Interoperability on XML Data

Laks V. S. Lakshmanan[1⋆] and Fereidoon Sadri[2⋆⋆]

[1] UBC, Vancouver, BC, Canada, laks@cs.ubc.edu
[2] UNCG, Greensboro, NC, USA, sadri@uncg.edu

Abstract. We study the problem of interoperability among XML data sources. We propose a lightweight infrastructure for this purpose that derives its inspiration from the recent semantic web initiative. Our approach is based on enriching local sources with semantic declarations so as to enable interoperability. These declarations expose the semantics of the information content of sources by mapping the concepts present therein to a common (application specific) vocabulary, in the spirit of RDF. In addition to this infrastructure, we discuss tools that may assist in generating semantic declarations, and formulation of global queries and address some interesting issues in query processing and optimization.

1 Introduction

Interoperability and data integration are long standing open problems with extensive research literature. Much of the work in the context of federated databases focused on integrating schemas by defining a global schema in an expressive data model and defining mappings from local schemas to the global one. More recently, in the context of integration of data sources on the internet, the so-called global-as-view and local-as-view paradigms have emerged out of projects such as TSIMMIS [22] and Information Manifold (IM) [14]. All of these have primarily concerned themselves with relational abstractions of data sources. Recently, the advent of XML as a standard for online data interchange holds much promise toward promoting interoperability and data integration. But XML, being a syntactic model, in itself cannot make interoperability happen automatically. Two main challenges to overcome are: (i) data sources may model XML data in heterogeneous ways, e.g., using different nestings or groupings or interchanging elements and attributes, and (ii) sources may employ different terminology, a classic problem even in multi-database interoperability. While earlier approaches to integration can be extended to handle XML, they suffer from significant overhead of designing a commonly agreed upon global schema. Can we interoperate without this overhead?
Indeed, there are a few recent proposals that do overcome the need for a global schema – Halevy et al. [10] and Miller et al. [18]. We will discuss them in detail in Section 5. In a nutshell, both of these approaches rely on source to source mappings. One problem here is that requiring such mappings for all pairs is too tedious and cumbersome. In order to mitigate this, one can merely insist, as [10]

⋆ Supported by NSERC.
⋆⋆ Supported by NSF grant IIS-0083312.

does, that the graph of pairs with available mappings be connected. A second problem is that when a source s_i is mapped to source s_j, if s_j does not have some of the concepts present in s_i, then they will be lost. E.g., s_i may include the ISBN for all its books while s_j may not.
Independently of all this, there has been a lot of recent excitement around the *semantic web* [21] initiative, coming with its own host of technologies such as resource description framework (RDF) [20] and ontology description languages such as DAML+OIL, OWL, and XTM [8, 17, 23]. The promise of the semantic web is to expose the semantics of the information content of web resources (including text, audio, video, etc.) using common transparent vocabulary thus taking the web to a higher semantic level, enabling easy exchange of data and applications. Can we leverage these developments and create a lightweight scalable infrastructure for interoperability? We believe the answer is yes and take the first steps toward this creation here.
Our thesis is that each source('s administrator) interested in participating in data and application sharing should "enrich" itself with adequate semantic declarations. These declarations would essentially expose the semantic concepts present in the source using common, but possibly application specific, terminology, in the spirit of the semantic web initiative and frameworks such as RDF. Queries across data sources so enriched can be composed over this common vocabulary, being agnostic to the specific modeling constructs or terminologies employed in the local sources. A detailed example motivating these issues appears in Section 2. A suggested architecture for semantic enrichment and query processing is depicted in Figure 1. Other architectures such as peer to peer are also possible.

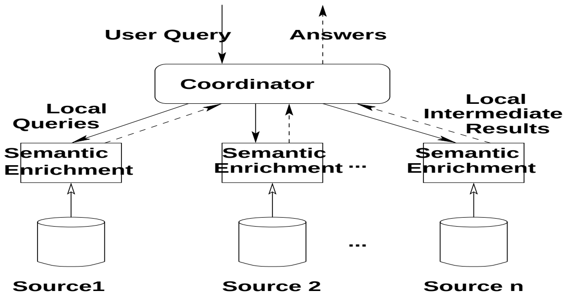

Fig. 1. An architecture for semantic markup and query processing.

We make the following contributions in this paper.

- We describe a lightweight infrastructure based on local semantic declarations for enabling interoperability across data sources (Section 3).
- These declarations are mapping rules that map data items in a source to a common vocabulary, inspired by RDF. Checking the validity of these mappings involves checking certain key constraints. We illustrate this issue and show how this checking can be done in general (Section 3).
- A user query, when translated against the data sources will in general lead to exponentially many inter-source queries and a linear number of intra-source

queries. The former are more expensive and their elimination/optimization is an important problem. We show under what conditions inter-source queries can be eliminated altogether. We also develop a generic technique for optimizing inter-source queries (Section 4).
– Optimizing inter-source queries requires us to infer key and foreign key constraints that hold for the predicates defining the vocabulary, given the source schema (e.g., XML schema or DTD) and the mappings. While previous work has addressed the derivation of key constraints, we develop an algorithm for inferring foreign key constraints (Section 4.3).

