Mapping-Driven XML Transformation*

Haifeng Jiang[1] Howard Ho[2] Lucian Popa[2] Wook-Shin Han[3] †
[1,2]IBM Almaden Research Center, San Jose, California
[3]Dept. of Computer Engineering, Kyungpook National University, Korea
[1]jianghf@us.ibm.com, [2]{ho, lucian}@almaden.ibm.com, [3]wshan@knu.ac.kr

ABSTRACT

Clio is an existing schema-mapping tool that provides user-friendly means to manage and facilitate the complex task of transformation and integration of heterogeneous data such as XML over the Web or in XML databases. By means of mappings from source to target schemas, Clio can help users conveniently establish the precise semantics of data transformation and integration. In this paper we study the problem of how to efficiently implement such data transformation (i.e., generating target data from the source data based on schema mappings). We present a three-phase framework for high-performance XML-to-XML transformation based on schema mappings, and discuss methodologies and algorithms for implementing these phases. In particular, we elaborate on novel techniques such as streamed extraction of mapped source values and scalable disk-based merging of overlapping data (including duplicate elimination). We compare our transformation framework with alternative methods such as using XQuery or SQL/XML provided by current commercial databases. The results demonstrate that the three-phase framework (although as simple as it is) is highly scalable and outperforms the alternative methods by orders of magnitude.

Categories and Subject Descriptors

H.2.m [**Information Systems**]: Database Management—*XML Data Management*

General Terms

Algorithms, Performance

Keywords

Schema Mapping, XML Transformation

1. INTRODUCTION

Transforming data from one format to another is frequently required in modern information systems and Web applications that need to exchange or integrate data. As XML becomes the *de facto* standard for data exchange among applications (over the Web), transformation of XML data also becomes increasingly important. XML-to-RDB (known as XML shredding) and RDB-to-XML (known as XML publishing) are special cases of XML-to-XML transformation.

Writing data transformation programs manually (even in high-level languages such as XQuery, XSLT or SQL/XML [8], which is a SQL extension for publishing tables as XML) is often time-consuming and error-prone. This is because a typical data transformation task may involve restructuring, cleansing and grouping of data, and implementing such operations can easily lead to large programs (queries) that are hard to comprehend and often hide the semantics of the transformation. Maintaining the transformations correctly, for example as database schemas evolve [22], can also be a potential problem. As a result, it is desirable to have tools to assist such data transformation tasks.

We have recently seen research aiming to provide high-level mapping languages and more intuitive graphical user interfaces (GUI) for users to specify transformation semantics in convenient ways. One of the earliest examples in this direction is our own[1] Clio system [18, 19, 14] that can be used to create mappings from a source schema to a target schema for data migration purposes. The tool includes a *schema matching* component whose role is to establish, semi-automatically, matchings between source XML-schema elements and target XML-schema elements. In a second phase, the *schema mapping* phase, the Clio system generates, also semi-automatically, a set of *logical constraints* (or logical mappings) that capture the precise relationship between an instance (or document) conforming to the source schema (the input to the transformation) and an instance (or document) that conforms to the target schema (the output of the transformation). Another example of a system that is focused on the high-level specification and generation of data transformation and data integration applications is Rondo [17], a generic platform for managing and manipulating models. As in Clio, mappings are specified by using logical constraints. Other examples include Piazza [15] and HePToX [3], which are also based on mappings but focus on query rewriting for data integration, instead of data transformation. In addition to the research prototypes, many industry tools such as IBM Rational Data Architect (with Clio technology inside), Microsoft ADO.NET (ER-to-SQL mapping system) and Stylus Studio's XML Mapper support the development of mappings.

The aforementioned research on schema mappings solves the problem of *specifying* the transformation semantics. However, the problem of correctly and efficiently *executing* such mapping-driven data transformations still remains. In this paper, we propose a three-phase framework for modelling the physical data transformation implied by schema mappings and describe efficient algorithms for implementing such data transformation.

The mapping-driven transformations that we address are focused

*This work was funded in part by the U.S. Air Force Office for Scientific Research under contract FA9550-06-1-0226.
†Wook-Shin Han was supported in part by MIC & IITA through IT Leading R&D Support Project in Korea.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

[1]IBM Almaden and University of Toronto.

on data restructuring and are schema based. As such, many of the more advanced querying (selection) features of XQuery/XSLT (the variety of XPath axes, complex predicates such as universal quantification and negation, etc.) are not always needed. In fact, there is a greater emphasis on a set of operations that include shredding (of XML data into relations), reformatting and scalar transformation functions (i.e., those with a single return value), duplicate elimination, nesting and hierarchical merge. Such operations are essential in applications that integrate data from multiple sources and are more reminiscent of ETL (Extract, Transform and Load) than of typical querying.

We show that with a very simple framework that essentially groups these basic operations into three stages (extract-transform-merge, or ETM) one can achieve relative simplicity in the transformation code along with good performance. We show that one of the key challenges in XML-to-XML transformation arises when the target data must be generated as a hierarchy with multiple levels of grouping. This task is further complicated if data is coming from multiple data sources and must be merged. We give two novel and relatively simple algorithms, one based on hashing (using internal memory) and one based on sorting (in external memory), that can deal with such complex restructuring of data even with increased number of levels of grouping and with large amounts of data.

We also compare the transformation engine with the alternative of generating queries (XQuery, XSLT or SQL/XML) that implement the schema mappings. In fact, the Clio system itself includes query generation components for these query languages. Our experience shows that escaping to these languages is fine for data that is not very large or when the transformation does not require significant restructuring. However, the current XQuery, XSLT and SQL/XML systems are not (yet) tailored for the task of complex and large scale data transformations. These languages lack the support for a flexible deep-union operation, which can hierarchically merge XML trees. Even for the simpler operation of removal of duplicate trees, in XQuery, one has to use the `distinct-values` scalar-function in a carefully crafted way so that the structure of the data is not lost (see, as an example, query `Q4` in the XQuery Use Cases [23]). If the data to be merged is coming from multiple sources, the resulting query is further complicated. The size of an XQuery query that correctly implements a schema mapping can, in fact, be prohibitive (from both the performance and usability point of view). The situation is not much better for XSLT or SQL/XML.

Our experiments show that using our transformation engine has significant performance benefits over the use of queries (even on commercial databases). One interesting research direction is investigating the applicability to XQuery, XSLT or SQL/XML of the transformation techniques (e.g., for merging) that we have implemented. One key challenge is identifying what constitutes a merge "pattern" inside generic queries.

Our approach actually has analogies from the relational-database field. They are the commercial ETL tools including IBM Information Server, Informatica and Oracle Warehouse Builder. These tools often provide in-house transformation engines rather than translate whole ETL workflows into SQL queries.

