# Mapping, Indexing & Querying of MPEG-7 Descriptors in RDBMS with IXMDB

Yang Chu, Liang-Tien Chia and Sourav S. Bhowmick

*Center for Multimedia and Network Technology, School of Computer Engineering, Nanyang Technological University, Singapore 639798*

**Abstract**

MPEG-7 is a promising standard for the description of multimedia content. A number of applications based on MPEG-7 media descriptions have been set up for research, commercial and industrial applications. Therefore, an efficient storage solution for large amounts of MPEG-7 descriptions is certainly desirable. As a kind of data-centric XML documents, MPEG-7 descriptions can be stored in the relational DBMS for efficient and effective management. The approaches of storing XML data in relational DBMS can be classified into two classes of storage model: schema-conscious and schema-oblivious. The schema-conscious model, however, cannot support complex XPath-based queries efficiently and the schema-oblivious approach lacks the flexibility in typed representation and access. Although the leading database systems have provided functionality for the XML document management, none of them can reach all the critical requirements for the MPEG-7 descriptions management. In this paper, we present a new storage approach, called IXMDB, for MPEG-7 documents storage solution. IXMDB integrates the advantages of both the schema-conscious method and the schema-oblivious method, and avoids the main drawbacks from each method. The design of IXMDB pays attention to both multimedia information exchange and multimedia data manipulation. Its features can reach the most critical requirements for the MPEG-7 documents storage and management. The translation mechanism for converting XQuery to SQL and the support of query from multimedia perspective are provided with IXMDB. Performance studies are conducted by performing a set of queries from the XML perspective and from the multimedia perspective. The experimental results are presented in the paper and initial results are encouraging.

*Key words:* MPEG-7, relational DBMS, storing XML documents, IXMDB

*Email addresses:* pg00815938@ntu.edu.sg (Yang Chu), asltchia@ntu.edu.sg (Liang-Tien Chia), assourav@ntu.edu.sg (Sourav S. Bhowmick).

## 1 Introduction

MPEG-7[1] is a standard for describing the content of different types of multimedia data. As the first standard of the Moving Picture Experts Group to focus not on compression, but rather on metadata or descriptions for the multimedia content, it offers richer semantics as compared with other existing audiovisual metadata like Dublin Core[2] and TV-Anytime[3]. MPEG-7 documents can be defined and modified with the help of the Description Definition Language (DDL), which is based on XML Schema with extensions to support array, matrix and some temporal data types. Fig.1 shows an MPEG-7 description example.

MPEG-7 provides a comprehensive standardized tool set for the detailed description of audiovisual media. Descriptions for the catalogue level (e.g. title), the semantic level (who, what, when, where) and the structural level (spatiotemporal region, color histogram, timbre, texture) can be used as the base of all application domains making use of multimedia, such as classic multimedia archives, broadcast media selection, digital libraries, home entertainment, e-commerce, AI, etc.

With MPEG-7, multimedia content can be exchanged between heterogeneous systems; plain text files can be used to store and share multimedia information; and multimedia data will be readily available to most users. Thanks to these advantages, more and more applications are based on MPEG-7 de-

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Mpeg7 xmlns="http://www.mpeg7.org/2001/MPEG-7_Schema"
       xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
  <DescriptionUnit xsi:type="DescriptorCollectionType">
    <Descriptor size="7" xsi:type="DominantColorType">
      <ColorSpace type="HMMD" colorReferenceFlag="false"/>
      <ColorQuantization>
        <Component>H</Component>
        <NumOfBins>360</NumOfBins>
        <Component>Sum</Component>
        <NumOfBins>100</NumOfBins>
        <Component>Diff</Component>
        <NumOfBins>100</NumOfBins>
      </ColorQuantization>
      <SpatialCoherency>21</SpatialCoherency>
      <Values>
        <Percentage>4</Percentage>
        <ColorValueIndex>216  23  43 </ColorValueIndex>
        <ColorVariance>0  0  0 </ColorVariance>
      </Values>
      ... ...
      <Values>
        <Percentage>0</Percentage>
        <ColorValueIndex>55  35  9 </ColorValueIndex>
        <ColorVariance>1  0  1 </ColorVariance>
      </Values>
    </Descriptor>
  </DescriptionUnit>
</Mpeg7>
```

Fig. 1. An example of MPEG-7 document

scriptions, and the amount of MPEG-7 descriptions is inevitably increasing dramatically. Therefore, a critical requirement has arisen: how to develop an adequate database solution for the management of larger numbers of MPEG-7 descriptions.

As discussed in [4], a suitable MPEG-7 storage solution should satisfy several critical requirements: fine-grained storage, representation and access, typed representation and access, providing both classic one-dimensional index structures and multidimensional index structures for efficient access and query, and providing path indexing to navigate through the hierarchical structure of MPEG-7 documents and efficiently extract desired information. In addition to these requirements, an appropriate MPEG-7 management solution should emphasize other special issues. As pointed out in [4], one challenge of the management of MPEG-7 descriptions is how to make use of MPEG-7 schemas and fulfill the requirement of accessing and processing arrays and matrices within the MPEG-7 documents which make up low-level multimedia content. Another challenge is to provide an extensible high-dimension index structure to support efficient multimedia retrieval applications based on MPEG-7 media descriptions.

In order to provide an adequate storage solution for the management of MPEG-7 descriptions, in this paper we introduce a novel XML storage method known as IXMDB, abbreviated from "***I****ntegrated* ***X****ML-Enabled* ***M****PEG-7 De-scriptions* ***D****ata****b****ase*." [1] The motivation of IXMDB is to integrate the advantages of two main RDBMS-based XML storage approaches: *schema-conscious* approach and *schema-oblivious* approach. It offers adequate means to fulfill fine-grained and typed representation and access requirements for the MPEG-7 description storage. In addition to benefiting from the sophisticated index structures provided by RDBMS, our technique provides the path index structure for MPEG-7 documents navigation. The flexible storage schema defined by our technique makes it efficient to store and manipulate the special datatypes within MPEG-7 descriptions, such as *array, matrix, basicTimePoint* and *basicDuration*. Furthermore, the extensible high-dimensional index mechanism could be created on these array or matrix data to support multimedia content retrieval. Our MPEG-7 descriptions storage solution could be used as the back-end MPEG-7 data repository for all kinds of MPEG-7-based multimedia applications.

The remainder of the paper is organized as follows: Section 2 elaborates the limitations of existing XML storage approaches for the management of MPEG-7 descriptions, and further represents the motivation of our work. Section 3 summarizes our novel approach for the storage of MPEG-7 descriptions with RDBMS. Section 4 presents the database schema of our proposed approach:

[1] Our earlier related work can be found in [5].

IXMDB. We will describe how an MPEG-7 description is mapped into an RDBMS using IXMDB. In Section 5, we will explain the insertion and extraction algorithm for IXMDB. In Section 6, we present how to translate XML queries to SQL in IXMDB and the optimization technique to improve query performance. This is followed by a discussion of the support of multimedia content retrieval in IXMDB. The performance results of IXMDB and corresponding analysis are given in Section 7. We give an introduction to the existing RDBMS-based XML storage solutions and MPEG-7 descriptions management systems in Section 8. Finally, in Section 9 the conclusions are presented.

## 2 Motivation

Since MPEG-7 descriptions are also XML documents, the first consideration of the management of MPEG-7 descriptions is how to employ an XML document storage schema to fulfill the MPEG-7 descriptions storage requirements. There exist many XML storage solutions: Native XML database solutions [20–22], XML extensions of leading DBMS[25–28], and third-part middleware for RDBMS-based XML management.

To store XML documents efficiently and effectively in a relational database, there is a need to map the XML DTD/Schema to the database schema. The RDBMS-based XML storage solutions can be classified into two major categories according to their mapping schemas: *schema-conscious* approach and *schema-oblivious* approach. In *schema-conscious* approach, design of the database schema is based on the understanding of DTD or XML Schema. It defines a relation for each DTD subgraph and uses primary-key and foreign-key to describe the parent-child relationship between two elements. While in *schema-oblivious* approach, a fixed database schema is used to store the structure and the data of any XML documents without the assistance of document schema. The *schema-conscious* approach supports typed representation and access for XML data. It has better query performance than *schema-oblivious* approach since it partitions XML data based on DTD/XML Schema. While the *schema-oblivious* approach keeps the whole hierarchical structure information of an XML document. It will thus perform complex XPath-based query more efficiently and make it easier to re-construct the data back into XML format than *schema-conscious* approach.

Since there exist various XML storage solutions with different efficiency and functions [15–18,11,12,19,29], the most puzzling problem is which one is the most suitable choice for the MPEG-7 documents storage.

Native XML databases are designed especially for XML documents storage.

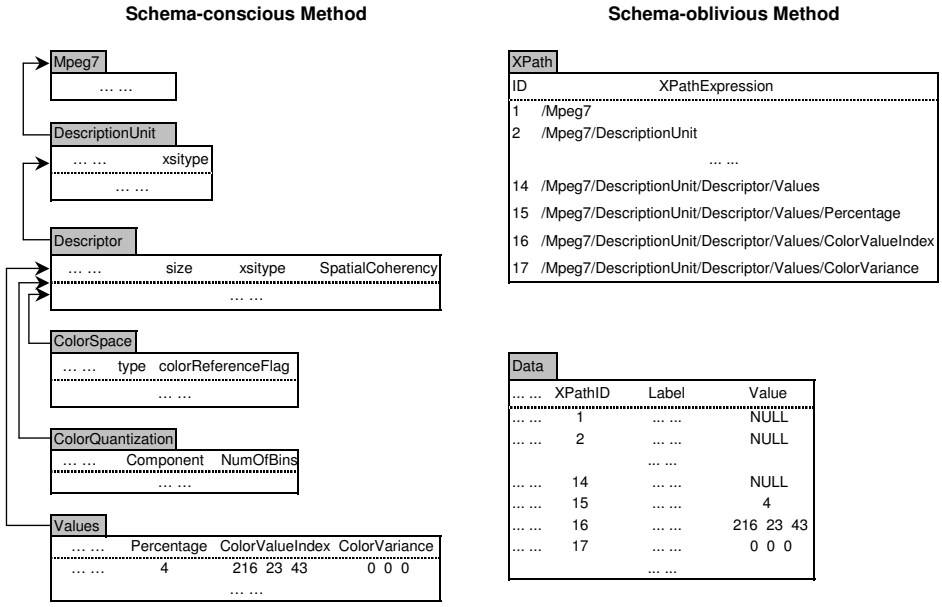



Fig. 2. Relational schemas for storing an MPEG-7 example with schema-conscious and schema-oblivious

Its fundamental logical storage unit is XML document, which is represented as text format. Traditional Native XML databases cannot support typed representations of the data within MPEG-7 documents since they represent the contents of an XML document as text. Recent research works on native XML database have proposed the powerful XML storage schemas to support appropriate access to non-textual data, e.g., Berkeley DB XML[23] and TDOM[24]. However, Berkeley DB XML is not extensible with index structures to support the index on individual items in the array/matrix datatype, and the multidimensional index on the multimedia data of which the MPEG-7 media description tools make heavy use. TDOM is designed to represent the basic contents of an XML document in a typed fashion and constitute a solid foundation for an XML database solution enabling the adequate management of MPEG-7 media descriptions [24]. However, as the other DOM-based native XML databases, it is costly to build in-memory trees of very large documents and then query those trees.

Furthermore, it is difficult for the Native XML database systems to create a flexible and extensible index structure on the data with various datatypes and query multimedia information across multiple MPEG-7 documents efficiently.

While for RDBMS-based XML storage solutions, no matter what approach we use, *schema-conscious* approach or *schema-oblivious* approach, we also cannot avoid their intrinsic drawbacks. Fig.2 shows the relational schema for storing an MPEG-7 example shown in Fig.1 with the basic idea of the *schema-conscious* approach and the *schema-oblivious* approach respectively. For the *schema-conscious* approach, it provides weak support for hierarchical structure

of the original XML documents. As shown in Fig.2, only the parent-child relationship between two elements can be reserved by creating the primary-foreign keys between corresponding tables, whereas the path expression from root element to an arbitrary element and the whole hierarchical structure information, including ancestor-descendant relationships, will be lost. This drawback makes it difficult to efficiently perform complex XPath-based queries. Furthermore, the process of re-constructing data from RDBMS into XML format may be expensive due to access to multiple tables and inflexible representations of structure information.

As shown in Fig.2, due to using a fixed table to store each element or attribute and the mapping process without the assistant of the DTD or the XML schema, the *schema-oblivious* approach has to establish only a single value column for storing the value of each element and attribute within XML documents as strings, the most generic type. Such a storage scheme would make it difficult to reflect all kinds of datatypes and then create an efficient index mechanism to speed up the queries with the conditions based on datatypes other than the string type.

All the leading database systems, such as IBM DB2, Microsoft SQL Server and Oracle, provide XML storage and management technology in their database products. However, as analyzed in [4], none of them can fulfill all the MPEG-7 descriptions storage requirements. Furthermore, although these XML-enabled databases support the queries of XML documents based on XPath, these XPath operations are evaluated by constructing DOM from CLOB and using functional evaluations. This can be very expensive when performing operations on large collections of documents.

In addition to the drawbacks mentioned above, none of the existing XML storage solutions address the problem of storage of the special datatypes introduced in the MPEG-7 DDL, such as *array, matrix, basicTimePoint* and *basicDuration*. One of the critical challenges for MPEG-7 descriptions management solution is to provide an extensible multidimensional index mechanism to support multimedia content retrieval. Unfortunately, the multidimensional access methods are rarely available in the most of current RDBMS-based XML storage solutions.

## 3 Overview of our approach

An MPEG-7 document can be viewed as an XML tree. In this tree structure, the internal node, the element type with element contents, represents the structure of document and can be viewed as the node that is only meaningful for document traversal. The leaf node, which is a single-valued attribute

or element type with text content, has little usage for XML tree navigation, as it is always a 'leaf' in the XML tree. So it can be viewed as the node that is only useful for holding value. IXMDB was designed to use *schema-oblivious* approach to store all the internal nodes and *schema-conscious* method to store all the leaf nodes. Since all the leaf nodes are stored with *schema-conscious* approach, the contents of MPEG-7 documents can be mapped into RDBMS with fine-grained manner and appropriate data types. As all the internal nodes are mapped with *schema-oblivious* approach, the complete hierarchical structure information of the original MPEG-7 documents can be kept in the database. Based on the fundamental scheme of IXMDB, we propose a method for storing complex datatypes within MPEG-7 descriptions with relational tables and integrate the GiST framework[10] for indexing high-dimensional data.

We carried out a set of experiments to investigate the storage efficiency and query performance of IXMDB. The amount of storage space IXMDB consumed is between that of the existing *schema-conscious* methods (e.g., *Shared-Inlining*) and *schema-oblivious* methods (e.g., *SUCXENT++*). The performance of mapping XML documents to RDBMS with IXMDB is similar to *SUCXENT++* and slightly slower than *Shared-Inlining*. IXMDB can reconstruct original XML documents from RDBMS up to two times faster than *Shared-Inlining*, and the same as *SUCXENT++*. For the XPath-based queries, IXMDB outperforms *SUCXENT++* by up to 12 times and *Shared-Inlining* by up to 20 times for most testing queries, including recursive queries and ordered XPath queries. Since IXMDB provides a special storage schema for the complex datatypes defined in MPEG-7 DDL, e.g. *basicTimePoint, basicDuration, array* and *matrix*, IXMDB outperforms all the other approaches for the queries on these datatypes. The reasons for the different performance between our technique and the existing approaches will be discussed in Section 7.

## 4 Relation Schema of IXMDB

An XML document is often represented as an XML tree. In an XML tree, the internal nodes correspond to the element types with element content in the XML document, while the leaf nodes correspond to the single-valued attributes and element types with PCDATA-only content in the XML document. In order to illustrate the tree structure of the XML documents and later introduce the storage scheme of IXMDB, an MPEG-7 document showed in Fig.1 is used as an example and its tree representation is shown in Fig.3.

The idea of IXMDB is to use *schema-oblivious* approach to map all the internal nodes and use *schema-conscious* approach to map all the leaf nodes. In an XML tree, the internal nodes depict the structure of the XML document

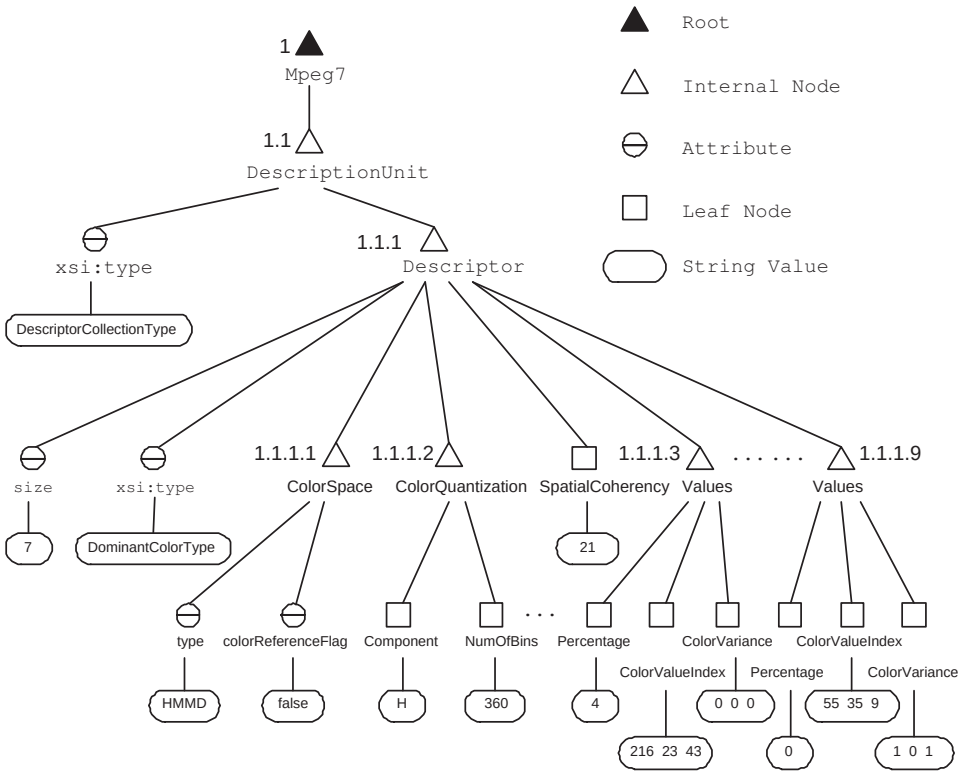



Fig. 3. Tree representation of the example in Fig.1

and are only useful for document navigation. Storing them by using *schema-oblivious* method, which can provide complete structure information of an XML document, can support efficient and easy document traversal. The leaf nodes hold all the data of XML document. They are the 'leaves' in the XML tree, so they have little usage for the XML tree navigation. They can be considered as the nodes only for storing the data of XML document. Using *schema-conscious* approach to store them can better represent appropriate datatype of each leaf node and provide a flexible storage scheme to satisfy different storage requests.

### *4.1 Internal node storage*

To speed up the processing of XML tree navigation, it is important to adopt an efficient numbering scheme to encode the nodes of a tree and quickly determine ancestor-descendant relationship between arbitrary two nodes in the XML tree based on such a numbering scheme. Thus, for RDBMS-based XML storage solutions, it is important to capture encoding information of each node into the relational data model. Note that, unlike most pure *schema-oblivious* methods that label all the nodes in an XML tree, our approach only needs to encode internal nodes.

Motivated by searching XML documents efficiently, several research efforts have addressed the problem of numbering scheme specification. In [6], the authors proposed three order encoding methods that can be used to represent XML order in the relational data model. These three methods are *Global Order, Local Order* and *Dewey Order*. Among them, as claimed by the authors, *Dewey Order* performs reasonably well on both queries and updates. With *Dewey Order*, each node is assigned an *id* value, a sequence of numeric values separated by a dot that represents the path from the document's root to the node. The root node is assigned a single numeric value. Child node *id* starts with the *id* of the parent node appended by a dot and the local order of the node, as illustrated in Fig.3.

With *Dewey Order*, the ancestor-descendant relationship can be determined using only the *id* value. However, the *id* length depends on the tree depth and a string comparison of the *ids* may degrade the query performance and deliver wrong results with respect to the total node order, e.g., comparing 1.9 and 1.10. In [7], the authors provided a solution for avoiding these shortcomings and proposed a novel hierarchical labeling scheme called *ORDPATH*.

*ORDPATH* provides a compressed binary representation of *Dewey Order*. It uses successive variable-length $L_i/O_i$ bitstrings to represent the *id* value of each node. Each $L_i$ bitstring, which are represented using a form of prefix-free encoding, specifies the length in bits of the succeeding $O_i$ bitstring. For example, if the $L_i$ bitstring 01 is assigned length 3, this $L_i$ will indicate a 3-bit $O_i$ bitstring. The bitstrings (000, 001, 010, ..., 111) can represent $O_i$ values of the first eight integers, (0, 1, 2, . . ., 7). Thus '01101' is the bitstring for ordinal '5' [7].

With *ORDPATH*, the *id* value of each node is constructed as binary string and document order can be preserved and yielded by simple bitstring comparison. The ancestor-descendent relationships between any two nodes $X$ and $Y$ can be determined equally simply: $X$ as a strict substring of $Y$ or vice versa implies there is an ancestry relationship.

IXMDB uses *ORDPATH* to encode the position of each internal node and construct the document structure information in the relational database model. Following are the relational schemas to store internal nodes:

xpath (xpathid, length, xpathexp)
internalnode (uid, xpathid, nodename, ordpath, parent, grdesc, lid, oid, tablename)

The semantics of the attributes in the above relations are as follows:

- The xpath table records the XPath information of the XML tree. xpathid and xpathexp represent the XPath identifier and the path expression. The number of edges for an XPath is recorded in the attribute length;
- The internalnode table represents the information of each internal node. uid

is used to identify each internal node. ordpath records the *ORDPATH* of this internal node. lid is the internal node local identifier, which depicts the position of a node among sibling nodes. oid is useful for the queries that include index predicates. It is an index of the node that occurs more than once in the XML document. tablename is used to indicate which table stores the value of this internal node's leaf nodes children; and

- The attributes parent and grdesc in the internalnode table are used for ancestor-descendant relationship determination between two internal nodes. In [7], the authors introduced two functions to determine the parent and an upper bound on all descendents of a given node. One is PARENT(ORDPATH X), which presents the parent of X, the other is GRDESC(ORDPATH X), which is the smallest ORDPATH-like value greater than any descendent of a node with ORDPATH X. We can use the user-defined functions (UDF) in RDBMS to implement these two functions. However, not all RDBMS support to create index on function, e.g., DB2. In order to benefit from index mechanism in RDBMS and improve query process, we introduce two columns, parent and grdesc, in the internalnode table to store the parent of corresponding internal node and the smallest value greater than any descendent of this node rather than using UDF.

### *4.2 Leaf node storage*

In order to identify the datatype of each leaf node and map them into database by using *schema-conscious* approach, a mapping schema is created to represent how to map them into database schema. The mapping schema is defined via mapping processing definition (MPD) file, which is also an XML file. The functionality of MPD file is like the DAD file in DB2 XML Extender, which provides a map of any XML data that is to be stored in the database. Currently, the MPD file is created manually. In the future work, we will develop an application to automatically generate a MPD file and a set of CREATE TABLE statements from a DTD or an XML schema. The end users can modify the MPD files according to their storage requirements. Fig.4 shows the DTD defined for MPD file and an example of MPD file for mapping the *DominantColor* descriptor.

IXMDB views an XML document as a tree of objects and then uses MPD file to map these objects to a relational database. In this view, an internal node is usually viewed as a class and mapped to a table. For example, as shown in Fig.4, the following declares the internal node *Descriptor* to be a class and maps it to the *dominantcolor* table:

```
<InternalNodeClass Name="Descriptor" ToTable="dominantcolor">
  ... ...
</InternalNodeClass>
```

The leaf nodes are usually viewed as properties and mapped to columns. For

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT MPD (InternalNodeClass+)>
<!ELEMENT InternalNodeClass (AttributeClass*,
        LeafNodeClass*,
        RepLeafClass*)>
<!ATTLIST InternalNodeClass
        Name CDATA #REQUIRED
        ToTable CDATA #IMPLIED
        RepLeafTable CDATA #IMPLIED
        ExtensionType CDATA #IMPLIED>
<!ELEMENT AttributeClass (AttributeType, ToColumn+)>
<!ELEMENT AttributeType EMPTY>
<!ATTLIST AttributeType
        Name CDATA #REQUIRED>
<!ELEMENT ToColumn EMPTY>
<!ATTLIST ToColumn
        Name CDATA #REQUIRED
        Datatype (char | varchar | smallint | integer
        | float | date | time | timestamp | timepoint
        | duration | array | matrix) #REQUIRED>
<!ELEMENT LeafNodeClass ((ElementType | PCDATA),
        ToColumn+)>
<!ELEMENT ElementType EMPTY>
<!ATTLIST ElementType
        Name CDATA #REQUIRED>
<!ELEMENT PCDATA EMPTY>
<!ELEMENT RepLeafClass EMPTY>
<!ATTLIST RepLeafClass
        Name CDATA #REQUIRED>
```

**DTD for MPD file**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MPD SYSTEM "mpd.dtd">
<MPD>
  <InternalNodeClass Name="Mpeg7" ToTable="mpeg7"/>
  <InternalNodeClass Name="DescriptionUnit"
            ToTable="descriptorcollection">
    <AttributeClass>
      <AttributeType Name="xsi:type"/>
      <ToColumn Name="xsitype" Datatype="varchar"/>
    </AttributeClass>
  </InternalNodeClass>
  <InternalNodeClass Name="Descriptor"
            ToTable="dominantcolor">
    <AttributeClass>
      <AttributeType Name="xsi:type"/>
      <ToColumn Name="xsitype" Datatype="varchar"/>
    </AttributeClass>
    <AttributeClass>
      <AttributeType Name="size"/>
      <ToColumn Name="size" Datatype="smallint"/>
    </AttributeClass>
    <LeafNodeClass>
      <ElementType Name="SpatialCoherency"/>
      <ToColumn Name="spatialcoherency"
                Datatype="smallint"/>
    </LeafNodeClass>
  </InternalNodeClass>

    … …

  <InternalNodeClass Name="Values" ToTable="values">
    <LeafNodeClass>
      <ElementType Name="Percentage"/>
      <ToColumn Name="percentage"
                Datatype="smallint"/>
    </LeafNodeClass>
    <LeafNodeClass>
      <ElementType Name="ColorValueIndex"/>
      <ToColumn Name="colorvalueindex"
                Datatype="array"/>
      <ToColumn Name="arrayid" Datatype="integer"/>
    </LeafNodeClass>
    <LeafNodeClass>
      <ElementType Name="ColorVariance"/>
      <ToColumn Name="colorvariance" Datatype="char"/>
    </LeafNodeClass>
  </InternalNodeClass>
</MPD>
```

**MPD for DominantColor Descriptor**

Fig. 4. DTD for MPD file and the MPD for the DominantColor descriptor

example, the following schema, which is nested inside the above mapping schema, declares the *xsi:type* and *size* attributes and the *SpatialCoherency* element to be properties and maps them to the *xsitype, size* and *spatialcoherency* columns respectively.

