ABSTRACT

Shah, Ashish

Improving Query Performance using Materialized XML Views: A Learning-based approach

(Under the direction of Dr. Rada Chirkova)

This thesis presents a novel approach in solving the problem of improving the efficiency of query processing on an XML interface of a relational database for frequent and important queries. The motivation of this research is provided by the need to eliminate processing overheads in converting relational data to an XML format by materializing beforehand answers to frequent and important queries (which we predefine as a query workload) in terms of an XML structure. The main contribution of this paper is to show that selective materialization of data as XML views reduces query-execution costs for the workload queries, in relatively static databases. Our learning-based approach precomputes and stores (materializes) parts of the answers to the workload queries as clustered XML views. In addition, the data in the materialized XML clusters are periodically incrementally refreshed and rearranged, to respond to the changes in the query workload. We use a collection of music data as a sample database to build our learning-based system. Our experiments show that the approach can significantly reduce processing costs for frequent and important queries on relational databases with XML interfaces.

Improving Query Performance using Materialized XML Views:
A Learning-based approach

by

Shah Ashish Narendra

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh, NC

2004

APPROVED BY:

Dr. Munindar Singh   Dr. Peng Ning

Dr. Rada Chirkova

BIOGRAPHY

Ashish Shah graduated with a Bachelor of Engineering (B.E) degree in Computer Engineering from University of Mumbai, Mumbai, India in June 1999.

In fall 2001, he joined the masters program in Computer Science at North Carolina State University, Raleigh, NC. He completed his masters thesis under the direction of Dr. Rada Chirkova.

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my advisor Dr. Rada Y. Chirkova, for her mentorship throughout the research progress and in believing and supporting my ideas. I thank Dr. Munindar P. Singh and Dr. Peng Ning for participating on my committee.

I would like to thank all my friends Pushkar, Prakash, Adi, Rachana and lately Mayuri and Chirag for bearing with me for all the long hours at research and writing my thesis. I would like to thank Naveen, Sameer and Arnav for the interesting research discussions in the lab. I would like to thank John Toebes, my manager at Cisco Systems for giving me the flexibility to complete my thesis.

Finally, I would like to thank my parents Narendra and Smita and my brother, Ankit for offering their love and support all through my education and sharing their knowledge and wisdom with me throughout my life.

INDEX

LIST OF FIGURES		v
1.	**INTRODUCTION**	**1**
1.1	Assumptions	3
1.2	CDDB: A motivating example	4
1.3	Organization of the remainder of the document	4
2.	**RELATED WORK**	**5**
2.1	Querying and data models	6
2.2	Semantic Caching review	6
2.3	View generation review	7
2.4	Effects of data updates on materialization	8
3.	**PRELIMINARIES AND PROBLEM FORMULATION**	**9**
3.1	Relational query (SQL) on XML data	9
3.2	XML query techniques versus relational query techniques (SQL)	9
3.3	Machine learning	9
3.4	Problem specification	10
3.5	The Cost model	10
4.	**OUTLINE OF THE APPROACH**	**12**
4.1	Defining key terms	12
4.2	Statistics-based learning approach	14
4.3	Clustering	14
4.4	Quality check of the materialized data	14
5.	**SYSTEM ARCHITECTURE**	**16**
5.1	Query processing subsystem	16
5.2	Setting up materialized XML clusters	17
6.	**THE LEARNING ALGORITHM**	**20**
6.1	Learning I: Discovering access patterns in the relations of interest	21
6.2	Learning II: Materializing XML and forming clusters (consolidation phase)	21
6.3	Building an expected query stream	22
6.4	Choosing the threshold value	22
7.	**CDDB AND THE LEARNING ALGORITHM**	**23**
7.1	Clustering the CDDB database	23
7.2	Query subsystem search path: An example	25
8.	**EXPERIMENTAL SETUP AND RESULTS**	**28**
8.1	Hardware and software setup	28
8.2	CDDB database setup	28
8.3	Experimental Results	30
9.	**CONCLUSIONS AND FURTHER WORK**	**35**
9.1	Future work	36
10.	**BIBLIOGRAPHY**	**37**

LIST OF FIGURES

Figure 1: Disc and Track relations	4
Figure 2: Query processing subsystem	16
Figure 3: Some tuples in the Disc relation in the CDDB database	23
Figure 4: Some tuples in the Track relation in the CDDB database	23
Figure 5: Simple cluster	24
Figure 6: Related simple cluster	24
Figure 7: Complex cluster	25
Figure 8: Sample XML query in XQuery syntax (1)	25
Figure 9: Sample XML query in XQuery syntax (2)	26
Figure 10: Partially rewritten query	26
Figure 11: Fully rewritten query	27
Figure 12: Some tuples in the Disc relation in the CDDB database	28
Figure 13: Some tuples in the Track relation in the CDDB database	29
Figure 14: Tuples in the Disc relation with the modified schema	29
Figure 15: Materialized Schema	29
Figure 16: Materialized XML	29
Figure 17: Time taken by the tagger routine to convert relational data to XML	30
Figure 18: Runtime for queries on materialized XML data and relational data	31
Figure 19: Query execution times for a large set of queries on XML clusters	33
Figure 20: Query execution times for a mixture of queries	33
Figure 21: Distribution of data at the cluster and relational level for queries in increasing time domain	33

1. INTRODUCTION

The extended markup language (XML) [26] is a simple and flexible format that is playing an increasingly important role in publishing and querying data in the World Wide Web. The XML standard supports multiple data type definitions that can help in the integration of disparate business applications. Applications using XML either use these schemas for data exchange. Thus, irrespective of the format applications exchange data seamlessly using the XML format.

As XML has become a de facto standard for business data exchange, it is imperative for businesses to make their existing data available in XML for their partners. At the same time, most business data are still stored in relational databases. A general way to publish XML data in relational databases is to provide XML interfaces over the stored relations and to enable querying the interfaces using XML query languages. In response to the demand for such frameworks, database systems with XML interfaces over non-XML data are increasingly available, notably relational systems from Oracle, IBM, and Microsoft.

In this thesis, we consider the problem of improving the efficiency of evaluating XML queries on relational databases with XML interfaces. When querying a data source using its XML interface, an application issues a query in an XML query language and expects an answer in XML. If the data source is a relational database, this way of interacting with the database adds new dimensions to the old problem of efficiently evaluating queries on relational data. In the standard scheme for evaluating queries on an XML interface of a relational database, the relational query-processing engine computes a relation that is an answer to the query on the stored relational data; see [13] for an overview. On top of this process, the query-processing engine has to (1) translate the query from an XML query language into SQL (the resulting query is then posed on the relational data), and (2) translate

the answer into XML. To efficiently process a query on an XML interface of a relational database, the query-processing engine has to efficiently perform all three tasks.

We propose an approach to reducing the amount of time the query-processing engine spends on answering frequent and important queries on XML interfaces of relational databases [32]. The idea of our approach is to circumvent the standard query-answering scheme described above, by precomputing and storing, or *materializing*, some of the relational data as XML views. If the DBMS has chosen the "right" data to materialize, it can use these XML views to answer some or most of the frequent and important queries on the data source without accessing the relational data. We show that our approach can significantly reduce the time to process frequent and important queries on relational databases with XML interfaces.