2 A Motivating Example

Consider a federation of catalog sales stores maintaining their merchandise information as shown in Figure 2.

```
<!ELEMENT store (warehouse*)>
<!ELEMENT warehouse (city, state, item*)>
<!ELEMENT item (id, name, description)>
<!ATTLIST warehouse id ID #REQUIRED>
```

DTD: Source1

```
<!ELEMENT store (items, warehouses)>
<!ELEMENT items (item*)>
<!ELEMENT item (id, name, description)>
<!ELEMENT warehouses (warehouse*)>
<!ELEMENT warehouse (city, state)>
<!ATTLIST item warehouse-id IDREFS #REQUIRED>
<!ATTLIST warehouse id ID #REQUIRED>
```

DTD: Source2

```
<!ELEMENT store (items,warehouses,inventory)>
<!ELEMENT items (i-tuple*)>
<!ELEMENT i-tuple (id,name,description)>
<!ELEMENT warehouses (w-tuple*)>
<!ELEMENT w-tuple (id,city,state)>
<!ELEMENT inventory (inv-tuple*)>
<!ELEMENT inv-tuple (item-id,warehouse-id)>
```

(Relational) **DTD: Source3**

Fig. 2. Schema of three sources.

Consider the query "For each state, list the state information and all (distinct) items available in warehouses in that state." How can we express this query? There are two sources of complexity in writing this query (say in XQuery):

– For each source, the user has to write source specific "access code" involving XPath expression specific to it.
– The query, in general, is *not* the union of three intra-source queries, as we must consider combinations of state/warehouse pairs in one source with warehouse/item pairs in others. As a result, the resulting query is a complex union of a large number of "joins" across sources in addition to the three intra-source queries.

In addition to the obvious burden on the user for composing such queries, without further knowledge, they are hard to optimize. *Note that even if we were to employ source-source mapping techniques of [10] or [18], the second of these difficulties does not go away.* The reason is that not every concept present in a source may have a counterpart in the source it is mapped to.
What we would ideally like is for the user to be able to write such queries easily, preferably by referring to some common vocabulary infrastructure created for the

application on hand. The infrastructure should alleviate the burden on the user of source specific access code writing. Besides, we want the query expression to be simple to compose and to comprehend. Finally, we would like query optimization to be handled by the system.

The main idea behind our approach is based on the observation that ontology specification frameworks such as RDF [20] and OWL [17] provide mechanisms for specifying metadata about a source that includes, not only metadata such as the author and creation date, but also the semantic concepts present therein. This is normally done by specifying concepts (*i.e.* classes or types of objects) and their properties (*i.e.* predicates describing relationships among objects). We will use the terms *property* and *predicate* interchangeably in this paper. E.g., suppose we have the following common vocabulary (properties) defined for describing catalog sales applications: [3]

`item-itemId`, `item-name`, `item-description`, `item-warehouse`,
`warehouse-warehouseId`, `warehouse-city`, `warehouse-state`

Each property takes two arguments where the arguments are required to play specific roles. In the spirit of RDF, the first argument is always a URI, which is interpreted as an identifier. The second argument can be a URI or a basic type. E.g., `item-name` takes two arguments where the first argument is the URI of an item whereas the second is a literal name string. The properties `item-itemId` and `warehouse-warehouseId` provide the mappings between the URIs and database-style identifiers of these objects. The roles of the (two) arguments of the other predicates is often self-evident from the predicate names. This is in line with RDF convention where a URI is associated with each subject, property (predicate), and value (or object). For lack of space, we do not show the RDF specs. In fact, in the spirit of semantic web, we may expect that in addition to the semantic marking up of each source, there may be additional ontological mappings (expressed in languages such as OWL [17]) that relate concepts in a class hierarchy as well as specify additional constraints on them. A final point is that an RDF-based semantic marking up for a large source can be tedious. Furthermore, storing the RDF marking up explicitly is redundant and wastes considerable space. To address these concerns, we can: (i) write a transformation program in, say XSLT that transforms an XML document into the required RDF markup specs, and (ii) make use of tools that assist in the writing of the XSLT program.

We expect users (local source administrators) will create such transformations or mappings with the aid of tools. In the paper, for simplicity, in place of the complex syntax of XSLT, we use a fragment of XPath together with predicates to express the mappings. An example mapping program for some of the properties of data sources of Figure 2 appears in Figure 3. We have used URI-generating functions f_I and f_W in the following mappings. A *URI-generating* function is a one-to-one function that takes an identifier of an object and generates a unique URI for that object. A simple scheme can be employed for URI generation. For

[3] In addition to the classes `item, warehouse, state`.

example, for an item id `$I`, f_I(`$I`) = `baseURI/item#$I`. This simplifies the task of determining whether two URIs correspond to the same object. This can be accomplished by stripping away the `baseURI` components, and comparing the extension for equal (or equivalent) object ids.

```
item-itemId((f_I($I),$I)         <- source1/store/warehouse/item/id $I.
item-name(f_I($I),$N)            <- source1/store/warehouse/item $X, $X/id $I, $X/name $N.
item-warehouse(f_I($I),f_W($W)) <- source1/store/warehouse $X, $X/item/id $I, $X/@id $W.
warehouse-state(f_W($W),$S)      <- source1/store/warehouse $X, $X/@id $W, $X/state $S.

item-itemId(f_I($I),$I)          <- source2/store/items/item/id $I.
item-name(f_I($I),$N)            <- source2/store/items/item $X, $X/id $I, $X/name $N.
item-warehouse(f_I($I),f_W($W)) <- source2/store/items/item $X, $X/id $I, $X/@warehouse-id $W.
warehouse-state(f_W($W),$S)      <- source2/store/warehouses/warehouse $X, $X/@id $W, $X/state $S.

item-itemId(f_I($I),$I)          <- source3/store/items/i-tuple/id $I.
item-name(f_I($I),$N)            <- source3/store/items/i-tuple $X, $X/id $I, $X/name $N.
item-warehouse(f_I($I),f_W($W)) <- source3/store/inventory/inv-tuple $X, $X/item-id $I, $X/warehouse-id $W.
warehouse-state(f_W($W),$S)      <- source3/store/warehouses/w-tuple $X, $X/id $W, $X/state $S.
```

Fig. 3. Mappings.

