Dissemination of Heterogeneous XML Data in Publish/Subscibe Systems

Yuan Ni
IBM China Research Lab
niyuan@cn.ibm.com

Chee-Yong Chan
National University of Singapore
chancy@comp.nus.edu.sg

ABSTRACT

The publish-subscribe paradigm is an effective approach for data publishers to asynchronously disseminate relevant data to a large number of data subscribers. A lot of recent research has focused on extending this paradigm to support content-based delivery of XML data using more expressive XML-based subscription specifications that allow constraints on both data contents as well as structure. However, due to the heterogeneous data schemas used by different data publishers even for data in the same domain, an important challenge is how to efficiently and effectively disseminate relevant data to subscribers whose subscriptions might be specified based on schemas that are different from those used by the data publishers. In this paper, we examine the options to resolve this schema heterogeneity problem in XML data dissemination, and propose a novel paradigm that is based on data rewriting. Our experimental results demonstrate the effectiveness of the data rewriting paradigm and identifies the tradeoffs of the various approaches.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: System-Query processing

General Terms

Algorithm Design Performance

Keywords

Data rewriting, dissemination, heterogeneous, XML

1. INTRODUCTION

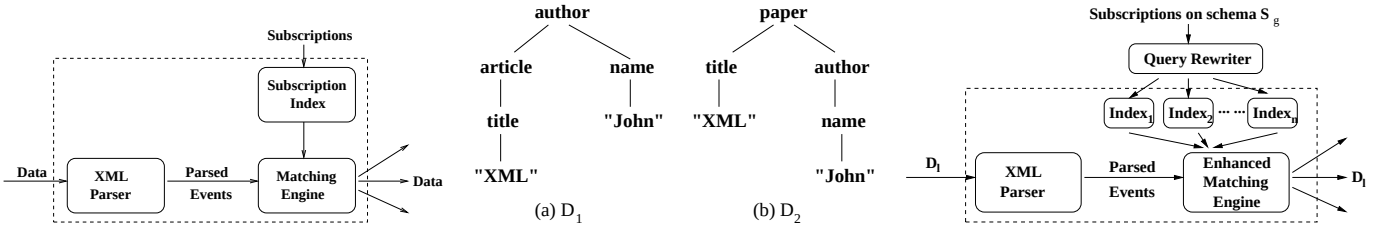
The ubiquity of XML data and the effectiveness of the content-based pub/sub-based paradigm of delivering relevant information has led to a lot of interest in content-based dissemination of XML data [2, 7, 8, 11, 19]. In the pub/sub environment, an overlay network of application-level *routers* (or message brokers) is used to asynchronously forward documents generated by *data publishers* to relevant *data subscribers* (or consumers) based on matching the document contents against the consumers' subscriptions. Fig. 1(a) shows a schematic diagram of the key components in a typical content-based router. An incoming XML document D is first parsed by an event-based XML document parser which generates basic events corresponding to the relevant document tokens (i.e., start-/end-element tags, attributes, and values). The parsed events are used to drive the matching engine which relies on an efficient index (e.g., [2, 7, 14]) on the subscriptions to quickly detect matching subscriptions in its routing table; D is then forwarded to neighboring routers and local subscribers with matching subscriptions.

Existing work on XML data dissemination (e.g., [2, 7, 14]), however, are all implicitly based on a *homogeneous schema* assumption where both the data published by different publishers as well as the users' subscriptions share the same schema. However, since the data publishers in a pub/sub system are autonomous and independent, they generally do not use the same schemas even when their published data are related and belong to the same domain (e.g., product catalogues). Consequently, if a user's subscription is based on the schema of a specific publisher (say P), then while the user can receive relevant documents from P that match his subscription, it is very likely that his subscription will not match relevant data from another publisher P' if the data schemas used by P and P' are different. Thus, the effectiveness of the pub/sub systems in pushing relevant data to consumers becomes diminished in the presence of heterogeneous data schemas.

For example, consider a user Alice who is interested in information on papers authored by "John", and has specified the following XPath subscription (based on the schema from some publisher P_0): /author[name = "John"]/paper/title. Consider the two XML documents D_1 and D_2 in Fig. 1(b) that are published by two different publishers P_1 and P_2. Although both documents describe papers authored by "John" and should be of interest to Alice, existing pub/sub systems would not have delivered these relevant documents to Alice because her subscription fail to match the data due to the difference in the schemas used by P_0, P_1, and P_2.

In this paper, we address the problem of how to improve the effectiveness of XML data dissemination in the presence of heterogenous data schemas [20]. For simplicity and without loss of generality, we assume that all the published data are of the same domain such that it is possible to use a single global schema to resolve the structural con-

CIKM'09, November 2–6, 2009, Hong Kong, China.



(a) Content-based routing of XML data (b) Sample XML documents (c) Query Rewriting Approach (QRA)

Figure 1: Conventional router architecture, sample XML documents & Query Rewriting Approach

flicts among different publishers' schemas (of the same domain). Our problem and proposed techniques can be easily extended to the general case by first partitioning the collection of publishers' schemas into groups of schemas with similar domains, and then generating a global schema for each group of related schemas.

Note that our *heterogeneous data dissemination problem* is different from the more well-known *data integration problem* [16, 28, 6, 29]. In data integration, the focus is on how to query multiple data sources with different schemas. In contrast, the problem that we are addressing is on how to compare a published data against a collection of queries (i.e., subscriptions) to identify the matching queries given that the data and queries are based on different schemas. Thus, a fundamental difference between these two problems, which are related by the presence of schema heterogeneity, is that the integration problem belongs to a single-query-multiple-data scenario while the dissemination problem belongs to a single-data-multiple-queries scenario.

To better appreciate the difference between the two heterogeneous schema problems and motivate our solution, let us consider how to apply the query rewriting idea in data integration problem to solve the heterogeneous data dissemination problem. The first step is to resolve the structural conflicts in the collection $\{S_1, S_2, \cdots, S_n\}$ of different publishers' schemas to generate a global schema S_g which is then made available to users to specify their subscriptions. Then, each "global" subscription q_g (which is based on global schema S_g) is rewritten into a set of local subscriptions $\{q_1, q_2, \cdots, q_n\}$, where each q_i is based on a local schema S_i. To enable efficient matching a published data (conforming to some local schema S_i, $i \in [1, n]$) against local subscriptions based on S_i, an index I_j is constructed for each collection of subscriptions based on local schema S_j, $j \in [1, n]$. Fig. 1(c) illustrates the above approach which is referred to as the *query rewriting approach (QRA)*. For each incoming data D_ℓ (based on some local schema S_ℓ), the matching engine only needs to compare D_ℓ against the appropriate set of local subscriptions via index I_ℓ.

