Scalable Data Integration by Mapping Data to Queries

Martin Hentschel[1], Donald Kossmann[1], Daniela Florescu[2], Laura Haas[3], Tim Kraska[1], and Renée J. Miller[4]

[1] Systems Group, Department of Computer Science, ETH Zurich, Switzerland
[2] Oracle, Redwood Shores, USA
[3] IBM Almaden Research Center, San Jose, USA
[4] University of Toronto, Toronto, Canada

martin.hentschel@inf.ethz.ch, donald.kossmann@inf.ethz.ch, dana.florescu@oracle.com, laura@almaden.ibm.com, miller@cs.toronto.edu, tim.kraska@inf.ethz.ch

Abstract The goal of a data integration system is to allow users to query diverse information sources through a schema that is familiar to them. However, there may be many different users who may have different preferred schemas, and the data may be stored in data sources which use still other schemas. To integrate data, mapping rules must be defined to map entities of the data sources to entities of the users' schemas. In large information systems with many data sources which serve sophisticated applications, there can be many such mapping rules and they can be complex. The purpose of this paper is to study the performance of alternative query processing techniques for data integration systems with many complex mapping rules. A new approach, mapping data to queries (MDQ), is presented. Through extensive performance experiments, it is shown that this approach performs well for complex mapping rules and queries, and scales significantly better with the number of rules than the state of the art, which is based on query rewrite. In fact, the performance is close to that of an ideal system in which there is only a single schema used by all sources and queries.

1 Introduction

Data integration is a mess. In any realistic scenario there is a hodge-podge of many different schemas. Often, several different schemas are used within a single document. In the health care industry, for example, hospitals and doctors may use different schemas in order to describe different diseases and therapies. As a result, a single electronic health record may contain entries with varying schemas from different hospitals and doctors. Likewise, lineitems of purchase orders may be represented using different schemas, as the result of customizations carried

Technical Report 633, Department of Computer Science, ETH Zurich, July 2009

out to the order processing software for specific classes of products, delivery terms, or business models.

The diversity of schemas is not going away. As the world becomes flatter and more interconnected due to globalization, data from more and more sources need to be integrated. Even international standardization bodies such as HL7 [14] and XBRL [32] have realized that it is impossible to find *the* "one size fits all" schema for a particular domain. Instead, these organizations focus on standardizing *conventions* for the design of schemas in order to facilitate interoperability across schemas. Rather than trying to eliminate diversity, the goal must be to live with an increasing diversity. This paper picks up this challenge.

In order to integrate data from diverse sources, *mapping rules* are required. Mapping rules relate concepts from one schema to concepts of another schema. Mapping rules decouple the complexity of data integration from the task of developing applications. Application developers can write queries and updates according to one specific target schema without being aware in which format and schema the data are actually stored.

The state of the art in processing queries and updates in the presence of mapping rules is to *rewrite* the queries and updates according to the mapping rules. Using this approach, queries issued over one schema are translated at compile time using the mapping rules into queries over mapped schemas. These translated queries are often quite large, containing many unions and disjunctions. As an example, if a query asks for all *orders* and there is a rule that specifies that all *purchase-orders* are *orders*, then the query is rewritten to a query that asks for all *orders OR purchase-orders*. If a query asks for the *grossprice* of all *orders* and there are mapping rules that specify that all *purchase-orders* are *orders* and that *netprice* in *purchase-orders* can be used to compute *grossprice* (using a specific transformation function f), then query rewrite would rewrite the query to find the *grossprice OR f(netprice)* of all *orders OR purchase-orders*. While such queries can be optimized by modern database systems, these examples demonstrate that the query rewrite approach does not scale well with the number of mapping rules.

The main contribution of this paper is an alternative query processing technique which scales well with the number of schemas and mapping rules. We call this technique *mapping data to queries (MDQ)*. Rather than rewriting queries at compile time, MDQ takes a *lazy* approach and annotates the data at runtime; only data that potentially matches the query is annotated. In order to scale with the number of mapping rules, the mapping rules are indexed so that they can be probed while the data is processed at runtime. This paper shows that MDQ is simple to implement and has excellent performance. It can scale to handle thousands of schemas and tens of thousands of mapping rules. In fact, the performance is close to that of a (theoretically) optimal system without any heterogeneity or mapping rules in which all data sources and application programmers agree on a single schema.

By shifting the bulk of the work to runtime, MDQ can take advantage of the data itself to reduce the work. By contrast, query rewriting happens at compile time, independent of the data. Hence, even if very few of the mapping rules

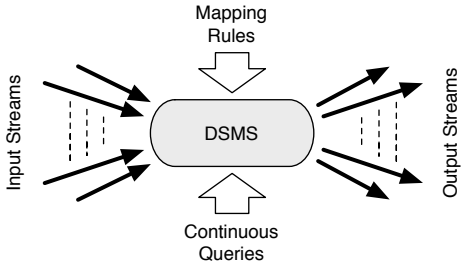


Figure 1. DSMS Integration Scenario

will produce data satisfying the query, the entire rewritten query is executed. Even with sophisticated optimizations, too much is unknown at compile time, resulting in extra work for the query rewriting approach. Another benefit of MDQ is that no schemas are actually needed (though they are not precluded); data can be directly examined and returned on an as-needed basis. This makes MDQ appropriate for environments with dynamic and evolving data formats.

While the idea of mapping data to queries is general and can be applied to any data integration system, we have implemented it in the context of a *data stream management system (DSMS)*. In this environment (Figure 1), incoming data feeds contain data in different formats (that is, conforming to different schemas) and, potentially, even mixed formats. From these feeds, a set of continuous queries, possibly written against still other schemas, produces a set of output streams, leveraging mapping rules to ensure that the right data are returned. Users can add mapping rules and queries independently at any time, and new data (in potentially new formats) can arrive at any time.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 covers necessary background, including the framework we adopt, the rule language used for our examples, query rewrite, in general, and our implementation in a DSMS. Section 4 presents the main contribution of this work, the mapping data to queries technique (MDQ). Section 5 shows the results of our performance experiments. Section 6 concludes and offers avenues for future work.

2 Related Work

The discovery and design of schema mappings has become a mature field, led by research projects such as Clio [22, 26], and covering both relational and XML data. Products leveraging these technologies include the Clio-based Rational Data Architect [10] and the design editor of BEA's AquaLogic Data Services Platform [4]. Lately there has even been work on a benchmark for mapping tools [2]. This field of work is orthogonal to our work in this paper; our work assumes that the mapping rules are already known and the goal is to process queries as efficiently as possible in the presence of mapping rules.

Given a set of mapping rules, the state-of-the-art in query processing for data integration is to rewrite queries according to the mapping rules. For declarative LAV (local-as-view) mappings, query rewriting is based on an approach called *answering queries using views* [19]. Query rewriting for LAV relational schemas has been considered in MiniCon [27] which evaluates the scalability of the rewriting algorithm for queries over a single (global) schema using large numbers of mappings (views). For XML data (and combinations of relational and XML data), there are several proposals for query rewriting [33, 25]. Again the assumption in evaluation is that queries are written against a single (global or target) schema, though the rewriting may use potentially large numbers of mappings or views. These solutions show the scalability of the rewriting process itself (with respect to the number of views), but they do not address the performance of actually executing the rewritten queries. Improving the runtime performance of queries is the primary goal of our work.

