs again her query, she is not obtaining results from New York, since the rewritten query does not match the new view of New York. Unfortunately, New York site just returns an empty sequence as result of the query, so Pisa has no way to distinguish between the error and an unsatisfied predicate. By typechecking the transformed query against the new view, the system would inform Pisa that an error is present in the query, which implies that the mapping is corrupted.

3 Type System

The central point in the proposed approach is the use of a type system capable of capturing incoherences between the schema specification and both the twigs and the predicates contained in a query. By relying on this feature, our technique can identify discrepancies between the transformed query and the target peer view, and provide detailed information about them.

Our type system is capable of precisely identifying type-wrong fragments of a query produced by mappings. Hence, mappings involved in the production of that wrong part (corrupted mappings) can easily be identified, as, starting from a sub-query, it is possible to retrieve the mapping that has produced that part. Also, we provide type information about the part of the database that cannot be matched by the wrong query fragment. Starting from this type information, it is possible to find an alternative and correct definition of the identified corrupted mapping.

The type system is an extension of the one presented in [3]. For reason of space, here we formalize input/output and the main properties of the type system, and explain the role these properties play in the detection of p2p corrupted mappings (some main definitions of the type system are in Appendix **??**).

The proposed type system is designed to statically detect the presence of two kinds of errors: sub-query emptiness and wrong comparisons in the *where* clause. It differs from that of [3] in two key points: unlike [3], we extend the check for correctness to the *where* clause and introduce a proper treatment of conditions; moreover, error messages returned by the type system contain not only the wrong sub-query, as in [3], but also the types of the schema involved in the error. We provide this detailed information since it can be very useful during mapping maintenance or debugging.

Detecting sub-query emptiness means discovering, at static time, the presence of sub-queries that, at run time, will evaluate to the empty sequence, in each valid evaluation of the query (a query evaluation is valid if it is done wrt a variable substitution which is valid wrt types of variables in the substitution). Essentially, these sub-queries consist of path expressions that never match the input data, and, therefore, are not correct wrt to the types of input data. Detecting wrong where-comparisons, instead, means to discover, at static time, the presence of comparisons between values of different types

Discovering the presence of such errors is crucial to discover incompatibilities between the query structural requirements and input data structural specifications (input types). While type rules of the type system are omitted in this paper, these errors are characterized quite rigorously in terms of query semantics. This characterization is then used to measure the accuracy of the type analysis.

4 Distributed Typechecking Algorithm

The approach we are proposing is based on the idea of typechecking queries at each involved peer. Consider a query Q originated at peer p_i, and assume that the query traverses peers p_2, ..., p_n, hence being rewritten by the mapping chain $\{\rho_1, \ldots, \rho_{n-1}\}$ (ρ_i is the mapping from p_i to p_{i+1}); the sequence of peers to be traversed can be precomputed on the basis of a crawling process, or it can be determined at run-time (the proposed approach is orthogonal to the algorithms used for finding the mapping chain corresponding to a given query Q). Let ϖ be the function transforming queries according to a given mapping. When $\varpi_{\rho_1 \circ \ldots \circ \rho_{j-1}}(Q)$ arrives at peer p_j, p_j type-checks it against its peer view in order to verify that the query requirements are *compatible* with its view; if the type checker raises an error, then the system knows for sure that an error is present in the mapping ρ_{j-1} (the mapping from p_{j-1} to p_j), so it can warn p_{j-1} and p_j about the problem.

Thus, we assume that queries, once transformed, are type-checked by any peer they reach before executing them; since the query execution cost is dominated by the communication costs, the adoption of this strategy should not affect the whole execution cost.

The *distributed* type-checking algorithm is shown in Algorithm 1. At this time, the algorithm is not able to locate the portion of the mapping containing errors, even though we believe that this extension should not be too complex.

As shown by the algorithm, the identification of errors in the mapping chain can benefit the query processing algorithm, since it allows the system to dynamically bypass potentially dangerous mappings. This is performed by backtracking to peer p_{j-1} (if an error was found in the mapping from p_{j-1} to p_j), and by recomputing the remaining part of the mapping chain. By doing so, the system could avoid global problems coming from the propagation of errors in the mapping chain.

To illustrate the effectiveness of the proposed approach, we can consider again the scenario described in Example 2, and see what happens when the transformed query arrives at New York. When the New York peer receives the transformed query Q', it starts type-checking Q' against its peer view. The local type-checking algorithm stops during the control for the correctness of the *where* clause (Rule CompWrong of Appendix **??**), since the query tries to compare

