Mapping XML to a Wide Sparse Table

Liang Jeff Chen[2*], Philip A. Bernstein[1], Peter Carlin[1], Dimitrije Filipovic[1], Michael Rys[1],
Nikita Shamgunov[3*], James F. Terwilliger[1], Milos Todic[1], Sasa Tomasevic[1], Dragan Tomic[1]

Microsoft Corporation[1] *UC San Diego*[2] *Facebook Inc.*[3]

[1]{philbe, peterca, dimfi, mrys, jamest, v-mitodi, satom, dragant}@microsoft.com
[2]jeffchen@cs.ucsd.edu [3]nikita.shamgunov@gmail.com

***Abstract*— XML is commonly supported by SQL database systems. However, existing mappings of XML to tables can only deliver satisfactory query performance for limited use cases. In this paper, we propose a novel mapping of XML data into one wide table whose columns are sparsely populated. This mapping provides good performance for document types and queries that are observed in enterprise applications but are not supported efficiently by existing work. XML queries are evaluated by translating them into SQL queries over the wide sparsely-populated table. We show how to translate full XPath 1.0 into SQL. Based on the characteristics of the new mapping, we present rewriting optimizations that minimize the number of joins. Experiments demonstrate that query evaluation over the new mapping delivers considerable improvements over existing techniques for the target use cases.**

I. Introduction

XML is commonly supported by SQL database systems [26]. XML documents are mapped into tables, and XML queries are executed using the SQL query engine, possibly extended to handle special XML data and query constructs. This enables the reuse of mature relational technology and infrastructure. Moreover, it integrates XML documents with relational data and provides interoperability between SQL queries and XML queries.

Mapping nested elements to flattened tables is the key problem for supporting XML on SQL databases. Many mapping schemes have been proposed to decompose nested structures into normalized tables. These existing mappings can be classified into two categories: normalization mappings [9], [14], [28] and node-encoding mappings [10], [16], [17]. *Normalization mappings* shred nested structures into normalized tables, where each table corresponds to one or more elements/attributes. The nested structure is captured by primary-foreign key relationships, so hierarchical navigation in XML queries is evaluated by primary-foreign key joins. *Node-encoding mappings* represent XML elements by numerical encodings of node locations in the XML tree, e.g., Dewey ID or preorder/postorder [29]. Hierarchical relationships and document order are implicitly captured by structural joins of the elements' encodings.

Previous results have suggested that normalization mappings can achieve good query efficiency [21]. Two cornerstones that enable such performance are schema and normalization. Data normalization lies at the foundation of query processing in relational database systems (RDBMSs). It not only reduces redundancy and avoids update anomalies, but also co-locates entities with one-to-one relationships in the same table, which minimizes the number of joins. Normalization mappings follow the same philosophy, and ensure that the number of physical joins is as few as if the original data were modeled by the relational tables.

XML schemas, on the other hand, reduce the complexity of some expensive query constructs specifically designed for XML. One example is the wild card (*) in path expressions, which represents all elements in a document. The expression A/* is implemented as a structural join between the A elements and all the elements in the database, which can be prohibitively slow. As another example, the ancestor-descendant axis A//B joins two elements without specifying a join sequence. Such a join is more expensive than a primary-foreign key join, because an unknown join sequence essentially implies a "range" comparison to enumerate all possible satisfied paths. In general, XML query languages contain unconventional constructs that bring more challenges. With XML schemas, the evaluation of such constructs can be greatly simplified: queries can be pre-evaluated against XML schemas; the star in A/* can be replaced by the elements appearing as A's children in the schema; and A//B can be translated into concrete paths. For this reason, schemas are even more important for query processing in XML than in RDBMSs, as some XML query constructs hide schema information and put an additional burden on the system.

Though normalization mappings provide schema- and normalization-based optimizations, they can only support a limited number of use cases. In particular, we identify three use cases that are widely observed in enterprise applications but cannot be supported by normalization mappings efficiently.

1) *Flexible schema*. XML's schema-less property is one of its major advantages over other schema-first data models. In normalization mappings, schema design of primary keys and foreign keys requires full knowledge of the XML schema, and cannot be applied to this use case. While the XML schema can be summarized from the initial document set [14], later updates may cause expensive schema evolution and table repartitioning.
2) *Many small XML documents*. Enterprise applications often use many small XML documents rather than a few large ones [5]. Normalization mappings typically shred one XML document over many tables, which causes fragmentation of small XML documents. As a consequence, the cost of XML reconstruction can be very high.

*Work done while at Microsoft Corp.

3) *Many optional elements/attributes* [21]. By data normalization, XML attributes are stored as additional columns in the tables of their parent elements. This strategy, however, results in many sparse tables and incurs storage overhead.

Node-encoding mappings are generic solutions for the three use cases. Since nested structures can be reconstructed from node encodings, conceptually, all elements and attributes can be stored in a four-column table: (doc id, node encoding, tag name, value). Given that the table schema and XML-to-SQL query translation are independent of XML schemas, node-encoding mappings are able to support XML with flexible schemas. Also, since the table is always dense and elements from the same XML document are stored consecutively, node-encoding mappings do not suffer from the storage issues of fragmentation and sparseness.

Unfortunately, query processing of node-encoding mappings is usually expensive. The fundamental reason is that XML query processing loses all of the aforementioned schema- and normalization-based optimizations: without normalization, traversing one step along every XPath axis requires a join, even if two input elements present a one-to-one relationship; and without schema simplification, expensive constructs must be evaluated naively. A lot of effort has been made to compensate for such deficiencies, e.g., path indexes and efficient structural joins. However, every solution has additional cost. For example, since A/* may correspond to multiple expressions (e.g., A/B and A/C), the path index lookup of A/* may need to access multiple keys, and thus need to access scattered disk pages; a structural join cannot compete with a selection after normalization, no matter how efficient it is. In principle, schema- and normalization-based optimizations are at the core of query optimization; their absence cannot be fully compensated for by other techniques.

In this paper, we propose a novel mapping that resembles normalization and retains all schema- and normalization-based optimizations for XML with flexible schemas. Conceptually, individual XML documents are normalized, and all the tables are concatenated into one wide table; physically, interpreted storage is used to efficiently represent the wide table, whose columns are often sparsely-populated. At query compile time, a global inferred schema is used to simplify query constructs and to apply the normalization principle to reduce joins.

Much existing work on query processing of XML with flexible schemas adopts one or more heuristics of normalization mappings. For example, optimizations based on structural summaries can achieve simplifications of XML query constructs. However, our sparse mapping is the first work that provides all schema- and normalization-based optimizations for the target use cases. Our main contributions are:

1) A new XML-to-relational mapping that resembles normalization.
2) An algorithm to translate XPath 1.0 into SQL over the sparse representation of XML.
3) A translation framework that provides all schema- and normalization-based optimizations for query processing.
4) Experiments that show the technique has good performance for the target use cases.

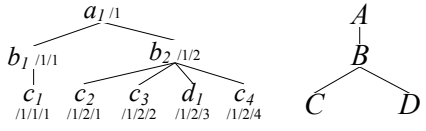
The paper is organized as follows. Section II describes the mapping rules. Section III shows how to translate XPath to SQL under the new mapping. Section IV presents an optimization framework that utilizes schema- and normalization-based optimizations. Section V covers extensions. Section VI reports on experimental results. Section VII covers related work. Section VIII concludes the paper.

II. Mapping XML to a Wide Table

In this section, we describe the mapping from XML to a wide, sparsely-populated table. We start by introducing Data Guides and a node encoding scheme. Then we elaborate the mapping rules and physical representation of the mapping table. Finally, we discuss optimizations on the mapping table.

A. Data Guide and Node Encoding

The Data Guide (DG) [18] of an XML corpus is a labelled tree that summarizes all paths in XML documents. All nodes that have the same root-to-node path are mapped into a single node in the DG so that each distinctly-labelled path appears exactly once. Figure 1(b) shows the DG for the XML document in Figure 1(a), where capitalized letters denote nodes in the DG and small letters denote elements in the XML document. Since multiple instances of the same document-label-path collapse to one path in the DG, the size of a DG is usually much smaller than that of the original documents.



