Indexed Semantic Mapping Rules

Haoning Zhang
(zhanghao@student.ethz.ch)

Master Thesis in Computer Science

March 2009

Supervised by

Prof. Donald Kossmann

Martin Hentschel

Tim Kraska
## Contents

Abstract ................................................................................................................................. 4  
1 Introduction .......................................................................................................................... 5  
  1.1 Motivation .................................................................................................................... 5  
  1.2 Project Goals .............................................................................................................. 5  
  1.3 Outline ....................................................................................................................... 7  
2 Fundamentals ..................................................................................................................... 8  
  2.1 XML & XQuery .......................................................................................................... 8  
    2.1.1 XML Basics ....................................................................................................... 8  
    2.1.2 Querying XML ................................................................................................. 8  
  2.2 Data Integration .......................................................................................................... 9  
    2.2.1 Mapping Queries to Data ................................................................................. 9  
    2.2.2 Mapping Data to Queries ............................................................................... 10  
  2.3 Tree-based Index ....................................................................................................... 10  
3 B+ Tree based Rule-Aware Index ..................................................................................... 12  
  3.1 Index Structure .......................................................................................................... 12  
    3.1.1 Dependency Graph ......................................................................................... 13  
    3.1.2 Cycle Detection ............................................................................................... 14  
    3.1.3 B+ Tree with Insert Reference Counting ....................................................... 17  
    3.1.4 Rule Path Table ............................................................................................... 18  
  3.2 Semantics ................................................................................................................... 19  
    3.2.1 Add “is-a” Rules .............................................................................................. 19  
    3.2.2 Scan new XML documents ............................................................................. 24  
    3.2.3 Remove XML documents ............................................................................... 28  
    3.2.4 Update XML documents ................................................................................ 31  
    3.2.5 Remove “is-a” Rules ...................................................................................... 31  
4 Integration into MDQ ........................................................................................................ 37
4.1 Architecture Overview ................................................................. 37
4.2 Index Management ................................................................. 38
4.3 Storage Management ................................................................. 39
4.4 MDQ Engine Interface ............................................................. 40
  4.4.1 Query XML Documents .......................................................... 40
  4.4.2 Extract Index Information ...................................................... 41
4.5 Query Interface ............................................................................ 42
5 Experimental Evaluation ............................................................. 44
  5.1 TPoX Benchmark ........................................................................ 44
  5.2 Query without Mapping Rules ................................................ 44
  5.3 Experiments with Mapping Rules ............................................. 48
    5.3.1 Building Index ....................................................................... 48
    5.3.2 Querying .............................................................................. 50
    5.3.3 Removing Data ..................................................................... 54
    5.3.4 Removing Rules .................................................................. 55
6 Conclusion & Future Work .............................................................. 57
Bibliography ....................................................................................... 58
Abstract

The Extensible Markup Language (XML), typically represented as text with explicit structure, is widely used as the standard in Internet, Data Exchange, and other technology domains. To manage different sources of information, several XML data integration techniques have emerged in last several years. The two most accepted solutions are information hub, which maps the data to queries, and query rewrite technique. One recent research project at ETH Zurich about Mapping Data to Queries (MDQ) provides a different approach, which introduces mapping rules to reflect data relationships and transform the data on-the-fly.

This master thesis presents an extended index structure that enables highly efficient and scalable XML query in existing MDQ engine. The key contributions of the work include: (1) Using index techniques from relational database to apply semantic mapping rules to large amounts of XML data; (2) Algorithms based on Dependency Graph to detect index impact when data or mapping rules are changed; (3) An implementation of that index as prototype that cooperates tightly with the existing MDQ engine.

In the experiments, the target solution shows significant query performance to process large amounts of data, when the mapping rules are fairly stable. The insertion or removing XML data or mapping rules apply the synchronization to the index transparently and with affordable cost.
1 Introduction

1.1 Motivation

XML as semi-structured data dominates data management and data exchange in these years. When the XML based data dramatically increases, the data management and integration problems are becoming more critical. XML data are fundamentally different from those in the traditional database management system in terms of their structure. In most institutions, enterprises, and within the whole Internet, XML data are kept in separated systems, where the data relationships are distributed maintained. As consequence, data integration is widely accepted and used technology to manage those data. To describe the relationship between data, the Mapping Data to Queries (MDQ) [4] solution provides a schema mapping technique to describe subset relationships.

By defining the mapping rules as the input of the MDQ engine, the solution evaluates related results by its data relationships in a on-the-fly model, so theoretically it solved the data integration problem. However, in the practical situations, it will lead to performance penalties when the data volume reaches specific levels. The original query operation needs a full scan of all documents from the data source. When the size of XML data grows larger or data are stored in the distributed environment, the query performance are hardly bearable.

The root cause is the original MDQ engine is mainly designed for streaming processing, not for the database, where index structure are usually implemented. Without index support for a non-join query has to completely go through the data sources with \( n \) entities and costs \( \Theta(n) \). In the current master thesis, the index mechanism is leveraged to support mapping rules in XQuery model.

1.2 Project Goals

How can we improve the performance when querying the XML data store with mapping rules? In the traditional database, index techniques are used to increase the performance. Ideally, we would like to enjoy the benefits of the index mechanism to fasten the data mapping related
Existing index structures and algorithms obviously are not aware of mapping rules. Tree-based indexes and its derivatives like the B+ tree [2] are widely employed in most database systems, the keys indexed in the tree nodes come from the same table or entity, which does not fit crossing different data schemas because of data mapping relationships. The example in Figure 1-1 shows one common business case in international companies. Assuming the company has two branches, one in United States with employee information stored (e.g., monthly salary) in US Dollar. Similarly, the other branch in Switzerland maintains the personal records with salary stored in Swiss Franc. Moreover, both branches designed their own data formats. Now the request comes. The group CEO needs to find out the people in global scope, whose salary is between 5000 and 6000 Swiss Francs. Considering the exchange rate is 1.2 for CHF/USD, to fulfill the request, the request needs to be split for both indexes. In particular, for the first index, the query range is adjusted to (5000/1.2, 6000/1.2). Finally, both result sets need to be merged as the final result. Although this on-the-fly mapping algorithm works, the maintenance cost is not bearable when the number of entity classes and entity instances becomes larger. Moreover, the ever-increasing complicated mapping logic will kill the performance because the query engine has to re-execute the mapping logics in each request. In Section 5.2 the experiment results show that the cost of naïve queries is linear to the size of XML data and number of schema mapping.

So the 1st goal of the master thesis is to propose one extended index structure to keep the intermediate mapping results. This idea is already implemented in traditional database as
Materialized Views, to avoid computation effort when specific queries are frequently issued. When the mapping source entities are stored in target index with transformation, querying performance will be significantly increased because no mapping logic needs to be executed.

However, as Materialized View in rational databases, storing pre-computed result leads to data consistency problems. In the former example, the index system needs to pay additional attention when (1) New employee entity comes from American branch, (2) exchange rate changed and it is not 1.2 anymore. In other words, the mapping rule is changed. So, the 2\textsuperscript{nd} goal of the research is that index structure needs to be aware of rule mapping relationship and its consequence. Moreover, the index needs to be transparent to users when the data entity or mapping rules are updated.

In addition to the theoretical part, one prototype system is implemented to integrate with the current MDQ engine. With the system users can build the mapping-aware index on XML data nodes. In query processing sessions the corresponding index provides the MDQ engine tailored candidates to delimit the query range.

1.3 Outline
This thesis is organized as follows. Chapter 2 summarizes the background about XML, Mapping Data to Query (MDQ), and Tree-based indexes, which are the fundamentals of this work. Then the proposed index structure and its algorithms are illustrated in Chapter 3. Chapter 4 describes the implementation details of the prototype and how the new index integrates with existing MDQ query engine. My experimental results will be presented in Chapter 5. Chapter 6 concludes the work of the thesis with a brief discussion of ongoing work aimed at developing a more optimized index structure.
2 Fundamentals

2.1 XML & XQuery

2.1.1 XML Basics

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages [14]. It is widely used to describe data and its structure in many application domains, where users define their own mark-up elements. The self-describing property aids information systems to understand each other and share their data. The text-based format makes the XML easily pass through layers of the communication model or even cross different platforms.

