Modeling and Querying E-Commerce Data in Hybrid Relational-XML DBMSs

Lipyeow Lim, Haixun Wang, and Min Wang

IBM T. J. Watson Research Center
{liplim,haixun,min}@us.ibm.com

Abstract. Data in many industrial application systems are often neither completely structured nor unstructured. Consequently semi-structured data models such as XML have become popular as a lowest common denominator to manage such data. The problem is that although XML is adequate to represent the flexible portion of the data, it fails to exploit the highly structured portion of the data. XML normalization theory could be used to factor out the structured portion of the data at the schema level, however, queries written against the original schema no longer run on the normalized XML data. In this paper, we propose a new approach called eXtricate that stores XML documents in a space-efficient decomposed way while supporting efficient processing on the original queries. Our method exploits the fact that considerable amount of information is shared among similar XML documents, and by regarding each document as consisting of a shared framework and a small diff script, we can leverage the strengths of both the relational and XML data models at the same time to handle such data effectively. We prototyped our approach on top of DB2 9 pureXML (a commercial hybrid relational-XML DBMS). Our experiments validate the amount of redundancy in real e-catalog data and show the effectiveness of our method.

1 Introduction

Real data in industrial application systems are complex. Most data do not fit neatly into structured, semi-structured or unstructured data models. It is often the case that industrial data have elements from each of these data models. As an example consider managing product catalog data in E-Commerce systems. Commercial e-Commerce solutions such as IBM's Websphere Product Center (WPC) have traditionally used a vertical schema [1, 2] in a relational DBMS to manage the highly variable product catalog data. In addition to the vertical schema, the research community has proposed several strategies for managing data with schema variability using relational DBMSs. These include variations on the horizontal, the vertical, and the binary schema [3, 4]. However, the relational data model remains ill-suited for storing and processing the highly flexible semi-structured e-catalog data efficiently. The flexibility of the XML data model, on the other hand, appears to be a good match for the required schema flexibility and Lim et al. [5] has proposed managing product catalog data using XML-enabled DBMS. However, the flexibility of XML in modeling semi-structured data usually comes with a big cost in terms of storage especially for XML documents that have a lot of information in common.

Example 1. Consider the XML fragments for the specifications of two plasma HDTV product on the `www.newegg.com` website. Not only do the two XML documents share many common structural elements (eg. "AspectRatio", "Resolution"), they also share many common values (eg. "16:9", "1200 cd/m2").

```
<ProductInfo>                                  <ProductInfo>
  <Model>                                        <Model>
    <Brand>Panasonic</Brand>                       <Brand>Philips</Brand>
    <ModelID>TH-58PH10UK</ModelID>                 <ModelID>42PFP5332D/37</ModelID>
  </Model>                                       </Model>
  <Display>                                      <Display>
    <ScreenSize>58in</ScreenSize>                  <ScreenSize>42in</ScreenSize>
    <AspectRatio>16:9</AspectRatio>                <AspectRatio>16:9</AspectRatio>
    <Resolution>1366 x 768</Resolution>            <Resolution>1024 x 768</Resolution>
    <Brightness>1200 cd/m2</Brightness>            <Brightness>1200 cd/m2</Brightness>
    <Contrast>10000:1</Contrast>                   <Contrast>10000:1</Contrast>
    <PixelPitch>0.942mm</PixelPitch>               <ViewingAngle>160(H)/160(V)</ViewingAngle>
  </Display>                                      </Display>
  ...                                            ...
</ProductInfo>                                  </ProductInfo>
        (a) Panasonic Plasma HDTV                       (b) Philips Plasma HDTV
```

Fig. 1. Two XML fragment of Plasma HDTV product info from on-line retailer `newegg.com`. Bold face denotes information that is unique to that XML document.

It is clear from Ex. 1 that XML descriptions of products in the same or similar category have many structural elements and values in common resulting in storage inefficiency. If minimizing storage were our only goal, XML compression methods (such as XMill [6] and XGrind [7]) could be used to eliminate most of the storage inefficiency at the expense of less efficient query processing due to the need for decompression. Another approach would be to apply XML normalization [8–10] on these XML documents assuming a super-root. Unfortunately, applying XML normalization poses two problems. First, XML normalization is a design-time process that requires both the schema and the functional dependencies of the XML data to be specified. In many real applications the functional dependencies are neither identified nor specified. In fact, some applications do not even require a schema to be specified. When a schema is specified, these schemas are typically part of industrial standards that do not admit any modification by a normalization process. Second, queries written against the original XML data no longer work against the normalized XML data[1]. This is a serious problem, because application developers typically write queries against the original XML data. The goal of our work is to exploit the redundancy in the XML data in order to support efficient storage of the data, efficient query processing over the data, and transparency to the user, i.e., the user need not re-design the schema of the XML data or rewrite the queries. In contrast to low-level compression methods and schema-based normalization methods, we explore structural as well as value similarities among a set of semi-structured data documents to create models that allow efficient storage and query processing. To our advantage, commercial DBMSs are rolling out native XML support [11, 12], which provides a new option to managing data with high schema variability [5]. We leverage such systems to build a hybrid model for semi-structured data.

