(will be inserted by the editor)

Storing and Querying XML Data Using Denormalized Relational Databases

Andrey Balmin*, Yannis Papakonstantinou**

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
e-mail: {abalmin,yannis}@cs.ucsd.edu

Abstract XML database systems emerge as a result of the acceptance of the XML data model. Recent works have followed the promising approach of building XML database management systems on underlying RDBMS's. Achieving query processing performance reduces to two questions: (i) How should the XML data be decomposed into data that are stored in the RDBMS? (ii) How should the XML query be translated into an efficient plan that sends one or more SQL queries to the underlying RDBMS and combines the data into the XML result? We provide a formal framework for XML Schema-driven decompositions, which encompasses the decompositions proposed in prior work and extends them with decompositions that employ denormalized tables and binary-coded XML fragments. We provide corresponding query processing algorithms that translate the XML query conditions into conditions on the relational tables and assemble the decomposed data into the XML query result. Our key performance focus is the response time for delivering the first results of a query. The most effective of the described decompositions have been implemented in XCacheDB, an XML DBMS built on top of a commercial RDBMS, which serves as our experimental basis. We present experiments and analysis that point to a class of decompositions, called inlined decompositions, that improve query performance for full results and first results, without significant increase in the size of the database.

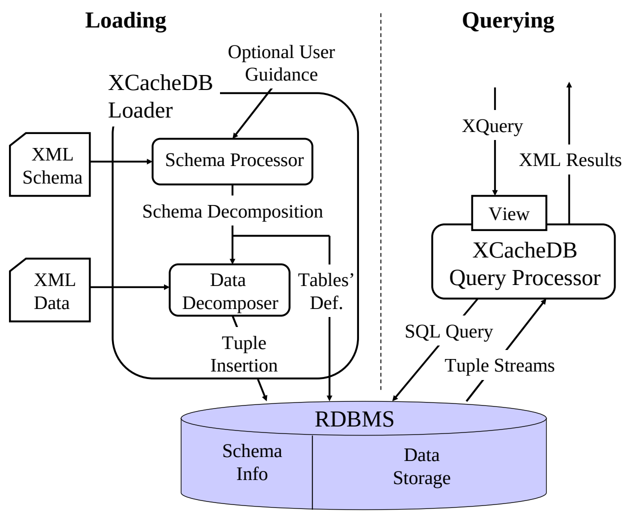


Fig. 1 The XML database architecture

1 Introduction

The acceptance and expansion of the XML model creates a need for XML database systems [STZ+99, BFRS02, DFS99, MFK+00, SKWW01, Rys01, JMC00, BKKM00, NDM+01, GMW99, SW00, eXc, X-H] . One approach towards building XML DBMS's is based on leveraging an underlying RDBMS for storing and querying the XML data. This approach allows the XML database to take advantage of mature relational technology, which provides reliability, scalability, high performance indices, concurrency control and other advanced functionality.

We provide a formal framework for XML Schema-driven decompositions of the XML data into relational data. The described framework encompasses the decompositions described in prior work on XML Schema-driven decompositions [STZ+99, BFRS02]
and extends prior work with a wide range of de-

* Andrey Balmin has been supported by NSF IRI-9734548.
** The authors built the XCacheDB system while on leave at Enosys Software, Inc., during 2000.

compositions that employ denormalized tables and binary-coded non-atomic XML fragments.

The most effective among the set of the described decompositions have been implemented in the presented *XCacheDB*, an XML DBMS built on top of a commercial RDBMS [BPSV00]. XCacheDB follows the typical architecture (see Figure 1) of an XML database built on top of a RDBMS [MFK+00, SKWW01, STZ+99, BFRS02, DFS99]. First, XML data, accompanied by their XML Schema [W3C01a], is loaded into the database using the XCacheDB *loader*, which consists of two modules: the *schema processor* and the *data decomposer*. The schema processor inputs the XML Schema and creates in the underlying relational database tables required to store any document conforming to the given XML schema. The conversion of the XML schema into relational may use optional user guidance. The mapping from the XML schema to the relational is called *schema decomposition*.[1] The data decomposer converts XML documents conforming to the XML schema into tuples that are inserted into the relational database.

XML data loaded into the relational database are queried by the XCacheDB *query processor*. The processor exports an XML view identical to the imported XML data. A client issues an XML query against the view. The processor translates the query into one or more SQL queries and combines the result tuples into the XML result. Notice that the underlying relational database is transparent to the query client.

The key challenges in XML databases built on relational systems are

1. how to decompose the XML data into relational data,
2. how to translate the XML query into a plan that sends one or more SQL queries to the underlying RDBMS and constructs an XML result from the relational tuple streams.

A number of decomposition schemes have been proposed [STZ+99, BFRS02, FK99, DFS99]. However all prior works have adhered to decomposing into normalized relational schemas. Normalized decompositions convert an XML document into a typically large number of tuples of different relations. Performance is hurt when an XML query that asks for some parts of the original XML document results into an SQL query (or SQL queries) that has to perform a large number of joins to retrieve and reconstruct all the necessary information.

[1] XCacheDB stores it in the relational database as well.

We provide a formal framework that describes a wide space of XML Schema-driven denormalized decompositions and we explore this space to optimize query performance. Note that denormalized decompositions may involve a set of relational design anomalies; namely, non-atomic values, functional dependencies and multivalued dependencies. Such anomalies introduce redundancy and impede the correct maintenance of the database [GMUW99]. However, given that the decomposition is transparent to the user, the introduced anomalies are irrelevant from a maintenance point of view. Moreover, the XML databases today are mostly used in web-based query systems where datasets are updated relatively infrequently and the query performance is crucial. Thus, in our analysis of the schema decompositions we focus primarily on their repercussions on query performance and secondarily on storage space and update speed.

The XCacheDB employs the most effective of the described decompositions. It employs two techniques that trade space for query performance by denormalizing the relational data.

– *non-Normal Form (non-NF) tables* eliminate many joins, along with the particularly expensive join start-up time.
– *BLOBs* are used to store pre-parsed XML fragments, hence facilitating the construction of XML results. BLOBs eliminate the joins and "order by" clauses that are needed for the efficient grouping of the flat relational data into nested XML structures, as it was previously shown in [SSB+00].

Overall, both of the techniques have a positive impact on *total query execution time* in most cases. The results are most impressive when we measure the *response time*, i.e. the time it takes to output the first few fragments of the result. Response time is important for web-based query systems where users tend to, first, issue under-constrained queries, for purposes of information discovery. They want to quickly, retrieve the first results and then issue a more precise query. At the same time, web interfaces do not need more than the first few results since the limited monitor space does not allow the display of too much data. Hence it is most important to produce the first few results quickly.

Our main contributions are:

– We provide a framework that organizes and formalizes a wide spectrum of decompositions of the XML data into relational databases.
– We classify the schema decompositions based on the dependencies in the produced relational

schemas. We identify a class of mappings called *inlined decompositions* that allow us to considerably improve query performance by reducing the number of joins in a query, without a significant increase in the size of the database.
– We describe data decomposition, conversion of an XML query into an SQL query to the underlying RDBMS, and composition of the relational result into the XML result.
– We have built in the XCacheDB system the most effective of the possible decompositions.
– Our experiments demonstrate that under typical conditions certain denormalized decompositions provide significant improvements in query performance and especially in query response time. In some cases, we observed up to 400% improvement in total time (Figure 23, Q1 with selectivity 0.1%) and 2-100 times in response time (Figure 23, Q1 with selectivity above 10%).

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3, we present definitions and framework. Section 4 presents the decompositions of XML Schemas into sets of relations. In Section 5, we present algorithms for translating the XML queries into SQL, and assembling the XML results. In Section 6, we discuss the architecture of XCacheDB along with interesting implementation aspects. In Section 7, we present the experiment results. We conclude and discuss directions for future work in Section 8.

2 Related Work

The use of relational databases for storing and querying XML has been advocated before by [BFRS02, STZ+99, FK99, MFK+00, DFS99, SKWW01]. Some of these works [FK99, MFK+00, DFS99] did not assume knowledge of an XML schema. In particular, the Agora project employed a fixed relational schema, which stores a tuple per XML element. This approach is flexible but it is less competitive than the other approaches, because of the performance problems caused by the large number of joins in the resulting SQL queries. The STORED system [DFS99] also employed a schema-less approach. However, STORED used data mining techniques to discover patterns in data and automatically generate XML-to-Relational mappings.

The works of [STZ+99] and [BFRS02] considered using DTD's and XML Schemas to guide mapping of XML documents into relations. [STZ+99] considered a number of decompositions leading to normalized tables. The "hybrid" approach, which provides the best performance, is identical to our "minimal 4NF decomposition". The other approaches of [STZ+99] can also be modeled by our framework. In one respect our model is more restrictive, as we only consider DAG schemas while [STZ+99] also takes into account cyclic schemas. It is possible to extend our approach to arbitrary schema graphs by utilizing their techniques. [BFRS02] studies horizontal and vertical partitioning of the minimal 4NF schemas. Their results are directly applicable in our case. However we chose not to experiment with those decompositions, since their effect, besides being already studied, tends to be less dramatic than the effect of producing denormalized relations. Note also that [BFRS02] uses a cost-based optimizer to find an optimal mapping for a given query mix. The query mix approach can benefit our work as well.

To the best of our knowledge, this is the first work to use denormalized decompositions to enhance query performance.

There are also other related works in the intersection of relational databases and XML. The construction of XML results from relational data was studied by [SSB+00, FTS00, FMS01]. [SSB+00] considered a variety of techniques for grouping and tagging results of the relational queries to produce the XML documents. It is interesting to note the comparison between the "sorted outer union" approach and BLOBs, which significantly improve query performance. The SilkRoute [FTS00, FMS01] considered using multiple SQL queries to answer a single XML Query and specified the optimal approach for various situations, which are applicable in our case as well.

Oracle 8i/9i, IBM DB2, and Microsoft SQL Server provide some basic XML support [BKKM00, Rys01, JMC00]. None of these products support XQuery or any other full-featured XML query language.

Another approach towards storying and querying XML is based on native XML and OODB technologies [SW00, NDM+01, GMW99]. The BLOBs resemble the common object-oriented technique of clustering together objects that are likely to be queried and retrieved jointly [BDK92]. Also, the non-normal form relations that we use are similar to path indices, such as the "access support relations" proposed by Kemper and Moerkotte [KM92]. An important difference is that we store data together with an index, similarly to Oracle's "index organized tables" [BKKM00].

A number of commercial XML databases are avaliable. Some of these systems [Kru, Ell, M/G] only support API data access and are effectively persistent implementations of the Document Object Model [W3C98a]. However, most of the sys-

tems [Coh, eXc, XML, Inf, Ipe, Wir, Neo, SW00, Ope, X-H, Apa, XYZ]
implement the XPath query language or its variations. Some vendors [SW00, eXc, Neo] have announced XQuery [W3C01b] support in the upcoming versions, however only X-Hive 3.0 XQuery processor [X-H] and Ipedo XML Database [Ipe] were publically available at the time of writing.

The majority of the above systems use native XML storage, but some [eXc, X-H, Wir] are implemented on top of object-oriented databases. Besides the query processing some of the commercial XML databases support full text searches [Ipe, X-H, XYZ], transactional updates [Coh, eXc, XML, Ipe, Wir, Neo] and document versioning [Ipe, Wir].

Even though XPath does not support heterogeneous joins, some systems [SW00, Ope] recognize their importance for the data integration applications and provide facilities that enable this feature.

Our work concentrates on selection and join queries. Another important class of XML queries involve path expressions. A number of schemes [LM01, HSKA$^+$] have been proposed recently that employ various node numbering techniques to facilitate evaluation of path expressions. For instance, [LM01] proposes to use pairs of numbers (start position and sub-tree size) to identify nodes. The XSearch system [XP] employs Dewey encoding of node IDs to quickly test for ancestor-descendant relationships. These techniques can be applied in the context of XCacheDB, since the only restriction that we place on node IDs is their uniqueness.

3 Framework

We use the conventional labeled tree notation to represent XML data. The nodes of the tree correspond to XML elements, and are labeled with the tags of the corresponding elements. Tags that start with the "@" symbol stand for attributes. Leaf nodes may also be labeled with values that correspond to the string content.

Note that we treat XML as a database model that allows for rich structures that contain nesting, irregularities, and structural variance across the objects. We assume the presence of XML Schema, and expect the data to be accessed via an XML query language such as XQuery. We have excluded many document oriented features of XML such as mixed content, comments and processing instructions.