Query rewrite is also the standard technique used for information integration in peer-to-peer systems like Piazza [12] and pay-as-you-go systems like iTrails [28]. In Piazza, queries may be written on any schema and are answered using data in a set of networked data sources using query rewriting and mapping composition. Piazza uses GLAV mappings for which query answering may be undecidable. The main contribution of iTrails is a coloring algorithm in order to deal with transitivity and cyclic mapping rules.

Query rewrite and MDQ are ways to implement query processing for *virtual data integration*. In this approach, the data is stored and queried from the heterogeneous data sources and feeds in its original formats. An alternative approach is to *cleanse* and *transform* the data *before* it is processed. This approach has been advocated for data warehouses and its main principle can also be applied for streaming data. This approach is beneficial if users can agree on a single (or few) "gold standard" schemas and if the investment to transform the data is amortized by running many queries on the transformed data. The goal of our work is to explore new (virtual) query processing techniques if these conditions are not met.

The Semantic Web also provides tools for data integration. RDF [18] and OWL [20] are ways to specify mappings between concepts in an open way. In some sense, RDF and OWL are more powerful than the mapping rules traditionally considered for database integration; for instance, OWL supports negation. On the other hand, RDF and OWL are less powerful because they do not allow the specification of complex mapping functions such as the concatenation of *first name* and *last name*. It is an important avenue for future work to study how the query processing techniques for data integration studied in this work can be extended to support OWL features such as negation.

As mentioned in the introduction, this work studies data integration in the context of data stream management systems and continuous query processing. Obviously, there has been a great deal of work in this area; e.g., [30, 6, 5, 23, 1]. All this work can be leveraged to implement a *heterogeneous* DSMS as depicted in Figure 1. To the best of our knowledge, however, there has not been any prior work that studies the processing of queries on heterogeneous data streams.

3 Background

In this section, we present the framework and goals of our work. We define the mapping rules we use, and explain how query rewrite employs these mappings for query processing. We also sketch the prototype implementation of query rewrite used for comparison with our new *Mapping Data to Queries* technique in the performance experiments of Section 5.

3.1 Framework

Our solutions are implemented in the context of a data stream management system. We will use the term *message* to refer to a unit of stream data over which a set of continuous queries will be processed. A message may be an XML document or (more commonly) a fragment of XML such as an element. A message can also be a fragment of JSON or indeed any format that can be serialized.

Messages from different data sources may have different schemas. Likewise, queries from different users and applications may have different schemas. If the messages involve purchase orders, for example, then the individual message formats may be tuned to the needs of particular business units or be dynamically modified to accommodate changing client needs. Hence, our solutions must process data when there is not a set of predefined schemas for all sources or when there are no predefined schemas for consumption of messages by queries.

Our implementation of a DSMS is done by extending Saxon, an open-source XQuery engine. Saxon is a full-function XQuery processor, conforming to the XQuery 1.0 specification. We simulate the streaming environment on top of Saxon, running pre-compiled queries against messages as they arrive. It was important to use a full-fledged XQuery processor for this work, both to realistically mimic the environment of our motivating applications, and to enable us to run standard benchmarks for our performance experiments (Section 5). We do not impose any restrictions on the queries.

3.2 Mapping Rules

To study the performance of queries over large numbers of mappings and schemas, we use an intuitive, yet powerful mapping formalism based on XQuery. We are making no innovations in the mapping formalism, but rather have been guided by the practical considerations described above in our choice. The mapping formalism encompasses the most common types of mappings used in practice, and avoids mappings for which query rewriting algorithms are not yet known. Query rewriting is based on query containment, a notoriously hard problem for XQuery. We avoid this problem by restricting our mappings to a set for which the containment algorithms of Miklau and Suciu [21] can be used.

Rules of varying degrees of complexity are needed for mappings, beginning with specifications of *is-a* relationships and *sameAs* relationships, and ranging to the ability to restructure data and to compute complex functions. As is

common, a mapping rule relates a query over a source to a query over a target. More specifically, mapping rules are described using the following syntax:

source-expr [**as** $*variable*] → *target-expr*

In this syntax, **as** and → are keywords. The source expression *source-expr* is any XPath 1.0 expression [7] with forward axis steps only.

For example, consider the XML elements shown in Figure 2, representing orders from two different business units in a large vendor where there may be dozens of business units each with their own version of an order. The simple rule

purchase-order → *order*

states that a `purchase-order` element can be used in lieu of an `order` element. The consequence is that a `purchase-order` element *matches* the query *//order*. The `purchase-order` element would be returned in its original form; it would not be transformed into an `order` element. The `purchase-order` is returned with all of its sub-elements, even if these sub-elements have not been mapped *a priori* to elements in other schemas. If, in addition to this first rule, we define the inverse rule (*order* → *purchase-order*), the two rules together specify that these elements are equivalent. In that case, both elements would match queries asking for `order` and/or `purchase-order`.

Predicates are allowed in source expressions and may involve any XQuery expression (including reverse axes). Predicates can be used to restrict the context for a mapping rule. In Figure 2, the `netprice` in the `purchase-order` element may be equivalent to the `grossprice` for `order`s if the `tax` is 0. This can be expressed using the following rule.

netprice[../purchase-order/tax=0] → *grossprice*

The use of predicates to scope the applicability of a rule only works for the source. This example also demonstrates the use of path expressions to navigate through the source. Path expressions can also be used to map elements at different levels of nesting, e.g.,

item → *orderlines/orderline*

This rule implements the unnesting of `orderline` elements in the second schema of Figure 2. In general, the target expression can be any XPath expression with only *child* steps and no predicates. Alternatively, a target expression can be an element constructor with any XQuery expression to compute the content of the element. The target expression may use the $*variable* which is bound to the *source-expr* in the **as** clause. The target expression does not allow any other free variables (i.e., existentials). Nevertheless, this framework allows us to express extremely powerful mappings because the full power of XQuery is applicable in predicates over the source and in element constructors.

As a final example, the following rule shows the use of an element constructor when the elements (and their sub-elements) in different schemas do not match

```
<order  customer='Sue'>
  <orderlines>
    <orderline>...</orderline>
    <orderline>...</orderline>
  </orderlines>
  <grossprice>45</grossprice>
</order>

<purchase-order  cust='Josh'>
  <item>...</item>
  <item>...</item>
  <netprice>35</netprice>
  <tax>10</tax>
</purchase-order>
```

Figure 2. Two purchase orders

directly. This rule models the mapping of a `netprice` to a `grossprice` for the data snippets of Figure 2. Such rules are frequent in practice, and this example motivates the need for a powerful expression language like XQuery in order to define such complex mappings. Notice that the *target-expr* uses the variable *$n* which is bound by the **as** clause to instances of the *source-expr*. Any sub-element of the source expression could also be used in the target element constructor.

netprice **as** $n \rightarrow$
  <grossprice>
    { *$n + $n * $n/../tax div* 100 }
  </grossprice>

The rules described above are quite general. As our examples show, the mapping rules also have significant expressive power (by leveraging XPath and XQuery as languages to define artifacts which are substitutable or equivalent). In addition, rules can be composed, e.g., from $a \rightarrow b$ and $b \rightarrow c$, it can be inferred that $a \rightarrow c$. Recursive rules such as $a \rightarrow a/a$ can also be specified. Our implementations of query rewrite and of mapping data to queries both handle composition and recursion in the rules. A more detailed description of what the rules can do and their semantics may be found in [15].