(a) XML document (b) DG

Fig. 1. An XML document and its DG

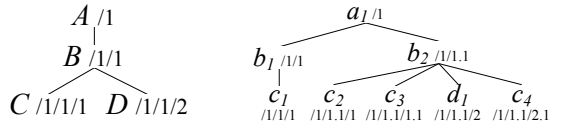
The DG of an XML corpus captures only the hierarchical tree structure, not the elements' positions in a document. To capture the latter, every element in a document is assigned an ordpath [24] which is similar to a Dewey ID [29] but provides efficient insertion and compression. Given the ordpaths of all the elements in a document, the whole XML tree can be reconstructed. Figure 1(a) shows one possible encoding of the tree, where all elements are annotated with ordpaths.

In our mapping, to encode elements in XML documents, the first step is to use ordpaths to encode DG nodes. In traditional DG's, there is no order among siblings. Here we enforce an order in the DG by assigning ordpaths to the DG nodes, as shown in Figure 2(a).

Definition 1. *The base ordpath of a DG node is the ordpath assigned to it.*

Definition 2. *The real ordpath of an XML element is the ordpath assigned to that element.*

The base ordpaths of the DG nodes are used as "boundaries" for real ordpaths of XML elements. We will see how this scheme can benefit storage efficiency later. Given the base ordpaths, the real ordpath assigned to an XML element v in



(a) DG with base ordpaths

(b) Elements' real ordpaths, based on their DG nodes' base ordpaths

Fig. 2. Ordpaths of a DG and XML elements

a document is encoded as follows:

1) The real ordpath of v is its parent's real ordpath plus the index assigned to v, called the *component* of v.
2) For collection nodes—consecutive siblings with the same element label—the component of the first element is the same as the component of its corresponding DG node. The components of the remaining elements in the collection fall in the range between the current and the next DG node. In Figure 2(b), c_2, c_3 are two collection nodes under b_2. c_2 is the first element and thus its component is the same as C in the DG, i.e., 1. c_3 is the second element and its component falls in the range between C (i.e., 1) and D (i.e., 2), i.e., 1.1.
3) For a non-collection node—the element label is unique among its siblings—its component is the same as the DG node to which it is mapped. In Figure 2(b), d_1 is a non-collection node; its component is the same as D, i.e., 2.

In practice, the order enforced in the DG might violate the document order of some XML documents. In Figure 2(b), c_4 follows d_1 as its sibling. It too is mapped to C in the DG. But since its component does not fall in the range between C and D, c_4 is called an *exception node*. To achieve a full-fidelity representation, the component of c_4 must be constrained by d_1: greater than the d_1's component. Given that there is no sibling following d_1, the component of c_4 is 2.1. So the real ordpath of c_4 is: /1/1.1/2.1. Appendix II discusses more on how to map exception nodes.

B. Mapping rules

Given the DG of an XML corpus, each DG node is mapped to a column. Each XML element (including attribute and text nodes) is stored in the column to which its corresponding DG node is mapped. Elements from the same document are stored in one or more rows consecutively. For an element v, its children are mapped as follows:

MAPPING RULE 1. *The children that are not collection elements are stored in the same row as v.*

MAPPING RULE 2. *For the children that belong to a collection, only store the first element in the same row as v. Store the other elements in separate rows.*

MAPPING RULE 3. *The children that are exception nodes are always stored in separate rows.*

Table I shows the result of applying the mapping rules to the XML document in Figure 2(b). b_1, b_2 are two elements of collection B. Since b_1 is the first element under a_1, it is stored in the same row as a_1 and b_2 is stored in a separate row. Similarly, since c_2 is the first element in collection C (under b_2), c_2 is stored in the same row as its parent b_2, and c_3 is stored in a separate row, as is the exception node c_4.

TABLE I

THE MAPPING TABLE OF THE XML IN FIGURE 2(B)

docid	A	B	D	C
5	a_1:/1	b_1:/1/1		c_1:/1/1/1
5		b_2:/1/1.1	d_1:/1/1.1/2	c_2:/1/1.1/1
5				c_3:/1/1.1/1.1
5				c_4:/1/1.1/2.1

This sparse mapping resembles normalization in the sense that if two elements are in the same row of a table after normalization, they are in the same row in this sparse-mapping table as well. In Table I, dashed lines highlight three logical sub-tables, $T_1(A), T_2(B, D), T_3(C)$, which are the tables we would get by normalizing the schema. Unlike conventional normalization, tuples in different sub-tables are connected by structural joins of the elements' ordpaths, rather than primary-foreign keys. Furthermore, tuples that are in the same sub-table but from different documents are not physically consecutive.

An important feature of this sparse mapping is that it mimics normalization without knowing the schema, because the table columns are specified by distinct paths (i.e., DG nodes) and the mapping rules are applied to individual elements. Moreover, updates and schema evolution do not require table repartitioning. For example, if a new element d_2 is inserted as d_1's sibling, the link between B and D evolves to a one-to-many relationship. The sparse mapping only needs to insert a new row, whereas conventional relational storage would partition $T_2(B, D)$ into two tables $T_4(B), T_5(D)$. (Inserting a new row does change logical sub-tables. As we will see in Section IV-A, logical sub-tables are expressed as metadata, whose maintenance cost is usually cheaper than table repartitioning.) Appendix I gives details of mapping maintenance due to updates.

C. Interpreted storage

The mapping table can be very wide, given the number of possible columns generated. Moreover, due to many-to-one relationships in XML documents, most columns are typically null. A wide sparse table would cause a huge storage blow-up in conventional RDBMSs, because conventional positional storage pre-allocates storage for all attributes and a null value usually takes at least one byte. The storage overhead would be very high when every row has thousands of columns, only a small fraction of which are not null.

Interpreted storage was proposed to handle wide sparse tables in [7], and is implemented in SQL Server 2008 [1]. Conceptually, each row in the table is represented by attribute-value pairs. Columns whose names do not appear in a row are null valued. Since only non-null values are explicitly stored, the space of a row is determined by the number of non-null entries. This property is well-suited for XML data with a large number of optional elements and attributes.

Given that our mapping maps every DG node to a column, updates to XML documents may cause the addition or deletion of DG nodes and hence the addition or deletion of columns. Thus, these schema changes must be cheap. Interpreted stor-

age provides an efficient mechanism to make such changes, since the attribute-value pairs maintain schema information for individual rows. Adding a column to the table does not affect existing rows. Removing a column requires changes only to rows that store a value for that column. By contrast, conventional positional storage may demand modifications to all the rows when the schema changes (though most RDBMSs implement schema updates lazily).

D. Optimizations

Compression We discuss the compression of the sparse representation of the XML corpus in the following.

DEFINITION 3. *The primary ordpath of a row is the longest common prefix of the ordpaths of all elements in that row.*

DEFINITION 4. *The base ordpath of a column is the base ordpath of the DG node that is mapped to that column.*

We compress each row in Table I by storing only the primary ordpath of the row. All of the other nonempty entries in the row are stored as one bit, which is set to one for elements that are present. The compressed table is shown in Table II whose primary key is (docid, p_ordpath).

TABLE II
COMPRESSED SPARSE MAPPING TABLE

docid	p_ordpath	A(bit)	B(bit)	D(bit)	C(bit)
5	/1/	1	1		1
5	/1/1.1		1	1	1
5	/1/1.1/1.1				1
5	/1/1.1/2.1				1

Table II stores only one ordpath in a row, and thus improves storage efficiency and access time. Given the primary ordpath of row x and the base ordpath of column y, the real ordpath of a non-null entry at (x, y) can be reconstructed as follows:

1) If the primary ordpath $ord_p(x)$ has the same length as the base ordpath $ord_b(y)$, then since $ord_p(x)$ is the longest common prefix of all the real ordpaths in row x, $ord_p(x)$ is also the real ordpath of the entry (x, y).
2) If the primary ordpath $ord_p(x)$ is shorter than the base ordpath $ord_b(y)$, the entry (x, y) must be the first element in a collection or a non-collection element. Otherwise, according to Mapping Rules 2 and 3, it would be stored in a separate row. Therefore, the component assigned to this element is the last component of $ord_b(y)$. The real ordpath of this element is the real ordpath of its parent plus its component, where its parent's real ordpath is derived recursively.