Our Approach. In this paper, we propose a new approach called EXTRICATE to manage data with high schema variability. Fig. 2 presents the EXTRICATE system on a high level. The EXTRICATE system consists of two main components: the EXTRICATE data modeler and the EXTRICATE query processor. The EXTRICATE data modeler takes as input the original XML collection, "extricates" a shared XML document (a *model*), stores the original documents as differences from the shared XML document, and gen-

---

[1] In theory, the queries on the original XML data could be transformed into queries on the normalized XML data. Unfortunately, there is no known query transformation algorithm for this purpose.
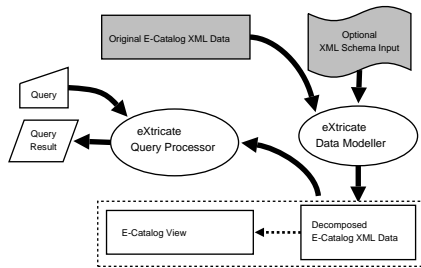
Fig. 2. Overview of our EXTRICATE system.
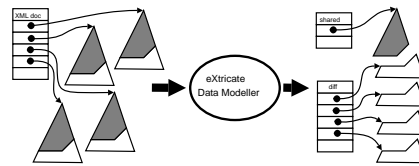
Fig. 3. A collection of XML documents are decomposed into a shared XML tree and a collection of *diff* scripts. The shaded portion of the XML trees denote common information.

erates a view of the decomposed data that has the same schema as the original data. The EXTRICATE query processor takes as input user queries written against the original data and transforms the query into a query processing plan for the decomposed data.

For concreteness, consider managing the electronic product catalog of an on-line retailer using a hybrid relational-XML DBMS. The product information can be stored in a table with an XML column using the following schema:

ecatalog(productID **INT**, categoryID **INT**, info **XML**).

Each product is uniquely identified by its productID. The categoryID encodes product category information like "Electronics > Plasma / LCD / DLP TV > Plasma TV > Panasonic" as a numerical identifier. The info field stores the detailed product information in XML form. These XML data can be queried, using embedded XPath expressions in SQL [11, 13] or using XQuery. As motivated by Ex. 1, it is safe to argue that products in the same category usually exhibit considerable structural and value similarity. Conversely, products in different categories exhibit more differences both structurally as well as in terms of values. For instance, MP3 players have attributes such as *Storage Capacity*, *Supported Audio Formats*, etc., that HDTVs typically do not.

Using EXTRICATE, the ecatalog table will be decomposed internally into two tables, namely:

categoryInfo(categoryID **INT**, sharedinfo **XML**)

productInfo(productID **INT**, categoryID **INT**, diff **XML**)

The categoryInfo table stores the XML tree shared by all products in a particular category. The productInfo table encodes each of the original XML document in the info column as the *diff* or *edit transcript* from the shared XML tree of the associated category. Each XML document is therefore decomposed into a shared XML tree and its diff from the shared XML tree. Since the shared XML tree is stored once for all the XML documents in the category, significant storage savings can be obtained for highly redundant XML data.

From the perspective of the user of the e-catalog database, nothing has changed, because the e-catalog data is presented to the user as a view, which has the same schema as the original ecatalog table. Applications and queries on the ecatalog table require no change either: EXTRICATE's query processor will transform any query against the original ecatalog table into queries against the categoryInfo table, and if necessary, the productInfo table (some queries can be answered by accessing

`categoryInfo` only). The query transformation is carried out by a rule-based query processor that is described in detail in Sect. 3.

Our Contributions. We summarize our contributions as follows:

– We propose a space-efficient and query-friendly model for storing semi-structured data that leverages commercial DBMSs with native XML support. We show that with the new model, we can dramatically reduce storage redundancy of XML data and improve query performance at the same time.
– We propose a query transformation algorithm to automatically rewrite user queries on the original data to queries on the data stored using our model without any user intervention. In many cases, the transformed queries can be answered much more efficiently than the original ones since they access much less amount of data.
– We show that our approach is friendly toward schema evolution, which occurs frequently in applications such as e-commerce data management.

Paper Organization. Sect. 2 presents EXTRICATE's data modeler. Query rewriting and processing is discussed in Sect. 3. We discuss data maintenance issues related to schema evolution in Sect. 5. Experimental results are presented in Sect. 6. We review related work in Sect. 6 and conclude in Sect. 7.

2 Data Modelling

Our EXTRICATE data modeler is inspired by the predictive compression paradigm, where the data is represented losslessly using a model that predicts the data approximately and the residue that is the difference between the predicted data and the true data. In an analogous fashion, EXTRICATE extracts a *shared* XML document from a set of XML documents and represents each of the original document using the shared document and the *diff* script. The shared XML document represents a "model" that predicts the XML documents in the collection. The *diff* script represents the "residue" that is a set of differences between the document and the shared document. Since the shared XML document is common to all the documents in the collection, we only need to store it once. Hence the entire collection of XML documents is stored as a single shared XML document and a collection of diff scripts, one diff script for each document in the collection (see Fig. 3).

Continuing with the e-catalog example, recall that the product information has been partitioned according to product categories and that the product information within the same category have both structural and value similarities. The product information are stored as XML documents and we will use a tree representation for XML documents in our discussion. Conceptually, the EXTRICATE data modeler first finds a *shared XML* tree for a collection of XML trees. The shared XML tree is the maximal sub-tree common to all the XML trees in the collection. The *diff* of each original XML tree from the shared XML tree can then be computed using known tree diff algorithms [14–17].

Example 2 (A Data Modelling Example). Consider the XML documents (represented as trees) of the two HDTV products shown in Fig. 1. The shared XML tree is shown in Fig. 4(a) and the differences are shown in Fig. 4(b).

Observe that the shared XML tree (1) is a single connected tree, (2) contains values in addition to structure. In practice, the shared tree may be further annotated with statistical information (e.g., the `min`, `max` values of, say, the `listprice` element).

(a) The shared XML tree.

