XQuery/IR: Integrating XML Document and Data Retrieval

Jan-Marco Bremer    Michael Gertz
Department of Computer Science
University of California, Davis, CA
{bremer|gertz}@cs.ucdavis.edu

1 Introduction

One of the most important features of XML is to provide a unified view to all kinds of structured and semistructured data as well as loosely structured documents. Here, document means a coherent unit of usually textual information while data stands for uninterpreted, raw content without a fixed context. Most content XML is applied to today is very text-rich. The structural aspects of the unified XML view are rigid enough to support data retrieval (DR) queries as known from database systems. Over the past few years, increasingly powerful query language, most notably the recent XQuery standard [2], have exploited this fact to provide expressive DR query capabilities for XML. On the other hand, XML’s structural aspects are transparent enough to treat arbitrary parts of the XML-represented data as documents. Document or information retrieval (IR) providing one of the most important capabilities for querying text-rich documents, however, is not supported by XQuery or any of the earlier XML query languages so far.

DR provides means to formulate queries based on exact matches of data. IR is based on the notion of (relative) relevance of documents within a document collection. Thus, some major issues for the integration of IR into an XML query language arise: Today’s (toy) collections of small XML documents are eventually to be replaced by large, hierarchically structured XML databases. In such databases, the notions of static document collection and document are lost and have to be replaced by some kind of view on documents and document collections that is established dynamically by a query. This poses challenges for new, embedded IR algorithms and their application within the host language rather than only as the final operation as in standard IR. However, besides these challenges, significant rewards in form of an increased expressiveness of the resulting integrated query language might arise.

IR in general is based on detecting fine differences in term distributions throughout a document collection leading to valuable information [12]. However, the value of information is highly context-dependent. Currently, the context for IR is always an entire, static document collection. This changes when term statistics for dynamic document views are available and exploited by an IR algorithm. A DR language is a natural means to specify a context for a relevance-based query. Moreover, when IR is truly embedded into the query language, derived information can be obtained after the IR-style ranking has identified the most relevant components. For example, the mean publication date for information related to a certain event reported in textual news naturally leads to an apex of that event.

In this paper, we present the results of our ongoing research to integrate IR capabilities into XQuery and through this, provide more expressive queries than DR or IR alone can answer. XQuery-related standards [2, 5] define document fragment sequences (DFSs) as input and output for all intermediate and final query results. We identify document fragments (DFs) and DFSs as the equivalent of documents and document collections within XML queries. We introduce a single, new operator called rank into XQuery. The operator orders DFs within arbitrary intermediate DFSs. It is used through a Rankby expression that is very similar to XQuery’s Sortby expression. The operator is orthogonal to other XQuery operators and can be arbitrarily nested.

The rank operator itself imposes no requirements on the implementation of the relevance-based ordering of DFs. Thereby, we establish a general framework for research into query language-embedded IR algorithms. However, we define a dynamic ranking principle that captures IR algorithms which allow the richest type of queries based on a local context. These algorithms are based on term statistics only within the current DFS as returned by a (sub-)query. We briefly study consequences of the rank operator for index structures and query optimization.

Related work. Oracle’s contains operator implements a standard, static IR approach based on table columns [1]. The operator is used within an SQL Where
clause. In [6], only exact keyword search within certain XML elements is introduced into XML-QL. Earlier related approaches for structured documents can be found in [11]. The aim of [13] is an improved Web search by means of an IR-enabled XML query language. The similarity operator of [3] is mainly a vehicle to execute fuzzy similarity joins based on atomic values within two DFs. Again, XML-QL is used as host language. Based on XQL, [8] proposes an IR-style operator that computes similarity between DFs and queries by means of XML leaf elements’ similarity weights that are propagated upwards. However, XQL is less expressive than XQuery and lacks important data retrieval capabilities.