```
<AttributeClass>
  <AttributeType Name="xsi:type"/>
  <ToColumn Name="xsitype" Datatype="varchar"/>
</AttributeClass>
<AttributeClass>
  <AttributeType Name="size"/>
  <ToColumn Name="size" Datatype="smallint"/>
</AttributeClass>
<LeafNodeClass>
  <ElementType Name="SpatialCoherency"/>
  <ToColumn Name="spatialcoherency" Datatype="smallint"/>
</LeafNodeClass>
```

Our mapping schema for the leaf nodes is much simpler than the existing *schema-conscious* methods. The information needed to map a single internal node class only includes the table to which the internal node is mapped and the information of each property in this internal node class. It is not necessary to list its related internal node classes, such as its parent and child internal nodes. Note that, the ancestor-descendant relationships have been mapped to the relational model via *schema-oblivious* technique.

*For internal nodes:*

| XPATHID | LENGTH | XPATHEXP |
|---|---|---|
| 1 | 1 | #/Mpeg7 |
| 2 | 2 | #/Mpeg7#/DescriptionUnit |
| 3 | 3 | #/Mpeg7#/DescriptionUnit#/Descriptor |
| 4 | 4 | #/Mpeg7#/DescriptionUnit#/Descriptor#/ColorSpace |
| 5 | 4 | #/Mpeg7#/DescriptionUnit#/Descriptor#/ColorQuantization |
| 6 | 4 | #/Mpeg7#/DescriptionUnit#/Descriptor#/Values |

(a) xpath table

| UID | XPATHID | NODENAME | ORDPATH | PARENT | GRDESC | LID | OID | TABLENAME |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Mpeg7 | x'48 | | x'49 | 1 | 1 | |
| 2 | 2 | DescriptionUnit | x'4A40 | x'48 | x'4A41 | 1 | 1 | descriptorcollection |
| 3 | 3 | Descriptor | x'4A52 | x'4A40 | x'4A53 | 1 | 1 | dominantcolor |
| 4 | 4 | ColorSpace | x'4A5290 | x'4A52 | x'4A5291 | 1 | 1 | colorspace |
| 5 | 5 | ColorQuantization | x'4A52B0 | x'4A52 | x'4A52B1 | 2 | 1 | colorquantization |
| 6 | 6 | Values | x'4A52D0 | x'4A52 | x'4A52D1 | 4 | 1 | values |
| 7 | 6 | Values | x'4A52F0 | x'4A52 | x'4A52F1 | 5 | 2 | values |
| 8 | 6 | Values | x'4A5304 | x'4A52 | x'4A5305 | 6 | 3 | values |
| 9 | 6 | Values | x'4A530C | x'4A52 | x'4A530D | 7 | 4 | values |
| 10 | 6 | Values | x'4A5314 | x'4A52 | x'4A5315 | 8 | 5 | values |
| 11 | 6 | Values | x'4A531C | x'4A52 | x'4A531D | 9 | 6 | values |
| 12 | 6 | Values | x'4A5324 | x'4A52 | x'4A5325 | 10 | 7 | values |

(b) internalnode table

*For leaf nodes:*

| UID | XSITYPE |
|---|---|
| 2 | DescriptorCollectionType |

(c) descriptorcollection table

| UID | SIZE | XSITYPE | SPATIALCOHERENCY |
|---|---|---|---|
| 3 | 5 | DominantColorType | 0 |

(d) dominantcolor table

| UID | LID | OID | NODENAME | VALUE |
|---|---|---|---|---|
| 5 | 1 | 1 | Component | H |
| 5 | 2 | 1 | NumOfBins | 360 |
| 5 | 3 | 2 | Component | Sum |
| 5 | 4 | 2 | NumOfBins | 100 |
| 5 | 5 | 3 | Component | Diff |
| 5 | 6 | 3 | NumOfBins | 100 |

(e) colorquantization table

| UID | TYPE | COLORREFERENCEFLAG |
|---|---|---|
| 4 | HMMD | FALSE |

(f) colorspace table

| UID | PERCENTAGE | COLORVALUEINDEX | ARRAYID |
|---|---|---|---|
| 6 | 4 | 216 23 43 | 1 |
| 7 | 10 | 44 67 30 | 2 |
| 8 | 2 | 210 33 39 | 3 |
| 9 | 6 | 150 80 2 | 4 |
| 10 | 4 | 209 60 15 | 5 |
| 11 | 3 | 206 42 22 | 6 |
| 12 | 0 | 55 35 9 | 7 |

(g) values table

| ARRAYID | COL0 | COL1 | COL2 |
|---|---|---|---|
| 1 | 216 | 23 | 43 |
| 2 | 44 | 67 | 30 |
| 3 | 210 | 33 | 39 |
| 4 | 150 | 80 | 2 |
| 5 | 209 | 60 | 15 |
| 6 | 206 | 42 | 22 |
| 7 | 55 | 35 | 9 |

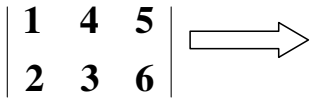
(h) array table

Fig. 5. IXMDB database schema for the example of MPEG-7 document in Fig.1

The database schema for the MPEG-7 document example shown in Fig.1 is shown in Fig.5.

### *4.3 Complex datatype representation*

As introduced previously, MPEG-7 descriptions not only use the standard datatypes, but also add the extension datatypes, including *array, matrix, basicTimePoint* and *basicDuration*. The appropriate MPEG-7 storage solution should fit for two application requirements: multimedia information exchange, and multimedia data manipulation. There is no problem for the storage of the




*Normalized schema:*

| MATRIXID | COLUMNID | ROWID | VALUE |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 4 |
| 1 | 2 | 0 | 5 |
| 1 | 0 | 1 | 2 |
| 1 | 1 | 1 | 3 |
| 1 | 2 | 1 | 6 |

*Semi-normalized schema:*

| MATRIXID | ROWID | COL0 | COL1 | COL2 |
|---|---|---|---|---|
| 1 | 0 | 1 | 4 | 5 |
| 1 | 1 | 2 | 3 | 6 |

*Denormalized schema:*

| MATRIXID | R0_C0 | R0_C1 | R0_C2 | R1_C0 | R1_C1 | R1_C2 |
|---|---|---|---|---|---|---|
| 1 | 1 | 4 | 5 | 2 | 3 | 6 |

Fig. 6. Three relational schemas for storing a matrix

data with standard datatypes with respect to these two requirements. However, it raises an efficiency problem that the storage schema for the above extension datatypes has to give attention to both multimedia information exchange and multimedia data manipulation. Undoubtedly, text-based format is the most efficient storage model for data exchange since it is not limited to any computer platforms and languages. Thus, storing the above special datatypes with text format in RDBMS enables the extraction of them from database without any additional operations and exchange of them between normally incompatible systems efficiently. However, such a storage schema makes it inefficient to manipulate these special datatypes due to the expensive process of character string parsing and datatype conversion. In this subsection, we will introduce how to store these complex datatypes in relational database and avoid the above efficiency problem.

***Array and Matrix***

In order to manipulate the array or matrix data efficiently, we can store individual cells in the array or matrix in a pure relational table with appropriate datatype. There are three relational schemas for array and matrix storage (Fig.6 illustrates these three schemas):

- Normalized schema - in this schema, each row will identify a cell in the array or matrix. The schema would be (ARRAYID, COLUMNID, VALUE) for array, or (MATRIXID, COLUMNID, ROWID, VALUE) for matrix;
- Semi-normalized schema - in this schema, each row will store a row in the matrix. The corresponding schema would be (MATRIXID, ROWID, COL0, COL1, ..., COLn) for $m \times n$ matrix; and
- Denormalized schema - in this schema, each row will record one array or matrix. It would be (ARRAYID, COL0, COL1, ..., COLn) for array, or (MATRIXID, R0_C0, R0_C1, ..., Rm_Cn) for $m \times n$ matrix.

These three storage schemas have different flexibility and efficiency for the

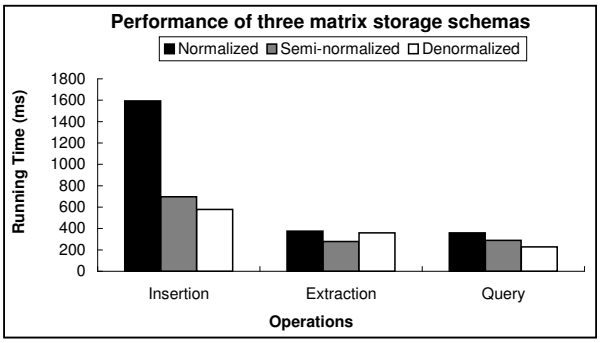



Fig. 7. Performance of three matrix storage schemas

operations of insertion, extraction and query. To evaluate the performance of these three array/matrix storage schemas, we performed a set of operations, including insertion, extraction and query (selecting an individual item), with 100 $21 \times 21$ matrices extracted from *SoundClassificationModel DS*. The experimental results are shown in Fig.7.

For the semi-normalized and denormalized schemas, the length of the array or matrix must be fixed, while the normalized schema is flexible enough to store the arrays or matrices of arbitrary cardinality. However, compared to the semi-normalized and denormalized schemas, the normalized schema needs to load many more tuples when performing insertion process, and need more joins to locate the desired individual items when performing queries. For array, the semi-normalized and denormalized schemas have the same storage schema; therefore, they have the same performance on insertion, extraction and query. For matrix, the denormalized schema has better performance on insertion and query than the semi-normalized schema because it inserts fewer tuples and needs fewer joins for the query process. With the semi-normalized schema, each row within a matrix will be still stored as a row in the database. Among these three schemas, it is the easiest for the semi-normalized schema to compose the data in the database to original matrix format. Thus, for extraction operation, the semi-normalized schema has the best performance, while the normalized schema performs the worst as it stores the array or matrix data fragmentally.

As a performance trade-off on the operations of insertion, extraction and query, the semi-normalized schema can be used to store the arrays or matrices with fixed cardinality. However, only normalized schema can be adopted to store the arrays or matrices with arbitrary length.

Even though the individual cell in the array or matrix could be stored in the special table with appropriate datatype, we still store the array or matrix as character string type in the table, since the character string type is still the most efficient format for data exchange. Although this storage schema requires more space, it speeds up the operations of reconstructing the data into XML format and exchanging required information between incompatible systems.

In addition, we define a column for storing *ARRAYID* or *MATRIXID*, which points to the corresponding record in the *ARRAY* or *MATRIX* table, which are defined to store array or matrix data. Referring to the right panel in Fig.4, we define two columns for the element *ColorValueIndex*, which is defined as an array type. One is for storing it as a character string type, and another is for storing the *ARRAYID* that points to the records in the *ARRAY* table. The corresponding example of database schema can be found in Fig.5.

Note that it is not necessary that the users use the above schema to store all the arrays and matrices in MPEG-7 descriptions. This schema is designed to speed up the operations on the individual items of an array/matrix. For many arrays or matrices, however, no one is interested in a certain individual item, e.g., the arrays in *ColorStructure* descriptor. The common operations on these arrays or matrices require all the items in them. The above schema cannot improve such operations since it needs more joins to get corresponding data and the index on individual items is meaningless for such operations. They can be only stored with character string datatype in the database.

The above storage schema, which stores the array/matrix data twice, may raise the problem of data consistency and integrity. This problem can occur during the updating of the array/matrix data. To solve this problem, we can define the triggers in the relational database to support the data consistency and integrity. If the array/matrix values stored as character string type are updated, the triggers can perform the actions to update the corresponding values stored in *ARRAY/MATRIX* table. In addition, one limitation of this storage schema is to fail to support the nested arrays/matrices. Fortunately, it is practically not very relevant to the storage of MPEG-7 descriptions.

***basicTimePoint and basicDuration***

The *basicTimePoint* datatype is used to describe a time point according to the Gregorian dates, day time and the time zone. It is represented in the following lexical format:

*YYYY-MM-DD*T*hh:mm:ss:nnn.ff*F*NNN*±*hh:mm* [8]

'T' is the delimiter for the time specification and 'F' stands for the number of fractions of one second. '±hh:mm' represents the time zone.

The *basicDuration* datatype is used to specify the interval of time according to days and time of day. The lexical format of this type is given by:

P*n*DT*n*H*n*M*n*S*n*N*nf*F*n*±*hh:mm*Z [8]

In this format, the separators specify the semantic of the number *n*: D (days), H (hours), M (minutes), S (seconds), N (number of fractions), f (for a decimal expression of fractions), F (number of fractions of one second) [8].

All the leading database systems support time point type, e.g., 'timestamp' datatype in DB2 and Oracle, and 'datetime' datatype in SQL Server. Although

***Example of timepoint and duration extracted from 'UserDescription' description:***

```
<MediaTime>
  <MediaTimePoint>2000-10-09T19:10:12</MediaTimePoint>
  <MediaDuration>PT1M45S</MediaDuration>
</MediaTime>
```

***Mapping scheme in MPD file:***

```
<NodeClass Name="MediaTime" ToTable="mediatime">
  <LeafNodeClass>
    <ElementType Name="MediaTimePoint" />
    <ToColumn Name="mediatimepoint" Datatype="timepoint" />
    <ToColumn Name="timestamp" Datatype="timestamp" />
  </LeafNodeClass>
<LeafNodeClass>
  <ElementType Name="MediaDuration" />
    <ToColumn Name="mediaduration" Datatype="duration" />
    <ToColumn Name="timestampduration" Datatype="varchar" />
  </LeafNodeClass>
</NodeClass>
```

***Database schema:***

| ... ... | MEDIATIMEPOINT | TIMESTAMP | MEDIADURATION | TIMESTAMPDURATION |
|---|---|---|---|---|
| ... ... | 2000-10-09T19:10:12 | 2000-10-09-19.10.12.000000 | PT1M45S | 145 |

Fig. 8. Storage scheme for basicTimePoint and basicDuration

the *basicTimePoint* datatype is based on ISO 8601, it is slightly different from the format accepted by the database system. We cannot directly insert this type into a relational table with built-in time point type in RDBMS without conversion. We also use two columns in relational table to store the data with *basicTimePoint* datatype to give attention to data exchange and manipulation. One is for storing *basicTimePoint* as a character string for efficient data exchange, and the other is defined as built-in time point datatype, e.g., timestamp in DB2, for time data operation. One of the most important operations on the time point data is time comparison. The original time point values in MPEG-7 descriptions are often represented with time zone. It is not efficient to directly compare the time point values with different time zones. During the mapping process, we first convert the time point data in MPEG-7 descriptions into a format acceptable to the database system, translate the time point data into local time, and store them into relational table with timestamp datatype; thereby utilizing the relational DB functionalities to directly and efficiently compare time point values.

The duration type may be involved in date and time arithmetic operations. All leading database systems support such operations as addition and subtraction. However, they differ in date arithmetic operations and the representation of duration operand. This article focuses on how to store *basicDuration* type in DB2. DB2 introduces four types of durations: labeled-duration, date duration, time duration and timestamp duration. Since the *basicTimePoint* type is stored as *timestamp* datatype, the timestamp duration type, which is ex-

pressed as a decimal number with precision 20 and scale 6, is the best mode to represent *basicDuration* type and then utilize the date and time arithmetic operations in DB2. For *basicDuration* type, we also use two columns, one for storage as character string, and the other for manipulating and recording as timestamp duration type.

Fig.8 uses an example extracted from a *UserDescription* description to illustrate the storage scheme for the *basicTimePoint* and *basicDuration* types. The date and time operations between *MediaTimePoint* and *MediaDuration* can be easily implemented with date and time functionalities provided by RDBMS. For example, if a user would like to get the result of adding *MediaDuration* to *MediaTimePoint*, the following simple SQL could be issued:

```
SELECT timestamp + decimal(timestampduration,20,6) FROM mediatime
```

### *4.4 Summary*

According to the above storage schema, all the leaf nodes within MPEG-7 descriptions are mapped in fine-grained manner to corresponding columns with appropriate datatype. For the complex datatypes defined by MPEG-7 DDL, such as *array, matrix, basicTimePoint* and *basicDuration*, a special storage schema is designed to store them in relational database. Although such a storage schema need more storage space and an additional mechanism to keep data consistency, it can speed up the special queries on these complex datatypes. It is somewhat similar to the data warehousing technique in RDBMS, which stores the operational data repeatedly and increases the complexity of storage process, but provides the powerful query capability. IXMDB can provide the most support for the fine-grained and typed representation and access of the contents of the MPEG-7 descriptions. With such storage schema, the MPEG-7 description content can be indexed easily by built-in database indexes. Since the path expressions of all the internal nodes are kept and each internal node is labelled by *ORDPATH*, the sufficient structure information of MPEG-7 documents can be stored in RDBMS.

## 5 Insertion and Extraction

### *5.1 Insertion Algorithm*

The algorithm for inserting MPEG-7 data into RDBMS is shown in the Fig.9. The input of the algorithm is the original MPEG-7 document. At the end of the algorithm, all the data and structure information of the MPEG-7 document would be stored in the relational database. The insertion proceeds as follows:

```
Input:  D - the MPEG-7 document to be mapped
Output: S - collection of 'insert' SQL statements

 1: interNodeRows is the collection of row data of each internal node
 2: leafNodeRows is the collection of row data of leaf nodes
 3: document = parse(D)
 4: root = document.getDocumentElement( )
 5: rootXPath = getXPath( root )
 6: rootORDPath = getORDPath( root )
 7: mappingNode( 1, root, rootXPath, rootORDPath, null )
 8: S.addInsertStatement( interNodeRows )
 9: S.addInsertStatement( leafNodeRows )
10: return S
```

(a) **Insertion Algorithm**

```
Input:  level, node, xpath, ordpath, parent_ordpath
Output: interNodeRows - collection of row data of internal nodes
        leafNodeRows - collection of row data of leaf nodes