The query rewriting approach, however, suffers from two drawbacks. First, the scalability of the approach is limited as each input subscription needs to be rewritten into one subscription for each local schema. This increases the space overhead for storing and indexing the expanded set of local subscriptions at each router. Note that although the input global subscriptions are not used directly for document matching, these subscriptions will still need to be maintained for generating new rewritings whenever a new local publisher schema is added (or changed). Second, the approach also incurs a high update cost. Whenever a new data schema S' is introduced (by an existing or new publisher), it is necessary to generate and install new subscriptions (for schema S') at each router by rewriting the global subscriptions registered at each router to corresponding local subscriptions on the new schema S'.

Another different direction taken to address the problem of schema heterogeneity is to apply *query relaxation* techniques (e.g., [23, 3]). This can be viewed as a *schema-independent query rewriting* approach where a query is "relaxed" to multiple queries without relying on knowledge of data schemas but based on making local structural changes to parts of the query. The motivation for this line of work is to enable retrieval of approximate answers to a query and it is often used in combination with some ranking and pruning mechanism during query evaluation at run-time to control the number of relaxed queries generated. However, it is unclear how this technique can be effectively applied to the context of the data dissemination problem since the number of queries registered at each router is large which makes run-time relaxation of a large set of queries a challenging problem. Alternatively, another possibility is to try to precompute the relaxed queries offline; but in the absence of the run-time data, it is unclear how the relaxed queries can be generated efficiently and in a controlled manner without a large set of relaxed queries being produced.

In this paper, we present a novel paradigm to solve the heterogeneous data dissemination problem that is based on the principle of *data rewriting*, which is called *DRA* for data rewriting approach. The conceptual idea of DRA is as follows. First, the collection of local schemas from the publishers is integrated to form a global schema S_g which is then made available to users to specify their subscriptions. Unlike QRA, our DRA does not require query rewriting which means that only the input global subscriptions are indexed at each router. For each incoming data D_ℓ (conforming to some local schema S_ℓ) to a router, our DRA rewrites D_ℓ to D_g (D_g may not be materialized here) such that the evaluation of subscriptions is actually conducted against D_g.

In contrast to QRA, our proposed DRA is more effective for the heterogeneous data dissemination problem because pub/sub systems are typically characterized by two properties : (1) the number of subscriptions at each router is large (which limits the scalability of QRA); (2) the data being disseminated is relatively small (which incurs only a small processing overhead for data rewriting). Thus, our propose DRA has three key advantages: it is space-efficient as it only stores the registered global subscriptions; it is also update-efficient as additions and changes to local schemas do not require updating of registered queries at the routers; and it is also time-efficient as the overhead of data rewriting is low

and the matching of the document against a (non-expanded) set of queries is fast.

To the best of our knowledge, this is the first paper that addresses the heterogeneous data dissemination problem for XML data. The only related work that addresses data heterogeneity in a pub/sub systems is a recent demonstration paper [21] that is focused on simple subscription queries (based on attribute-value pairs) and resolves only attribute names heterogeneity by enhancing the matching engine with semantic ontologies.

Organization. The rest of this paper is organized as follows. Section 2 presents our novel data rewriting framework. We discuss implementation issues for the various approaches in Section 3. Section 4 discusses related work; and our experimental results are presented in Section 5. We conclude our paper in Section 6.

2. DATA REWRITING FRAMEWORK

In this section, we present our new framework to solve the heterogeneous data dissemination problem by using *data rewriting* techniques. It is important to emphasize that our data rewriting framework is orthogonal to the specific techniques for schema integration and mapping in Section 2.3 and can be combined with other techniques as well.

2.1 System Architecture

We use S_ℓ to denote some publisher's local schema, and S_g to denote a global schema integrated from a collection of local schemas of the same domain. We use D_ℓ (resp., D_g) to denote a document conforming to schema S_ℓ (resp. S_g).

Similar to existing pub/sub systems, we have a *mediator agent (MA)* that serves as a coordinator between the data publishers and routers [9, 4]. Besides collecting schemas from publishers and registering queries for users, the MA is also responsible for resolving the structural conflicts among various schemas to generate a global schema. The MA creates a *schema mapping*, denoted by $M_{\ell,g}$, for each local schema S_ℓ that is integrated to a global schema S_g. A schema mapping $M_{\ell,g}$ is essentially a data transformation specification that enables an input document D_ℓ to be mapped into an output document D_g that preserves the appropriate information content of D_ℓ. The details of schema mappings used in this paper are discussed in Section 2.3. The *mediator agent* will distribute the schema mapping $M_{\ell,g}$ to each router. The router will leverage the $M_{\ell,g}$, i.e. the data transformation specification, to rewrite each incoming document.

Once a collection of documents that conform to a new schema have to be published, the data publisher should register the new schema to the *mediator agent* at first. The *mediator agent* takes charge of generating the mapping between the new schema and the global schema, and tries to keep the global schema stable. It may happen that the global schema has to be refined sometime, then the *mediator agent* will adjust the schema mapping for other local schemas correspondingly and will distribute a new version of schema mappings to each router.

In this paper, our data subscriptions are based on a commonly used and expressive fragment of XPath that uses only the child ("/") and descendant ("//") axes. The node test for each XPath step can be either an element name or a wildcard "*". Nested XPath expressions are allowed as predicates.

2.2 Data Rewriting Approaches

In the following, we give an overview of the proposed three data rewriting approaches. It is important to note that similar to the conventional approach and QRA, the data rewriting approaches also deliver the original document D_ℓ (and not D_g) from the publishers to the users. The purpose of rewriting D_ℓ to D_g is to enable the document to be matched against the global subscriptions. Delivering the original document to users is important as it enables users to verify the integrity of the received documents if the documents have been digitally signed [26, 13].

2.2.1 Static Data Rewriting (SDR)

In the *static data rewriting (SDR)* approach (illustrated in Fig. 2(a)), each published data D_ℓ is rewritten to D_g statically (but only once) by the MA. The advantage of employing the MA to rewrite the data is that the publishers are shielded from the details of the schema mappings and rewriting processing; this requires each publisher to first forward D_ℓ to the MA for the rewriting before the MA forwards the transformed data to the routers for dissemination.

Once D_ℓ has been rewritten to D_g, both D_ℓ and D_g are forwarded together to the network of routers for dissemination. Since the subscriptions stored in each router are based on the global schema S_g, D_g is used for matching against the subscriptions to detect matching subscriptions and decide to which router(s) the data needs to be forwarded next; D_ℓ (possibly with an attached digital signature for verification of data integrity) is forwarded to any matching local subscribers at a router.

One advantage of SDR is that it is a non-intrusive approach that can be easily implemented. However, the tradeoff is that the amount of data that is being forwarded is roughly doubled compared to the conventional approach.