Owning those advantages, XML becomes one standard of data storage format. Several XML database systems [10] [11] have emerged in recent years, some of them are dedicatedly using the XML data model, and others are extending relational database such as Oracle and DB2. In terms of the physical storage, one naive approach is to store the XML text in one table in the relational database, where the traditional advantages as security, transaction control, and index mechanism could be re-used. Another approach is to map the hierarchical XML data into multiple tables [12].

2.1.2 Querying XML

When the amount of XML documents dramatically increases, several problems emerge. The first problem is the query technique, in other words: how to find the right information efficiently from huge amounts of data. In the world of structured data, information is stored as records in relational databases, which also provides a standard query language and interface (SQL) to users or its applications. The similar query mechanism must be provided before XML information could be used in large-scale information systems.

The philosophy of XML is that the correlation of data and schema is not necessary, the data and schema relationship could be M:N.
XQuery [1] is a query language that is designed to solve the above problems; it provides the definition to access information in XML data. In the view of XQuery, the collection of XML documents will be accessed as tables in relational database. The standard is already implemented in several open source projects like Saxon [8] and Zebra [9].

In addition to retrieve the information, XQuery also supports to construct new XML elements or documents in its query result. This property make XQuery is very successful in content management, media, and web application domains.

2.2 Data Integration

When the size of unstructured data is greatly increased, the data integration problem becomes one of the most significant issues to solve. Data integration is the process of combining data residing at different sources and providing the user with a unified view of these data. For example, one typical automobile manufactory may require querying its thousands of vendors' inventory system to collect supply chain information. Each vendor uses its own data schema to describe inventory details.

To provider a unified interface for users or upper level applications, scalable schema mapping technique is widely used to solve this problem. Because the query and data are not consistent with each other, two approaches could be involved in this integration process: either (1) Mapping queries to data, or (2) Mapping data to queries.

2.2.1 Mapping Queries to Data

The first approach is to query those information sources respectively, and then merge the result for host application. This mechanism provides an uniform query interface over a mediated schema. This query is then transformed into specialized queries to the original databases.

Mapping queries to data is often used due to the simplicity. The central application aligns the queries against data sources, and those data sources are not necessary to change, which is critical in lots of business scenarios. This implementation is also called as Local As View (LAV) [13], while the local data sources are considered as database views. To map queries to data, several query rewrite techniques are proposed.

Query rewrite algorithm captures the original query statement and splits it into a set of source queries to data sources. After successfully receiving all the result sets from data
sources, the algorithm merges (union operation) the result set and sends it back to user.

The idea of rewriting the query comes from the relational database domain (i.e. views). The drawback is that users have to rewrite the query (views in SQL) whenever a new data source is to be integrated or an existing source changes its schema. Moreover, the performance for querying against large number of diverse schemas is becoming more inefficient. The rewritten queries can become very large. In addition, the effort to manage the data dictionary is not bearable. So, this approach is not scalable, it is only suitable for small number of schemas.

2.2.2 Mapping Data to Queries

Instead of rewriting the query to meet different schema, the Mapping Data to Queries (MDQ) [3] [4] technique introduces an alternative approach through rule-driven mapping mechanism.

The MDQ models the XML data as a Directed Acyclic Graph (DAG) to represent the data mapping relationship. Each edge in the DAG represents a parent-child relationship. The “is-a” rule adds more edges to the graph, and those rules support transitive property.

By this approach, several achievements are valuable in real scenarios: (1) the local data schemas are unchanged; (2) the effort to maintain the rules is obviously smaller than managing queries in query rewriting model; (3) the user applications are not aware of the schema diversity; and (4) the transformation is fired on-the-fly, which reduces the impact when updating the rules.

2.3 Tree-based Index

One critical feature of relational databases is to handle large amounts of data. In order to store and retrieve those data efficiently, index techniques are widely used. In relational databases the B+ tree [2] is widely used to represent sorted data and offer an efficient way to query, insert, or remove records by keys.

The B+ tree extends the B tree to store all the data at the leaf level, and the interior nodes only store keys. Moreover, the B+ tree builds the bi-directional link chain at the leaf level, which makes the range query more efficient. The structure of B+ Tree is shown in Figure 2-1.
For a $h$-order B+ tree with $h$ levels of index, performing a range query with $k$ elements occurring within the range requires $\Theta(logbn + k)$ operations in the worst case, and the space required to store the tree is $\Theta(n)$. 
3 B+ Tree based Rule-Aware Index

The B+ Tree based Rule-Aware Index extends the traditional B+ Tree by storing the mapping rules lineage to fasten the query performance. In the current chapter, we first discuss the structure and theory of the new index, and then illustrate the related semantics and their examples.

3.1 Index Structure

Normally, in traditional B+ Tree indexes, specific fields (table columns) in the table are indexed in ascend or descend sorted order. When the "a is-a b" Rule exists and user requests on “//b”, it leads to three situations:

1) When both a and b are indexed, the cost = 2 * Index Scan + 1 * Merge Sort;
2) When only one of them is indexed, the cost = 1 * Index Scan + 1 * Table Full Scan + 1 * Merge Sort;
3) When neither of them is indexed, the cost = 2 * Table Full Scan + 1 * Merge Sort.

In this processing model, the query engine calculates the result on-the-fly. The advantage is that the original B+ Tree index can be reused with zero maintenance cost in the index part, but the drawback is also obvious: In the experiments introduced in Chapter 5, the system in this naïve query model has a huge burden when the rule related queries are frequently requested.

The alternative approach, which is introduced in this chapter, is to provide the index with a stable query performance of $\Theta(\log n)$, where $n$ represents the size of XML data. Instead of applying the mapping rules after receiving the query request, we pre-compute the rule mapping and transformation in advance and resemble the B+ Tree index. In this circumstance, when users request the XQuery like “//b”, the query engine directly access the index on b, without being aware of the rule logics.

To achieve this goal, the index system must be aware of the mapping rules, and is designed to maintain the relationship between data and rules.

The proposed index structure, shown in Figure 3-1, contains three parts: (1) Dependency
Graph, (2) B+ Tree with Reference Counting, and (3) Rule Path Table. They will be introduced in the next sections in this chapter.

The transitive properties of mapping rules makes the rules not standalone or independent. They may influence each other to represent more implicit rules. For example, the rule \( a \textit{is-a} b \) and \( b \textit{is-a} c \) create the implicit rule \( a \textit{is-a} c \). The implicit rule makes the index hard to maintain its data. As its mechanism, the indexes always get one “copy” from the main data and make it be synchronized when main data is updated. In the previous example, when “\( c \)” is indexed, this index need to be informed when “\( a \)” is updated. In the target solution, the dependency graph is used to dynamically probe the transitive paths and their implicit rules.

A dependency graph is a directed graph representing dependencies of several objects towards each other [15] . It is possible to derive an evaluation order or the absence of an evaluation order that respects the given dependencies from the dependency graph.

When we leverage the dependency graph to analyze transitive rules, the graph nodes can be seen as XML classes and edges as mapping rules. Figure 3-2 shows how the dependency graph works in mapping rules analysis. In the sample, the edge \( R1 \) represents \( a \textit{is-a} b \), and \( R2 \) represents \( b \textit{is-a} c \). Because of the transitive nature of the “Is-A” Rule, we can induce that \( a \textit{is-a} c \). The mechanism also produces more implicit rules like \( b \textit{is-a} x \) and \( a \textit{is-a} x \).
Indexed Semantic Mapping Rules

To clarify the algorithms in following sections, we define the following concepts.