1.1 Main Contributions

We model mapping-based XML transformation as a three-phase process: extract, transform and merge (ETM). For each phase, we use tailored algorithms to implement it:

- **Extract**. The extraction phase extracts mapped values from data sources. For XML data, we present a simple yet efficient streamlined algorithm for matching set-path expressions that extract atomic values in XML data into flattened tuples. When the source resides in relational tables, we try to leverage the relational databases and generate SQL queries for value extraction.
- **Transform.** This phase takes one flat tuple at a time and generates an XML fragment based on the mapping semantics. The transformation can be very flexible in that multiple fields can be merged into a single target field (e.g., merging of first name and last name with a space or comma in-between), and a single field can be split into multiple target fields (e.g., splitting of names into first name and last name). In addition, the transformation can be applied with any data reformatting or data cleansing function. For example, one can easily reformat a date "1/2/06" to "January 2, 2006" or "February 1, 2006", depending on the value of another input parameter, say a "country" field.
- **Merge.** The merging phase groups the resulting XML fragments (from the second phase) based on schema hierarchy, by default, and, more generally, based on application-specific grouping conditions. This phase requires aggregating and merging of large amount of data. We discuss both hash-based and scalable, sort-based, algorithms.

The transformation framework has been implemented with Java and is fully functional. It can readily be plugged into existing schema mapping tools. We shall use Clio as the reference schema-mapping system throughout the paper. However our work can be applied for all schema mapping tools that use a similar language to describe mappings as Clio.

As already mentioned, the experimental results confirm that our approach is sound and has significant performance benefits over the generated queries running on commercial databases or freely available XQuery processors.

1.2 Paper Organization

The rest of the paper proceeds as follows. In Section 2, we give an overview of Clio mappings and the three-phrase transformation framework. We then sketch the algorithms for implementing the three-phrase process in Section 3 and Section 4. In Section 5, we show how one can use XQuery queries (or SQL/XML queries, when the source is relational) as an alternative to running the mapping-based transformation engine. Section 6 presents our experience on the performance of the transformation engine and compares with the alternative of executing queries. We discuss related work in Section 7 and we conclude in Section 8.

2. TRANSFORMATION FRAMEWORK

Tools such as Clio help a user create schema mappings from source to target schemas. The transformation framework that we are going to present takes source data (that conforms to source schemas) and the mappings as input, and executes the mappings to generate target data (that conforms to target schemas).

We describe Clio mappings in Section 2.1 and sketch the three-phase framework with running examples in Section 2.2.

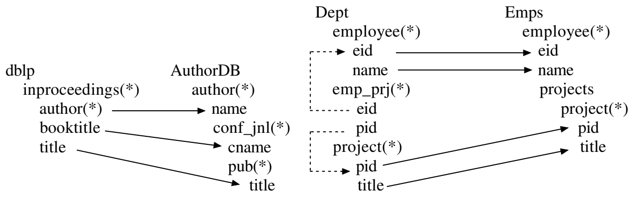
2.1 Mappings in Clio

Clio [14] is a schema-mapping system for discovering data transformation queries among different data sources. It provides a GUI for users to specify matchings (or correspondences) between source and target schemas and to establish accurate mapping semantics in an interactive way. Clio was initially designed to work for relational schemas [18]. Popa et al. extended it in [19] so that the source, the target, or both, can be XML.

In a nutshell, the schema mappings in Clio can be represented as a set of tgds (tuple-generating dependencies) [9]. We explain tgds with two examples.

EXAMPLE 1. *Consider the scenario described graphically in Figure 1(a), where we map from the DBLP schema to an AuthorDB schema. DBLP is a collection of computer science bibliography entries published in XML format*[2]*. (We use a simplified version here.) The structure of DBLP data is relatively flat. In contrast, AuthorDB presents a more hierarchical view of the data, where entries can be grouped based on author names and conferences. In Clio and in this paper, we use a simple nested representation of XML (and relational) schemas that abstracts away the details that are not essential. Repeatable elements are marked with '*'. We do not require sibling elements in a source document to exactly match the ordering given in the schema (e.g., we could have* `title` *appear before* `author`*). The leaf nodes are atomic elements that can be mapped. In particular,* correspondences *between such atomic elements in the source and in the target can be entered by a user in Clio's GUI (or, alternatively, can be inferred by an automated schema matcher). Based on such correspondences, Clio can generate a more precise mapping semantics. For our example, this semantics can be expressed as the following tgd:*

l_1 : *for* i *in* *dblp.inproceedings*, a *in* i*.author*
exists a' *in* *AuthorDB.author*, c *in* a'*.conf_jnl*, p *in* c*.pub*
where a'*.name* = a *and* p*.title* = i*.title* *and* c*.cname* = i*.booktitle*



(a) From DBLP to AuthorDB (b) From relations to XML

Figure 1: Two example mappings

A tgd is nothing but a constraint between the source and the target schemas that expresses what kind of target data should exist (see the exists clause and its associated where clause) given a specific pattern of source data (satisfying the for clause and its associated where clause, if any). In a tgd, variables are used to range over repeatable elements, while a simple projection operator ('.') is used to navigate through non-repeatable elements.

We also use the term "logical mapping" to refer to one tgd. In the previous example, one logical mapping was enough. However, in general, a schema mapping may consist of multiple logical mappings. For example, if a source schema contains two repeatable elements on the same level and there is no referential constraint to relate them, Clio would generate two separate logical mappings (simulating a union) to cover the two elements, respectively. The next example illustrates a different case where we map repeatable elements that are connected by referential constraints, but we still need multiple logical mappings (simulating an outer-join). The exact mechanism of how Clio generates tgds from user inputs is outside the scope of this paper. We refer the interested reader to [9] for detail.

EXAMPLE 2. *Figure 1(b) shows a mapping from a relational database to an XML schema. (The dashed lines represent foreign-key constraints.) The mapping implied by the correspondences consists now of two logical mappings (l_2 and l_3):*

l_2 : *for* e *in* *Dept.employee*, ep *in* *Dept.emp_prj*, p *in* *Dept.project*
where e*.eid* = ep*.eid* *and* ep*.pid* = p*.pid*
exists e' *in* *Emps.employee*, p' *in* e'*.projects.project*
where e'*.eid* = e*.eid* *and* e'*.name* = e*.name*
and p'*.pid* = p*.pid* *and* p'*.title* = p*.title*

l_3 : *for* e *in* *Dept.employee*
exists e' *in* *Emps.employee*
where e'*.eid* = e*.eid* *and* e'*.name* = e*.name*

Note that l_2 includes a three-way join on the source (expressed using the two equalities in the source where *clause). The second logical mapping l_3 is needed to transform the data for employees without any project.*

Although not shown in the examples, the equalities that express target fields in terms of the source fields can also include scalar functions (e.g, for data formatting).