Our approach is not the first view-based approach to the problem of efficiently computing XML data on relational databases. To clarify how our approach differs from previous work, we use the terms (1) *view definitions*, which are data specifications given in terms of stored data (or possibly in terms of other views), and (2) *view answers*, which are the data that satisfy the definition of a view on the database. In past work, researchers have looked into the problem of efficiently evaluating XML queries over XML view definitions of relational data (e.g., SilkRoute [12] or XPERANTO [23]). We build on the past work by adding a new component to this framework: We *incrementally materialize XML view answers* to frequent and important XML queries (which we predefine as a query workload) on a relational database, using a learning approach. To the best of our knowledge, we are the first to propose this approach.

The following are the contributions of this thesis:

- We develop a learning-based approach to materializing relational data in XML.

- We propose a system architecture that takes advantage of the materialized XML to reduce the total query-execution times for predefined query workloads.
- We show how to transform a purely relational database system to accommodate materialized XML and our system architecture.

Using our approach may result in significant efficiency gains on relatively static databases (databases not requiring frequent updates). Moreover, it is possible to combine our solution with the orthogonal approaches described in [12, 23], thus achieving the combined advantages of the two solutions.

1.1 Assumptions

This section discusses specific assumptions that need to be considered while using the approach discussed in this thesis.

Predefined query workloads

We assume that there is a fixed set of frequent and important queries on the database; these queries are predefined as a *query workload*. (Only these queries are answered in terms of the XML views that we materialize.)

Lazy approach for updates

We use a lazy approach for updates in this system. As a result, our materialized clusters are not always up-to-date with the underlying relational database. We sacrifice an optimal result set for a quick response and near optimal results. For an effective response time to workload queries, it is necessary that the materialized data remain unchanged while querying takes place to fully realize the advantages of materialization. On the other hand, if frequent updates to the materialized data are necessary, the workload queries have to access the underlying relations instead of the materialized data. This issue is further discussed in Chapter 9.

1.2 CDDB: A motivating example

We use the CD database (CDDB [30]) as a motivating example to explain our approach. CDDB is a collection of information about music CDs, each with one or more tracks. The relational database consists of two relations, the *Disc* relation and the *Track* relation. The *Disc* relation holds the names of all the CD titles along with other information like *genre*, which classifies the type of a disc, while the *Track* relation contains the name of all the tracks for each disc.

Relation name: **Disc**

cd_id	cd_title	genre	num_of_tracks
a50b550c	Air Supply	Cddb/Misc	10

Relation name: **Track**

cd_id	track_title
a50b550c	All out of Love

Figure 1: Disc and Track relations

CDDB can be considered as a database that does not require frequent updates. When an application queries the CDDB using a relational query mechanism (*eg.* SQL) about a track, it also retrieves information about the disc and other tracks in the same collection. In addition, if the application desires the output be in XML format, then there is an additional task of converting the relational data to XML.

1.3 Organization of the remainder of the document

Chapter 2 discusses related research. Chapter 3 formally defines the problem and states the assumptions. Chapter 4 presents a brief outline of the approach. Chapter 5 discusses the architecture of the system. Chapter 6 describes the learning algorithm in detail. Chapter 7 shows a practical application of the learning algorithm. Chapter 8 discusses the experimental setup and the results of experiments using this system. Chapter 9 discusses future work and states our contributions.

2. RELATED WORK

In this chapter, we describe areas of research that are related to our approach.

2.1 Querying and data models

The Document Object Model (DOM) [31] and the Simple API for XML (SAX) [31] data models deal with nodes (each branch of the XML tree) and node events (an update, insert or delete event on a node) respectively. XML, XQuery and XQL [31] are some XML query techniques used to query these data models. We describe related work that discusses the storage and querying of XML data by applications.

[11] proposes a system for representing and querying semi-structured data. The system provides an interface over nested tables that can efficiently store semi-structured data. The system also provides a query interface that can take advantage of nested tables. The objective is to reduce complexity found in typical query languages for semi-structured data. This approach is a parallel to our approach. However, in our approach the data is stored in an XML format as an attribute of a relation instead of as nested tables. We use a combination of SQL and XQuery query techniques to access these data.

[14] proposes the Wiccap Data Model, an XML data model that maps Web information sources into commonly perceived logical models. The system extracts information from websites and arranges the extracted data as a data model. Users can then easily access information using this data model. Our approach materializes data in XML using a machine-learning algorithm. The data is mapped onto relations and stored as a DOM model.

[21] is a system devised to collect data coming from different XML data sources, and store them in a format independent of the source format; its query language is able to query

the database and generate new XML documents. The paper discusses the integration of three different basic technologies, Relational DBMS, Java and XSLT, under a unifying framework. We use a Relational DBMS, XQuery and Java to design an interface over the relational engine that provides a mechanism to materialize XML data from relations as well as query materialized data. We do not generate any XML documents.

Rainbow [28] is designed to exploit relational database technology to manage XML data based on a flexible mapping strategy. Input queries expressed as XQuery are used to generate an extraction view query to create a view. A user's request combined with this extracted view, pushes down as much computation as possible to the underlying relational database and returns the XML results back to the user. In our system, we use a machine-learning algorithm to extract and materialize views unlike the Rainbow system that only defines views created by user queries. While the Rainbow system handles updates of extracted views, our system currently does not support the update of materialized views.

2.2 Semantic Caching review

[3, 5, 33] discuss Semantic Caching. In [3, 5] Semantic Caching is a method for caching XML data from various sources. The query mechanism is optimized to use cached XML queries and their resulting datasets. One aspect that distinguishes semantic caching from other tuple-caching methods is that the client-side caching of data is organized in terms of cached XML queries. Semantic Caching uses cached views and shows XQuery containment by exploring the connections between XML and tree automata. XQuery is used to express both queries and views. Our system also uses XQuery to query materialized views, though the concept of server-side caching is more applicable to our system than client-side caching.

Besides, we organize data in terms of the frequency of access and not in terms of XML queries.

2.3 View generation review

SilkRoute [12] and XPERANTO [23] are systems that use XML interface-based view generation techniques. These systems are most closely related to our system. XPERANTO actually forms a basis for our approach. We extend the XPERANTO system by adding XML materialization and learning layers which is discussed in the following sections (5.1, 5.2, 6).

XPERANTO creates virtual XML views of the underlying relational data. Businesses can then exchange data by querying these XML views. Incoming XQuery queries are converted to SQL to execute using the relational engine. The resulting relational tuples are converted to XML and returned to the querying application. The middleware layer does both query translation and XML tagging. The system also has a default XML view on which application specific views can be created.

SilkRoute is a framework for publishing relational data using XML view definitions. Tailored virtual views are constructed by the database administrator. A client application then issues queries on these virtual views. The input queries use the XQuery query language. SilkRoute converts these input XQueries to SQL, executes them on the relational database, and assembles the output relational streams as an XML document.

SilkRoute and XPERANTO do not actually materialize any of the views, but rather defines virtual views. In our approach, we materialize all the views that are generated. The middleware layer in our system does query translation and XML materialization based on learning.