Every node has a unique id invented by the system. The id's play an important role in the conversion of the tree to relational data, as well as in the reconstruction of the XML fragments from the relational query results.

Definition 1 (**XML document**) An XML document is a tree where

1. Every node has a label l coming from the set of element tags L
2. Every node has a unique id
3. Every atomic node has an additional label v coming from the set of values V. Atomic nodes can only be leafs of the document tree. [2]

$\diamondsuit$

Figure 2 shows an example of an XML document tree. We will use this tree as our running example. We consider only unordered trees. We can extend our approach to ordered trees because the node id's are assigned by a depth first traversal of the XML documents, and can be used to order sibling nodes.

3.1 XML Schema

We use *schema graphs* to abstract the syntax of XML Schema Definitions [W3C01a]. The following example illustrates the connection between XML Schemas and schema graphs.

Example 1 Consider the XML Schema of Figure 3 and the corresponding schema graph of Figure 4. They both correspond to the TPC-H [Cou99] data of Figure 2. The schema indicates that the XML data set has a root element named *Customers*, which contains one or more *Customer* elements. Each *Customer* contains (in some order) all of the atomic elements *Name*, *Address*, and *MarketSegment*, as well as zero or more complex elements *Order* and *PreferedSupplier*. These complex elements in turn contain other sets of elements.

Notice that XML schemas and schema graphs are in some respect more powerful than DTDs [W3C98b]. For example, in the schema graph of Figure 4 both *Customer* and *Supplier* have *Address* subelements, but the customer's address is simply a string, while the supplier's address consists of *Street* and *City* elements. DTD's cannot contain elements with the same name, but different content types.

Definition 2 (**Schema Graph**) A schema is a directed graph G where:

[2] However, not every leaf has to be an atomic node. Leafs can also be empty elements.

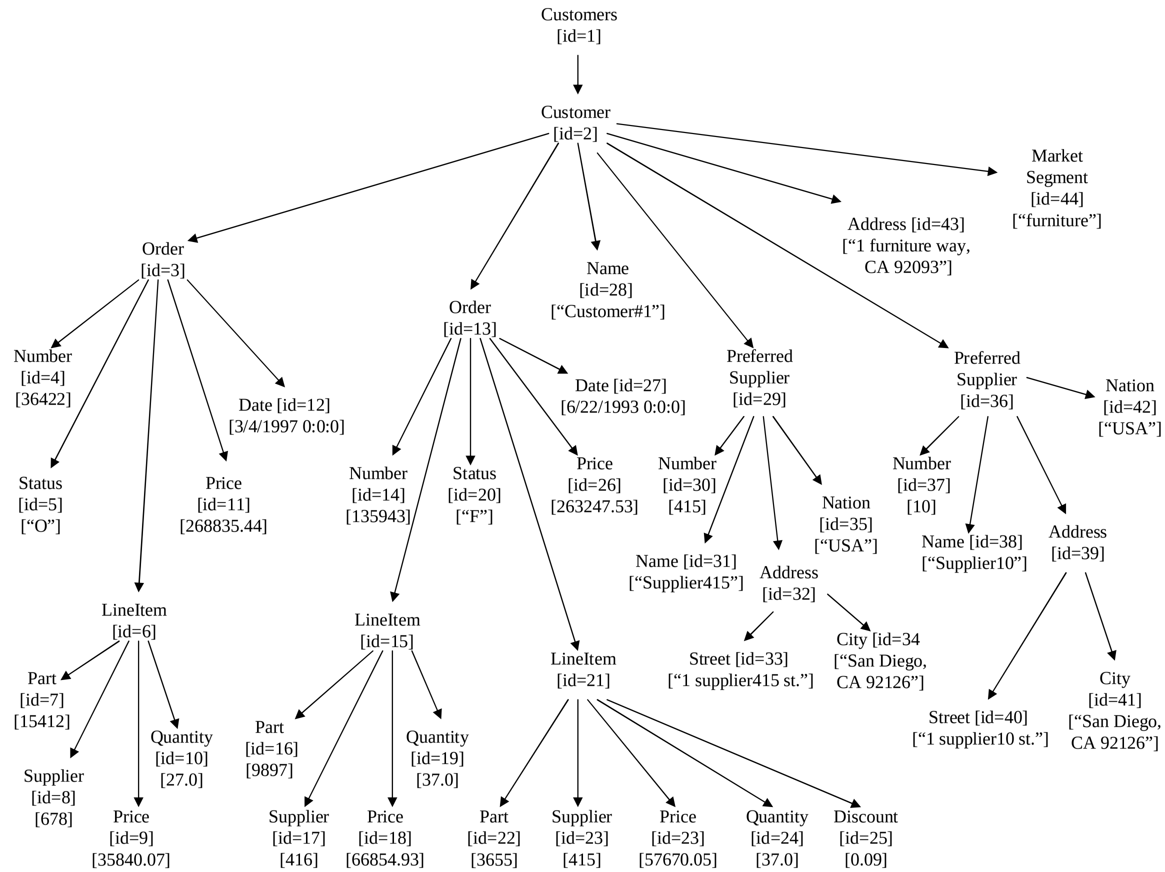


Fig. 2 A sample TPCH-like XML data set. Id's and data values appear in brackets.

1. Every node has a label l that is one of "all", or "choice", or is coming from the set of element tags L. Nodes labeled "all" and "choice" have at least two children.
2. Every leaf node has a label t coming from the set of types T.
3. Every edge is annotated with "minOccurs" and "maxOccurs" labels, which can be a non-negative integer or "unbounded".
4. A single node r is identified as the "root". Every node of G is reachable from r.

$\diamondsuit$

Schema graph nodes labeled with element tags are called *tag nodes*; the rest of the nodes are called *link nodes*.

Since we use an unordered data model, we do not include "sequence" nodes in the schema graphs. Their treatment is identical to that of "all" nodes. We also modify the usual definition of a *valid* document to account for the unordered model. To do that, we, first, define the *content type* of a schema node, which defines bags of sibling XML elements that are valid with respect to the schema node.

Definition 3 (**Content Type**) Every node g of a schema graph G is assigned a *content type* $\mathcal{T}(g)$, which is set of bags of schema nodes, defined by the following recursive rules.

– If g is a tag node, $\mathcal{T}(g) = \{\{g\}\}$
– If g is a "choice" node $g = choice(g_1, \ldots, g_n)$, with min/maxOccur labels of the $g \rightarrow g_i$ edge denoted min_i and max_i, then $\mathcal{T}(g) = \bigcup_{i=1}^{n} \mathcal{T}_{min_i}^{max_i}(g_i)$, where $\mathcal{T}_{min_i}^{max_i}(g_i)$ is a union of all bags obtained by concatenation of k, not necessarily distinct, bags from $\mathcal{T}(g_i)$, where $min_i \leq k \leq max_i$, or $min_i \leq k$ if $max_i =$ "unbounded". If $min_i = 0$, $\mathcal{T}_{min_i}^{max_i}(g_i)$ also includes an empty bag.
– If g is an "all" node $g = all(g_1, \ldots, g_n)$, then $\mathcal{T}(g)$ is a union of all bags obtained by concatenation of n bags – one from each $\mathcal{T}_{min_i}^{max_i}(g_i)$.

$\diamondsuit$

Definition 4 (**Document Tree Valid wrt Schema Graph**) We say that a document tree T is valid with respect to schema graph G, if there exist a total mapping $\mathcal{M}$ of nodes of T to the tag nodes of G, such that $root(T)$ maps to $root(G)$, and for every pair $(t, g) \in \mathcal{M}$, the following holds:

1. $label(t) = label(g)$
2. A bag of schema nodes to which the children of t map is a member of $\mathcal{T}_{min}^{max}(g_c)$, where g_c is the