2.2 The Three-Phase Process

A Clio mapping can consist of multiple logical mappings. We model the transformation dictated by each logical mapping as a three-phase process shown in Figure 2:

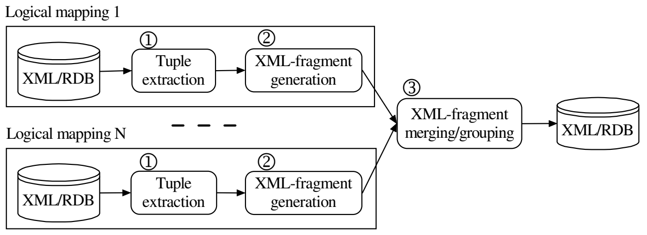


Figure 2: A three-phase execution model for schema mappings

Logical mappings have independent tuple-extraction and fragment-generation phases, but share the same merging phase. The tuple-extraction phase emits one tuple at a time and feeds the tuple to the fragment-generation phase, which in turn feeds the XML fragment to the merging phase. We now explain these three phases.

2.2.1 Tuple Extraction

The result of tuple extraction is a table of flat tuples, which is obtained by taking all the possible instantiations (with respect to the source data) of the variables in the for clause (provided that the associated where clause is satisfied); then, for each such variable, we include all its atomic subelements that are exported into the target (i.e., that appear in the where clause of the exists clause). For example, given the mapping in Example 1 and the source data in Figure 3, we will have four extracted tuples, shown in Table 1. The three columns correspond to the three "exported" atomic expressions: a, i.booktitle and i.title. The column names in the table are for reference purpose and can be arbitrary in implementation.

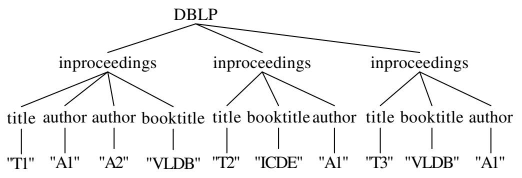


Figure 3: DBLP data for Example 1

For the logical mapping l_2 and data in Table 2, the extracted tuples are in Table 3. The extracted tuples for the logical mapping l_3 are the same as Table 2(a).

To implement tuple extraction, we use different algorithms based on the types of data sources and also the semantics of the extraction. Details are discussed in Section 3.

[2]http://www.informatik.uni-trier.de/ ley/db/

	author	booktitle	title
t_1	A1	VLDB	T1
t_2	A2	VLDB	T1
t_3	A1	ICDE	T2
t_4	A1	VLDB	T3

Table 1: Extracted tuples for l_1

(a) employee

eid	name
e1	Jack
e2	Mary
e3	Linda

(b) emp_prj

eid	pid
e1	p1
e1	p2
e2	p2

(c) project

pid	title
p1	Schema mapping
p2	DB2

Table 2: Relational data for Example 2

2.2.2 Generating XML Fragments

This phase takes one flat tuple at a time and transforms it into an XML fragment based on each logical mapping. The transformation is represented with an XML-fragment template as shown on the left side of Figure 4. This template corresponds to the exists clause and its associated where clause in a tgd. The leaf nodes are atomic elements and their text values are instantiated with extracted source fields. The right side of Figure 4 shows the resulting XML fragment for the source tuple t_1. Functions can be used in the field mappings.

It is possible that the same source field appears multiple times in the XML-fragment template. Conversely, multiple source fields can be mapped to the same template field—a function is mandatory in order to "combine" the multiple fields into the template field.

2.2.3 Merging XML Fragments

The last phase merges the resulting XML fragments into one XML document. The main purpose is to assemble a coherent and non-redundant target instance, by eliminating duplicate XML fragments and, furthermore, by merging XML fragments that share the same "key". As an example of the kind of merging we do, the four XML fragments resulting from the four tuples in Table 1 and the template in Figure 4 are merged into the final XML document shown in Figure 5. There, each author name appears only once, the relevant conferences are grouped under each author (again, without duplication, for a given author), and for each conference all the relevant publications are listed underneath (without duplication).

To achieve this merging, `name` is considered to be a key for the top-level `author` elements, `cname` is a key for the `conf_jnl` elements under each `author`, and `title` is a key for each `pub` element under a `conf_jnl` element. In Clio, the default key value of a repeatable element is the combination of all the atomic values that can be reached from that element without passing through another repeatable element. Moreover, the key of the parent repeatable element must also be included (so that, we know, for example that we are referring to the set of `conf_jnl` elements that are nested under a certain `author` element and not under a different `author` element). This key definition is similar to the *relative* key concept defined in [4]. More generally, the keys that will be used for merging can be individually defined based on application semantics via parameterized grouping functions [13]. Our merging techniques (described in the next section) can be extended to deal with such scenarios as well.

	eid	name	pid	title
t_1	e1	Jack	p1	Schema mapping
t_2	e1	Jack	p2	DB2
t_3	e2	Mary	p2	DB2

Table 3: Extracted tuples for l_2

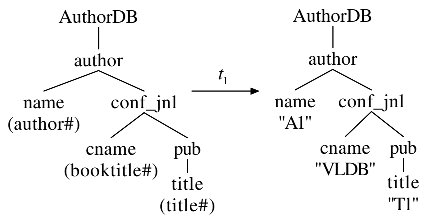


Figure 4: The XML-fragment template (left) for Example 1 and the corresponding XML fragment for t_1 from Table 1

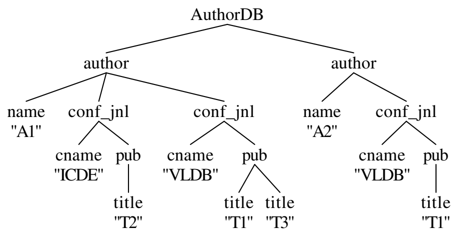


Figure 5: Target XML document for Example 1

There are two steps involved during the merging:

1. Obtain a single document by merging XML fragments on their common root node. Intuitively, all the resulting XML fragments are stitched together by their root element.
2. Merge sibling repeatable elements that have the same tag and moreover have the same key. To merge such elements we keep the common structure (the key) and union their corresponding sets of repeatable subelements (tag-wise). Moreover, the same merging is then applied recursively for each of these sets (if any).

We shall sometimes use the alternative terms *deep-union* and *grouping* (of the target data) for the merging process described above. Note that the elimination of duplicate XML fragments (at any level) occurs naturally as a special case of merging XML fragments with the same key.

Data merging is an essential part of data transformation tasks. There are three main reasons why data merging is necessary: we may need to group data in certain ways (like in our DBLP example), we may have multiple mappings and we need to merge their results, or we may have multiple data sources with overlapping data (a common situation in data integration scenarios). As we shall see, data merging can also be computationally intensive.