   Algorithmically, this process can be summarized as a single operation: $ord_p(x)||suf\big(ord_b(y), |ord_p(x)|\big)$, where $||$ denotes concatenation, and $suf\big(ord_b(y), |ord_p(x)|\big)$ denotes the suffix of $ord_b(y)$ without the first $|ord_p(x)|$ components.

EXAMPLE 1. *Consider column C in the third row of Table II. Since the primary ordpath (/1/1.1/1.1) has the same length as the base ordpath of column C (/1/1/1), the real ordpath of the entry is the primary ordpath /1/1.1/1.1. In the second row, the primary ordpath is shorter than the base ordpath of column C (/1/1.1). Since $suf\big(ord_b(C), |ord_p(2)|\big)$ = 1, the real ordpath of the entry is /1/1.1/1.*

TABLE III
THE MAPPING TABLE OF THE XML IN FIGURE 1(A)

docid	A	B	D	C
5	a_1:/1	b_1:/1/1		c_1:/1/1/1
5		b_2:/1/2	d_1:/1/2/2	c_2:/1/2/1
5				c_3:/1/2/3
5				c_4:/1/2/4

The base ordpaths in the DG are fixed boundaries for all values in a row, making the reconstruction of real ordpaths easy. By comparison, mapping the encodings in Figure 1(b), which do not use the base ordpaths as fixed boundaries, would also give a full-fidelity representation of XML, as shown in Table III. However, in that case, decoding the compressed representation would require a more expensive algorithm. For example, since the component assigned to d_1 is not fixed and determined by the number of elements in collection C, reconstructing d_1's component would result in an expensive COUNT aggregation on collection C.

A physical representation of table records after compression is shown as follows, where thin bars separate fields of a record:

5|/1|(A, 1)(B, 1)(C, 1)	5|/1/1.1|(B, 1)(D, 1)(C, 1)	.....

The first two columns (docid, p_ordpath) are stored as ordinary columns, and no attribute names are attached. Parenthesized entries after the second bar in each record are attribute-value pairs. The pairs can be further compressed, e.g., represent consecutive columns as a range. XML attribute values and text nodes can be packed in attribute-value pairs too.

Secondary Indexes A number of secondary indexes can be built on the mapping table. A *filtered index* [1] is an index structure that uses a filter predicate to index a portion of rows in the table. For the wide sparse table T, the statement

CREATE INDEX b_index on $T(B)$
WHERE B IS NOT NULL

creates an index that provides fast accesses to rows that have a non-null value in column B. Filtered indexes can also be built on columns that represent XML attribute values and text nodes.

III. XPATH-TO-SQL TRANSLATION

This section explains how to translate XPath to relational algebra expressions over the wide sparse table. The translation described here focuses on expressiveness. Schema- and normalization-based optimizations provided by the sparse mapping will be incorporated in the next section.

We first define some notation. Let T denote the wide sparse table to which an XML corpus is mapped. Without loss of generality, assume the table is not compressed, and the table entries represent real ordpaths. Let e denote an XPath expression that returns a value of one of four types, namely *node set*, *number*, *string* or *boolean*, which are represented by $R(e)$, $num(e)$, $str(e)$, $bool(e)$ respectively. In the context of relational algebra, $R(e)$ is a set of elements returned by e, represented by binary tuples $\langle id, ord\rangle$ where id is the

TABLE IV

JOIN CONDITIONS OF XPATH AXES

$e\chi e_0$	Join condition C_χ
child	$R(e).id = R(e_0).id \wedge$ $R(e).ord = R(e_0).ord.anc(1)$
parent	$R(e).id = R(e_0).id \wedge$ $R(e).ord.anc(1) = R(e_0).ord$
descendant	$R(e).id = R(e_0).id \wedge$ $R(e).ord$ is prefix of $R(e_0).ord$
ancestor	$R(e).id = R(e_0).id \wedge$ $R(e_0).ord$ is prefix of $R(e).ord$
following	$R(e).id = R(e_0).id \wedge R(e).ord < R(e_0).ord$
preceding	$R(e).id = R(e_0).id \wedge R(e).ord > R(e_0).ord$
following-sibling	$R(e).id = R(e_0).id \wedge R(e).ord < R(e_0).ord$ $\wedge R(e).ord.anc(1) = R(e_0).ord.anc(1)$
preceding-sibling	$R(e).id = R(e_0).id \wedge R(e).ord > R(e_0).ord$ $\wedge R(e).ord.anc(1) = R(e_0).ord.anc(1)$

document ID and ord is its real ordpath. Given an ordpath ord, $ord.anc(k)$ returns the k-level higher ancestor of ord. For example, $ord.anc(1)$ returns the parent of ord and $ord.anc(2)$ returns its grandparent. For compactness, we use the rename operator $\rightarrow$ in the projection list. For instance, $\pi_{id,A\rightarrow ord}(T)$ renames the projected attribute A to ord.

Base expression Let e_0 be either a tag name A or the symbol $*$. Their relational algebra expressions are:

$$R(A) = \bigcup_{A_i \in \mathcal{A}} (\pi_{id,A_i \rightarrow ord}(\sigma_{A_i \neq null} T))$$
$$R(*) = \bigcup_{X} (\pi_{id,X \rightarrow ord}(\sigma_{X \neq null} T))$$

where $\mathcal{A}$ is the set of columns whose tag names are A, and X is any column in T. For example, the algebra expression of the tag name B over Table I is: $\pi_{id,B\rightarrow ord}(\sigma_{B\neq null}T)$, which returns two binary tuples: $\langle 5, /1/1\rangle$, $\langle 5, /1/1.1\rangle$, corresponding to elements b_1 and b_2 respectively.

XPath axes XPath location paths are expressed as $e\chi e_0$ where χ is one of the axes, e is an XPath expression, and e_0 is one of the base expressions. Each axis corresponds to a relational algebra expression of the form:

$$\pi_{R(e_0).id,R(e_0).ord}\left(R(e) \bowtie_{C_\chi} R(e_0)\right)$$

where C_χ is a join condition. The conditions for all axes are shown in Table IV.

Predicates A predicate filters a node set with respect to a predicate expression pe, which also returns a value of one of the four types. Consider the expression $e[pe]$. If pe returns a node set, let pe be one or more relative location paths that start from the projection node of e. The translation of pe is the same as e except that $R(pe)$ projects the starting node. For example, for the expression $/A[./B/C]$, $R(pe) = R(A) \bowtie R(B) \bowtie R(C)$ and projects $R(A).ord$. The algebra expression for $e[pe]$ is

$$\pi_{R(e).id,R(e).ord}(R(e) \ltimes R(pe))$$

Notice that a semijoin is used, as $e[pe]$ does not project any node within pe.

If pe returns a boolean value, pe can be viewed as a function such that for each tuple t in $R(e)$, $pe(t)$ returns a boolean value. The algebra expression for $e[pe]$ is

$$\pi_{R(e).id,R(e).ord}(\sigma_{pe(t\in R(e))=\text{true}} R(e))$$

For example, for the query $/A/B/@attr[. \leq 5]$, pe is a comparison function that selects attribute nodes whose values are at most 5. The algebra expression for the query is: $\sigma_{attr\leq 5}R(e)$ where $e = /A/B/@attr$.

XPath 1.0 includes four types of functions: node set functions, string functions, boolean functions and number functions. Functions that do not contain a node set as input or output are trivial in algebra expressions. Thus, we summarize only those functions that involve a node set. For the node set $R(e)$, we use $R(e).val$ to denote string values of nodes in the set: if the node is an element, val is its tag name; if the node is a text or attribute element, val is its string value.