[25] motivates the concept of a data warehouse architecture and proposes technical issues when using the architecture. The data warehouse architecture is related to our work because it uses an eager or the *in-advance* approach to data integration. In this approach, interesting data is extracted in advance; filtered, translated and merged with data from various other sources to create an in-advance repository. Future queries are then made directly on the in-advance repository instead of the original data source. This data warehouse architecture is very similar to the architecture we use. However, in our approach the issues are related to the storage and querying of XML. The paper also discusses issues involved with translation, change detection, integration and maintenance. Our approach addresses the integration issue using concept (or rule) learning [16, 17].

2.4 Effects of data updates on materialization

[2] discusses storage techniques for materialized views and methods to handle schema / data changes in base relations as well as methods to handle incoming queries because of these changes. Materialized views are kept up-to-date using compensation queries that propagate data changes from the underlying relational database. However, if the schema changes then anomalies arise in these compensation queries. The paper attempts to answer the question: “what if data or schema changes when materialization is taking place”. Traditional DBMS systems solved this problem using the ACID properties to make the entire join process as a single transaction [34]. In a similar way, this paper considers using the entire process of propagating compensation queries as on transaction.

While we do not address the issues of updating materialized clusters in our system, we expect to face similar issues in the case when the XML clusters would be updated.

3. PRELIMINARIES AND PROBLEM FORMULATION

In this chapter, we introduce some preliminary concepts. We then define the problem and key terms associated with it. We discuss the cost model used to measure the performance of the queries. We state the assumptions in our system.

3.1 Relational query (SQL) on XML data

Relational query techniques are effective on relations that store data in the form of tuples. For example, achieving a join on two relations based on a conjunctive condition is trivial using relational techniques while the same join operation on data stored in an XML format will be a non-trivial task. The query engine needs to parse the tree structure several times before it can output a result using the join condition.

3.2 XML query techniques versus relational query techniques (SQL)

XML query languages are based on *regular path expressions* that traverse the XML graph structure or tree. Typical XML query processing systems convert XML data into an object representation. The regular path expressions are evaluated either using joins across these objects or as index structures. Creating indexes is often an expensive operation, which means that XML query techniques come out slower in terms of data access speeds as compared to their SQL counterparts.

3.3 Machine learning

[16] describes the concept of machine learning. Machine learning can be defined as a strategy that automatically improves an application's performance using past instances. Learning approaches are used in various fields, one such learning approach is the Statistical

Learning method. The learning strategy discussed in our approach is based on machine learning concepts and statistical learning. We describe our system as one that learns on certain sets of rules and concepts.

3.4 Problem specification

Queries made on relational databases that require output in an XML format incur an additional overhead of converting the relational data to XML. The incoming XML query is rewritten as a relational query and input to the relational engine. The result set from the relational engine is converted to XML and returned to the querying application. For application workload queries that repeat over time, it is beneficial to cache the results of such queries. However, the question is how to achieve a tradeoff between the amounts of data that can be cached versus a quick response time for incoming queries.

3.4.1 Definitions of key terms

We define key terms related to the problem specification.

XML relational data source

In this document, we denote a relational database system and XML interface to the data in the system as an "XML relational data source", and abbreviate as "data source". For a query in an XML query language (e.g., XQuery), to *evaluate the query on a data source* means to obtain an XML answer to the query via the XML interface of the source.

Query workload

Let there exist a finite set of important queries that a user or application needs to pose on the data. These important queries are called a *query workload*. Each query in this workload has an associated frequency, which shows the relative weight of the query in the workload.

3.5 The Cost model

Our cost model for the system is the *total cost of evaluating a query workload*; we define this cost as the weighted sum of the costs of evaluating all workload queries, using the relative weights (frequencies) of the individual queries in the workload. We consider the problem of improving the efficiency of evaluating a query workload on a data source; the goal is to minimize the total cost of evaluating the query workload on a XML relational data-source. To reduce the average cost of evaluating a workload query; we concentrate on using novel techniques to materialize and query data. We incrementally materialize XML views that contain workload-relevant data.

4. OUTLINE OF THE APPROACH

In this chapter, we present a general outline of our approach. We fist define a few key terms. We then discuss the use of a statistics-based learning approach and clustering of materialized XML views.

In the proposed system, when answering a workload query, the query-processing engine first searches the materialized XML views, rather than the relational tables; if the answer to the query can be computed using just the views, the query-processing engine does not access the underlying relations. Using this approach may result in significant efficiency gains when the underlying relational data do not change very often.

When constructing materialized views, we need to decide which data to materialize in XML. A simple solution is to have the entire relational database materialized as an XML view answer. This way, we would be able to avoid the overhead of translating *all possible* queries (rather than just the workload) on the data source into SQL, and of translating the relational answers to the queries into XML. However, query performance might degrade considerably, as XML query-answering techniques are slower than their relational counterparts (Section 3.2). In addition, the system incurs a significant overhead of keeping the XML data consistent with the underlying relations.

4.1 Defining key terms

Using a relatively static database

We select a database that is not changing constantly or a relatively static database for the relational module. The CDDB is a good example. A collection of discs in the CDDB remains

unchanged until another disc is released. In addition, the tracks on existing discs remain constant.

View materialization

To *materialize a view* means to compute and store in the database the answer to an application's query or in short a *view* on the database. We materialize views in XML rather than as relations to reduce the time required to translate the workload queries from an XML query language into SQL and the relational answers to the queries into XML.

Learning

In the context of our approach, learning can be defined as the process of minimizing a result set of tuples (returned by a query on a relational database) based on their access statistics. Learning is based on concepts and rules [16, 17].

Clusters

We define a cluster as a collection of smaller materialized XML views derived from relations. These views are merged together to form a single larger XML view.

Related and interesting XML records

We define *related XML records* as a collection of tuples selected in the learning process that are subsequently used to form clusters. For example in the CDDB database, if two *discs* feature the same artist, then both the discs and their tracks will be returned as related records. Further, only a subset of related tuples are actually selected to form clusters. These records are called interesting related records.

Threshold value

Threshold value is an empirically determined value for the access count of relational tuples determined by workload queries returning these tuples as part of a result set; exceeding which the Materialization phase is invoked.

4.2 Statistics-based learning approach

We use a learning-based approach to materialize only the data that are needed to answer the most frequent and important queries on the data source. In database systems, it is common to maintain statistics on the stored data, for the purposes of query optimization [13]. We maintain similar statistics on access rates to the data in the stored relations, by keeping an access count for every tuple in the relations of interest; each access to a tuple increments its access count. We materialize the most frequently accessed tuples in XML. We use learning techniques combined with these access rate statistics to decide when and how to change, incrementally, the set of records materialized in XML.

4.3 Clustering

We manage the materialized data using the concept of clustering. In our approach, we define clustering as combining related XML records (Section 3.4.1) into a single materialized XML structure. Related XML records are tuples selected in the learning process that can be merged together to form a cluster. These XML structures are stored in a special relation in the database and can be queried using the data source's XML query language. (In the remainder of this thesis, we assume that XQuery [31] is the language of choice.) Storing the most frequently accessed tuples in materialized XML clusters increases the probability that future workload queries will be satisfied by the clusters. To answer those queries that are not satisfied by the XML clusters, we use the relational query-processing engine.