(b) The *diff*'s.
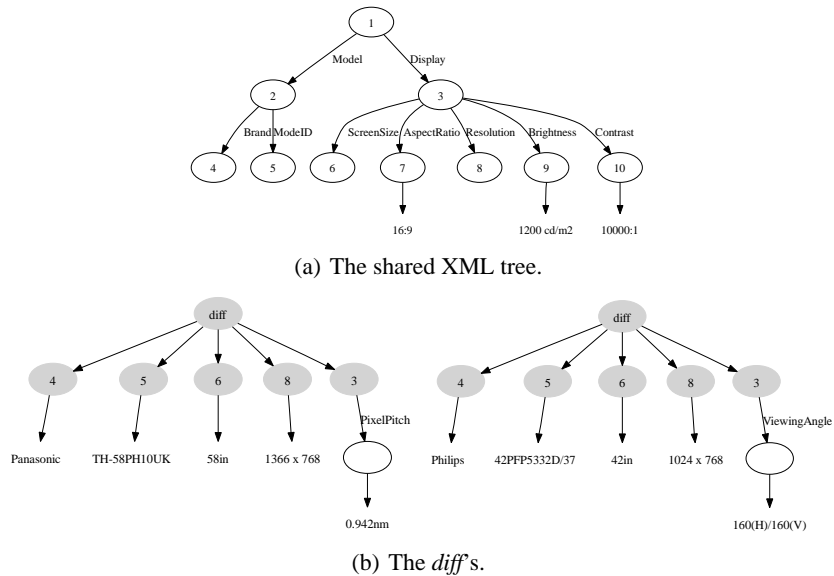
Fig. 4. The decomposed representation of the two HDTV XML documents in Fig. 1

Observe that each *diff* document represents a set of insertion operations onto the shared XML tree. Each of the two *diff* trees represents 4 insertions, as there are 4 child nodes under the root node. The name of these child nodes are node identifiers (NIDs), which identify the nodes or the location of insertion in the shared XML tree. These NIDs on the shared XML tree may be either implicitly maintained by the DBMS or explicitly annotated on the tree nodes.

It is clear that by inserting each sub-tree in the *diff* as the child of the node in the shared XML tree with the specified NID, the original XML document can be recovered.

A direct benefit of our decomposition is storage efficiency. Information common to all product documents within a category is "extricated" and stored once in the shared XML document, instead of being redundantly stored in every document. However, in contrast to XML compressors (such as XMill [6] and XGrind [7]) the goal of our method is not low-level compression: our decomposition method operates at the data model level with a focus on efficient query processing. In other words, the decomposition must be done in a query-friendly way, that is, we need to ensure queries on the original table can be mapped to queries on the decomposed tables and processed efficiently thereafter.

2.1 Finding the Shared XML Tree

In this section we discuss the first step of the decomposition process: finding the (largest) shared XML tree, given a set of XML documents having similar structures and values.

This problem is similar to the *maximal matching* problem [16] and is complementary to the problem of finding the smallest diff between two XML trees. Efficient algorithms exists for ordered trees [16]. For unordered trees, the problem is NP-hard [14],
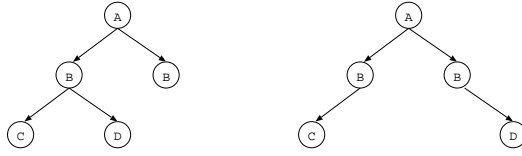
Fig. 5. A Shared XML document and the *diff*

but polynomial time approximate algorithms [14, 15] exist. However, our problem differs from the work mentioned above in that the the maximal matching need to be a connected tree. The goal of the above-mentioned related work is to find the maximal common fragments, i.e., a common forest among two XML trees in order to minimize the edit distance between the two XML trees. In contrast, our goal is to find a single rooted tree that is common to a collection of XML trees.

We discuss the case of finding shared XML documents for unordered trees, which is more difficult than for ordered trees. One difficulty in finding a single shared tree among a set of XML trees is that a set of unordered children nodes may have the same node name. The difficulty can be illustrated by the two XML documents shown in Fig. 5. The node B beneath A occurs twice in both documents. The largest shared document is not unique for these two documents. One alternative will contain all nodes except for node C, and the other all nodes except for node D. To find the largest shared document among a set of documents, we must store all such alternatives at every step, which makes the complexity of the entire procedure exponential.

In this work, we use a greedy approach. We find the shared document of two documents starting from their root nodes. Let n_1 and n_2 be the root nodes of two XML documents. Two nodes are said to match if they have the same node type, and either they have same names (for element/attribute nodes) or they have the same values (for value nodes). If n_1 and n_2 do not match, then the shared document is an empty document. Otherwise, we recursively find matches for each of their child nodes. Special consideration is given to the case where several child nodes have the same name and the child nodes are unordered. Assume $\mathcal{C}_1(l) = \{s_{11}, \cdots, s_{1m}\}$ and $\mathcal{C}_2(l) = \{s_{21}, \cdots, s_{2n}\}$ are two sets of child nodes with the same name that we need to match. We find recursively the shared XML sub-tree for every pair (s_{1i}, s_{2j}) of the instances. Out of the $m \times n$ shared XML trees, we pick the shared XML sub-tree r_{pq} with the largest size and add r_{pq} as the child node of the current shared XML tree r. We remove all the shared trees associated with either s_{1p} or s_{2q} from the candidate set $\mathcal{M}$ so that they will not be chosen anymore. Then, we find the next largest in the remaining $(m-1)(n-1)$ candidate shared XML sub-trees. We repeat this process until no shared sub-trees can be found.

Algorithm 1 outlines the MATCHTREE procedure that finds a shared document between two XML documents. Based on the MATCHTREE procedure, Algorithm 2 finds the shared document among a set of XML documents.

