

Colorful XML: One Hierarchy Isn't Enough

H. V. Jagadish[†]
U. of Michigan
jag@eecs.umich.edu

Laks V. S. Lakshmanan[‡]
U. of British Columbia
laks@cs.ubc.ca

Monica Scannapieco
U. di Roma La Sapienza[‡]
monscan@dis.uniroma1.it

Divesh Srivastava
AT&T Labs Research
divesh@research.att.com

Nuwee Wiwatwattana
U. of Michigan
nuwee@eecs.umich.edu

ABSTRACT

XML has a tree-structured data model, which is used to uniformly represent structured as well as semi-structured data, and also enable concise query specification in XQuery, via the use of its XPath (twig) patterns. This in turn can leverage the recently developed technology of structural join algorithms to evaluate the query efficiently. In this paper, we identify a fundamental tension in XML data modeling: (i) data represented as deep trees (which can make effective use of twig patterns) are often un-normalized, leading to update anomalies, while (ii) normalized data tends to be shallow, resulting in heavy use of expensive value-based joins in queries.

Our solution to this data modeling problem is a novel multi-colored trees (MCT) logical data model, which is an evolutionary extension of the XML data model, and permits trees with multi-colored nodes to signify their participation in multiple hierarchies. This adds significant semantic structure to individual data nodes. We extend XQuery expressions to navigate between structurally related nodes, taking color into account, and also to create new colored trees as restructurings of an MCT database. While MCT serves as a significant evolutionary extension to XML as a logical data model, one of the key roles of XML is for information exchange. To enable exchange of MCT information, we develop algorithms for optimally serializing an MCT database as XML. We discuss alternative physical representations for MCT databases, using relational and native XML databases, and describe an implementation on top of the Timber native XML database. Experimental evaluation, using our prototype implementation, shows that not only are MCT queries/updates more succinct and easier to express than equivalent shallow tree XML queries, but they can also be significantly more efficient to evaluate than equivalent deep and shallow tree XML queries/updates.

[†]Supported in part by NSF under grants IIS-0219513 and IIS-0208852.

[‡]Supported in part by grants from NSERC (Canada) and BC Advanced Systems Institute.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2004 June 13-18, 2004, Paris, France.

Copyright 2004 ACM 1-58113-859-8/04/06 ... \$5.00.

1. INTRODUCTION

XML (eXtensible Markup Language) is rapidly becoming the de facto standard for exchanging data between applications, and publishing data, on the Web. The data model that underlies XML is tree-structured, comprising nodes, atomic values and sequences [14], and is used by query languages such as XPath [5] and XQuery [7].

The importance of the tree structure in the XML data model can be appreciated from the rich variety of ways in which XPath and XQuery support navigation between structurally related nodes in an XML database. In contrast, for XML nodes that are related only through values (of the ID/IDREF attributes, or otherwise), one needs to explicitly perform value-based joins, one edge at a time, akin to SQL queries over relational databases. The following example is illustrative:

EXAMPLE 1.1. [Deep Trees vs Shallow Trees]

Consider a movie database, with elements *movie*, *movie-genre*, *movie-award*, *actor* and *movie-role*. There are many ways of organizing this information in XML, two of which are:

Deep-1: The *movie-genre* nodes are hierarchically organized, with each *movie* as a child of its primary *movie-genre* node. Each *movie* has children *movie-award* and *movie-role* nodes, and *movie-role* nodes have children *actor* nodes.

Shallow-1: The nodes are in a shallower tree structure, and the relationships between *movie-genre* and *movie* nodes, between *movie-award* and *movie* nodes, and between *movie*, *actor* and *movie-role* nodes, are captured via attributes values. Nodes have an *id* attribute, a *movie* may have a *movieAwardIdRefs* and a *movieGenreIdRefs* attributes, and *movie* and *actor* nodes have the *roleIdRefs* attributes.

Consider the query *Return names of comedy movies nominated for an Oscar, in which Bette Davis acted*. In the Deep-1 approach, one can write the following XQuery expression:

```
for $m in document("mdb.xml")//movie-genre
  [name = "Comedy"]//movie[../actor/name
    = "Bette Davis"]
where contains($m/movie-award/name, "Oscar")
return <m-name> || $m/name || </m-name>
```

In the Shallow-1 approach, one would have to use value-based joins. The XQuery expression would be:

```
for $mg in document("mdb.xml")//movie-genre
  [name = "Comedy"]//movie-genre,
  $m in document("mdb.xml")//movie,
  $ma in document("mdb.xml")//movie-award,
  $a in document("mdb.xml")//actor
```

```

    [name = "Bette Davis"],
    $r in document("mdb.xml")//movie-role
where contains($ma/name, "Oscar") and
    $mg/@id = $m/@movieGenreIdRef and
    contains($m/@movieAwardIdRefs, $ma/@id) and
    contains($m/@roleIdRefs, $r/@id) and
    contains($a/@roleIdRefs, $r/@id)
return <m-name> || $m/name || </m-name>

```

Note that the Deep-1 query expression is much simpler than that of Shallow-1. The increased complexity of the Shallow-1 expression would also (typically) result in an expensive evaluation, which cannot make an effective use of structural joins developed for the efficient evaluation of XQuery path expressions [2, 8]. ■

The improved query specification and evaluation in deeper trees over shallower trees comes at a cost. The deeper representations are un-normalized [3], and the replication of data (e.g., the actor and the movie-award nodes, in the above example) raises the problem of update anomalies (e.g., if one wanted to add a subelement birthDate to an actor). It appears that neither the deep tree approach nor the shallow tree approach is ideal both for queries and for updates. What is an XML database designer to do?

The solution proposed in this paper to effectively address the above-mentioned inadequacies of the conventional XML data model is a novel logical data model, referred to as *multi-colored trees* (MCT). Our MCT data model is an evolutionary extension of the XML data model of [14] and, intuitively, permits multiple colored trees to add semantic structure over the individual nodes in the XML data. Individual nodes can have one or more colors, permitting them to be hierarchically related to other nodes in a variety of ways, instead of only in one way. This allows (extended) XQuery expressions to navigate between structurally related nodes, taking color into account, instead of relying heavily on value-based joins. An (enhanced) XQuery expression can be used to create a new colored tree over a combination of newly created and existing nodes, and an (enhanced) update expression can be used to modify existing data in the MCT data model. We develop our technical contributions in the rest of the paper as follows:

- We present the MCT logical data model, consisting of evolutionary extensions to the XML data model, in Section 3.
- We propose extensions to the XQuery query language, for the MCT logical data model, in Section 4.
- While MCT serves as a significant evolutionary extension to XML as a logical data model, one of the key roles of XML is for information exchange. To enable exchange of MCT information, we develop an algorithm for serializing an MCT database in a schema-optimal way, as XML, in Section 5.
- We discuss alternative ways in which a logical MCT database can be physically represented and manipulated, using relational and native XML databases, and describe an implementation on top of the Timber native XML database, in Section 6.
- We used our prototype implementation to experimentally compare MCT queries and updates against equivalent XML queries and updates, both for shallow and for deep XML trees, in Section 7. Our results demonstrate that not only are MCT queries/updates more succinct and easier to express than equivalent shallow tree XML queries/updates, but they can also be significantly more efficient to evaluate than equivalent deep and shallow tree XML queries/updates.

Anecdotal evidence suggests to us that choosing a suitable hierarchy structure is one of the more difficult tasks in XML database schema design. The use of a multi-colored tree model eases the burden of the designer, while at the same time permitting concise query specification and efficient query evaluation over a range of queries that could not all be well supported with a single choice of hierarchy.

Next, in Section 2, we present an overview of our MCT data model, and highlight its benefits over the conventional XML data model using examples. Related work is discussed in Section 8, and we conclude in Section 9, outlining several areas of research opened up by the MCT data model.