**Definition 1:** In a dependency graph, the node $N_a$ represents the XML class $a$.

**Definition 2:** In a dependency graph, the edge $R_N$ represents the mapping rule $N$.

**Definition 3:** In a dependency graph, the node which has the index is *Terminating Node*.

**Definition 4:** For specific rule $R_N$, $a$ is-a $b$, $a$ is *StartNode* $(R_N)$, and $b$ is the *EndNode* $(R_N)$.

**Definition 5:** In a dependency graph, a *Path* $(N_a, N_b)$, is a vector of rules, which explicitly or implicitly define the rule $a$ is-a $b$.

**Definition 6:** In a dependency graph, a *PathSet* $(N_a, N_b)$ is the set of *Path* $(N_a, N_b)$, which includes all possible paths from $a$ to $b$.

In the previous sample in Figure 3-2, $<R_1, R_2, R_5>$ is a *Path* $(a, x)$, and $<R_3, R_4, R_5>$ is also a *Path* $(a, x)$. The *PathSet* $(a, x) = \{<R_1, R_2, R_5>, <R_3, R_4, R_5>, <R_1, R_4, R_5>, <R_3, R_2, R_5>\}$

**Definition 7:** In a dependency graph, the *PredecessorSet* $(a)$ is the set of nodes, which contains all predecessor nodes in the graph.

### 3.1.2 Cycle Detection

Through Deep-First-Search (DFS) algorithm we can easily explore the *PathSet* $(N_a, N_b)$ from $N_a$ to $N_b$. However, when the directional graph contains cycles, the algorithm quits when it meets the visited nodes. In the real world, the rule cycle may represent meaningful logic. Here Figure 3-3 shows one rule model in a company. In the graph exists three XML classes, *emp* as Employees, *senior* as Senior Employees, and *committee* as Committee Members. Given these
Indexed Semantic Mapping Rules

classes, three rules are defined as:

![Figure 3-3 Sample of Cycle Detection](image)

<table>
<thead>
<tr>
<th>#</th>
<th>Mapping Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>emp[salary &gt; 10] is-a senior</td>
<td>Employees whose Salary Level great than 10 are Senior Employees;</td>
</tr>
<tr>
<td>R2</td>
<td>senior[year &gt; 5] is-a &lt;emp salary=&quot;25&quot;&gt;</td>
<td>If one Senior Employee work for 6 years or more, his/her Salary Level must be 25.</td>
</tr>
<tr>
<td>R3</td>
<td>senior[salary &gt; 20] is-a committee</td>
<td>Senior Employees whose Salary Level great than 20 are Committee Members.</td>
</tr>
</tbody>
</table>

When the following XML document is indexed, all of the three entities belong to *committee*.

```xml
<doc>
  <emp id="01" salary="22" year="3"/>
  <emp id="02" salary="8" year="7"/>
  <senior id="03" salary="25" year="8"/>
</doc>
```

The first “emp” (id = “01”) and the “senior” (id = “03”) obviously is-a Committee, because the former “emp” passes *R1* and *R3* and then reach the node “committee”, and the *R3* defines the “senior” is-a “committee” because the salary level is great than 20. Now we watch the second
“emp” (id = “02”). He cannot directly reach “committee” by the shortest path <R1, R3>. However, through R1 he is a qualified “senior” first, and through R2 he becomes an “emp” with salary level 25 because he worked for more than 5 years. Finally he successfully passes <R1, R3>, so his transform path is <R1, R2, R1, R3>.

From the previous sample we notice the transform path for the second “emp” is <R1, R2, R1, R3>. The rule R1 shows two times because of the cycle. The mechanism of Automata provides us the traditional way to solve the problems like regular expression. However, in this MDQ case the automata solution is not so qualified. The first reason is, in the MDQ engine each mapping rule only fires once. In other words, the transform paths have limited steps (length). The second reason is that, it is useless to input duplicate rules into MDQ as parameter. The MDQ engine is able to detect cycles on-the-fly. The third reason is the irrelevance of the rule order. When MDQ reads the rule set, the order of the input rules are not critical.

**Theorem 1:** If $P = <R_a, ..., R_b, ..., R_c>$ is a Path($N_a, N_b$), and $R_b$ is in the Cycle $C = <R_1, R_2, ..., R_b, ..., R_1>$, then $(P \cup C)$ is a Path($N_a, N_b$).

The correspond algorithm to support cycle detection is shown here:

**Step 1** Find all non-cycle paths from $N_a$ to $N_b$ through graph traverse algorithm;

**Step 2** Find all cycles in the graph by topological sort algorithm;

**Step 3** Probe new paths with Cycle

```java
foreach p in Paths {
    foreach c in Cycles {
        if (p intersect c) != {} {
            Path += (p union c);
        }
    }
}
```

Applying the above algorithm, the cycle included dependency graph in Figure 3-4 shows the result of $PathSet(a, y) = \{P1 = <R1, R4>; P2 = <R5, R6, R7, R1, R4>; P3 = <R1, R2, R3, R4>; P4 = <R1, R2, R3, R5, R6, R7, R4>\}$
3.1.3 B+ Tree with Insert Reference Counting

Normally the B+ tree supports duplicate keys, which means the B+ tree can store the entries with same key but they point to different entities. When the mapping rules are introduced, it is possible that one record is duplicated stored in the B+ tree.

If \( a \) is-a \( b \), then one specific data item on “\( a \)” probably has more than one path to reach “\( b \)”, which leads to the duplication problem. When the size of \( PathSet(\mathbf{N}_a, \mathbf{N}_b) \) is greater than one, which includes more than one \( Path(\mathbf{N}_a, \mathbf{N}_b) \), we still expect only one “\( a \)” instance in the index, because we do not expect returning several “\( a \)” result when we scan the index in a range query request on “\( b \)”.

The duplicate problem can be solved by extending the leaf nodes in the B+ tree. We add one integer field as “reference count” to the leaf node, and then slightly modify the semantics of the B+ tree as below:

**3.1.3.1 Insert**

When one entry needs to be inserted into the B+ tree, we probe the tree from the root and go as the original algorithm to locate in the leaf level. If the entry does not exist, we insert the entry information as usual, and set the “reference count” to 1. If the entry already exists, we simply increase the “reference count” by 1.

**3.1.3.2 Delete**

To remove entries in the B+ tree, firstly we locate the specific leaves with the key, decrease the “reference count” by 1. If the “reference count” is still greater than 0, then algorithm quits; otherwise, the leaf node needs to be physically removed. In this mechanism, we can guarantee that the entry will be removed only when no path can lead it to the **Terminating Node**.
3.1.4 Rule Path Table

Given one Dependency Graph and a specific node, we can derive all possible paths from each node to the *Terminating Node* in the graph by the algorithm introduced in Chapter 3.1.2. If we populate all those possible paths into one table, we call this table the “Rule Path Table”. Figure 3-5 shows one sample of the Rule Path Table, which represents the dependency graph in Figure 3-2. In the sample, the *Terminate Node* is “x”, the table maintains all possible paths from all nodes to the *Terminate Node*.

Note that even “x” has one empty entry in the table, which means no transformation request.

<table>
<thead>
<tr>
<th>From</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>R1, R2, R5</td>
</tr>
<tr>
<td>a</td>
<td>R1, R4, R5</td>
</tr>
<tr>
<td>a</td>
<td>R3, R2, R5</td>
</tr>
<tr>
<td>a</td>
<td>R3, R4, R5</td>
</tr>
<tr>
<td>b</td>
<td>R2, R5</td>
</tr>
<tr>
<td>b</td>
<td>R4, R5</td>
</tr>
<tr>
<td>c</td>
<td>R5</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-5 Rule Path Table Sample

**Definition 8:** A *Leaf Reference List* (*Path(N_a, N_b)*) is a list, which records the address of B+ Tree leaves, which originally belong to a, and through *Path(N_a, N_b)* could reach b.