Algorithm 1 Distributed type-checking algorithm

Main Algorithm (at peer $p = (id, db, \mathcal{T}, \mathcal{V}, \{\rho_{ij}\}_j)$**)**
type-check locally Q against p view ($\mathcal{V}$)
if error **then**
  send user a warning $(Q, \mathcal{S}, \mathcal{W}, E)$
  stop
else
  find a mapping chain for Q
  $chain = < \rho_i, p_i >_i$
  $Q_1 = \varpi_{\rho_1}(Q)$
  send(p_2,Q_1, typecheck, *chain*)
  wait for errors
end if
TypeCheckReceive(*peer*, Q', typecheck, *chain*)
typecheck locally Q' against *peer* view
if error **then**
  send(sender,"error", $(Q, \mathcal{S}, \mathcal{W}, E)$)
  stop
else
  $\rho_i = nextMapping(chain)$
  $Q'' = \varpi_{\rho_i}(Q')$
  send(p_{i+1},Q'', typecheck, *chain*)
  wait for errors
end if

a complex element with a base type value. The system, hence, notifies to Pisa and New York that an error in the sub-query `where $aut = $mf` wrt the type *Author* has occurred; by using this information, Pisa can update its mapping to New York and/or change the routing for the query. More in details, the β returned by the type-checker, pointing to the error `where $aut = $mf` and sent to Pisa, tells Pisa how to directly pick up the corrupted mapping (its easy to find which mapping is involved in the production of a β sub-query). While the not matched type *Author* tells Pisa how to modify the corrupted mapping.

More formally, we can say that if a peer p is notified of an error message $(Q, \mathcal{S}, \mathcal{W}, E)$, with, say, $(\beta.\alpha, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2))) \in \mathcal{W}$, then p can rely on a systematic technique that uses information in $(Q, \mathcal{S}, \mathcal{W}, E)$ in order to

- retrieve a quite precise set of candidate corrupted mappings which contains effective corrupted mappings;
- suggest, by using the type information in $\mathcal{S}, \mathcal{W}$, and E, a set of updated mappings, fixing the issues in the previous mappings.

This sheds light about the different roles of error-positions and not-matched types, both computed by our type-checker; the formal definition of the above mentioned systematic approach for retrieval-correction of corrupted mappings is the core of our current efforts in continuing this work, and also represents the formal connection between our type analysis technique and the p2p infrastructure.

5 Related Works

We are not aware of studies related to the use of the static analysis for discovering corrupted mappings in p2p systems. The advantages of our type analysis with respect to current version of W3C XQuery type system [4] has already been discussed in [3]. In a nutshell, our type system ensures complete type inference and error checking in most cases, while ensuring good performance at the same time. Being the definition of W3C XQuery type system still a work in progress, no results are known about the complexity of the whole system, and the system does not ensure completeness of path error-checking (see [3]). Moreover, for the moment, W3C type system does not contain mechanisms able to precisely locate the position of wrong sub-queries. We believe that this problem is not difficult to solve for an implementor of the W3C type system; however, from the experience we gained in our previous investigations, we understood that the problem is definitively not obvious.

We conclude by observing that, in order to have a powerful type-checking algorithm, in our setting we can benefit from the absence of many features that complicate typing in a full XQuery language, but are often unnecessary in p2p systems.

6 Conclusions

This paper presented a technique for automatically discovering corrupted mappings in peer data management systems. This technique is based on the use of a distributed type-checking algorithm that type-checks a query at any involved peer; by matching a (transformed) query against the peer view of a given peer, the algorithm can find inconsistencies in the mapping used for transforming the query, and, then, warn the user about problems in the mapping.

The distributed type-checking algorithm exploits a local type-checking algorithm, based on a type system extending the one of [3]; this new type system is able to capture errors in the query paths as well as errors in the *where* clause, and returns detailed information about the errors found in the query; in particular, the type-checker specifies the sub-query containing errors as well as the fragment of peer view against which the query failed.

As a future work, we plan to further extend the type system so to precisely identify the fragment of mapping containing errors, as well as to develop techniques for suggesting updated mappings: with this supplementary information, the site owner can easily and quickly fix the corrupted mapping. Furthermore, we plan to extend the fragment of XQuery covered by the type system as well as to shift to an XQuery-like type system.

References

1. Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. *First Monday*, 10(5), 2000.
2. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 241–251, 2004.

3. Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Types for path correctness of XML queries. In *Proceedings of the 2004 International Conference on Functional Programming (ICFP), Snowbird, Utah, September 19-22, 2004*, 2004.
4. Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, feb 2004. W3C Working Draft.
5. Enrico Franconi, Gabriel M. Kuper, Andrei Lopatenko, and Ilya Zaihrayeu. Queries and Updates in the coDB Peer to Peer Database System. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1277–1280, 2004.
6. Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor 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.
7. Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.
8. Luciano Serafini, Fausto Giunchiglia, John Mylopoulos, and Philip A. Bernstein. Local relational model: A logical formalization of database coordination. In Patrick Blackburn, Chiara Ghidini, Roy M. Turner, and Fausto Giunchiglia, editors, *Modeling and Using Context, 4th International and Interdisciplinary Conference, CONTEXT 2003, Stanford, CA, USA, June 23-25, 2003, Proceedings*, volume 2680 of *Lecture Notes in Computer Science*, pages 286–299. springer, 2003.
9. Igor Tatarinov and Alon Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD Conference*, pages 539–550, 2004.
10. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
11. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.