2.2.2 Dynamic Data Rewriting (DDR)

To avoid the transmission overhead of SDR, an alternative strategy is for each router to forward only D_ℓ but the tradeoff is that each router now needs to rewrite the data D_ℓ *dynamically*. We refer to this approach as *dynamic data rewriting (DDR)* approach. Note that DDR does not modify D_ℓ and also does not physically materialize D_g. Instead, the rewriting of D_ℓ to D_g is performed dynamically as D_ℓ is being parsed. Specifically, the parsed events from D_ℓ are used to generate parsed events corresponding to D_g which are matched against the subscriptions, and D_ℓ is then forwarded to any matching routers/subscribers. It should be pointed out that in DDR, the original document is parsed only once, where the rewriting of data is conducted during the evaluation of subscriptions.

We have proposed two dynamic data rewriting approaches based on where the data rewriting is performed.

NDDR. The first option is to perform the rewriting outside of the matching engine by installing a new software component, called the *data rewriter*, between the document parser and matching engine as shown in Fig. 2(b). The data rewriter essentially rewrites D_ℓ to D_g by intercepting the sequence of events E_ℓ that is generated by the event-based XML parser (as it parses the input document D_ℓ) and generating a modified sequence of events E_g to the matching engine such that E_g is equivalent to the sequence of events generated by parsing D_g. We refer to this approach as *non-intrusive dynamic data rewriting (NDDR)* approach since it

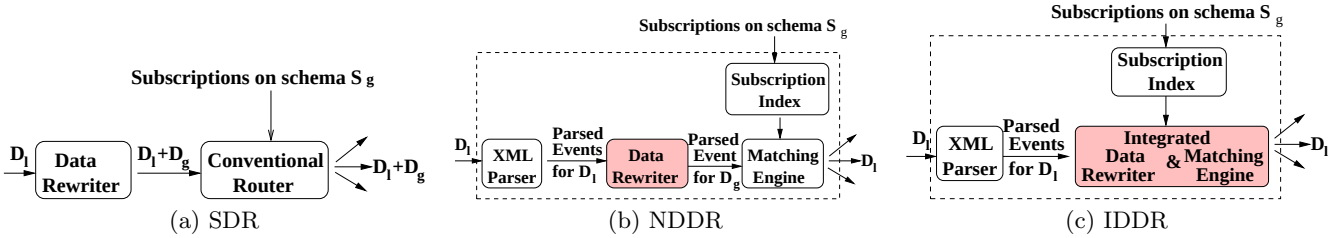


Figure 2: Data Rewriting Approaches: (a) static (b) non-intrusive, dynamic (c) intrusive, dynamic

does not require making any changes to the existing XML parser and matching engine components.

IDDR. The second option is to rewrite the data within the matching engine itself as shown in Fig. 2(c); we refer to this approach as *intrusive dynamic data rewriting (IDDR)* approach as it entails making modifications to the matching engine.

In order for a router R to perform data rewriting, R needs to have access to the schema mappings generated by the MA. There are two possible options for routers to access the schema mappings. The first option is to let MA disseminate its generated schema mappings to all the routers during an initialization process. This option is less space-efficient since the schema mapping information is replicated in every router; consequently, it is also more costly to maintain when updates arise . The second option is for each published data D_ℓ to be disseminated along with its appropriate schema mapping $M_{\ell,g}$ as part of the data's header information. In contrast, by not replicating the schema mapping information, the second option is more space- and update-efficient at the cost of a slightly higher transmission overhead. Our experimental evaluations of these two options (for both NDDR and IDDR) showed that the overhead of transmitting the appropriate schema mapping together with the data does not impact performance. For this reason, when we refer to NDDR and IDDR approaches in the rest of this paper, the second option of accessing schema mappings is assumed to be used.

2.3 Schema Mapping

A schema mapping, denoted by $M_{\ell,g}$, is a specification that enables an input document D_ℓ (that conforms to a source schema S_ℓ) to be transformed to an output document (that conforms to a target schema S_g) such that the appropriate information content of D_ℓ is preserved in D_g. Each schema mapping can be generated as part of the schema integration process. In this paper, we adopt a simple schema mapping specification that consists of a tree representation of the source schema (i.e. local schema) annotated with data rewriting operators.

It is important to emphasize that the focus of this paper is on using a *data rewriting approach* to solve the heterogeneous data dissemination problem and not on *schema mapping* per se. Thus, we have decided on a schema mapping specification that is reasonably expressive that supports a variety of data transformations (based on existing ideas [25, 28]) which is also amenable to an efficient implementation. While we make no claim that our adopted mapping scheme is complete, we believe it is sufficiently expressive as evidenced by its application in the THALIA benchmark [10]. It is important to note that our proposed data rewriting paradigm is orthogonal to the actual choice of schema mapping specification and implementation.

We model an XML schema using a tree structure, called a *schema tree*, where tree nodes represent element types and tree edges represent element-subelement relationships. Each node tree is optionally associated with a symbol (?, *, or +) that represents the cardinality of the element that it represents. For simplicity, we do not consider the union and recursion constructs in our schema model. Note that even though a XML schema typically has common substructures and can be more concisely modeled as a graph, it is often convenient to duplicate the common substructures to model the schema as a tree [15] as this makes it easy to specify different transformation operations to different instances of the same substructure. An example schema tree for S_g is shown in Fig. 3(a).

We represent a *schema mapping* $M_{\ell,g}$ by an annotated schema tree of S_ℓ. Each node in the schema tree is annotated with a (possibly empty) sequence of *data rewriting operators* (to be discussed shortly). With this schema mapping, we can transform an input data D_ℓ (conforming to S_ℓ) to a data D_g (conforming to S_g that preserves the information contents of D_ℓ) by traversing each element e in D_ℓ (in document order) and applying the sequence of rewriting operations associated with element e in the annotated schema tree. The mechanism to generate $M_{\ell,g}$ will be discussed in Section 2.3.2

2.3.1 Data Rewriting Operators

This section presents six basic data rewriting operators that can express a wide variety of data transformations. The example schema mapping $M_{\ell,g}$ shown in Fig. 3 is used as our running example to illustrate the operators. We use E, E', or E_i to denote an element type, and $child(E)$ to denote the set of child subelement types of an element type E.

Rename(E, E'): this operator renames E to E'. In Fig. 3, the operator `Rename`(department, dept) is applied to rename the `department` element in S_ℓ to `dept` in S_g. This operator is used to resolve the naming conflict between two schemas as one schema could define a department element with the full name department and another schema could define it with the short name dept.

ToElement(E, A): this operator converts an attribute A of E to become a subelement of E such that the value of A becomes the contents of the new element A. In Fig. 3, the `code` attribute of `course` element in S_ℓ is converted to be a new subelement named `code` of element `course` in S_g.