Global queries can now be formulated using these vocabulary predicates. No knowledge of local sources is needed for this task.

Example 1 (A global query). Revisit the query "For each state, list the state information and all (distinct) items available in warehouses in that state." For simplicity, we have assumed different sources are using the same domain for item ids (such as the manufacturer's item id), and the same URI-generating functions for each domain[4]. We can formulate this query as follows:

```
for $S in distinct-values(warehouse-state/tuple/state)
return <state> {$S}
  for $X in warehouse-state/tuple[state=$S]
      $Y in item-warehouse/tuple[warehouse=$X/warehouse]
      $Z in item-itemId/tuple[item=$Y/item]
  return
   <item> <id> {distinct($Z/itemId)} </id> <item>
  </state>
```

Note that the only information needed to formulate this query is the knowledge that the federation contains the vocabulary predicates – `item-warehouse` listing items and the warehouses for each item, `warehouse-state` listing warehouses and their respective states, and `item-itemId` relating URIs to database-style identifiers. Also note that, in general, each source in the federation may have none, some, or all of the predicates specified in the global query.
The rest, optimization and execution of query, is the responsibility of the system. We will discuss query optimization and processing in Section 4. ■

[4] In general, local sources may use their own id domains and URI-generating functions. In such cases, mappings (for example between local item ids and manufacturer's item ids) should be available to relate different id domains and URIs in order to make interoperability possible.

3 Local Source Declarations

The main idea behind our approach is to provide a simple yet powerful model for semantic markups for local information sources that makes interoperability possible. Any approach to interoperability and integration is ultimately dependent on standards, common vocabularies, and higher level declarations that relate vocabularies to each other. Our approach is based on *ontologies*. An application specific ontology specifies the classes (concepts) and predicates (properties) that are needed for the modeling of the application. For example, in our catalog stores application, the classes are `items`, `warehouses`, `cities`, `states`, etc., and properties are `item-itemId`, `item-name`, `item-description`, `item-warehouse`, `warehouse-warehouseId`, `warehouse-city`, and `warehouse-state`.
Where do ontologies come from? And aren't they a little like a global schema which we wanted not to have to assume? Firstly, currently there are several efforts underway toward developing just such ontologies [4] (also see the bibliography ontology in [16]). We envisage that groups of parties engaged in specific applications will converge to standard ontologies. Secondly, unlike the problem with global schema, we are *not* attempting to determine a commonly agreed upon global schema into which local source schemas can be mapped. Rather, the idea is that whenever a source/site wants to join an interoperability effort, it must map its data to the standard ontology that already exists. In other words, we get the global schema for free, as long as an application specific standard ontology exists.
In general, we expect ontologies containing the following information to be useful for interoperability:

- Property names and usages.
- Domain and range of each property. We will treat this as the two arguments of the corresponding predicate. For example, the property `item-warehouse` takes two arguments, which are URIs for item (domain) and warehouse (range), respectively.
- Key and cardinality information for predicates. For example, `person` is the key in `person-salary`, and `person-child` is [1-2]-to-n.
- Additional type information for arguments. For example, in the predicate `person-salary`, the type information for salary may specify additional information such as currency, and period (*e.g.* annual, monthly, bi-weekly).

If different terminologies are used in the declaration of the semantic contents of different sources, then higher-level declarations (typically domain level ontologies) relating the concepts and properties of different applications in the domain (which are used by different sources) are needed to make interoperability possible.

A local source exists in its native model (XML data, relational, spreadsheet, etc.). To make interoperability possible, the user (or local database administrator) should (1) choose the set of properties from a relevant ontology to model the source contents, and (2) provide *mappings* (or transformation rules) that map source data to the properties.

3.1 Mappings

Once the set of properties to model the contents of a source has been determined, mappings from XML data (or other source types) to properties should be established. The language we use to specify XML-to-property mappings is based on a (subset of) XPath and is similar to mapping languages (also called *transformation rules*) in the literature (For example, [9]). The mapping for a binary property p has the following form

```
p(f($X), g($Y)) <- path1 $G, $G/path2 $X, $G/path3 $Y.
```

where `$X` and `$Y` correspond to the arguments of p, `f($X)` is either \$X itself, or f_X`($X)`, where f_X is the URI-generating function for objects `$X` of type X, and similarly for `g($Y)`. The variable `$G` is called the *glue* variable, and is used to restrict the (`$X`,`$Y`) pairs to have the same `$G` ancestor element in the document.

We envision tools to assist users (or database administrators of local sources) in defining the mappings. The graphical interface displays the structure of the XML document as a graph. To specify the mapping for a property `p`, three nodes in the XML schema graph should be selected by the user to determine (1) and (2): Variables associated with each of the arguments, respectively, and (3): The glue variable. Normally, the node specifying the glue variable is the least common ancestor of the first two nodes that specify the arguments. Sometimes, gluing may be accomplished by equating values of elements related to the first two argument elements. The user should also specify whether URI-generating functions are applied to the first and/or second argument. To be consistent with RDF, we require the first argument to be a URI. Hence, if `$X` is not a URI, then a URI-generating function must be applied to it.