 1: interNodeRow ← new InternalNodeRowClass(level, node, xpath, ordpath )
 2: interNodeRows.add( interNodeRow )
 3: leafNodeRow ← new RowClass( )
 4: attributes = node.getAttributes( )
 5: processAttributes( leafNodeRow, attributes )
 6: for ( childNode = node.getFirstChild();
 7:       childNode != null;
 8:       childNode = childNode.getNextSibling( ) ) do
 9:   if ( childNode is internal node ) then
10:     chileNode_xpath = getXPath( childNode )
11:     childNode_ordpath = getORDPath( childNode.parent_ordpath )
12:     mappingNode( level+1, childnode, childNode_xpath, childNode_ordpath, ordpath )
13:   else if ( childNode is leaf node ) then
14:     processLeafNode( leafNodeRow, childNode )
15:   end if
16: end for
17: leafNodeRows.add( leafNodeRow )
18: return interNodeRows, leafNodeRows
```

(b) **Procedure mappingNode**

Fig. 9. Insertion Algorithm

(1) The original MPEG-7 document is first parsed and the corresponding document tree is generated (line 3);
(2) Obtain the root element and corresponding *XPath* and *ORDPath* of root element (lines 4 to 6);
(3) Call the function *mappingNode( )* to process the root element (line 7);
(4) Generate 'insert' SQL statements (lines 8 to 9); and
(5) The function *mappingNode* is used to map each internal node which proceeds as follows (refer to Fig.9(b)):
   - generate one row data for this internal node and add it to internal node rows collection (lines 1 to 2);
   - define a variable *leafNodeRow* as an instance of *RowClass* class to record the row data of each leaf node that belongs to this internal node (line 3);
   - call function *processAttributes* to process attributes of this internal node with the aid of MPD file. After this process, the corresponding column information and the value with appropriate datatype of each attribute will be recorded into *leafNodeRow* (lines 4 to 5);
   - process all children of this internal node (lines 6 to 16). If the child

node is also an internal node, call the function *mappingNode* to map this node (lines 9 to 12). If the child node is a leaf node, call function *processLeafNode* to process it, including getting column name, converting datatype and adding corresponding row information to *leafNodeRow* (lines 13 to 14); and

- add *leafNodeRow* to leaf node rows collection (line 17).

### *5.2 Extraction Algorithm*

```
Input:  L1 {l1,l2,...lk} - list of internal nodes data ordered by
            UID, which is depth-first order of internal node
        L2 - list of all leaf nodes data
Output: D - an XML document
 1: n, p are instance of Class 'NodeClass' which including
        XML node and level information.
 2: s is a stack.
 3: for all elements in L1 do
 4:   if n = null then
 5:     n.node = D.createElement(li.nodeName)
 6:     n.level = li.level
 7:     D.appendChild(n.node)
 8:     setLeafNodeChildren(n.node, L2)
 9:     p = n
10:   else if
11:     n.node = D.createElement(li.nodeName)
12:     n.level = li.level
13:     if n.level > p.level then
14:       s.push(p)
15:     else if n.level = p.level then
16:       p = s.peek()
17:     else if
18:       while n.level <= p.level do
19:         s.pop()
20:         p = s.peek()
21:       end while
22:     end if
23:     setLeafNodeChildren(n.node, L2)
24:     p.node.appendChild(n.node)
25:     p = n
26:   end if
27: end for
28: return D
```

Fig. 10. Extraction algorithm

The extraction is the counter-procedure of insertion process. The first step is to extract data from the database, and then reconstruct them to revert to original XML format. The algorithm for reconstruction is presented in Fig.10. The extraction proceeds as follows:

(1) Extract all the internal nodes data and leaf nodes data from the database as the input parameters of reconstruction algorithm;
(2) The variables $n$ and $p$ are the instance of *NodeClass* class that records XML node and its level information. $n$ is for current node and $p$ is for the parent node of $n$ and the node processed just previously;
(3) For all the internal nodes, the root element is first processed (lines 4 to 9). Then, the rest internal nodes are treated one by one;
(4) For each internal node data, after creating corresponding XML node (line 11) and getting its level information (line 12), we can find its parent node information (lines 13 to 22). Then, add leaf nodes to the current internal node (line 23) and append the current internal node to the parent node as a child (line 24); and
(5) The function *setLeafNodeChildren* (line 8 and line 23) is used to insert the leaf nodes to the current internal node as its children. With the aid of MPD file, we can identify the element type of these leaf nodes, i.e. attribute or element type with PCDATA-only content, and then insert

them into current internal node with correct positions.

## 6 Querying MPEG-7

### *6.1 XPath-based query*

With IXMDB, applications can directly use SQL to access the contents of media descriptions in a fine-grained manner. However, the database schema of MPEG-7 storage solution is often not transparent to the users or applications. They prefer to access MPEG-7 media descriptions through some form of declarative XML query language, e.g., XPath and XQuery. In order to support the requirement of XML query language, it is necessary for the MPEG-7 description management solution to be powerful enough to provide appropriate translators from XPath and XQuery to SQL.

XQuery[9] is a query language built on XPath expressions to find and extract elements and attributes from XML documents. It has been broadly applicable across many types of XML data sources. XQuery offers iterative and transformative capabilities through FLWOR expressions, which stand for the five major clauses: *for, let, where, order by* and *return*. To support XQuery in IXMDB, we developed a translation mechanism for converting XQuery to SQL. This mechanism supports many features of XQuery, which include simple or recursive path expressions, predicate expressions (including order predicate and value comparison predicate), arithmetic expressions, comparison expressions and logical expressions[9]. However, due to the complexity of XQuery and the gaps between XQuery and SQL, it is difficult to translate all the features of XQuery into SQL. Fig.11 demonstrates the XQuery expression.

#### *6.1.1 Query Translation Algorithm*

**XQuery** :

```
for
  $b in doc('dominantcolor.xml')/Mpeg7
           /DescriptionUnit/Descriptor
where
  $b/Values/ColorValueIndex = '44  67  30'
  and $b/Values/Percentage > 5
return
  $b/SpatialCoherency
```

**SQL** :

```
select T1.spatialcoherency
from   xpath X1, xpath X2,
       internalnode I1, internalnode I2,
       descriptor T1, values T2
where  X1.xpathexp = '/Mpeg7/DescriptionUnit
                      /Descriptor'
       and X1.xpathid = I1.xpathid
       and X2.xpathexp = '/Mpeg7/DescriptionUnit
                      /Descriptor/Values'
       and X2.xpathid = I2.xpathid
       and I2.parent = I1.ordpath
       and T1.uid = I1.uid
       and T2.uid = I2.uid
       and T2.colorvalueindex = '44  67  30'
       and T2.percentage > 5
```

Fig. 11. An example of XQuery and corresponding SQL translated by IXMDB

There are several steps in our query translation process. First step is to use a parser generator to parse the XQuery and then generate corresponding parse tree. One example of a parser generator tool is JavaCC, the parser generator used with Java applications. The second step is to walk this parse tree and generate all the *PathExpr* and *ComparisonExpr* within the XQuery in question. In XQuery, *PathExpr* represents a path expression that can be used to locate nodes within XML tree, and *ComparisonExpr* represents a comparison expression that allows two values to be compared. For example, in the XQuery shown in Fig.11, the *PathExprs* and *ComparisonExprs* are listed in Fig.12. Finally, these *PathExprs* and *ComparisonExprs* can be translated into corresponding SQL component. Fig.13 shows the translation algorithm for IXMDB.

***PathExpr :***
//Descriptor
//Descriptor/Values/ColorValueIndex ($b/Values/ColorValueIndex)
//Descriptor/Values/Percentage ($b/Values/Percentage)
//Descriptor/SpatialCoherency ($b/SpatialCoherency)

***ComparisonExpr :***
$b/Values/ColorValueIndex = '44 67 30'
$b/Values/Percentage > 5

Fig. 12. PathExprs and ComparisonExprs of the XQuery in Fig.11

```
Input:  XQuery query X
Output: Translated SQL query S

 1: parse tree T = parse (X)
 2: E = walkParseTree(T)
 3: SelectClause s = S.getSelectClause()
 4: FromClause f = S.getFromClause()
 5: WhereClause w = S.getWhereClause()
 6: for all PathExpr p_i in E.allPathExpr do
 7:   processPathExpr(p_i,f,w)
 8:   if p_i included in return clause then
 9:     s.add("Ti." + p_i.getColumn())
10:   end if
11: end for
12: for all ComparisonExpr c_i in E.allComparisonExpr do
13:   processComparisonExpr(c_i,f,w)
14: end for
15: return S
```

Fig. 13. Translation algorithm

The translation proceeds as follows:

(1) Parse the XQuery and generate parse tree (line 1). Then, walk the parse tree and generate all the *PathExprs* and *ComparisonExprs* in XQuery (line 2);
(2) Process all the *PathExprs* (lines 6 to 11). Procedure *processPathExpr* will be called to generate corresponding 'from' and 'where' clause; and

```
Input:
   PathExpression p_i, FromClause f, WhereClause w
Output:
   FromClause f, WhereClause w

 1: f.add("xpath Xi, internalnode Ii")
 2: w.add("Xi.xpathexp = " + p_i.getPathExpr() + " and Xi.xpathid = Ii.xpathid")
 3: if the end node in p_i is leaf node then
 4:   table_i = p_i.getTable() /* get the table which stores this leaf node */
 5:   f.add(table_i + " Ti")
 6: end if
 7: for all FilterExpr f_j in p_i do
 8:   PathExpr p_ij0 = prefix pathexpr of f_j
 9:   if p_ij0.length = p_i.length then
10:     p_ij0 = p_i
11:   else
12:     f.add("xpath Xij0, internalnode Iij0")
13:     w.add("Xij0.xpathexp = " + p_ij0.getPathExpr() + " and Xij0.xpathid = Iij0.xpathid")
14:     addRelationship(p_ij0, p_i, w)
15:   end if
16:   if f_j is index predicate then  /* handle PathExpr like //Descriptor[6]/Values/… */
17:     if the end node in p_ij0 is leaf node then
18:       w.add("Ti.oid=" + f_j.getIndexPredicate())
19:     else   /* the end node in p_ij0 is internal node */
20:       w.add("Iij0.oid=" + f_j.getIndexPredicate())
21:     end if
22:   else if f_j is comparison express then
23:     if the first node in filter is leaf node then
24:       table_ij0 = p_ij0.getTable()
25:       column = column corresponding to this leaf node
26:       f.add(table_ij0 + " Tij0")
27:       w.add("Tij0." + column + f_j.getOperator() + f_j.getRightOperand())
28:     else  /* the first node in the filter is internal node */
29:       p_ij1 = full pathexpr of left operand of the comparison express in f_j
30:       table_ij1 = p_ij1.getTable()
31:       column = p_ij1.getColumn()
32:       f.add(table_ij1 + " Tij1")
33:       w.add("Tij1." + column + f_j.getOperator() + f_j.getRightOperand())
34:       addRelationship(p_ij0, p_ij1, w)
35:     end if
36:   end if
37: end for
38: return f,w
```

Fig. 14. Procedure *processPathExpr*

(3) Procedure *processComparisonExpr* is called to process all the *ComparisonExprs* (lines 12 to 14).

Procedure of *processPathExpr* is shown in Fig.14 and this procedure proceeds as follow:

(1) Since the path expressions are stored in 'xpath' table and the internal node information is stored in 'internalnode' table, we need to join the two tables to get the internal node information (lines 1 to 2);
(2) If the end node of this path expression is leaf node, add corresponding table name to *FROM* clause (lines 3 to 6);
(3) Lines 7 to 37 are for processing filter expressions in the path expression. First, parse each filter expression to extract prefix path expression and filter path expression, and then add corresponding *FROM* and *WHERE* clauses (lines 8 to 15). For example, in the following path expression:

    `//Descriptor[Values/Percentage='3']/SpatialCoherency`

    the prefix path expression is `//Descriptor`, the filter path expression is `//Descriptor/Values/Percentage`, and path expression `//Descriptor/`

`SpatialCoherency` could be called a destination path expression. In later processes, we may check the ancestor-descendant relationships among these three types of path expression to gain the final result; and

(4) Handle the conditions in filter expression (lines 16 to 36). There are two types of conditions to be handled: index predicate and comparison predicate.

- index predicate

  Consider the following example:

  ```
  //Descriptor[6]/Values/...
  ```

  In our storage schema, the column *oid* is introduced to store the ordinal information of each element. In the above example, if element *Descriptor* is a leaf node (repeatable leaf node), the following statement would be added in *WHERE* clause: *T.oid = 6*, where *T* is the alias of the table that stores this leaf node value (lines 17 to 18). In the tables that store the repeatable leaf nodes, we also introduce the column *oid* to record the corresponding ordinal information of each repeatable leaf node (see Fig.5 (e) colorquantization table). If element *Descriptor* is an internal node, the following statement would be added in *WHERE* clause: *I.oid = 6*, where *I* is the alias of corresponding *internalnode* table (lines 19 to 20); and

- comparison predicate

  If the first element of filter path expression is a leaf node, for example, the element *Percentage* in the path expression

  `//Descriptor/Values[Percentage='3']/...`,

  the following statement would be added in *WHERE* clause: *T.percentage = 3*, where *T* is the alias of the table that corresponds to the internal node *Values* (lines 23 to 27). If the first element of filter path expression is an internal node, for example, the element *Values* in the path expression

  `//Descriptor[Values/Percentage='3']/...`,

  after adding the statement *T.percentage = 3*, we need to add the statement to check the parent-child relationship between `//Descriptor` and `//Descriptor/Values` (lines 28 to 34). In subsection 3.1, we have discussed how to get parent-child or ancestor-descendant relationship between two nodes with *ORDPath*. The following statement would implement the above relationship evaluation: $I_1.ordpath = I_2.parent$, where $I_1$ is the alias of the *internalnode* table that corresponds to `//Descriptor` and $I_2$ corresponds to `//Descriptor/Values`.

Fig.15 shows the procedure of *processComparisonExpr*. This procedure follows:

(1) Obtain left and right operands and operator in the comparison expression (lines 1 to 3); and

(2) Two types of comparison expression need to be handled.

- In the first type, the right operand is literal, e.g.