2.2 Finding the Differences

After the shared XML tree is found, the difference between each document in the original collection and the shared XML tree can be found using existing tree diff algorithms [14–17]. In addition to finding the differences, we optionally annotate the shared

Algorithm 1 MATCHTREE(n_1,n_2)

Input: n_1, n_2: root node of the two XML tree
Output: r: a shared subtree

1: **if** n_1 matches n_2 **then**
2:    $r \leftarrow$ new node
3:    copy n_1 to r
4: let $\mathcal{C}_1 = \{$child nodes of $n_1\}$
5: let $\mathcal{C}_2 = \{$child nodes of $n_2\}$
6: let $\mathcal{L} = \{$ node names common to $\mathcal{C}_1$ and $\mathcal{C}_2\}$
7: **for** each node name $l \in \mathcal{L}$ **do**
8:    let $\mathcal{C}_1(l) = \{$nodes from $\mathcal{C}_1$ with name $l\} = \{s_{11}, \cdots, s_{1m}\}$
9:    let $\mathcal{C}_2(l) = \{$nodes from $\mathcal{C}_2$ with name $l\} = \{s_{21}, \cdots, s_{2n}\}$
10:    **for** each $(s_{1i}, s_{2j}) \in \mathcal{C}_1(l) \times \mathcal{C}_2(l)$ **do**
11:       $r_{ij} \leftarrow$ MATCHTREE(s_{1i}, s_{2j})
12:    let $\mathcal{M} = \{r_{ij} : \forall i, j\}$
13:    **while** $\mathcal{M} \neq \emptyset$ **do**
14:       $r_{pq} \leftarrow \arg\max_{r_{ij} \in \mathcal{M}}$ SizeOf(r_{ij})
15:       add r_{pq} as a child node of r
16:       remove r_{pk} and r_{kq} from $\mathcal{M}, \forall k$
17: return r

Algorithm 2 Finding the shared XML document in a set of XML documents

Input: D: a set of XML documents (represented by their root nodes)
Output: r: a shared XML document

1: Assume $D = \{d_1, d_2, \cdots, d_n\}$
2: s $\leftarrow d_1$
3: **for** each document $d \in D$ **do**
4:    s $\leftarrow$ MATCHTREE(s, d)

XML tree with statistics collected during the scan through each original document for finding the differences.

Difference model. We model the differences between an original document and the shared XML tree as a set of sub-tree insertions. In the most trivial case, a sub-tree can just be a single value. Each insertion is represented using an XML fragment consisting of a node specifying the node identifier (NID) that uniquely identifies the insertion point in the shared XML tree. The subtree rooted at that node is the subtree to be inserted. The fragments for all the insertions associated with a single original document are then collected under root node `<diff>`, forming the XML representation for the *diff* (see Fig. 4(b) for an example). A richer difference model may include deletions as well. However, we do not discuss this option in this paper.

Optional annotations. We annotate the shared XML tree with two types of additional information: node identifier (NID) annotation and value annotation. The NIDs are used by the difference representation to specify insertion location. As discussed in the example of Sect. 2, NIDs can be implicitly maintained by the DBMS or explicitly annotated as attributes in the shared XML tree. In Fig. 4(a), the node labels 7 and 9 denote the NIDs of the nodes `AspectRatio`, and `Brightness` respectively. These NID of these nodes are used in the *diff*'s .

To facilitate efficient query processing, we may also optionally annotate nodes in the shared XML tree with some simple statistics about the values associated with those nodes. For example, consider the `ScreenSize` element in Fig. 4(a). Each of the two document has a different value for `ScreenSize`. We can annotate the `ScreenSize` element in the shared XML tree by the minimum and the maximum value (in this case

Algorithm 3 PROCESSQUERY(S, D, p)

Input: S shared trees, D diff's, p XPath
Output: R satisfying productIDs
1: $R \leftarrow \emptyset$
2: **for all** each shared tree $s \in S$ **do**
3:     let N be the set of NIDs of the maximal matching nodes
4:     let p' be the unmatched suffix of the XPath p
5:     $(N, p') \leftarrow$ MaxMatchXPath(s, p)
6:     **if** $p' = \epsilon$ **then**
7:        /* p completely matched */
8:        $R \leftarrow R \cup$ fetchProductIDbyCat(catID(s))
9:     **else**
10:        **for** each diff $d \in D$ s.t. catID(d)=catID(s) **do**
11:           **for** each continuation node $c \in$ fetch(N, d) **do**
12:              **if** MatchXPath (c, p') **then**
13:                 $R \leftarrow R \cup$ fetchProductID(d)
14: return R

`42in` and `58in`) so that query predicates such as `/productInfo/Display[ScreenSize < 10]` can be evaluated without accessing the *diff*'s. These value annotations are collected while scanning the documents (for computing the *diff*) and added after the *diff*'s are processed.

3   Query Processing

In this section, we show how queries against the original set of XML documents can be decomposed into queries against a set of shared trees and queries against their *diff*'s. We focus on filtering queries because these are the most common type of query on e-catalog data. A filtering query consists of a filtering condition specified in XPath on the XML column in the original table and fetches the tuples (or a projection thereof) that satisfy the filtering condition.

Algorithm 3 outlines the process of how filtering queries over documents of varied schemata are processed using sub-queries over the shared documents and the *diff*'s. The key idea is to first process the XPath filtering condition p on the shared trees S. For the shared trees that completely satisfy the filtering condition, the associated set of diff's need not be checked, instead all the associated tuples can be returned as satisfying the query. For the shared trees that contains partial matches to p, the associated diff's need to be consulted to complete the matching. The diff's that complete the matching of the remaining unmatched sub-path p' satisfy the query and the corresponding tuple can be returned. Note that in practice Algorithm 3 is implemented by leveraging SQL/XML queries over a hybrid relational-XML DBMS.