3.3 Query Rewrite

In data integration scenarios, queries are typically rewritten at compile time based on the mapping rules. The key idea is to generate *union* expressions whenever the right-hand side of a mapping rule is subsumed by a sub-expression of the query. A simple example is given in the introduction: The query *//orders* is rewritten to *//(orders | purchase-orders)* given the rule *purchase-orders* $\rightarrow$ *orders*.

Rewritten queries can become very large. Continuing the example from the introduction, given rules *purchase-orders* → *orders* and *f(netprice)* → *grossprice*, the query *//orders/grossprice* is rewritten to the following query.

//(orders/grossprice | orders/f(netprice) |
purchase-orders/grossprice | purchase-orders/f(netprice))

If every business unit in our large vendor defines its own namespace and vocabulary, then the size of even simple queries grows linearly with the number of business units, which impacts the running time of the rewritten query. For complex queries, the situation can get even worse, as will be shown in Section 5. Of course, not all mapping rules affect every query; only the relevant rules will be applied in the rewriting. Still, monster queries that take all business units into account are generated, even if only a few business units have relevant data. The pool of mapping rules must be carefully administered because bad rules (for example, rules that apply to no, or very little data or queries) hurt the performance of the whole system.

Another disadvantage of query rewriting is that queries must be recompiled whenever new mapping rules are added to the system. (All queries must be checked against the new rule, even if it proves not to be relevant to the query.) In addition to the overhead and time to do the recompilation, in our environment, recompilation of a continuous query involves temporarily disabling the continuous query which impacts the availability of the system.

The major advantage of the query rewrite approach is that after the rewrite, the query compiler can simplify and optimize the query. After query rewrite, all state-of-the-art query processing techniques are applicable. In fact, the query rewrite approach can be implemented using a pre-processor and an off-the-shelf DBMS (or in our case, DSMS) of choice.

3.4 Implementing Query Rewrite

Our implementation of query rewrite (based on Miklau and Suciu [21]) consists of two steps: (a) generating *union* expressions for sub-expressions of a query based on mapping rules, and (b) substituting the generated union expressions for the sub-expressions in the query. For every sub-expression of a query that subsumes the right-hand side (i.e., target expression) of at least one mapping rule, a union expression must be generated which involves the original sub-expression and an expression based on the left-hand side of the mapping rule (i.e., the source expression). If a sub-expression subsumes the target expression of several mapping rules, then the union expression naturally involves the source expression of all matching mapping rules. Therefore, we group the mapping rules by their target expressions in order to facilitate the generation of these union expressions.

Table 1 lists the templates we need to generate union expressions for the mapping rules of our benchmark (Section 5). The first three templates are straightforward: for simple path expressions, the union expression is constructed using the source expression of the mapping rule. If the target expression of the

Mapping Rule	Union
a → x	(x | a)
a[b] → x	(x | a[b])
a/b → x	(x | a/b)
a as $a → <x> {...} </x>	(x | (for $a in a return <x> {...} </x>))

Table 1. Templates for Union Expressions
for a query which involves an "x"; e.g., $//x$

mapping involves an element constructor (the last template of Table 1), then an alternative in the union must be generated for each instantiation of the variable bound to the source expression of the mapping rule. This can be implemented in XQuery with a simple FLWR expression.

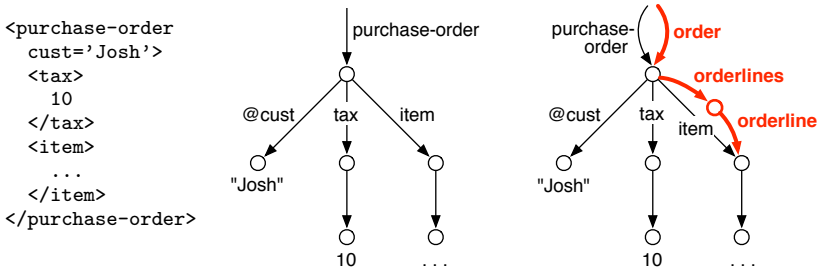
The second step, after the generation of union expressions, is to substitute steps in the path expressions of the query with the generated union expressions. Every XPath step expression is replaced with the corresponding union expression (which we found using a path containment checking algorithm [21]).

Table 1 does not deal with the case that the target expression of a mapping rule is a multi-step path expression; e.g., $a \rightarrow x/y$. This case is special because checking path containment is difficult in the general case (i.e., it is difficult to find the right places to do the substitution), so writing a general equivalence is not possible. We do include such rules in our benchmark (again using the algorithms of [21]).

4 Mapping Data to Queries

This section describes a fundamentally new approach to (continuous) query processing and data integration with mapping rules. This technique maps the data to the queries at runtime, rather than mapping the queries to the data at compile time as in the query rewrite approach. Consequently, we call this method *mapping data to queries* or *MDQ*, for short.

The main idea of MDQ is to annotate the data while it is processed in order to represent the effects of mapping rules in the data (rather than in the query). That is, the internal representation of a message is augmented at runtime to reflect how mapping rules affect parts of the message. Augmenting the internal representation of a message is carried out lazily as part of query processing. As a result, only those parts of a message are augmented that are relevant for at least one query. If the whole message is irrelevant for all queries, then the mapping rules will not be executed and no data will be annotated. In order to scale with the number of mapping rules, MDQ indexes the source expressions of mapping rules so that mapping rules that match a part of a message can be found quickly at runtime.



(a) XML Snippet (b) Tree (c) Annotated Tree

Figure 3. Annotating Data with MDQ

As will be shown in Section 5, the big advantage of the MDQ approach is that it scales well with the number of mapping rules. MDQ also performs better than query rewrite if the mapping rules are complex or if the queries are complex and involve many path expressions. Furthermore, MDQ has another nice property: It is possible to add (and remove) mapping rules without recompiling any queries. In this way, MDQ implements a pay-as-you-go data integration approach [13] and enables high availability at the same time. The query rewrite approach, on the other hand, involves recompilation of queries whenever a new rule is added or a rule is deleted, thereby making the system unavailable for a short period of time.

The main disadvantage of the MDQ approach is that it involves modifications to the core of the DSMS; all key components such as the storage manager, the compiler, and the runtime system need to be adjusted. Fortunately, these adjustments are straightforward to implement. As a proof of concept, we integrated the MDQ approach into an open source XQuery processor (Saxon) and the effort was reasonable (about one person year). The remainder of this section describes the adjustments we made as part of this exercise.