Each entry in the Rule Path Table, which represents one path from specific node to the indexed node, maintains one *Leaf Reference List*, which points to specific B+ Tree leaves. The *Leaf Reference List* of path p records the location of leaf nodes, which reach the indexed node through the path p.

If a *is-a* b, then one specific data item on “a” probably has more than one path to reach “b”, which means several path may contain the reference to the same leaf node.

As the extension of B+ tree, the Rule Path Table is the critical part to maintain the relationship between rules and data. However, as the pay-off, we need large space to store the Rule Path...
Table. In the B+ tree, each leaf node exists with a positive reference count, which means at least one Leaf Reference points to it. So the result is, the total size of Leaf References is greater or equal than the size of its B+ tree.

When the mapping rules are fairly stable and not asked to be removed, the Rule Path Table is not necessary when user build the index. Even the remove is occasionally required, the Rule Path Table can be generated dynamically.

### 3.2 Semantics

The semantics of Rule-Aware Index are defined in the following sections. The target index supports not only traditional index semantics like insert and remove, but also reflects changes when mapping rules are added or removed. In Chapter 3.2.1, the semantics of adding a “Is-A” rule is defined. Chapter 3.2.2 - 3.2.4 introduces the semantics of data insert, remove, and update. Chapter 3.2.5 defines the semantic of rule removal.

#### 3.2.1 Add “is-a” Rules

Assuming node “a” and “b” in one Dependency Graph, when one new "a is-a b" Rule is introduced, only “a” and its predecessors are impacted. The impacts include: (1) some nodes could reach “b” and its successors because of the new rule; (2) some nodes are able to reach “b” and its successors by alternative paths.

We explain this theory with an example in Figure 3-6, which shows the Dependency Graph with a Terminating Node “z”. Assuming the R5 is the last rule added into the graph, then the impacted nodes are \{m, x, s\}. Note that although data from “s” and “x” can already reach “z”, R5 provides them an alternative path to the Terminating Node.

![Figure 3-6 Mapping Rules in the Dependency Graph](image)
3.2.1.1 Algorithm

**Step 1** New paths are evaluated. When one new rule $R$ is introduced, it may generate more paths from $R$’s predecessors to the terminate node;

$$\text{Incremental Path Set} = \{P_1, P_2, P_3, ..., P_n\}$$

**Step 2** Pick up the first Path $p$ from Incremental Path Set,

**Step 3** Query the MDQ engine, capsule $p$ into Rule Set as the parameter, then we receive the result set from $\text{QueryMDQ}(p)$. The result set needs to filter the data from $\text{StartNode}(p)$.

**Step 4** Sort the result set, and then insert the key/address pairs into the B+ tree index. Through the insertion, we get the references of the B+ Tree leaves as $\text{Leaf Reference Set}$.

**Step 5** Add an entry in Rule Path Table for $p$, the “From” field is the Starting Point of $p$ field. Store the $\text{Leaf Reference Set}$ in the entry.

**Step 6** Remove $p$ from Incremental Path Set. If the $\text{Incremental Path Set}$ is not empty, go to **Step 2**.

The pseudo code is shown in Figure 3-7.
3.2.1.2 Example

This example illustrates the effects when new rules are introduced. Giving the following XML document.

```xml
<doc>
```
Figure 3-8 shows the final Dependency Graph, which represents the following three rules with predicates.

\[
\begin{align*}
\text{Rule 1:} & \quad a \text{ is-a } b \\
\text{Rule 2:} & \quad b[@c = 2] \text{ is-a } x \\
\text{Rule 3:} & \quad b[@d = 4] \text{ is-a } x
\end{align*}
\]

In the example, the \( R2 \) and \( R3 \) already exist, and now we add Rule 1. As the consequence the explicit rule “\( a \text{ is-a } b \)” is obviously created, and one implicit rule “\( a \text{ is-a } x \)” is generated as well. The predicates on \( R2 \) and \( R3 \) make the data instance on node “\( a \)” may pass either \( R2 \) or \( R3 \), or even both. The initial status of the rule path table is shown in Table 3-1.

<table>
<thead>
<tr>
<th>FROM</th>
<th>PATH</th>
<th>REFERENCE LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(&lt;x c=&quot;5&quot; /&gt;)</td>
</tr>
</tbody>
</table>
| b    | R2   | \(<b c="2" />\) \\
|      |      | \(<b c="2" d="4" />\) |
| b    | R3   | \(<b c="2" d="4" />\) \\
|      |      | \(<b d="4" />\) |

Table 3-1 Initial Status
Step 1 Evaluate new paths from Node a and its predecessors to the terminate node x;

Incremental Path Set = {<R1, R2>, <R1, R3>}

Step 2a Pick up the first Path <R1, R2> from Incremental Path Set,

Step 3a Query the MDQ Engine, capsule <R1, R2> into Rule Set as the parameter, then we receive the result set.

Result Set = {<a c="2" d="4" />, <b c="2" />, <b c="2" d="4" />, <x c="5" />}

then, Filtered Result Set (a) = {<a c="2" d="4" />}

Step 4a Insert the key/address pairs into the B+ tree index. Through the insertion, we get the address of the B+ Tree leaves as Leaf Reference Set.

Step 5a Add an entry in Rule Path Table for <R1, R2>, store the Leaf Reference Set in the entry as shown in Table 3-2.

<table>
<thead>
<tr>
<th>FROM</th>
<th>PATH</th>
<th>REFERENCE LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>&lt;x c=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>b</td>
<td>R2</td>
<td>&lt;b c=&quot;2&quot; /&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;b c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>b</td>
<td>R3</td>
<td>&lt;b c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;b d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>a</td>
<td>R1, R2</td>
<td>&lt;a c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
</tbody>
</table>

Table 3-2

Step 6a Remove <R1, R2> from Incremental Path Set. The Incremental Path Set is not empty, go to Step 2b.

Step 2b Pick up the first Path <R1, R3> from Incremental Path Set,

Step 3b Query the MDQ Engine, capsule <R1, R3> into Rule Set as the parameter, then we receive the result set.

Result Set = {<a c="2" d="4" />, <b c="2" d="4" />, <b d="4" />, <x c="5" />}

then, Filtered Result Set (a) = {<a c="2" d="4" />}

Step 4b Insert the key/address pairs into the B+ tree index. Through the insertion, we get the address of the B+ Tree leaves as Leaf Reference Set.

Step 5b Add an entry in Rule Path Table for <R1, R3>, store the Leaf Reference Set in the entry as shown in Table 3-3.
### Indexed Semantic Mapping Rules

<table>
<thead>
<tr>
<th>FROM</th>
<th>PATH</th>
<th>REFERENCE LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>&lt;x c=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>b</td>
<td>R2</td>
<td>&lt;b c=&quot;2&quot; /&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;b c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>b</td>
<td>R3</td>
<td>&lt;b c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;b d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>a</td>
<td>R1, R2</td>
<td>&lt;a c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>a</td>
<td>R1, R3</td>
<td>&lt;a c=&quot;2&quot; d=&quot;4&quot; /&gt;</td>
</tr>
</tbody>
</table>

Table 3-3

**Step 6b** Remove $<R1, R3>$ from Incremental Path Set. Now the *Incremental Path Set* is empty, End.

### 3.2.2 Scan new XML documents

When new XML documents are added, the data instances need to be scanned one by one before the related indexes are picked up to update. Assume there is one index that was built on XML class "x", and one instance of "a" are scanned by the MDQ Engine. That "a" needs to be indexed if the mapping rule "a is-a x" exists.