4.4 Quality check of the materialized data

We ensure the quality of materialized clusters by comparing the answers to the workload queries on the materialized XML to the answers to the same queries on the underlying

relational data. Only when these answers become similar we start answering real workload queries using materialized XML. The quality check stage is part of the system startup; this stage involves the following steps:

1. Test-run all the queries in the predefined query workload on the materialized clusters.
2. Run the same set of workload queries on the relational database and apply the tagging routine (See Section 5.1).
3. Compare the resulting XML data from steps 1 and 2.

The results from Step 3 help determine whether the data in the materialized clusters are in-sync with the data in the underlying relational database. On successful completion of Step 3, we can say that the system has reached a steady state. Now the materialized XML can be used to answer workload queries from users; the expected query paths are described in Chapter 5.

5. SYSTEM ARCHITECTURE

In this chapter, we discuss the architecture of the system. We divide the discussion into three sections. First, we discuss the query processing subsystem and explain how a typical query behaves within the system. Second, we discuss the required changes needed to the schema of the existing relational database to accommodate the learning approach. Finally, we discuss the process of generating XML data from stored relations using workload queries and the process of setting up materialized clusters.

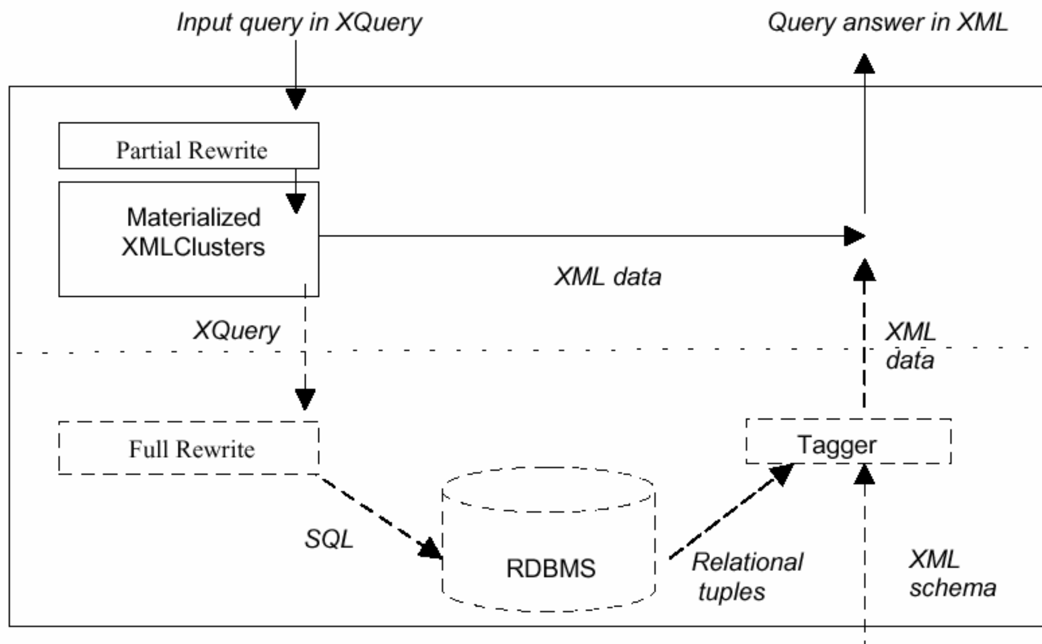

Figure 2: Query processing subsystem

5.1 Query processing subsystem

We discuss the typical query path taken by an input query. The solid lines in Figure 2 show the primary query path, which is tried for all queries on the data. If a workload query can be answered by the materialized XML clusters, then only the primary path is taken. Otherwise, the query next follows the secondary query path, shown in dotted lines in Figure 2; here, the input query is pushed down to the relational level and is answered using the stored relations, rather than the materialized XML. The resulting tuples are then converted into an XML

format using the *Tagger* module. The schema for the output format is given by the database administrator.

The XML clusters are stored as values of an attribute in a special relation. The system queries this relation using SQL to find the most relevant cluster, and then poses the XQuery query on the cluster. The schema for the clusters is specified by the database administrator.

5.2 Setting up materialized XML clusters

In this section, we discuss the process of forming complex XML clusters (Section 4.3). This section describes a method for transforming a purely relational database schema to a schema that can accommodate materialized XML views. Consider a relational schema with just two relations (for simplicity) $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_n)$. A_1 is the primary key for the relation *R.* In the following section (5.2.1), we show that this approach can also be extended to more than two relations.

5.2.1 Modifying the given relational schemas

We need to keep track of the stored tuples that are accessed by workload queries. For this purpose, we modify one or more of the relations in the original schema to include an additional attribute to store the *access count* for each accessed tuple. The most likely candidates for selection to add this extra attribute are relations that have a high access rate. These may be relations that appear in query projections or relations that are involved in expensive joins. For instance, suppose that in the input workload we have a query that involves a join of the relations *R* and *S*. If the relation *R* in the data source is large, the query would be expensive to evaluate, hence we consider *R* as a suitable candidate for the schema change. (Alternatively, the database administrator can choose the schema to modify.)

We decide to add an attribute $A_{(n+1)}$ to the schema of the relation R. This attribute will store access counts for the tuples in relation R. Initially, the value of the attribute $A_{(n+1)}$ is NULL in all tuples of the relation R.

Original Schema

$R(A_1, \ldots, A_n)$

$S(B_1, \ldots, B_n)$

Modified Schema

$R(A_1, \ldots, A_n) \rightarrow R(A_1, \ldots, A_n, A_{(n+1)})$

$S(B_1, \ldots, B_n)$

Similarly if there were three relations in the original schema, $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_n)$, $P(C_1, \ldots, C_n)$ and it is known that relations R and S are high access relations, then both R and S are modified.

Original Schema

$R(A_1, \ldots, A_n)$

$S(B_1, \ldots, B_n)$

$P(C_1, \ldots, C_n)$

Modified Schema

$R(A_1, \ldots, A_n) \rightarrow R(A_1, \ldots, A_n, A_{(n+1)})$

$S(B_1, \ldots, B_n) \rightarrow S(B_1, \ldots, B_n, B_{(n+1)})$

$P(C_1, \ldots, C_n)$

This approach can be applied to any number of relations but in this discussion, we use the two-relation schema for simplicity.

5.2.2 Creating relations for materialized XML clusters

The next step involves the creation of a special relation that will store materialized XML clusters. Let this relation be T. It has two attributes A_1 and C. Recall that A_1 is the key of the

relation R; using this attribute in the relation T helps us index the materialized XML clusters in the same way as the relation R. The attribute C is used to store the materialized XML clusters in text format.

To summarize, we set up materialized XML clusters in the following steps:

Original Schema

$R(A_1, \dots, A_n)$

$S(B_1, \dots, B_n)$

Modified Schema

$R(A_1, \dots, A_n) \rightarrow R(A_1, \dots, A_n, A_{(n+1)})$

$S(B_1, \dots, B_n)$

$T(A_1, C)$

1. Select a relation of interest (R in the above example) to modify.
2. Add an access-count attribute to the schema of the selected relation.
3. Create a new relation (T in the above example), to store the materialized XML version of the data in the selected relation (R) of interest.

The relation S remains unchanged as we selected relation R in the data source as a suitable candidate for the schema change (Section 5.2.1). In the general algorithm, if there are r relations then Step 1 consists of $r-1$ relations.