2. OVERVIEW OF THE MCT MODEL

The W3C has focused considerable recent attention to developing a logical model and query language for XML (see, for example, [14, 27, 6, 7]). The XML data model is an ordered tree of nodes, with atomic values associated with leaf nodes.

Using examples, we next highlight the benefits of permitting multiple trees, instead of just a single tree, to add semantic structure over the individual data nodes. Each tree is distinguished from the others by a *color*, and the resulting logical model is called the *multi-colored tree* (MCT) data model. We will present a formal development of the MCT data model in subsequent sections.

2.1 movie Nodes with Multiple Colors

Consider, again, the movie database from Example 1.1. There are several natural hierarchies: movie genres are akin to a topic hierarchy (e.g., comedy and action are sibling genres, and slapstick is a sub-genre of comedy), and the Oscar best-movie awards can be organized into a temporal hierarchy. Individual movies can be naturally classified into *each* of these hierarchies: a movie can be a child of its most-specific primary movie genre, and, if the movie was nominated for a best-movie Oscar award in a particular year, it can be made a child of that year's node in the best-movie award hierarchy.

Explicitly modeling such hierarchies in XML allows XQuery expressions to be effectively used for formulating queries like query Q1 (in Figure 1), without having to identify the most-specific genre of the movie. While XML allows either of these hierarchies to be modeled, it does not permit a natural modeling of *both* these hierarchies simultaneously; one of these hierarchical relationships would need to be captured using attribute values, increasing the complexity of the XQuery specification of a query like Q2 (in Figure 1).

Our multi-colored tree data model extends the XML data model in permitting *both* these hierarchies to be first-class semantic hierarchies over the data nodes, simultaneously. Queries like Q2 can be easily expressed in a simple extension of XQuery, that takes color into account in its path expressions. We illustrate an example MCT database in Figure 2.

Example MCT Database: We depict a multi-colored tree database by showing each colored tree separately. The example MCT movie database in Figure 2 has trees of three colors: red, green and blue. For the moment, focus on just the red and the green trees. The red tree consists of, among other nodes, the hierarchy of movie-genre nodes, and their associated children name nodes. The green tree consists of, among other nodes, the temporal hierarchy of Oscar movie-award nodes, and their associated children name nodes. All edges in a colored tree have the same color, depicting the parent-child relationships in that colored tree.

A node is stored *once* in the database irrespective of how many colored trees it participates in. A node that has multiple colors

- Q1 : Return names of comedy movies whose title contains the word Eve.
- Q2 : Return names of comedy movies that were nominated for an Oscar, whose title contains the word Eve.
- Q3 : Return names of comedy movies that were nominated for an Oscar, in which Bette Davis acted.
- Q4 : Return names of actors in movies nominated for an Oscar, with more than 10 votes.
- Q5 : Return the list of Oscar nominated movies, grouped by the number of votes received.

Figure 1: Example queries against movie database

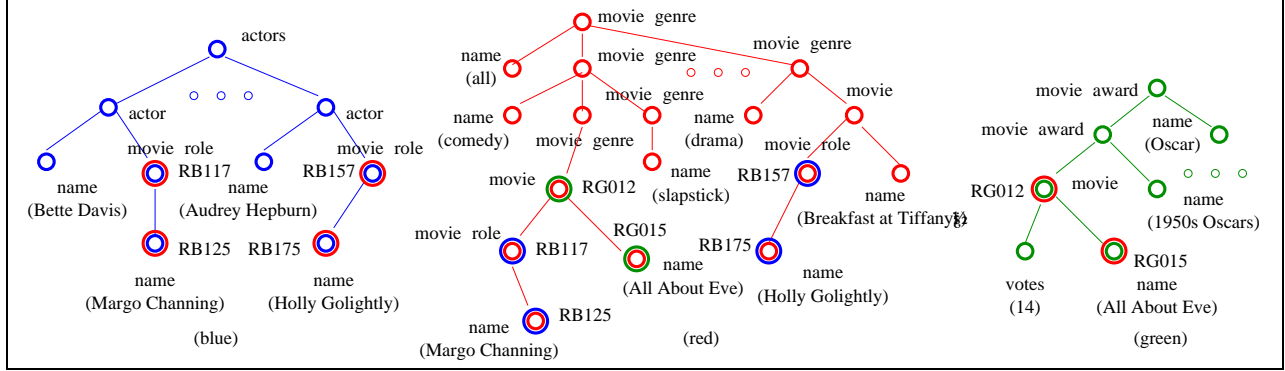


Figure 2: Example MCT database comprising three single-colored trees. Multi-colored nodes occur in multiple trees, are represented with multiple circles, and have associated an identifier label comprising the node colors (by their initials) and a unique node number, e.g., RG012.

is represented in *each* of its colored trees, e.g., as a green circle with a red outer circle in the green tree, and as a red circle with a green outer circle in the red tree. In the example MCT movie database, a movie node is both red and green (i.e., it participates in both colored hierarchies), if it has been nominated for an Oscar movie-award. A movie node is only red if it has not been nominated for an Oscar movie-award. In the example, the children name nodes of movie nodes have all the same colors as their parents. In addition, movie nodes that are both green and red have green children votes nodes, indicating the number of first-place votes received.

Example Queries: As in XQuery, *multi-colored XQuery* (MCXQuery) queries are FLWOR expressions (for, let, where, order by, return), with path expressions replaced by their colored counterparts. An ordinary path expression identifies nodes within a single tree by executing different location steps; each step generates a sequence of nodes and alters the sequence by zero or more predicates. A colored path expression additionally specifies *colored labels* with each location step, using curly braces, identifying the colored tree(s) to navigate in that location step. Unabbreviated MCXQuery expressions for queries Q1 and Q2 (of Figure 1) are given in Figure 3.

2.2 movie-role Nodes with Multiple Colors

Consider, again, our movie database in Figure 2. All actor nodes and their children name nodes are represented in a (relatively) shallow blue hierarchy. Since each movie-role node captures the relationship of an actor with a movie, these nodes (and their children name nodes) can be made both red and blue: its red parent is the movie node, and its blue parent is the actor node participating in this specific relationship. Note that, to demon-

strate the flexibility of our MCT data model, we have chosen (arbitrarily) not to let movie-role nodes be green, even if the movie was nominated for an Oscar movie-award.

Having modeled actor nodes and movie-role nodes, we can now use multi-colored XQuery to concisely express query Q3, as in Figure 3. Note that multiple colored path expressions are needed in the for clause, since we decided to conservatively extend XQuery, which currently does not support the ancestor and ancestor-or-self axes [7]. If these axes were supported, query Q3 would be expressible using a *single* colored path expression, with different colors used at different location steps.

2.3 Colors are not Views

An alternative to the MCT model is to use XML views: create deep tree views over the (stored) shallow tree data, and let users pose queries against the deep tree views. For example, one could specify Deep-1, in Example 1.1, as an XQuery view over Shallow-1. While this would ease query specification, query evaluation over an unmaterialized view would still be expensive. Further, and more importantly, updates would still be problematic. Since updates through XML views can be ambiguous in general (just as for SQL views), users would be forced to specify updates over the shallow tree representation, requiring them to be aware of two representations, one for querying and one for updates. Materializing the view can address the query evaluation efficiency, but leaves the update issue unresolved.