4.1 Storage Manager

In order to implement the MDQ approach, the storage manager, which maintains the internal representation of all data, must be extended to store annotations generated by the evaluation of mapping rules on the data. Figure 3 illustrates how MDQ annotates data with the help of an example. Figure 3a shows a small XML snippet. Figure 3b shows how this XML snippet is represented as a tree. The labels of edges represent the qualified names of attributes or elements (QNames) in the original XML snippet. Leaf nodes contain values (e.g., the string "Josh") of the original XML snippet. Figure 3c shows the annotated tree after applying the mapping rules *purchase-order* → *order* and *item* → *orderlines/orderline*. The first rule, *purchase-order* → *order*, adds an additional edge into the tree in order to represent that the root node also matches a query which asks for an *order*. The second rule, *item* → *orderlines/orderline*,

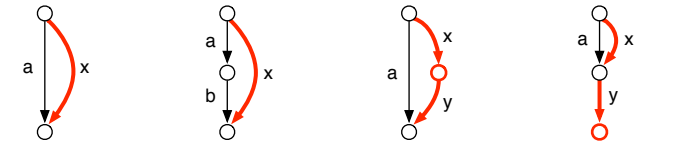
adds an additional node and two edges in order to represent that items can be reached by the path expression *//orderlines/orderline*.

Figure 4 shows that there are four ways a mapping rule can impact the tree that represents a message. The simplest case is shown in Figure 4a for simple aliasing mapping rules which map one QName to another QName. In this case, an additional edge from a parent to a child is added (just as for *purchase-order* → *order* in Figure 3c). If the *source expression* of the mapping rule involves a path expression (i.e., unnesting), then the edge can go across several levels of the tree; this pattern is shown in Figure 4b. Since the source expression of a mapping rule only involves forward axes, an edge always goes from a node to one of its descendants (never to an ancestor) so the annotated tree is always acyclic. Figures 4c and 4d show how new nodes can be created through the application of a mapping rule. Nodes are generated if the target expression of a mapping rule involves a path expression (i.e., nesting as shown in Figure 4c) or a constructor (Figure 4d). The pattern shown in Figure 4c corresponds to the application of the *item* → *orderlines/orderline* mapping rule in the example of Figure 3c.

Even though it is not shown in Figures 3 and 4, it is possible that a node has several annotations. For instance, the root of the annotated tree of Figure 3c could have three incoming edges, if there were an additional rule *order* → *po*: (a) the `purchase-order` edge derived from the original XML snippet; (b) the `order` annotation generated from the *purchase-order* → *order* rule; and (c) a `po` annotation generated from the *order* → *po* rule. In this example, it is important that the `order` annotation is generated before the `po` annotation because, otherwise, the *order* → *po* rule would not become applicable. Nevertheless, the annotations themselves in the annotated tree are not ordered. Annotations are only needed during query processing to detect whether a particular node of the annotated tree matches a query. For this purpose, at least one annotation needs to match; if several annotations match, the order of evaluation does not matter. As a result, the annotated tree does not need to order annotations. The flexibility to add annotations in any order to the annotated tree is important. First, it allows MDQ to be applied *lazily* at runtime, as described in Section 4.3. Second, unordered rules are significantly easier to maintain than ordered.

An annotated tree can be serialized to XML (or JSON, or any other serialization format) in order to produce query results in exactly the same way as classic (un-annotated) trees. As part of serialization, the additional edges and nodes generated to encode the mapping rules are simply ignored. As a result, the query *//order* returns a `purchase-order` element with `item` sub-elements in the example of Figure 3c; the result does not contain any `order`, `orderlines`, or `orderline` elements.

For practical purposes, there are many alternative ways to implement trees in XQuery processors. Saxon, for instance, uses the TinyTree model [17]. Annotated trees, as required for MDQ, are no longer trees in the strict sense; they are acyclic directed graphs. Nevertheless, all the implementations that we are aware of can easily be extended in order to implement such directed acyclic graphs with annotations. In our implementation for Saxon, we used an array representation



(a) $a \rightarrow x$ (b) $a/b \rightarrow x$ (c) $a \rightarrow x/y$ (d) $a \rightarrow <x><y/></x>$

Figure 4. Expansion Cases

as proposed in the Pathfinder project [9, 3].

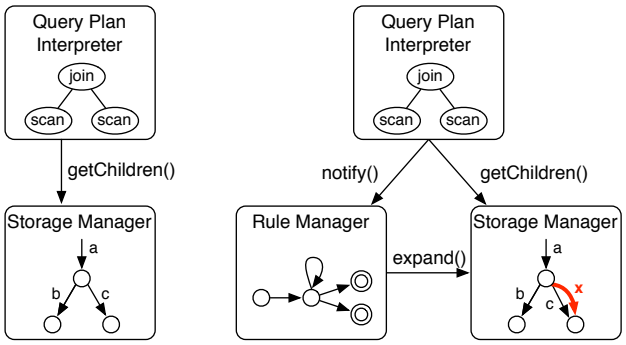
4.2 Query Compiler

Although the change in the storage manager from a tree to an acyclic graph with annotations seems significant, only a few adjustments must be made to the query compiler of an XQuery processor. The two most significant changes involve additional duplicate elimination and sorting a query result in document order. For instance, duplicate elimination becomes necessary for the expression *(/purchase-order | /order)* applied to the annotated tree of Figure 3c. Without duplicate elimination, the root node would be returned twice as a query result because this root node matches both paths. Additional sorting in document order can be necessary if annotations are labelled in the same way as edges of the original (un-annotated) tree.

Obviously, duplicate elimination and sorting in document order are not always needed. For instance, duplicate elimination is not necessary for the query */purchase-order*. As a result, the compiler's optimizer must be extended in order to detect such situations and add these costly operators in a query plan only when they are needed. Another important optimization is to identify at compile time the set of potentially relevant mapping rules. Eliminating mapping rules which have no impact obviously reduces the running time because irrelevant mapping rules are never applied. This optimization works in the same way for the MDQ approach as for the query rewrite approach described in Section 3.4.

4.3 Runtime System

As mentioned at the beginning of this section, MDQ works at runtime. Figure 5 shows how the runtime system of a traditional query processor is extended to integrate the MDQ approach. The runtime system of a traditional query processor is composed of an *interpreter* which executes a query plan by executing each operator of a query (e.g., scans and joins) with the help of the *storage manager*. This architecture is extended in three ways (Figure 5b): First, the storage manager must be extended in order to implement annotated trees, as described in Section 4.1. Second, a new component, the rule manager, is added which interprets the mapping rules, thereby creating new edges and nodes in the



(a) Traditional (b) MDQ

Figure 5. Extended Runtime System

annotated tree. Third, the query interpreter must be extended in order to call the rule manager so that the interpretation of mapping rules in the rule manager and the interpretation of queries is interleaved. The following subsections explain in more detail how the interpreter and rule manager interact during query processing using MDQ.

Lazy Evaluation of Mapping Rules A naïve way to implement MDQ is to apply all mapping rules to a message eagerly *before* processing queries on that document for the first time. That is, when a new message is processed, then that message is parsed and immediately all mapping rules are applied to it in order to create the completely annotated tree for that message. After that, all queries can be interpreted using that annotated tree in the same way as in a traditional query processor. In fact, the query interpreter and its operators need not be extended at all in this approach.