```
Input:
   ComparisonExpr c_i, WhereClause w
Output:
   WhereClause w

 1: PathExpr l = c_i.getLeftOperand()
 2: operator = c_i.getOperator()
 3: r = c_i.getRightOperand()
 4: if r is Literal then
 5:   w.add(l.getTableAlias() + ".” + l.getColumn() + operator
          + r.getLiteralExpr())
 6: else if r is PathExpr then
 7:   w.add(l.getTableAlias() + ".” + l.getColumn() + operator
          + r. getTableAlias() + “.” + r.getColumn())
 8: end if
 9: return w
```

Fig. 15. Procedure *processComparisonExpr*

`//Descriptor/Values/Percentage='5'`.

The following SQL statement would be added in *WHERE* clause:

$T.percentage = 5$,

where $T$ is the alias of the table that corresponds to the internal node *Values* (lines 4 to 5); and

- In the second type, the right operand is also a path expression, e.g.

  `//FilteringAndSearchPreferences/PreferenceCondition/Place/Name = //BrowsingPreferences/PreferenceCondition/Place/Name`.

  The following SQL statement would be added in *WHERE* clause:

  $T_l.name = T_r.name$,

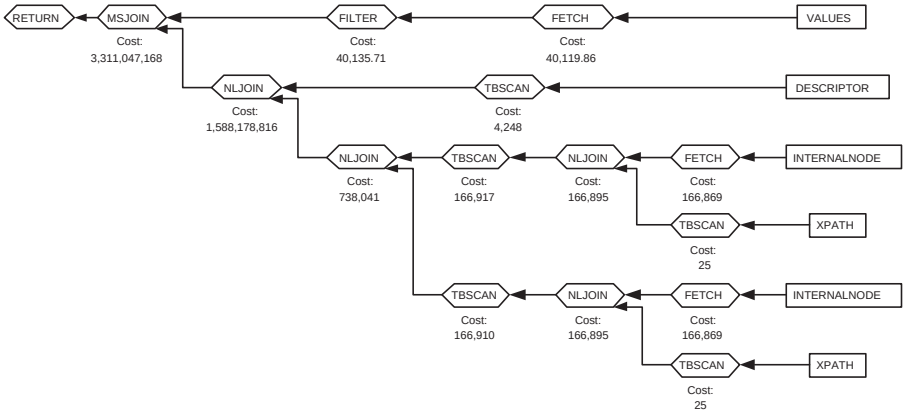
  where $T_l$ is the alias of the table that corresponds to the left path expression and $T_r$ corresponds to the right path expression (lines 6 to 7).

The SQL result for translating the XQuery example in Fig.11 according to the above translation algorithm is shown in the right panel in Fig.11.

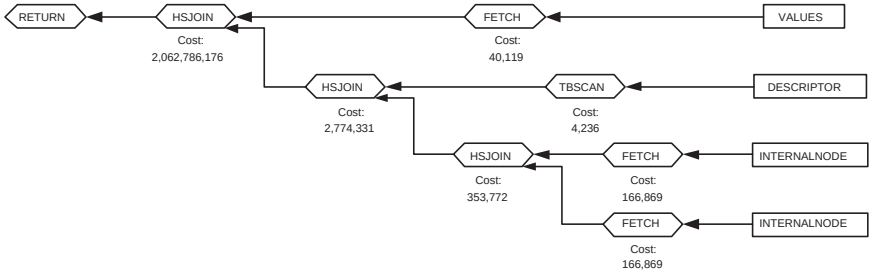
With the aid of the XPath expressions in XQuery and the position information of related nodes stored in the database, the return of results could be organized with XML format. The output algorithm is similar to the corresponding parts of extraction algorithm.

### *6.1.2 Query Rewriting*

To evaluate the performance of the above translation procedure, we captured information about the access plan of the above SQL statement. The captured information helps us understand how individual SQL statements are executed so that we can tune the statements. The access plan of the SQL statement in Fig.11 that is based on 1GB MPEG-7 dataset is shown in the Fig.16 (a). In this access plan graph, rectangles represent tables and diamonds represent operators. Operator is either an action that must be performed on data, or the output from a table or an index, when the access plan for an SQL statement




(a) Without optimization




(b) With optimization

Fig. 16. Access plan for SQL statement in Fig.11

is executed. The operators occurred in this graph are explained as follows:

**RETURN** – Represents the return of data from the query to the user;

**MSJOIN** – Represents a merge join, where both outer and inner tables must be in join-predicate order;

**NLJOIN** – Represents a nested loop join that accesses an inner table once for each row of the outer table;

**FILTER** – Filters data by applying one or more predicates to it;

**TBSCAN** – Retrieves rows by reading all required data directly from the data pages; and

**FETCH** – Fetches columns from a table using a specific record identifier.

The number under each operator indicates the total cost that is the estimated total resource usage necessary to execute corresponding operation. Cost is derived from a combination of CPU cost (in number of instructions) and I/O cost (in numbers of seeks and page transfers).

In our optimization study with access plan, we noticed that the joins between the *xpath* and *internalnode* tables consumed a considerable portion of the query processing time, and the cost of such joins cannot be decreased with the query optimizer of database system. The main effects of the joins between *xpath* and *internalnode* on the total query performance for the large dataset, is that large amounts of records in *internalnode* table increase the size of joins between *xpath* and *internalnode* tables dramatically.

In order to avoid these time-consuming joins, we re-wrote the queries to opti-

mize the query process. This process is similar to the optimization technique discussed in *SUCXENT++*[11]. In the query rewriting process, the join expression:

xpath.xpathexp = *xpath* and internalnode.xpathid = xpath.xpathid

would be placed with:

internalnode.xpathid = *n*

where *n* is the *xpathid* value corresponding to the path expression in the *xpath* table. Similarly, the following expression:

xpath.xpathexp like *xpath*% and internalnode.xpathid = xpath.xpathid

is replaced with:

internalnode.xpathid in (...)

where the value in the parentheses is the set of *xpathid* values corresponding to the path expression in the *xpath* table. The rewriting query process includes two steps: first, accessing *xpath* table and getting the *xpathid* value corresponding to the path expression; second, using these values to write the final SQL statement and then perform it to get the final result. The optimized access plan for SQL in Fig.11 is shown in Fig.16 (b).

Although such optimized query process need to access database multiple times, it avoids the joins between *xpath* and *internalnode* tables. When there exists a large amount of data in *internalnode* table, the query performance is improved dramatically (up to 10 times in our experiment).

### *6.2 Multimedia Content Retrieval*

To enhance our system to support extensible indexing mechanism, we integrated the GiST framework [10] into IXMDB. GiST provides a framework for building any kind of balanced index tree on multidimensional data. However, the GiST framework runs as its own process separately from the database system. In order to connect our RDBMS-based MPEG-7 storage solution to the GiST framework, we developed a set of user defined functions. With these database functions, the users can access the external multidimensional index system via simple SQL statements. The current GiST version is prepackaged with extensions for some spatial access methods, such as R-tree[32], R*-tree[33], SS-tree[34] and SR-tree[35]. To support the similarity searching function based on metric space, the M-tree[36], a representative of metric-based indices is added in IXMDB.

## 7 Experimental Results

In order to check the effectiveness of our method we have implemented IXMDB using JDK1.5 and carried out a series of performance experiments. In this

section, experimental results will be presented. First, we present the elapsed time for insertion and extraction executions and storage space requirements of IXMDB. We compare it with SUCXENT++[11], which has been proven to outperform the other existing *schema-oblivious* approaches, and Shared-Inlining[12], which is the representative of existing *schema-conscious* methods. Next, we compare the query performances of IXMDB to these approaches. To test how our system supports the queries from multimedia perspective efficiently, we performed a set of experiments to evaluate the efficiency of the multidimensional index system. The hardware platform used is a Dell PowerEdge 2650 with Xeon CPU 2.8GHz and 1.00GB RAM running Windows Server 2003 Enterprise Edition. The data base system is IBM DB2 Universal Database Enterprise Server Edition V8.1. The application has been developed and the running environment is Java JDK 1.5.

### *7.1 Data Set*

| Dataset | No of Internal Nodes | No of Leaf Nodes | No of Attributes | No of Nodes | Size (MB) | Max Depth |
|---|---|---|---|---|---|---|
| BENCH001 | 7802 | 9330 | 3919 | 21051 | 1.1 | 12 |
| BENCH01 | 76749 | 91116 | 38265 | 206130 | 11.3 | 12 |
| BENCH | 762838 | 903477 | 381878 | 2048193 | 113.0 | 12 |
| MPEG-7 | 5013953 | 10158454 | 3819292 | 18991699 | 1030.0 | 11 |

Fig. 17. Characteristics of the experimental data sets

We used two experimental data sets. One is a synthetic dataset, which is from the XMark project[13], a benchmark for XML data management. The other is a real dataset, which consists of MPEG-7 descriptions. These two data sets were used for comparison of storage size, insertion and extraction times. Since MPEG-7 descriptions are also XML documents, in order to test the effectiveness of IXMDB from XML perspective, we used XMark benchmark data as one of experimental data sets. We generated the XMark benchmark data with different scale factors. Three different sizes of data are used: BENCH001 (which means 1% of the original BENCH) with 1.1MB size, BENCH01 with 11.3MB size and BENCH with 113MB size. The MPEG-7 descriptions data set with the size of 1GB includes four Description Schemes: *UserDescription DS*, *Object DS* (including *SpatialMask D*), *StateTransitionModel DS* and *SoundClassificationModel DS*, and eight low-level descriptors extracted from about 240,000 pictures: *ColorLayout, ColorStructure, ContourShape, DominantColor, EdgeHistogram, HomogeneousTexture, RegionShape* and *ScalableColor*. Fig.17 summarizes the characteristics of the experimental data sets.

Test queries need to be carefully selected for the performance study. XMark issues 20 benchmark queries that cover different aspects of XML queries for accessing XML data. [2] We also issued twelve common queries to test query

[2] These queries could be found on the following web site: www.xml-benchmark.org

**MPEG-7**

- MQ1. Find whether user 'John Doe' requests that his identity is revealed to third parties or not.
- MQ2. List the collected actions' type for user 'John Doe' in his usage history.
- MQ3. Return the user name who conducted 'Record' action over a period of six hours on the evening of October 10,2000.
- MQ4. Return the end time point of the user John Doe's observation that started at 18:00, October 10,2000.
- MQ5. Return the DominantColor descriptor information of the sixth image.
- MQ6. Return the value of ColorValueIndex with the maximum Percentage value for the 88th image.
- MQ7. Return the ScalableColor descriptor information of the sixth image.
- MQ8. Return the images that have red dominant color with more than 30 Percentage value.
- MQ9. Return the local-edge distribution information (edge histogram descriptor) of the images that have blue dominant color with more than 20 Percentage value.
- MQ10. Return the object name which locates at the right of point (10,10) in the image which media URL is 'image0.jpg'.
- MQ11. Return the transition probability from 'Pass' to 'Shot on goal'.
- MQ12. Return the largest transition probability among the states and the labels of corresponding two states in the sound model which ID is 'ID3'.

Fig. 18. Queries

performance on MPEG-7 documents. These queries are shown in Fig.18. In our performance study, we labelled the XMark queries as Q1-Q20, and the MPEG-7 queries as MQ1-MQ12. The corresponding SQLs with IXMDB for these queries are shown in Appendix A. To evaluate the effectiveness of the multidimensional index system, we performed similarity search on four low-level descriptors with different dimensions.

### *7.2 Insertion Performance and Storage Size*

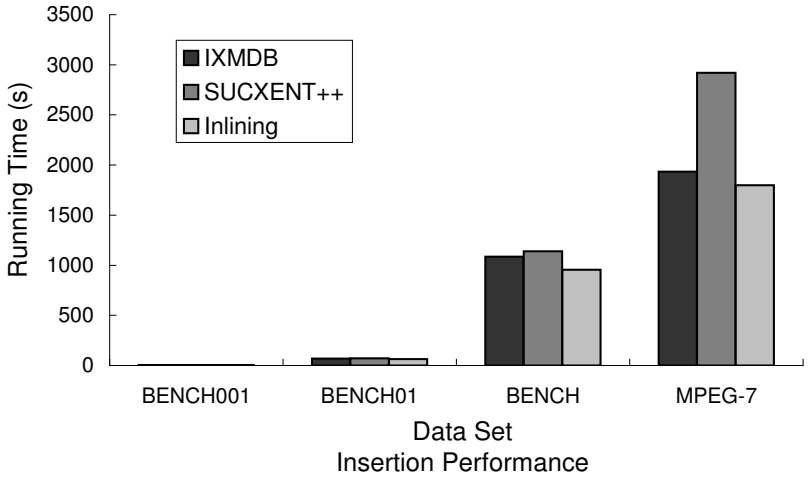



Fig. 19. Experimental Result: Insertion Performance

Fig. 19 presents the insertion performance (including index creation during insertion process) for different sample data set. This figure shows that the Shared-Inlining is the best, while IXMDB and SUCXENT++ have similar performance. This is because these methods store different amounts of data during insertion process.

Shared-Inlining method creates tables for the internal nodes, and their leaf node children, and some of leaf node descendants may be inlined into these

| Approach | Dataset | No of Tuples | Table size (MB) | Index size (MB) |
|---|---|---|---|---|
| IXMDB | BENCH001 | 18498 | 1.6 | 0.6 |
| | BENCH01 | 183326 | 15.4 | 6.6 |
| | BENCH | 1814296 | 149.2 | 60.8 |
| | MPEG-7 | 9411958 | 910.3 | 420.5 |
| SUCXENT | BENCH001 | 13946 | 1.3 | 0.8 |
| | BENCH01 | 136262 | 12.4 | 9.1 |
| | BENCH | 1339355 | 120.5 | 113.3 |
| | MPEG-7 | 14409746 | 1126.9 | 814.8 |
| Shared-Inlining | BENCH001 | 7971 | 0.6 | 0.3 |
| | BENCH01 | 78126 | 5.8 | 3.6 |
| | BENCH | 772838 | 56.8 | 32.5 |
| | MPEG-7 | 5089703 | 660.7 | 129.5 |

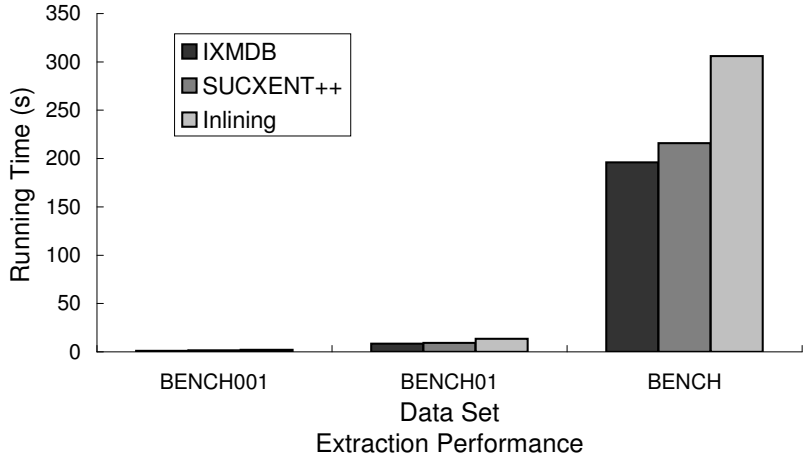
Fig. 20. Storage size

tables. Therefore, the number of inserted tuples with Shared-Inlining is close to the number of internal nodes. SUCXENT++ only stores leaf nodes, thus the number of its inserted tuples is equal to the number of leaf nodes. For IXMDB, all internal nodes need to be inserted; and for each internal node that has attributes or leaf nodes, one tuple would be inserted to store its leaf node children. Therefore, the number of inserted tuples with IXMDB is close to double of the number of internal nodes. Compared to SUCXENT++, Shared-Inlining and IXMDB have a disadvantage for insertion process. That is, they need to insert data into more tables than SUCXENT++ does. However, such disadvantages can be compensated by the smaller size of inserted tuples and less elapsed time for index creation with Shared-Inlining and IXMDB. Since SUCXENT++ stores the data in several fixed tables, inserting the value of each leaf node would require additional storage process, i.e., storing additional information into the columns other than the column that records the value of leaf node in the table. While for Shared-Inlining and IXMDB, the value of each leaf node is stored in corresponding column and do not raise additional storage requirement. Furthermore, SUCXENT++ needs to create the index on all the values with character string type, while IXMDB and Shared-Inlining generate the indexes on the required columns with different datatypes. Thus, IXMDB and Shared-Inlining take less time to create indexes. In summary, these methods have slight difference in insertion performance due to their intrinsic advantage and disadvantage in insertion process.

Fig.20 shows their different storage requirements. The above discussions can also explain why SUCXENT++ needs more storage space to store the values of leaf nodes and consumes more storage size for indexes.

### *7.3 Extraction Performance*

Fig. 21 shows the extraction performance of these approaches. Extraction is the reverse operation of insertion process. That means extraction operation




| | BENCH001 | | BENCH01 | | BENCH | |
|---|---|---|---|---|---|---|
| | Extraction | Construction | Extraction | Construction | Extraction | Construction |
| **IXMDB** | 0.6 | 0.4 | 5.8 | 2.8 | 141 | 55 |
| **SUCXENT++** | 0.4 | 1.1 | 2.6 | 6.8 | 70 | 146 |
| **Inlining** | 1.5 | 0.6 | 10.5 | 3.1 | 240 | 66 |

**Running time in terms of two steps of extraction process**

Fig. 21. Experimental Result: Extraction Performance

is the process to extract the data from database and reconstruct them with original XML format. The extraction time is made up of the time taken to extract the relevant data from database and main memory processing time to reconstruct the data into XML document. Fig.21 also shows the running time of these methods in terms of the two steps of extraction process. Based on this figure, we observe the followings: there are slight difference between IXMDB and SUCXENT++, and they are about 50% faster than Shared-Inlining method in terms of extraction performance.

In the process of extracting relevant data from database, the performance of SUCXENT++ is the best since it only retrieves the data of leaf nodes. IXMDB needs to retrieve all internal nodes and corresponding leaf node children. Note that it is not necessary to join *internalnode* table and each leaf nodes table to get the leaf nodes data, since with the aid of MPD file and the given document identifier, it is easy to extract all the leaf nodes value from database without any internal nodes information.

However, for the process of reconstructing the extracted data into XML format, SUCXENT++ performs the worst due to its storage schema. SUCXENT++ only stores the path expressions of leaf nodes and there is a lack of the information about each internal node. When SUCXENT++ creates the XML document tree, it needs to first parse the path expression of each leaf node, gain the internal nodes within this path expression, and then decide the appropriate positions of these internal nodes and leaf nodes in the document tree. Therefore, with SUCXENT++, although the performance of extract-

ing data from database is better, the time taken for reconstruction is more. IXMDB consumes the least memory processing time to reconstruct XML document becuase it creates the document tree by only organizing the internal nodes. With the aid of an MPD file the corresponding leaf nodes of each internal node can be efficiently placed to the appropriate position in the document tree.

Although Shared-Inlining returns the smallest number of tuples among these approaches when extracting data from database, the performance is worse than the others because Shared-Inlining uses primary key and foreign key to represent the parent-child relationship between two nodes. Such relationship information needs to be gained to generate XML document tree. This means a large amount of join queries need to be executed to extract data. The extraction process would become more expensive when there exist many relations for storing XML documents.

## *7.4 Query Performance*

### *7.4.1 Queries from the XML perspective*

Fig.22 represents the query performance of each approach for different queries introduced in the beginning of this section. This figure shows that IXMDB has encouraging query performance for most of the queries. Fig.23 summarizes the number of queries that each method performed best or worst, and also shows the query performance statistic, including the minimum, maximum, average, and standard deviation of the query performance for all the test queries. According to Fig.23(a), we can observe that IXMDB primely converges the advantages of *schema-conscious* approach and *schema-oblivious* approach (having the largest number of queries with the best performance), and avoids their intrinsic disadvantages (no queries with the worst performance). We discuss the observations in detail as follows.

***IXMDB vs. SUCXENT++***

The common feature of the above queries is to apply predicates related to several sub-elements. In general, for such queries, the *schema-conscious* approach outperforms the *schema-oblivious* approach due to the reason described in [14]. *Schema-conscious* approach clusters elements corresponding to the same real world object while *schema-oblivious* approach loses such benefit and it has to issue more SQL joins to capture the parent-children or ancestor-descendant relationships between XML elements. Consider the following XQuery:

```
//Descriptor/Values[Percentage>5]/ColorValueIndex
```

With *schema-conscious* approach, elements *Percentage* and *ColorValueIndex*, which are children of element *Values*, would be clustered as two attributes of

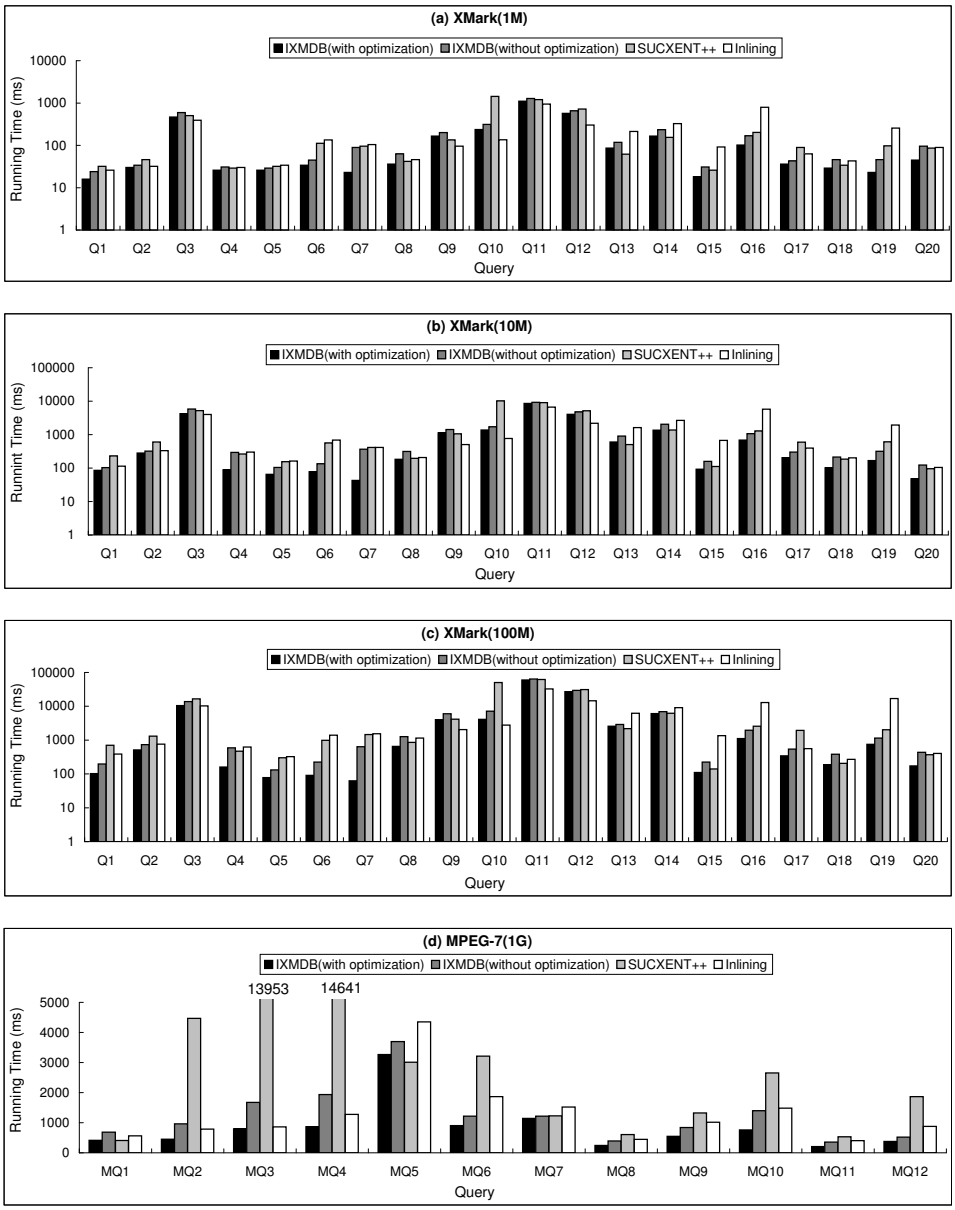



Fig. 22. Experimental Result: Query Performance from the XML perspective

one relation. The above XQuery can be translated into a simple SQL:

```
select colorvalueindex from values where percentage>5
```

where table *values* corresponds to the element *Values* and columns *percentage* and *colorvalueindex* correspond to the elements *Percentage* and *ColorValueIndex* respectively. While with *schema-oblivious* approach, these two elements cannot be clustered and would be stored in separate rows. *Schema-oblivious* approach needs more joins to check sibling relationship between elements *Percentage* and *ColorValueIndex*. Fig.24 shows another example, the translated SQL with SUCXENT++ for the XQuery shown in Fig. 11. According to its storage schema, IXMDB can also benefit from the above advantage of *schema-conscious* approach. Thus, SUCXENT++ needs more joins to check sibling

| Approach | XMark | | MPEG-7 | |
|---|---|---|---|---|
| | Best | Worst | Best | Worst |
| IXMDB | 13 | 0 | 8 | 0 |
| SUCXENT++ | 2 | 7 | 2 | 7 |
| Inlining | 5 | 13 | 0 | 3 |

(a) Query performance summary

| | Min | Max | Ave. | S.D. |
|---|---|---|---|---|
| MQ1 | 412 | 562 | 463 | 85 |
| MQ2 | 453 | 4466 | 1901 | 2227 |
| MQ3 | 797 | 13953 | 5203 | 7577 |
| MQ4 | 866 | 14641 | 5594 | 7837 |
| MQ5 | 3012 | 4352 | 3543 | 711 |
| MQ6 | 906 | 3215 | 1995 | 1160 |
| MQ7 | 1144 | 1521 | 1298 | 197 |
| MQ8 | 245 | 605 | 432 | 180 |
| MQ9 | 547 | 1325 | 962 | 391 |
| MQ10 | 760 | 2652 | 1632 | 954 |
| MQ11 | 210 | 530 | 382 | 161 |
| MQ12 | 378 | 1865 | 1039 | 756 |

(b) Query performance statistics on MPEG-7

| | Min | Max | Ave. | S.D. |
|---|---|---|---|---|
| Q1 | 102 | 703 | 397 | 300 |
| Q2 | 515 | 1312 | 861 | 408 |
| Q3 | 10254 | 16523 | 12441 | 3537 |
| Q4 | 160 | 625 | 418 | 236 |
| Q5 | 78 | 325 | 234 | 136 |
| Q6 | 91 | 1394 | 823 | 666 |
| Q7 | 63 | 1550 | 1024 | 833 |
| Q8 | 656 | 1147 | 886 | 246 |
| Q9 | 2037 | 4150 | 3406 | 1187 |
| Q10 | 2775 | 50437 | 19111 | 27137 |
| Q11 | 32703 | 61721 | 51418 | 16235 |
| Q12 | 14563 | 31253 | 24280 | 8676 |
| Q13 | 2181 | 6234 | 3659 | 2238 |
| Q14 | 6105 | 9136 | 7164 | 1709 |
| Q15 | 110 | 1360 | 537 | 712 |
| Q16 | 1116 | 12946 | 5540 | 6453 |
| Q17 | 343 | 1932 | 945 | 861 |
| Q18 | 188 | 271 | 221 | 43 |
| Q19 | 750 | 16856 | 6540 | 8955 |
| Q20 | 172 | 403 | 315 | 125 |

(c) Query performance statistics on XMark

Fig. 23. Query performance summary and statistic

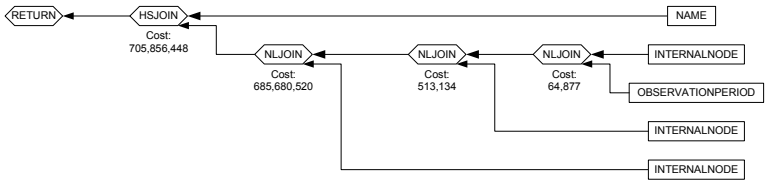
```
SELECT      v3.LeafValue
FROM        Path p1, Path p2, Path p3,
            PathValue v1, PathValue v2, PathValue v3,
            DocumentRValue r1, DocumentRValue r2
WHERE       p1.PathExp = '/Mpeg7/DescriptionUnit/Descriptor/Values/ColorValueIndex'
            and p2.PathExp = '/Mpeg7/DescriptionUnit/Descriptor/Values/Percentage'
            and p3.PathExp = '/Mpeg7/DescriptionUnit/Descriptor/SpatialCoherency'
            and v1.PathId = p1.PathId
            and v2.PathId = p2.PathId
            and v3.PathId = p3.PathId
            and v1.LeafValue = '44  67  30'
            and cast(v2.LeafValue as integer) > 5
            and r1.Level = 4
            and abs(v1.BranchOrderSum - v2.BranchOrderSum) < r1.RValue
            and r2.Level = 3
            and abs(v1.BranchOrderSum - v3.BranchOrderSum) < r2.RValue
```

Fig. 24. Translated SQL with SUCXENT++

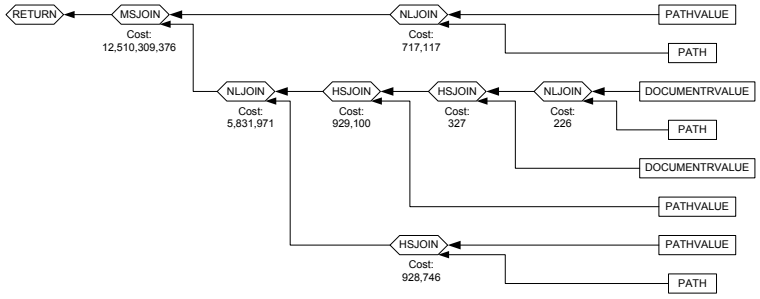
relationships in comparison with the translated SQL with IXMDB shown in Fig.11.

*Schema-conscious* approach and IXMDB store the leaf nodes value in many different tables, while *schema-oblivious* approach, e.g. SUCXENT++, stores all leaf nodes value within a single table. Therefore, *schema-conscious* approach and IXMDB require the join of many smaller tables when performing queries, whereas *schema-oblivious* approach needs to self-join a single large table. The performance of *schema-oblivious* approach is substantially degraded when large collections of XML documents exist. Fig.25 shows the access plan for MQ3 with IXMDB and SUCXENT++ respectively. According to this figure, we can observe that SUCXENT++ requires more joins and performs this query at a much higher cost.

The above analysis explains why IXMDB outperforms SUCXENT++ for




(a) IXMDB




(b) SUCXENT++

Fig. 25. Access plan for MQ3

queries Q1, Q8-10, Q14, Q17, MQ1-3, etc.

Furthermore, *schema-oblivious* approach cannot provide the typed representation and access of the content within the XML documents. With *schema-oblivious* approach, the data within the XML documents will be only stored as character string datatype and the corresponding index system is only created on string datatype. However, IXMDB and *schema-conscious* approach can benefit from the efficient index mechanism created on all kinds of datatypes. For some test queries, SUCXENT++ has to spend time to convert datatype, for example, Q5, Q11, Q12 and Q18, which include numeric comparison.

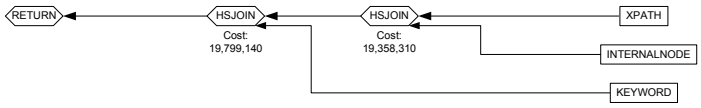
Compared to the other *schema-oblivious* approaches, SUCXENT++ only stores the leaf nodes information and the internal nodes information has been eliminated. This gives rise to a drawback that it is inefficient to implement the queries with conditions on internal nodes, for example, Q2-4, MQ5, MQ7 and MQ8, which are ordered access queries, and Q6 and Q7, which are to count the occurrence of given internal nodes. To perform these queries, SUCXENT++ needs the assistance of an additional programming code.

Because the required data is stored in several tables and the extraction process involves several joins with IXMDB; for reconstructing a fragment of original XML document, e.g., Q13, MQ5 and MQ7, IXMDB performs worse than SUCXENT++. According to Fig.21, the extraction performance of IXMDB is slightly better than SUCXENT++. However, for IXMDB, the process of reconstructing a fragment of document is different from the process of reconstructing the whole document. During the former process, several joins

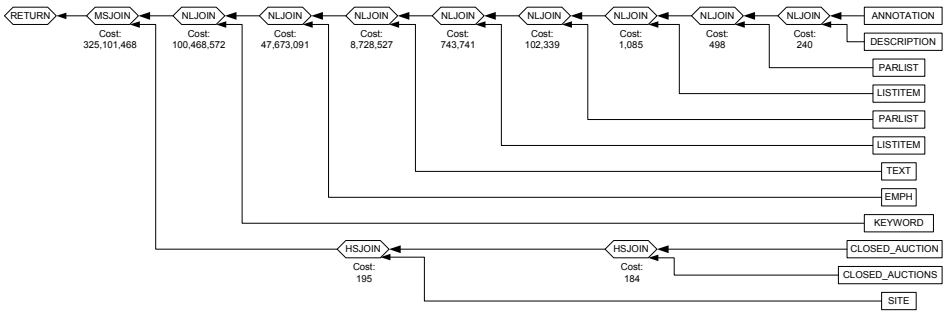
between the leaf node tables and the internal node tables are involved when extracting the required data, while for reconstructing the whole document, these joins are not necessary, since with the document identifier and the table information kept in MPD file, we can extract the required data from leaf node tables directly.

***IXMDB vs. Shared-Inlining***

The main drawback of the *schema-conscious* approach, like Shared-Inlining, is the lack of path expression information and path index. For the recursive queries (e.g. Q6, Q7, Q14 and Q19) and the queries including longer path expressions (e.g. Q15 and Q16), the Shared-Inlining method performs worse than IXMDB and the *schema-oblivious* approaches. Shared-Inlining only keeps the parent-children relationships by defining a set of primary keys and foreign keys. IXMDB only requests two $\theta$-joins to check the ancestor-descendent relationships, while Shared-Inlining may perform large number of equijoins to check ancestor-descendent relationships or travel from root node to destination node along the path expression to access appropriate data. The number of equijoins depends on the depth of the path expression. Therefore, the recursive queries in which the exact depth is unknown and the path expression with large depth affect the query performance of Shared-Inlining. Fig.26 shows the access plan for Q15 with IXMDB and Shared-Inlining respectively, and illustrates the great impact of the long path expression on the performance of Shared-Inlining.




(a) IXMDB




(b) Shared-Inlining

Fig. 26. Access plan for Q15

***Queries on complex datatypes in MPEG-7 descriptions***

MQ4 and MQ8-12 are the queries for testing the performance on complex datatypes defined in MPEG-7 DDL. None of the existing XML storage so-

lutions provide the schema to handle the complex datatypes in MPEG-7 descriptions. These datatypes are only stored as character string type in the database. It makes manipulation on these datatypes very inefficient.

MQ4 is to test the performance of date arithmetic operation on *basicTimePoint* and *basicDuration* types. As these types are stored as character string, the existing XML solutions need additional program code to implement date arithmetic operations. While IXMDB only requests a single SQL to implement this query.

Queries MQ8-12 include the operations on array and matrix data. The detailed discussion can be found in the following subsection.

### *7.4.2 Performance on array and matrix data*

To evaluate further the efficiency of our storage schema for array and matrix data, we generated additional MPEG-7 description data sets with the size of 1MB, 10MB and 100MB, which include *DominantColor D, Object DS, StateTransitionModel DS* and *SoundClassificationModel DS*. *DominantColor D* has the array data with 3 dimensions and we used semi-normalized schema to store them. *StateTransitionModel DS* has $3 \times 3$ matrices, while the matrices in *Object DS* and *SoundClassificationModel DS* have the dimensions of $2 \times 4$, $2 \times 5$, $20 \times 20$, $31 \times 20$, etc. We used normalized schema to store these matrices. Fig.27 shows the characteristic of these three data sets and the table sizes of IXMDB for storing these arrays and matrices. We used MQ8-12 for testing and Fig.28 shows the experimental results.

Queries MQ8 and MQ9 include the operations on the individual item within the array data. For example, for the *DominantColor* descriptor, if the colour space is *HMMD*, the value of *ColorValueIndex*, which is defined as array type with 3 length, would be a set of three components: *Hue, Diff* and *Sum*. As shown in MQ9, if the users want to find the pictures with blue dominant colour, the following condition will be issued: "the value of Hue should be from 160 to 210" (this value range specifies the blue colour). According to the storage schema for the array or matrix data designed with IXMDB, IXMDB can benefit from the index on the individual items of the array data and speed up these queries. However, the other methods, which store the array data as character string, need to parse each candidate array string, get the individual item within the array string, convert the datatype, and then test whether it accords with the query condition. Thus, IXMDB outperforms them undoubtedly.

Queries MQ10-12 are the queries for testing the performance on the matrix datatype. They are issued against *Object DS*, *StateTransitionModel DS* and *SoundClassificationModel DS* respectively. Same as the performance on array

| Dataset | Num of arrays | Num of matrices | Num of tuples in array table | Size of array table (MB) | Num of tuples in matrix table | Size of matrix table (MB) |
|---|---|---|---|---|---|---|
| 1M | 4048 | 58 | 4048 | 0.17 | 19816 | 0.65 |
| 10M | 44431 | 592 | 44431 | 1.85 | 118773 | 3.92 |
| 100M | 440965 | 6069 | 440965 | 18.32 | 1189151 | 39.04 |

Fig. 27. Features of array/matrix storage

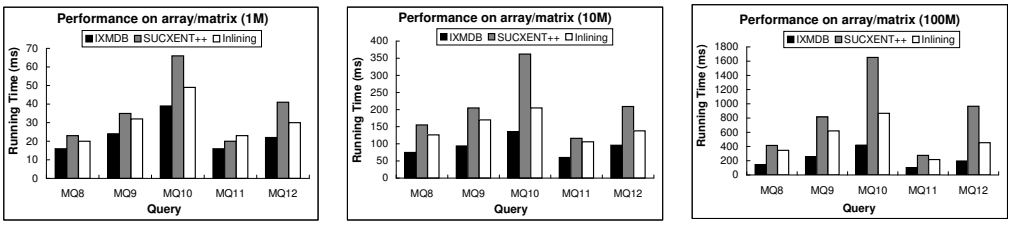



Fig. 28. Experimental Result: Query Performance on Array/Matrix

data, SUCXENT++ and Shared-Inlining underperform IXMDB due to the additional operations on character string parsing, datatype conversion and without index on the individual items of the matrices. Furthermore, to parse the character string, SUCXENT++ and Shared-Inlining need the value of the attribute *dim*, which records the dimension information of the matrix. This results in one more join for SUCXENT++ to get the value of *dim* that corresponds to the matrix in question.

According to the Fig.28, we observed that for the small data set, the advantage of IXMDB is slight. In fact, our array/matrix storage schema has a disadvantage. It needs more joins (joins between leaf node table and array/matrix table) to get required array/matrix data. Compared to the other methods without the indices on the individual items of array/matrix data, however, with large collections of array/matrix data, the impact of index mechanism in IXMDB results in the significant efficient performance.

### *7.4.3 Queries from the multimedia perspective*

As mentioned in the previous section, the GiST framework is connected with our RDBMS-based MPEG-7 storage system via a set of UDFs. This enables our system to support efficient multimedia-related queries. To test the performance from the multimedia perspective, we performed a similarity search on four low-level MPEG-7 descriptors extracted from 300,000 pictures. The tested descriptors include *ColorLayout(CLD)* with 12 dimensions, *RegionShape(RSD)* with 35 dimensions, *EdgeHistogram(EHD)* with 80 dimensions, and *ColorStructure(CSD)* with 128 dimensions. The similarity search in our experiment is to find the top 10 objects that are the most similar to the given object based on each descriptor. In this experiment, we adopted M-Tree in IXMDB, since M-Tree is based on distance function. Fig.29 presents the corresponding experimental result.

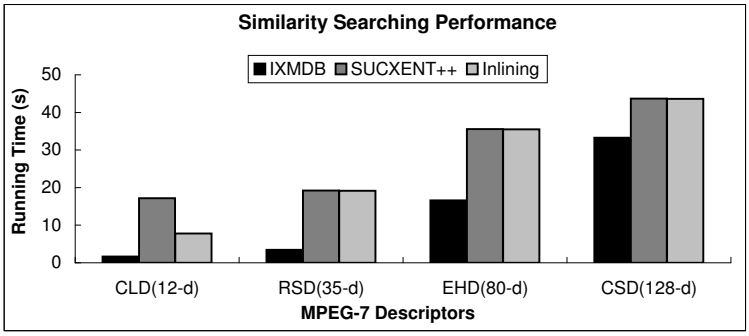



Fig. 29. Experimental Result: Query Performance from the multimedia perspective

According to this figure, IXMDB outperforms the existing XML storage systems for similarity searching. Due to lack of efficient high-dimensional index mechanism, the existing XML storage systems can only perform sequence scan to achieve similarity matching, while IXMDB can benefit from the GiST framework. Furthermore, the existing XML storage systems need to extract desired multimedia content from their MPEG-7 descriptions repository before performing similarity searching. The performance of this extraction process depends on the complexity of related multimedia content and their storage schema. Such an extraction process may be expensive. For example, for the *ColorLayout* descriptor, the related multimedia content is stored in several elements. Thus, SUCXENT++ needs several joins to extract related information and this operation is expensive.

## 8 Related Research

As introduced in the previous section, the existing RDBMS-based XML storage approaches can be classified into two major categories: *schema-conscious* approach and *schema-oblivious* approach. Some typical examples of *schema-oblivious* approach include The Edge Approach[15], Monet[16], Xrel[17], XParent[18], SUCXENT++[11], etc. The Edge approach stores XML data graphs (a directed graph) in a single *Edge* table. As a variation of the Edge approach, Monet partitions the schema of XML documents by all possible paths. For each unique path, Monet creates a table. Unlike Monet, XRel explicitly keeps all unique path expression as tuples in a table. XRel records elements relationships using the notion of *region*, which is a pair of numbers that represents the start and end positions, respectively, of a node in an XML document. XParent is a four (or five) table database schema and materializes the ancestor-descendant relationship in a special table. SUCXENT++ is different from the above approaches in that it only stores leaf nodes and their associated paths.

Some examples of *schema-conscious* approach can be found in Basic, Shared and Hybrid Inlining Technique[12] and LegoDB[19]. The Inlining technique is

an early proposal of *schema-conscious* approach. With this approach, complex DTD specifications are first simplified with a set of transformation rules and corresponding transformed DTD graphs are obtained. Then, three techniques, including Basic Inlining, Shared Inlining and Hybrid Inlining, are used for converting the simplified DTD to a relational schema. The LegoDB can automatically find an efficient relational configuration for a target XML application.

The database products of leading database system vendors have extended their relational database functions to support the XML documents storage. They can be viewed as XML-Enabled databases, which also provide RDBMS-based XML storage solutions. Some products of these include *DB2 XML Extender*[25], *XML Support in SQL Server*[26,27] and *Oracle XML DB*[28]. These XML-Enabled database systems introduce one or more special datatypes for XML storage and provide a set of functions on these datatypes to support XML management. The intact XML content can be stored in a column with VARCHAR or CLOB datatype. A set of powerful built-in functions or methods is provided to query and modify XML instances and they also accept XQuery.

There exist some research works that focus on the hybrid relational and XML database system, e.g. *System RX*[29], which enables XML and relational data to co-exist and complement each other. Some other research works focus on how to implement XQuery in a relational database. These works can be found in [30,31].

There are a few research works on the MPEG-7 descriptions management systems. The examples of these works are the MPEG-7 Multimedia Data Cartridge (MDC)[37] and PTDOM[38]. MPEG-7 MDC is a system extension of the Oracle 9i DBMS to provide a new indexing and query framework for various types of retrieval operations and a semantically rich metadata model for multimedia content relying on the MPEG-7 standard. Although the MPEG-7 MDC provides a robust storage solution for MPEG-7 descriptions, it falls short when evaluated in terms of the requirements listed in [4]. First, MDC is a system extension of the Oracle 9i DBMS. It defines a set of object types to map the MPEG-7 standard into a database model. The data within such object types, e.g., *XMLType* object type, may not be represented and accessed in fine-grained and typed manner. Second, the predefined object types may not be suitable for the non-fixed MPEG-7 descriptions with volatile structures. Finally, it is difficult to provide path index for navigation and extraction of fragment information within MPEG-7 descriptions. PTDOM is a schema-aware native XML database system originally developed for the management of MPEG-7 descriptions. The core of PTDOM is made up of a schema catalog capable of managing schema definitions written in MPEG-7 DDL [38]. To represent the contents of the MPEG-7 descriptions, the document manager of

PTDOM applies the TDOM[24], an object model in the tradition of DOM.

## 9 Conclusions

In this paper, IXMDB, a new approach to mapping, indexing and retrieving MPEG-7 documents and other data-centric XML documents using relational database system, has been described. IXMDB integrates the advantages of *schema-conscious* approach and *schema-oblivious* approach. Unlike the *schema-conscious* method, IXMDB supports XPath-based query efficiently without involving many joins in SQL. Compared with the *schema-oblivious* method, IXMDB solves the datatype problem in *schema-oblivious* approach without sacrificing the performance, and IXMDB even performs better than most *schema-oblivious* approach in the case of many XPath based queries. Furthermore, IXMDB provides a flexible storage schema to satisfy all kinds of storage requirements, especially for the special datatypes within MPEG-7 descriptions, such as *array, matrix, basicTimePoint* and *basicDuration*. Although IXMDB cannot avoid the assistance of MPEG-7 scheme, it can support arbitrary MPEG-7 description storage. IXMDB supports the most critical requirements for the MPEG-7 descriptions management, such as fine grained and typed representation and access, index system and XPath-based query. Finally, we also introduced a multidimensional index system based on extensible GiST framework to support multimedia content retrieval.

## References


[1] José M. Martínez, MPEG-7 Overview (version 8), ISO/IECJTC1/SC29/WG11-N4980, Klangenfurt, 2002, Available at http://www.mpeg-industry.com/mp7a/w4980_mp7_Overview1.html.

[2] Diane Hillmann, Using Dublin Core, Dublin Core Metadata Initiative (DCMI), 2005, Available at http://dublincore.org/documents/usageguide/.

[3] The TV-Anytime Forum, Specification Series: S-3 On: Metadata, 2003, Available at http://www.tv-anytime.org/.

[4] Utz Westermann and Wolfgang Klas, An Analysis of XML Database Solutions for the Management of MPEG-7 Media Descriptions, ACM Computing Surveys 35(4) 2003 pp. 331-373.

[5] Yang Chu, Liang-Tien Chia and Sourav S. Bhowmick, SM3+: An XML Database Solution for the Management of MPEG-7 Descriptions, in: Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA), Copenhagen, 2005, p. 134.

[6] Igor Tatarinov and Stratis D. Viglas, Storing and Querying Ordered XML Using a Relational Database System, in: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin, 2002, pp. 204-215.

[7] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller and Nigel Westbury, ORDPATHs: Insert Friendly XML Node Labels, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, Paris, France, 2004, pp. 903-908.

[8] ISO/IEC JTC 1/SC 29, Information Technology - Multimedia Content Description Interface - Part 2: Description Definition Language, ISO/IEC FDIS 15938-2, 2001.

[9] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie and Jérôme Siméon, XQuery 1.0: An XML Query Language, W3C Working Draft 29, 2004, Avalibale at http://www.w3.org/TR/xquery/.

[10] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer, Generalized Search Trees for Database Systems, in: Proceedings of 21st International Conference on Very Large Data Bases (VLDB'95), Zurich, Switzerland, 1995, pp. 562-573.

[11] Sandeep Prakash, Sourav S. Bhowmick and Sanjay Madria, Efficient Recursive XML Query Processing in Relational Database Systems, in: Proceedings of 23rd International Conference on Conceptual Modeling (ER2004), Shanghai, China, 2004, pp. 493-510.

[12] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt and Jeffrey Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities, in: Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, 1999, pp. 302-314.

[13] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu and Ralph Busse, XMark: A Benchmark for XML Data Management, in: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), Hong Kong, China, 2002, pp. 974-985.

[14] Feng Tian, David J. DeWitt, Jianjun Chen and Chun Zhang, The design and performance evaluation of alternative XML storage strategies, ACM Sigmod Record 31(1) (2002) pp. 5-10.

[15] Daniela Florescu and Donald Kossmann, Storing and querying XML data using an RDBMS, IEEE Data Engineering Bulletin 22(3) (1999) pp. 27-34.

[16] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer and Florian Waas, Efficient Relational Storage and Retrieval of XML Documents, in: Proceedings of the Third International Workshop on the Web and Databases (WebDB 2000), Dallas, Texas, USA, 2000, pp. 137-150.

[17] Masatoshi Yoshikawa and Toshiyuki Amagasa, XRel: A path-based approach to storage and retrieval of XML documents using relational databases, ACM Transactions on Internet Technology 1(1) (2001) pp. 110-141.

[18] Haifeng Jiang, Hongjun Lu, Wei Wang and Jeffrey Xu Yu, XParent: An Efficient RDBMS-Based XML Database System, in: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), San Jose, CA, USA, 2002, p. 335.

[19] Philip Bohannon, Juliana Freire, Prasan Roy, Jérôme Siméon, From XML Schema to Relations: A Cost-Based Approach to XML Storage, in: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), San Jose, CA, USA, 2002, p. 64.

[20] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom, Lore: A database management system for semistructured data, SIGMOD Record, 26(3) (1997) pp. 54-66.

[21] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, et al, Timber: A native xml database, The VLDB Journal, 11(4) (2002) pp. 274-291.

[22] Carl-Christian Kanne and Guido Moerkotte, Efficient storage of XML data, in: Proceedings of the 16th International Conference on Data Engineering (ICDE'00), San Diego, California, USA, 2000, p. 198.

[23] Sleepycat Software, Introduction to Berkeley DB XML, Technical Documentation, Sleepycat Software, Massachusetts, USA, 2005.

[24] Utz Westermann and Wolfgang Klas, A Typed DOM for the Management of MPEG-7 Media Descriptions, Multimedia Tools and Applications, 27(3) (2005) pp. 291-322.

[25] Josephine Cheng and Jane Xu, IBM DB2 XML Extender: An end-to-end solution for storing and retrieving XML documents, Reprinted, with permission from the 16th International Conference on Data Engineering (ICDE '00), San Diego, California, USA, 2000.

[26] Shankar Pal, Mark Fussell, and Irwin Dolobowsky, XML Support in Microsoft SQL Server 2005, The Microsoft Developer Network (MSDN), Microsoft Corporation, 2004.

[27] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis and Vasili Zolotov, Indexing XML Data Stored in a Relational Database, in: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04), Toronto, Canada, 2004, pp. 1134-1145.

[28] Mark Drake, Oracle XML DB, Oracle Technical White Paper, Oracle Corporation, 2004.

[29] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, et al, System RX: one part relational, one part XML, in: Proceedings of the 2005 ACM SIGMOD, Baltimore, Maryland, 2005, pp. 347-358.

[30] David DeHaan, David Toman, Mariano P. Consens and M. Tamer özsu, A comprehensive XQuery to SQL translation using dynamic interval encoding, in: Proceedings of the 2003 ACM SIGMOD, San Diego, California, 2003, pp. 623-634.

[31] Shankar Pal, Istvan Cseri, Oliver Seeliger, et al, XQuery implementation in a relational database system, in: Proceedings of the 31st international conference on Very large data bases (VLDB), Trondheim, Norway, 2005, pp. 1175-1186.

[32] Antonin Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, Boston, Massachusetts, 1984, pp. 47-57.

[33] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD international conference on Management of data, Atlantic City, New Jersey, United States, 1990, pp. 322-331.

[34] David A. White and Ramesh Jain, Similarity Indexing with the SS-tree, in: Proceedings of the 12th International Conference on Data Engineering, Lillehammer, Norway, 1996, pp. 516-523.

[35] Norio Katayama and Shin'ichi Satoh, The SR-tree: an index structure for high-dimensional nearest neighbor queries, in: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, Tucson, Arizona, United States, 1997, pp. 369-380.

[36] Paolo Ciaccia, Marco Patella and Pavel Zezula, M-tree: an efficient access method for similarity search in metric spaces, in: Proceedings of the 23rd International Conference on Very Large Databases (VLDB'97), Athens, Greece, 1997, pp. 426-435.

[37] Mario Döllera and Harald Koscha, An MPEG-7 Multimedia Data Cartridge, in: Proceedings of SPIE Conference on Multimedia Computing and Networking 2003 (MMCN 2003), Santa Clara, 2003, pp. 126-137.

[38] Utz Westermann and Wolfgang Klas, PTDOM: a schema-aware XML database system for MPEG-7 media descriptions, Software-Practice & Experience, 36(8) (2006) pp. 785-834.

# APPENDIX

## A SQL queries in IXMDB

We don't list all the SQLs. The unlisted queries are similar to the following queries, or cannot be implemented with single SQL.

**Q1**