In the following chapters (6.1, 6.2, 7.1), we describe the process of selection and formatting of materialized data to be stored in the relation T.

6. THE LEARNING ALGORITHM

In this chapter, we describe a learning algorithm that populates and incrementally maintains materialized XML clusters. We describe the process of selecting relational tuples for materialization and explain our clustering strategy for building an XML tree of *related and interesting records*. Related and interesting XML records are a subset of tuples selected in the learning process that can be merged together to form a cluster. (See Section 4.1).

Our general approach is as follows. When answering queries, we first pose each query on the materialized XML clusters in the relation *T* that we have added to the original schema. Whenever a query cannot be answered using the materialized XML clusters (or at system startup, see next paragraph), the query is translated from an XML query language into SQL and pushed down to the stored relations. Each time this happens, the system increments access counts for all tuples that contribute to the answer to the SQL query.

At the time of system startup, the relation *T* that holds the materialized XML is empty. As a result, all queries on the data source have to be translated into SQL and pushed to the relational query-processing engine. The materialization phase starts when the access counts in the relations of interest exceed an empirically determined *threshold value*; all tuples whose access counts are greater than the threshold value are materialized into XML. The schema for the materialized XML is specified by the database administrator. As the learning algorithm executes, the most frequently accessed tuples in the relations of interest are materialized into XML and stored in the relation *T*.

6.1 Learning I: Discovering access patterns in the relations of interest

To incrementally materialize and maintain XML clusters of workload-relevant data, the system periodically runs a learning process that translates frequently accessed relational

tuples into XML and reorders the resulting records in a hierarchy of clusters. In this section we describe the first stage of the learning process, where the system discovers access patterns in the relations of interest by using the access-count attribute. Once an access pattern is established, the system translates the most frequently accessed tuples in the relations into XML. To obtain the current access pattern, the system needs to execute the following steps.

1. (This step is executed during startup of the system, before any real queries are posed on the system.) Input an expected query stream and set up the desired output XML schema.
2. Pose the incoming workload queries on the stored relations; in answering the queries, increment the access counts for those tuples in the relations of interest that contribute to the answers to the queries.

6.2 Learning II: Materializing XML and forming clusters (consolidation phase)

Once the first stage of the learning process has discovered access patterns in the relations of interest, the system performs, in several iterations, the following steps (step 3 is discussed in more detail in Section 5.2):

1. To generate materialized XML records, retrieve from the relations of interest all tuples whose access counts are greater than the predetermined threshold value (See Section 4.1).
2. Translate the retrieved relational data into XML and store the results in the materialized XML relation.
3. Form clusters:
    a. Find all relational tuples that are related to the materialized XML, w.r.t. the workload queries.

b. Select the tuples whose access counts exceed the threshold value, and translate them into XML.
c. Cluster the selected tuples and materialized XML into a single XML tree.

6.3 Building an expected query stream

During the startup of the system, we use expected, rather than real, queries to determine access patterns in the relations of interest. For example, if each workload query may use one of the given 250K keywords with given frequencies, then we select the 1000 most-frequent keywords as our expected queries.

Bursts in query streams are a real world characteristic of workload queries. Sudden bursts in queries bias the relational tuple access counts that are used in the learning algorithm. To avoid this problem, we run a count-reset procedure that resets the access counts for a randomly chosen set of tuples.

6.4 Choosing the threshold value

In our approach, we determine the threshold value empirically: At system startup time, we repeat the two learning stages (Section 6.1 and Section 6.2) several times to arrive at a suitable value. The choice of the threshold value is a tradeoff between having a large materialized relation and having better query execution times. A lower threshold value means a larger proportion of tuples is selected in the materialization phase; consequently, more queries will be satisfied in the XML clusters. A higher threshold value prevents a large proportion of relational tuples from being selected for materialization. This limits the number of queries that can be satisfied in the clusters. The key is to strike a balance between the point at which the system materializes tuples and the proportion of records to be materialized.

7. CDDB AND THE LEARNING ALGORITHM

We used the CDDB database as a motivating example (Section 1.2) to explain our approach and as the experimental database in the query processing subsystem discussed in Section 5.1. The CDDB is a collection of data about music records each having a disc title and one or more tracks. The relational database consists of two relations, the *Disc* relation and the *Track* relation. The *Disc* relation holds the name of all CD titles along with other information.

Relation name: **Disc**

cd_id	cd_title	genre	num_of_tracks
a50b550c	Air Supply	Cddb/Misc	10
020e6d12	Eric Clapton	Cddb/Misc	13

Figure 3: Some tuples in the *Disc* relation in the CDDB database

Relation name: **Track**

cd_id	track_title
a50b550c	All out of Love
a50b550c	Lost in Love
020e6d12	Tears in Heaven

Figure 4: Some tuples in the *Track* relation in the CDDB database

7.1 Clustering the CDDB database

In this section, we describe the clustering process. Clustering in this context is defined as the process of consolidating several related simple clusters into a complex cluster. A simple cluster is created using the learning process to select *interesting related records* from the underlying relational database. A complex cluster is a collection of simple clusters that are related. (See *interesting related records* in Section 4.1).

<disc_tracks>
<id>a50b550c</id>
<title>Air Supply</title>
<genre>cddb/misc</genre>
<track index = '1' offset='150'>All out of Love</track>
<track index = '2' offset='981'>Lost in Love</track>
</disc_tracks>

Figure 5: Simple cluster

Figure 5 shows an example of a simple cluster that is materialized in the first step of the materialization process. The data in the cluster consist of tuples from two relations, *Disc* and *Track*. In this case it is the result of a simple join on these two relations with a keyword match on “Air Supply”.

<disc_tracks>
<id>ed111311</id>
<title>Air Supply / The Ultimate Collection</title>
<genre>cddb/misc</genre>
<track index = '1' offset='153'>The Scene</track>
</disc_tracks>

Figure 6: Related simple cluster

Figure 6 shows an example of a simple cluster that shares a condition with the cluster shown in Figure 5. We can say Figure 5 and Figure 6 are related clusters based on a keyword match on the *title* – “Air Supply”. A related cluster is found during the second learning phase (Section 6.2) and the resulting set of related simple clusters are merged into a single XML structure called a complex cluster.

```
<disc_cluster>
    <disc_tracks>
        <id> a50b550c </id>
        <title>Air  Supply / Air Supply</title>
        <genre>cddb/misc</genre>
        <track index = '1' offset='150'>All out of Love</track>
        <track index = '2' offset='981'>Lost in Love</track>
    </disc_tracks>
    <disc_tracks>
        <id>ed111311</id>
        <title>Air Supply / The Ultimate Collection</title>
        <genre>cddb/misc</genre>
        <track index = '1' offset='153'>The Scene</track>
    </disc_tracks>
</disc_cluster>
```

Figure 7: Complex cluster

Figure 7 shows an example of a complex cluster formed by the integration of simple clusters shown in Figure 5 and Figure 6.

7.2 Query subsystem search path: An example

This section provides conditions on which the consolidated XML structure can be searched and it provides a schema that is used to construct the resulting XML. Alternatively, the schema for this materialized XML structure can be specified by a database administrator.