Example 2 (Mapping Tool). In the schema graph of source 1 (below) user selects a property `item-warehouse` from an appropriate ontology, and specifies the three nodes `id`, `@id`, and `warehouse` for argument1, argument2, and glue, respectively. He also specifies that URI-generating functions should be applied to the arguments. The following mapping is then generated by the tool.

```
item-warehouse(f_I($I),f_W($W)) <- source1/store/warehouse $X,
                                   $X/item/id $I, $X/@id $W
```

```
          store
            |*
        warehouse
     /  |    \   \*
   @id city state item
                 /  |  \
                id name description
```

Note that the type of the URI-generating functions is determined by type information contained in the ontology. For example, since the domain of `item-warehouse` (*i.e.*, its first argument) is of type `item`, then the URI-generating function for items, f_I, is applied to the first argument. Similarly, f_W, the URI-generating function for warehouses, is applied to the second argument. ■

The mapping tool can also assist the user by performing certain *type* and *role* checking on the arguments of the predicate, and verifying or deriving integrity constraints for the predicates. The extent to which the tool can succeed in these tasks depends on the richness of local source (native) declarations of types and constraints. For example, a rich declaration for XML data would use a powerful schema declaration language (such as XML Schema) and contain elaborate integrity constraints satisfied by source data.

Example 3. For the property `warehouse-state`, the ontology may specify that the first argument should be an identifier for warehouses, and should be the key for the property. These requirements can be verified by the tool if adequate declarations exist for the sources. For example, in the DTD of source 1, the `@id` attribute of `warehouse` is of type `ID`, hence it is an identifier for `warehouse`. As a result, f_W`($id)` is also an identifier for warehouse (recall that URI-generating functions are one-to-one). Alternatively, if XML Schema was used, the requirements could be derived from the following constraint[5] $(\epsilon, (source1/store/warehouse, (\{@id\})))$ using techniques of [9]. ■

We will study the problem of deriving foreign key constraints (or referential integrity constraints) for the properties from XML source schema specifications and mapping rules (Section 4.3). These constraints play a very important role in query optimization in this approach. Davidson et al. [9] have studied the problem of deriving or verifying functional dependencies from XML key constraints. Their algorithms can be used by the mapping tool for the verification of mapping correctness by checking argument roles and constraints that should be satisfied by the properties.

4 Query Translation, Processing, and Optimization

In this section, we discuss query specification and translation, and present algorithms for the optimization and efficient processing of queries. Recall that we use the terms *property* and *predicate* interchangeably. The latter is used predominantly in this section to emphasize the relational view of properties (in the database sense). Our simple model of binary predicates significantly simplifies the task of query formulation. The federation consists of local sources, where the content of each source has been declared by a set of predicates, together with data mappings to these predicates. Let's assume, for the time being, that all local declarations use the same ontology. We will relax this assumption later in Section 4.4. In this case, the "global schema" is simply the collection of all local predicates.

Given a predicate p, a source i may contain a fragment of p, which we denote by p_i. In general, fragments of the same predicate at different sources may contain overlapping data. We refer to the (conceptual) union of all fragments p_i of a predicate p as the *global* predicate p.

---
[5] Which says in the context of the whole document, attribute `@id` uniquely identifies a warehouse.

A query is formulated in terms of global predicates and predicate fragments. The query processor architecture is shown in Figure 1. A naive implementation approach is to materialize these predicates at the coordinator using the local mapping specifications, and then execute the query using materialized predicates. This approach will be very inefficient in general.

A better approach involves expanding the query by replacing each global predicate by the union of its fragments. The outcome, in general, consists of a collection of *local* (*i.e.*, intra-source) and *inter-source* sub-queries, the results of which should be combined to obtain the answer to the query. The coordinator is in charge of query expansion, query preparation and transformation (to local native schemas), and coordination of sub-query execution at different sources. It then collects results of the sub-queries and combines them to form the final answer.

A local sub-query is one where all the predicate fragments specified in the sub-query belong to the same source. A local sub-query can be rewritten in the native model of the source, and executed locally at that source. We will demonstrate this with an example below.

In an inter-source query, fragments from different sources are specified. We present various techniques for the translation, optimization, and processing of inter-source queries (Section 4.2). Processing these queries normally involves data transmission between sources so optimizing inter-source queries is important.

Example 4 (A local query). Let's consider the global query Q of Example 1. The local sub-query obtained from Q for source 1 is obtained by substituting predicate fragments `warehouse-state-1`, `item-warehouse-1`, and `item-itemId-1` for the corresponding predicates in Q.

The local query is then transformed by substituting its variable declarations by variable declarations obtained from mappings for source 1 (see Figure 3). We also make the observation that URIs were generated for the predicates (properties) to be consistent with RDF. When translating back to the native source data, we can use the object identifier instead of its corresponding URI. This follows from the fact that URI-generating functions are one-to-one. For example, `@id` is used instead of f_W`(@id)` in the transformation of our query. The resulting sub-query for source 1 is shown below.

```
for $S in distinct-values(source1/store/warehouse/state)
return <state> {$S}
  for $XG in source1/store/warehouse[state=$S],
      $YG in source1/store/warehouse[@id=$XG/@id]
  return <item> <id> {distinct($YG/item/id)} </id> <item>
  </state>
```

The last step for local queries is native query optimization, which may be carried out by the coordinator, the source, or both. In this example, the optimizer realizes `$XG` and `$YG` are the same, and eliminates one to get the following local query to be executed by the local source. Physical query optimization by a source follows this logical optimization.

```
for $S in distinct-values(source1/store/warehouse/state)
return <state> {$S}
  for $XG in source1/store/warehouse[state=$S]
  return
   <item> <id> {distinct($XG/item/id)} </id> <item>
  </state>
```