While this eager approach is extremely simple to implement, it exhibits poor performance because it applies all mapping rules on all nodes of the message. Even if the set of relevant mapping rules is pruned at compile time (Section 4.2), many mapping rules are applied to many nodes unnecessarily. For instance, if the query asks for all *orders* of Customer "Sue", in the example of Figure 3, then applying the $item \rightarrow orderlines/orderline$ rule is unnecessary because the document does not match the query anyway. Such a situation cannot be detected at compile time. The eager approach is particularly bad in the presence of *recursive mapping rules*. An example for a recursive mapping rule is $a \rightarrow a/a$. If such a recursive mapping rule is applicable, the process of applying mapping rules eagerly does not terminate because the recursive mapping rule is applicable again and again to each new node that is created in the annotated tree.

The alternative to the eager approach is to interweave the processing of mapping rules with the processing of queries and add edges to the annotated tree *lazily*. This requires extensions to the query interpreter of a DSMS as shown in Figure 5b. Specifically, the *scan* operator which navigates through an incoming

message must be extended (all the other operators are unchanged). In a DSMS which does not index incoming messages, scanning a message starts at the root and involves iterating through parent-child relationships of each visited node. Whenever the *scan* operator visits a child for the first time, it notifies the rule manager and informs it of the newly observed node. No other extensions to the scan operator are needed and there exists no memory overhead in the notification step. The rule manager, in turn, checks which mapping rules are applicable to the newly observed node and expands the annotated tree (adds edges and nodes) if one or more mapping rules fire. This firing of mapping rules is depicted by the *expand* arrow in Figure 5b. Newly created edges can be visited in the next iteration of the *scan* operator which triggers additional notifications of the rule manager. Transitivity in the application of rules is implemented in this way. We describe how the rule manager determines which rules are applicable to a newly visited node in the next subsection.

As will be shown in Section 5, the lazy approach performs much better than the eager approach. Continuing the example from above, no mapping rules are applied for Customer "Josh", if no query asks for this order. Furthermore, the lazy approach deals gracefully with recursive mapping rules. If a continuous query asks for $/a/a/a$, the incoming message is <a/>, and there is a mapping rule $a \rightarrow a/a$, then the lazy approach applies this mapping rule exactly twice in order to discover that the message matches the query, but not more than that. If the continuous query involves recursion itself (e.g., $//a$), then even the lazy approach does not terminate because the query has an infinite result [15]. Such situations cannot be avoided even in the absence of mapping rules because XQuery is a Turing-complete programming language.

Rule Manager In order to detect when one or more mapping rules fire, the rule manager implements a non-deterministic finite automaton (NFA) as proposed in the YFilter project [8]. A single NFA is constructed in order to implement the source expressions of all mapping rules. Whenever a source expression of a mapping rule is matched, the NFA reaches an accepting state which indicates the firing of a mapping rule. The accepting states of the NFA are annotated with the target expressions of the corresponding mapping rules and a pointer to the state that represents the root node of the source expression, denoted as *root* of the rule. This root node is needed in order to detect the starting point of the new edge added to the annotated tree that represents the message in the storage manager. The target expression is evaluated when a mapping rule fires in order to compute the end of the new edge, the labels of the new edge, and possibly additional intermediary nodes and edges as shown in Figures 4c and d.

Figure 6 shows the NFA for a single and simple mapping rule, *purchase-order* $\rightarrow$ *order*. This rule fires whenever a *purchase-order* node is visited for the first time. Consequently, the NFA reaches its accepting state (S_2) whenever it is notified of a *purchase-order* node. Whenever the NFA reaches State S_2, the rule manager adds an *order* edge from the *root* node to the node that triggered the state transition.

Figure 7 shows how the NFA of Figure 6 would be extended if three mapping

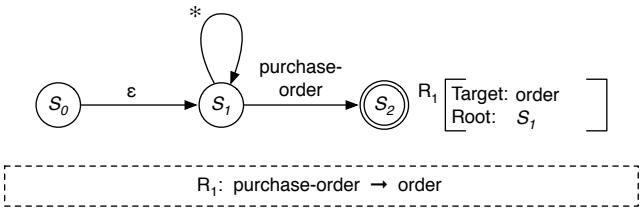


Figure 6. NFA for *purchase-order* → *order*

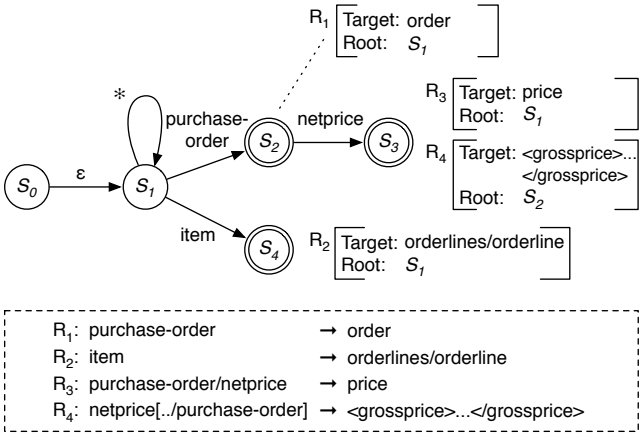


Figure 7. NFA for Multiple Mapping Rules

rules were added. This figure shows how the NFA implements the sharing of prefixes of paths of the source expressions of several mapping rules. Furthermore, this automaton shows the need for annotating the states that correspond to *root* nodes of source expressions if the source expression involves complex path expressions. As in the automaton of Figure 6, the NFA of Figure 7 reaches an accepting state (i.e., S_2) whenever a `purchase-order` element is found; in this case Rule R_1 fires and operates in the same way as in the NFA of Figure 6. In addition, the NFA of Figure 7 reaches an accepting state (i.e., S_4) whenever an `item` element is found; in this case Rule R_2 fires. Finally, the NFA of Figure 7 reaches an accepting state if a sequence of `purchase-order` and `netprice` elements are found (i.e., S_3); in this case, Rules R_3 and R_4 both fire. If two or more rules fire at a state transition, these rules can be executed in any order because annotations in the annotated tree are unordered as explained in Section 4.1.

The construction of NFAs for the purpose of indexing path expressions has been described in full detail for YFilter [8]. In fact, YFilter can be directly applied to implement the MDQ rule manager. YFilter also supports predicates (not shown in the examples of this section, for brevity). As a result, YFilter supports all possible source expressions in the mapping rules of Section 3.2,

and our implementation of the rule manager makes use of the original YFilter algorithms and data structures [8]. Only a small adjustment is needed in order to implement a stack to compute the *Root* when a mapping rule fires. We omit a description of the details of this adjustment in this paper for ease of presentation.

5 Experiments and Results

We implemented both the query rewrite and MDQ approaches using the Saxon XQuery engine. This section studies the performance of these two approaches using the XMark benchmark [29], the TPoX benchmark [24], and a series of micro-benchmarks. These benchmarks cover a wide spectrum of different use cases.

5.1 Software and Hardware Used

All experiments were carried out using a machine with an AMD 2.4GHz CPU and 6GB RAM running RedHat Enterprise 4. Both approaches (query rewrite and MDQ) were implemented in Java 1.6 on top of Saxon 9 [16], a main memory XQuery processor.

Query rewrite was implemented using a benchmark-specific pre-processor to generate the rewritten queries. That is, the pre-processor was designed for these specific benchmarks and we made sure in an additional manual step that the best possible query rewritings were used in all cases. In particular, we made sure that only relevant mapping rules were considered during the rewriting of each query. That is, relatively few rules were applied to each query even in experiments which involved thousands of schemas. As a consequence, the results for the query rewrite approach are better than would be expected with a general-purpose implementation of query rewrite. The rewritten queries were then executed using Saxon as is; i.e., Saxon was not changed.