Example 1: All disc titles with 'Air Supply' in its title

```
<discs>
  {
   FOR $d IN disc
     WHERE $d/title = "Air Supply%"
   RETURN $d
  }
</discs>
```

Figure 8: Sample XML query in XQuery syntax (1)

Figure 8 shows an example of an input query that uses the XQuery syntax to query materialized XML clusters for all discs that have "Air Supply" in the *title*.

Example 2: All disc tracks with a keyword match on *Love* in one of their track titles

```
<discs>
  {
   FOR $d IN disc
     WHERE $d/title = "Air Supply%"
            FOR $t IN $d/track
               WHERE $t/track_title =”%Love%”

   RETURN $d
  }
</discs>
```

Figure 9: Sample XML query in XQuery syntax (2)

Figure 9 shows an example of an input query that uses the XQuery syntax that returns all *discs* that have “Air Supply” as *title* and have “Love” in any of their *track titles*.

7.2.1 Partial rewrite to SQL

```
1. Select xmlcluster from cluster_store
where cd_title like “Air Supply%”

2. Select xmlcluster from cluster_store
where cd_title like “Air Supply%”
```

Figure 10: Partially rewritten query

Figure 10 shows a partial query rewrite of the XQuery queries shown in Figure 8 and Figure 9 into SQL. This query is a selection query that returns the relevant materialized clusters (See Figures 6 and 7) stored in the materialized relation (See Relation *T* in Section 5.2.2). For the relation *T(A1, C)* defined in Section 5.2.2, *cluster_store* is the relation *T*, *xmlcluster* is the attribute *C* where the clusters are stored and *cd_title* is the primary key *A1*.

7.2.2 Full rewrite to SQL

1. SELECT * FROM disc, track WHERE
disc.cd_id = track.cd_id AND disc.cd_title=”Air Supply%”

2. SELECT * FROM disc, track WHERE
disc.cd_id = track.cd_id AND disc.cd_title=”Air Supply%” AND
track.track_title like “%Love%”

Figure 11: Fully rewritten query

Figure 11 shows a full rewrite of XML queries in XQuery syntax shown in Figure 8 and Figure 9 into SQL. We described in Section 5.1 that workload queries are tried on the materialized clusters first and then on the relational database. The partial rewrite phase is invoked to narrow down this set of materialized clusters while the full rewrite phase is invoked when the materialized clusters do not return any results.

We show in our experimental results that a large proportion of queries are satisfied before reaching the relational phase (See Figures 19, 20 and 21 which show the decrease in query times as queries are satisfied by the materialized clusters). In cases where the materialized XML is unable to satisfy the input query- only then is the *full rewrite* phase invoked.

8. EXPERIMENTAL SETUP AND RESULTS

In this chapter, we describe the experimental setup in terms of the hardware and software used for the system. We then describe a systematic application of the schema change process discussed in Chapter 6 to the CDDB schema. Finally, we present the results of our experiments.

8.1 Hardware and software setup

In our experiments, we used Oracle 9.2 as the relational database to store the CDDB collection. We used a Dell Server P4600 with an Intel Xeon processor at 2GHz and 2GB of memory running on Microsoft Windows 2000. We implemented the middleware interface program in Java using Sun JDK 1.4. The program was run on an Intel Pentium II 333MHz machine with 128MB of memory running on Red Hat Linux 7.3. We conducted fifteen trial runs to ensure that network delays did not affect the results of our experiments.

8.2 CDDB database setup

The schema comprises two relations, *Disc(cd_id, cd_title, genre, num_of_tracks)* and *Track(cd_id, track_title)* (for simplicity, we omit other attributes of relations in the CDDB collection). Logically, each disc has one or more tracks. The *Disc* relation has 250K tuples. Each CD has an average of 10 tracks; these tracks are stored in the *Track* relation. Figures 12, 13 show some tuples in the two relations of CDDB.

Relation name: **Disc**

cd_id	cd_title	genre	num_of_tracks
a50b550c	Air Supply	Cddb/Misc	10
020e6d12	Eric Clapton	Cddb/Misc	13

Figure 12: Some tuples in the *Disc* relation in the CDDB database

Relation name: **Track**

cd_id	track_title
a50b550c	All out of Love
a50b550c	Lost in Love
020e6d12	Tears in Heaven

Figure 13: Some tuples in the *Track* relation in the CDDB database

Relation name: **Disc** (modified)

cd_id	cd_title	genre	count	num_of_tracks
a50b550c	Air Supply	Cddb/Misc	**0**	10
020e6d12	Eric Clapton	Cddb/Misc	**0**	13

Figure 14: Tuples in the *Disc* relation with the modified schema

To determine access patterns for the *Disc* relation, we add a new attribute, *count,* to the schema; this attribute holds an access count for each record in the *disc* relation. The rest of the database schema is unchanged (Section 5.2.1 explains how to choose relations for the schema change). Figure 14 shows tuples in the *Disc* relation from Figure 13, with the modified schema.

Relation name: **XmlDiscTrack**

cd_id	count	cd_title	mat XML
a50b550c	300	Air Supply	

Figure 15: Materialized Schema

<disc_tracks>
<id>a60bf90c</id>
<title>Air Supply</title>
<genre>cddb/misc</genre>
<track index = '1' offset='150'>All out of Love</track>
<track index = '2' offset='981'>Lost in Love</track>
</disc_tracks>

Figure 16: Materialized XML

Figure 15 shows the table *XmlDiscTrack*. This new relation holds materialized XML as variable length strings in record format. The process of defining this materialized table is

explained in Section 5.2.2. In the *XmlDiscTrack* relation the attributes *cd_id* and *count* remain same as the respective attributes in the *Disc* relation. The value of the *count* attribute in *XmlDiscTrack* equals the value of the *count* attribute in the corresponding tuple in the *Disc* relation, at the point in time when that tuple was selected in the materialization phase. The *matXML* attribute in *XMLDiscTrack* holds the materialized XML. For example, the value of the *matXML* attribute for the *Air Supply* disc is shown in Figure 16.

8.3 Experimental Results

This section discusses experiments performed using the algorithm discussed in Chapter 6. Using these results, we show the feasibility of our learning-based materialization approach in query optimization.