Algorithm 1 (Local sub-query generation for an XML source)
Input: Global query Q, source mappings m_i for source i
Output: Local sub-query Q_i at source i resulting from Q.
Method: The idea is to replace variable declarations of Q with variable declarations of source i using the mapping rules m_i. Certain details should be observed in this translation as follows:
(1) If a variable declaration `$X` in Q corresponds to a *tuple* of a predicate `p`, then replace it by the declaration for the *glue* variable in the mapping rule for `p` in m_i.
(2) If a variable declaration `$X` in Q corresponds to an *argument* of a predicate `p`, and the declaration also has a selection condition on the other argument, then replace it by the declaration for the *glue* variable in the mapping rule for `p` in m_i, and include the selection condition as well.
(3) If a variable declaration `$X` in Q corresponds to an *argument* of a predicate `p`, and rule (2) above does not apply, then replace it by the declaration for the argument obtained from the mapping rule for `p` in m_i.
(4) If a URI is used in Q, then Q_i will use its corresponding object id. (Note: we are assuming global queries do not ask for URIs. Otherwise, only for the output, Q_i applies the URI-generating function to the corresponding object id.)
(5) Variables declared in Q with respect to an `object-objectId` predicate do not need a counterpart declaration in Q_i if the `object-objectId` predicate is joined with another predicate `p` having the same `object` as one of its arguments. In such cases the declaration in Q_i corresponding to `p` can supply the required variable.

Example 5. Consider our running query Q of Example 1, and its local sub-query Q_1 of Example 4 (before the final local optimization step). Variables $X and $Y in Q correspond to step (1) of the algorithm. In Q_1 they are replaced by $XG and $YG (their corresponding glue variables). Query Q does not contain any variables corresponding to step (2). An example would be the following declaration: `for $X in warehouse-state/tuple[state=$S]/warehouse`. Variable $S in Q corresponds to step (3). The condition `...[@id=$XG/@id]` in $Q1$ is an example of step (4). It resulted from `...[warehouse=$X/warehouse]` in Q. Finally, as an example of step (5) of the algorithm, Q_1 does not have a variable that corresponds to `$Z` is Q. The output `$Z/itemID` has been replaced by `$YG/item/id` in Q_1. ■

4.1 When can Inter-source Processing be Eliminated?

A global query often involves more than one predicate. If there are n sources and the query involves m predicates, then the expansion of the query results

in $O(n^m)$ queries. Only $O(n)$ of these queries are local queries. The rest are expensive inter-source queries.
We are interested in characterizing cases where inter-source queries can be eliminated. If we can eliminate all inter-source queries, then we have reduced the number of queries to $O(n)$, instead of $O(n^m)$. In the remainder of this section we will study a simple case where the global query involves the joining of two binary predicates. Results of this section can be extended to more general cases.

Let the global query involve the join of predicates $p(A, B)$ and $q(B, C)$ on argument B. Our Example 1 was of this type, joining `item-warehouse` and `warehouse-state` on the `warehouse` argument. In general, predicate fragments from different sources contain overlapping data. Often a `distinct` clause must be used in the query to deal with multiplicities that result from this data duplication.

Consistency Condition: We say a predicate p satisfies the *consistency condition*, provided: (i) whenever the constraint $p(\alpha)\rightarrow p(\beta)$ holds in p, then it holds for every fragment p_i; (ii) if for fragments p_i and p_j we have $t_i \in p_i$, $t_j \in p_j$, such that $t_i(\alpha) = t_j(\alpha)$, then $t_i(\beta) = t_j(\beta)$.

The following theorem characterizes the cases where inter-source joins are not needed. Let Q be a query involving the join of $p(A, B)$ and $q(B, C)$, and suppose Q has the `distinct` clause to eliminate duplicates. Let k be a source.
Theorem 1. Suppose the key constraint $B\rightarrow C$ holds in q. No inter-source processing involving p_k is needed if and only if the following conditions hold: (1) The consistency condition holds for q, and (2) There is a foreign key constraint from p_k to q_k on the attribute B.

Proof. (*if*) We should show that any tuple in the join of p_k with q_i, $i \neq k$, is also in the join of p_k with q_k. Let $(a, b, c) \in p_k \bowtie q_i$. Then $(a, b) \in p_k$ and $(b, c) \in q_i$. By the foreign key constraint, there exists a tuple $(b, c') \in q_k$, for some c'. By the consistency requirement, and since q satisfies $B\rightarrow C$, we have $c' = c$, hence $(b, c) \in q_k$. It follows that $(a, b, c) \in p_k \bowtie q_k$.
(*only if*) We should show that if any of the conditions does not hold, then $p_k \bowtie q_i$, $i \neq k$, can contain a tuple that is not in $p_k \bowtie q_k$ (nor in any other $p_k \bowtie q_j$, $j \neq k$ and $j \neq i$). First, assume the consistency condition does not hold. Then, p_k and q_i can contain tuples (a, b) and (b, c), respectively. But q_k can have a tuple (b, c'), $c \neq c'$. Similarly, other q_j's may contain $(b, c")$, $c \neq c"$, or contain no tuple t_j where $t_j(A) = a$. Hence, the tuple (a, b, c) is only in $p_k \bowtie q_i$. The case when q_k does not satisfy $B\rightarrow C$ is similar. Finally, if the foreign key constraint from p_k to q_k does not hold, then p_k and q_i can contain tuples (a, b) and (b, c), respectively. But other q_j's, $j \neq i$ (including q_k) may contain no tuple t_j where $t_j(A) = a$. Again, the tuple (a, b, c) is only in $p_k \bowtie q_i$. ■

We should emphasize that in this theorem we are assuming *only local information in the form of key and foreign key constraints, plus a consistency condition which is basically a statement of correctness of data across sources.* If other forms of constraints are permitted, the theorem can be extended by relaxing the conditions. An extension of the theorem is given below.