In contrast, each element (content and attributes) is stored precisely once in the MCT model, irrespective of the number of colors it has. Also, edges between MCT nodes are *independently* specified in each colored tree, and are expected to be semantically independent; if there are dependencies between edges, these must explicitly

```

Q1:  for $m in document("mdb.xml")/||red||descendant::movie-genre[||red||child::name = "Comedy"]/
      ||red||descendant::movie[contains( ||red||child::name, "Eve")]
      return createColor(black, <m-name> || $m/||red||child::name || </m-name>)

Q2:  for $m in document("mdb.xml")/||red||descendant::movie-genre[||red||child::name = "Comedy"]/
      ||red||descendant::movie[contains( ||red||child::name, "Eve")],
      $m in document("mdb.xml")/||green||descendant::movie-award
      [contains( ||green||child::name, "Oscar")]||green||descendant::movie
      return createColor(black, <m-name> || $m/||red||child::name || </m-name>)

Q3:  for $m in document("mdb.xml")/||green||descendant::movie-award
      [contains( ||green||child::name, "Oscar")]||green||descendant::movie,
      $r in document("mdb.xml")/||red||descendant::movie-genre[||red||child::name = "Comedy"]/
      ||red||descendant::movie[. = $m]/||red||child::movie-role,
      $r in document("mdb.xml")/||blue||descendant::actor
      [||blue||child::name = "Bette Davis"]/||blue||child::movie-role
      return createColor(black, <m-name> || $m/||red||child::name || </m-name>)

Q4:  for $a in document("mdb.xml")/||green||descendant::movie-award
      [contains( ||green||child::name, "Oscar")]||green||descendant::movie
      [||green||child::votes = 10]/||red||child::movie-role/||blue||parent::actor
      return createColor(black, <a-name> || $a/||blue||child::name || </a-name>)

Q5:  createColor(black, <byvotes> ||
      for $v in distinct-values(document("mdb.xml")/||green||descendant::votes)
      order by $v
      return
        <award-byvotes>
          ||
          for $m in document("mdb.xml")/||green||descendant::movie[||green||child::votes = $v]
          return $m
          ||
        <votes> || $v || </votes>
      </award-byvotes>
    || </byvotes>)

```

Figure 3: Example MCXQuery queries

be specified as constraints. Thus, using MCT, one can avoid redundant storage and also update anomalies.

2.4 Overview of the Rest of the Paper

Data modeling traditionally distinguishes between the *logical data model*, which consists of data values, structured by a schema and manipulated by the query language, and the *physical data model*, which focuses on the storage, indexing and transformation of these data values. We make the same distinction, and, in addition, consider an *exchange data model*, which deals with the serialization of data values for exchange between applications.

MCT is our logical data model, structuring data values using multiple hierarchies, and the bulk of this paper is devoted to MCT (Section 3), and its related query and update languages (Section 4). One of the key roles of XML is for information exchange, so we develop algorithms for serializing, as XML, an MCT database in Section 5. There are multiple physical data models currently being investigated for storing XML data, including relational and native approaches; in Section 6, we illustrate how these approaches could be extended for storing and manipulating MCT databases, and describe our implementation of MCT on top of the Timber native XML database. Finally, in Section 7, we validate our intuitions about the many benefits of MCT queries and updates, using our prototype implementation.

3. THE MCT LOGICAL DATA MODEL

In this section, we formally develop the MCT logical data model, which we motivated and illustrated using examples in the previous

section. MCT is an evolutionary extension of the XML data model of [14], and, hence, is presented as such. As we shall see in the next section, this evolutionary approach allows us to build on query and update languages proposed for XML to obtain manipulation languages for MCT databases.

3.1 Multi-Colored Trees

Nodes in the XML data model are organized in a data tree, which defines a *global document order* of nodes, obtained by a pre-order, left-to-right traversal. Every XML data model value is a sequence of zero or more items, where an *item* is either a node or an atomic value. The *multi-colored trees* (MCT) data model enhances the XML data model in two significant ways:

- Each node has an additional property, referred to as a *color*, and nodes can have one or more colors from a finite set of colors.
- A database consists of one or more colored trees, where each node in has color (as one of its colors).

More formally, we have:

DEFINITION 3.1. [Colored tree] Let be a finite set of nodes of the seven kinds defined by the XML data model, and be a finite set of colors. A colored tree $T = (N, E, C)$, where (i) The set of nodes N ; (ii) The set of edges E ; (iii) The set of colors C , defines an ordered, rooted tree, satisfying the tree relationships imposed by the XML data model between the different kinds of nodes, with a

```

dm:parent($n as Node,
  $c as xs:string) as Node?
dm:string-value($n as Node,
  $c as xs:string) as xs:string?
dm:typed-value($n as Node,
  $c as xs:string) as AtomicValue*
dm:children($n as Node,
  $c as xs:string) as Node*

```

Figure 4: Modified node accessors

triple $\langle n, p, l \rangle$ specifying that node n has p as its parent and l as its left sibling.¹ ■

Essentially, a single colored tree is just like an XML tree. Allowing for multiple colored trees permits richer semantic structure to be added over the individual nodes in the database.

DEFINITION 3.2. [MCT database] A multi-colored tree (MCT) is defined as a triple $\langle T, C, A \rangle$, where (i) each T_c , $c \in C$, is a colored tree; (ii) $C = \{c_1, \dots, c_k\}$; and (iii) each attribute, text and namespace node n associated with an element node e in any of the colored trees has all the colors of e , and has e as its parent node in each of its colored trees.

An MCT is said to be an MCT database if the root of each of its colored trees is the same document node (which, hence, has all colors in C), else it is an MCT database fragment. ■

Intuitively, in an MCT database (fragment), a node belongs to exactly one rooted colored tree, for each of its colors. This is similar to the XML data model, where a node can belong to exactly one rooted tree. Unlike an XML database, however, there is no global document order of nodes in an MCT database: each colored tree defines its own local order of nodes, obtained by a pre-order, left-to-right traversal of the nodes in the colored tree.

3.2 Node accessors

In the XML data model [14], ten accessors are defined for all seven kinds of nodes. Four of these accessors, namely, `dm:parent`, `dm:string-value`, `dm:typed-value`, and `dm:children`, would need to be extended to take a color into account. Their signatures are given in Figure 4. If the node on which these accessors are called does not have the color that is passed as an argument to the accessor, an empty sequence is returned. Otherwise the node and the accessor are said to be *color compatible*, and the desired result is returned from the appropriate colored tree.

The other six accessors defined in the XML data model, namely, `dm:base-uri`, `dm:node-kind`, `dm:node-name`, `dm:type`, `dm:attributes`, and `dm:namespaces`, are not influenced by the color of the node, and continue to have the same signature and meaning as in the XML data model.

In addition, a new accessor needs to be defined on all node kinds to determine the colors of a given node:

```
dm:colors($n as Node) as xs:string+
```

3.3 Node Constructors

In the XML data model, each node kind defines its constructors, which always return a new node with unique identity. This is feasible since the nodes can be constructed iteratively, in a bottom-up fashion in the XML tree. In our MCT data model, it is not always

¹We use the convention $\langle n, p, l \rangle$ to identify node n with parent p , and no left sibling.

possible to construct a node only after all its children in *each* of its colors have been constructed, e.g., element node n may be a child of element node p in one color, but a parent in a different color. To effectively permit the construction of multi-colored trees, we define two different types of constructors for each node kind.

- *First-color* node constructors are like constructors in the XML data model, except that they are extended to take a color into account, and return a new node with unique identity.
- *Next-color* node constructors take a previously constructed node, and add a color and the tree relationships in that color; the same node is returned.

Example constructor signatures for the element node are depicted in Figure 5. Note that the signature of the next-color constructor is somewhat smaller than that of the first-color constructor, since one does not need to repeat some of its properties, and its attribute and namespace nodes.

The MCT logical data model defines allowable syntactic structures. The semantics of the database are captured by its schema. The XML schema language proposed by the W3C deals with both structure [27] and datatypes [6]. While we briefly use MCT schemas in Section 5, formally extending XML schema to multi-colored trees is an interesting direction of future work.

3.4 Shallow and Deep

We conclude this section by characterizing the intuitive notions of shallow and deep XML trees the we have used in our examples.