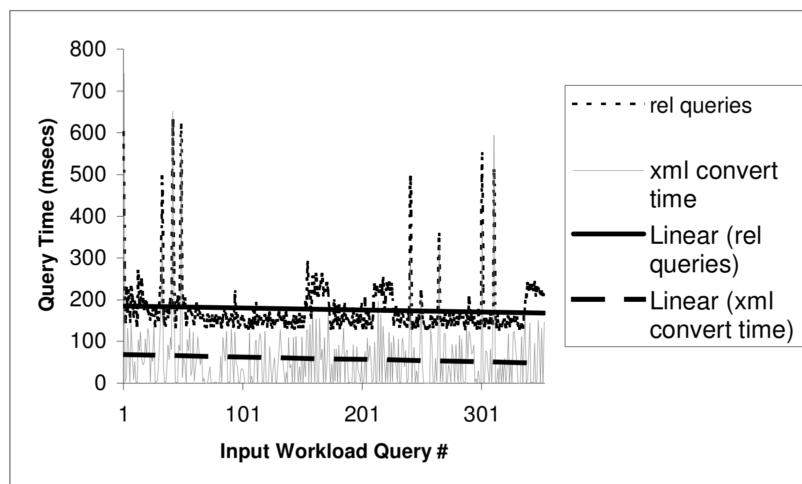

Figure 17: Time taken by the tagger routine to convert relational data to XML

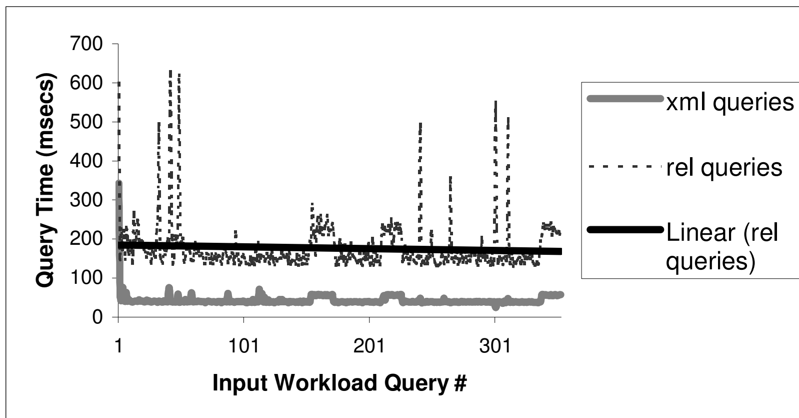

Figure 18: Runtime for queries on materialized XML data and relational data

8.3.1 Tagging-time analysis: Querying different data sources

- The objective of this experiment was to show the efficiency of querying materialized XML to that of querying with SQL counterparts on the underlying stored relational data.

The graph in Figure 17 shows the query execution times for 5000 XML cluster records. The relational tables hold 2.5 million tuples (250K CDs times 10 tracks). The cluster records are similar to the XML shown in Figure 6. These cluster records were selected after an initial round of learning (see Chapter 6). The graph is a plot of query execution times for XQuery queries based on the *cd_title* attribute of the *Disc* relation. The experiment shows that processing a query on a subset of the data in XML is faster than using SQL on the relational tables and then converting the retrieved data to XML. In this experiment, we observed that pushing XQuery queries to the relational data and converting the relational answers into an XML format is a major overhead.

8.3.2 Tagging-time analysis: Converting relational data to XML

- The goal was to analyze the maximum time spent in converting relational query answers to an XML format.

The graph in Figure 18 is a plot of the query execution times for SQL queries based on the *cd_title* attribute of the *Disc* relation.

The graph shows, as a solid line, the mean execution times for relational queries plus the times to convert the answers to the queries into XML. We see that of the total time of around 190 ms, the process of converting relational data into XML takes around 60ms (the dotted line in the figure). While the relational query takes 190 ms – 60 ms = 130 ms to execute, there is an overhead of 60 ms in converting the relational data to XML. These results are the motivation for using materialization techniques in relational databases with XML query interfaces.

8.3.3 Querying XML clusters

- The objective of this simulation run was to show the decrease in total query execution times when querying the materialized XML alongside stored relational data.

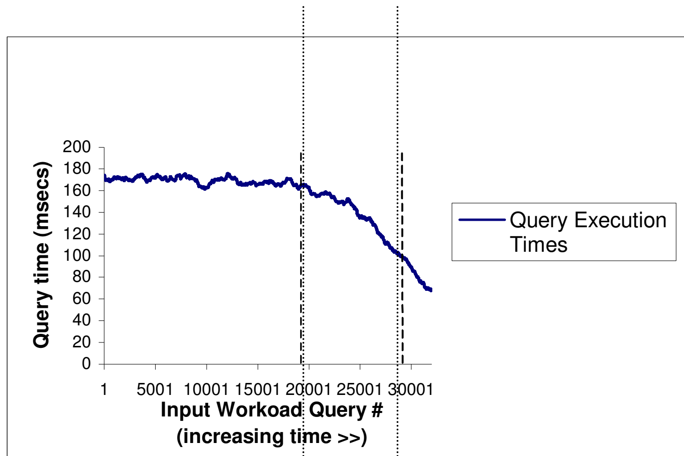

Figure 19: Query execution times for a large set of queries on XML clusters

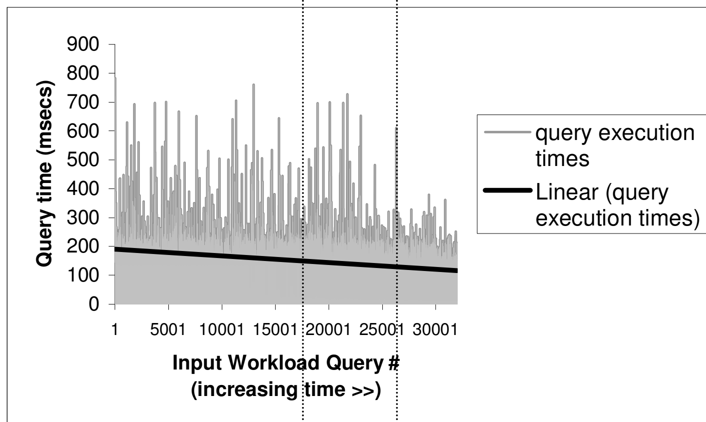

Figure 20: Query execution times for a mixture of queries

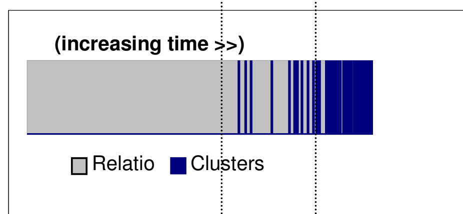

Figure 21: Distribution of data at the cluster and relational level for queries in increasing time domain

Figure 19 shows 30 simulation runs for a query workload of 1000 randomly selected CD titles. The X-axis shows the query sequence number (*query#*) for the input query in the query

workload, while the Y-axis shows the query execution times. The vertical dotted lines show the points at which XML materialization took place. (Recall that the system periodically runs the learning algorithm, see Section 6) It can be seen in Figure 19 that after every learning stage, the slope of the curve falls. Intuitively, after new learning has taken place, the XML clusters can satisfy a larger number of queries, with greater efficiency. Figure 20 shows a linear pattern that shows a fall in the slope of the query execution times for the set of 1000 queries repeated several times. The X-axis in all the figures denotes a linear progression of time as new workload queries are posed on the system.

Figure 21 shows the slow saturation (dark shading) of the materialized relation over time as the materialization process creates more clusters to be stored. We note that as the proportion of materialized clusters increases, the query response time to the workload queries decreases, because more and more workload queries are satisfied in the materialized clusters (these queries follow the solid line path shown in Figure 2 for the query subsystem in Section 5.1).

The dotted lines across Figures 19, 20 and 21 show the decrease in query times as more queries are satisfied by data found in clusters than in the relational database. For example at the end of 20,000 queries in Figures 19 and 20 there is virtually no decrease in query execution times, while at the end of around 28,000 queries we see there is a significant drop in the query execution times.

9. CONCLUSIONS AND FURTHER WORK

We have described a view- and learning-based solution to the problem of reducing total query-execution times in relational data sources with XML interfaces. Our approach combines learning techniques with selective materialization; our experiments show that these techniques can prove beneficial in improving query execution speeds in relatively static databases.