In the next two sections, we detail some of the algorithmic implementation of the two phases: the tuple-extraction phase and the merging phase. Although technically less challenging, the transformation phase is useful from the system point of view because operations such as data cleansing and data reformatting can be plugged in here as user defined functions.

3. TUPLE EXTRACTION

3.1 Streamlined Tuple Extraction from XML

In XML data, repeatable elements (or concepts) nest among each other according to their application semantics. For example, in the source schema in Example 1, we have a set of `inproceedings` elements under the root element `dblp`. In turn, each `inproceedings` element can have a number of `author` elements. The non-repeatable elements can be seen as the *attributes* of their parent/ancestor repeatable elements. The goal to tuple extraction from XML data is to *unnest* the nested concepts and form flattened tuples for transformation.

3.1.1 Set-Path Expressions

The extraction of flat tuples from XML data can be seen as matching of a *set-path expression*. A set-path expression is similar to an XPath expression except that each location step ends with a repeatable element and each repeatable element can have multiple atomic subelements or attributes for extraction.

Each repeatable element in a set-path expression corresponds to a bound variable in the for statement of a tgd. In addition, as in logical mappings, we use projection ('.') to express navigation through multiple levels of non-repeatable elements. For Example 1, the set-path expression for tuple extraction is:

`Q1 = /DBLP.inproceedings{title,booktitle}/author{.}`

Here, the two repeatable elements `inproceedings` and `author` are the end points of the two location steps. Their attributes (within braces) are the element values to be extracted.

3.1.2 Matching Set-Path Expressions

To match a set-path expression against an XML document, we use a publicly available SAX parser to assemble records corresponding to the set-elements in the set-path expression and then match these records based on their parent-child relationship specified in the expression. The record assembler, the record buffers and the matching algorithm are what we implemented.

EXAMPLE 3. *For the set-path expression `Q1` and the first two `inproceedings` elements in Figure 3, the record sequence (generated from the record assembler) is: a_1=[(A1), p_1], /a_1, a_2=[(A2), p_1], /a_2, p_1=[(T1, VLDB), -], /p_1, p_2=[(T2, ICDE), -], a_3=[(A1), p_2], /a_3, /p_2. A record prefixed with a slash (such as /a_1) is an end-record. An end-record must have a preceding start-record, such as a_1. Each start-record has two components: the tuple with extracted values and the id of its parent record. When there is no ambiguity, we also call a start-record record. For the record a_1, its tuple is (A1) and the id of its parent record is p_1. (The reason why p_1 arrives* after *a_1 and a_2 is explained after the example.) The record p_1 has the extracted tuple (T1, VLDB). Since it does not have a parent record, we put a dash symbol ("-") in its second component.*

There are three points worth noticing about the record-creation process. First, a parent start-record may come *after* its child start-records. In Example 3, p_1 comes after its children a_1 and a_2 because we need to wait for the `booktitle` element. Second, if all the extraction elements of a set-element have arrived, we can send the start-record (to the matching algorithm) before the set-element closes (i.e., its end-element SAX event arrives). For example, the start-record p_2 is sent before its child record a_3. Third, if a set-element has a missing subelement that is specified for extraction, we fill the corresponding field with a null value when the set-element closes. A missing extraction subelement is possible if the corresponding schema subelement is optional.

The actual matching of the set-path expression can take place as new records arrive from the record assembler. If we see each record as a start-element SAX event, what needs to be done is essentially the streamlined matching of simple path expression queries. Streamlined matching of path expressions is not a new problem and has been studied in different contexts. What is interesting in this scenario is that the parent record may come *after* child elements—analogously, the parent start-element event comes after the child start-element event. This phenomenon violates the assumption of most streamlined matching algorithms, which assume SAX events of elements come in document order.

Our solution to the disordered SAX events is to use blocking. If the parent record has not arrived, we need to block all its descendant records and keep them in buffer. Figure 6 shows the matching process of the query `Q1` given the sequence of records in Example 3.

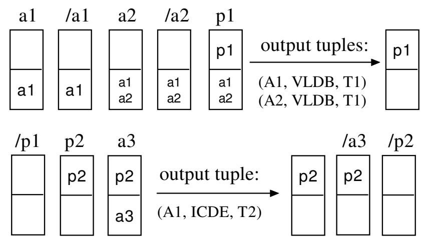


Figure 6: An example of streamlined matching of set-paths

In Figure 6, before the start-record p_1 comes, we need to block its child records a_1 and a_2 in the buffer. On the other hand, since p_2 comes before its child a_3, we can do the matching and output the result as soon as a_3 arrives.

In the worst-case scenario, if a blocked parent record has a large number of child records, we may need to buffer all these child records. We implemented a hybrid record buffer, which automatically swaps records to a temporary file when it runs out of prescribed physical memory. The swapped records can be read back from disk sequentially.

3.1.3 Computation Sharing Among Set-Paths

When there are multiple set-path expressions (from multiple logical mappings) on the same input XML document, it is possible to coordinate the matching of set-path expressions to achieve better performance.

There are two alternatives. With a loose-integration approach, all the set-path expressions share the same SAX parser for reading the XML document while each of them has a separate record assembler and matching module. This approach only requires one scan of the input document but the computation of matching is not shared.

Alternatively, we can have a tight-integration approach: we can merge multiple set-path expressions into a complex (possibly tree) structure and match them together. For example, if we have another set-path expression on top of `Q1`,

`Q2 = /DBLP.inproceedings{title,booktitle}`

then we can combine `Q1` and `Q2` and match them together. In this case, the combined structure is the same as `Q1`. The difference is that, in addition to the outputs in Figure 6, we should also output the two tuples, (VLDB, T1) and (ICDE, T2), for `Q2`.

3.2 SQL Queries for Relational Data Sources

We can generate SQL queries to extract tuples from relational data sources so that the mature relational technology is leveraged.

We can follow three rules to translate the tuple extraction of a logical mapping into a SQL query: (1) the referenced tables in the for statement of the logical mapping should appear in the from clause of the SQL query; (2) the conditions in the where statement appear in the SQL where clause and (3) the referred fields appear in the select clause of the SQL query.

For example, the two corresponding SQL queries for the two logical mappings in Example 2 are as follows:

q_2 : select eid, name, pid, title
from employee e, emp_prj ep, project p
where e.eid = ep.eid and p.pid = ep.pid
q_3 : select eid, name
from employee

In particular, the result of q_2 is listed in Table 3.