Position operator The function position() returns the position of every input node within its context. The conventional relational algebra operators only manipulate sets of tuples, which have no order. Still it can be defined by a relational algebraic operator that corresponds to operations that are available in many RDBMSs. For example, the rank() function in T-SQL in Microsoft SQL Server can be used to rank the inputs and give the tuple positions in the sorted list, which can reconstruct the position function. The relational algebraic expression to calculate position() in the XPath expression is

$$position()[\![e\chi e_0]\!] =_{R(e).ord} G_{rank(R(e_0).ord)}(R(e) \bowtie R(e_0))$$

where G is the grouping operator and $rank()$ gives each tuple's position in each group. The left subscript of G defines the group-by column, i.e., the set of rows with the same value of this column. The right subscript is the function to apply to each set defined by the left subscript, where the rows are sorted by increasing value of ord. The expression returns a set of triples $\langle id, ord, rk\rangle$ where id and ord identify the element and rk is the rank of the element according to the above expression. When position() is used in the predicate, a selection on rk is added. For example, $e\chi e_0[position() = 7]$ is translated to:

$$\pi_{id,ord}(\sigma_{rk=7}({}_{R(e).ord}G_{rank(R(e_0).ord)\rightarrow rk}(R(e) \bowtie R(e_0))))$$

Aggregation The function count(e) returns the number of nodes in a node set. Its algebra expression is:

$$count(e) = G_{count}(R(e))$$

Other functions Table V lists other functions that involve node sets. For similar functions, only one of them is listed. In the third row, I is a list of columns corresponding to ID attributes[1] in the DG.

TABLE V

OTHER OPERATIONS IN XPATH

Operator	Relational algebra
$\mathcal{F}$ [sum: $nset \rightarrow num$] $(R(e))$	$G_{\text{sum}}(\pi_{\text{to\_num}(val)}R(e))$
$\mathcal{F}$ [sum: $nset \rightarrow str$] $(R(e))$	$G_{\text{concatenate\_str}}(\pi_{(val)}R(e))$
$\mathcal{F}[id : str \rightarrow nset](str)$	$\cup_I (\pi_{id,\text{realOP}(I)}(\sigma_{I=str}T))$
$\mathcal{F}[id : nset \rightarrow nset]$ $(R(e))$	$\cup_{r\in R(e)}\ \mathcal{F}[id](r.val)$
$\mathcal{F}$ [RelOp: $nset \times nset$ $\rightarrow bool$] $(R(e_1), R(e_2))$	$R(e_1) \bowtie_C R(e_2)$ where C: $R(e_1).id = R(e_2).id \wedge$ $R(e_1).val$ RelOp $R(e_2).val$
$\mathcal{F}$ [RelOp: $nset \times num$ $\rightarrow bool$] $(R(e), v)$	$\sigma_{val\ \text{RelOp}\ v}(R(e))$

IV. TRANSLATION OPTIMIZATIONS

In this section, we incorporate schema- and normalization-based optimizations in query translation. The DG is the structural summary of an XML corpus and mimics the underlying

[1]http://www.w3.org/TR/xpath/#unique-id

schema. Hence, the DG can be used to simplify XML query constructs. The sparse-mapping table mimics normalization through logical sub-tables. Though the algebra expressions discussed previously operates on columns in a single wide table, by identifying logical sub-tables, we can reduce joins to selections, if the joins are on columns in the same sub-table.

A. Augmenting the DG

We first augment the DG definition to accommodate normalization-based optimizations. The key to enable such optimizations is the identification of sub-tables—which columns would form a logical sub-table if we normalized the schema. To this end, each node in the DG is annotated by one of the following symbols, indicating the number of occurrences of this DG node under its parent in the XML corpus: (1) *****: zero or more; (2) **+**: at least one; (3) **?**: zero or one; (4) **!**: exactly one occurrence. Also, if exception nodes are mapped to a DG node, this node is annotated with **#**.

In addition to the symbols, each DG node n is assigned an alias of T with an integer subscript such that

1) If n is the root, assign T_1 to it.
2) If n is annotated with *, + or #, assign a new alias to it.
3) Otherwise, assign its parent's alias to it.

Figure 3 shows the annotated DG of the document in Figure 2(b).

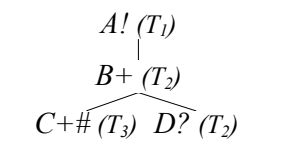


Fig. 3. The annotated DG

Aliases track one-to-one and many-to-one relationships in the hierarchies of XML documents. According to normalization, two entities with a one-to-one relationship can be stored in the same table, without violating normal forms. Therefore, aliases are identifications of sub-tables: columns with the same aliases form a logical sub-table.

Since exception nodes are stored in separate rows than their parents, mapping an exception node to a column is equivalent to partitioning this column from the parent sub-table. Hence, if a DG node is annotated with #, it has a new alias.

Maintenance of annotated DG The annotated DG is generated from the initial corpus. When the corpus is updated, the corresponding DG nodes are updated. Inserting an element to an existing document or adding a document to the corpus may change a one-to-one relationship between a parent and child DG node to a many-to-one relationship. This change demands a new alias for the child, which requires re-numbering aliases for the child's descendants. Since the DG is typically small and kept in main memory, these updates can be done instantly.

Deleting an element from a document or removing a document from the corpus may change a many-to-one relationship between a parent and child DG node to a one-to-one relationship. This change, however, cannot be completed until we scan the corpus and check whether another document implies a many-to-one relationship between these two DG nodes.

Such DG updates upon deletion are not necessary for translation correctness. When an XML query is translated to SQL, the annotations identify logical sub-tables, which are used to eliminate joins. If we falsely annotate a many-to-one relationship between two DG nodes that are supposed to be related one-to-one, the translation is still correct but only suboptimal. Therefore, updates of annotations due to deletion can be deferred until indexes are rebuilt on the XML database.

In essence, DG annotations are metadata that maintain normalization, i.e., identifications of logical sub-tables. Maintaining metadata is usually much cheaper than manipulating original data upon schema evolution. For example, in normalization mappings, schema evolution often results in table repartitioning, which is expensive when the table is large.

B. Framework

A high level representation of the translation framework is shown in Figure 4. The evaluation module and the rewriting module perform the schema- and normalization-based optimizations respectively. The evaluation module receives a parsed query, evaluates it over the DG, and translates it into an algebra tree. In the translation, a column selection uses the table alias assigned to the corresponding DG node. For example, the selection of column B is $\sigma_{B\neq null}(T_2)$ where T_2 is the table alias of the B node in the DG in Figure 3. Query constructs are simplified in two ways:

1) The algebra expressions of XPath axes only includes the columns that potentially satisfy the query. Consider the query $/A/*$. By the translation of the base expression in Section III, the algebra expression is a join between column A and the union of all other columns. Evaluating the query over the DG only keeps those columns that appear as A's children in the DG, i.e., $\sigma_{A\neq null}(T_1) \bowtie \sigma_{B\neq null}(T_2)$.
2) The join conditions of hierarchical axes are reduced to equi-joins. For example, the original join condition of $A//C$ is: $R(A).ord$ is a prefix of $R(C).ord$. Given that C is a grandchild of A in the DG, the join condition can be rewritten to $R(A).ord = R(C).ord.anc(2)$. Recall that tuples across sub-tables are connected through structural joins. Translating to equi-joins simplifies the join conditions, and simulates primary-foreign key joins (which are equi-joins) across sub-tables.

When the query is evaluated over the DG, hierarchical axes (i.e., child, descendant, etc.) can be matched precisely. However, since the order between the DG nodes is enforced, it may not fully reflect the actual order in the XML documents. To guarantee the correctness of the translation for axes related to the document order (e.g., following-siblings), all the DG nodes that *might* satisfy the condition must be included in the algebra expression. For example, consider the query $//C$/following-sibling::* over the DG in Figure 3. While D is the following sibling of C, the C node is annotated by +, which means that multiple c nodes may appear as siblings in an XML document. Therefore, in the translation, column C should join both column D and itself. Furthermore, exception nodes violate the order of the DG

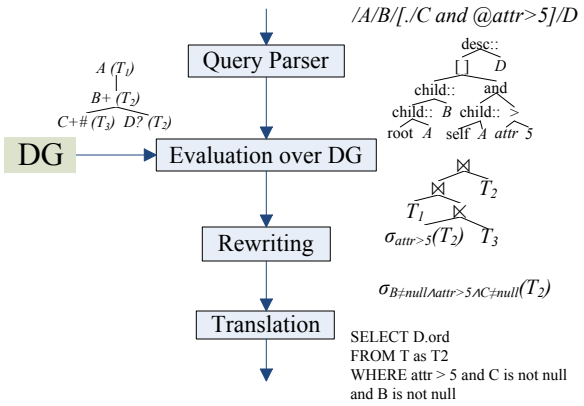


Fig. 4. Translation framework

nodes and their positions in the documents are not predictable through the DG. Consequently, they should be considered as well. Consider the query $//D/$following-sibling::C. In Figure 3, although C precedes D, since C is annotated with #, the translated expression is $\sigma_{D\neq null}(T_2) \bowtie \sigma_{C\neq null}(T_3)$, because "exception" means that some c elements may follow a sibling d element.