#### 3.2.2.1 Algorithm

**Step 1** Calculate the $\text{PathSet}(N_a, N_x)$, where the data comes from a and the target index is built on "x";

**Step 2** Pick up the first Path $p$ from $\text{PathSet}(N_a, N_x)$;

**Step 3** Query the MDQ Engine, capfulate $p$ into Rule Set as the parameter, then we receive the result set (key/address pairs) from $\text{QueryMDQ}(p, \text{doc})$. The result set needs to filter the data from $\text{StartNode}(p)$;

**Step 4** Insert the result items into B+ Index. After insertion, we get all inserted leaf addresses;

**Step 5** Append the leaf addresses to the entry $\text{LeafReferenceList}(p)$ in the Rule Path Table;

**Step 6** Remove $p$ from $\text{PathSet}(N_a, N_x)$. If the $\text{PathSet}(N_a, N_x)$ is not empty, go to Step 2.
The pseudo code is shown in Figure 3-9.

```java
// Insert new data on node a
// B+ Tree Index on node x
insertData(Node a, Node x, Document newData, RulePathTable table) {

    // Find all possible paths from a to x
    Set pathSet = getAllPath(a, x);

    foreach p in pathSet {
        // Get all the results from new Doc
        // passing through p
        List result = queryMDQ(p, "//x", newData);
        // Only the data from the start node is needed
        result.filter(p.startNode);

        // Insert into B+ Tree, and then append the
        // Reference List in the Rule Path Table
        foreach r in result {
            LeafRef leaf = insertToIndex(r.key);
            if (!table.exists(p)) {
                table.addEntry(p);
            }
            table.getEntry(p).addReference(leaf);
        }
    }
}
```

**Figure 3-9 Pseudo Code of Adding New Data**

### 3.2.2.2 Example

This example illustrates how the MDQ index evaluates and inserts new data. Giving the following XML document, the dependency graph and the initial status of related index structure are shown in Figure 3-10 and Figure 3-11 respectively.

```xml
<doc>
    <a c="2" d="4"/>
</doc>
```
Step 1  Calculate the $PathSet(N_a, N_x) = \{<R1, R2>, <R1, R3>\}$.

Step 2a  Pick up the first Path $p = <R1, R2>$ from $PathSet(N_a, N_x)$;

Step 3a  Query the MDQ Engine, capsule $p$ into Rule Set as the parameter, then we receive the result set from $QueryMDQ(p, doc)$;

Step 4a  Insert the result items into B+ Index. After insertion, we get all inserted leaf addresses;

Step 5a  Append the leaf addresses to the entry $LeafReferenceList(p)$ in the Rule Path Table. Figure 3-12 shows the index snapshot at this point.
Step 6a  Remove $p$ from $PathSet(N_a, N_x)$. $PathSet(N_a, N_x)$ is not empty, go to Step 2b.

Step 2b  Pick up the first Path $p = <R1, R3>$ from $PathSet(N_a, N_x)$;

Step 3b  Query the MDQ Engine, capsize $p$ into Rule Set as the parameter, then we receive the result set from $QueryMDQ(p, doc)$;

Step 4b  Insert the result items into B+ Index. Because the leaf already exists, the B+ Tree only increase the Reference Counting to 2. After insertion, we get all inserted leaf addresses;

Step 5b  Append the leaf addresses to the entry $LeafReferenceList(p)$ in the Rule Path Table;

Step 6b  Remove $p$ from $PathSet(N_a, N_x)$. Now the $PathSet(N_a, N_x)$ is empty, algorithm quits. The final status of the index is presented in Figure 3-13.
3.2.3 Remove XML documents

When XML documents are required to be removed, the removing data instances needed to be scanned to analyze the impacts to existing indexes. Assume there is one index was built on XML class “x”, and one instance of “a” is requested to be removed. The entry for “a” in index “x” must be removed before the deleting, if the mapping rule “a is-a x” exists.

3.2.3.1 Algorithm

**Step 1** Calculate the $PathSet (N_a \ N_x)$, where the data removed from a and the target index is built on “x”.

**Step 2** Pick up the first Path $p$ from $PathSet (N_a \ N_x)$;

**Step 3** Query the MDQ Engine, capsule $p$ into Rule Set as the parameter, then we receive the result set (key/address pairs) from $QueryMDQ(p, doc)$. The result set needs to filter the data from $StartNode(p)$;

**Step 4** Remove the key/address pairs from the B+ tree index. Through the remove, we get the addresses of affected leaves in the B+ Tree leaves as $LeafReferenceList(p)$.

**Step 5** Locate the entry in Rule Path Table for $p$, the “From” field is the Starting Point of
p field. Remove $\text{LeafReferenceList}(p)$ from the entry.

**Step 6** Remove $p$ from $\text{PathSet}(N_a, N_x)$. If the $\text{PathSet}(N_a, N_x)$ is not empty, go to Step 2.

The pseudo code is shown in Figure 3-14.

```
// Remove data on node a
// B+ Tree Index on node x
removeData(Node a, Node x, Document oldData, RulePathTable table) {

    // Find all possible paths from a to x
    Set pathSet = getAllPath(a, x);

    foreach p in pathSet {

        // Get all the results from old Doc
        // passing through p
        List result = queryMDQ(p, "//x", oldData);

        // Only the data from the start node is needed
        result.filter(p.startNode);

        // Remove from B+ Tree, then remove the related
        // Reference Node in Rule Path Table
        foreach r in results {
            LeafRef leaf = removeFromIndex(r.key);
            table.getEntry(p).removeLink(leaf);
        }
    }
}
```

**Figure 3-14 Pseudo Code of Removing Data**

### 3.2.3.2 Example

This example illustrates how the MDQ index evaluates and removes data. In the example we try to remove the following XML document.

```
<doc>
    <a c="2" d="4"/>
</doc>
```

We continue to use the example in the previous section, so the initiation status is shown in
Figure 3-13.

**Step 1** Calculate the $\text{PathSet}(\mathcal{N}_a, \mathcal{N}_x) = \{<R1, R2>, <R1, R3>\}$;

**Step 2a** Pick up the first Path $p=<R1, R2>$ from $\text{PathSet}(\mathcal{N}_a, \mathcal{N}_x)$;

**Step 3a** Query the MDQ Engine, capssulate $p$ into Rule Set as the parameter, then we receive the result set (key/address pairs) from $\text{QueryMDQ}(p, \text{doc})$;

**Step 4a** Remove the key/address pairs from the B+ tree index. Because the Reference Counting number is great than 1, the remove action only minus 1 from the number, Through the remove, we get the addresses of affected leaves in the B+ Tree.

**Step 5a** Locate the entry in Rule Path Table for $p$, remove the leaf addresses from its $\text{LeafReferenceList}(p)$. Figure 3-15 shows the snapshot of the index.

**Step 6a** Remove $p$ from $\text{PathSet}(\mathcal{N}_a, \mathcal{N}_x)$. The $\text{PathSet}(\mathcal{N}_a, \mathcal{N}_x)$ is not empty, go to Step 2b.

**Step 2b** Pick up the first Path $p=<R1, R3>$ from $\text{PathSet}(\mathcal{N}_a, \mathcal{N}_x)$;

**Step 3b** Query the MDQ Engine, capssulate $p$ into Rule Set as the parameter, then we receive the result set (key/address pairs) from $\text{QueryMDQ}(p, \text{doc})$;

**Step 4b** Remove the key/address pairs from the B+ tree index. After decreasing the Reference Counting, the number equals to 0, so the leaf is ready to be removed. Through the remove, we get the addresses of affected leaves in the B+ Tree.

**Step 5b** Locate the entry in Rule Path Table for $p$, remove the leaf addresses from its

---

**Figure 3-15**

<table>
<thead>
<tr>
<th>From</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>R2</td>
</tr>
<tr>
<td>a</td>
<td>R1, R2</td>
</tr>
<tr>
<td>b</td>
<td>R3</td>
</tr>
<tr>
<td>a</td>
<td>R1, R3</td>
</tr>
</tbody>
</table>

"Rule Path" Table
Leaf Reference List \(p\).

**Step 6b** Remove \( p \) from \( \text{PathSet}(N_a, N_x) \). The \( \text{PathSet}(N_a, N_x) \) is empty, algorithm quits. The final status of the index is presented in Figure 3-16.