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema">
  <xsd:element name = "customers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "customer" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "customer">
    <xsd:complexType>
      <xsd:all>
        <xsd:element ref = "number"/>
        <xsd:element ref = "name"/>
        <xsd:element ref = "address"/>
        <xsd:element ref = "market"/>
        <xsd:element ref = "orders" minOccurs = "0" maxOccurs = "unbounded"/>
        <xsd:element ref = "preferred_supplier" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "number" type = "xsd:integer"/>
  <xsd:element name = "name" type = "xsd:string"/>
  <xsd:element name = "address" type = "xsd:string"/>
  <xsd:element name = "market" type = "xsd:string"
  <xsd:element name = "orders">
    <xsd:complexType>
      <xsd:all>
        <xsd:element ref = "number"/>
        <xsd:element ref = "status"/>
        <xsd:element ref = "price"/>
        <xsd:element ref = "date"/>
        <xsd:element ref = "lineitem" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "preferred_supplier">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "number"/>
        <xsd:element ref = "name"/>
        <xsd:element ref = "address"/>
        <xsd:element ref = "nation"/>
        <xsd:element ref = "balance"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "status" type = "xsd:string"/>
  <xsd:element name = "price" type = "xsd:float"/>
  <xsd:element name = "date " type = "xsd:string"/>
  <xsd:element name = "lineitem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "part"/>
        <xsd:element ref = "supplier"/>
        <xsd:element ref = "quantity"/>
        <xsd:element ref = "price"/>
        <xsd:element ref = "disc ount" minOccurs = "0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "part" type = "xsd:integer"/>
  <xsd:element name = "supplier" type = "xsd:integer"/>
  <xsd:element name = "quantity" type = "xsd:float"/>
  <xsd:element name = "discount" type = "xsd:float"/>
  <xsd:element name = "nation" type = "xsd:string"/>
  <xsd:element name = "balance" type = "xsd:float"/>
</xsd:schema>
```

Fig. 3 The TPCH XML Schema

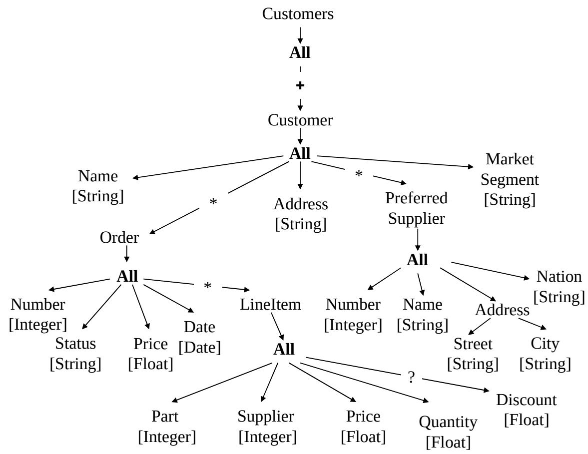


Fig. 4 Schema Graph notation

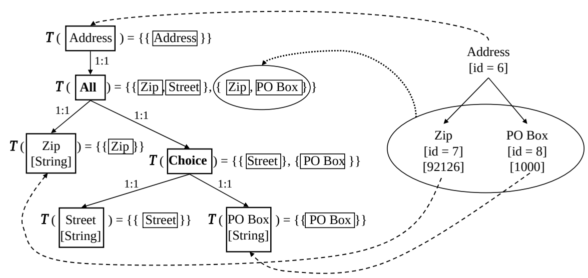


Fig. 5 Content Types and Document Tree Validation

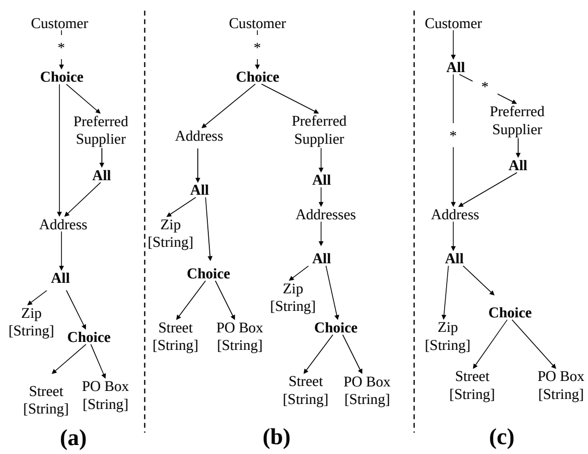


Fig. 6 Schema graphs (a) and (b) are equivalent. Graph (c) is normalization of graph (a).

child of g, and min and max are min/maxOccur labels of the $g \rightarrow g_c$ edge.

$\diamondsuit$

Figure 5 illustrates how the content types are assigned and used in the document validation. The *Address* element on the right is valid with respect to the schema graph on the left. Each schema node is annotated with its content type. For example, the type of the "choice" node is $\{\{Street\},\{PO\ Box\}\}$. The document validation is done by mapping XML tree nodes to the tag nodes of the schema graph (mappings are shown by the dashed lines), in such a way that the bag of types corresponding to the children of every XML node is a member of the content type of the child of the corresponding schema node. For example, the children of the *Address* element belong to the content type of the "all" node.

Normalized Schema Graphs To simplify the presentation we only consider *normalized* schema graphs, where all incoming edges of *link* nodes have $maxOccurs = 1$. Any schema graph can be converted into, a possibly less restrictive, normalized schema graph by a top-down breadth-first traversal of the schema graph that applies the following rules. For every link node N that has an incoming edge with $minOccurs = inMin$ and $maxOccurs = inMax$, where $inMax > 1$, the $maxOccurs$ is set to 1 and the $maxOccurs$ of every outgoing edge of N is multiplied by $inMax$. The result of the product is "unbounded" if at least one parameter is "unbounded". Similarly, if $inMin > 1$, the $minOccurs$ is set to 1 and the $minOccurs$ of every outgoing edge of N is multiplied by $inMin$. Also, if N is a "choice", it gets replaced with an "all" node with the same set of children, and for every outgoing edge the $minOccur$ is set to 0. For example, the schema graph of Figure 6(a) will be normalized into the graph of Figure 6(c). Notice that the topmost "choice" node is replaced by "all", since a customer may contain multiple addresses and preferred supplier records.

Without loss of generality to the decomposition algorithms described next, we only consider schemas where $minOccurs \in \{0, 1\}$ and $manOccurs$ is either 1 or *unbounded*. We use the following symbols: "1", "*", "?", "+", to encode the "minOccurs"/"maxOccurs" pairs. For brevity, we omit "1" annotations in the figures. We also omit "all" nodes if their incoming edges are labeled "1", whenever this doesn't cause an ambiguity.

We only consider acyclic schema graphs. Schema graph nodes that are pointed by a "*" or a "+" will be called *repeatable*.

4 XML Decompositions

We describe next the steps of decomposing an XML document into a relational database. First, we produce a schema decomposition, i.e., we use the schema graph to create a relational schema. Second, we decompose the XML data and load it into the corresponding tables. We use the schema decomposition to guide the data load.

The generation of an equivalent relational schema proceeds in two steps. First, we decompose the schema graph into fragments. Second, we generate a relational table definition for each fragment.

Definition 5 (**Schema Decomposition**) A schema decomposition of a schema graph G is a set of fragments $F_1, \ldots, F_n$, where each fragment is a subset of nodes of G that form a connected DAG. Every tag node of G has to be member of at least one fragment. $\diamondsuit$

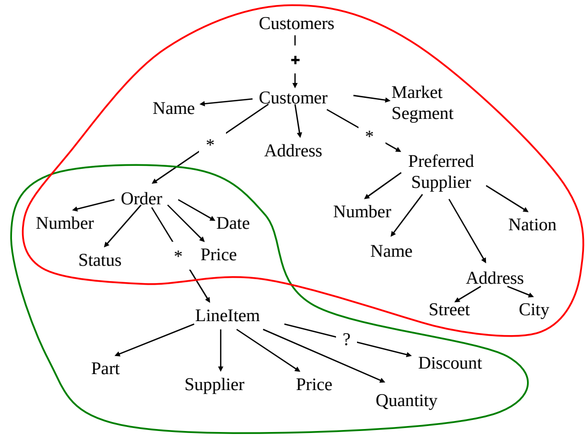


Fig. 7 An XML Schema decomposition

Due to acyclicity of the schema graphs, each fragment has at least one *fragment root* node, i.e., a node that does not have incoming edges from any other node of the fragment. Similarly, *fragment leaf* nodes are the nodes that do not have outgoing edges that lead to other nodes of the fragment. Note that a schema decomposition is not necessarily a partition of the schema graph – a node may be included in multiple fragments (Figure 7).

Some fragments may contain only "choice" and "all" nodes. We call these fragments *trivial*, since they correspond to empty data fragments. We only consider decompositions which contain connected, *non-trivial* fragments, where all fragment leafs are tag nodes.

DAG schemas offer an extra degree of freedom, since an equivalent schema can be obtained by "splitting" some of the nodes that have more than one ancestor. For example, the schema of Figure 6(b), can be obtained from the schema of Figure 6(a) by splitting at element *Address*. Such a split corresponds to a *derived horizontal partitioning* of a relational schema [ÖV99].

Similarly, element nodes may also be eliminated by "combining" nodes. For example, an $all(a*, b, a*)$ may be reduced to $all(a*, b)$ if types of both a's are equal [3]. Since we consider an unordered data model, the queries cannot distinguish between "first" and "second" a's in the data. Thus, we do not need to differentiate between them. A similar DTD reduction process was used in [STZ$^+$99]. However, unlike [STZ$^+$99] our decompositions do not require reduction and offer flexibility needed to support the document order. Similar functionality is included in LegoDB [BFRS02].

[3] We say that types A and B are equal, if every element that is valid wrt A is also valid wrt B, and vice versa.

Definition 6 (**Path Set, Equivalent Schema Graphs**) A path set of a schema graph G, denoted $PS(G)$, is the set of all possible paths in G that originate at the root of G. Two schema graphs G_1 and G_2 are *equivalent* if $PS(G_1) = PS(G_2)$. ♢

We define the set of *generalized schema decompositions* of a graph G to be the set of schema decompositions of all graphs G' that are equivalent to G (including the schema decompositions of G itself.) Whenever it is obvious from the context we will say "set of schema decompositions" implying the set of generalized schema decompositions.

Definition 7 (**Root Fragments, Parent Fragments**) A *root fragment* is a fragment that contains the root of the schema graph. For each non-root fragment F we define its *Parent Fragments* in the following way: Let R be a root node of F, and let P be a parent of R in the schema graph. Any fragment that contains P is called a parent fragment of F. [4] ♢

Definition 8 (**Fragment Table**) A *Fragment Table* T corresponds to every fragment F. T has an attribute $A_{N_{ID}}$ of the special "ID" datatype[5] for every tag node N of the fragment. If N is an atomic node the schema tree T also has an attribute A_N of the same datatype as N. If F is not a root fragment, T also includes a parent reference column, of type ID, for each distinct path that leads to a root of F from a repeatable ancestor A and does not include any intermediate repeatable ancestors. The parent reference columns store the value of the ID attribute of A. ♢

For example, consider the `Address` fragment table of Figure 8. Regardless of other fragments present in the decomposition, the `Address` table will have two parent reference columns. One column will refer to the *Customer* element and another to the *Supplier*. Since we consider only tree data, every tuple of the `Address` table will have exactly one non-null parent reference.

A fragment table is named after the left-most root of the corresponding fragment. Since multiple schema nodes can have the same name, name collisions are resolved by appending a unique integer.

We use null values in ID columns to represent missing optional elements. For example, the null value in the `POBox_id` of the first tuple of the `Address` table indicates that the *Address* element with id=2 does not have a *POBox* subelement. An empty XML element N is denoted by a non-null value in $A_{N_{ID}}$ and a null in A_N.

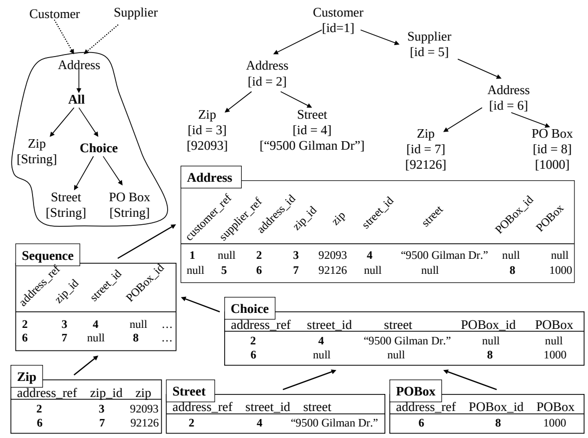


Fig. 8 Loading data into fragment tables

Data Load We use the following inductive definition of fragment tables' content. First, we define the data content of a fragment consisting of a single tag node N. The fragment table T_N, called *node table*, contains an ID attribute $A_{N_{ID}}$, a value attribute A_N, and one or more parent attributes. Let us consider a *Typed Document Tree* D', where each node of D is mapped to a node of the schema graph. A tuple is stored in T_N for each node $d \in D$, such that $(d \to N) \in D'$. Assume that d is a child of the node $p \in D$, such that $(p \to P) \in D'$. The table T_N will be populated with the following tuple: $\langle A_{P_{ID}} = p_{id}, A_{N_{ID}} = d_{id}, A_N = d \rangle$. If T_N contains parent attributes other than $A_{P_{ID}}$, they are set to null.

A table T corresponding to an internal node N is populated depending on the type of the node.

– If N is an "all", then T is the result of a join of all children tables on parent reference attributes.
– If N is a "choice", then T is the result of an outer union [6] of all children tables.
– If N is a tag node, which by definition has exactly one child node with a corresponding table T_C, then $T = T_N ⟕ T_C$

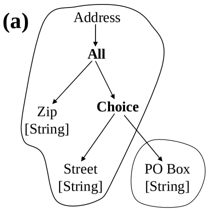
The following example illustrates the above definition. Notice that the XCacheDB Loader does not use the brute force implementation suggested in the example. We employ optimizations that eliminate the majority of the joins.

[4] Note that a decomposition can have multiple root fragments, and a fragment can have multiple parent fragments.

[5] In RDBMS's we use the "integer" type to represent the "ID" datatype.

[6] Outer union of two tables P and Q is a table T, with a set of attributes $attr(T) = attr(P) \cup attr(Q)$. The table T contains all tuples of P and Q extended with nulls in all the attributes that were not present in the original.

(a)



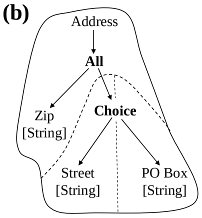
Address

Address_id	Zip_id	Zip	Street_id	Street
2	3	92093	4	"9500 Gilman Dr."
6	7	92126	null	null

POBox

Addresses_id	POBox_id	POBox
2	8	1000

(b)



Address1

Address_id	Zip_id	Zip	Street_id	Street
2	3	92093	4	"9500 Gilman Dr."

Address2

Address_id	Zip_id	Zip	POBox_id	POBox
6	7	92126	8	1000

Fig. 9 Alternative fragmentations of data of Figure 8

Example 2 Consider the schema graph fragment, and the corresponding data fragment of Figure 8. The `Address` fragment table is built from node tables `Zip`, `Street`, and `POBox`, according to the algorithm described above. A table corresponding to the "choice" node in the schema graph is built by taking an outer union of `Street` and `POBox`. The result is joined with `Zip` to obtain the table corresponding to the "all" node. The result of the join is, in turn, joined with the `Address` node table (not shown) which contains three attributes "customer_ref", "supplier_ref", and "address_id".

Alternatively, the "Address" fragment of Figure 8 can be split in two as shown in Figure 9(a) and (b). The dashed lines in Figure 9(b) indicates that a *horizontal partitioning* of the fragment should occur along the "choice" node. This line indicates that the fragment table should be split into two. Each table projects out attributes corresponding to one side of the "choice". The tuples of the original table are partitioned into the two tables based on the null values of the projected attributes. This operation is similar to the "union distribution" discussed in [BFRS02]. Horizontal partitioning improves the performance of queries that access either side of the union (e.g., either *Street* or *POBox* elements). However, performance may degrade for queries that access only *Zip* elements. Since we assume no knowledge of the query workload, we do not perform horizontal partitioning automatically, but leave it as an option to the system administrator.

The following example illustrates decomposing TPCH-like XML schema of Figure 4 and loading it with data of Figure 2.

Example 3 Consider the schema decomposition of Figure 10. The decomposition consists of three fragments rooted at the elements *Customers*, *Order*, and *Address*. Hence the corresponding relational schema has tables `Customers`, `Order`, and `Address`. The bottom part of Figure 10 illustrates the contents of each table for the dataset of Figure 2. Notice that the tables `Customers` and `Order` are not in BCNF.

For example, the table `Order` has the non-key functional dependency "$order\_id \rightarrow number\_id$", which introduces redundancy.

We use "(FK)" labels in Figure 10 to indicate parent references. Technically these references are not foreign keys since they do not necessarily refer to a primary key.

Alternatively one could have decomposed the example schema as shown in Figure 7. In this case there is a non-FD multi-valued dependency (MVD) in the `Customers` table, i.e., an MVD that is not implied by a functional dependency. Orders and preferred suppliers of every customer are independent of each other:

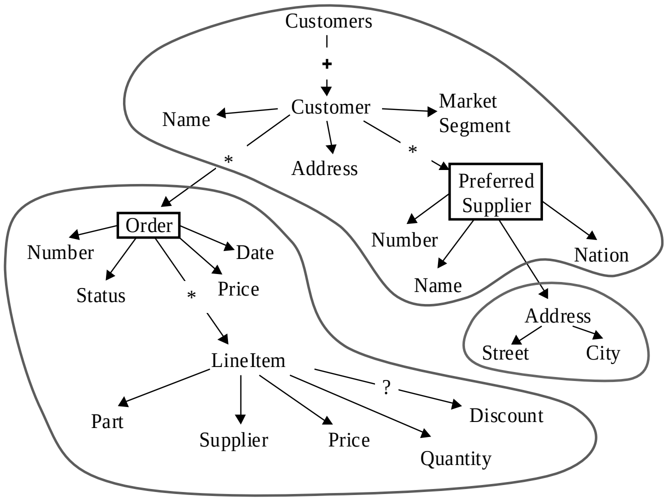
$$customers\_id, customer\_id, c\_name\_id, c\_address\_id, c\_marketSegment\_id, c\_name, c\_address, c\_marketSegment \twoheadrightarrow c\_preferredSupplier\_id, p\_name\_id, p\_number\_id, p\_nation\_id, p\_name, p\_number, p\_nation, p\_address\_id, a\_street\_id, a\_city\_id, a\_street, a\_city$$