We call an XML schema *shallow* provided it is in XNF, as defined in [3]. More precisely,

DEFINITION 3.3. [Shallow, Deep Schemas] Let $\langle DTD, F \rangle$ be a schema, where DTD is a DTD and F is a set of functional dependencies. Then, $\langle DTD, F \rangle$ is shallow provided for every non-trivial functional dependency $f: X \twoheadrightarrow Y$, X or Y is not a path from the root. $\langle DTD, F \rangle$ is said to be deep if it is not shallow. ■

It is easy to verify that the Deep-1 and Shallow-1 trees used in Example 1.1 indeed satisfy the above definition. Note that this definition permits schemas with non-trivial hierarchies to be characterized as shallow. Further, a shallow schema is not necessarily unique, e.g., a schema with a non-trivial hierarchy can always be flattened (using ID-IDREFS), while preserving its shallowness.

4. DATA MANIPULATION LANGUAGES

We now formally develop the MCXQuery logical query language, which we motivated and illustrated using examples in Section 2. The MCT data model, being an evolutionary extension of the XML data model, allows us to naturally build our logical query language as an extension to XQuery [7].

4.1 MCXQuery Path Expressions

An XQuery *path expression* can be used to locate nodes in tree-structured XML data. Here we discuss only the *unabbreviated* syntax for path expressions; abbreviated syntax can be developed for some expressions (as used in examples early in the paper).

Examples of (unabbreviated) XQuery path expressions include:

```

document("mdb.xml")/child::movie-genre
descendant::movie-genre[name = "Comedy"]
$c/parent::node()

```

```

dm:element-node($sname as xs:QName, $nsnodes as NamespaceNode*, $attrnodes as AttributeNode*,
  $children as Node*, $type as xs:QName, $color as xs:string) as ElementNode
dm:element-node($self as ElementNode, $children as Node*, $color as xs:string) as ElementNode

```

Figure 5: Modified and new node constructors

In the MCT logical data model, a node may have multiple colors, in which case it would belong to multiple colored trees. Hence, an axis and a node test specification (e.g., `parent::node()`) does not suffice to *uniquely* identify the navigation to be performed in a single step, from a context node. For example, in the MCT database of Figure 2, the `movie` node `RG012` has two parent nodes: a `movie-genre` node in the red tree, and a `movie-award` node in the green tree. However, since a node belongs to exactly one rooted colored tree, for each of its colors, augmenting the specification of a step by a *color* would serve to provide the necessary disambiguation.

We achieve this by enclosing the color specification in curly braces, preceding the axis specification in the step expression, e.g., `[[red]]descendant::movie`, `[[blue]]child::movie-role`. The extensions to the relevant productions in the grammar of XQuery are shown in Figure 6. In general, different steps in an MCXQuery path expression may have different color specifications, and the resulting navigation over the MCT database can be quite sophisticated. The result of evaluating an MCXQuery path expression is, as before, a sequence of items, *all* of which have the same color, as determined by the color specification of the final step in the path expression. The order of items in the result sequence is determined by their local order in the corresponding colored tree.

Figure 3 presents a few illustrative path expressions in MCXQuery, with each step augmented by a color specification. Query Q4, in particular, illustrates the use of different color specifications in different steps of the path expression.

4.2 MCXQuery Constructor Expressions

XQuery provides constructor expressions that can create XML tree structures within a query, based on the notion of constructors for the different node kinds in the XML data model.

When the name of the element to be constructed is a constant, the element constructor is based on standard XML notation. *Enclosed expressions*, delimited by curly braces (to distinguish them from literal text),² can be used inside constructors to compute the *content* of the constructed node, and also its attributes.³ Enclosed expressions are evaluated and replaced by their value (which may be any sequence of items). For example, the `return` clauses of the `Deep-1` and `Shallow-1` queries in the introduction have constructor expressions with enclosed expressions.

Since every tree in the MCT logical data model is a colored tree, XQuery constructor expressions are suitable for the creation of new colored trees in an MCT database as well. One such constructor expression could be used for the creation of each colored tree, and an MCT database/fragment could be created using multiple constructor expressions. One key issue needs to be resolved, however. The result of an element constructor in XQuery is always a new ele-

ment node, with its own identity; all the attribute and descendant nodes of the new element node are also new nodes with their own identities, even though they may be copies of existing nodes.

Always creating a *new* node is inappropriate for constructor expressions in MCXQuery, since such a node would have a different identity from existing nodes in the MCT database, limiting the capability of MCXQuery constructor expressions in creating MCT databases/fragments, where nodes belong to multiple colored trees. To effectively permit the construction of multi-colored trees, MCXQuery constructor expressions need the ability to *reuse* existing nodes and their descendants, in addition to being able to create element nodes with new identities. This is achieved as follows.

- When an enclosed expression is evaluated, its value (a sequence of items) *retains* the identities of nodes in the sequence, instead of creating copies by default. This is similar to the behavior of MCXQuery path expressions.
- To create node copies, MCXQuery provides a function named `createCopy`. The `createCopy` function takes any sequence of items as its argument, and returns copies of the items in the sequence, in the same order.

For example, the result of evaluating the enclosed expression in the `return` clauses of queries Q1, Q2 and Q3 in Figure 3 would contain the node with identity `RG015`, since identities are preserved when the enclosed expression is evaluated. If, however, the `return` clause contained the constructor expression:

```

<m-name>
  createCopy( [[ $m/[[red]]child::name ]] )
</m-name>

```

the result would contain a new node, with a different identity.

To associate a color with the result of a constructor expression, MCXQuery provides a function named `createColor`. This function takes two arguments: a color literal as its first argument, and any sequence of items as its second argument. It *adds* the specified color to the set of colors associated with each node in its second argument.

Finally, we address an interesting issue that arises if node identities are retained when evaluating enclosed expressions in a constructed expression, especially when this result is colored. Since a node can be present at most once in any colored tree, the result of any constructed expression *should not* have a node (with a given identity) occur at more than one position in the colored tree. Such a situation can arise, as the following constructed expression illustrates:

```

createColor(black, <dupl-problem>
  <m1> [[ $m/[[red]]child::name ]] </m1>
  <m2> [[ $m/[[red]]child::name ]] </m2>
</dupl-problem>)

```

In this case, the expression raises a dynamic error. Note that such a situation doesn't arise if the `createCopy` function is appropriately used.

²Note that the use of curly braces for enclosed expressions does not conflict with their use to specify color when navigating steps in path expressions.

³A special form of constructor called a computed constructor can be used in XQuery to create an element node or attribute node with a computed name or to create a document node or a text node. These can also be extended, in an analogous fashion, for MCXQuery.

```

85: ForwardStep ::= (Color ForwardAxis NodeTest) | (Color AbbreviatedForwardStep)
86: ReverseStep ::= (Color ReverseAxis NodeTest) | (Color AbbreviatedReverseStep)
151: Color ::= ("||" Literal "||")

```

Figure 6: Productions for MCXQuery path expressions

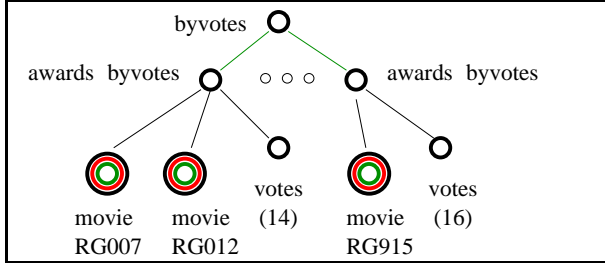


Figure 7: Result of evaluating Q5

4.3 XQuery Expressions

We present an example depicting how an MCXQuery expression can be used to add a new colored tree, consisting of new nodes and previously existing nodes, to an MCT database. Consider query Q5 from Figure 1. The MCXQuery expression is shown in Figure 3. The result of evaluating this expression against the MCT database of Figure 2 is shown in Figure 7. Notice that `movie` nodes now have three colors: red (because of their participation in the `movie-genre` hierarchy), green (because of their participation in the `movie-award` hierarchy), and black. All other nodes, including the newly created `votes` nodes in the result, are only black. Note that the result is a tree since each `movie` in the green `movie-award` hierarchy has only one child element named `votes`.