In Section 4.3 we discuss how referential integrities for predicates (properties) can be derived from local source native XML specifications and the predicate mappings. We also present an example (Example 6) in that section demonstrating that conditions of Theorem 1 are satisfied by source 1 for our running query example.

Relaxing the conditions of Theorem 1

The first condition of Theorem 1 states that the argument B of $q(B, C)$ is the key for q. There are cases where the argument is not a key, but, semantically, a functional relationship exists. Consider, for example, the predicate `person-child`. It has two arguments. The first argument is a URI for person. The second argument is also a person URI (for a child of the first argument). The constraint that should hold is: Each person (URI) determines the *set* of his/her children (URIs). A relaxed consistency condition states the equality of the *sets* of associated objects:

Relaxed Consistency Assumption: We say a predicate p satisfies the (relaxed) consistency condition $p(\alpha) \rightarrow [p(\beta)]$ if for every pair of fragments p_i and p_j of p and $a \in p_i(\alpha) \cap p_j(\alpha)$ we have $\{t_i(\beta) \mid t_i \in p_i \text{ and } t_i(\alpha) = a\} = \{t_j(\beta) \mid t_j \in p_j \text{ and } t_j(\alpha) = a\}$.
The relaxed version of Theorem 1 is as follows.

Theorem 2. Let $p(A, B)$ and $q(B, C)$ be the predicates, and k be a source. If the following conditions hold: (1) The relaxed consistency condition $B \rightarrow [C]$ holds for q, and and (2) There is a referential integrity constraint from p_k to q_k on the attribute B.
Then no inter-source joins involving p_k is needed.
Proof. Proof is similar to the proof of Theorem 1 and is omitted. ■

4.2 Efficient Inter-source Processing (When Needed)

Let's consider a query involving the join of predicates p and q, and concentrate on the inter-source processing of $p_i \bowtie q_j$, $i \neq j$. We will discuss several approaches. The optimization process, in general, will need a *cost-based* approach which estimates the cost of different approaches and selects the best.

A Naive Algorithm. A naive approach for calculating $p_i \bowtie q_j$ is to ship one complete document to the other site, and then proceed as before by transforming variable declarations of the query to variable declarations on the documents (that are now locally available) and executing the query locally.

Algorithm 2. A more promising approach for calculating $p_i \bowtie q_j$, $i \neq j$ is to first materialize one of the predicates locally and send it to the other site. Then only transform query variable declarations on the second predicate to its corresponding document, and execute the query (using the first predicate and the second document). For example, p_i can be materialized at site i and shipped to site j. Then query is transformed to one that uses predicate p_i and the document at site j. This approach may be more efficient than the first approach since the volume of data transferred may be significantly lower (predicate p_i versus the whole document at site i.) ■

If certain conditions hold, we can use the *semi anti-join* approach discussed below in the calculation of inter-source joins. The idea is to *avoid* generation of duplicate answers, thereby further optimizing query processing.

Assume the global query involves join of predicates $p(A,B)$ and $q(B,C)$. Conditions under which semi anti-join approach can be used in the calculation of inter-source joins are: (1) The global query has a `distinct` qualifier, and (2) The key constraints $A \rightarrow B$ and $B \rightarrow C$ and the consistency condition hold for p and q. (This condition can be reduced to relaxed consistency conditions $A \rightarrow [B]$ and $B \rightarrow [C]$ for p and q.)

The semi anti-join technique: When calculating inter-source join $p_i \bowtie q_j$ as a step towards calculating $p \bowtie q$, proceed as follows:
(1) Let the partial result of $p \bowtie q$ calculated so far be r. Ship $r[A]$ to site i.
(2) At site i, generate $p'_i = \{t \mid t \in p_i, t(A) \notin r[A]\}$.
(3) Ship p'_i to site j.
(4) Calculate $p'_i \bowtie q_j$.
(5) Combine the result with r to obtain the updated partial result.

We also present a variation on the semi anti-join algorithm that may be more efficient in some cases.

The maintenance of the partial result in the above algorithm may be expensive to the extent of making the overall join calculation sub-optimal. In the following version of the algorithm each source maintains its own partial result. The initial partial result at site i is $p_i \bowtie q_i$ which is calculated locally.

The modified semi anti-join technique: Let r_i be the (local) partial result at site i. To calculate $p_i \bowtie q_j$:
(1) At site i, generate $p'_i = \{t \mid t \in p_i, t(A) \notin r_i[A]\}$.
(2) Ship p'_i to site j.
(3) Calculate $p'_i \bowtie q_j$.
(4) Combine the result with r_j to obtain the updated partial result at site j.

4.3 Deriving Referential Integrities from XML schema and constraints

Theorem 1 can be used for query optimization to determine inter-source joins that can be avoided. To apply this theorem we need to know key and foreign key constraints on the predicates of each source. Results of [9] can be used to derive key constraints for predicates from schema specification and key constraints of the underlying XML data. In this section we study the problem of deriving referential integrity and foreign key constraints for the predicates.