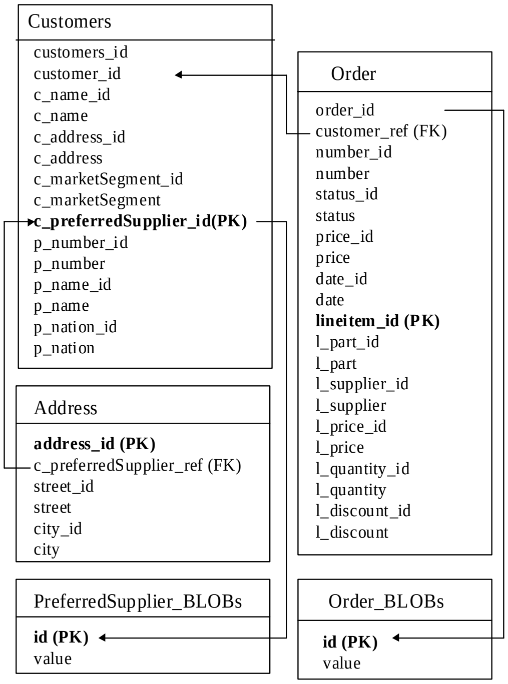
The decompositions that contain non-FD MVD's are called *MVD decompositions*.

Vertical Partitioning In the schema of Figure 10 the *Address* element is not repeatable, which means that there is at most one address per supplier. Using a separate `Address` table is an example of *vertical partitioning* because there is a one-to-one relationship between the `Address` table and its parent table `Customers`. The vertical partitioning of XML data was studied in [BFRS02], which suggests that partitioning can improve performance if the query workload is known in advance. Knowing the groups of attributes that get accessed together, the vertical partitioning can be used to reduce table width without incurring a big penalty from the extra joins. We do not consider vertical partitioning in this paper, but the results of [BFRS02] can be carried over to our approach. We use the term *minimal* to refer to decompositions without vertical partitioning.

Definition 9 (**Minimal Decompositions**) A decomposition is *minimal* if all edges connecting nodes of different fragments are labeled with "*" or "+". ♢

Figure 7 and Figure 11 show two different minimal decompositions of the same schema. We call





PreferedSupplier_BLOBs

id	value
29	PreferredSupplier(29)[Number(30)[415],...
36	PreferredSupplier(36)[Number(37)[10],...

Order_BLOBs

id	value
3	Order(3)[Number(4)[36422],Status...
13	Order(13)[Number(14)[135943],...

Customers

customers_id	customer_id	c_name_id	c_name	c_address_id	c_address	c_market Segment_id	c_market Segment	c_preferred Supplier_id	p_number_id	p_number	p_name_id	p_name	p_nation_id	p_nation
1	**2**	28	“Customer1”	43	“1 Furniture...”	44	“furniture”	**29**	30	415	31	“Supplier415”	35	“USA”
1	**2**	28	“Customer1”	43	“1 Furniture...”	44	“furniture”	**36**	37	10	38	“Supplier10”	42	“USA”

Order

order_id	customer_ref	number_id	number	status_id	status	price_id	price	date_id	date	lineitem_id	l_part_id	l_part	l_supplier_id	l_supplier	l_ price_id	l_ price	l_quantity_id	l_quantity	l_discount_id	l_discount
3	**2**	4	36422	5	“O”	11	268835.44	12	3/4/1997 0:0:0	**6**	7	15412	8	678	9	35840.07	10	27.0	Null	Null
13	**2**	14	135943	20	“F”	26	263224.53	27	6/22/1993 0:0:0	**15**	16	9897	17	416	18	66854.93	19	37.0	Null	Null
13	**2**	14	135943	20	“F”	26	263224.53	27	6/22/1993 0:0:0	**21**	22	3655	23	415	23	57670.05	24	37.0	25	0.09

Address

address_id	c_preferredSupplier_ref	street_id	street	city_id	city
32	**29**	33	“1 supplier10 St.”	34	“San Diego, CA 92126”
39	**36**	40	“1 supplier415 St.”	41	“San Diego, CA 92126”

Fig. 10 XML Schema and Data decomposition

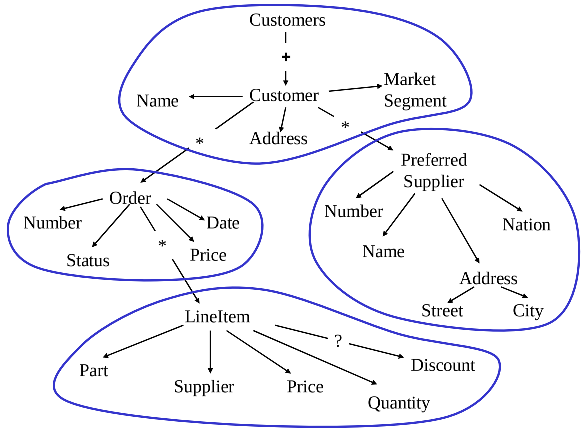


Fig. 11 Minimal 4NF XML Schema decomposition

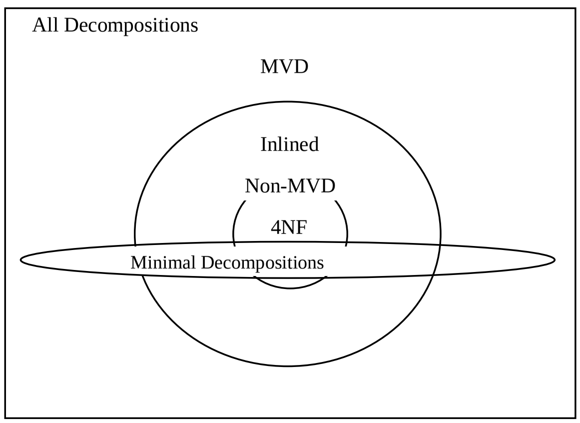


Fig. 12 Classification of Schema decompositions

the decomposition of Figure 11 a *4NF decomposition* because all its fragments are *4NF fragments* (i.e. the fragment tables are in 4NF). Note that a fragment is 4NF if and only if it does not include any "*" or "+" labeled edges, i.e. no two nodes of the fragment are connected by a "*" or "+" labeled edge. We assume that the only dependencies present are those derived by the decomposition.

Every XML Schema tree has exactly one minimal 4NF decomposition, which minimizes the space requirements. From here on, we only consider minimal decompositions.

Prior work [STZ+99, BFRS02] considers only 4NF decompositions. However we employ denormalized decompositions to improve query execution time as well as response time. Particularly important for performance purposes is the class of inlined decompositions described below. The inlined decompositions improve query performance by reducing the number of joins, and (unlike MVD decompositions) the space overhead that they introduce depends only on the schema and not on the dataset.

Definition 10 (**Non-MVD Decompositions and Inlined Decompositions**) A non-MVD fragment is one where all "*" and "+" labeled edges appear in a single path. A non-MVD decomposition is one that has only non-MVD fragments. An *inlined* fragment is a non-MVD fragment that is not a 4NF fragment. An inlined decomposition is a non-MVD decompositions that is not a 4NF decomposition. ◇

The non-MVD fragment tables may have functional dependencies (FD's) that violate the BCNF condition (and also the 3NF condition [GMUW99]), but they have no non-FD MVD's. For example, the *Customers* table of Figure 10 contains the FD

$$customer\_ID \rightarrow c\_name$$

that breaks the BCNF condition, since the key is "c_preferredSupplier_id". However, the table has no non-FD MVD's.

From the point of view of the relational data, an *inlined fragment* table is the join of fragment tables that correspond to a line of two or more 4NF fragments. For example, the fragment table *Customers* of Figure 10 is the join of the fragment tables that correspond to the 4NF fragments *Customers* and *PreferredSupplier* of Figure 11. The tables that correspond to inlined fragments are very useful because they reduce the number of joins while they keep the number of tuples in the fragment tables low.

Lemma 1 (Space Overhead as a Function of Schema Size) *Consider two non-MVD fragments F_1 and F_2 such that when unioned, they result in an inlined fragment F.* [7] *For every XML data set, the number of tuples of F is less than the total number of tuples of F_1 and F_2.*

Proof Let's consider the following three cases. First, if the schema tree edge that connects F_1 and F_2 is labeled with "1" or "?", the tuples of F_2 will be inlined with F_1. Thus F will have the same number of tuples as F_1.
Second, if the edge is labeled with "+", F will have the same number of tuples as F_2, since F will be the result of the join of F_1 and F_2, and the schema implies that for every tuple in F_2, there is exactly one matching tuple, but no more in F_1.
Third, if the edge is labeled with "*", F will have fewer tuples than the total of F_1 and F_2, since F will be the result of the left outer join of F_1 and F_2. □

[7] A fragment consisting of two non-MVD fragments connected together, is not guaranteed to be non-MVD
.

We found that the inlined decompositions can provide significant query performance improvement. Noticeably, the storage space overhead of such decompositions is limited, even if the decomposition include all possible non-MVD fragments.

Definition 11 (**Complete Non-MVD Decompositions**) A complete non-MVD decomposition, *complete* for short, is one that contains all possible non-MVD fragments. ◇

The complete non-MVD decompositions are only intended for the illustrative purpose, and we are not advocating their practical use.

Note that a complete non-MVD decomposition includes all fragments of the 4NF decomposition. The other fragments of the complete decomposition consist of fragments of the 4NF decomposition connected together. In fact, a 4NF decomposition can be viewed as a tree of 4NF fragments, called *4NF fragment tree*. The fragments of a complete minimal non-MVD decomposition correspond to the set of paths in this tree. The space overhead of a complete decompositions is a function of the size of the 4NF fragment tree.

Lemma 2 (Space Overhead of a Complete Decomposition as a Function of Schema) *Consider a schema graph G, its complete decomposition $D_C(G) = \{F_1, \ldots, F_k\}$, and a 4NF decomposition $D_{4NF}(G)$. For every XML data set, the number of tuples of the complete decomposition is*

$$|D_C(G)| = \sum_{i=1}^{k} |F_i| < |D_{4NF}(G)| * h * n$$

where h is the height of the 4NF fragment tree of G, and n is the number of fragments in $D_{4NF}(G)$.

Proof Consider a *record tree* R constructed from an XML document tree T in the following fashion. A node of the record tree is created for every tuple of the 4NF data decomposition $D_{4NF}(T)$. Edges of the record tree denote child-parent relationships between tuples. There is a one to one mapping from paths in the record tree to paths in its 4NF fragment tree, and the height of the record tree h equals to the height of the 4NF fragment tree. Since any fragment of $D_C(G)$ maps to a path in the 4NF fragment tree, every tuple of $D_C(T)$ maps to a path in the record tree. The number of path's in the record tree $P(R)$ can be computed by the following recursive expression: $P(R) = N(R) + P(R_1) + \ldots + P(R_n)$, where $N(R)$ is the number of nodes in the record tree and stands for all the paths that start at the root. R_i's denote subtrees rooted at the children of the root. The maximum depth of the recursion is h. At each level of the recursion, after the first one, the total number of added paths is less than N. Thus $P(R) < hN$.

Multiple tuples of $D_C(T)$ may map to the same path in the record tree, because each tuple of $D_C(T)$ is a result of some outerjoin of tuples of $D_{4NF}(T)$, and the same tuple may be a result of multiple outer joins (e.g. $A ⟕ B = A ⟕ B ⟕ C$, if C is empty.) However the same tuple cannot be a result of more than n distinct left outerjoins. Thus $|D_C(G)| \leq P(R) * n$. By definition $|D_{4NF}(G)| = N$; hence $|D_C(G)| < |D_{4NF}(G)| * h * n$. □

4.1 BLOBs

To speed up construction of the XML results from the relational result-sets XCacheDB stores a binary image of pre-parsed XML subtrees as Binary Large OBjects (BLOBs). The binary format is optimized for efficient navigation and printing of the XML fragments. The fragments are stored in special BLOBs tables that use node IDs as foreign keys to associate the XML fragments to the appropriate data elements.

By default, every subtree of the document except the trivial ones (the entire document and separate leaf elements) is stored in the Blobs table. This approach may have unnecessarily high space overhead because the data gets replicated up to $H - 2$ times, where H is the depth of the schema tree. We reduce the overhead by providing a graphical utility, the *XCacheDB Loader*, which allows the user to control which schema nodes get "BLOB-ed", by annotating the XML Schema. The user should BLOB only those elements that are likely to be returned by the queries. For example, in the decomposition of Figure 10 only `Order` and `PreferredSupplier` elements were chosen to be BLOB-ed, as indicated by the boxes. `Customer` elements may be too large and too infrequently requested by a query, while `LineItem` is small and can be constructed quickly and efficiently without BLOB's.

We chose not to store Blobs in the same tables as data to avoid unnecessary increase in table size, since Blob structures can be fairly large. In fact, a Blob has similar size to the XML subtree that it encodes. The size of an XML document (without the header and whitespace) can be computed as

$$XML_{Size} = E_N * (2E_{Size} + 5) + T_N * T_{Size}$$

where E_N is the number of elements, E_{Size} is the average size of the element tag, T_N is how many elements contain text (i.e. leafs) and T_{Size} is the average text size. The size of a BLOB is:

$$BLOB_{Size} = E_N * (E_{Size} + 10) + T_N * (T_{Size} + 3)$$

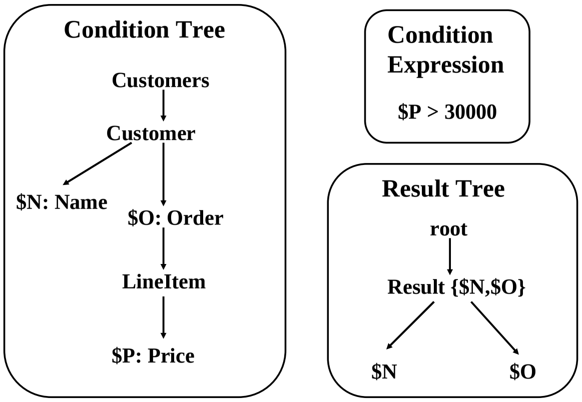


Fig. 13 XML Query notation

The separate Blobs table also gives us an option of using a separate SQL query to retrieve Blobs which improves the query response time.

5 XML Query Processing

We represent XML queries with a tree notation similar to *loto-ql* [PV00]. The query notation facilitates explanation of query processing and corresponds to FOR-WHERE-RETURN queries of the XQuery standard [W3C01b].

Definition 12 (**Query**) A query is a tuple $\langle C, E, R\rangle$, where C is called *condition tree*, E is called *condition expression*, and R is called *result tree*.

C is a labeled tree that consists of:
  - Element nodes that are labeled with labels from L. Each element node n may also be labeled with a variable $Var(n)$.
  - Union nodes. The same set of variables must occur in all children subtrees of a Union node. Two nodes cannot be labeled with the same variable, unless their lowest common ancestor is a Union node.

E is a logical expression involving logical predicates, logical connectives, constants, and variables that occur in C.

R is a tree where internal nodes are labeled with constants and leaf nodes are labeled either with variables that occur in C or with constants. Some nodes may also have "group-by" labels consisting of one or more variables that occur in C. If a variable V labels a leaf $l \in R$ then V is in the group-by label of l or the group-by label of an ancestor of l.

$\diamondsuit$

The query semantics are based on first matching the condition tree with the XML data to obtain bindings and then using the result tree to structure the bindings into the XML result.

The semantics of the condition tree are defined in two steps. First, we remove Union nodes and produce a forest of *conjunctive condition trees*, by traversing the condition tree bottom-up and replacing each Union node non-deterministically by one of its children. This process is similar to producing a disjunctive normal form of a logical expression. Set of bindings produced by the condition tree is defined as a union of sets of bindings produced by each of the conjunctive condition trees.

Formally, let C be a condition tree of a query and t be the XML document tree. conjunctive Let $Var(C)$ be the set of variables in C. Let $C_1 ... C_l$ be a set of all conjunctive condition trees of C. Note that $Var(C) = Var(C_i), \forall i \in [1, l]$. A *variable binding* $\hat{\beta}$ maps each variable of $Var(C)$ to a node of t. The set of variable bindings is computed based on the set of condition tree bindings. A condition tree binding β maps each node n of some conjunctive condition tree C_i to a node of t. The condition tree binding is valid if $\beta(root(C_i)) = root(t)$ and recursively, traversing C depth-first left-to-right, for each child c_j of a node $c \in C_i$, assuming c is mapped to $x \in t$, there exists a child x_j of x such that $\beta(c_j)) = x_j$ and $label(c_j) = label(x_j)$.

The set of variable bindings consists of all bindings $\hat{\beta} = [V_1 \mapsto x_1, \ldots, V_n \mapsto x_n]$ such that there is a condition tree binding $\beta = [c_1 \mapsto x_1, \ldots, c_n \mapsto x_n, \ldots]$, such that $V_1 = Var(c_1), \ldots, V_n = Var(c_n)$.

The condition expression E is evaluated using the binding values and if it evaluates to true, the variable binding is qualified. Notice that the variables bind to XML elements and not to their content values. In order to evaluate the condition expression, all variables are coerced to the content values of the elements to which they bind. For example, in Figure 13 the variable P binds to an XML element "price". However, when evaluating the condition expression we use the integer value of "price".

Once a set of qualified bindings is identified, the resulting XML document tree is constructed by structural recursion on the result tree R as follows. The recursion starts at the root of R with the full set of qualified bindings B. Traversing R top-down, for each sub-tree $R(n)$ rooted at node n, given a partial set of bindings B' (we explain how B' gets constructed next) we construct a forest $F(n, B')$ following one of the cases below:

Label: If n consists of a tag label L without a group-by label, the result is an XML tree with