Given the algebra tree, the rewriting module rewrites the algebra tree based on a set of rewriting rules. Annotations on the DG are used as hints to infer if a join can be eliminated or simplified to a selection. The simplified algebra tree is finally translated into a SQL query.

C. Algebra Rewriting

Next, we discuss rewriting rules to minimize the number of joins. The rewriting coincides with the normalization principle and leverages characteristics of XML query languages.

REWRITING RULE 1. $\sigma_{s_1}(T_1) \bowtie \sigma_{s_2}(T_1) \rightarrow \sigma_{s_1 \wedge s_2}(T_1)$, *if*

(1) the join is an equi-join.
(2) s_2 does not contain a position predicate.

Equi-joins correspond to hierarchical axes in XPath, e.g., child, descendant, etc. Columns with the same alias are in the same logical sub-table. Therefore, the join can be simplified by merging the two selection predicates.

In XPath, elements' positions are dependent on context nodes. When s_2 contains position predicates, T_1 specifies the context nodes. Hence, the join cannot be simplified.

REWRITING RULE 2. $\sigma_{s_1}(T_1) \bowtie \sigma_{s_2}(T_2) \rightarrow \sigma_{s_2}(T_2)$, *if*

(1) the join is an equi-join.
(2) s_1 is a single "not-null" predicate.
(3) s_2 does not contain a position predicate.

XPath axes only project on the target node. For example, for the query $//B/C$, when there is no predicate on B, all the elements in column C satisfy the query and are projected. Therefore, the join can be eliminated. However, if there is a predicate on B, e.g., $//B[@attr = 5]/C$, in Table I, b_1 or b_2 may be filtered out and some elements in column C will not match a b element any more. In such a case, a join is required to associate c elements with satisfied b elements.

REWRITING RULE 3. $\sigma_{s_1}(T_1) \ltimes \sigma_{s_2}(T_1) \rightarrow \sigma_{s_1 \wedge s_2}(T_1)$, *if*

(1) the semijoin is an equi-semijoin.
(2) s_2 does not contain a position predicate.

Semijoins correspond to path expressions within predicates. Similar to Rewriting Rule 1, the semijoin satisfying the conditions can also be simplified by merging the selection predicates of the input relations.

REWRITING RULE 4. $\sigma_{s_1}(T_1) \ltimes \sigma_{s_2}(T_2) \rightarrow \sigma_{s_1 \wedge s_2}(T_1)$, *if*

(1) the semijoin is an equi-semijoin.
(2) s_2 is a single not-null selection condition.
(3) in the path between the two projected DG nodes of T_1 and T_2, either (a) no +/ node is followed by a */? node, and no node is annotated by #, or (b) there are only +/! nodes (and # can co-occur).*

The semantics of the path expression within a predicate is "existence". That is: as long as one such path exists, the context node survives. In the translation, if it is guaranteed that every element in T_1 can be joined with an element in T_2, the join can be eliminated. Consider the query $/A[./B]$ where B is annotated by + in Figure 3, i.e., one or more b elements under an a element. According to Mapping Rule 2 in Section II, if the B collection has at least one element, a b element must appear in the same row as a. Therefore, the join can be simplified to a selection $\sigma_{A\neq null \wedge B\neq null}(T_1)$.

This property is transitive only within DG nodes annotated by + or !. Consider the query $A[.//D]$. In Table I, d_1 and a_1 are in separate rows, though d_1 is still a descendant of a_1. More formally, if either * or ? follows * or + (not necessarily consecutively) in the path from A to D in the DG, then the only $a - d$ path in an XML document might have endpoints that do not appear in the same row. Likewise, if there exists an exception in the path, the two endpoints of the path may not be in the same row, unless there are only + and !, which guarantees a path in that row.

Similar to Rewriting Rule 2, if T_2 contains additional predicates, e.g., $/A[./B[@attr = 1]]$, then the b node that is in the same row as a may be filtered out. However, there may be another row that associates a to a surviving b node. Hence, the join cannot be simplified to a selection.

EXAMPLE 2. *Let $\perp$ denote the null value. Consider the XPath query /A/B[./C and @attr>5]/D over the DG in Figure 3. The original algebra tree of the query is shown in Figure 5(a).*
(a) The semijoin

$$\sigma_{B\neq\perp\wedge attr>5}(T_2) \ltimes \sigma_{C\neq\perp}(T_3)$$

is rewritten using Rewriting Rule 4. Since the input relations correspond to B and C in the DG and there are only + and ! between them (even though # co-occurs), this semijoin can be eliminated, as shown in Figure 5(b).
(b) The join

$$\sigma_{A\neq\perp}(T_1) \bowtie \sigma_{B\neq\perp\wedge attr>5}(T_2)$$

is further simplified by Rewriting Rule 2, yielding Figure 5(c).
(c) Finally, the join expression in Figure 5(c) is simplified into a single selection by Rewriting Rule 1:

$$\sigma_{B\neq\perp\wedge attr>5\wedge D\neq\perp}(T_2)$$

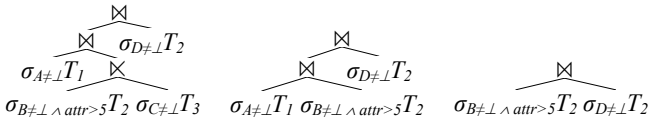


Fig. 5. Algebra rewriting for the example query

Rewriting queries using indexes There is only one table in the sparse mapping. An index on one column may also provide fast access for another column. For example, consider a selection on one column, e.g., $\sigma_{A\neq null}(T)$, a frequent operation in translated algebra expressions. If there is no index on column A, the selection requires a full table scan. If the table has an index on column B and the DG indicates that every row with an a value must have a b value, then rewriting the query into $\sigma_{A\neq null\wedge B\neq null}(T)$ utilizes the index on B to first filter out unrelated rows and avoid a full scan of the table.

In general, DG nodes with the same alias can use the same filtered index for column selections. Since these columns form a logical sub-table, such a filtered index essentially provides fast accesses to the sub-table, even though tuples from the sub-table are not consecutive in physical storage.

V. Extensions

Applicability for XML with a static schema While the sparse mapping was designed for the target use cases, it also has great potential for XML data with static schema. Physical storage of documents in the sparse mapping follows the same principle of normalization. With the annotated DG, query rewriting can reduce all unnecessary joins. Therefore, for XML data with static schema, the execution plan of a query in the sparse mapping has at most the same number of joins as a normalization mapping (the Rewriting Rule 4 may cause fewer joins). For this reason, we believe that the sparse mapping would be competitive on standard benchmarks—a few large XML documents conforming to a static schema; further testing will be required to confirm this conclusion.

The extra cost of the sparse mapping, compared with normalization mappings, is the storage overhead of each row, because interpreted storage maintains schema information for individual rows. In addition, rows have variable length in the sparse mapping. Some may even be very long. This increases the row-access cost.

Extending to XQuery Until now we have discussed XPath translation over the sparse mapping. The translation algorithm can be extended to XQuery as well. In the literature, several papers have studied comprehensive translation from XQuery to SQL, e.g., [13], [19], in which relational algebra is based on the node-encoding mapping table. In our mapping, the output table of a translated algebra expression has the same semantics as the algebra expressions under the node-encoding mapping. That is: for an XPath expression e that represents a node set, $R(e)$ represents a binary relation $\langle docid, ordpath\rangle$, which is the same as algebra expressions in the node-encoding mapping. After XPath expressions are translated into algebra expressions under the sparse mapping, techniques discussed in [13], [19] can be used to translate the other XQuery constructs.