There is as yet no standard for specifying updates to XML data. In [25], the authors propose an extension to XQuery (using `for`, `let`, `where` and `update` clauses) to perform updates to XML documents. It is easy to see that the MCXQuery extensions to XQuery path expressions and constructor expressions, described previously, can be used in a straightforward manner in conjunction with the update extensions of [25], to unambiguously update an MCT database. Each of the update operations can be performed on *existing* colored trees, once the tuple of bindings is returned. Note that update operations implicitly add existing colors to new nodes, or to existing nodes. Creating *new* colored trees is done via extensions to the constructor expressions in MCXQuery.

5. SERIALIZATION OF MCT DATABASES

While the MCT logical data model is the basis for the query and update languages, and the MCT physical data model (discussed in Section 6) is the basis for storage of the data values, these are not appropriate for *exchanging* information in a *flexible* manner, which is crucial in today's networked world. What is needed is an *exchange data model*, which deals with the serialization of data values for exchange between applications.

Regular XML is the de facto standard for data exchange. So we need to develop a (serialized) XML representation of an MCT database, such that the original MCT database can be reconstructed efficiently from the serialized representation at the receiver's end. In addition, we would like this serialization to be compact.

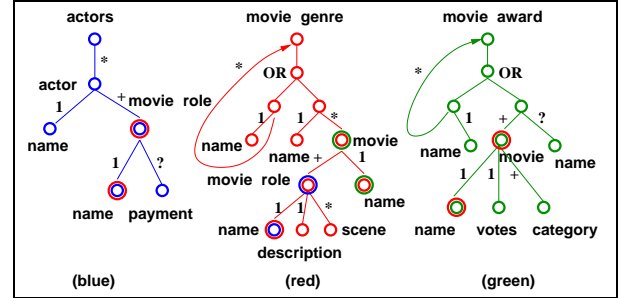


Figure 8: Example MCT Schema

5.1 Real and Primary Colors

Consider our running example database of Figure 2. For illustration, suppose movie elements additionally have category subelements in the green hierarchy, `movie-role` can have `payment` subelement in the blue hierarchy and `description` and `scene` subelements in the red hierarchy. The corresponding schema is illustrated in Figure 8. For each element node type, call the colors (hierarchies) in which it appears in the MCT database its *real* colors. Call the hierarchy (color) in which an element is represented in a serialization, its *primary* color w.r.t. the serialization. So, movie elements have red and green as real colors and these are their only possible primary color choices. Not every movie element need be both red and green in an instance but ignore this for now: we will revisit this point. The name subelement of `movie` has the same primary color choices, although its choice is determined by that of its parent `movie` element. The `movie-role` element can have blue or red as its primary color, since it is present in those hierarchies. But, surprisingly green is also a primary color choice for `movie-role`, even though it is not its real color. This is because when a `movie` element's primary color is chosen to be green, we have the option of using the same color as the primary color for all its subelements in *any* colored hierarchy, regardless of *their* real colors. In our example, some of the options for primary color choices are: (i) blue for `movie-role`, `movie-role/name`, `payment`, `description`, and `scene`, red for `movie`, `movie/name`, `votes`, and `category`; (ii) red for `movie` and all its descendants⁴; (iii) blue for `movie-role` and all its descendants in any color, and green for `movie` and its immediate subelements in any color, except `movie-role`; and (iv) green for `movie` and all its descendants in any color.

5.2 Cost-based Serialization

To serialize using option (i), we need to associate a `movieId` attribute with each (red) `movie` node and a `movieIdRef` attribute with each (blue) `movie-role` node. Similarly, other parent pointers have to be set up, as suitable IDREFs. In addition to setting up such parent pointers, we need to associate a color attribute with certain elements in the serialization. The color attribute

⁴We can traverse multiple colors, so `payment` is considered a descendant of `movie` for this purpose.

type is a set of strings of the form $\langle \text{color} \rangle \langle \text{type} \rangle$, where $\langle \text{color} \rangle$ is a string representing a color and $\langle \text{type} \rangle$ is one of movie , role , and is optional. If the color attribute of an element e contains the value blue it means e as well as its entire subtree in the serial representation have color blue (in addition to others). Similarly, $\text{blue} \rightarrow$ says the relevant subtree is not blue, whereas $\text{blue} \rightarrow$ only affects e 's color. These color denotations can override: e.g., $\text{blue} \rightarrow$ at a descendant of e whose color includes $\text{blue} \rightarrow$ overrides the latter and says the descendant has color blue. In finding a serialization of minimal expected cost, we have to account for the cost of such encoding of color information in addition to parent pointer setup. One additional piece of information that is relevant for deciding the cost of a particular primary color choice for an element is the average number of its children of each element type. For example, each *movie-role* may have only one name and description but, say, 3 scenes on an average. We assume statistical summary information of this kind is available. Let $\text{cost}(e, c)$, for element type e and color c , represent the cost of choosing c as the primary color for e . For our running example, $\text{cost}(\text{movie}, \text{red})$ can be calculated as:

```

cost(movie, red) = 1 * name + 1 * votes + 1 * category +
1 * movie-role

```

The 1 's represent the average number of each type of subelement for a movie element. The 1 's accounts for the inclusion of color = red for non-red subelements. The last 1 's accounts for setting up the parent pointer (as an ID/IDREF) for a movie element in the green hierarchy. In general, for each element type, the minimal cost, among possible primary color choices for each element, as well as the best color choice itself can be determined using a dynamic programming approach.

5.3 Optimal Serialization

In this paper, we consider only serializations where one of the colors present in the original MCT database is chosen as the primary color for any element. Furthermore, we assume for simplicity that elements that are multi-colored are not involved in any cycles in the schema, and that for each multi-colored element type, there is only one production in its schema grammar. Even then, determining the optimal serialization is non-trivial. The optimal serialization algorithm is given in Figure 9. The algorithm uses a helper function $\text{cost}(e, c)$ that for an element type e and color c , returns the average number of children of type e for an element corresponding to e 's parent type in the hierarchy with color c . For example, $\text{cost}(\text{movie-role}, \text{red}) = 10$ means on an average, a movie has 10 movie-roles. The serialization scheme itself can be obtained by running the algorithm and then associating with each element of a given type, the best primary color choice found by the algorithm, and then following the ideas described earlier regarding setting up of parent pointers and color attribute values for various nodes.

Finally, recall so far we have assumed whenever an element type has multiple colors, all its instances will appear in each of its colored hierarchies. This is not always true; e.g., some movie elements may not appear in the green hierarchy since they were not nominated for an Oscar. Our algorithm can be easily extended to this case by noting: (i) the calculations and book-keeping used to determine the best primary color choice, for an element type, can be used to maintain a ranked set of color choices from best to the worst, and (ii) whenever an actual element of a given type does not appear in the current primary color choice for that type, use

```

Algorithm
Input: An MCT schema, together with stats
Output: Optimal serialization scheme
for (each color  $c$ )
  identify, proceeding top-down, the
  multi-colored element types;
  for (each such element type  $e$ )
    find  $\text{cost}(e, c)$ ,  $\text{cost}(e, \text{red})$ ,  $\text{cost}(e, \text{green})$ ,  $\text{cost}(e, \text{blue})$ ;
    pick the  $\text{cost}(e, c)$  with the least cost;
end Algorithm

function  $\text{cost}(e, c)$ ;
Input: element type  $e$  and color  $c$ , one of its
legal primary color choices;
Output: cost of choosing  $c$  as  $e$ 's primary color;
if ( $c$  is a leaf)
  if ( $c$  is a leaf)
    if ( $c$  is a leaf)
      else if ( $c$  is a child of a node whose
        color includes  $c$ )
          return  $\text{cost}(e, c)$ ;
    else
      return  $\text{cost}(e, c)$ ;
  //parent pointer setup cost for other colors;
  for (each color  $c'$ )
    let  $n_1, n_2, \dots, n_k$  be  $e$ 's production in color  $c'$ ,
    where  $n_i$  is 1, ?, +, or *;
    for (each  $n_i$ )
      let  $c_i$  be the primary color choice  $c'$  with min.
      cost for  $n_i$ , subject to the constraint that
       $c_i$  choice is  $c$ ;
    return  $\text{cost}(e, c)$ ;
end function

function  $\text{optSerialize}(e, c)$ ;
Input: given  $e$  parent  $p$  has  $c$  as primary
color choice, find best primary color choice  $c'$  for  $e$ ;
set  $c'$  to that color  $c'$  that minimizes the cost  $\text{cost}(e, c')$ ;