In order to implement MDQ, we extended the Saxon compiler and runtime system as described in Section 4. Without any mapping rules the original Saxon and the MDQ-enabled Saxon showed roughly the same performance, so our implementation provides an apples to apples comparison of query rewrite vs. MDQ in Saxon. We expect that similar extensions to other XQuery processors would produce similar results, as we did not exploit any features specific to Saxon in our implementation.

As a baseline, an *ideal* world was studied in all experiments. In such an *ideal* world, all data and queries conform to the same schema and no mapping rules are needed. These experiments could be carried out with "off the shelf" Saxon. Obviously, the performance in an ideal world is better than the performance that can be achieved using either query rewriting or MDQ to resolve heterogeneity. Comparing the results for *ideal* with those for query rewrite and MDQ reveals the *cost of heterogeneity*.

All experiments reported in this section study runtime performance only; i.e., message throughput in a data streaming scenario and query response times

for a traditional database scenario. In all experiments, the queries were pre-compiled. Furthermore, the NFA for indexing all mapping rules was already created. Messages were sent in a binary format (i.e., pre-parsed), but did not contain any annotations for the MDQ approach. That is, the effort to create these annotations was measured as part of the throughput and response times. If several queries were executed on a single message, the annotations were shared between all queries so that the same annotation did not have to be generated twice.

Saxon is a streaming XQuery processor so it is a good match for these performance experiments. Saxon, however, does not implement multi-query optimization and other tricks to optimize and scale with the number of continuous queries. Unfortunately, no XQuery product to date does this. As a result, all experiments were carried out with a relatively small number of continuous queries (at most twenty for the XMark benchmark). An interesting topic for future work is to study the performance of the alternative approaches with an XQuery processor that is tuned for scalability in the number of queries, once one is available. However, we do not expect that such experiments will reveal any significant differences in the trade-offs between query rewrite and MDQ.

5.2 Benchmark Environment

This section reports on results for three different benchmarks: XMark [29], TPoX [24], and a set of micro benchmarks which we designed specifically to study effects that could not be observed with the XMark and TPoX benchmarks.

The XMark benchmark models an *online auction system* and messages involve bids on items [29]. The XMark queries vary from simple path expressions that select certain properties of a message to complex queries that involve joins and aggregation. We implemented all twenty queries of the XMark benchmark and studied them individually (i.e., only one query is applied to each message) and as a whole (i.e., all twenty XMark queries are applied to all messages). If not stated otherwise, the XMark data generator was set to generate messages of size of about 1.1 MB. In order to study the effects of different message sizes, this parameter was varied from a few KBs to 100 MB in a separate experiment.

The TPoX benchmark models a finance institution and involves a database of financial orders [24]. The TPoX queries vary from simple selection of orders (based on their order number) to more complex queries (e.g., aggregates on a subset of orders). Due to the nature of the queries and the TPoX data generator, the TPoX benchmark cannot be applied as a measure of the performance of data stream management systems. As a result, we used this benchmark in our experiments in order to study the trade-offs of query rewrite and MDQ in a more traditional database scenario. That is, we loaded the TPoX database into the Saxon (in-memory) storage manager, ran the TPoX queries using Saxon, and measured the response time (rather than message throughput) of each TPoX query for both query rewrite and MDQ. Since Saxon has no support for indexes, we used the smallest possible TPoX database which contains 200 MB of XML data.

The XMark and TPoX benchmarks were used to study the performance trade-offs of a variety of queries, as well as database and message configurations (i.e., size, data skew, and formats). Due to the complexity of the queries and because there is no general-purpose implementation of the query-rewrite approach available yet, however, it was not possible to study different types of mapping rules using the XMark and TPoX benchmarks. In order to simulate heterogeneity, we were limited to using *aliasing* mapping rules (i.e., rules of the form $a \rightarrow x$, without path expressions and constructors) for these two benchmarks because query rewrite is straight-forward to implement for such mapping rules. Specifically, we took the original XMark and TPoX schemas as a starting point and created new schemas by renaming all the element and attribute names. For the XMark benchmark 46 mapping rules were generated per schema; for the TPoX benchmark each schema involved adding 10 new mapping rules. Depending on the query, however, only a fraction (between 5-10 for XMark, 2-10 for TPoX) of these mapping rules were applicable for each query.

In order to study the effects of complex mapping rules, we also experimented with micro benchmarks which were tailored for this specific purpose. For the micro benchmark results reported in this paper, //x/y was used as the only query and different sets of mapping rules were used, varying from aliasing rules (i.e., $a \rightarrow x$), nesting / unnesting rules (i.e., $a/b \rightarrow x/y$), to element construction (i.e., $a \rightarrow <x/>$).

In all experiments reported in this section, we varied the number of schemas in order to study how well query rewrite and MDQ scale in this dimension. (Obviously, query performance in an ideal world is independent of this parameter, since there is only one schema.) From discussions with organizations that use data integration technology, it seems that tens to hundreds of schemas are common today. With emerging standards such as HL7 and XBRL, we expect that data integration of thousands of schemas will be required in the near future. Establishing interoperability between all hospitals of the USA would easily involve such a large number of schemas. As a result, we varied the number of schemas from tens to thousands, to cover the broad range of present and near-future scenarios.

XMark Queries The first set of experiments studied the message throughput of query rewrite and MDQ for the queries of the XMark benchmark and a varying number of schemas. Figures 8 to 10 show the throughputs for query rewrite, MDQ, and ideal for three sample XMark queries: Q1 is a simple path expression; Q4 involves several path expressions with predicates; Q11 involves heavy computation (joins and sorting), but does not involve sophisticated navigation through the message with path expressions. For all three queries, it can be observed that the throughput of MDQ is independent of the number of schemas (i.e., the number of mapping rules), whereas the message throughput of query rewrite drops sharply with an increasing number of schemas. As mentioned in Section 4 the reason is that the size of the queries grows with the number of schemas and that all (potentially relevant) mapping rules are encoded in every query for every message even though only a small subset of mapping rules are

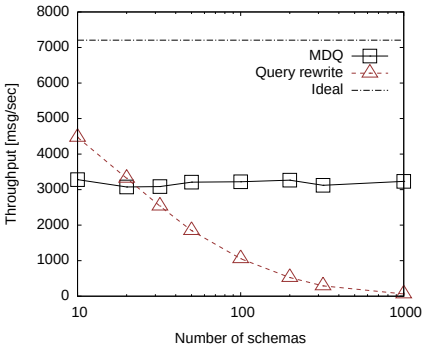


Figure 8. msg/sec: XMark Q1

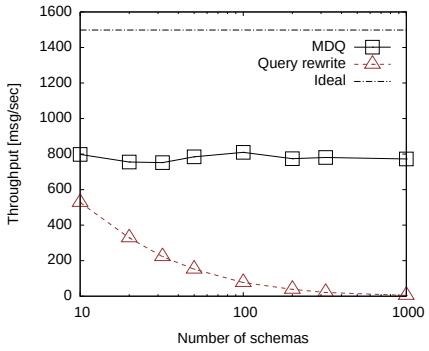


Figure 9. msg/sec: XMark Q4

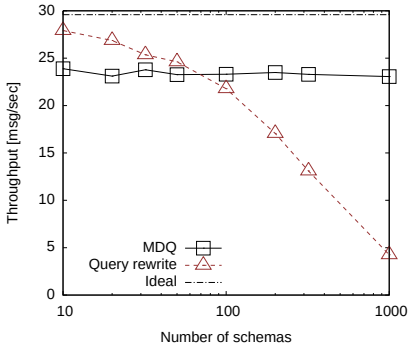


Figure 10. msg/sec: XMark Q11

applicable to each particular message.