TABLE VI
Real-life Data Set Parameters

Table	doc #	Document size (KB)			
		average	min	max	DG
T1	21,875	2.02	1.84	2.4	5.59
T2	152,303	0.94	0.19	1.33	6.52
T3	50,000	18.47	17.36	19.71	58.65
T4	130,528	1.76	1.04	48.24	2.97

VI. Experimental Results

We experimentally evaluate the new mapping and the translation algorithm in this section. Several XML benchmarks have been designed for XML processing, e.g., XMark [27] for analytical queries and TPoX [23] for transactional queries. However, none of them capture our target use cases, i.e., many small XML documents with flexible schemas. Characterizing such cases is not trivial. At one extreme, all XML documents in the database have the same static schema; at the other extreme, every XML document has its own schema and distinct tag names. An ideal benchmark should identify a small range of middle cases that capture the needs of most enterprise applications and quantify the data characteristics.

We use two data sets to measure performance. The first data set and its query workloads are from real-life customer verticals, such as banking, insurance and life sciences. The data set is characterized in Table VI, where the "DG size" is the size of the XML document to which DG is serialized. The schemas of T1-T4 are [ID varchar(200) primary key, doc xml], where ID denotes the document id, and doc is a column declared as the xml type. As we can see, average document sizes from three document sets are smaller than 4 KB, which means that most documents fit in a single disk page. Furthermore, DG sizes are 1.6-7 times larger than the average document sizes. This indicates that schemas of individual documents are much smaller than the global schemas of the document sets. These documents must have many optional elements/attributes, and their schemas are very flexible.

The second data set is derived from XMark. Although the original XMark document does not satisfy the properties of the use cases we target, its query workloads have been well studied. To simulate the desired properties, we implement a data generator that inputs one large XMark document and randomly strips it down to a much smaller document. Specifically, for every many-to-one relationship in XMark, the generator randomly picks a small number (at most three) of elements in the "many" side. The generator is run repeatedly to generate 40,000 documents. The total space of the documents is 424MB. XMark's schema is assumed to be unavailable. Query workloads of the two data sets are shown in Table VII.

XML query specifications, e.g., XQuery and SQL/XML, include functions to retrieve and construct XML data. An XML query can return document IDs, full documents, or fragments constructed from elements selected by the query. XML reconstruction involves a number of joins to connect retrieved elements. EXRT [11], a recent benchmark, has revealed that XML reconstruction can be cheap or fairly

TABLE VII

QUERY WORKLOADS

	Real-life
R1	FROM T1 WHERE doc.exist('/row/E2[./text()="AB"]') = 1 and doc.exist('/row /E1 [./text()=19991031] ') = 1
R2	FROM T1 WHERE doc.exist('/row/E10[contains(.,"1_NAME")]') = 1
R3	FROM T2 WHERE doc.exist('/row//E6')=1
R4	FROM T2 WHERE doc.exist('/row/E6[./text()]')=1
R5	FROM T2 WHERE doc.exist('/row[./E6/text()=20061130 and ./E5/text()=60]')=1
R6	FROM T2 WHERE doc.exist('/row/descendant::E7/text()')=1
R7	FROM T2 WHERE doc.exist('/row[descendant::E7]')=1
R8	FROM T2 WHERE doc.exist('/row/E7[.=21]')=1
R9	FROM T2 WHERE doc.exist('/row[./E8/text()=./E9/text()]')=1
R10	FROM T2 WHERE doc.exist('/row/E6[@A1 = "3"][. = " 20080213 "] ')=1
R11	FROM T3 WHERE doc.exist('//E12/text()[. $\geq$ "20051001" and . $\leq$ "20051031"]') = 1
R12	FROM T3 WHERE doc.exist('//E13[E14/text() = ("HENRY")]') = 1
R13	FROM T3 WHERE doc.exist('//E13[E15/text() = "JAMES"]') = 1
R14	FROM T4 WHERE doc.exist('E16/E17[@A2="N"]/E19[.="ABC (DEF, 3GH)"]')=1
R15	FROM T4 WHERE doc.exist('E16/E17[@A2="IJKLMN"]/E18[. = 4]') = 1
R16	FROM T4 WHERE doc.exist('//E18[. $\geq$ 30 and . $\leq$ 60]') = 1
	XMark
X1	FROM Xmark WHERE doc.exist('/site/closed_auctions / closed_ auction / annotation / description / parlist / listitem') = 1
X2	FROM Xmark WHERE doc.exist('/site//europe//mail') = 1
X3	FROM Xmark WHERE doc.exist('/site/regions/*/item') = 1
X4	FROM Xmark WHERE doc.exist('/site/regions/*/item[parent :: namerica]') = 1
X5	FROM Xmark WHERE doc.exist('/site/people/person[./address and (./phone or ./homepage)]') = 1
X6	FROM Xmark WHERE doc.exist('/site/regions/*/item[@id = "item0" and ((./mailbox/mail/from) or (./mailbox/mail/to))]') = 1
X7	FROM Xmark WHERE doc.exist('/site/people/person[./profile/ education/ text() = "Graduate School" and ./ address/ country/ text() = "United States"]/ name') = 1
X8	FROM Xmark WHERE doc.exist('/site//closed_auction/annotation [./happiness/text() = '1']//parlist') = 1
X9	FROM Xmark WHERE doc.exist('/site//person//province [contains(., "Virginia")]') = 1
X10	FROM Xmark WHERE doc.exist('/site/open_auctions/ open_auction / bidder[position() $<$ 3]') = 1

expensive, depending on the "width" of the touched data. In our experiments, XML queries only project document IDs. This eliminates the effect of XML reconstruction, and helps us focus on the performance of XML query evaluation.

In the future, we plan to explore reconstruction techniques and study their performance under the new mapping. Intuitively, the sparse mapping can benefit the reconstruction in two respects. First, the sparse mapping provides efficient storage representations of individual documents. Structural joins of the reconstruction do not need to access many pages. Second, the mapping stores in one row elements that have one-to-one relationships. With the annotated DG, connecting such elements does not need to perform joins.

We implemented the new mapper and translation framework on top of Microsoft SQL Server. Columns in the sparse-mapping table are declared as sparse columns in SQL Server 2008. Filtered indexes are created on the mapping tables of T3, T4 and the XMark table, for the XML elements that appear in the query workloads (Recall that columns from a sub-table can share the same filtered index.) T1 and T2 have no secondary indexes on the sparse-mapping table, as their data sizes are small. Translated SQL queries are evaluated by the relational engine. All the experiments are performed on a server with an Intel Core2 Quad CPU 2.5GHz, 4G RAM, and Windows Server 2008 R2.

A. Baseline

To evaluate query performance under the new mapping, we compare the sparse mapping with node-encoding mappings. As discussed in the introduction, the latter are the only alternatives for the target use cases in SQL-based XML databases.

The node-encoding mapping is implemented in SQL Server as a *primary XML index* [25]. A simple representation of the mapping table is (docid, ordpath, tag name, value, path). Every XML element is mapped to a row in the table. For XML elements that have no value, the value column is null. The path column materializes root-to-node paths for elements, which is the key to improving query performance.

A number of secondary indexes can be built on the node-encoding mapping table. In particular, with indexes on the path column, path expressions in XML queries can be evaluated by index lookups, which reduces the number of structural joins significantly. We explore three indexes, and compare them with the sparse mapping.

- The *PATH index* is built on (path, value) of the mapping table. It locates an element first by its path, then by its value.
- The *VALUE index* is build on (value, path), which indexes the same columns as the PATH index, but in reverse order.
- The *PROPERTY index* is built on (docid, ordpath, path, value). Since it contains the primary key of the mapping table, it helps search multi-valued properties in the same XML document.

The performance differences of the three indexes are determined by predicate selectivities. Consider the query $/A/B/@attr[. < 5]$. If the attribute predicate is highly selective, using the VALUE index leads to the best performance. If the path expression $/A/B/@attr$ is much more selective than the attribute predicate, then using the PATH index is more efficient.