```

Figure 9: Algorithm optSerialize

the next best choice for that element and proceed iteratively for its subelements. We omit details. We can show:

THEOREM 5.1 (OPTIMALITY OF SERIALIZATION). *Let $\langle \text{MCT} \rangle$ be an MCT schema together with summary information for each color, of the average number of child elements of each type for each multi-colored parent element type. Then the serialization scheme found by Algorithm optSerialize is optimal w.r.t. $\langle \text{MCT} \rangle$.*

6. IMPLEMENTATION

There are many physical data models currently being investigated for storing XML data, including relational and native approaches. First, we briefly discuss how these could be enhanced for the physical representation and manipulation of an MCT database. Next, we present greater detail with regard to the specifics of a physical structure we have implemented.

6.1 Physical Model

One popular technique for the physical representation of XML data is to map the data to an existing (relational) database system. Several mapping techniques have been proposed (see, e.g., [16, 23, 30, 26]) to map tree-based XML data to flat tables in a relational schema. Due to the heterogeneity of XML data, a simple XML schema often produces a relational schema with many tables. Structural information in the tree-based XML schema is

modeled by joins between tables in the relational schema. Two main strategies have been proposed for this purpose. First, one could use primary-key foreign-key joins in relational databases to model the parent-child relationships in the XML tree. Second, one could use a (start, end, parent-start) interval encoding or a Dewey-style encoding of each node in an XML tree, as the node's key, represented in one or more attributes of the relation, to more directly determine relationships like ancestor-descendant and preceding-following between nodes in the XML tree.

Since an MCT database consists of multiple colored trees, each of which is akin to an XML tree, relational approaches for the physical representation of XML are easily extended to handle MCT databases. Essentially, an MCT node content can be fragmented into relations as with XML data. The structural participation of a node in multiple colored trees can be represented using foreign-keys, (start, end, parent-start) interval encodings, or Dewey-style encodings, *for each colored hierarchy, separately*.

There are also several native XML databases, where the physical representation and manipulation of XML data is independent of relational databases (see, e.g., [22, 18, 15]). Such native XML database systems store XML data directly, retaining its natural tree structure, but often take recourse to the previously mentioned encodings for node identification, especially for indexing purposes. Two choices suggest themselves for the native physical representation of MCT databases. First, one can rely on mapping MCT databases to XML (as discussed in Section 5), and then store this using the native XML systems. Alternatively, one could use the native XML system for separately storing each colored tree. Where one element appears in multiple colored trees, straightforward data structures can be used to link the multiple occurrences of this element, once in each tree, and to minimize the amount of data replicated across these multiple occurrences.

6.2 Our Implementation

We modified the Timber [18] database system to implement the MCT data model. In this system, element content and attributes are stored separately from the element structural relationships. Thus, each (traditional) XML element is represented as one structural node, a separate content node, if the element has content, and an attribute node, if the element has attributes. Sub-elements have a similar representation of their own, and are merely linked to their parent through the structural node.

For a multi-colored element, the content and attribute nodes remain the same as before. However, we create one structural relationships node for each color hierarchy that the element participates in. Since the structural relationships for any one color are no different than in the single color case, no modification is required to the representation of the structural relationship node, or to the manner in which nodes are indexed.

With the design just described, we have an effective representation of multi-colored elements, but with one critical shortcoming: there is no way to determine the multiple colors of a given node. Given a particular structural node, we can obtain its attribute and content parts. But the XML database system we had did not provide a means to navigate in the opposite direction. We addressed this by introducing additional attributes for multi-colored nodes that provide links back to each of the corresponding single-colored structural nodes. The physical structures corresponding to a portion of the logical data of Figure 2 are depicted in Figure 10.

Each MC-query is decomposed into components that have a single color. Each single color query evaluation proceeds in the normal manner. A color transition is accomplished by a *cross-tree join* access method, which simply follows the links described above to

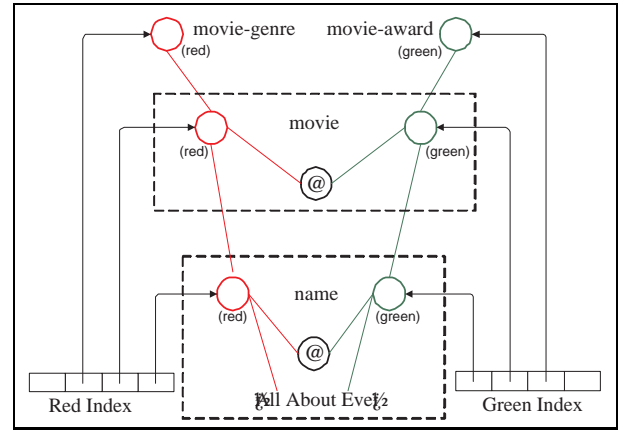


Figure 10: Physical Model

obtain the structural node of each element for the color being transitioned to. This bulk access method is implemented in a straightforward fashion as an attribute-value based join. Since color transitions are not free, alternative plans are often worth considering. For example, we could choose to evaluate multiple single-color queries first, and perform cross-tree joins at the end, minimizing the cost of the cross-tree join. Alternatively, it may be preferable to perform a single-color query, then a cross-tree join, before evaluating the next single-color query, to benefit from a selection that greatly reduces the size of the latter computation. Such choices must be evaluated by a query optimizer in choosing a good plan. While we do not anticipate any significant new challenges on account of having one more type of join operator, the query optimizer design is beyond the scope of this paper. For all the experimentation described next, we manually specified the query plan, always choosing the one expected to be the best.

7. EXPERIMENTAL EVALUATION

Whereas the primary motivation for the multi-colored model is ease of data modeling, it is frequently the case that better data models lead not just to simpler query specification but also to superior performance. To test whether this is the case for MCT, we performed an extensive experimental analysis, using the implementation described above. All experiments were performed on a single processor Pentium III 866MHz equipped with 512 Mbytes of memory, 30Gbytes of disk storage and Windows 2000 operating system. The buffer pool size was set to 256Mbytes and data page size configured to 8Kbytes.

We used two popular data sets drawn from very different domains: TPC-W and SIGMOD-Record. The TPC-W benchmark [28] has been converted to XML format by XBench [29] using ToX-gene [4]. The workload simulates an Internet commerce-oriented transactional web server. We cannot directly use the XBench conversion of TPC-W because it is not multi-colored. Instead, we used the same tool, ToxGene, to generate TPC-W data in a multi-colored schema of our design. As a baseline, we also generated the same data in a shallow tree schema and in a deep tree schema. The shallow tree schema is a minor enhancement of the schema used by XBench. The deep tree schema places *customer* at the top level of the hierarchy, then *order*, *address*, *country*, *item*, and finally *author*. The MCT schema is comprised of 5 single-colored hierarchies:

- *customer--order--orderline*,

		MCT	Shallow	Deep
TPC-W	Num. Elements	1,502,357	1,502,357	3,883,320
	Num. Attrs	153,713	153,713	339,674
	Content Nodes	1,295,818	1,295,818	3,307,589
	Data MBytes	786.27	329.02	893.09
	Index MBytes	520	215	538
SIGMOD-Record	Num. Elements	112,408	112,407	125,403
	Num. Attrs	110,086	110,086	111,961
	Content Nodes	108,823	108,823	118,202
	Data MBytes	103.81	88.05	152.95
	Index MBytes	29.7	18.7	20.5

Table 1: Storage Requirement

- billing address--order--orderline,
- shipping address--order--orderline,
- date--order--orderline, and
- author--item--orderline.