Consider two binary predicates, p and q, with the following mappings. `$G` and `$H` are the glue variables of each rule, respectively.

```
p($X, $Y) <- path1 $G, $G/path2 $X, $G/path3 $Y.
q($Z, $W) <- path4 $H, $H/path5 $Z, $H/path6 $W.
```

Theorem 3 (referential integrities). If the following conditions hold for $p(A,B)$ and $q(B,C)$, then there is a referential integrity from p to q on B.

1. `path1/path3 = path4/path5`. Further, the element associated with `path1/path3` (or `path4/path5`) is declared by the source schema declaration to have non-null values.
2. The element associated with `path4/path6` is declared by the source schema declaration (*i*) to be required and (*ii*) to have non-null values. ■

Example 6. Let's revisit Example 1. This query involves the join of predicates `item-warehouse` and `warehouse-state`. Let's consider source 1. If we can verify the conditions of Theorem 1 for the fragments of these predicates at source 1, we can eliminate all inter-source joins involving `item-warehouse-1`. The first condition, namely $B \rightarrow C$ in `warehouse-state-1`(B, C), was verified in Example 3. The second condition, namely foreign key constraint from `item-warehouse-1` to `warehouse-state-1`, can be verified using Theorem 3. The mappings for these predicates were given in Section 2. From the DTDs in Figure 2, it is easy to verify that `path1/path3` = `source1/store/warehouse/@id` = `path4/path5`, and `@id` is a required attribute. For Condition 2, we notice `path4/path6` = `source1/store/warehouse/state`, and `state` element is also required.
If the conditions of Theorem 1 can also be verified for sources 2 and 3, then no inter-source processing at all is needed and query can the whole query can be processed by computing local queries and then combining the results. ■

4.4 Case of Multiple Ontologies

Up to this point, we had assumed local declarations used the same ontology for all sources. In this section, we relax this assumption and investigate the case where multiple ontologies are used in local declarations. We show how this affects query specification and translation, and argue that our query optimization techniques remain largely unchanged under the new model.
It has been argued that Description Logics (DL) provide the mathematical basis for ontology languages. In fact, OWL itself has been inspired and influenced by DL [3, 7]. We will use DL in our discussions rather than the more verbose RDF or OWL specifications. An ontology specification, modeled as a DL theory, contains a set of base concepts and properties. New concepts and properties can be declared using base or already declared concepts and properties using operators of the DL. A concept is interpreted as a set of objects (belonging to, or having the type of, that concept), and a property is interpreted as a relationship among its underlying concepts. Several DLs have been proposed for ontology specifications, such as $\mathcal{DLR}$ and $\mathcal{SHIQ}$ [3, 7], which differ in their set of operators and declaration power. The main goal has been to identify the language that provides the desired functionality, without becoming computationally undecidable or infeasible. DLs also provide mechanisms to declare concept hierarchies (taxonomies) and constraints.
When multiple ontologies are used in local declarations, providing interoperability is only possible if we are able to relate concepts and properties of different ontologies. This can be achieved if higher-level declarations relating ontologies are available, or through the use of tools that discover relationships among ontologies and produce the higher-level declarations.

There are recent investigations into the problem of *integration of ontologies*. It has been argued that a mechanism similar to the view mechanism in database systems is required for ontology integration in the general case [7]. In this section, we will concentrate on simple cases of ontology relationships that occur frequently in practice. More complicated cases need further investigation.
Given two ontologies O_1 and O_2, we study the case where concepts and properties of the two ontologies are related by *equivalence* and *sub-concept/sub-property* relationships, denoted by the symbols $\doteq$ and $\dot{\sqsubseteq}$. We say concepts (or properties) C_1 and C_2 in O_1 and O_2 are equivalent, $C_1 \doteq C_2$, if they are semantically the same concept (property). Similarly, $C_1 \dot{\sqsubseteq} C_2$ if C_1 is semantically a subconcept (subproperty) of C_2. The usual axioms apply to $\doteq$ and $\dot{\sqsubseteq}$, namely, (1) $\dot{\sqsubseteq}$ is reflextive, transitive, and antisymmetric, and (2) $C_1 \doteq C_2$ *iff* $C_1 \dot{\sqsubseteq} C_2$ and $C_2 \dot{\sqsubseteq} C_1$. It is important to note that $C_1 \dot{\sqsubseteq} C_2$ does *not* mean the extension of C_1 in a specific source is a subset of the extension of C_2 in another source. It merely states a semantic relationship between concepts across ontologies.

Recall that when a single ontology was used for all local declarations, we interpreted a global property p as the union of its fragments p_i for each source i. We extend this view to the case of multiple ontologies related by $\doteq$ and $\dot{\sqsubseteq}$ relationships as follows. Let p_i be a property at source i. Concept/property p_i is from ontology O_i. We interpret the *global* property p as the union of all properties that are equivalent to or subproperty of p_i, namely, $p = \bigcup\{q_j \mid q_j \doteq p_i \text{ or } q_j \dot{\sqsubseteq} p_i\}$. In this way, global queries can use a language that is the combination of the ontologies in the federation. Any symbol from any ontology can be used in the formulation of the query, and it is interpreted according to the semantics discussed above. Translation of queries to local and inter-source sub-queries is also a simple extension of the case for single ontology: The source-j fragments of a predicate p in the query are q_j's where $q_j \doteq p$ or $q_j \dot{\sqsubseteq} p$. Note that, *unlike the case for single ontology, here a source may have more than one fragment corresponding to a concept or property.* For example, a source may have properties `student-salary` and `staff-salary` which are sub-properties (and, hence, fragments) of `person-salary`. The first step of the translation algorithm replaces each property in the query with the union of its fragments, and generates a set of local and inter-source queries. Once this is done, the translation of sub-queries to local (native) models and their optimization can proceed as discussed in previous sections for the single ontology case.