B. Results

We measure two metrics in the experiments: query execution time and logical reads. The number of logical reads is a measure of the total data requested by the database engine in the process of query evaluation. A logical read occurs every time the database engine requests a page from the buffer cache. A physical read is triggered to read a disk page into the buffer cache, when the requested page does not reside in the cache. All numbers from the two data sets are shown in Figures 6 and 7, in which the x-axis represents the query ID. Queries on the x-axis are classified into two groups, separated by a white space. The first group are pure path queries, including wild card * and //. The second group are the remaining queries, including predicates, tree patterns, and other query constructs.

Path queries We first examine path queries. As expected, the PATH index outperforms the other two indexes in both data sets, as it indexes the elements' paths directly. Evaluating a path query over the sparse mapping consists of two steps: (1) matching the path over the DG, which locates the columns containing the result elements; and (2) column selections.

In both data sets, the sparse mapping outperforms the PATH index by a factor of three or more, with only one exception: X1. Query X1 consists of only parent-child axes. Its execution only needs to locate a single key in the PATH index, and thus can be very efficient. The other path queries, however, include wild card * or ancestor-descendant axes. Such expressions

may correspond to multiple paths. For example, $A/*$ can be A/B, $A/C, \ldots$, and $A//B$ can be $A/B, A/X/B, A/X/Y/B, \ldots$. Evaluating such expressions must access multiple index keys to locate all satisfied paths. Although there are not many distinct keys in the index (i.e., distinct paths), every key indexes a list of elements, which is too large to fit in main memory. PATH index lookups then need to access scattered disk pages.

The sparse mapping, on the other hand, simplifies query constructs using the DG and translates them to column selections. Since the DG is kept in main memory, query simplification incurs no disk access. If a column has no filtered index, its selection is evaluated by a table scan; otherwise, the filtered index directly points to the rows to access. By comparison, path indexes mix various paths in one index tree, a great portion of which must be traversed at runtime.

In short, though path-based indexes can reduce some query expressions to index lookups, they still have significant overhead, compared with schema-based simplifications.

Queries with predicates and tree patterns Next, we examine queries with predicates and tree patterns, which include all queries from the second group except X10. Execution plans of these queries under the node-encoding mapping are more complicated. It is not straightforward to determine which index will have the best performance for each query. In fact, index selection is a difficult problem for query processing of the node-encoding mapping.

Compared with the best performance of the three indexes, the sparse mapping is 1.1 to 63 times faster. The speedup is due to schema-based simplification and normalization-based join reductions. Schema-based simplifications reduces the complexity of query constructs at compile time, yielding much fewer data accesses at runtime, as demonstrated by the number of logical reads. The join reduction reduces CPU cycles, as selections are much cheaper than joins.

Queries R11-R13 over T3 present limited improvements (1.1-3 times faster) of the sparse mapping. This is because the three queries include value predicates on text, all of which are highly selective. The cost of I/O and CPU computation is very low, after the predicate pruning.

Position function Query X10 contains a position predicate. The sparse mapping does not show advantages over the three indexes. The position predicate prevents join reduction. Its physical plan includes three operators: join, grouping and sorting in each group. The cost of these operators dominates the overall performance of X10.

Further analysis of join reduction Though query processing in databases is I/O bound in most cases, some queries here show the opposite. For example, for queries R2-R8 and R10 from the real-life data set, the sparse mapping accesses about the same amount of data (in terms of logical reads) as the three indexes, though its query efficiency consistently outperforms them. It even accesses more data than the PATH index for R7.

Data access pattern is one reason that could lead to this result. To access the same amount of data, sequential reads are much faster than random reads. Query evaluation under the node-encoding mapping relies heavily on index lookups, which are mainly random reads. Query translation under the sparse mapping compiles path and tree expressions into columns selections. Given that T1 and T2 have no secondary indexes on the sparse-mapping tables, these queries are evaluated by sequential reads.

Another reason that explains the result is join reduction. Recall that most documents from T1 and T2 are smaller than 4 KB. A single document can even fit in the on-chip cache. Query evaluation on a document is CPU-bound once it is loaded into the on-chip cache. The sparse mapping resembles normalization and reduces unnecessary joins to selections, which are much cheaper than structural joins.

VII. Related Work

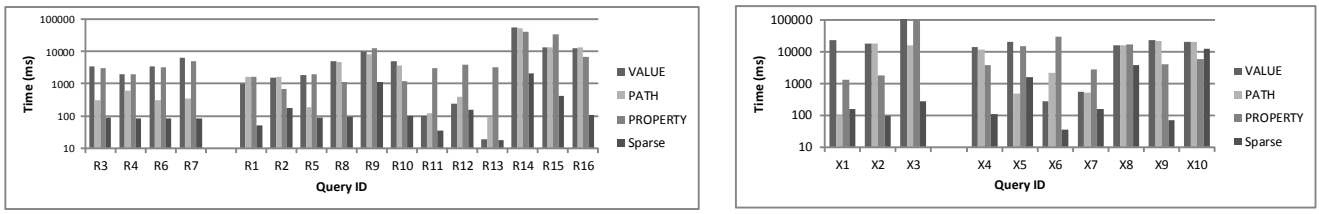
Our work is related to building XML databases on top of relational engines. Most such efforts were discussed in the introduction. In this section, we discuss other related work, e.g., native XML databases, from two viewpoints: schema- and normalization-based optimizations.

Using schemas to simplify query constructs is a straightforward optimization. Query minimization [2], [12] aims to use schemas and constraints to minimize tree patterns at query compile time. This idea can be extended by the integration of structural summaries (in case the schema is not available) and indexes, in which XML elements are linked to the corresponding summary nodes [18], [6], [15]. This enables the engine to simplify query patterns at compile time and to quickly access instances at execution time.

However, structure indexes are not the only schema-based optimizations. Schemas or structural summaries can be coupled with underlying storage as well, i.e., schema-driven storage. To understand the difference, consider node-encoding mappings in which all XML elements are stored in a single pivot table. In those systems, even if queries are simplified, locating the matched elements cannot avoid accessing many irrelevant elements, because those elements are not physically separated.

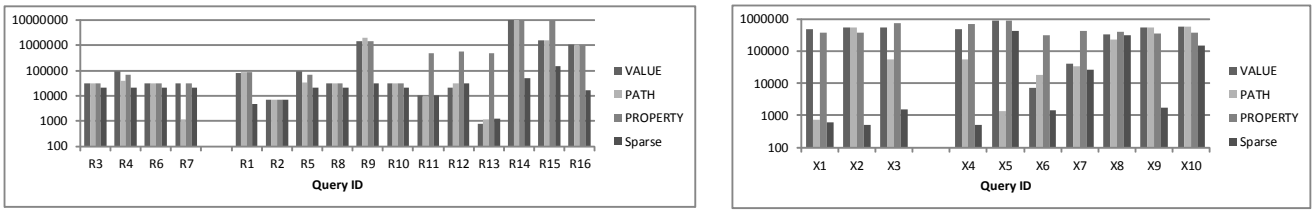
Schema-driven XML storage techniques proposed in the literature are based on document partition, e.g., by tag names [17] or by paths [4]. The benefit is that once a query expression is simplified to concrete paths, only corresponding elements need to be scanned. Moreover, such techniques facilitate compression [3], [20], as it has been observed that values under the same root-to-node paths are similar. A more popular category of native XML storage aims at locality-oriented partition [22], [8]: XML trees are partitioned into physical records, in a depth-first or breadth-first way, so that a query expression may only need to touch a few records.

While schema-based optimizations have been adopted in many systems, normalization-based optimizations have never been achieved for flexible schemas. The partition schemes of existing XML storages do not consider one-to-one relationships, and evidently incur many joins in query execution. In fact, path-partitioned storage can be viewed as a column store



(a) Real-life data sets (b) XMark benchmark

Fig. 6. Query execution time (smaller is better)



(a) Real-life data sets (b) XMark benchmark

Fig. 7. Number of logical reads (smaller is better)

of the sparse-mapping table: elements from one column have the same path. Analogous to relational column stores, path-partitioned storage facilitates value compression, but introduces more joins. From this perspective, the sparse-mapping table can be vertically partitioned as well, as long as logical sub-tables are kept holistically. Each partitioned wide table then evolves separately.