Insert$(E, E_1/E_2/\ldots/E_k,\ S)$, $S \subseteq child(E)$: this operator first moves each child subelement E' of E, where $E' \in S$, to become a child subelement of E_k, where $E_1/E_2/\ldots/E_k$

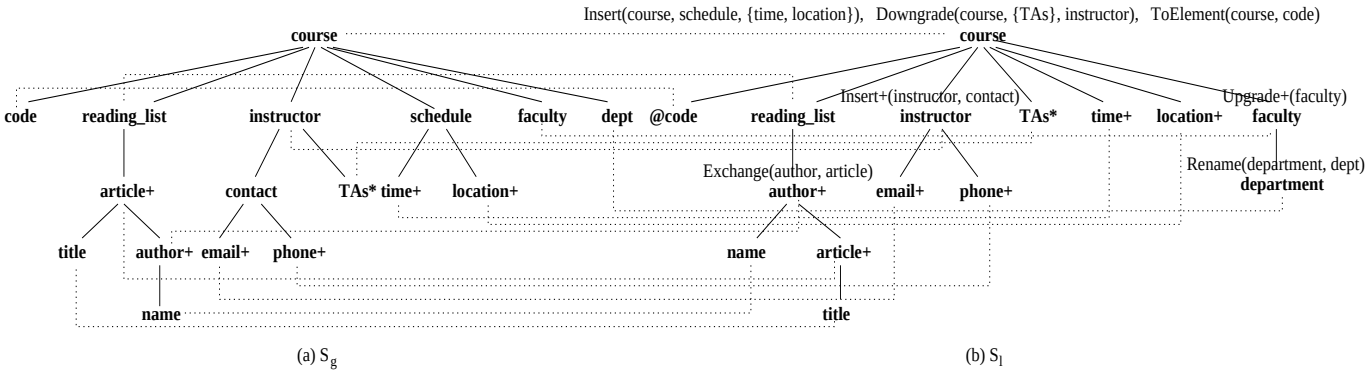


Figure 3: Example Schema Mapping $M_{\ell,g}$

is a new path of elements. The entire subtree rooted at E_1 is then inserted to become a child subtree of E. `Insert`$^+(E, E_1/E_2/\ldots/E_k)$ is a special case of the `Insert` operator that is equivalent to `Insert`$(E, E_1/E_2/\ldots/E_k,\ child(E))$. In Fig. 3, the operator `Insert`(course, schedule, {time, location}) is applied in S_ℓ to effectively group both the `time` and `location` subelements of `course` to become subelements of a new `schedule` element which is inserted as a new subelement of `course`.

Upgrade(E, S), $S \subseteq child(E)$: this operator "upgrades" each child subelement E_i of E (together with the subtree rooted at E_i), where $E_i \in S$, to become a sibling of E. Here the element E should not be the root of the XML document. `Upgrade`$^+(E)$ is a special case of the `Upgrade` operator that is equivalent to `Upgrade`$(E, child(E))$. In Fig. 3, the operator `Upgrade`$^+$(faculty) is applied in S_ℓ to move each child subelement of `faculty` (only `department` element in this example) to become a sibling element of `faculty`.

Downgrade(E, S, E'), $S \subseteq child(E)$, $E' \in child(E) - S$: this operator "downgrades" each child subelement E_i of E (together with the subtree rooted at E_i), where $E_i \in S$, to become a child subelement of E'. In Fig. 3, the operator `Downgrade`(course,{TAs},instructor) is applied in S_ℓ to move the `TA` subelement of `course` to become a child subelement of `instructor` (which is a child subelement of `course`).

Exchange(E, E'), $E' \in child(E)$: this operator swaps the roles of E and E' so that E becomes a child subelement of E'. More specifically, the subtree rooted at E (excluding the subtree rooted at E') becomes a new child subtree of E'; and the parent element of E becomes the parent element of E'. In Fig. 3, the operator `Exchange`(author,article) is applied in S_ℓ to swap their parent-child roles so that `author` becomes a subelement of `article`. Note that the `Exchange`(E, E') operator can result in the data subtree rooted at E (excluding the subtree rooted at E') to be duplicated multiple times when rewriting data. For example, if in some D_ℓ, one author has multiple articles, this author would appear multiple times in D_g. In D_g, the information for an article with multiple authors will not be merged into one subtree. This is because the `Exchange` operator is not a "group by" operator; the latter is more a complex operator that requires some notion of keys for grouping information which is outside the scope of this work.

Discussion. Our proposed rewriting operators attempt to balance the tradeoff between the expressiveness of the rewriting and the efficiency of the rewriting implementation. Thus, we have focused on structural conflicts in heterogeneous schemas, and our proposed operators are able to resolve all the schema heterogeneity scenarios mentioned in [27]. Specifically, the Rename operator resolves name conflicts; the ToElement operator resolves attribute-subelement conflicts; the Insert operator resolves generalization conflicts; the Upgrade/Downgrade operators resolve child-sibling conflicts; and the Exchange operator resolves parent-child conflicts.

2.3.2 *Deriving Data Rewriting Operators*

Given S_ℓ and S_g, $M_{\ell,g}$ can be computed in two steps. Firstly, a *schema matching* is computed from S_ℓ to S_g using some existing method (e.g., [17]). The schema matching essentially specifies a 1-to-1 mapping between the elements of S_ℓ and S_g. An example of a schema matching between S_ℓ and S_g is shown in Fig. 3 where the 1-to-1 mappings are indicated by the dotted lines. Next, the sequence of rewriting operations associated with each element e in S_ℓ (denoted by $op(e)$) is computed using the computed schema matching and the following six rules.

Given an element e in S_ℓ, we use $par(e)$ to denote the parent of e in S_ℓ, and $map(e)$ to denote the mapped element of e in S_g.

Rename Rule: If the labels of e and $map(e)$ are different, then add `Rename`$(e, map(e))$ to $op(e)$.

ToElement Rule: If e has an attribute $attr$ such that $map(attr)$ is a child of $map(e)$ in S_g, then add `ToElement` $(e, attr)$ to $op(e)$.

Insert Rule: If $map(par(e))$ is an ancestor of $map(e)$ in S_g and for each element $e_i \in p$, where p is the path from $map(par(e))$ to $map(e)$, there is no element in S_ℓ that is mapped to it, then add `Insert`$(par(e), p, e)$ to $op(par(e))$.

Downgrade Rule: If e has a sibling element e' such that $map(e')$ is a child of $map(e)$ in S_g, then add `Downgrade` $(par(e), e', e)$ to $op(par(e))$.

Upgrade Rule: If e has a child element e' such that $map(e')$ is a sibling of $map(e)$ in S_g, then add `Upgrade`(e, e') to $op(e)$.

Exchange Rule: If $map(par(e))$ is the child of $map(e)$ in S_g, then add `Exchange`$(par(e), e)$ to $op(par(e))$.

The above rules are applied in two phases. In the first phase, S_ℓ is traversed in preorder to update $op(e)$ for each visited element e using only the *Exchange* rule. Based on those $op(e)$, S_ℓ is transformed to S'_ℓ. In the second phase,

S'_ℓ is traversed in preorder to update $op(e)$ for each visited element e using only the remaining five rules. For each e visited, the rules are applied in any order to update $op(e)$ if the rule conditions are satisfied. The application of the *Exchange* rule needs to be performed first before the other rules to avoid the ambiguity on other operators caused by the *Exchange* operator. Fig. 3(b) shows the derived operators based on the schema mapping $M_{\ell,g}$ illustrated in Fig. 3.