5 Related Work

Data integration has received significant attention since the early days of databases. Most approaches to data integration involve *deriving* a global schema, which integrates local schemas. Then mappings are provided between the global schema and local schemas. There are two general approaches: *local as view* where local sources are considered as views of the global; and *global as view* where rules are provided to derive global data from local sources. There are a number of excellent surveys and discussions on data integration and related topics [5, 11–13, 19].

A more recent approach attempts at providing data sharing by mapping sources to each other. These approaches eliminate the need for deriving the global schema, and hence eliminate a substantial overhead in system design and creation. The approach of [10] is particularly attractive as it does not require schema mappings between every pair of sources. Rather, schema mappings can be *composed* to derive new mappings. Hence, it suffices for a source to provide a mapping to one other source, preferably the one closest to its structure. Every source-pair needs to be connected by a sequence of these peer-to-peer mappings. This approach works well when sources have more or less the same information content. Otherwise some concepts in a source may not have counterparts in another source, and the mapping will lose these concepts. The loss will be further propagated by mapping compositions. The Clio project has made significant contributions to the problems of data and schema integration from various source types (legacy, relational, object-oriented, semistructured) [2, 15, 18]. Their approach is based on providing tools to map any two schemas to each other, and hence it can be used for source-source as well as source-global approaches. Amann et al. [1] propose mappings of XML sources into a global schema using the local as view approach and propose algorithms for query rewriting and optimization.

Our approach differs from most of the previous approaches in that we provide simple but powerful declarations that are local to each source. Global interoperability is then obtained by using the predicates (properties) declared for each source.

Effective query optimization in our approach depends on rich schema specification and consistency rules declaration for local sources, and the ability to derive key and foreign key (or referential integrity) constraints for the predicates. There has been a lot of recent interest in the declaration of constraints for XML data, and derivation of constraints when XML data is stored as relational. Buneman et al. study the question of constraints declaration for XML [6]. The problem of propagating XML key constraints to relational is studied in [9]. They provide algorithms to determine, given XML key constraints and XML to relational transformation rules, whether a given FD is implied by the XML key constraints. We can utilize these results, as well as our results in Section 4.3, to derive key and foreign key constraints that are used in query optimization.

6 Conclusion

The vision behind our project is to create a lightweight scalable infrastructure for interoperating over XML data sources. Thereto, we presented an approach that derives its inspiration from the semantic web initiative, specifically RDF and OWL, to semantically mark up the information contents of data sources using application specific common vocabulary, by means of mappings, specified in a language like XSLT. User queries are expressed over this vocabulary and are translated into source specific access code and optimized. We addressed issues of eliminating or optimizing the costly inter-source queries as well as inference of integrity constraints over the vocabulary predicates, across the mapping rules. We are currently implementing a prototype system incorporating these ideas.

Our ongoing work addresses various exciting technical challenges arising from this investigation.

References

1. B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying XML sources using an ontology-based mediator. In *Proceedings of International Conference on Cooperative Information Systems*, pages 429–448, 2002.
2. P. Andritsos, R. Fagin, A. Fuxman, L. M. Haas, M. A. Hernández, C.-T. Ho, A. Kementsietsidis, R. J. Miller, F. Naumann, L. Popa, Y. Velegrakis, C. Vilarem, and L.-L. Yan. Schema management. *IEEE Data Engineering Bulletin*, 25(3), 2002.
3. F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In D. Hutter and W. Stephan, editors, *Festschrift in honor of Jörg Siekmann*, Lecture Notes in Artificial Intelligence. Springer, 2003. To appear.
4. T. Berners-Lee and E. Miller. The semantic web lifts off. *ERCIM News*, 51, pp. 9–11, 2002.
5. P. A. Bernstein. Applying model management to classical meta data problems. In *Proceedings of Conference on Innovative Data Systems Research*, 2003.
6. P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W.-C. Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002.
7. D. Calvanese, G. D. Giacomo, and M. Lenzerini. A framework for ontology integration. In *Proceedings of Semantic Web Working Symposium*, 2001.
8. The DARPA agent markup language (DAML). `http://www.daml.org/`.
9. S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML constraints to relations. In *Proceedings of IEEE International Conference on Data Engineering*, 2003.
10. A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhaven, L. McDowell, and I. Tatarinov. Crossing the structure chasm. In *Proceedings of Conference on Innovative Data Systems Research*, 2003.
11. A. Y. Halevy. Answering queries using views. *The VLDB Journal*, 10(4):270–294, 2001.
12. *IEEE Data Engineering Bulletin*, 25(3), 2002. Special issue on Integration Management.
13. M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 233–246, 2002.
14. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of International Conference on Very Large Databases*, pages 251–262, 1996.
15. R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1), 2001.
16. E. Mena. `http://sol1.cps.unizar.es:5080/OBSERVER/ontologies.html`.
17. OWL web ontology language. `http://www.w3c.org/TR/owl-features`.
18. L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *Proceedings of International Conference on Very Large Databases*, 2002.
19. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.
20. Resource description framework (RDF). `http://www.w3c.org/RDF/`.
21. Semantic web. `http://www.w3c.org/2001/sw/`.
22. The TSIMMIS project home page. `http://www-db.stanford.edu/tsimmis/tsimmis.html`.
23. XML topic maps (XTM). `http://www.topicmaps.org/xtm/`.