SIGMOD-Record is first scaled up by a factor of 100 (from 600KB to 60MB), and then dealt with in the same manner. Its MCT schema is comprised of 2 colored hierarchies:

- date--issue--articles, and
- editor--topic--articles.

The shallow tree schema has 3 trees: articles, date--issue, and editor--topic. The deep tree schema is a minor enhancement of the original schema.

Each experiment was run five times. The lowest and highest readings were ignored and the other three were averaged. For our experiments, we constructed an index on element tag name and attribute id. We also constructed indices on element content and attribute value, where needed.

7.1 Storage Requirements

First, we compare storage requirements of the three approaches. Table 1 shows the numbers. As expected, the deep tree approach has many more elements and requires considerably greater storage due to its replication of data. The MCT approach has exactly the same number of elements as shallow, but requires storage that is greater than shallow but less than deep. The reason for this is that each multi-colored element is physically stored as multiple structural nodes (see Figure 10), one for each color, with an attendant overhead for this storage.

Looking at the sizes of indices created, we find similar trends. Since it is the structural nodes that are most interesting to index, the size of index in the case of MCT is comparable to deep and much larger than for shallow.

7.2 Query Processing Time

Table 2 shows the execution time in seconds of 20 queries from the TPC-W Xbench workload and 7 queries from the SIGMOD-Record workload. In addition, it also shows times for a few update statements that we defined.⁵

We ran these experiments for a range of buffer pool sizes, and found no significant differences in the trends for the results obtained. As such, we report results for only on a buffer size of 256

⁵All queries will be made available on a website as supplemental data upon paper acceptance.

Query	Results	MCT	Shallow	Deep	Colors	Trees
TQ1	1	0.12	0.11	0.12	1	1
TQ2	719	0.60	0.60	0.61	1	1
TQ3	4	0.82	0.83	0.16	2	2
TQ4	726	0.37	0.39	0.38	1	1
TQ5	3	0.05	0.05	0.05	1	1
TQ6	3244	1.79	1.72	1.81	1	1
TQ7	58	0.02	0.01	112.25	1	1
TQ7D	44929			2.79	1	1
TQ8	1	0.35	0.35	0.67	1	1
TQ9	5110	0.55	30.16	0.76	1	2
TQ10	90	6.61	8.96	0.71	2	2
TQ11	63	0.23	9.68	0.25	1	2
TQ12	1	0.01	0.01	0.54	1	1
TQ12D	3			0.54	1	1
TQ13	2893	0.11	2.36	0.23	1	2
TQ14	253	0.09	2.29	0.25	1	2
TQ15	97	0.72	38.11	1.34	1	2
TQ16	92	0.40	20.09	34.61	1	2
TU1	1	0.01	0.02		1	1
TU1D	335			3.18		
TU2	1	0.03	0.02		1	1
TU2D	5			0.19		
TU3	22	0.36	15.14	0.65	1	2
TU4	1	0.12	0.49		1	2
TU4D	10			0.33		
SQ1	1	0.01	0.01	0.01	1	1
SQ2	3	0.02	0.91	0.02	1	2
SQ3	20	0.02	10.32	0.02	1	2
SQ4	6	0.01	0.01	0.30	1	1
SQ4D	1994			0.13		
SQ5	84	0.01	3.11	0.01	1	2
SU1	5	0.01	0.01		1	1
SU1D	25			0.04		
SU2	1	0.01	0.01		1	2
SU2D	7			0.05		

Table 2: Query Processing Time in Seconds. The first letter of the query label indicates the data set used: T=TPC-W, S=SIGMOD-Record. The second letter indicates query type: Q=Read-only, U=Update. The results column indicates the number of results produced for a read-only query, and the number of elements updated for an update query.

Megabytes. We also ran experiments for a range of data-set sizes and found that most of the times scaled linearly with data set size. The only exceptions were the two queries involving an inequality value join, which is implemented as nested loops, and hence has a quadratic dependence on data set size. Once more, in the interests of space, we report numbers only for the full size data set.

We repeated our experiments under both cold cache conditions (by flushing all buffers completely before each query evaluation) and warm cache conditions (where a first time execution of the query is allowed to populate the buffer for subsequent warm cache executions). The trends were similar in both cases. We choose to report numbers here for the warm cache case since the differences stand out more in the cold cache case, even a query plan that is very good about managing memory pays at least some penalty for getting data into the buffer, and this penalty is more closely related to the size of data stored than to the locality/quality of the query plan.

Overall, one can immediately see that MCT in all cases is either comparable to shallow or substantially faster. However, deep seems to have a large variance in performing much better some times, and much worse at others.

Additional annotations in Table 2 help to clarify the picture. For each query we have indicated the number of different trees in-

volved, indicating the number of value joins that were required by shallow, and the number of colors involved, indicating the number of color transitions required by MCT. We observe that structural joins are substantially cheaper to evaluate than value joins, with color crossings having a cost only slightly less than that of a value join in our implementation. (A more sophisticated implementation could bring down the cost of a color crossing substantially but that is only speculation on our part at this time.) No value joins are required for the MCT and deep representations. Obviously, the concept of color crossings only applies to MCT. The relative cost of a query is immediately determined by the number of value joins or color crossings. When there are not any, shallow and MCT have comparable performance, and are never beaten by deep. When there are value joins or color crossings, performance suffers. MCT beats shallow precisely in the cases where it does not need a color crossing, because it was able to fold the hierarchy relevant to the query into a single color, whereas shallow had to join trees. Since MCT has multiple colors available to it, it is indeed possible to have multiple hierarchies that could each be the one most appropriate for a query, and choose one of them a priori as part of database design.

Furthermore, with value-joins, the total running time of the query is very sensitive to the sizes of inputs to the join. Consider TQ9 and TQ11, which are similar, except that the former computes a larger join (input sizes to the join are 5110 and 10000) than the latter (with input sizes 33 and 25912). The final result cardinality of the former is also correspondingly higher. The running time of the shallow tree query is dominated by the value join, and grows linearly with the size of the join. In contrast, the running times of deep and MCT change very little between these two queries.

The queries where deep does poorly all involve duplicate results. (The other queries happened to have no duplicates, due to the schema specifications.) Deep not only has the cost of retrieving more (duplicate) results, but it also has to perform costly duplicate elimination afterwards. To tease these two factors apart, for each of these queries, we report two versions for deep, with and without duplicate elimination. (Queries run without duplicate elimination are marked with a *DU* at the end in Table 2.) The conclusion is that any one of these factors is enough to render deep uncompetitive $\frac{1}{2}$ the two together just compound the difficulty.

TQ16 is particularly interesting since it both requires value joins in shallow and also generates duplicate intermediate results in deep. In consequence, MCT is able to perform better than both shallow as well as deep. Note that TQ16 includes grouping with duplicate elimination as part of the query specification: since the duplicates are in intermediate results rather than in the final results, we are not able to define and run a separate TQ16D query to measure the performance of deep without duplicate elimination.

The update queries showed trends similar to read-only queries. Where the update specification was simple, and there were no duplicates, all three schemes performed comparably. Once duplicates are involved, the performance of deep suffers because of having to update multiple copies. When the update specification is complex enough to require a value join to identify the nodes to be updated, shallow takes a performance hit.