The results of the two SQL queries q_2 and q_3 overlap on the employees that have projects. Sometimes it might be better off to combine these two SQL queries into one outer-join query:

q_{23} : select eid, name, pid, title
from employee e left outer join
(emp_prj ep inner join project p on p.pid = ep.pid)
on e.eid = ep.eid

Whether to combine multiple SQL queries (from logical mappings) and how to combine them is a non-trivial problem. It depends on not only the original SQL queries but also how the resulting tuples are used in each logical mapping.

An appealing extension is to use the new nested mapping framework of Clio described in [13], which can combine multiple logical mappings into one, whenever appropriate. This has the advantage that the decision of when and how to combine multiple logical mappings into one is taken by the mapping generation component of Clio, driven by considerations of mapping semantics rather than query optimization. Outer joins could then be generated directly from the combined logical mappings.

4. HYBRID NESTED MERGING

The problem of merging XML fragments (as described in Section 2.2.3) is a nested variation of the union operation with duplicate removal in relational databases (RDBMS). Although relational databases nowadays are highly scalable and efficient, they are not well suited for processing such a nested merge operation. For example, to apply RDBMS for merging XML fragments, we may need to sort the keys at higher levels of XML fragments first, merge the XML fragments based on the higher-level keys, and then sort the lower-level keys for each common higher-level key. Obviously there is a lot of overhead in carrying around intermediate XML fragments.

We describe hybrid merge algorithms that perform the nested merge of XML fragments in main memory with linked hash tables and then dynamically switch to sort-based algorithms if hash tables use up available memory. In particular, the worst-case I/O cost of the sort-based algorithm is $O(N \log N)$, where N is the size of XML fragments being merged.

4.1 Hash-Based Merging

An efficient way to do the nested merge of XML fragments is to use main-memory hash tables. Our idea is to create a chain of hash tables to do the grouping in a top-down fashion defined by the grouping semantics. Since the merging takes place for repeatable elements based on the key values of their parents, we need to create one hash table for each repeatable element under each distinct parent key entry (see the next example).

EXAMPLE 4. *Figure 7 shows the process of creating hash tables when merging the four XML fragments in Example 1. There is only one hash table for the top-level `author` set-element. For each entry (with key `name`) of the `author` element, we create a `conf_jnl` hash table. Similarly, for each distinct entry (with key `cname`) in such a hash table, there is a `pub` hash table (whose entries are the distinct values of `title`). In the figure we show the evolution of the hash tables with the incoming XML fragments. Recall that these fragments were obtained, in phase two, from the tuples t_1, t_2, t_3, and t_4, by applying the transformation described in Figure 4. We can now generate the XML document in Figure 5 by flushing the final hash tables.*

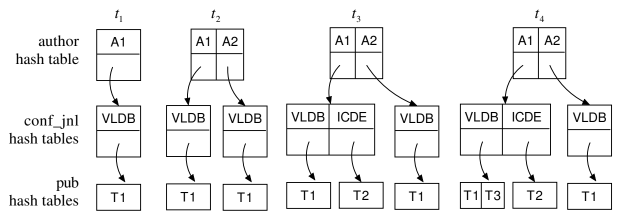


Figure 7: Changes of hash tables with new XML fragments

The proposed hash-based merging can gracefully handle target schemas with multiple sibling set-elements. Suppose, in Example 1, the `conf_jnl` set-element has a sibling set-element `book` that has the same subtree structure as `conf_jnl`. The content of the `book` subtree is mapped from other data sources on books. With such a mapping, each key in the `author` hash table of Figure 7 may have two child hash tables, one for `conf_jnl` and the other for `book`—in other words, these two hash tables share the same key entry.

4.2 Hybrid Sort-Based Merging

Although efficient, the hash-based merging is limited by the amount of available main memory because all the hash tables must reside in memory for them to work effectively. We address the scalability problem with a sort-based algorithm that builds on top of the hash-based merging.

When the hash tables take up all the allocated physical memory, we write them to a disk file as a file-run (described next) and then free up all the hash tables for subsequent incoming XML fragments. When all the XML fragments are processed into file-runs, we merge the disk-resident file-runs.

When outputting a file-run from the linked hash tables, to achieve linear I/O cost when merging file-runs, we enforce an appropriate ordering among the keys of the same hash table. Specifically, we start the serialization from the root hash table (the `author` hash table in Figure 7), sort the key values in a predetermined order (in our implementation, the ascending order) and then output the keys in that order. After each key is output, we recursively serialize its child hash tables (the number of hash tables is the same as the number of corresponding sibling set-elements). For example, the file-run for the final hash tables in Figure 7 (after t_4 is merged) is as follows:

$$A1[ICDE[T2], VLDB[T1, T3]], A2[VLDB[T1]]$$

For clarity, each list of keys is enclosed by a pair of "[" and "]" (except for the outer-most list), and is separated by a comma. In this example, the key "ICDE" appears ahead of the key "VLDB" for "A1" according to the ascending order.

We can merge multiple file-runs in one sequential scan. Suppose we have another file-run as follows:

$$A2[VLDB[T4], WWW[T5]]$$

To merge the above two file-runs, we first compare the first two keys (i.e., A1 and A2) from the file-runs. Since A1 is larger than A2, it means that nothing else in the second file-run can merge with A1 so we directly output all the content for A1 (including the nested content). Now the current point in the first file-run becomes A2. The two file-runs share the same key. As a result, we output key A2 and then recursively merge the two smaller file-runs nested inside these two keys. The result of the merge is as follows:

$$A1[ICDE[T2], VLDB[T1, T3]], A2[VLDB[T1, T4], WWW[T5]]$$

We have explained how to create file-runs from hash tables and how to merge file-runs in one sequence scan. Usually, each file-run would require a small amount of physical memory (such as the size of one disk page) during the multi-way merge. If the number of file-runs is extremely large, we may not have enough main memory just to allocate one disk page for each file-run. In that case, since the output of the merge process is also a valid file-run, we can apply multi-stage merging as commonly used in relational databases.

5. USING EXISTING QUERY LANGUAGES

We have shown the three-phase transformation framework and briefly described the algorithms for implementing it. An alternative to this approach that we shall compare with is the generation of queries that implement, with the same semantics as our transformation engine, the schema mappings. In this section, we discuss the shape of the queries needed to implement such semantics. The performance of the generated queries is then compared with our transformation engine in Section 6. We focus on XQuery and SQL/XML, but the issues for XSLT are pretty much the same.

The positive side of using existing query languages is that we can leverage existing query engines. Most commercial database systems have started to provide support for popular XML query languages such as XQuery, SQL/XML and even XSLT, and we can expect that these commercial engines will be continuously improved and become more scalable and efficient. On the negative side, the queries that are needed to implement the transformation semantics implied by schema mappings (in particular, the merging or grouping phase) can be quite complex. This in turn affects the performance, as we shall see.