3. IMPLEMENTATION ISSUES

In this section, we discuss the implementation issues for our two dynamic data rewriting approaches.

3.1 Non-intrusive Dynamic Data Rewriting

In NDDR (Fig. 2(b)), the key component being introduced is the *data rewriter* which is responsible for generating parsed events for D_g from the parsed events of D_ℓ thereby giving the matching engine the illusion that it is matching its global subscriptions against D_g. In this way, we can avoid changing the complex matching engine component.

Cached-Tree. To dynamically rewrite the data, the *data rewriter* needs to change the sequence of the parsed events. Some events can be forwarded to the matching engine immediately while some events have to be delayed until other events happen. For those events that are to be delayed, the *data rewriter* uses a tree structure, called a *cached-tree*, to store them in main memory. Each element in the document corresponds to one node in the *cached-tree*, which records the element's name, attributes and content value (if any). If an element e_i is the subelement of element e_j in the document, then the node corresponding to e_i is a child node of the node corresponding to e_j in *cached-tree*.

For each event start_element E received by the data rewriter, the rewriter will initiate the sequence of rewriting operations associated with element E in $M_{\ell,g}$. The complexity and therefore the cost of a rewriting operator depends on whether the operator is *blocking* or *non-blocking*. An operator is classified as *non-blocking* if the effect of its rewriting can be pipelined by the data rewriter (in the form of a parsed event for D_g) to the matching engine immediately. `Rename`, `ToElement`, `Upgrade`$^+$ and `Insert`$^+$ are non-blocking operators. An operator is classified as *blocking* if the effect of its rewriting requires the *cached-tree* to temporarily buffer some parsed events. The *data rewriter* informs the matching engine about a batch of events from the *cached-tree* once some further event is parsed. `Exchange` , `Insert`, `Upgrade` and `Downgrade` are blocking operators.

Handling *non-blocking* operators. For each parsed event from the XML parser, the data rewriter can simply pipeline the event to the matching engine. For example, given an element E with the `Rename`(E, E') operator, on receiving the start-tag for E, the data rewriter immediately forwards the start-tag for element E' to the matching engine; similarly, on receiving the end-tag for E, the end-tag for E' can again be immediately pipelined to the matching engine. Given an element E with the `ToElement`(E, A) operator, on receiving the start-tag for E, the rewriter firstly extracts the attribute A and the value of A (denoted as $val(A)$). Then the rewriter pipelines the start-tag for element E to the matching engine. After that the rewriter forwards the start-tag of element A to the matching engine followed by the $val(A)$ as the content of A, and finally the end-tag of element A to the matching engine; on receiving the end-tag for element E, the data rewriter just immediately forwards the end-tag for element E to the matching engine.

Handling *blocking* operators. On the other hand, if the associated rewrite operation for an element E is *blocking*, then the data rewriter needs to cache some relevant parsed events within the *cached-tree*.

`Exchange`(E, E'): On receiving the start-tag of element E, the data rewriter creates a cached-tree T_E and starts to cache the parsed events within the subtree rooted at E into T_E. Subsequently, when the start-tag for element E' is received by the data rewriter, the parsed events within the subtree rooted at E' will be cached into another cache-tree $T_{E'}$. When the end-tag for element E' is received, the caching into $T_{E'}$ ends and the caching into T_E resumes. Finally, when the end-tag for element E is received, the caching for T_E also ends. The data rewriter then traverses the tree $T_{E'}$ in preorder sequence, and for each node N in $T_{E'}$, the start-tag of N is forwarded to the matching engine when the node is visited for the first time; and the end-tag of N is issued to the matching engine when the traversal traces back from the node. After issuing the end-tag for E', the tree T_E is then traversed and processed.

`Insert`$(E, E_1/E_2/\ldots/E_k, S)$: On receiving the start-tag of element E, the data rewriter immediately forwards the start-tag of E to the matching engine. Meanwhile, the data rewriter creates a *cached-tree* T_E rooted at node E_1 with a child path $E_2/\ldots/E_k$. For each following start-tag of element E' which is the subelement of E, the data rewriter checks whether $E' \in S$. If $E' \in S$, the parsed events within the subtree rooted at E' is cached into T_E as the child subtree of E_k; otherwise, the data rewriter simply pipelines the event to the matching engine. When the end-tag of E is received, the data rewriter traverses T_E and issues the corresponding events to the matching engine in the same way as the `Exchange` operator. Finally, the data rewriter forwards the end-tag of E to the matching engine.

`Upgrade`(E, S): On receiving the start-tag of element E from the parser, the data rewriter forwards it to the matching engine immediately. For each element E' which is the subelement of E and $E' \in S$, the data rewriter caches the parsed events within the subtree rooted at E' into a cached-tree $T_{E'}$. The caching ends when the end-tag of element E' is received. On receiving the end-tag of element E, the data rewriter first forwards the end-tag of E to the matching engine, and then it traverses the set of *cached-tree* $T_{E'}$ and issues the corresponding events to the matching engine.

`Downgrade`(E, S, E'): When the start-tag of element E' is received, the data rewriter creates a *cached-tree* $T_{E'}$ to cache the subtree rooted at E'. When the start-tag of element e, where $e \in S$, is received, the parsed events within the subtree rooted at e will be cached into another *cached-tree* T_e by the data rewriter. These cached elements could be issued to the matching engine when the end-tag of element E is received. The data rewriter then traverses $T_{E'}$ in preorder and forwards the corresponding events to the matching engine. Before forwarding the end-tag of element E', the data rewriter traverses T_e to forward the events in T_e to the matching engine. Finally, the end-tag of E' is forwarded followed by the end-tag of E to the matching engine.

By judiciously caching the appropriate subtrees and blocking the output of parsed events, this ensures that the correct

parsed events are output corresponding to the effect of different *blocking* operations.

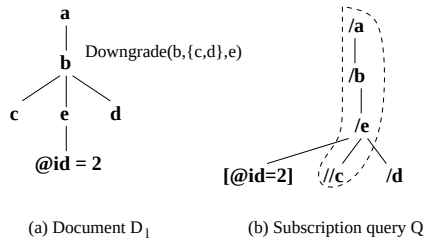
3.2 Intrusive Dynamic Data Rewriting

Among the three data rewriting approaches, IDDR (Fig. 2(c)) is the most complex to implement as it is an intrusive approach that necessitates modifying the matching engine so that it integrates both the dynamic rewriting functionality as well as the subscription matching functionality. To realize this dual functionality efficiently, the matching engine actually maintains partial matchings of subscriptions based on the assembled fragments of D_g that are rewritten from the parsed events of D_ℓ. In this way, we do not need to first materialize the rewritten data D_g before the subscription matching can commence.