VIII. Conclusion

Schema and normalization have a profound impact on query processing in database systems. While XML is an ideal model for applications with flexible schemas, conventional query processing for such XML data loses all schema- and normalization-based optimizations. In this paper, we proposed a novel mapping that resembles normalization for XML with flexible schemas. The key is to retain normalization through logical sub-tables in one wide sparse table, which is physically represented by interpreted storage. Logical sub-tables are maintained as metadata—the annotated DG—to minimize the cost of schema evolution. At query compilation time, the annotated DG is used to eliminate joins. Experimental results demonstrate significant improvements of query performance under the new mapping.

The sparse mapping can have many variations. For example, instead of mapping an entire XML document, the system could map only parts of the document that are queried. As future work, it would be useful to study which combinations of documents and workload this mapping is and is not well suited. Even more useful would be a system that automatically select a mapping and translate queries with minimum user input (sample documents, sample query workload).

References

[1] S. Acharya, P. Carlin, C. A. Galindo-Legaria, K. Kozielczyk, P. Terlecki, and P. Zabback. Relational support for flexible schema scenarios. *PVLDB*, 2008.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[3] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed xml data. In *EDBT*, 2004.

[4] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern xml databases. *World Wide Web*, 11(1), 2008.

[5] A. Balmin, K. S. Beyer, F. Özcan, and M. Nicola. On the path to efficient xml queries. In *VLDB*, 2006.

[6] A. Barta, M. P. Consens, and A. O. Mendelzon. Benefits of path summaries in an xml query optimizer supporting multiple access methods. In *VLDB*, 2005.

[7] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending rdbmss to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.

[8] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, R. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System rx: One part relational, one part xml. In *SIGMOD*, 2005.

[9] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.

[10] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD*, 2006.

[11] M. J. Carey, L. Ling, M. Nicola, and L. Shao. Exrt: Towards a simple benchmark for xml readiness testing. In *TPCTC*, 2010.

[12] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB*, 2003.

[13] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *SIGMOD*, 2003.

[14] A. Deutsch, M. F. Fernández, and D. Suciu. Storing semistructured data with stored. In *SIGMOD*, 1999.

[15] G. H. L. Fletcher, D. V. Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of xpath with structural indexes for xml documents. *Inf. Syst.*, 34(7), 2009.

[16] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 1999.

[17] H. Georgiadis and V. Vassalos. Xpath on steroids: exploiting relational engines for xpath performance. In *SIGMOD*, 2007.

[18] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[19] T. Grust, S. Sakr, and J. Teubner. Xquery on sql hosts. In *VLDB*, 2004.

[20] H. Liefke and D. Suciu. Xmill: An efficient compressor for xml data. In *SIGMOD Conference*, 2000.

[21] Z. H. Liu and R. Murthy. A decade of xml data management: An industrial experience report from oracle. In *ICDE*, 2009.
[22] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy. Towards an enterprise xml architecture. In *SIGMOD*, 2005.
[23] M. Nicola, I. Kogan, and B. Schiefer. An xml transaction processing benchmark. In *SIGMOD*, 2007.
[24] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD*, 2004.
[25] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. V. Zolotov. Indexing xml data stored in a relational database. In *VLDB*, 2004.
[26] M. Rys, D. D. Chamberlin, and D. Florescu. Xml and relational database management systems: the inside story. In *SIGMOD*, 2005.
[27] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, 2002.
[28] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, 1999.
[29] 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 *SIGMOD*, 2002.

APPENDIX I
MAPPING TABLE MAINTENANCE

This section explains the maintenance of a compressed mapping table. Updating the mapping table when a document is added or removed is trivial: the corresponding rows are inserted or deleted. We focus on inserting/deleting an element v to/from an existing XML document whose id is $docid_0$.

Non-collection node If v is mapped to a non-collection node, it should be stored in the same row as its parent. Insertion (deletion) requires adding (removing) a column to (from) its parent's row. Let V be the DG node to which v is mapped, and $ord_p(parent(v))$ be the primary ordpath of the row containing v's parent. The corresponding SQL queries are shown in the first half of Table VIII.

Collection node If v is mapped to a collection node, two cases need to be considered: (1) if v is not the first element in the collection, v is in a separate row from its parent. Inserting/deleting v to/from the document is equivalent to inserting/deleting one row to/from the table. (2) If v is the first element in the collection, inserting v will first move the former first element $\check{v}$ plus $\check{v}$'s descendants in that row to a new row, and then store v in the parent's row; deleting v will first replace the v entry by null, and move the entries in the row of the new first element (if there is one) to the parent's corresponding entries.

Let $realOP()$ be a function that computes the real ordpath of a node. If v is not the first element in the collection and is stored in a separate row, $realOP(v)$ is also the primary ordpath of that row. If v is the first element in the collection, let $\check{v}$ be the first element in the collection before v's addition, $\hat{v}$ be the first element in the collection after v's deletion, and $V_{d_1}, \ldots, V_{d_m}$ be the descendants of V in the DG. The SQL queries that correspond to the update of a collection node are shown in the second half of Table VIII.

TABLE VIII
SQL QUERIES FOR MAPPING MAINTENANCE

Inserting/deleting a non-collection node
Insertion: UPDATE T SET V = '1' WHERE docid = $docid_0$ AND ordpath=$ord_p(parent(v))$
Deletion: UPDATE T SET V = NULL WHERE docid = $docid_0$ AND ordpath=$ord_p(parent(v))$
Inserting/deleting a collection node
Insertion (non-first): INSERT INTO T (docid, ordpath, V) VALUES ($docid_0$, $realord(v)$, '1')
Deletion (non-first): DELETE FROM T WHERE docid = $docid_0$ AND ordpath = $realOP(v)$
Insertion (first): INSERT INTO T SELECT docid, $realOP(\check{v})$, $V_{d_1}, \ldots, V_{d_m}$ FROM T WHERE docid = $docid_0$ AND ordpath = $ord_p(parent(v))$; UPDATE T SET V = '1' WHERE docid = $docid_0$ AND ordpath = $ord_p(parent(v))$;
Deletion (first) UPDATE T SET $V = E.V, V_{d_1} = E.V_{d_1}, \ldots, V_{d_m} = E.V_{d_m}$ FROM (SELECT $V, V_{d_1}, \ldots, V_{d_m}$ FROM T WHERE docid = $docid_0$ AND ordpath = $realOP(\hat{v})$) AS E WHERE docid = $docid_0$ AND ordpath = $ord_p(parent(v))$;

APPENDIX II
MAPPING EXCEPTION NODES

There can be different ways to map ordered siblings in XML documents to ordered DG nodes, which may result in different exception nodes. For example, c and d elements are interleaved in the document fragment in Figure 8(b) whose DG is Figure 8(a). Thus, some nodes will be mapped to exception nodes. Mapping c_1, c_2 to C as normal nodes causes d_1 to be an exception node, while mapping d_1, d_2 to D as normal nodes causes c_1, c_2 to be exception nodes.

(a) A DG fragment (b) A document fragment

Fig. 8. A document that contains exception nodes

When c_1, c_2 are mapped to C as normal nodes, d_1 will be mapped to an exception node, and d_2 can still be mapped to a normal node. However, notice that storing d_2 in the same row as its parent b_1 contradicts Mapping Rule 2, because d_1 is the first element in the D collection. Therefore, both d_1 and d_2 are stored in separate rows in such a case. In general, when the first element in a collection is mapped to an exception node, all the elements in the collection are identified as exception nodes and are stored in separate rows.

In theory, mapping ordered siblings in a document to ordered DG nodes can be a combinatorial optimization problem: find a mapping that minimizes the number of rows that represent siblings. Our current mapper implements a naive algorithm: siblings are traversed in document order. Earlier visited nodes are mapped to normal nodes if possible. In the above example, d_1 is visited first, and is mapped to D as a normal node. Thereafter, c_1, c_2 are mapped to exception nodes.