5.1 Using XQuery Queries

The XQuery for the mapping in Example 1 is shown below (the variable $doc stands for the root of the input document):

```
<authorDB>
{for   $x1 in distinct-values ($doc/dblp/inproceedings/author)
return <author>
          <name> {$x1} </name>
          {for   $x2 in distinct-values($doc/dblp/inproceedings
                        [author=$x1]/booktitle)
           return <conf_jour>
                    <name> {$x2} </name>
                    {for   $x3 in distinct-values($doc/dblp/inproceedings
                                  [author=$x1 and booktitle=$x2]/title)
                     return <pub><title> {$x3} </title></pub>
                    }
                  </conf_jour>}
       </author>}
</authorDB>
```

The mapping in Example 1 indicates that we need to group by `author` value first, then `booktitle` and then remove duplicate `title` values. To implement this semantics, a `distinct-values` function must be used at each level:

- The variable $x1 iterates over the sequence of distinct `author` values. All the items in the sequence returned by the function `distinct-values` are literal strings with the tags stripped off [2].
- The variable $x2 iterates over the sequence of distinct booktitles for which the corresponding `inproceedings` element has at least one `author` with the value $x1. This correctly implements the grouping semantics for the second nesting level in the target schema (i.e., produce `booktitle` values for the particular `author` $x1).
- In the inner-most for loop, we use the variable $x3 to iterate over a sequence of distinct `title` values such that the `title` belongs to an `inproceedings` element with the `author` $x1 and the `booktitle` $x2. This correctly implements the grouping semantics for the third nesting level in the target schema.

When we generalize the above methodology to arbitrary logical mappings, we note that the resulting query can quickly increase in complexity. First, if multiple fields are used as the grouping key, the above pattern necessarily becomes more complex, since XQuery does not support duplicate removal at the "tuple" level. Instead we need to apply the `distinct-values` function multiple times, once for each of the fields. Furthermore, the applications must be correlated so that we do not lose the structural association between the fields. We illustrate this with an example.

Suppose in the mapping in Example 1, we have an additional atomic element `pages` under the inner-most set-element pub. In that case, the `title` and `pub` elements become the key for the duplicate removal (i.e., if two `pub` elements have the same `title` and `pages` child elements, we should remove one). The following query fragment (for the inner-most nesting) is incorrect because the structural relationship between `title` and `pages` elements is lost, and we end up pairing up all the `title` and `pages` elements, even from different `inproceedings` elements.

```
for    $x3 in distinct-values($doc/dblp/inproceedings
                 [author=$x1 and booktitle=$x2]/title),
       $x4 in distinct-values($doc/dblp/inproceedings
                 [author=$x1 and booktitle=$x2]/pages)
return <pub><title>{$x3}</title> <pages>{$x4}</pages></pub>
```

To keep the association between the bound `title` and `pages` elements, we should restrict the `pages` iterated by the variable $x4 to belong to the `inproceedings` elements with the `title` $x3. The correct binding for $x4 should have an additional predicate "title=$x3", as shown below:

```
$x4 in distinct-values($doc/dblp/inproceedings
         [author=$x1 and booktitle=$x2 and title=$x3]/pages)
```

Furthermore, when there are multiple logical mappings (and multiple data sources) in a schema mapping, we must produce the results from each of the logical mappings before we can merge the resulting target data based on their keys. Thus, there is a union phase followed by a merging and duplicate elimination phase. The needed query will start looking more like the three-phase (ETM) transformation approach. Finally, we note that expressing the merging phase in XQuery (as it can be seen from the above query pattern) requires nested loops. This is one of the main causes of inefficiency. Moreover, it is not clear how to make use of any indexes that may exist on the input data; what we need instead are indexes on the generated data. Our nested merging algorithm can in fact be seen as a combination of indexing (by hashing) and sorting.

We conclude with the observation that the XQuery queries that implement schema mappings can be rather complex, mainly due to the data merging requirements. For large mappings and input data, the XQuery queries can also be prohibitively costly to run.

5.2 Leveraging SQL/XML

SQL/XML [8] is an extension of the SQL language for publishing XML data from relational tables. It is widely supported by commercial databases.

When source data resides in relational tables, we can translate schema mappings into SQL/XML queries. In SQL/XML, the grouping can be implemented by the `distinct` keyword together with the function `xmlagg`.

Assume that the source DBLP data in Example 1 is shredded and stored in relational tables with the following schema: `paper(pk, title, booktitle)` and `author(name, pk)`. The field `pk` is the key for the `paper` table. Each `inproceedings` element in the source XML data is assigned a unique `pk` value. The `pk` field in the `author` table is a foreign key pointing to the `paper` table. Given the relational schema, the SQL/XML query block that implements the mapping is shown below:

```
select xmlelement(name ”authorDB”,
     xmlagg
          (xmlelement (name “author”,
          xmlelement (name “name”, x0.name),
          (select xmlagg
               (xmlelement (name “conf_jour”,
               xmlelement (name “name”, x2L1.booktitle),
               (select xmlagg
                    (xmlelement (name “pub”,
                    xmlelement (name “title”, x3L2.title))
                    ) as XML — end of third xmlagg
               from LATERAL(select distinct title
                    from author x0L2, inproceedings x1L2
                    where x0L2.pk=x1L2.pk and
                         x2L1.booktitle = x1L2.booktitle
                         and x0.name=x0L2.name) as x3L2))
               ) as XML — end of second xmlagg
          from LATERAL(select distinct booktitle
               from author x0L1, inproceedings x1L1
               where x0L1.pk=x1L1.pk and
                    x0.name=x0L1.name) as x2L1 ))
     ) as XML — end of first xmlagg
     ) — end of top-most xmlelement
from (select distinct name from author) x0
```

Similar to the `distinct-values` in XQuery, “select distinct” is used to guarantee that we generate exactly one tuple for each unique key in the corresponding grouping level. Keyword `LATERAL` is required for outer reference from derived tables.

We note that, when it comes to merging of the target data, SQL/XML suffers from the same drawbacks that we pointed out for XQuery.

6. PERFORMANCE EXPERIENCE

We evaluate our transformation engine from the performance point of view. We study the scalability of the system and also compare its performance with that of the corresponding XQuery and SQL/XML queries. The performance of XSLT queries is qualitatively similar to that of XQuery and SQL/XML and is omitted.

6.1 Systems Compared

The transformation engine is implemented with Java (specifically IBM Java 1.4.2). It takes as input the source data specification (such as XML files or relational tables) and a schema mapping saved from Clio, and executes the mapping to generate target data.