To understand why matching in IDDR becomes more complex than the conventional matching in SDR and NDDR, note that the matching engine works by maintaining partial matches of subscriptions as the document is being parsed and the parsed events are being incrementally processed. Once a start-tag for an element E is encountered, the matching engine updates any partial matchings with the new element E at the current context; and once an end-tag for element E is encountered, the matching engine eliminates the partial matchings that are guaranteed to not lead to any complete matchings. The matchings of the elements and the elimination of partial matchings are based on two basic properties of conventional event-based XML parsers: (1) once the start-tag event for an element E is received, all the ancestor elements of E must necessarily have been parsed; and (2) once the end-tag event for an element E is received, all the descendant elements of E must necessarily have been processed. Based on the first property, the matching engine can detect all the partial matchings involving element E for the start-tag event for E; and based on the second property, when end-tag event for element E is encountered, the matching engine can safely eliminate all partial matchings that entail the matchings in the subtree rooted at E.

However, the above two properties that facilitate the updating of partial matchings are no longer applicable for IDDR for two reasons. Firstly, some elements in D_g may have been parsed earlier than their ancestor elements. For example, the operator $\texttt{Downgrade}(E, S, E')$ will move the subtree rooted at E_i, where $E_i \in S$ to become a child subtree of E'. Consequently, element E_i may precede element E' in the document such that the start-tag of E_i is output by the parser before the start-tag of E'. This means that the matching engine has to process E_i without its ancestor element E'. Secondly, when the end-tag of element E_i is encountered, it may happen that not all of E_i's descendants in D_g have been parsed. Consider again the operator $\texttt{Downgrade}(E, S, E')$. When element E' precedes element E_i, where $E_i \in S$ in the document, the end-tag for E' is reported by the parser before element E_i which should be the descendants of E'. The operators Exchange(E, E') and Insert$(E, E_1/E_2/\ldots/E_k,\ S)$ face this issue as well.

To handle this complexity, the integrated matching engine therefore maintains two types of partial matchings. Given a start-tag for element E, if all its ancestors have already been parsed, then the partial matchings detected for E are confirmed. We call such partial matchings as *confirmed partial matchings*; otherwise, if some of its ancestor elements have yet-to-be-parsed, the partial matchings detected by element E cannot be determined. We call such matchings as *potential partial matchings*. The matching engine maintains both *confirmed partial matchings* and *potential partial matchings* that are detected for E. Once an element that is relevant to the potential partial matchings has been parsed, the matching engine uses this element to verify the potential partial matchings. The successfully matched potential partial matchings are handled in the same way as the confirmed partial matchings. To handle the second problem that the descendant elements of an element E could be parsed after the end-tag of E, the matching engine continues to keep the partial matchings that can be combined with the matchings from the descendant elements of E to generate larger matchings. These partial matchings are eliminated once the matching engine determines that all the descendants of E have been processed.



(a) Document D_1 (b) Subscription query Q

Figure 4: IDDR Example

Example 3.1 Consider the document D_ℓ and query Q in Figs. 4(a) and (b), respectively. Suppose that the operator $\texttt{Downgrade}$ $(b, \{c, d\}, e)$ is to be performed on D_ℓ. When the start-tag for element c is received by the matching engine, the engine knows that element c should be matched under element e, which has yet to be parsed. Thus, the matching engine can only detect the partial matching $/a/b/e//c$ as a potential partial matching, which is shown by the path of query nodes enclosed by the dashed region in Fig. 4(b). This is because the matching engine does not yet know whether the element e contains an attribute "id" with a value of 2. Subsequently, when the start-tag of element e is parsed, the *potential partial matching* $/a/b/e//c$ is confirmed. When the end-tag for element e is encountered, since the matching engine knows that there might be some elements that need to be downgraded as descendants of e, the partial matchings detected by e are still maintained. After processing the start-tag of d, the complete matching of query Q is detected. Notice that if Q had not been matched when the end-tag of element b is encountered, the partial matchings detected by element e can be eliminated since it is guaranteed that no elements will be processed as descendant elements of e. □

4. RELATED WORK

Many approaches have been proposed to address the efficient dissemination of XML data in pub/sub system by exploiting some index structure on XPath expressions [2, 7, 8, 11, 19, 24]. However, they assume the context of homogeneous schema which is orthogonal to our work.

The only work that addresses the heterogeneous data dissemination problem that we are aware of is the demonstration paper [21]. The focus in that paper is on simple subscriptions (based on attribute-value pairs) with attribute

name heterogeneity. Their solution uses semantic ontologies within the matching engine to resolve the attribute name heterogeneity. In contrast, our work addresses the problem in the context of XML data with more complex XPath-based subscriptions, and the scope of schema heterogeneity examined in our work is much broader involving both structural heterogeneity as well as attribute name heterogeneity. Our proposed approach is different from theirs.

The use of query rewriting techniques for querying heterogeneous data is a well studied area [16, 28, 6, 29]. As discussed in the introduction, applying the query rewriting idea to solve the heterogeneous data dissemination problem has low space-efficiency and high update cost due to the different nature of the problems (single-query-multiple-data vs single-data-multiple-queries).

In terms of work on data translation, an early work by Milo and Zohar [18] uses rules derived from schema mapping to perform data translation. Their focus is on translating data from one format to another (e.g., from SGML to OODB). Thus most of their rules are proposed to address the difference of schema definitions for the various data formats, which is not the case in our work. While there are other approaches that address the data transformation problems [22, 12, 5], these approaches would require an document to be parsed twice due to the separate data transformation and query evaluation procedures.

Su et. al [25] introduce a set of transformation operators at the schema level to measure the transformation cost for mapping one schema to another. Some of their operators, such as *fold* and *unfold*, which do not affect the query matching results, are not relevant for our problem; their transformations do not address the parent-child structural conflicts handled by our *exchange* operator.

5. PERFORMANCE STUDY

To demonstrate the effectiveness of our proposed data rewriting approaches, we conducted extensive experiments to compare these approaches. Our results show that IDDR and NDDR outperform SDR under various conditions.

5.1 Experimental Testbed

We experimented using both the NS2 network simulator [1] (extended with application code for content-based routing) as well as a real network (denoted as *real*). For types of topology, we used both linear and tree structures (a complete binary tree with four levels and a total of 15 routers) for the network topology. The bandwidth of network is varied from 10, **50**, 100 (Mbps). For our experiments on the real network, we used a linear topology consisting of four computers connected in a LAN.

Data Sets. We used the THALIA benchmark [10], which contains 40 similar XML schemas representing various university course catalogs. Based on the collection of similar XML schemas, we manually created a global schema and a schema mapping between each local schema and the global schema. The breakdown of the total number of rewriting operators used by our schema mappings for the 40 local schemas are as follows: 89 `Rename`, 31 `Insert`, 17 `ToElement`, 6 `Upgrade`, 3 `Downgrade`, and 2 `Exchange`. Observe that among the 148 rewriting operators used for the schema mappings, about 72% of these operators are *non-blocking* (i.e., `Rename` and `ToElement`).