![Figure 3-16](image)

### 3.2.4 Update XML documents

In general, the update procedure in MDQ indexes is same as the traditional B+ tree. When indexed entries are requested to be updated, two different situations need to be considered.

In the first case, the key field remains, but the value field in the leaf node is modified, which happens when the data instance is moved to a different file. The updating algorithm first finds the leaf node in the tree, then updates the value field.

In the second case, when the key field is changed, the updating procedure can be seen as one remove operation in Section 3.2.3 plus one insert operation in Section 3.2.2.

### 3.2.5 Remove “is-a” Rules

In the dependency graph, removing one rule (edge) may nullify several rule paths. Through the Rule Path Table, we can easily find all entries whose rule path contains specific rules. So, when one rule is removed, all related entries in the Rule Path Table need to be removed, and this
operation is the only operation which does not query the MDQ engine.

### 3.2.5.1 Algorithm

**Step 1** Find all entries from Rule Path Table which contains the removing rule $R$. Store the result into $RemovePathSet$.

**Step 2** Pick up the first Path $p$ from $RemovePathSet$.

**Step 3** Get the $LeafReferenceList(p)$, for each list item, decrease the reference count of its B+ tree leaves by 1. If the reference count equals zero, remove the leaf.

**Step 4** Remove $p$ from Rule Path Table. If the $RemovePathSet$ is not empty, go to Step 2.

The related pseudo code is shown in Figure 3-17.
### 3.2.5.2 Example

This example illustrates how the MDQ index remove one specific mapping rule. In the example we try to remove R1 from Figure 3-8. The initial index status shows in Figure 3-18.
Step 1 In the Rule Path Table, only two entries contain the rule R1. So \( RemovePathSet = \{<R1, R2>, <R1, R3>\} \)

Step 2 Pick up the first Path \( p = <R1, R2> \) from \( RemovePathSet \);

Step 3 Get the \( LeafReferenceList(p) \), for each list item, find its reference leaves from the B+ Tree and then decrease the Reference Count by 1. In this example, two leaves affected, but neither of them needs to remove, because those Reference Count still greater than 0. The process is shown in Figure 3-19.
Step 4  Remove $p$ from Rule Path Table. The $RemovePathSet$ is not empty, go to Step 2b.

Step 2b  Pick up the first Path $p = \langle R_1, R_3 \rangle$ from $RemovePathSet$;

Step 3b  Get the $LeafReferenceList(p)$, for each list item, remove its reference leaves from the B+ Tree. In this example, both leaves need to be removed, because after the removal those Reference Count equal 0. The result is shown in Figure 3-20.
**Step 4b**  Remove $p$ from Rule Path Table. The $RemovePathSet$ is now empty, algorithm quits and the final status is shown in Figure 3-21.

**“Rule Path” Table**

<table>
<thead>
<tr>
<th>From</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>R2</td>
</tr>
<tr>
<td>$b$</td>
<td>R3</td>
</tr>
</tbody>
</table>

Figure 3-21
4 Integration into MDQ

In this chapter the prototype system which implements the theory part of the previous chapter will be introduced. The prototype is implemented as an Eclipse RCP program in Java.

4.1 Architecture Overview

As shown in Figure 4-1, the whole system was designed in four parts: (1) Index Management, (2) Storage Management, (3) Interface to MDQ Engine, and (4) Query Interface.

When user queries or manipulates the from the Query Interface, the request is first passed to the Index Management module, which analyzes the possibility to leverage indexes. In particular, when mapping rules are introduced or removed, the Index Management module checks the dependency graph to analyze the impact of existing indexes.

If a specific index is picked up for the user request, the related range query is executed in its B+ Tree structure and then one list of related XML file IDs are returned. Those file IDs are candidates for the query. Through the Storage Management part the system gets the file handles to the actual file system.

Once the XML file handles are ready, the MDQ Engine Interface part packs it to the MDQ Engine, combined with rules exported from the dependency graph. The execution of mapping
query generates the output as XML results, and they are shown to users in the Query Interface.

4.2 Index Management

The Index Management module has two major parts. The first part is the pool of Rule-Aware Indexes, which includes one traditional B+ tree with reference counting, and one attached Rule Path Table per index. The second part is the Dependency Graph, which only exists once and is shared among all indexes. In the prototype system, the Catalog window, shown in Figure 4-2, manages XML data, index instances, and elements (nodes and rules) in the Dependency Graph.

![Figure 4-2 Catalog Window](image)

The structure and algorithm of Rule-Aware Index are fully described in Chapter 3. In addition, two design issues need to be solved. (1) Which indexes should be used during the query, and (2) Which indexes should be updated when mapping rules are changed (insert or remove).

The prototype solves the first problem in the manual approach. To pick up the right index automatically, the XQuery statement need to be analyzed, which is out of scope of the thesis. The manual way to assign the index is introduced in Chapter 4.5.

For the second problem, the following algorithm finds out all the impacted indexes when the mapping rule “a is-a b” is either inserted or removed:

1. **Step 1** Calculate the successor node set \( s \) of \( N_a \) in the Dependency Graph;
2. **Step 2** Calculate the node set \( t \), whose elements bind at least one index;
3. **Step 3** Update the index:
Indexed Semantic Mapping Rules

```java
foreach n in (s intersect t) {
    foreach i in indexSet(n) {
        i.update();
    }
}
```

4.3 Storage Management

In the tree-based index structure, the database system extracts the key range from the target query and then tries to find the tree leaves which hold the keys. In B+ tree leaves, the location of target information piece should be found.

In this solution for XML documents, instead of storing the location of XML nodes, the tree leaves store the location of XML file (absolute file path). The advantage of this approach is that we bypass the complexity of unstructured text context when building the index, and make more flexible to ad-hoc queries.

However, the absolute file path, which is stored as String type is still two large for B+ Tree leaves. So in the implementation all files are indexed in one array of strings, and the index of the array member is the File ID, which stores at the leaves of the B+ Tree. Figure 4-3 illustrates the storage structure.
Besides simplicity, this approach is very efficient when each XML document only store small amount of indexed entities. In most commercial application systems, the information entities are stored one instance for each file. However, in another extreme case, the query performance is absolute destroyed when every records are maintained in one huge XML file. In that way, all the index nodes record the same File ID, which makes the whole index useless.

### 4.4 MDQ Engine Interface

The MDQ Interface is taken as the single connect point to the MDQ Engine. The main tasks of the interface consist of (1) Issue query request against XML documents, and (2) Scan XML documents to extract index information for indexing.

#### 4.4.1 Query XML Documents

As shown in Figure 4-4, the index management model communicate with MDQ engine through `queryMDQ()` function, which includes two variants. The former queries against the whole XML storage, and the latter only targets one specific XML data file.
1. Query All Docs: List result = queryMDQ(RuleSet, XQuery);
2. Query Specific Doc: List result = queryMDQ(RuleSet, XQuery, DocumentId);

Refer to Section 3.1.1, the index management model maintains a dependency graph, which is shared to all indexes to analyze the dependency relationship. In the graph each edge represents one mapping rule, so one specific path in this directional graph equals to one vector of mapping rules.

In the example in Figure 4-4, the node “a” can reach the terminating node “x” through maximally four paths \(<R1, R2, R5>, <R1, R4, R5>, <R3, R2, R5>, <R3, R4, R5>\). Particularly, to find out all the instance of “a” that could be mapped to “x” through the specific path <R3, R2, R5>, the following function call retrieves the candidate set:

\[ \text{List candidates = queryMDQ(<R3, R2, R5>, "/x")} \]

Notice that the candidates result set candidates includes the data instance not only from “a”, but also possibly from “b”, “c”, and “x”. So, to get all the results from “a”, one more filter step need to be fulfilled.

Figure 4-4 Interface to MDQ

### 4.4.2 Extract Index Information

To build the index we need two pieces of information: keys and document identities. As we
discussed in previous section, we simply store the File ID, the offset number of one array, as the location.