The proposed approach to improving query performance in relational databases with XML query interfaces is to store materialized XML views in a database using a learning algorithm. An alternative approach is to materialize the entire relational database in XML and then use a native XML engine to answer queries. This way, we could avoid the overhead of translating *all possible* queries on the data source into SQL, and of translating the relational answers to the queries into XML. However, query performance would degrade considerably as XML query-answering techniques are slower than relational query techniques (See Section 3.2). In addition, the system would incur a significant overhead in keeping the materialized XML clusters consistent with the underlying relations over time.

In Chapter 4.3, we described the process of grouping together related records in XML clusters. This approach allows a database system to incrementally find an optimal proportion of XML records that can be accessed faster than relational tables. This optimal proportion can be arrived at by varying the size of the clustered XML as well as the threshold value of the learning algorithm. Additional improvements can be made when user applications maintain local caches. It may be beneficial to prefetch XML data into the application's cache, so that future queries from the application have a higher chance of being satisfied locally.

In our approach, access counts (see Section 5.2.1 and Figure 14) in the modified schemas have to be updated frequently. An alternative would be to update tuple access counts offline periodically during low query loads.

We materialize only frequently accessed tuples; thus, only a fraction of the database (5%) is materialized as XML at any given time. (The clusters are recomputed from scratch every time the learning phase is invoked.) The advantage of the learning approach is to balance the proportion of data in relations and in XML clusters, by only materializing the tuples that are in the answers to multiple queries.

However, as materialized XML contains only a fraction of the data in the relational database, there may be queries that need to access both materialized XML data and relational data. We do not handle this case as part of our system.

9.1 Future work

One direction of future research is to automate schema definition for materialized XML clusters by using the information about past query workloads and the relations accessed by these workloads. Another enhancement would be to dynamically select the threshold value used in the materialization and clustering phases (Sections 6.2, 6.4). Better strategies for (1) prioritizing XML records within clusters, and (2) automatically dematerializing obsolete XML data are needed. Automating the choice of the relations of interest, given a query workload is an interesting topic to pursue. These tasks form interesting directions for future research.

10. BIBLIOGRAPHY

[1] L. Bellatreche, M. Schneider, M.K. Mohania, and B.K. Bhargava. PartJoin: An efficient storage and query execution for data warehouses. In Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2002), pages 296–306, 2002.

[2] J. Chen, S. Chen, and E.A. Rundensteiner. A transactional model for data warehouse maintenance. In Proceedings of the 21st International Conference on Conceptual Modeling (ER), 2002.

[3] L. Chen, E.A. Rundensteiner, and S. Wang. XCache: a semantic caching system for XML queries. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002.

[4] K.T. Claypool, E.A. Rundensteiner, X. Zhang, H. Su, H.A. Kuno, W.C. Lee, and G. Mitchell. Gangam — a solution to support multiple data models, their mappings and maintenance. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, 2001.

[5] L. Chen, S. Wang, E. Cash, B. Ryder, I. Hobbs, and E.A. Rundensteiner. A fine-grained replacement strategy for XML query cache. In Proceedings of the Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002), pages 76–83, 2002.

[6] J. Chen, X. Zhang, S. Chen, A. Koeller, and E.A. Rundensteiner. DyDa: Data warehouse maintenance in fully concurrent environments. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, 2001.

[7] H. Engstrom, S. Chakravarthy, and B. Lings. Implementation and comparative evaluation of maintenance policies in a data warehouse environment. In Proceedings of the 19th British National Conference on Databases, pages 90–102, 2002.

[8] D.W. Embley and W.Y. Mok. Developing XML Documents with Guaranteed "Good" Properties. In Proceedings of the 20th International Conference on Conceptual Modeling (ER), pages (426-441), 2001.

[9] eXcelon's eXtensible Information Server (XIS). http://www.exceloncorp.com/products/xis

[10] XML Extender Administration and Programming. http://www3.ibm.com/software/data/db2/extenders/xmlext/

[11] I.M.R.E. Filha, A.S. da Silva, A.H.F. Laender, and D.W. Embley. Using nested tables for representing and querying semistructured web data. In Proceedings of the Advanced Information Systems Engineering, 14th International Conference (CAiSE 2002), 2002.

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

[13] Yannis E. Ioannidis. Query optimization. In Allen B. Tucker, editor, The Computer Science and Engineering Handbook, pages 1038–1057. CRC Press, 1997.

[14] Z. Liu, F. Li, and W.K. Ng. Wiccap data model: Mapping physical websites to logical views. In Proceedings of the 21st International Conference on Conceptual Modeling (ER), 2002.

[15] Liu Mengchi. A logical foundation for XML. In Proceedings of the Advanced Information Systems Engineering, 14th International Conference (CAiSE 2002), pages 568–583, 2002.

[16] Tom M. Mitchell. Generalization as search. Artificial Intelligence, 18:203–226, 1982.

[17] Tom M. Mitchell. Learning and problem solving. In Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI 1983), pages 1139–1151, 1983.

[18] Liu Mengchi and Tok Wang Ling. Towards declarative XML querying. In Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE 2002), pages 127–138, 2002.

[19] Oracle XML SQL Utility (XSU). http://otn.oracle.com/tech/xml/oracle_xsu/content.html.

[20] K. Passi, L. Lane, S.K.Madria, B.C. Sakamuri, M.K. Mohania, and S.S. Bhowmick. A model for XML schema integration. In Proceedings of the 3rd International Conference on E-Commerce and Web Technologies (EC-Web 2002), 2002.

[21] Giuseppe Psaila. ERX: An experience in integrating entity-relationship models, relational databases, and XML technologies. In Proceedings of the XML-Based Data Management and Multimedia Engineering, EDBT Workshops 2002, 2002.

[22] Michael Rys. State-of-the-art XML support in RDBMS: Microsoft SQL server's XML features. IEEE Database Engineering Bulletin, 24(2), 2001.

[23] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In Proceedings of the 27th Int'l Conference on Very Large Data Bases (VLDB), pages 261–270, 2001.

[24] Tamino XML server. http://www.softwareag.com/tamino.

[25] Jennifer Widom. Research problems in data warehousing. In Proceedings of the Fourth International Conference on Information and Knowledge Management, pages 25–30, 1995.

[26] Extensible Markup Language (XML) http://www.w3.org/XML.

[27] X. Zhang, L. Ding, and E.A. Rundensteiner. Parallel multi-source view maintenance. VLDB Journal: Very Large DataBases, 2003. (To appear).

[28] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E.A. Rundensteiner. Rainbow: mapping-driven XQuery processing system. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002.

[29] Xin Zhang and Elke A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses using a cooperative framework. Information Systems, 27:219–243, 2002.

[30] The CDDB database. http://www.freedb.org.

[31] The World Wide Web Consortium. http://www.w3c.org

[32] Ashish Shah and Rada Chirkova: "Improving query performance using materialized XML views: A learning-based approach", *in the Proceedings of the First International Workshop on XML Schema and Data Management (XSDM'03)* (in conjunction with ER 2003)

[33] Arthur M. Keller, Julie Basu: A Predicate-based Caching Scheme for Client-Server Database Architectures. VLDB J. 5(1): 35-47(1996)

[34] *Database Systems: The Complete Book*, by Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, Prentice Hall, 2nd edition, ISBN 0-13-031995-3.