Documents were generated using the THALIA benchmark for each of the 40 schemas. We vary the size of data sets from 10K, 20K, 40K to 1M where a data set size of x actually represents a size in the range $[x, x + 10\text{KB})$.

Subscriptions. The XPath queries were generated using the XPath generator in [8], where the maximum number of steps is set to be 8; the probability of wildcard $*$ and the probability of nested paths are both set to be 0.2.

Algorithms and Metric. We studied the performance for SDR, NDDR and IDDR. We use the approach XTrie [7] as the matching engine at each router; however, note that the specific matching engine used is orthogonal to our proposed data rewriting approach so long as the matching engine can be enhanced for the IDDR approach.

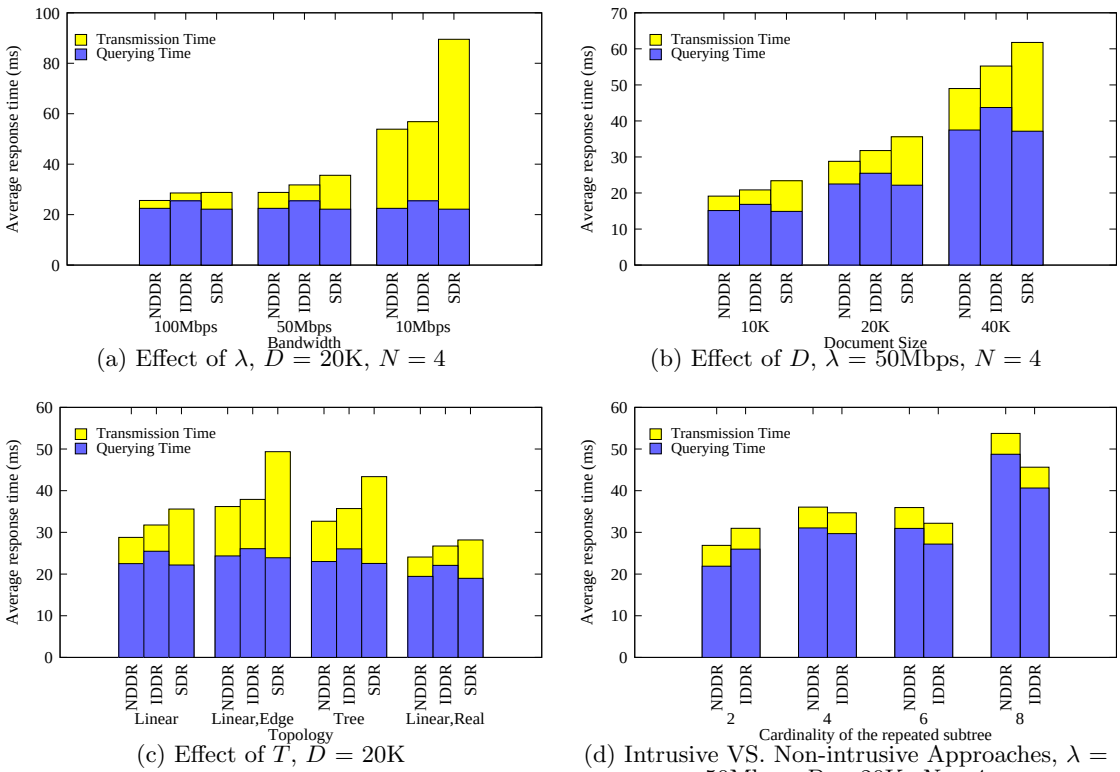
The performance metric used is the *average response time* (ms), which is defined as the average time for a document to be delivered to all relevant users. The average response time (denoted by T) is comprised of two components: (1) *querying time* (denoted by T_q) which is the CPU time incurred for parsing the document, dynamically rewriting the document (for NDDR and IDDR), and matching the document against the queries; and (2) *transmission time* (denoted by T_t), which is the time incurred for transmitting data in the network. Thus, $T = T_q + T_t$. In the following ,we use the stacked barcharts to show the average response time T and its two components, i.e. T_q and T_t.

Our experiments were conducted on a 3GHz Intel Pentium IV machine with 1GB main memory running Windows XP, and all algorithms were implemented in C++.

5.2 Experimental Results

Comparison of different approaches. The middle bars in Fig. 5(a) compare the performance of different approaches by setting $\lambda = 50$Mbps and $D = 20$K. Firstly, it shows that the dynamic data rewritten approaches (i.e. NDDR and IDDR) outperforms the approach SDR. NDDR obtains similar querying time with SDR, which means that the additional cost for dynamic data rewriting in NDDR is trivial. The amount of data transmitted in SDR is about twice of the amount in NDDR and IDDR, thus SDR incurs much larger T_t. Therefore, the performance of SDR is outperformed by NDDR and IDDR. NDDR and IDDR have the same T_t. However, due to the complicated matching algorithm in IDDR, it incurs slightly larger T_q. Thus NDDR achieves better performance than IDDR.

Effect of the bandwidth, λ. We demonstrate the effect of network bandwidth in Fig. 5(a) by decreasing λ from 100Mbps, 50Mbps, to 10Mbps. As λ decreases, the components of T_t for each approach grow. The effect of λ to NDDR and IDDR is the same, since they transmit same amount of data. SDR deteriorates faster as the decreasing of λ, since the amount of data transmitted in SDR is twice of NDDR and IDDR. For $\lambda = 100$Mbps, the component of T_t is very small. When $\lambda = 50$Mbps, the component of T_t takes a small part of the response time. However, when $\lambda = 10$Mbps, the component for T_t takes a large part of response time, especially for SDR that T_t is about 78% of the response time. Thus as λ decreases, the improvement of NDDR over SDR increases from 12% at $\lambda = 100$Mbps to 41% at $\lambda = 10$Mbps. The internet develops fast in recent years, however the bandwidth is still the critical resource, which makes SDR not suitable for small bandwidth environment.



(a) Effect of λ, $D = 20$K, $N = 4$

(b) Effect of D, $\lambda = 50$Mbps, $N = 4$

(c) Effect of T, $D = 20$K

(d) Intrusive VS. Non-intrusive Approaches, $\lambda = 50$Mbps, $D = 20$K, $N = 4$

Figure 5: Experimental Results

Effect of the document size, D. Fig. 5(b) shows the performance by varying D from 10K, 20K, 40K to 1M, while $\lambda = 50$Mbps. As D increases, the *average response time* for all approaches increases due to larger T_q and T_t. We observe the performance gap between NDDR and IDDR becomes slightly larger, since larger documents have more affected on IDDR due to the more complicated matching algorithm in IDDR. It also shows that the improvement of NDDR over SDR becomes larger as D increases, from 20% at $D = 10$K to 25% at $D = 40$K. Similarly, the improvement of IDDR over SDR also increases. The reason is that larger documents incur larger transmission delay in SDR. The results for 1M dataset are omitted here since its average response time is much larger than other datasets, which is not suitable to be shown in the same chart. We observe that the trends from 10K to 40K also keeps at $D = 1M$, that is the improvement of NDDR over IDDR is 10%, and over SDR is 28%.