Depending on the kind of query, query rewrite sometimes performs better than MDQ for data integration scenarios with relatively few schemas (e.g., for Query 1 and Query 11 and less than 100 schemas). As a general trend, the more navigation (with path expressions) is involved in a query the better MDQ performs as compared to query rewrite even if only a few schemas need to be integrated. In fact, Queries 1 and 11 represent two of only four queries of the XMark benchmark for which there is a cross-over point between MDQ and query rewrite; these are the only queries in which MDQ does not outperform query rewrite regardless of the number of schemas (Table 2, discussed below).

Comparing MDQ to an ideal scenario (without any heterogeneity and mapping rules), it can be seen that MDQ typically comes within 50 percent of the throughput of *ideal*; in many cases it comes even closer (Table 2). These results are encouraging because they show that good system performance can be achieved even on heterogeneous and diverse data sources.

Table 2 shows the results of the complete XMark benchmark, varying the

	10		100		1000		
	MDQ	QR	MDQ	QR	MDQ	QR	Ideal
Q1	3281	4472	3221	1055	3233	64	7206
Q2	1513	1526	1585	379	1515	23	2565
Q3	744	496	735	77	740	3.0	1329
Q4	798	527	810	77	772	4.4	1498
Q5	3234	2080	2986	336	3171	16	5413
Q6	510	137	516	13	509	0.62	2278
Q7	286	60	285	6.1	282	0.28	1001
Q8	26	12	26	1.8	26	0.10	32
Q9	21	10	21	1.8	21	0.11	26
Q10	44	33	41	7.2	45	0.37	58
Q11	24	28	23	22	23	4.3	30
Q12	54	74	53	44	57	5.3	88
Q13	725	837	750	427	737	39	778
Q14	206	112	199	16	213	0.73	334
Q15	2648	1597	2627	214	2610	12	5846
Q16	1937	1347	1926	207	1837	8.7	3591
Q17	857	787	865	179	878	10	1352
Q18	1250	1002	1235	198	1215	11	2000
Q19	259	166	257	27	256	1.2	414
Q20	726	389	715	63	731	3.2	1066
All	**5.2**	**3.2**	**5.2**	**0.63**	**5.1**	**0.03**	**6.5**

Table 2. msg/sec: XMark Bench., Vary #Schemas

number of schemas from 10 to 1000. Overall, the results of Table 2 confirm the observations made for Figures 8 to 10. Table 2 shows the message throughputs of running every query individually, that is, assuming that it was the only query executing against the stream of messages. Furthermore, Table 2 shows the message throughput achieved when executing all twenty XMark queries on every message (denoted as *All* in Table 2). Obviously, the message throughput is much smaller if all twenty queries are executed than if only one query is executed. Comparing MDQ with ideal in the *All* case, it can be seen that MDQ is quite close to ideal; its throughput is only about 20 percent below the ideal throughput. One reason is that the effort to annotate the messages according to the mapping rules is amortized across queries, when multiple queries are executed on each message. As a result, MDQ becomes more attractive the more queries are executed on a single message. (In Table 2, query rewrite is denoted as *QR*.)

Vary Message Size Figure 11 shows the throughputs of query rewrite, MDQ, and ideal for XMark Query 1, 100 schemas, and a varying message size. That is, the XMark data generator was configured to produce messages of different size in this experiment. As can be seen, the message size has no impact on

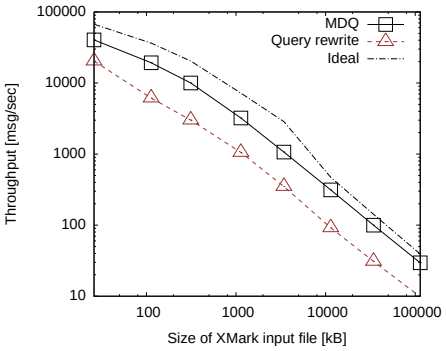


Figure 11. XMark Q1, Vary Msg. Size, 100 Schemas

	10	100	1000
Eager, No Index	0.77	0.078	0.0078
Eager, With Index	461	470	457
Lazy, With Index	3281	3221	3233

Table 3. msg/sec: XMark Q1, MDQ Variants

the trade-offs between query rewrite and MDQ. In this particular case, MDQ outperforms query rewrite independent of the message size. These results were confirmed by experiments made with the other XMark queries, TPoX queries, and the micro benchmarks whose results are presented below.

5.3 Different MDQ strategies

We also used the XMark benchmark in order to study the importance of indexing mapping rules and of a lazy approach to applying mapping rules as described in Section 4.3. Table 3 shows the throughput of MDQ for XMark Query 1, a varying number of schemas, and three different MDQ configurations. First, MDQ with an eager approach to applying mapping rules and without a YFilter-index for the mapping rules shows extremely poor performance. Likewise, the second approach, an MDQ that eagerly applies mapping rules and uses a YFilter index, shows poor performance. Neither of these approaches are competitive compared to the query rewrite approach. Only the proposed MDQ variant (i.e., lazy and with indexing of mapping rules) is competitive.

5.4 Micro Benchmarks

As mentioned in Section 5.2, we carried out a large number of micro benchmark experiments which involved handcrafted queries, mapping rules, and messages