Example 3 (A single-constraint query). Suppose we want to find all products in the e-catalog that are blue in color. It is straightforward to ask such a query against the original `ecatalog` table using the following filtering query specified in the SQL/XML query language,

```
SELECT  productID
FROM    ecatalog AS C
WHERE   XMLExists('$t/productInfo[color ="Blue"]'
                  PASSING BY REF C.info AS "t").
```
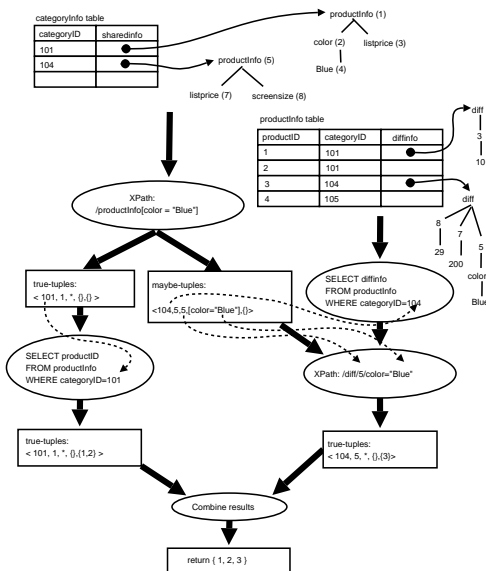
Fig. 6. Evaluating single constraint product search queries. The numbers in parenthesis beside the node labels denote the NIDs of the nodes.
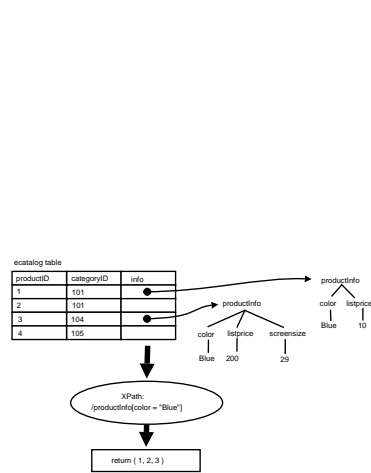
Fig. 7. Evaluating single constraint product search queries on the original `ecatalog` table.

The query in Example 3 scans each row of the `ecatalog` table and evaluates the `XMLExists` predicate on the XML tree in the `info` field. If the predicate evaluates to true, the `productID` field of the row is returned. The SQL/XML function `XMLExists` returns true if a node matching the given XPath expression exists in the given XML tree.

Fig. 6 illustrates the process of evaluating the sample query (Example 3) against the two tables, `categoryInfo` and `productInfo`, in the e-catalog application. First, we evaluate the XPath expression against `categoryInfo` table. The result is a sequence of tuples that represent temporary matches. These temporary tuples fall into two types: true-tuples and maybe-tuples. **true-tuples** contain return nodes that satisfy the entire XPath expression. Since the predicate has been satisfied, the last matched node and the remaining XPath fields are empty. **maybe-tuples** contain return nodes that may satisfy the predicate when combined with the *diff* information.

For the true-tuples, all the documents in the category represented by the tuple satisfy the query. We therefore retrieve their document IDs from the *diff* table. The maybe-tuples, however, need to be further evaluated against the *diff* table in order to reject or accept them. To do this, we use the `categoryID` and the NID of the last matched node of each maybe-tuple to filter out irrelevant rows and nodes in the *diff* table. The remaining XPath expression of each maybe-tuple is then evaluated on the filtered node set from the the `diff` column. The maybe-tuples whose remaining XPath expression are satisfied by some `diff` are then returned as true-tuples.

Note that when the path expression contains the wildcard element '//', the above procedure may not work as described. To see this, let the path expression be in the form

of "/a/b//c/d". If it is split into two path expressions at the position between "a" and "b" or between "c" and "d", then no change to the above procedure is necessary. There is, however, the possibility that the query dividing position occurs "inside" the '//' element. Thus, any node x under the path "/a/b" in a shared document will satisfy the prefix query. As a result, multiple queries in the form of "/x//c/d" will be issued against the *diff* table. In particular, if "/a/b" is empty, we will query the *diff* tables directly with the path expression "//c/d". This represents the worst case, wherein the shared document table has no filtering power, and we need to process all of the *diff* tables for the query. In other words, we degenerate into the case of querying against the original table.

We focus on filtering queries, because they are arguably the most important type of queries on semi-structured data such as the e-catalog, which consists of documents of similar structures. In other applications, more complex types of queries may call for more sophisticated query transformation and processing techniques. As future work, we will study how to extend our simple algorithm to handle more query types and how to balance the load between the query transformation and the query execution processes.

Discussion: Query Performance. The query performance of EXTRICATE is dependent on the amount of redundancy in the XML documents. We briefly discuss this relationship by comparing the processing steps for querying the `categoryInfo` and `productInfo` tables to those for just querying the `ecatalog` table. For ease of exposition, we focus on single constraint queries, because, conceptually, multi-constraint queries are processed as multiple single constraint queries with an intersection operation at the end to combine the results.

Consider the high-level query processing plan for `categoryInfo` and `productInfo` in Fig. 6 and for the original `ecatalog` table in Fig. 7. The most expensive operator in terms of running time is the XPath operator. The XPath operator is a logical operator that retrieves nodes that satisfy a given XPath expression. In IBM's DB2 two physical operators [12, 11] can be used to implement the XPath operator during runtime: (1) the XML navigation operator (XNav), which traverses a collection of XML trees in order to find nodes that match the given XPath expression, and (2) the XML index scan operator (XIScan), which retrieves all matching nodes from an XML index. In both cases, the running time of each physical operator is dependent on the following data characteristics: (1) the number of rows in the XML column, and (2) the size of the XML tree in each row. In our method, the data characteristics of the `categoryInfo` and `productInfo` tables depends on the level of redundancy in the product information of each category. In the following, we consider the worst case and the best case scenarios, and show how we can reap the most benefits from our scheme.