```
SELECT p.name
FROM person p,internalnode i, xpath x
WHERE x.xpathexp = '#/site#/people#/person'
        and i.xpathid = x.xpathid
        and p.id='person0'
        and p.uid=i.uid
```

**Q2**

```
SELECT b.increase
FROM bidder b, internalnode i, xpath x
WHERE x.xpathexp = '#/site#/open_auctions#/open_auction#/bidder'
        and i.xpathid = x.xpathid
         and i.oid=1 and b.uid=i.uid
```

**Q5**

```
SELECT count(c.price)
FROM closed_auction c,internalnode i, xpath x
WHERE x.xpathexp = '#/site#/closed_auctions#/closed_auction'
        and i.xpathid = x.xpathid
        and c.uid=i.uid
        and c.price>=40
```

**Q6**

```
SELECT count(i.uid)
FROM internalnode i, xpath x
WHERE x.xpathexp like '#/site#/regions#%/item'
        and i.xpathid = x.xpathid
```

**Q8**

```
SELECT p.name, count(b.person)
FROM xpath x1,xpath x2,internalnode i1,internalnode i2,buyer b,person p
WHERE x1.xpathexp = '#/site#/closed_auctions#/closed_auction#/buyer'
        and x2.xpathexp = '#/site#/people#/person'
        and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid
        and b.uid=i1.uid and p.uid=i2.uid
        and b.person=p.id
GROUP BY p.name
```

**Q11**

```
SELECT pe.name,count(o.uid)
FROM xpath x1,xpath x2,xpath x3,internalnode i1,internalnode i2,internalnode i3,
      open_auction o,profile pr,person pe
WHERE x1.xpathexp = '#/site#/people#/person#/profile'
        and x2.xpathexp = '#/site#/open_auctions#/open_auction'
        and x3.xpathexp = '#/site#/people#/person'
        and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid and i3.xpathid = x3.xpathid
        and pr.uid = i1.uid and o.uid=i2.uid and pe.uid=i3.uid
        and pr.income > 5000*o.initial
        and i1.parent = i3.ordpath
GROUP BY pe.name
```

**Q15**

```
FROM xpath x,internalnode i,keyword k
WHERE x. xpathexp =   '#/site#/closed_auctions#/closed_auction#/annotation#/description#/parlist#/listitem
                      #/parlist#/listitem#/text#/emph#/keyword'
        and i.xpathid = x.xpathid
        and k.uid=i.uid";
```

**Q17**

```
SELECT p.name
FROM xpath x,internalnode i,person p
WHERE x. xpathexp = '#/site#/people#/person'
        and i.xpathid = x.xpathid
        and p.uid = i.uid
        and p.homepage is null
```

**Q18**

```
SELECT MULTIPLY_ALT(2.20371,o.reserve)
FROM xpath x,internalnode i, open_auction o
WHERE x. xpathexp = '#/site#/open_auctions#/open_auction'
        and i.xpathid = x.xpathid
        and o.uid=i.uid"
        and o.reserve is not null
```

**Q19**

```
SELECT it.name,it.location
FROM xpath x,internalnode i,item it
WHERE x. xpathexp like '#/site#/regions#%/item'
        and i.xpathid = x.xpathid
        and it.uid=i.uid
ORDER BY it.name
```

**Q20**

```
WITH temp(id,level) AS
      ( SELECT p.uid, CASE
                        WHEN p.income >= 100000 THEN 'preferred'
                        WHEN p.income >= 30000 and p.income < 100000 THEN 'standard'
                        WHEN p.income < 30000 THEN 'challenge'
                        ELSE 'na'
                      END
        FROM xpath x,internalnode i,profile p
        WHERE x. xpathexp = '#/site#/people#/person#/profile'
              and i.xpathid = x.xpathid
              and p.uid=i.uid )
SELECT t.level, count(t.id)
FROM temp t
GROUP BY t.level
```

**MQ1**

```
SELECT u.protected
FROM xpath x1,xpath x2,internalnode i1,internalnode i2,useridentifier u,name n
WHERE x1.xpathexp like '#%/UserIdentifier#/Name'
        and x2.xpathexp like '#%/UserIdentifier'
        and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid
        and n.uid = i1.uid and u.uid = i2.uid
        and i1.parent=i2.ordpath
        and n.value = 'John Doe'
```

**MQ4**

```
SELECT o.timestamp + decimal(o.timestampduration,20,6)
FROM xpath x1,xpath x2,xpath x3,internalnode i1,internalnode i2,internalnode i3,name n,observationperiod o
WHERE x1.xpathexp like '#%/UsageHistory'
        and x2.xpathexp like '#%/UsageHistory#/UserIdentifier#/Name'
        and x3.xpathexp like '#%/UsageHistory#/UserActionHistory#/ObservationPeriod'
        and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid and i3.xpathid = x3.xpathid
        and n.uid = i2.uid and o.uid = i3.uid
        and i2.ordpath > i1.ordpath and i2.ordpath < i1.grdesc
        and i3.ordpath > i1.ordpath and i3.ordpath < i1.grdesc
        and n.value = 'John Doe'
        and o.timepoint like '%2000-10-10T18:00%'
```

**MQ9**

```
SELECT e.bincounts
FROM xpath x1,xpath x2,xpath x3,internalnode i1,internalnode i2,internalnode i3,
        values v,edgehistogram e,array a
WHERE x1.xpathexp like '#%/Descriptor#/Values'
          and x2.xpathexp like '#%/Descriptor'
          and x3.xpathexp like '#%/Descriptor'
          and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid and i3.xpathid = x3.xpathid
          and i1.parent = i2.ordpath
          and v.uid = i1.uid
          and e.uid = i3.uid
          and i3.oid = i2.oid
          and v.percentage > 20
          and v.arrayid = a.id
          and a.col0 > 160 and a.col0 < 210
```

**MQ10**

```
WITH temp(id,value) AS (SELECT matrixid,min(value) FROM matrix WHERE columnid=0 GROUP BY matrixid )
SELECT l.name
FROM xpath x1,xpath x2,xpath x3,xpath x4,internalnode i1,internalnode i2,internalnode i3,internalnode i4
        label l, medialocator m, coords c,temp t
WHERE x1.xpathexp like ' #%/Object'
          and x2.xpathexp like ' #%/Object#/Label'
          and x3.xpathexp like ' #%/Object#/MediaOccurrence#/MediaLocator'
          and x4.xpathexp like ' #%/Object#/MediaOccurrence#/SubRegion#/Polygon#/Coords'
          and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid
          and i3.xpathid = x3.xpathid and i4.xpathid = x4.xpathid
          and i2.parent = i1.ordpath
          and i3.ordpath > i1.ordpath and i3.ordpath < i1.grdesc
          and i4.ordpath > i1.ordpath and i4.ordpath < i1.grdesc
          and l.uid=i2.uid and m.uid = i3.uid and c.uid = i4.uid
          and m. mediauri = ‘image0.jpg’
          and c.matrixid=t.id
          and t.value>10
```

**MQ11**

```
WITH temp(id,value) AS
        ( SELECT i1.oid,l.name
          FROM xpath x1,xpath x2,internalnode i1,internalnode i2, label l
          WHERE x1.xpathexp like '#%/Model#/State'
                and x2.xpathexp like '#%/Model#/State#/Label'
                and i1.xpathid = x1.xpathid and i2.xpathid = x2.xpathid
                and i2.parent = i1.ordpath)
SELECT m.value
FROM xpath x1,internalnode i1,transitions t,matrix m,temp te
WHERE x1.xpathexp like '#%/Model#/Transitions '
          and i1.xpathid = x1.xpathid
          and t.uid=i1.uid
          and t.matrixid=m.matrixid
          and m.columnid=te.id and te.value=’Pass’
          and m.rowid=te.id and te.value=’Shot on goal’
```