For phase one, relational data sources are stored in relational tables and accessed through JDBC. The streamlined matching algorithm (for set-path expressions) used the Xalan SAX parser for processing source XML documents. In phase three, the sort-based algorithm stored file-runs in temporary files, using buffered I/O streams (i.e., BufferedOutputStream and BufferedInputStream Java classes). Roughly 200MB memory is allocated to the hash tables.

We used the latest versions of two leading commercial databases in our comparison. To keep anonymity, we refer to them as DB$_x$ and DB$_y$. Both databases provide full support for XQuery and SQL/XML. Our transformation engine reads input XML documents from the file system and writes the output back as a file. For the two databases, we preloaded tested XML documents into tables and the output was inserted into a result table. The time for loading XML documents into tables was not counted.

All the experiments were conducted on a Pentium IV 2.80GHz PC with 1GB RAM, 160G hard disk and Windows XP.

6.2 Overall Performance

We study the scalability of our framework, using the mapping in Example 1 and two other mappings derived from it. We denote the mapping in Example 1 as $\mathcal{M}_{n3}$ to reflect the fact that there are three nesting levels in the generated data. We derived a mapping $\mathcal{M}_{n2}$ with two nesting levels by removing the correspondence from `booktitle` to `cname`—we group publications by authors only (and the `cname` element in the middle will be empty). The third mapping, $\mathcal{M}_{n4}$, has four nested levels and three levels of grouping. We derived it by inserting a new set-element `year` between the `conf_jnl` and `pub`, and mapping from `year` (which is not shown in Example 1) of `inproceedings` to the `year` in the target. The mapping means that we want to group conferences by author, then group by conference, and then group publications by year.

For the scalability study, we prepared a few DBLP data sets with varying sizes ranging from less than 1MB up to 1.2GB. The conference entries (i.e., `inproceedings` elements) in the original DBLP data add up to less than 200MB in size. In order to scale up, we replicate the DBLP data by changing the values of atomic elements. We ran the transformation implied by the three mappings for each data set, using our transformation engine. Figure 8 shows the timings for the three mappings, for different input data sizes.

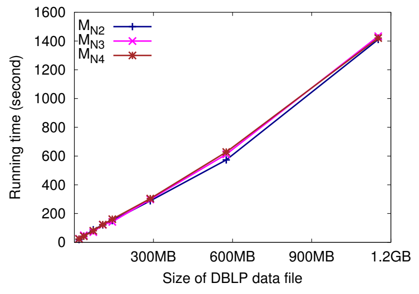


Figure 8: Scalability test of our transformation engine

Figure 8 shows that the three-phrase framework is scalable in terms of the data size: the running time is almost linear to the input data size. An interesting finding is that the running time is not sensitive to the number of nesting levels in a mapping. Given a DBLP document of a particular size, the running time is almost the same for the three mappings. The main reason for this is the use of the in-memory hash tables to perform grouping. When we sort and output each hash table to a temporary file, the dominant cost factor is disk I/Os, and this is bound by the input data size.

To investigate the performance of each component in the transformation engine, we break down the times for transforming the 1.2GB DBLP data. Figure 9 shows the results. The time spent on the sort-and-merge takes up most of the running time (over 70%). The sort-and-merge includes sorting hash tables, writing them to temporary run-files and merging the run-files into the final XML document. According to experiment results, a mapping with one more nesting level used about 20 more seconds on hashing. However the impact of hashing on the total time is small because the sort-merge dominates the total time.

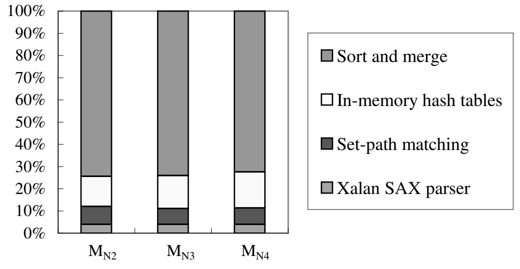


Figure 9: Time breakdown for the transformation engine

6.3 Comparing with XQuery Queries

We now compare with the two commercial databases using XQuery queries to implement the schema mappings.

Figure 10 shows the running times of the XQuery queries for the three mappings on the two databases DB$_x$ and DB$_y$. The X-axis in the figure lists the number of the `inproceedings` elements in each data set. For the data set with about 1000 `inproceedings` elements, the file size is only about 420KB. We did not test the queries for larger data sizes because the findings are rather clear with the shown tests.

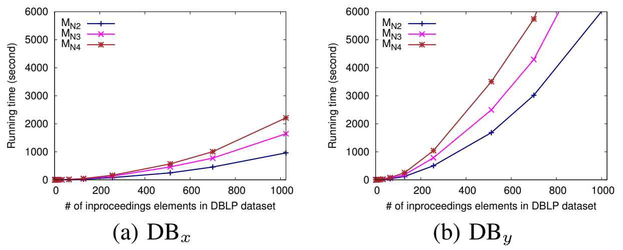


Figure 10: Timings of the three mappings using XQuery queries on two commercial databases. In contrast, our engine finishes transforming the data with 1000 `inproceedings` elements for mapping $\mathcal{M}_{n4}$ in less than two seconds.

We find a significant difference in the running times for the two databases. However, they exhibit the same behavior: the running time is polynomial (rather than linear) in the input data size. The more the nesting levels, the more quickly the running time increases with the input size. For mapping $\mathcal{M}_{n4}$, it took DB$_x$ about 35 minutes to transform the data with only 1000 `inproceedings` elements. In contrast, our engine can finish the most costly transformation in less than two seconds, outperforming the tested databases by three orders of magnitude!

The nested correlated subqueries with the `distinct-values` functions (see, Section 5) are the main cause of the degraded performance. We checked the query access plan used by the databases and found that nested loop joins (NLPJ) are used for each for clause in the query. As a result, the query execution time is $O(|D|^{N_i})$, where $|D|$ is the size of the input document and N_i is the number of nesting levels in the mapping.

6.4 Comparison with SQL/XML

We ran the three mappings with SQL/XML by shredding each DBLP data set into two relational tables, a `paper` table and an `author` table (see, Section 5.2). To have a fair comparison, our engine used SQL queries to extract tuples from the relational tables through JDBC.

Figure 11 shows the running times for the two databases. We observed a similar performance advantage of our engine over SQL/XML queries. For example, DB$_x$ took about 1000 seconds for 4.5MB DBLP data, while our engine can finish $\mathcal{M}_{n4}$ for this data set in 2.1 seconds.

As in the case of XQuery, both databases demonstrate low performance for SQL/XML queries because they need to use nested queries to generate the nested target data. Although the running times differ for the two databases, the basic trend of their performance is the same: the more nesting levels, the more quickly the running time increases. It is worth noting that the database DB$_y$ does not support the keyword `LATERAL` in SQL/XML. For DB$_y$, we used approximate queries (we un-nest the queries that use `LATERAL`) to run the tests.