Effect of #subscription per router, P. We performed an experiment to vary P from 1000 to 2000 to 4000, while the result is not shown here. As P increases, T_q for all three approaches increase correspondingly. The increasing of querying time for NDDR and SDR is the same, thus the performance gap between NDDR and SDR keeps the same. However, due to its complicated matching algorithm, the increasing of T_q for IDDR is larger NDDR and SDR, thus improvement of NDDR over IDDR becomes larger.

Effect of the network topology. This section studies the effect of the network topology on the performance. Firstly, we test the case when only leaf router (the router without the downstream router) has the subscriptions from the users. The results are shown by second cluster of bars in Fig. 5(c). The transmission to the leaf router incurs larger delay compared with upstream routers. Thus the performance gap between NDDR (also IDDR) and SDR becomes larger. Then, we show the results on the $Tree$ topology using the third group of bars in Fig. 5(c). Compared with $Linear$, the $Tree$ topology has more routers as the leaf routers. As aforementioned, queries on leaf routers incur larger T_t, thus the performance margin between NDDR (as well as IDDR) and SDR becomes larger.

Results on the real network. We also experimented on a real network *Real* as described in Section 5.1. The forth group bars in Fig. 5(c) show the results on *Real*. As we known, the bandwidth in the LAN is usually large. The bandwidth in the LAN we used is more than 50Mbps. We can see that the performance of all approaches on *Real* have similar trends with the performance of them on NS2 with λ = 50Mbps. NDDR achieves the best performance among all approaches. It proves that the simulation using NS2 measures the performance well.

NDDR vs. IDDR. The previous results show that IDDR is slightly outperformed by NDDR since IDDR makes the matching algorithm more complicated. However, in some situation, IDDR is more efficient than NDDR by sharing the processing of repeated subtrees. For example, operator $Exchange(N_1, N_2)$ makes the subtree rooted at N_1 be repeated at the subtree rooted at N_2 if the cardinality of element N_2 in document is larger than 1. In this experiment, we select the document that contains the operator $Exchange(N_1, N_2)$ and vary the cardinality of N_2 (denoted

as r) from 2 to 8 in the step of 2. We observe that when $n = 2$, NDDR is better than IDDR due to the complicated matching in IDDR. However, when $n = 4$, IDDR starts to outperform NDDR, and as n increases, the improvement of IDDR over NDDR becomes larger. The reason is that in IDDR, the processor is aware of the data rewriting, and knows that same subtree rooted at N_1 is repeated under each N_2, thus IDDR shares the processing of the subtree rooted at N_1. As n increases, the improvement of IDDR by sharing the repeated subtree becomes larger, which compensates the performance loss due to complicated matching algorithm.

5.2.1 Discussion

Based on our experimental results, we have the following observations on the efficiency of various approaches. First, to disseminate schema mapping $M_{\ell,g}$ as the data's header information incurs little overhead, and this approach is space- and update-efficient. To compare among IDDR, NDDR and SDR, SDR does not perform well due to the transmission of additional data, especially when the bandwidth is small or the number of hops to subscribers is large. Moreover, IDDR does not scale well as the number of subscriptions or the size of documents increases since it complicates the matching algorithm. Generally speaking NDDR achieves the best performance, and we have measured that the memory usage in NDDR of most documents is around 5% of the document size, which is small enough. However, IDDR performs better than NDDR when there are many duplicated subtrees in the rewriting of D_ℓ to D_g.

6. CONCLUSIONS

In this paper, we have introduced the heterogeneous data dissemination problem for XML data and have proposed a novel paradigm based on the principle of *data rewriting* to address the problem. We have explored several architectural options and their tradeoffs for this new approach. Our experimental results show that the *non-intrusive dynamic data rewriting* approach has the overall best performance.

Acknowledgements This research is supported in part by NUS Grant R-252-000-237-112.

7. REFERENCES

[1] NS2. version ns-2.1b8. http://www.isi.edu/nsnam/ns/.
[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, 2000.
[3] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for XML. In *VLDB*, 2005.
[4] M. Antollini, M. Cilia, and A. Buchmann. Implementing a high level pub/sub layer for enterprise information systems. In *ICEIS*, 2006.
[5] A. Boukottaya and C. Vanoirbeek. Schema matching for transforming structured documents. In *DocEng*, 2005.
[6] S. D. Camillo, C. A. Heuser, and R. dos Santos Mello. Querying heterogeneous XML sources through a conceptual schema. In *ER*, 2003.
[7] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB*, 11(4), 2002.
[8] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4), 2003.
[9] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.
[10] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA : Test harness for the assessment of legacy information integration approaches. In *ICDE*, 2005.
[11] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.
[12] H. Jiang, H. Ho, L. Popa, and W.-S. Han. Mapping-driven XML transformation. In *WWW*, 2007.
[13] H. Khurana. Scalable security and accounting services for content-based publish/subscribe systems. In *SAC*, 2005.
[14] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: Scalable XML document filtering by sequencing twig patterns. In *VLDB*, 2005.
[15] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, 2001.
[16] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneouse data sources. In *VLDB*, 2001.
[17] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding : a versatile graph matching algorithm and its application to schema matching. In *ICDE*, 2002.
[18] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.
[19] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *VLDB*, 2007.
[20] Y. Ni, C.-Y. Chan. Dissemination of Heterogeneous XML Data. In *WWW(Poster)*, 2008.
[21] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS : semantic Toronto publish/subscribe system. In *VLDB*, 2003.
[22] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *VLDB*, 2002.
[23] T. Schlieder. Schema-driven evaluation of approximate tree-pattern queries. In *EDBT*, 2002.
[24] P. Silvasti, S. Sippu, and E. S. Soininen. Schema conscious filtering of XML documents. In *EDBT*, 2009.
[25] H. Su, H. Kuno, and E. A. Rundensteiner. Automating the transformation of XML document. In *WIDM*, 2001.
[26] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *HICSS*, 2002.
[27] X. Yang, M. L. Lee, and T. W. Ling. Resolving structural conflicts in the integration of XML schemas : a semantic approach. In *ER*, 2003.
[28] X. Yang, M. L. Lee, T. W. Ling, and G. Dobbie. A semantic approach to query rewritting for integrated XML data. In *ER*, 2005.
[29] C. Yu and L. Popa. Constraint-based XML query rewritting for data integration. In *SIGMOD*, 2004.