```
</root>
     FOR $C IN document(''customers.xml")/
               Customers/Customer
         $N IN $c/Name
         $O IN $c/Order
     WHERE not empty(
         FOR $P IN $o/LineItem/Price
         WHERE $P > 30000
         RETURN $P
     )
     RETURN
         <result>
             {$N}
             {$O}
         </result>
</root>
```

Fig. 14 The XQuery equivalent to the query of Figure 13

root labeled L. The list of children of the root is the concatenation $F(n_1, B'')\#\ldots\#F(n_m, B'')$, where
$n_1, n_2, \ldots, n_m$ are the children of n. For each of the children, the partial set of bindings is $B'' = B'$.

Group-By: If n is of the form $L\{V_1, \ldots, V_k\}$, where $V_1, \ldots, V_k$ are group-by variables, $F(n, B')$ contains an XML tree $T_{v_1,\ldots,v_k}$ for each distinct set $v_1, \ldots, v_k$ of values of $V_1, \ldots, V_k$ in B'. Each $T_{v_1,\ldots,v_k}$ has its root labeled L. The list of children of the root is the concatenation $F(n_1, B'_1)\#\ldots\#F(n_m, B'_m)$, where $n_1, n_2, \ldots, n_m$ are the children of n. For $T_{v_1,\ldots,v_k}$ and n_i the partial set of bindings is
$B'_i = \Pi_{V(n_i)}(\sigma_{V_1=v_1 AND \ldots AND V_k=v_k} B')$, where $V(n_i)$ is the set of variables that occur in the tree rooted at n_i.

Leaf Group-By: If n is a leaf node of form
$V\{V_1, \ldots, V_k\}$, the result is a list of values of V, for each distinct set $v_1, \ldots, v_k$ of values of $V_1, \ldots, V_k$ in B'.

Leaf Variable: If n is a single variable V, and V binds to an element E in B', the result is E. If the query plan is valid, B' will contain only a single tuple.

The result of the query is the forest $F(r, B)$, where r is the root of the result tree and B is the set of bindings delivered by the condition tree and condition expression. However, since in our work we want to enforce that the result is a single XML tree, we require that r does not have a "group-by" label.

Example 4 The condition tree and expression of the query of Figure 13 retrieve tuples $\langle N, O\rangle$ where N is the *Name* element of a *Customer* element with an *Order* O that has at least one *LineItem* that has *Price* greater than 30000. For each tuple $\langle N, O\rangle$ a *Result* element is produced that contains the N and the O. This is essentially query number 18 of the TPC-H benchmark suite [Cou99], modified not to aggregate across lineitems of the order. It is equivalent to the XQuery of Figure 14.

For example, if the query is executed on data of Figure 2, the following set of bindings is produced, assuming that the *Order* elements are BLOB-ed.

$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_3, \$P/Price_9[35840.07]\rangle$
$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_{13}, \$P/Price_{18}[66854.93]\rangle$
$\langle \$N/Name_{29}[\text{“}Customer1\text{”}],$
$\$O/Order_{13}, \$P/Price_{24}[57670.05]\rangle$

Numbers in subscript indicate node ID's of the elements; square brackets denote values of atomic elements and subelements of complex elements. First, a single *root* element is created. Then, the group-by on the *Result* node partitions the bindings into two groups (for $Order_3$ and $Order_{13}$), and creates a *Result* element for each group. The second group-by creates two *Order* elements from the following two sets of bindings.

$\langle \$O/Order_3, \$P/Price_9[35840.07]\rangle$
and
$\langle \$O/Order_{13}, \$P/Price_{18}[66854.93]\rangle$
$\langle \$O/Order_{13}, \$P/Price_{24}[57670.05]\rangle$

The final result of the query is the following document tree:

```
root_100[
   Result_101[
      Name_29[''Customer1"],
      Order_3[...],
   Result_102[
      Name_29[''Customer1"],
      Order_13[...]
   ]
]
```

We can extend our query semantics to ordered XML model. To support order-preserving XML semantics, group-by operators will produce lists, given sorted lists of source bindings. In particular the group-by operator will order the output elements according to the node ID's of the bindings of the group-by variables. For example, the group-by in query of Figure 13 will produces lists of pairs of names and orders, sorted by name ID and order ID.

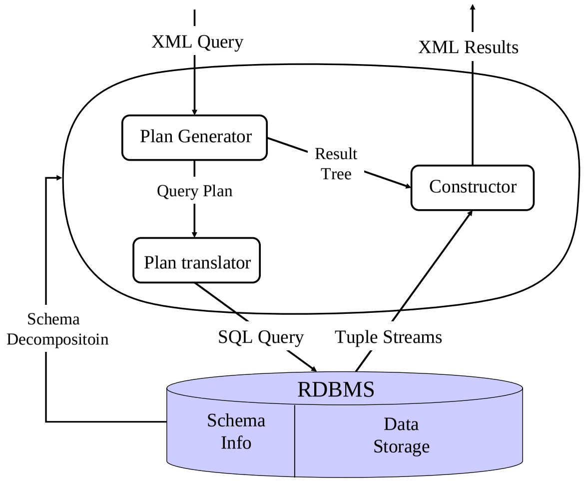


Fig. 15 Query Processing Architecture

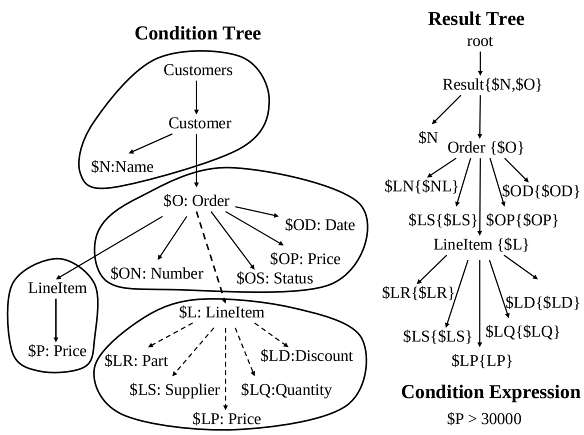


Fig. 16 Query Plan

5.1 Query Processing

Figure 15 illustrates the typical query processing steps followed by XML databases built on relational databases; the architecture of XCacheDB is indeed based on the one of Figure 15. The *plan generator* receives an XML query and a schema decomposition. It produces a plan, which consists of the *condition tree*, the *condition expression*, the *plan decomposition*, and the *result tree*. The *plan translator* turns the query plan into an SQL query. Plan result trees outline how the qualified data of fragments are composed into the XML result. The *constructor* receives the tuples in the SQL results and structures them into the XML result following the plan result tree. Formally a query plan is defined as follows.

Definition 13 (**(Valid) Query Plan**) A query plan wrt a schema decomposition D, is a tuple $\langle C', P', E, R' \rangle$, where C' is a plan condition tree, P' is a *plan decomposition*, and R' is a *plan result tree.*

C' has the structure of a query condition, except that some edges may be labeled as "soft". However, no path may contain a non-soft edge after a soft one. That is, all the edges below a soft edge have to be soft.

P' is a pair $\langle P, f \rangle$, where P is a partition of C' into fragments $P_1, \ldots, P_n$, and f is a mapping from P into the fragments of D. Every P_i has to be covered by the fragment $f(P_i)$ in the sense that for every node in P_i there is a corresponding schema node in $f(P_i)$.

R' is a tree that has the same structure as a query result tree. All variables that appear in R' outside the group-by labels, must bind to atomic elements[8] or bind to elements that are BLOB-ed in D.

◇

C' and R' are constructed from C, R and the schema decomposition D by the following nondeterministic algorithm. For every variable V, that occurs in R on node N_R and in C on node N_C, find the schema node S that corresponds to N_C, i.e. the path from the root of C to N_C and the path from the schema root to S have the same sequence of node labels. If S exists and is not atomic, there are two options:

1. Do not perform any transformations. In this case V will bind to BLOBs assuming that S is BLOB-ed in D.
2. Extend N_C with all the children of S. Label every new edge as "soft" if the corresponding schema edge has a "*" or a "?" label, or if the incoming edge of N_C is soft. Label every new node with a new unique variable V_i. If S is not repeatable, remove label V from N_C; otherwise, V will be used by a "group-by" label in R'. For every V_i that was added to N_C, extend N_R with a new child node labeled V_i. If S is repeatable, add a group-by label $\{V\}$ to N_R.

The above procedure is applied recursively to all the nodes of C'. For example, Figure 16 shows one of the query plans for the query of Figure 13. First, the *Order* is extended with *Number*, *Status*, *LineItem*, *Price* and *Date*. Then the *LineItem* is extended with all its attributes. The edge between the *Order* and the *LineItem* is soft (indicated by the dotted line) because, according to the schema, *LineItem* is an optional child of *Order*. Since the incoming edge of the *LineItem* is soft, all its outgoing edges are also soft. Group-by labels on *Order* and *LineItem* indicate that nested structures will

[8] It is easy to verify this property using the schema graph.

be constructed for these elements. Given the decomposition of Figure 17 which includes BLOBs of *Order* elements, another valid plan for the query of Figure 13 will be identical to the query itself, with a plan decomposition consisting of a single fragment.

We illustrate the translation of query plans into the SQL queries with the following example.

Example 5 Consider the valid query plan of Figure 16, which assumes the 4NF decomposition without BLOBs of Figure 11. This plan will be translated into SQL by the following process. First, we identify the tables that should appear in the SQL `FROM` clause. Since the condition tree is partitioned into four fragments, the `FROM` clause will feature four tables:

```
FROM Customer C, Order O,
LineItem L1, LineItem L2
```

Second, for each fragment of the condition tree, we identify variables defined in this fragment that also appear in the result tree. For every such variable, the corresponding fragment table attribute is added to the `SELECT` clause. In our case, the result includes all variables, with the exception of $\$P$:

```
SELECT DISTINCT C.name, O.order_id, O.number,
O.status, O.price, O.date,L1.lineitem_id,
L1.part_number, L1.supplier_number,
L1.price, L1.quantity, L1.discount
```

Third, we construct a `WHERE` clause from the plan condition expression and by inspecting the edges that connect the fragments of the plan decomposition. If the edge that connects a parent fragment P with a child fragment C is a regular edge, then we introduce the condition
`tbl_P.parent_attr = tbl_C.parent_ref`
If the edge is "soft", the condition
`tbl_P.parent_attr =* tbl_C.parent_ref`, where "`=*`" denotes a left outerjoin. An outerjoin is needed to ensure accurate reconstruction of the original document. For example, an order can appear in the result even if it does not have any lineitems. In our case, the `WHERE` clause contains the following conditions:

```
WHERE C.cust_id = O.cust_ref
AND O.order_id = L2.order_ref
AND O.order_id =* L1.order_ref
AND L2.price > 30000
```

Notice, that the above where clause can be optimized by replacing the outerjoin with a natural join because the selection condition on $L2$ implies that the order O will have at least one lineitem.

Finally, the clause `ORDER BY O.order_id` is appended to the query to facilitate the grouping of lines of the same order, which allows the XML result to be constructed by a constant space tagger [SSB+00].

The resulting SQL query is:

```
SELECT DISTINCT C.name, O.order_id, O.number,
O.status, O.price, O.date,L1.lineitem_id,
L1.part_number, L1.supplier_number,
L1.price, L1.quantity, L1.discount
FROM Customer C, Order O,
LineItem L1, LineItem L2
WHERE C.cust_id = O.cust_ref
AND O.order_id = L2.order_ref
AND O.order_id = L1.order_ref
AND L2.price > 30000
ORDER BY O.order_id
```

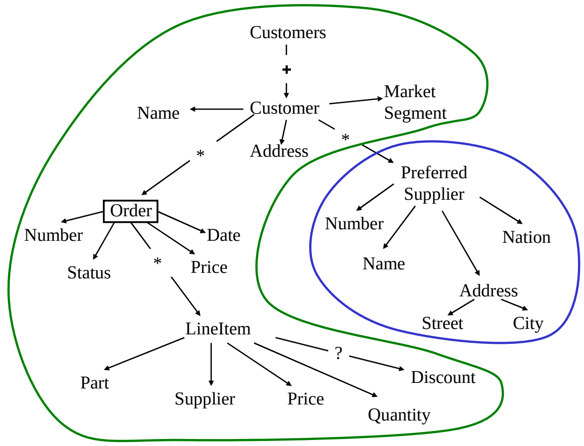


Fig. 17 Inlined schema decomposition used for the experiments

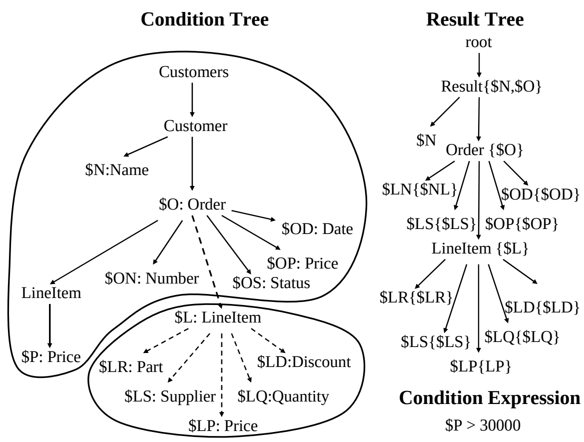


Fig. 18 A Possible Query Plan

Now consider a complete decomposition without BLOBs. Recall, that a complete decomposition consists of all possible non-MVD fragments. This decomposition, for instance, includes a non-MVD fragment Customer-Order-LineItem (COL for short) that contains all Customer, Order and LineItem information. This fragment is illustrated in Figure 17. The COL fragment can be used to answer the above query with only one join, using the query plan illustrated in Figure 18.

```
SELECT DISTINCT COL.name, COL.order_id,
COL.number, COL.status, COL.price, COL.date,
L1.lineitem_id, L1.part_number, L1.price,
L1.supplier_number, L1.quantity, L1.discount
FROM COL, LineItem L1
WHERE COL.order_id = L1.order_ref
AND COL.line_price > 30000
ORDER BY COL.order_id
```

Finally, consider the same complete decomposition that also features *Order* BLOBs. We can use the query plan identical to the query itself (Figure 13), with a single fragment plan decomposition. Again a single join is needed (with Blobs table), but the result does not have to be tagged afterwards. This also means that the ORDER BY clause is not needed.

```
SELECT DISTINCT COL.cust_name, Blobs.value
FROM COL, Blobs
WHERE COL.line_price > 30000
AND COL.order_id = Blobs.id
```

The XCacheDB also has an option of retrieving BLOB values with a separate query, in order to improve the query response time. Using this option we eliminate the join with the Blobs table. The query becomes

```
SELECT DISTINCT COL.cust_name, COL.order_id
FROM COL WHERE COL.line_price > 30000
```

The BLOB values are retrieved by the following prepared query:

```
SELECT value FROM Blobs WHERE id = ?
```

The above example demonstrates that the BLOBs can be used to facilitate construction of the results, while the non-4NF materialized views can reduce the number of joins and simplify the final query. The BLOBs and inlined decompositions are two independent techniques that trade space for performance. Both of the techniques have their pros and cons.

Effects of the BLOBs

Positive: Use of BLOBs may replace a number of joins with a single join with the Blobs table, which, as our experiments show, typically improves performance. BLOBs eliminate the need for the order-by clause, which improves query performance, especially the response time. BLOBs do not require tagging, which also saves time. BLOBs can be retrieved by a separate query which significantly improves the response time.

Negative: The BLOBs introduce significant space overhead. The join with the Blobs table can be expensive especially when the query results are large.

Effects of the Inlined decomposition

Positive: The denormalized decompositions reduce number of joins, which may lead to better performance. For instance, eliminating high start-up costs of some joins (e.g. hash join), improves query response time. Since the query has fewer joins, it is simpler to process; as the result, query performance is much less dependant on the relational optimizer. During our experiments with the normalized decompositions, we encountered cases when a plan produced by the relational optimizer simply could not be executed by our server. For example, one of such plans called for a Cartesian product of 5000 tuples with 600000. We never encountered such problems while experimenting with the inlined decompositions.

Negative: The scans of denormalized tables take longer because of the increased width. The inlining, also introduces space overhead.

5.2 Minimal Plans

Out of the multiple possible valid plans we are interested in the ones that minimize the number of joins.

Definition 14 (**Minimal Plan**) A valid plan is *minimal* if its plan decomposition P' contains the smallest possible number of partitions P_i. $\diamondsuit$

Still, there may be situations where there are multiple minimal plans. In this case the plan generator uses the following heuristic algorithm, which is linear in the size of the query and the schema decomposition. When the algorithm is applied on a minimal non-MVD decomposition it is guaranteed to produce a minimal plan.

– 1. Pick any leaf node N of the query.

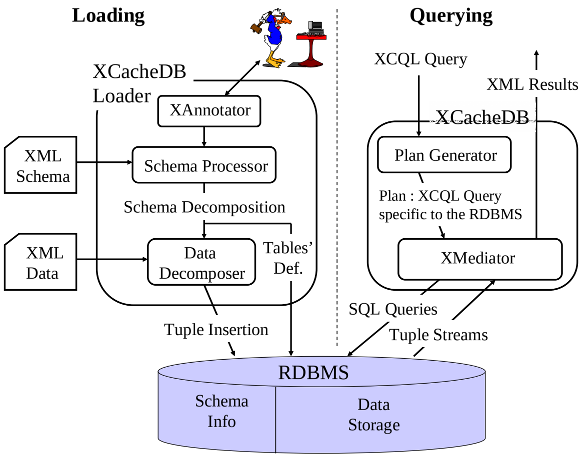


Fig. 19 The XCacheDB architecture

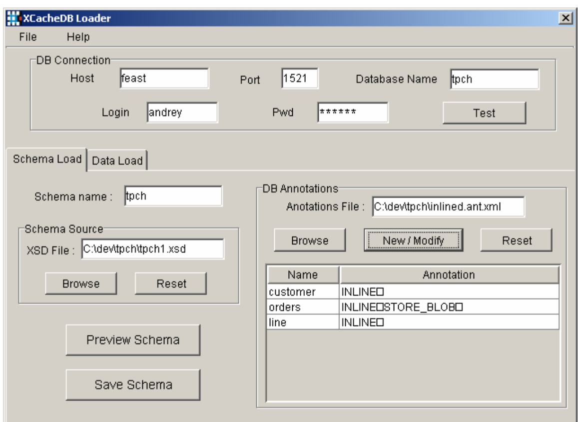


Fig. 20 The XCacheDB Loader utility

– 2. Find the fragment F that covers N and goes as far up as possible (covers the most remote ancestor of N)
– 3. Remove from the query tree, the subtree covered by F
– 4. Repeat the above steps until all nodes of the query are covered.

The advantage of this algorithm is that it avoids joins at the lower levels of the query – where most of the data is usually located. For example, in the TPC-H dataset we used for the experiments (it conforms to the schema of Figure 4, the `Order` $\bowtie$ `LineItem` join is 40 times bigger (and potentially more expensive) than the `Customer` $\bowtie$ `Order` join.

6 Implementation

The XCacheDB system [BPSV00] of Enosys Software, Inc., is an XML database built on top of commercial JDBC-compliant relational database systems. The abstract architecture of Figure 15 has been reduced to the one of Figure 19, where the plan translation and construction functions of the query processor are provided by the XMediator [ES00] product of Enosys Software, Inc. Finally, the "optional user guidance" of Figure 1 is provided via the *XAnnotator* user interface, which produces a set of *schema annotations* that affect decomposition.

The XCacheDB loader supports acyclic schemas, which by default are transformed into tree schemas. By default, the XCacheDB loader creates the minimal 4NF decomposition. However, the user can control the decomposition using the schema annotations and can instruct the XCacheDB what to inline and what to BLOB. In particular, the XAnnotator (Figure 21) displays the XML Schema and allows the user to associate a set of annotation keywords with nodes of the schema graph. The following six annotation keywords are supported. We provide a brief informal description of their meaning:

`INLINE`: placed on a schema node n it forces the fragment rooted at n to be merged with the fragment of the parent of n.
`TABLE`: placed on a schema node n directs the loader to create a new fragment rooted at n.
`STORE_BLOB`: placed on a schema node n it indicates that a BLOB should be created for elements that correspond to this node.
`BLOB_ONLY`: implies that the elements that correspond to the annotated schema node should be BLOB-ed and not decomposed any further.
`RENAME`, `DATATYPE`: those annotations enable the user to change names of the tables and columns in the database, and data types of the columns respectively.

A single schema node can have more than one annotation. The only exception is that `INLINE` and `TABLE` annotations cannot appear together, as they contradict each other.

The XCacheDB loader automatically creates a set of indices for each table that it loads. By default, an index is created for every data column to improve performance of selection conditions, but it can be switched off. An index is also created for a parent reference column, and for every node-ID column that gets referenced by another table. These indices facilitate efficient joins between fragments.

Query processing in XCacheDB leverages the XMediator, which can export an XML view of a relational database and allow queries on it. The plan generator takes an XML query, which was XCQL [PV01] and is now becoming XQuery, and produces a query algebra plan that refers directly to the tables of the underlying relational database.

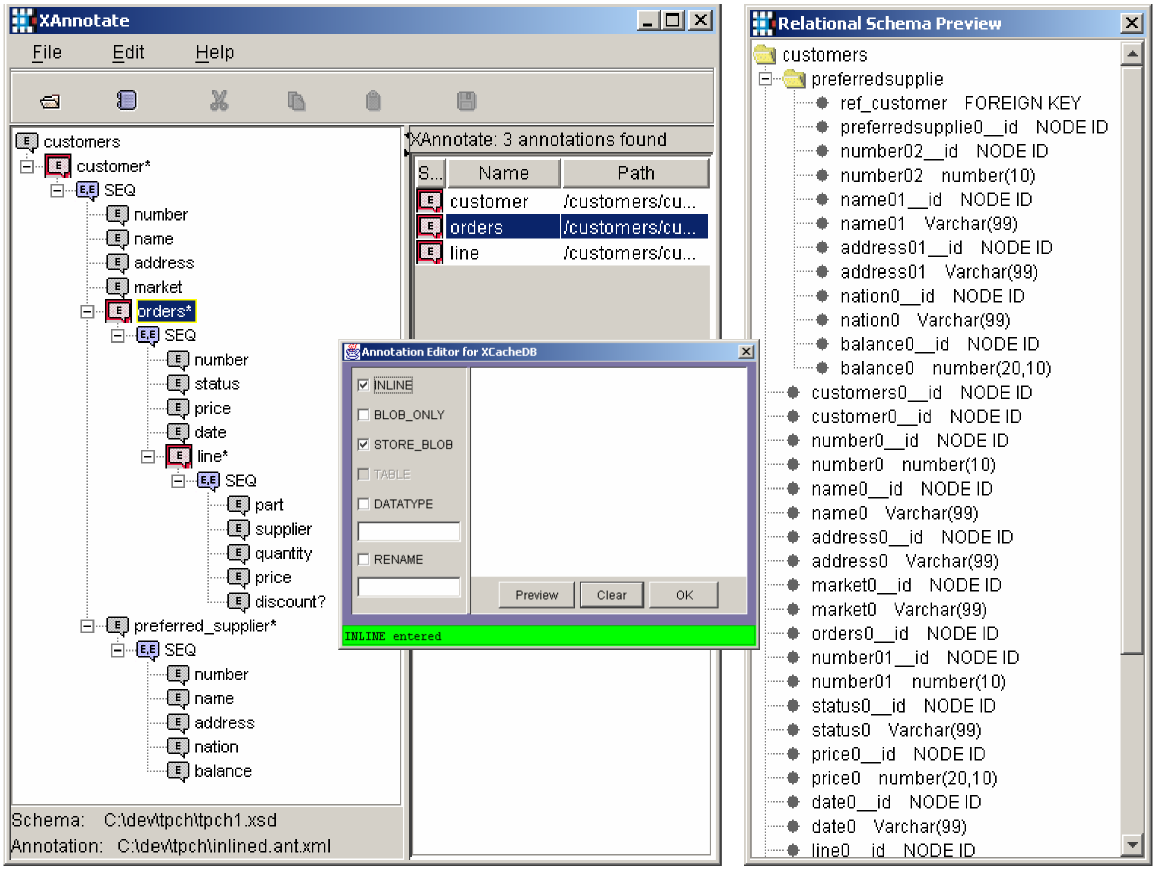


Fig. 21 Annotating the XML schema and resulting relational schema

This plan can be run directly by the XMediator's engine, since it is expressed in the algebra that the mediator uses internally.

7 Experimentation

This section evaluates the impact of BLOBs and different schema decompositions on query performance. All experiments are done using an "TPC-H like" XML dataset that conforms to the schema of Figure 4. The dataset contains 10000 customers, 150000 orders, ~120000 suppliers and ~600000 lineitems. The size of the XML file is 160 MB. Unless otherwise noted, the following system configuration is used. The XCacheDB is running on a Pentium 3 333MHz system with 384MB of RAM. The underlying relational database resides on a dual Pentium 3 300MHz system with 512MB of RAM and 10000RPM hard drives connected to a 40MBps SCSI controller. The database server is configured to use 64MB of RAM for buffer space. We flush the buffers between runs, to ensure the independence of the experiments. Statistics are collected for all tables and the relational database is set to use the cost-based optimizer, since the underlying database allows both cost-based and rule-based optimization. The XCacheDB connects to the database through a 100Mb switched Ethernet network. We also provide experiments with 11Mb wireless Ethernet connection between the systems, and show the effects of a lower-bandwidth, high-latency connection.

All previous work on XML query processing, concentrated on a single performance metric – *total time*, i.e. time to execute the entire query and output the complete result. However, we are also interested in *response time*. We define the response time as the time it takes to output the first ten results.

Queries We use the following three queries (see Figure 22):

Q1 The selection query of Example 4 returns pairs of customer names and their orders, if the order contains at least one lineitem with $price > P$, where P is a parameter that ranges from 75000 (qualifies about 15% of tuples) to 96000 (qualifies no tuples).

Q2 also has a range condition on the supplier. The parameter of the supplier condition is chosen to

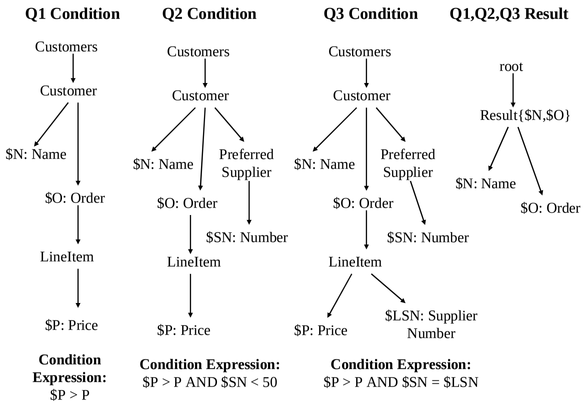


Fig. 22 Three queries used for the experiments

filter out about 50% of customers on the average. Notice that since this query refers to both orders and suppliers, it cannot be answered using a single non-MVD fragment.

Q3 This query finds customers that have placed expensive orders with preferred suppliers (i.e. customer contains a prefered supplier and an order with an expensive item from this supplier.) Notice the join between `Supplier` and `LineItem`.

Testing various decompositions We compare the following query decompositions:

1. 4NF schema decomposition without BLOBs, which consists of the following four tables: `Customer`, `Order`, `LineItem`, and `PrefSupplier` (Figure 11). The above four tables occupy 64 MB of disk space. This case corresponds to a typical decomposition considered in the previous work [STZ+99, BFRS02].
2. Same 4NF decomposition as above with the addition of a BLOBs table that stores Order subtrees. This table takes up 150 MB.
3. Inlined decomposition of Figure 17, which includes two non-MVD fragments: `Supplier` and `Customer-Order-Line`. These two tables occupy 137.5 MB. This decomposition also includes Order BLOBs.

We also consider a decomposition that contains an MVD fragment `Customer-Order-Supplier` and a separate table for `LineItem`. However, the experiments show that this approach is not competitive. The space overhead (the two tables take up almost 600 MB) translates into poor query performance.

Discussion The left side of Figure 23 shows the total execution time of the three queries plotted against the selectivity of the condition on price, which essentially controls the size of the result. For Q1, 1% selectivity translates into a 1.5 MB result XML file. For Q2 and Q3 the rates are about 0.75 MB and 0.4 MB respectively. The right side of Figure 23 shows the response time of the same queries. Recall, in the response time experiments the queries return top ten top-level objects, i.e. the result size is constantly around 10 KB.

All the "total time" graphs exhibit the same trend. The "4NF" line starts higher than the "inlined" one, because of the time it takes the database to initiate and execute multiway joins. However, the slope of the "inlined" line is steeper because of the space and I/O overhead derived from the denormalization. Table scans take longer on the "inlined" tables.

BLOBs improve performance of the small queries, but their effects also diminish as the result sizes grow. For smaller results (less than 2 MB of XML) 4NF with BLOBs consistently outperforms 4NF without BLOBs by 200% to 300%. As the result sizes increase, join with the BLOBs table becomes more expensive in comparison to the extra joins needed to reconstruct the result fragments.

The main advantage of the XCacheDB is its response time. Both inlining and BLOBs significantly simplify the SQL query which is sent to the relational database, which allows the server to create the result cursor, in some cases, almost instantaneously.

The irregularities that are be observed in the graphs (e.g. a notch on the "4NF without BLOBs" line of all three "total time" graphs around the 1% selectivity) are mostly due to the different plans picked by the relational optimizer for different values of the parameter. Notice that on the Q3 response time graph the "Inlined" line uncharacteristically dips between the 6-th and 7-th points (selectivity values 1.6% and 4.3%). It turns out that at that point the optimizer reversed the sides of the hash join of `COL` and `Supplier` tables, which improved performance.

7.1 Effects of higher CPU/bandwidth ratio

Query processing on the inlined schema requires less CPU resources than on the 4NF schema, since fewer joins need to be performed. However, prejoined data needs to be read from the disk, which requires more I/O operations than reading data required for the join. If the database optimizer correctly picks join ordering and join strategies, a table will not be scanned more than twice for a join, and most of the time, a single scan will be sufficient [GMUW99]. This tradeoff was observed when the database was

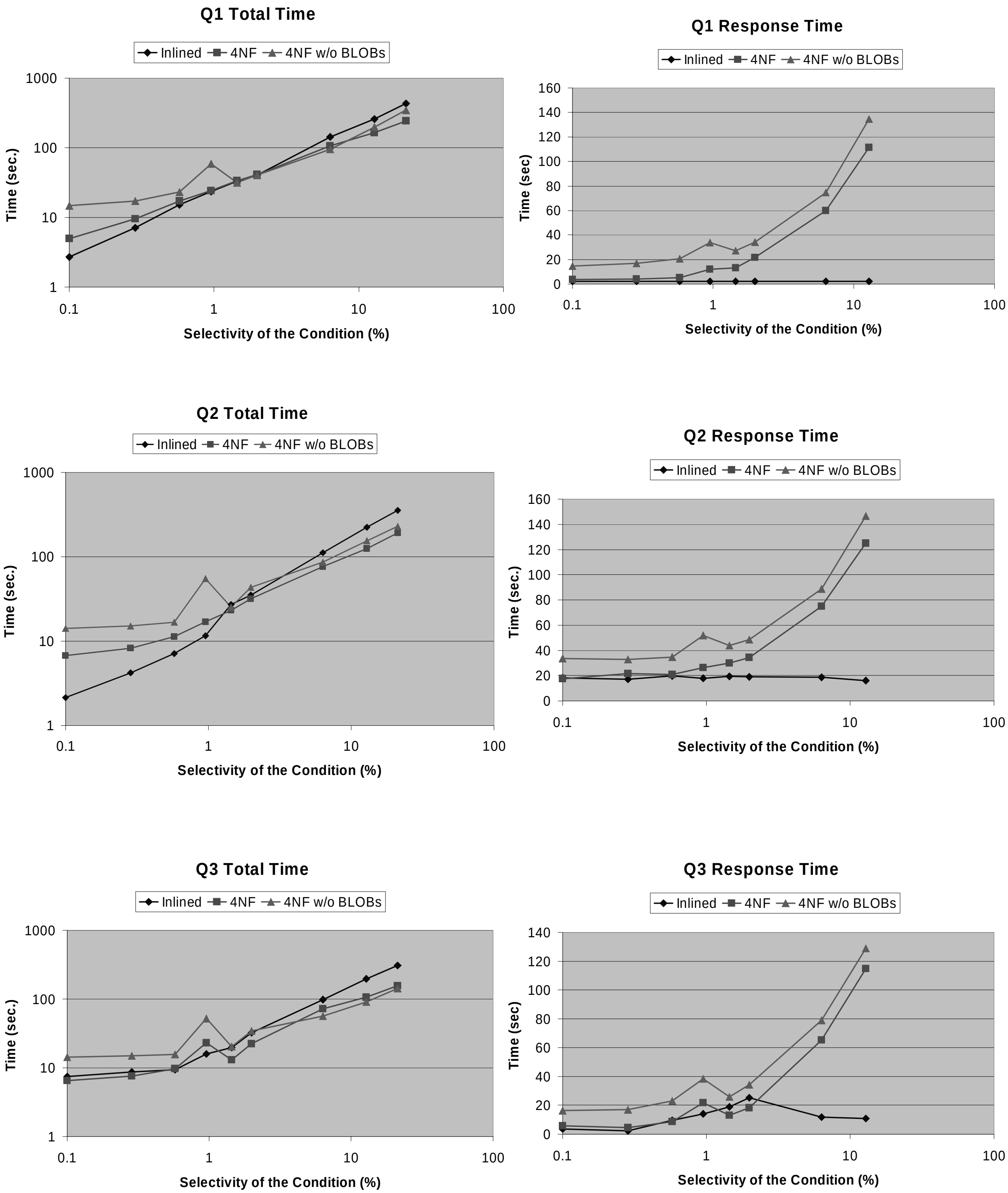


Fig. 23 Experimental results

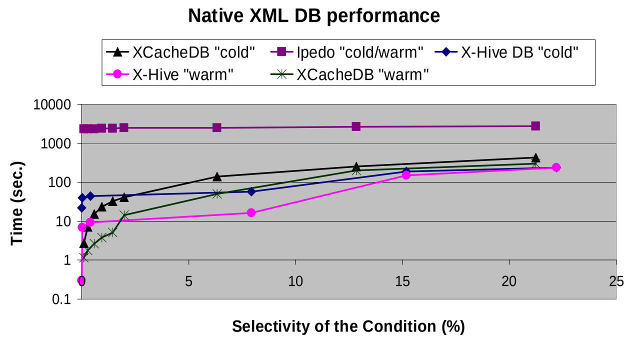


Fig. 24 The total execution time of Q1 on native XML databases vs. XCacheDB

installed on a 500MHz system with a slow (4200RPM) IDE disk. In this setting, the 4NF decomposition with BLOBs often provided for faster querying than the inlined one. For example, in a fast disk setup a Q1-type query with result size 2 MB according to Figure 23 takes about 7.5 sec on both 4NF and Inlined schemas. On a server with slower disk the same query took 8.2 sec with the 4NF decomposition and 11.6 sec with the Inlined decomposition.

BLOBs are sensitive to interconnect speeds between the database server and the XCacheDB, since they include tags and structure information in addition to the data itself. BLOB-ed query results are somewhat larger than those containing only atomic values, and on slower, high-latency links, network speed can become the bottleneck. For example, Q1 with BLOBs takes 34.2 seconds to complete on a 11Mb wireless network. The same query in the same setup, but on a 100Mb Ethernet takes only 7.5 sec.

7.2 Comparison with a Commercial XML Database

We compared the performance of XCacheDB against two commercial native XML database systems: X-Hive 4.0 [X-H] and Ipedo 3.1 [Ipe]. For this set of experiments we only measured total execution time, because these two databases could not compete with the XCacheDB in response time, since they are unable to return the first result object before the query execution is completed.

Both systems support subsets of XQuery which include the query Q1, as it appears in Example 4 and as it was used in the XCacheDB experiments above. However, we did not use Q1 because the X-Hive was not able to use the value index to speed-up range queries. Thus, we replaced range conditions on price elements with equality conditions on "part", "supplier", and "quantity" elements, which have different selectivities.

For Ipedo we were not able to rewrite the query in a way that would enable the system to take advantage of the value indices. As a result, the performance of the Ipedo database was not competitive (Figure 24), since a full scan of the database was needed every time to answer the query.

In all previous experiments we measured and reported "cold-start" execution times, which for X-Hive were significantly slower than when the query ran on "warm" cache. For instance, the first execution of a query that used a value index, generated more disk traffic than the second one. It may be the case that X-Hive reads from disk the entire index used by the query. This would explain relatively long (22 seconds) execution time for the query that returned only four results. The second execution of the same query took 0.3 seconds. For the less selective queries the difference was barely noticeable as the "warm" line of Figure 24 indicates.

We do not report results for the Q3 query, since both X-Hive and Ipedo where able to answer it only by a full scan of the database, and hence they were not competitive.

8 Conclusions and Future Work

Our approach towards building XML DBMS's is based on leveraging an underlying RDBMS for storing and querying the XML data in the presence of XML Schemas. We provide a formal framework for schema-driven decompositions of the XML data into relational data. The framework encompasses the decompositions described in prior work and takes advantage of two novel techniques that employ denormalized tables and binary-coded XML fragments suitable for fast navigation and output. The new spectrum of decompositions allows us to trade storage space for query performance.

We classify the decompositions based on the dependencies in the produced relational schemas. We notice that non-MVD relational schemas that feature inlined repeatable elements, provide a significant improvement in the query performance (and especially in response time) by reducing the number of joins in a query, with a limited increase in the size of the database.

We implemented the two novel techniques in XCacheDB – an XML DBMS built on top of a commercial RDBMS. Our performance study indicates that XCacheDB can deliver significant (up to 400%) improvement in query execution time.

Most importantly, the XCacheDB can provide orders of magnitude improvement in query response time, which is critical for typical web-based applications.

We identify the following directions for future work:

– Extend to more complex queries.
– Extend our schema model from DAG's to arbitrary graphs. This extension will increase the query processing complexity, since it will allow recursive queries which cannot be evaluated in standard SQL.
– Consider a cost-based approach for determining a schema decomposition given a query mix, along the lines of [BFRS02].
– Enhance the query processing to consider plans where some of the joins may be evaluated by the XCacheDB. Similar work was done by [FMS01], however, they focused on materializing large XML results, whereas our first priority is minimizing the response time.

References

[Apa] Apache Software Foundation. Xindice. http://xml.apache.org/xindice/.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system : the story of O2*. Morgan-Kaufmann, 1992.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of ICDE*, 2002.

[BKKM00] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. Oracle8i - the XML enabled data management system. In *ICDE 2000, Proceedings of the 16th International Conference on Data Engineering*, pages 561–568. IEEE Computer Society, 2000.

[BPSV00] A. Balmin, Y. Papakonstantinou, K. Stathatos, and V. Vassalos. System for querying markup language data stored in a relational database according to markup language schema, 2000. Submitted by Enosys Software Inc. to USPTO.

[Coh] Coherity. Coherity XML database (CXD). http://www.coherity.com.

[Cou99] Transaction Processing Performance Council. TPC benchmark H, 1999. Standard specification available at http://www.tpc.org.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*. ACM Press, 1999.

[Ell] Ellipsis. DOM-Safe. http://www.ellipsis.nl.

[ES00] Inc. Enosys Software. The universal, real-time data integration platform, 2000. White paper at http://www.enosyssoftware.com.

[eXc] eXcelon Corp. eXtensible information server (XIS). http://www.exln.com.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FMS01] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD Conference*, 2001.

[FTS00] Mary F. Fernandez, Wang Chiew Tan, and Dan Suciu. SilkRoute: trading between relations and XML. In *WWW9 / Computer Networks*, pages 723–745, 2000.

[GMUW99] H. Garcia-Molina, J. Ullman, and J. Widom. *Principles of Database Systems*. Prentice Hall, 1999.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the lore data model and query language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[HSKA$^+$] H.V.Jagadish, S.Al-Khalifa, A.Chapman, L.V.S.Lakshmanan, A.Nierman, S.Paparizos, J.Patel, D.Srivastava, N.Wiwatwattana, Y.Wu, and C.Yu. TIMBER: A native XML database. *To appear in VLDB Journal*.

[Inf] Infonyte. Infonyte DB. http://www.infonyte.com.

[Ipe] Ipedo. Ipedo XML DB. http://www.ipedo.com.

[JMC00] Jane Xu Josephine M. Cheng. XML and DB2. In *ICDE 2000, Proceedings of the 16th International Conference on Data Engineering*, pages 569–573. IEEE Computer Society, 2000.

[KM92] Alfons Kemper and Guido Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *IS 17(2)*, pages 117–145, 1992.

[Kru] Ari Krupnikov. DBDOM. http://dbdom.sourceforge.net/.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *VLDB 2001, Proceedings of the 27th International Conference on Very Large Databases*, pages 361–370, 2001.

[MFK$^+$00] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Dan

Olteanu. Agora: Living with XML and relational. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[M/G] M/Gateway Developments Ltd. eXtc. http://www.mgateway.tzo.com/eXtc.htm.

[NDM+01] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara internet query system. In *IEEE Data Engineering Bulletin 24(2)*, pages 27–33, 2001.

[Neo] NeoCore. Neocore XML management system. http://www.neocore.com.

[Ope] OpenLink Software. Virtuoso. http://www.openlinksw.com/virtuoso/.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice Hall, 1999.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 35–46. ACM, 2000.

[PV01] Yannis Papakonstantinou and Vasilis Vassalos. The Enosys Markets data integration platform: Lessons from the trenches. *CIKM*, pages 538–540, 2001.

[Rys01] Michael Rys. State-of-the-art XML support in RDBMS: Microsoft SQL server's XML features. In *IEEE Data Engineering Bulletin 24(2)*, pages 3–11, 2001.

[SKWW01] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. In Dan Suciu and Gottfried Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 47–52. Springer, 2001.

[SSB+00] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 65–76. Morgan Kaufmann, 2000.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SW00] Harald Schöning and Jürgen Wäsch. Tamino - an internet database system. In *EDBT 2000, Proceedings of the 7th International Conference on Extending Database Technology*, pages 383–387, 2000.

[W3C98a] W3C. Document object model (DOM), 1998. W3C Recomendation at http://www.w3c.org/DOM/.

[W3C98b] W3C. The extensible markup language (XML), 1998. W3C Recomendation at http://www.w3c.org/XML.

[W3C01a] W3C. XML schema definition, 2001. W3C Recomendation at http://www.w3c.org/XML/Schema.

[W3C01b] W3C. XQuery: A query language for XML, 2001. W3C Working Draft at http://www.w3c.org/XML/Query.

[Wir] Wired Minds. MindSuite XDB. http://xdb.wiredminds.com/.

[X-H] X-Hive Corporation. X-Hive/DB. http://www.x-hive.com.

[XML] XML Global. GoXML. http://www.xmlglobal.com.

[XP] Yu Xu and Yannis Papakonstantinou. XSearch demo. http://www.db.ucsd.edu/People/yu/xsearch/.

[XYZ] XYZFind Corporation. XYZFind server. http://www.xyzfind.com.