Leveraging the XQuery functionality from MDQ Engine, we can easily calculate the address as XQuery result by the following request, the key and fileId attributes could be inserted into the B+ Tree.

```xml
for $a in //element
return <address fileID="ID" key="{$a/key}" />
```

Figure 4-5 Indexing Documents

The Figure 4-5 illustrates the process of indexing documents. When new documents need to be indexed, the index management model issues the above XQuery to MDQ engine. After that the query engine returns the address list with key value and document identity as XML format. Finally the index management model extracts the address results and inserts into specific B+ Tree index.

**4.5 Query Interface**

Query Interface, the user interface to the MDQ engine and its index management module,
provides a easy-to-use environment for querying with index support. As shown in Figure 4-6, the Eclipse form allows user to input the XQuery statement and specify maximal two indexes and their key range. When two indexes are applied, the user need to clarify the AND/OR predicates according to the XQuery’s WHERE clause.

Once the query is successfully executed, the query result is shown in the middle column, and the execution cost and its index selectivity are listed on the right “History” table.

![Figure 4-6 Query Interface GUI](image)
5 Experimental Evaluation

5.1 TPoX Benchmark

Transaction Processing over XML (TPoX) [6] [7] is an application-level XML database benchmark based on a financial institution scenario. Three major entities, Customer, Security and Order describe the simple security trading system. It is used to evaluate the comprehensive performance of XML database systems. In the current experiment environment we test the correctness and performance of the new index structure by using the documents generated from the TPoX Benchmark as the original scenarios. To evaluate the mapping rule aware scenarios, some of original data instances are rewritten.

The prototype system was implemented in Java and all experiments are performed on a PC-compatible machine, with a single Intel 2.5GHz T9300 CPU and 2GB RAM, running Windows Vista Enterprise (SP1) and JRE 1.6.0. Each experiment is repeated five times and the average of the five results is used as the measurement.

5.2 Query without Mapping Rules

In the TPoX benchmark, seven core queries are defined to measure the query performance. The first six queries only query against one schema, the seventh query which needs to join two different schemas is not supported by current implementation of the MDQ index. So in the final experiments, the following six core query were used.

<table>
<thead>
<tr>
<th>Core Queries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. get_order</td>
<td>Returns full order document without the FIXML root element.</td>
</tr>
<tr>
<td>2. get_security</td>
<td>Returns a full security document.</td>
</tr>
<tr>
<td>3. customer_profile</td>
<td>Extracts customer elements and constructs a new profile document.</td>
</tr>
</tbody>
</table>
The corresponding XQuery statements are shown below.

<table>
<thead>
<tr>
<th>Core Queries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_order</td>
<td>for $ord in //FIXML where $ord/Order/@ID=&quot;103318&quot; return $ord/Order</td>
</tr>
<tr>
<td>get_security</td>
<td>for $s in //Security where $s/Symbol= &quot;ACTP&quot; return $s</td>
</tr>
</tbody>
</table>
| customer_profile | for $cust in //Customer where $cust/@id=1052 return <Customer_Profile CUSTOMERID="{$cust/@id}">
        {%cust/Name}
        {%cust/DateOfBirth}
        {%cust/Gender}
        {%cust/Nationality}
        {%cust/CountryOfResidence}
        {%cust/Languages}
        {%cust/Addresses}
        {%cust/EmailAddresses}
    </Customer_Profile> |
| search_securities | for $sec in //Security where $sec/SecurityInformation/*/Sector= "Energy" and $sec/PE[. >=30 and . <35] and $sec/Yield>4.5 return <Security>
        {%sec/Symbol}
        {%sec/Name}
        {%sec/SecurityType}
        {%sec/SecurityInformation//Sector}
        {%sec/PE}
        {%sec/Yield}
    </Security> |
For each core query, two configurations need to be evaluated. We first execute the XQuery against specific number of XML files without any index support, then in the second configuration we build the indexes and re-evaluate the query performance. The former configuration is called “Naïve Mode”, and the latter is shown as “Indexed Mode”.

Table 5-1 and Figure 5-1 summarizes the execution results in our environment, and the conclusions are:

- In Naïve Mode, the costs are shown as linear curves. The reason is that, all the files related to the same schema need to be scanned, no matter of the selectivity of the predicates in the XQuery;

- The Indexed Mode show significant improvements, and the cost are almost stable. The underlying B+ Tree provides the performance of $O(\log n)$, where $n$ represents the size of XML documents.
### Indexed Semantic Mapping Rules

#### Table 5-1 Query Performance Without Mapping Rules

<table>
<thead>
<tr>
<th>Num of Files</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>500</td>
<td>1,000</td>
<td>5,000</td>
<td>10,000</td>
<td>100</td>
</tr>
<tr>
<td>Naïve (ms)</td>
<td>176</td>
<td>665</td>
<td>1,476</td>
<td>13,582</td>
<td>28,983</td>
<td>203</td>
</tr>
<tr>
<td>Indexed (ms)</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

#### Figure 5-1 Query Performance Without Mapping Rules

1. **Q1: get_order**
2. **Q2: get_security**
3. **Q3: customer_profile**
4. **Q4: search_securities**
5. **Q5: account_summary**
6. **Q6: get_security_price**
5.3 Experiments with Mapping Rules

To evaluate the performance when mapping rules involve in XML data, we derivate the TPoX data source to generate 3 different data set, which include 10, 100, and 1000 schemas respectively. Meanwhile the mapping rule set for those schemas are provided.

5.3.1 Building Index

The cost to build the MDQ index with Rule Path Table is expensive. In the algorithm introduced in Chapter 3.2, each XML document need to be scanned for each Rule Path in the Rule Path Table of the MDQ index. So theoretically the cost model could be described as :

$$\Theta(m \times n)$$

$m$: the size of XML data, $n$: the number of rule paths

In the index without Rule Path Table, only one rule path that includes all the mapping rules is built. So in this case $n = 1$, so we can deduce that the building cost for indexes without Rule Path Table is $\Theta(m)$, where $m$ is the size of XML data.

In the experiments, the result proved the theory in the thesis. Table 5-2 to Table 5-4 illustrate the cost to index 10000 files is linear to the number of schema and size of XML data, when the algorithm take extra computation to track the rule-data relationships in Rule Path Table. The experiments also show the efficiency in building the index without the table. The absence of the table does not impact the correctness of query. In addition, it saves the memory space as well. However, as the penalty, the whole index has to be rebuilt when any of rules is updated or removed.

<table>
<thead>
<tr>
<th>Order</th>
<th>Num. of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>9,934</td>
</tr>
<tr>
<td>1,000</td>
<td>93,254</td>
</tr>
<tr>
<td>10,000</td>
<td>942,103</td>
</tr>
<tr>
<td>10</td>
<td>985</td>
</tr>
<tr>
<td>100</td>
<td>7,406</td>
</tr>
<tr>
<td>1,000</td>
<td>66,764</td>
</tr>
</tbody>
</table>

Table 5-2 Cost (ms) for building Index on Order (Number of Files = 10000)
Building the MDQ index without Rule Path Table is more efficient in mid or large size environments. Even considering the index rebuild when rules are changed, the total cost is still smaller than the approach with table.
5.3.2 Querying

In this part we quantify the benefit of MDQ Index enabled query by measuring the processing time in different set of data volume and rules, and analyze the sensibility to (1) the number of XML documents, (2) the number of mapping rules, and (3) the number of schemas.