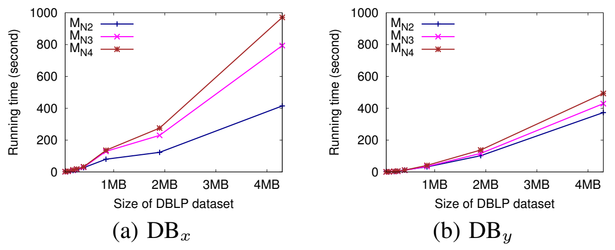


Figure 11: Running times for the three mappings using SQL/XML on two commercial databases. As a comparison, our engine finishes $\mathcal{M}_{n4}$ for the largest data set in 2.1 seconds.

6.5 Summary of Results

We have presented experimental results on the performance aspects of our three-phase transformation framework. The conclusion is that such a framework suits well for mapping-driven transformation and it is highly efficient and scalable.

We compared our implementation with the approach of using XQuery and SQL/XML queries executed on two leading commercial databases. The results show that our implementation of schema mappings can be significantly faster than XQuery and SQL/XML queries (with the speedup of a few orders of magnitude). The current state-of-the-art databases are not as efficient as ours because of the lack of direct support of a deep-union operation that recursively groups sibling subtrees based on keys. The deep-union operation plays a key role in generating meaningful and concise data from schema mappings.

We also tested a few publicly available XQuery engines, including Galax (version 0.5.0)[3], Quip (version 2.2.1.1)[4], MonetDB/XQuery (version 0.10.2)[5] and Saxon for .NET (version 8.7)[6]. Their performance for the tested transformation queries is no better than the two commercial databases that we tested. We are also aware of other XQuery engines such as the BEA/SQRL [12], which we did not test. However none of the XQuery engines we know provides a deep-union operation.

We stress that our results by no means imply that the commercial databases provide poor support for all XQuery applications. Mapping-driven large-scale data transformation may not be among the typical applications that XQuery is best suited for. If XQuery is going to be successfully applied for such transformation, we suggest a deep-union operator be introduced.

[3]http://www.galaxquery.org/
[4]http://www.softwareag.com
[5]http://monetdb.cwi.nl/XQuery
[6]http://saxon.sourceforge.net

7. RELATED WORK

XML Publishing SilkRoute [11] and XPeranto [6, 20] are prototype systems that publish object-relational data as XML using views and query composition techniques. One major focus is the composition of user queries (such as XQuery or XML-QL) with the XML views to generate SQL queries to pull data out of relational databases. After that, an XML tagger is used to construct hierarchical XML data. The uniqueness of our approach is that, we start from XML, obtain flattened tuples (Extract), transform them to XML fragments (Transform), and, finally, merge the obtained results (Merge).

XML Value Extraction TurboXPath [16] can process XML pattern queries with multiple extraction nodes. It supports a wider class of pattern queries and can only process one pattern at a time. Our algorithm for set-path expressions is designed for efficient tuple extraction for schema mappings and, more important, can do tuple extraction for multiple set-path expressions together (like outer-joins in relational databases).

XML Merging and Aggregation XML merging is also studied in the context of XML archiving [5]. We note that part of the algorithm in [5] shares similarity with the sort-based algorithm presented in this paper. However, their focus was on reducing the size of the resulting XML archive and did not report results on the performance of execution. In fact, our performance improvement benefits from additional techniques. NEXSORT [21] sorts the sibling elements of an XML data tree based on a given user criterion. It uses a bottom-up approach so that the I/O cost for sorting is almost linear to the data size. NEXSORT is different from the nested-merge operation because it sorts tree nodes *locally* and never combines tree nodes from different parent tree nodes.

XML OLAP There is research on providing better support for grouping operations in XQuery either at the algebraic level or the physical operator level [7, 10]. In general, detecting a "group by pattern" from a complex XQuery is very difficult and often results in a poor execution plan [1]. There has been a proposal [1] to extend the XQuery syntax to include explicit grouping operations such as "group by". We note that the target application for these works is XML OLAP instead of the hierarchical nested merge that arises in schema mappings.

8. CONCLUSION

We have presented an end-to-end system for mapping-driven XML transformation. The system takes source XML documents and transforms them into target XML documents based on schema mappings. The proposed approach can be highlighted as *efficient* (transformation speed close to linear in the input data size), *scalable* (handling GB-range data sizes under limited memory consumption) and *extensible* (supporting user-defined functions through standard Java interfaces). We achieve the efficiency and scalability in transformation by interpreting schema mappings directly and providing best suited algorithms to implement them, such as SQL queries, streamlined tuple extraction and disk-based tree merging. These high-level semantics might be hard to recover and hence optimize if buried in generated queries.

9. REFERENCES

[1] K. S. Beyer, D. D. Chamberlin, L. S. Colby, F. Özcan, H. Pirahesh, and Y. Xu. Extending XQuery for analytics. In *SIGMOD*, pages 503–514, 2005.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, W3C Working Draft, 2003.

[3] A. Bonifati, E. Q. Chang, T. Ho, L. V. S. Lakshmanan, and R. Pottinger. Heptox: Marrying XML and heterogeneity in your p2p databases. In *VLDB*, pages 1267–1270, 2005.

[4] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for XML. In *WWW*, pages 201–210, 2001.

[5] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, 29:2–42, 2004.

[6] M. J. Carey, D. Florescu, Z. G. Ives, L. u. Ying, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB (Selected Papers)*, pages 105–110, 2000.

[7] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT framework for logical XQuery optimization. In *VLDB*, pages 168–179, 2004.

[8] A. Eisenberg and J. Melton. SQL/XML and the SQLX informal group of companies. *SIGMOD Record*, 30(3):105–108, 2001.

[9] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[10] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed XML data. In *CIKM*, pages 126–133, 2002.

[11] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. Silkroute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, 2002.

[12] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *VLDB*, pages 997–1008, 2003.

[13] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: Schema mapping reloaded. In *VLDB*, pages 67–78, 2006.

[14] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: From research prototype to industrial tool. In *SIGMOD*, pages 805–810, 2005.

[15] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.

[16] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *VLDB Journal*, pages 197–210, 2005.

[17] S. Melnik, P. A. Bernstein, A. Y. Halevy, and E. Rahm. Supporting executable mappings in model management. In *SIGMOD*, pages 167–178, 2005.

[18] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[19] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[20] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *VLDB*, pages 65–76, 2000.

[21] A. Silberstein and J. Yang. NEXSORT: Sorting XML in external memory. In *ICDE*, pages 695–707, 2004.

[22] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[23] XML Query Use Cases. Website, 2006. `http://www.w3.org/TR/xquery-use-cases/`.