Worst Case: no data redundancy. In the worst case, there is no data redundancy: the shared XML document will be small, and all information stays with the *diff* documents. Most queries will have to go through the *diff* documents. Thus, our method will have no advantage over the method that processes the original `ecatalog` table directly. In this case, we should not apply our method at all.

Best Case: full data redundancy. In this extreme case, all the product information within a category is the same. The `diff` column of the `productInfo` table is empty and the `sharedinfo` column of the `categoryInfo` table contains all the information. However, the size of the `categoryInfo` table is much smaller than the size of the `ecatalog` table (note that the XML document size remains unchanged). In the processing plan for our method, the right branch (see Fig. 6) is never executed that is, no maybe-

tuples will be produced, because the XPath can always be evaluated with certainty using the information in the `categoryInfo` table. Since the input size (`categoryInfo` table size) of the XPath operator in our method is significantly smaller than the input size (`ecatalog` table size) of the original method, our method will be significantly more efficient.

Most real life semi-structured data, including the e-catalog data used in our example, is likely to fall somewhere between the best and the worst case. We do not have control over data redundancy, however, the redundancy of the *diff* information within each category can be tuned by the data decomposition. In this paper, we assume that the data schema designer is responsible for data partitioning.

In general, the performance of our method highly depends upon the data redundancy level and the query workload. The higher the level of data redundancy level, the better our method will perform. Also, the more queries in the workload that can be answered by accessing the common documents only, the better our method will perform. As future work, we are exploring how to solve two problems within the data modeler: (1) How to measure the level of data redundancy when a data partition is given, (2) Given a query workload and a set of XML documents, how to form the best data partition so that our method will achieve the optimal performance. Note that if each original XML document is completely different from each other, no partitioning algorithm can produce partitions with high redundancy and our method is not suitable for such cases.

4   E-Catalog Maintenance

One of the challenges of managing data of varied schemata is maintaining the data when new objects, possibly of a different schema, are added. In our scheme, we need to address issues that arise when new data categories are added or old data categories are deleted. In this section, we show how to manage these tasks in our data model.

Deleting a category. Assume we want to remove an unwanted category whose `categoryID` is 102. Users will issue SQL delete statement on the original `ecatalog` view. Our EX-TRICATE system will re-write the delete statement into corresponding delete statements on the decomposed tables, `categoryInfo` and `productInfo`.

Deleting individual documents. To delete an individual document, we simple remove a row that corresponds to the document in the `productInfo` table. Note that after many documents in a document category are deleted, the shared XML tree for that category may no longer be optimal (for instance, the remaining set of documents in the category may have more sharing). Hence a re-organization using the EXTRICATE data modeller may be necessary.

Adding individual documents of new category. When adding a collection of products belonging to a new category, we must identify the the shared document for the new category. We use the EXTRICATE data modeller to obtain the *diff*'s.

Adding an individual document of an existing category. Since the category of that document already exists, there must be at least one product in that category and there must be a shared XML tree for that category. When inserting a new document of that category, we compute a diff between the new document and the shared XML tree, and we store the diff output into the `productInfo` table. Because our difference model only supports insertions at this point, when the new document requires deletions in the shared document, we create new category for the new document. These documents

Source category	electronics/televisions/HDTVs
No. of products	90
Size of original XML	296,267 bytes
No. of shared XML	26
Size of shared XML	30,537 bytes
Size of diff XML	210,244 bytes
Redundancy	1.23

Table 1. Analysis of the redundancy in real e-catalog data.

can be re-integrated with the original category during re-organization. Extending our difference model to include deletions is part of our future work.

Adding a document with new product attributes. By new product attributes, we mean that the product attributes are new to the documents in the same category that are currently in the database. Note that the new product attribute may be encoded as an XML element or an XML attribute. Since the new product attribute does not occur in any other documents in the database, it will be stored as part of the diff in the `productInfo` table. No changes to the shared XML document is required. Note that XML (re-)validation and migration issues are beyond the scope of this paper. Our simple strategy for handling new attributes is therefore comparable to the vertical schema method in terms of cost-efficiency.

5 Experiments

We implemented a prototype of our method and investigated two issues in our experiments: (1) the amount of redundancy in real e-catalog data, and (2) the performance of EXTRICATE under different conditions. The EXTRICATE method exploits redundancy in the data; therefore, analyzing and understanding the amount of redundancy in real e-catalog data is essential. To investigate the performance of EXTRICATE, it is necessary to generate synthetic data that simulates different levels of redundancy in order to understand how our method performs under various conditions.

Redundancy in Real E-Catalog Data. We analyzed the product information of several categories of products at the on-line store, www.bestbuy.com and present the analysis on a representative subset of the data we collected.

In order to quantify the amount of redundancy in the data, we use the ratio of the data set size (in bytes) between the original data set and the decomposed data set,

$$redundancy = \frac{Size\ in\ bytes\ of\ original\ dataset}{Size\ in\ bytes\ of\ decomposed\ dataset}, \tag{1}$$

where the decomposed dataset is the size of all the shared XML document and the size of all the diff's produced by our method.

We use the EXTRICATE data modeler to decomposed the product information for all the HDTV products that we downloaded from www.bestbuy.com and measured the redundancy of the decomposed data with respect to the original data. The key statistics are tabulated in Table 1. We retained the website's categorization of the HDTV products into the "flat panel", "projection", and "up to 39 inches" sub-categories, and further partitioned the products in each of these sub-categories by their manufacturer Our results

validate our assumption that e-catalog data contains significant amount of redundancy that can be exploited by EXTRICATE.