7.3 Query Simplicity

We have looked at performance metrics above, including costs for storage, for queries, and for updates. But for a data model, perhaps a more important metric is query simplicity. A central goal of a good data model should be to make it easy to express complex queries. While simplicity itself is hard to quantify, we have identified several metrics that are likely to be correlated with the number of path expressions and the number of variable bindings. For each

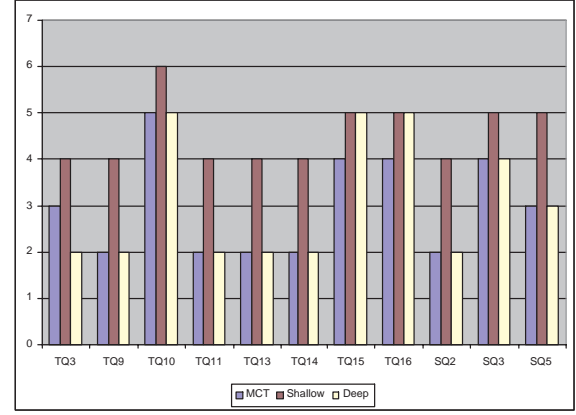


Figure 11: Query Specification Complexity: Number of Path Expressions

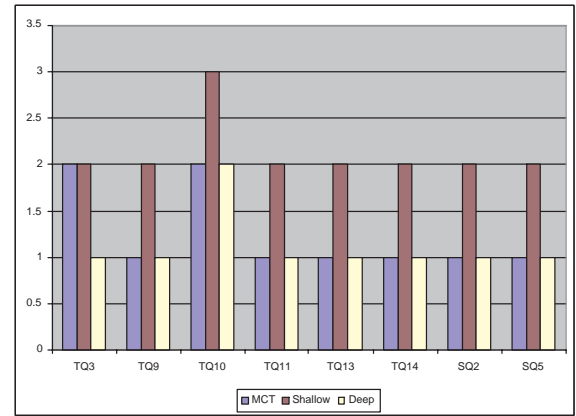


Figure 12: Query Specification Complexity: Number of Variable Bindings

of the TPC-W queries we studied above, we present these metrics in Figure 11 and Figure 12; queries that result in identical numbers for all three strategies are not reported. Our conclusion is that MCT and deep are comparable, with the equivalent shallow tree query being quite a bit more complex. The reason is that structural traversals are succinctly expressed in XPath (and XQuery) whereas value-based joins, as required by shallow, require the definition of multiple variables and the introduction of a predicate in the WHERE clause stating the join condition.

8. RELATED WORK

Graph-based models have been investigated for data modeling in depth, initially in a general context (e.g., see GraphLog [11], Hy+ [10]), and more recently, for semi-structured data (e.g., see StruQL [13], UnQL [9], Lorel [1]). Indeed, there have been recent proposals to even extend the already expressive and powerful semi-structured data models (e.g., see [12, 24]). Our work is distinct from all of these in that on a per-color basis, our model is tree-based, with all the simplicity and performance benefits trees have to offer. Besides, our extension is specifically set in the context of XML. Indeed, we have discussed at length how the basic XML model and query language syntax (XQuery) can be extended to take advantage of multiple colored hierarchies.

Data warehouses typically provide support for multiple hierarchies, one (or more) for each dimension (see, e.g., [17, 19, 20]). A data warehouse schema consists of one or more fact tables and a number of dimension tables. The latter model hierarchical relationship among members of a dimension: e.g., *coke* and *pepsi* are children of *soda*, while *soda* and *juice* may be children of *softdrinks*. A tuple in a fact table can be thought of having a presence in each dimension for which it has values. However, to our knowledge, none of the works in data warehousing/OLAP leverage this perspective in any formal way.

Finally, Pedersen et al. [21] are investigating the integration of OLAP technology with XML data. However, their main concerns are modeling cost and query optimization in the context of providing OLAP-style functionality for heterogeneous XML data. Given the applicability of the MCT data model for XML data warehousing and OLAP, their work neatly complements ours.

9. CONCLUSIONS

We have developed a multi-colored tree model, which eases the restriction of developing a single hierarchy over data to be represented in XML. We described how this logical data model could be specified using only evolutionary extensions to the XML data model and to XQuery. Given the importance of exchanging data, we presented an algorithm to obtain a size-optimal serialization of data represented in our model, rendering it in pure XML. We also discussed the changes necessary to XML databases to be able to support multi-colored trees, and describe an implementation on top of the Timber native XML database. Finally, an experimental evaluation, using our implementation, demonstrates the many advantages of MCT over shallow and deep XML trees.

The multi-colored tree model proposed in this paper has three major benefits:

- **Ease of Schema Design:** The hard choices required for a deep tree design (i.e., which element to put below which) are made easier by permitting multiple hierarchies to co-exist over the same data. An added benefit is that, like shallow, MCT can avoid update anomalies.
- **Ease of Query Specification:** XQuery (and XPath) make it much easier to specify hierarchical structural navigation than value-based or ID-IDREF joins. Multi-colored trees help avoid the latter and use the former instead.
- **Efficiency in Query Processing:** Structural (containment) joins are much cheaper to compute than value-based (or pointer) joins. This makes it much cheaper to evaluate MCT queries compared with equivalent single-color queries that require value-based or ID-IDREF joins. Another way of thinking about this is that more pre-computed structural paths are available (as opposed to ad hoc join paths).

10. REFERENCES

- [1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1996.
- [2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE*, 2002.
- [3] M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. PODS*, 2002.
- [4] D. Barbosa, A. Mendelzon, J. Keenleyside and K. Lyons. ToXgene: A Template-based Data Generator for XML. In *Proc. Fifth Intl. Workshop on the Web and Databases (WebDB 2002)*. Madison, Wisconsin - June 6-7, 2002
- [5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (XPath) 2.0. W3C Working Draft. Available from <http://www.w3.org/TR/xpath20/>, Nov. 2002.
- [6] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes. W3C Recommendation. Available from <http://www.w3.org/TR/xmlschema-2/>, May 2001.
- [7] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C Working Draft. Available from <http://www.w3.org/TR/xquery>, Nov. 2003.
- [8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, 2002.
- [9] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. SIGMOD*, 1996.
- [10] M. Consens and A. Mendelzon. Hy^{III}: A hygraph-based query and visualization system. In *Proc. SIGMOD*, 1993.
- [11] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proc. PODS*, 1990.
- [12] C. E. Dyreson, M. H. Böhlen, and C. S. Jensen. Capturing and querying multiple aspects of semistructured data. In *Proc. VLDB*, 1999.
- [13] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):411, Sept. 1997.
- [14] M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model. W3C Working Draft. Available from <http://www.w3.org/TR/query-datamodel/>, Nov. 2002.
- [15] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4):292-314, 2002.
- [16] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):278-284, 1999.
- [17] C. Hurtado and A. Mendelzon. OLAP dimension constraints, In *Proc. PODS*, 2002.
- [18] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, D. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274-291, 2002.
- [19] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses? In *Proc. VLDB*, 1999.
- [20] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. A foundation for capturing and querying complex multidimensional data. *Information Systems*, 26(5): 383-423, 2001.
- [21] D. Pedersen, K. Riis, and T. B. Pedersen. Query optimization for OLAP-XML federations. In *Proc. DOLAP*, 2002.
- [22] H. Schoning. Tamino - A DBMS designed for XML. In *Proc. ICDE*, 2001.
- [23] J. Shanmugasundaram, K. Tuft, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. VLDB*, 1999.
- [24] Y. Stavrakas and M. Gergatsoulis. Multidimensional semistructured data: Representing context dependent information on the Web. In *Proc. CAiSE*, 2002.
- [25] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. SIGMOD*, 2001.
- [26] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. SIGMOD*, 2002.
- [27] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Recommendation. Available from <http://www.w3.org/TR/xmlschema-1/>, May 2001.
- [28] TPC-W, A Transactional Web E-Commerce Benchmark. Available at <http://www.tpc.org/tpcw/>.
- [29] B. B. Yao and M. Tamer Ozsu. Evaluation of DBMSs using XBench benchmark. TR-CS-2003-24, University of Waterloo, August 2003.
- [30] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM TOIT*, 1(1):110-141, 2001.