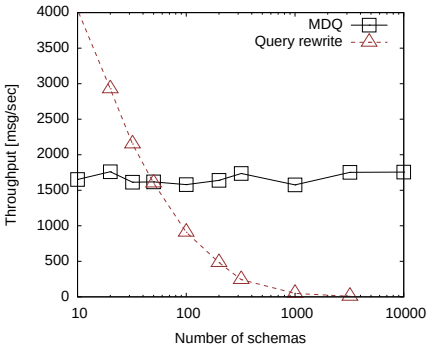


Figure 12. msg/sec: Micro Bench. Aliasing: $a \rightarrow x$

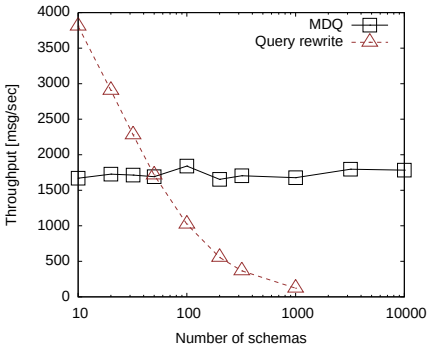


Figure 13. msg/sec: Micro Bench. Unnesting: $a/b \rightarrow x/y$

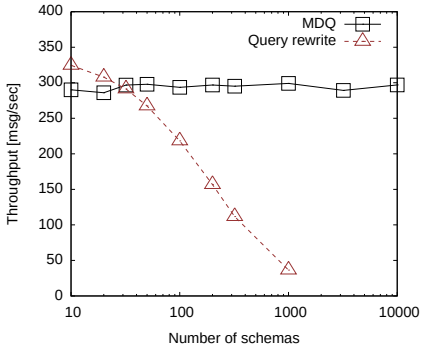


Figure 14. msg/sec: Micro Bench. El. Constr.: $a \rightarrow <x/>$

in order to study the trade-offs of query rewrite versus MDQ for different scenarios. Overall, these experiments confirmed the results and observations made with the XMark benchmark. For brevity, this section only reports on results that were achieved by running simple path expression queries (i.e., //x/y) on simple messages of size 16kB, while varying the complexity of the mapping rules. Figure 12 shows the message throughputs of query rewrite and MDQ with a varying number of schemas, if the mapping rules involve only *aliases* (as in the XMark benchmark). Figure 12 corresponds to Figure 8 and has roughly the same shape. Figure 13 shows the results if the mapping rules involve nesting and unnesting (i.e., path expressions as part of source and target expressions). Figure 14 shows the results if the mapping rules involve element construction as part of the target expression. As a general trend, it can be observed that MDQ compares more favorably to query rewrite as the complexity of the mapping rules increases. For unnesting rules and element construction, Saxon with the

	MDQ (cold)	MDQ (hot)	QR	Ideal
Q1	125	78	1141	54
Q2	47	40	459	37
Q3	20	16	39	13
Q4	125	92	796	84
Q5	19	16	38	14
Q6	67	51	457	43
Q7	166	117	1192	73

Table 4. Resp. Time [msec]: TPoX, 100 Schemas

query rewrite approach crashed for more than 1000 schemas because of a stack overflow.

5.5 TPoX Benchmark

The experiments with the XMark benchmark and all the micro benchmarks studied the performance of query rewrite and MDQ in a *data streaming* environment. This section gives initial performance results for static data; that is, a more traditional database scenario. To this end, we use the TPoX benchmark [24] which was designed to evaluate the performance of an XML database. As mentioned in Section 5.2, the TPoX benchmark involves a database of financial orders and queries of varying complexity.

Table 4 shows the running times of running the seven TPoX benchmark queries with 100 schemas using the MDQ and query rewrite (QR) approaches and in an *ideal* world (no heterogeneity). For MDQ, two scenarios were studied:

MDQ (cold): The TPoX database is stored in Saxon's storage manager, but no mapping rules have been applied yet. (That is, no additional edges or nodes have been added to the tree representation of the documents.)

MDQ (hot): The TPoX database is already fully annotated with additional edges and nodes after applying all relevant mapping rules. This was achieved by running the whole benchark twice and measuring the second run only.

The results of this experiment confirm all the observations of the previous experiments: MDQ clearly outperforms query rewrite; in some cases by as much as an order of magnitude. Furthermore, MDQ is close to an *ideal* world without any heterogeneity. Comparing the MDQ(cold) and MDQ(hot) running times, it can be seen that the overhead of applying the mapping rules and adding edges and nodes to the MDQ Data Model is roughly 20 percent in this experiment. (The overhead varies for the individual queries.)

6 Conclusion

The key innovation presented in this paper is MDQ, a new query processing technique for virtual data integration. As compared to query rewrite, the state-of-

the-art in query processing for virtual data integration, MDQ scales significantly better in the number of mapping rules and schemas that need to be integrated. To demonstrate the benefits of MDQ, we extended an off-the-shelf data stream management system (DSMS) with MDQ, and compared the performance of the extended DSMS to the same DSMS fitted with the traditional query-rewrite approach. Our performance experiments show that MDQ significantly outperforms query rewrite for complex queries and rules. Furthermore, MDQ scales better with an increasing number of mapping rules. Most surprisingly, it could be shown that MDQ performs well compared to a DSMS in a *perfect world* (i.e., no heterogeneity in the data feeds or query schemas). In other words, this higher level of data independence provided by MDQ was accomplished with reasonable performance overhead.

There are several avenues for future work. One is to study MDQ for traditional heterogeneous query processing in a mediator [31] or federated database system [11]. Our choice of rules in this first work was based on the practical needs of the applications we studied; we would like to compare them to more traditional mapping rules and rewriting engines [33, 25]. Another interesting direction for future work is to expand our method to other kinds of rules; for instance, a negative rule stating that two entities are not the same or rules that model other relationships used in RDF Schema and OWL. This could potentially allow ontologies as defined by the Semantic Web community to be integrated into large-scale XML query processing using XQuery, bringing together benefits of both the Semantic Web and Database worlds.

Acknowledgments We would like to thank Jonas Rutishauser for his help in implementing the prototypes used in the experiments.

References

[1] Daniel Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[2] B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

[3] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery - the relational way. In *VLDB*, pages 1322–1325, 2005.

[4] Vinayak R. Borkar, Michael J. Carey, Dmitry Lychagin, and Till Westmann. The BEA AquaLogic data services platform (demo). In *SIGMOD Conference*, pages 742–744, 2006.

[5] Sirish Chandrasekaran et al. TelegraphCQ: continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[7] James Clark and Steve DeRose. XPath 1.0. `http://www.w3.org/TR/xpath`.

[8] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 28(4):467–516, 2003.

[9] Torsten Grust. Accelerating XPath location steps. In *SIGMOD Conference*, pages 109–120, 2002.

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

[11] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.

[12] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Transactions On Knowledge and Data Engineering*, 16(7):787–798, 2004.

[13] Alon Y. Halevy, Michael J. Franklin, and David Maier. Principles of dataspace systems. In *PODS*, pages 1–9, 2006.

[14] Health Level Seven Inc. Healthlevel 7. `http://hl7.org`.

[15] Martin Hentschel, Donald Kossmann, Tim Kraska, Jonas Rutishauser, and Daniela Florescu. Mapping data to queries: Semantics of the is-a rule. `http://www.systems.ethz.ch/research/publications`. Technical Report, ETH Zurich, November, 2007.

[16] Michael Kay. Saxon XSLT and XQuery processor. `http://saxon.sourceforge.net`.

[17] Micheal Kay. Ten reasons why Saxon XQuery is fast. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 31(4):65–74, 2008.

[18] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation, 2004.

[19] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, May 1995.

[20] Deborah L. McGuinness, Frank van Harmelen, et al. OWL web ontology language overview. W3C Recommendation, 2004.

[21] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

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

[23] Rajeev Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[24] Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML transaction processing benchmark. In *SIGMOD Conference*, pages 937–948, 2007.

[25] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD Conference*, pages 443–454, 2006.

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

[27] R. Pottinger and A. Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2–3):182–198, 2001.

[28] Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. iTrails: Pay-as-you-go information integration in dataspaces. In *VLDB*, pages 663–674, 2007.

[29] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[30] Douglas B. Terry, David Goldberg, David Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *SIGMOD Conference*, pages 321–330, 1992.

[31] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[32] xbrl.org. XBRL extensible business reporting language. `http://www.xbrl.org`.

[33] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD Conference*, pages 371–382, 2004.