Performance of eXtricate. We investigate the performance characteristics of EXTRICATE over data with different characteristics as well as over different types of queries. Our goal is to understand how well our method performs on data and queries with different characteristics, and not so much on how well it performs on a single, specific set of real data and queries.

Data. We generated XML documents that are complete binary trees characterized by depth. A complete XML tree of depth d, will have $2^d - 1$ elements and 2^{d-1} leaf values. We simulate the amount of shared data between two XML documents by specifying the depth of the tree that is shared between them. For example, two XML trees of depth 6 with a shared depth of 4, will have common elements and structure up to a depth of 4. The sub-trees hanging off depth 4 will be different. The XML documents we used in our experiments have depth 6.

Each dataset consists of a number of XML documents from 4 categories. The amount of redundancy in each dataset is determined by (1) the number of documents in each category, and (2) the shared depth of the documents within each category.

Queries. We present our results for single constraint queries in this paper. We measure the performance of the same query on the original dataset (denoted in the plots by "query on original data") and on the decomposed dataset. For the decomposed dataset, we further measure the performance when the query requires both the shared documents and the diff's to answer the query (denoted in the plots by "query on shared + diff"), and when the query only requires accessing the shared document to answer the query (denoted in the plots by "query on shared". We run each query 1000 times on a multi-user machine running AIX unix and record the average elapsed time.

Performance vs Redundancy. We use the same approximate measure for redundancy as shown in Eq. 1 and measure the running time of the 3 different query types on different datasets with shared depth ranging from 2 to 5. The number of documents in each category, i.e., the number of documents sharing a single shared document, is set to 1000. We plot the running time against the redundancy of the data on Fig. 8(a). We observe that EXTRICATE provides significant improvement in query performance whether the diff's are used or not. When the query can be answered just by looking at the shared documents, the reduction in processing time is even more dramatic. Moreover, the performance improvements are not sensitive to the size of the shared XML tree among the documents in the same category.

Fig. 8(b) shows the same results in terms of speedup with respect to processing the query on the original, non-decomposed dataset. Observe that (1) the speedup for queries requiring access only to the shared document is much greater than the speedup for queries requiring access to the diff documents, (2) the speedups are decreasing slightly as the size of the shared tree increases. For queries requiring access only to the shared tree, increasing the size of the shared tree would logically lead to a smaller speedup and this is reflected in our experiments. For queries that require both the shared document and the diff's to answer, the decrease in speedup is mostly due to the increased processing time on the larger shared trees, and to a lesser extent on the number of sub-trees under the "⟨diff⟩" element in each diff document. The number of sub-trees under each "⟨diff⟩" element increases (even though each sub-tree is smaller) as the shared depth increases. When processing an XPath expression on the diff documents, this increases

Performance vs Data Redundancy
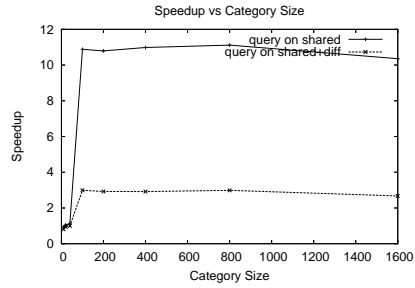
Speedup vs Data Redundancy
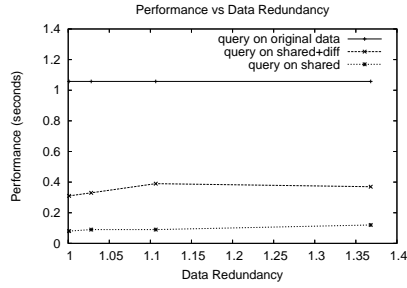
(a) Running time on single constraint queries over datasets with varying amount of shared data.
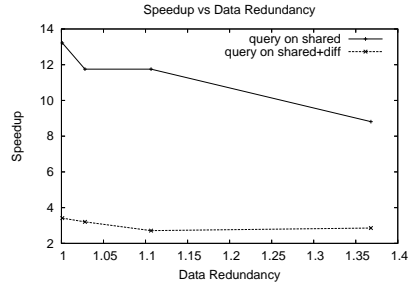
(b) Speedup of running the single-constraint queries on datasets with varying amount of shared data.

Performance vs Category Size

Speedup vs Category Size
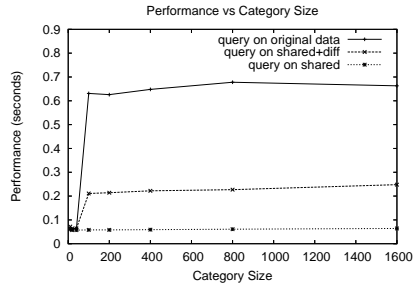
(c) Running time for single-constraint queries on datasets with varying number of documents sharing a single shared sub-tree.

(d) Speedup of running the single-constraint queries on datasets with varying number of documents sharing a single shared sub-tree.

Fig. 8. Performance of EXTRICATE over different conditions.

the number of sub-trees on which to match the XPath expression. For example if the shared depth is 5, the number of sub-trees in each diff document would be 2^4.

Performance vs Category Size. Another important factor affecting the amount of redundancy in the data is the number of documents sharing a single shared tree (this number is the same as the category size). We set the shared depth to 4 and vary the number of documents within a category. The same queries were run and the elapsed time plotted in Fig. 8(d). Observe that the processing time remain constant for queries that only require access to the shared documents, because the table for shared documents is not affected by category size at all. When the category size is very small, the performance of EXTRICATE is almost comparable to running the query against the original data; however, when the category size becomes larger than 40, the speedup of our method becomes significant.

The speedups are plotted in Fig. 8(d). Our results match our intuition that as the number of documents sharing each shared document increases, the speedup increases.