We define $N_F$ as the number of XML documents, $N_R$ as the number of mapping rules, and $N_S$ as the number of schemas. Three environments are set as follow:

1. $N_F = 100, 500, 1000, 5000$ and $10000$; $N_R = 10$; $N_S = 10$;
2. $N_R = 10, 100, and 1000$; $N_S = 10$; $N_F = 10000$;
3. $N_S = 10, 100, and 1000$; $N_R = 10$; $N_F = 10000$;

Experiment Results are shown in Table 5-5 to Table 5-7, and Figure 5-3 to Figure 5-5 plot the results in scatter graphs. From the experiments we summarize that:

- Holding $N_R$ and $N_S$ constant, naïve mode queries show positive linear cost to $N_F$, and indexed mode queries present stable performance;

- Holding $N_S$ and $N_F$ constant, naïve mode queries show positive linear cost to $N_R$, and indexed mode queries present stable performance;

- Holding $N_R$ and $N_F$ constant, both naïve mode and indexed mode queries show stable performance;

The index mode queries show similar performance as those in non-mapping environment in chapter 5.2. The underlying reason is that the TPoX benchmark queries have very high selectivity, so only a few documents are qualified for the predicates. MDQ indexes return the file IDs and only those qualified files are sent to MDQ engine, so those influences on $N_F$, $N_S$, and $N_R$ only slightly impact the performance in TPoX environment.

In query scenarios, whether the rule path table is created does not influence the performance, because the index search is based on the standard B+ tree algorithm and does not access the rule path table. In addition, in our experiment environment, the memory is big enough to hold the B+ tree and the rule path table, so the cost for the swap files between memory and hard disk can be ignored.
### Indexed Semantic Mapping Rules

#### Table 5-5 Cost (ms) of Query (Number of Rules = 10, Number of Schemas = 10)

<table>
<thead>
<tr>
<th>Q1: get_order</th>
<th>214</th>
<th>1,096</th>
<th>2,128</th>
<th>11,995</th>
<th>21,876</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2: get_security</th>
<th>188</th>
<th>1,045</th>
<th>2,006</th>
<th>9,688</th>
<th>19,656</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3: customer_profile</th>
<th>388</th>
<th>1,787</th>
<th>3,733</th>
<th>13,556</th>
<th>24,381</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q4: search_securities</th>
<th>399</th>
<th>1,834</th>
<th>3,986</th>
<th>19,644</th>
<th>35,580</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q5: account_summary</th>
<th>390</th>
<th>2,002</th>
<th>4,043</th>
<th>20,465</th>
<th>26,498</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q6: get_security_price</th>
<th>210</th>
<th>1,029</th>
<th>2,139</th>
<th>11,538</th>
<th>19,132</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (ms)</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

#### Figure 5-3 Cost of Query (Number of Rules = 10, Number of Schemas = 10)
Indexed Semantic Mapping Rules

<table>
<thead>
<tr>
<th>Num of Rules</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naïve (ms)</strong></td>
<td>19,552</td>
<td>30,858</td>
<td>24,082</td>
<td>38,541</td>
<td>24,847</td>
<td>31,774</td>
</tr>
<tr>
<td><strong>Indexed (ms)</strong></td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>16</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5-6 Cost (ms) of Query (Number of Files = 10000, Number of Schemas = 10)

![Graphs of Q1: get_order, Q2: get_security, Q3: customer_profile, Q4: search_securities, Q5: account_summary, Q6: get_security_price](image)

Figure 5-4 Cost of Query (Number of Files = 10000, Number of Schemas = 10)
Indexed Semantic Mapping Rules

Table 5-7 Cost (ms) of Query (Number of Files = 10000, Number of Rules = 10)

<table>
<thead>
<tr>
<th>Q1</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>45,620</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>42,580</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>50,225</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>56,236</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>61,411</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>62,495</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>44,399</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>51,495</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>39,995</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q4</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>35,865</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>37,701</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>40,396</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q5</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>30,039</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>28,766</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>25,594</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q6</th>
<th>Num of Schema</th>
<th>Naïve (ms)</th>
<th>Indexed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>29,497</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>30,367</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>32,942</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 5-5 Cost of Query (Number of Files = 10000, Number of Rules = 10)
5.3.3 Removing Data

If the index contains the rule path table, the algorithm structure of removing data is similar to constructing the index. To remove a specific data instance on node \(a\), all rule paths with starting node \(a\) need to be considered. The experiments proved this point, Table 5-8 and Figure 5-6 illustrate that the average cost to remove a document is sensitive to \(N_R\). Because the size of rule path table is irrelevant to the \(N_S\), holding the \(N_R\) constant the cost is stable while \(N_S\) increases.

Table 5-8 Average Cost (ms) of Removing one XML Document (Number of Files = 10000)

<table>
<thead>
<tr>
<th>Num of Rules</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
<td>431</td>
<td>4,018</td>
</tr>
<tr>
<td>100</td>
<td>41</td>
<td>428</td>
<td>4,117</td>
</tr>
<tr>
<td>1,000</td>
<td>41</td>
<td>433</td>
<td>4,190</td>
</tr>
</tbody>
</table>

Figure 5-6 Average Cost (ms) of Removing one XML Document (Number of Files = 10000)

Without the rule path table, the index does not maintain the lineage of the data, and acts as a normal B+ tree index. With \(n\) documents, the average removal cost is \(\Theta(\log n)\). In the experiment with 10000 documents, the cost to remove one document is around 1 ms.
5.3.4 Removing Rules

Refer to the algorithm in chapter 3.2.5, to remove a mapping rule does not require querying the MDQ engine, which makes the rule dispose much faster than removing documents.

<table>
<thead>
<tr>
<th>Num of Schema</th>
<th>Num of Rules</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>41</td>
<td>42</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>46</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>2</td>
<td>6</td>
<td>52</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-9 Average Cost (ms) of Removing one Mapping Rule (Number of Files = 10000)

Figure 5-7 Average Cost (ms) of Removing one Mapping Rule (Number of Files = 10000)

Table 5-9 and Figure 5-7 show the average cost to remove one rule in the MDQ index with 10000 documents. One special trend is shown in the result, that sometimes the cost decreases significantly when $N_S$ increases. The underlying reason is that when $N_S$ is greater than $N_R$, less documents are indexed in the MDQ index, because the size of rule path table mainly depends on $N_R$. However, the increase in cost is not appreciable when specific number of files are indexed, and shows its resistance to $N_S$ and $N_R$.

When the index is built without the rule path table, the entries in the B+ tree cannot be easily
removed. Refer to chapter 3.1.4, one B+ tree leaf may have multiple references if it is qualified to several rule paths. Without the rule path table, the algorithm never knows whether there exists other rule paths to reach the data instance. So in that case, the whole index has to be rebuilt (Refer to chapter 5.3.1).
6 Conclusion & Future Work

In the thesis the mapping rules aware index model for XML database is proposed and one prototype is implemented, which leverages the traditional B+ tree structure to reduce the data access in high-volume data processing environment. The main contributions include (1) the Dependency Graph is used to analyze the impact to indexes, (2) through the Rule Path Table the relationship between data and rule is traceable, which makes the rule modification more efficient, and (3) algorithms to implement the semantics of the index.

The MDQ index shows its significant advantage than naïve query model, when we query large amount of XML documents with mapping rules. By delimiting the range of qualified documents, the index-powered query model can maximally minimize the data input for the MDQ engine. However, since the cost to maintain the rule path table of the index is positive relevant to the number of mapping rules, the rule path table is not recommended to build when mapping rules are frequently changed. In that case, a MDQ index without rule path table is more suitable.

There are several parts could be discussed and improved in the future work. First, currently XML documents are stored in the file system in a flat structure and there is no extra metadata to categorize those documents. When building the Leaf Reference List for each Rule Path, all the XML data in the storage have to be scanned once, which is not necessary in some cases. So, the storage model need to categorize XML documents to minimize the cost when rules are changed. Second, in the mature XML database system the index should be picked up automatically by analyzing the query statement, which was implemented manually in current prototype. Finally, current queryMDQ() interface is not efficient, each cycle only one XML document is evaluated.
Bibliography

[2] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Index
[3] Daniela Florescu, Martin Hentschel, Donald Kossmann, Tim Kraska: On the fly Data Integration
[5] Cong Yu, Lucian Popa: Constraint-Based XML Query Rewriting for Data Integration
[12] Daniela Florescu, Donald Kossmann: Storing and Querying XML Data using an RDMBS