6 Related Work

Managing e-catalog data using various relational DBMS strategies has been explored in [1, 3, 4, 2]. The *horizontal schema* approach stores all the product information in a single table where the columns are the union of the attribute sets across all products and the rows represents the products. The *binary schema* [3, 4] approach creates one table for each distinct attribute. Each table stores value-OID pairs. The *vertical schema* approach uses only one table with three columns: OID, $Attribute$, and $Value$. Each product is represented as a number of tuples in this table, one for each attribute-value combination. These approaches all have serious problems with data sparsity, schema evolution, usability, query processing complexity, difficulty in query optimization. Agrawal et al. proposed creating a horizontal view on top of a vertical schema [1] to ease the difficulty of writing queries on vertical schemata, but does not address the query performance problems. A middle-ware solution called *SAL*, Search Assistant Layer, has also been proposed in [2] to optimize queries written against vertical schemas. However, SAL is not able to leverage the full capability of the query optimizer, because it is not integrated with the DBMS engine.

On the XML front, a method using DBMS with native XML support to manage e-catalog data was proposed in [5]. The schema evolution problem is addressed using XML. Query performance is dependent on the XQuery/XPath processing engine inside the DBMS engine. Our paper builds upon their approach by decomposing the XML documents for storing product information in order to reduce storage redundancy and improve query performance.

Our decomposition-based storage model is also motivated by data compression techniques. While many XML compressors (such as XMill [6]) achieve higher compression rate, their goal is solely to minimize storage space. Query processing require decompression. In contrast, our goal is two pronged: reduce redundancy in storage and provide efficient query support. XGrind [7] is a query-friendly XML compressor that supports query processing in the compressed domain; however, XGrind is still a low level compressor relying on traditional encoding techniques (e.g., Huffman code) to encode strings in an XML document. Our EXTRICATE system does not perform string-level encoding, but models the redundancy in a collection of XML documents at a logical level.

XML normalization theory [8–10] also models the redundancy of XML data at the logical level. The fundamental difference between EXTRICATE and XML normalization is that XML normalization is a design-time process and requires the schema and the functional dependencies to be fully specified, whereas EXTRICATE makes no assumptions on the schema or the functional dependencies and is completely data-centric. Moreover XML normalization does not address transformation of queries to run on the normalized schema.

The problem of finding the intersection and diff of two XML trees is also a well studied problem. The survey article by Cobena et al. [17] provides a comprehensive description of all the techniques. In this paper, our focus is not on algorithms for find the intersection and diff, but on using these algorithms in a novel way to solve data management problems in E-commerce.

7 Conclusion

The native XML support in DBMSs promises to offer database users a new level of flexibility in managing semi-structured and unstructured data in relational DBMSs. However, this advantage may come with a significant cost in data storage and query processing if we do not use it wisely.

In this paper, we demonstrate how to combine the strengths of relational DBMSs and the flexibility of the XML support by a case study on managing E-commerce data. We argue that while the e-catalog data appears to be lacking a unified structure, they do share common sub-structures and common values among different product descriptions, and the degree of such sharing could be very high for products in the same category. The extreme approach of storing each product description as a complete XML document without any schema constraint will thus result in huge overhead in terms of both storage and query processing. Handling the common parts and the differences separately leads to a natural approach: We only use the freedom when we really need it.

References

1. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: VLDB, Morgan Kaufmann (2001)
2. Wang, M., Chang, Y., Padmanabhan, S.: Supporting efficient parametric search of e-commerce data: A loosely-coupled solution. In: EDBT. (2002) 409–426
3. Copeland, G.P., Khoshafian, S.: A decomposition storage model. In: SIGMOD, ACM Press (1985) 268–279
4. Khoshafian, S., Copeland, G.P., Jagodis, T., Boral, H., Valduriez, P.: A query processing strategy for the decomposed storage model. In: ICDE, IEEE (1987) 636–643
5. Lim, L., Wang, M.: Managing e-commerce catalogs in a DBMS with native XML support. In: ICEBE, IEEE (2005)
6. Liefke, H., Suciu, D.: XMill: An efficient compressor for xml data. In Chen, W., Naughton, J.F., Bernstein, P.A., eds.: SIGMOD. (2000) 153–164
7. Tolani, P., Haritsa, J.R.: XGrind: A query-friendly XML compressor. In: ICDE. (2002)
8. Arenas, M., Libkin, L.: A normal form for XML documents. In: PODS. (2002) 85–96
9. Libkin, L.: Normalization theory for xml. In Barbosa, D., Bonifati, A., Bellahsene, Z., Hunt, E., Unland, R., eds.: XSym. Volume 4704 of Lecture Notes in Computer Science., Springer (2007) 1–13
10. Arenas, M.: Normalization theory for xml. SIGMOD Rec. **35** (2006) 57–64
11. Nicola, M., der Linden, B.V.: Native XML support in DB2 universal database. In: VLDB. (2005) 1164–1174
12. Ozcan, F., Cochrane, R., Pirahesh, H., Kleewein, J., Beyer, K., Josifovski, V., Zhang, C.: System RX: One part relational, one part XML. In: SIGMOD. (2005)
13. Funderburk, J.E., Malaika, S., Reinwald, B.: XML programming with SQL/XML and XQuery. IBM Systems Journal **41** (2002)
14. Zhang, K.: A constrained edit distance between unordered labeled trees. Algorithmica **15** (1996) 205–222
15. Wang, Y., DeWitt, D.J., yi Cai, J.: X-Diff: An effective change detection algorithm for XML documents. In: ICDE. (2003) 519–530
16. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD, ACM Press (1996) 493–504
17. Cobena, G., Abdessalem, T., Hinnach, Y.: A comparative study of XML diff tools (2002) http://www.deltaxml.com/pdf/is2004.pdf.