Common to all of these approaches is the extension of a Where clause which implies a selection and thus, a partitioning of the input into relevant and non-relevant objects. The relevance-based sorting employed by our scheme more naturally reflects IR’s ordering properties while still allowing the partitioning by means of a relevance threshold. Moreover, unlike our approach, none of the existing approaches is able to demonstrate meaningful and useful nested queries involving both DR and IR operations. We have not yet seen other work applying IR to a local context established by a DR (sub-) query or a previous IR-style query.

Passage retrieval approaches [9] have used more fine-grained document components to improve the ranking but without supporting IR on document parts as a local context. The feasibility and usefulness of IR index structures for semistructured data as required by our approach have been shown in [10].

Overview. In Section 2, we establish the background for this paper. In Section 3, we introduce the new operator and demonstrate its application. Section 4 discusses implementation issues of the operator including dynamic ranking, index structures and query optimization. In Section 5, we conclude this paper and briefly point out some future work.

## 2 Background

### 2.1 XML Data Model

We employ an XML model in which documents and fragments of documents are represented as ordered, node-labeled trees. We leave out details such as comments, processing instructions, references, and the distinction between elements and attributes.

Assume a set $E$ of element names and a set $T$ of text string values disjoint from $E$. Given a set $X$, let $L(X)$ denote the set of all lists that can be built over elements from $X$. Then an XML document fragment $F$ is a 4-tuple $(V, r, label, elem)$ where

- $V$ is a set of vertices with a distinguished element $r$, called the root node,
- $label$ is a mapping from vertices to element labels, i.e., $label : V \rightarrow E$, and
- $elem$ is a mapping from vertices to their children, i.e., $elem : V \rightarrow L(V \cup T)$.

Let an XML document $D$ be an XML document fragment (DF). Furthermore, let $\mathcal{F}$ denote the set of all document fragments over $E$ and $T$. Then an XML document fragment sequence (DFS) $S$ is a sequence of elements of $\mathcal{F}$, i.e., $S \in L(\mathcal{F})$. Let $First_k(S)$ $(k \in \mathbb{N} \cup \infty)$ denote the sub-sequence of $S$ consisting of the first $k$ or $|S|$ elements, whichever is smaller.

### 2.2 XQuery

XQuery [2] combines features from several earlier XML query languages, in particular XPath [4]. Through XPath, DFs can be extracted from an XML document. Nested loops iterate over these fragments to further extract DFs and construct sequences of output DFs. Variable assignment supports complex computations based on content and structure of the input.

Details of XQuery’s formal semantics can be found in [7] and are not discussed here. For our approach, it is sufficient to note that in- and output to XQuery queries are always DFSs, i.e., a query $q$ is a mapping $q : L(\mathcal{F}) \rightarrow L(\mathcal{F})$. We illustrate the syntax and semantics of XQuery by means of a few simple examples. We will extend these examples in Section 3 to illustrate our new operator.

**Example 2.1** Select paragraphs of articles dating back to Feb. 15th, 2002 from the news document database.

document("news.xml")
//article[./date="2002-02-15"]//paragraph

The example consists of a single XPath expression, which alone is a valid XQuery query.

**Example 2.2** List all articles that appeared before 1996 with their first author and title, in sorted order.

FOR $a$ IN document("bib.xml")/article
WHERE $a$/year < 1996
RETURN
<early_paper>
Example 2.2 consists of a single loop that goes through articles and extracts the author and title. The complete title DF builds a part of the result fragments. A new tag name fstAuth is introduced for the author. The returned fragments are ordered by author and title.

Example 2.3 Convert a list of news articles classified under a certain category to a list of categories with their related articles.

Example 2.3 represents a join between category and article DFs based on a news category identifier. The join is implemented as two nested loops.

2.3 Document Retrieval

Document retrieval or, more commonly used, information retrieval (IR) is concerned with ordering documents from a document collection by relevance to a query [12]. The query is usually simple and consists of just a few terms. Relevance is based on term distribution statistics. A numerical weight is assigned to each term occurrence in each document. The weight represents the term’s significance within the document content. The relevance ordering is obtained by summing up query term weights for every document and determining the highest sum. A second flavor of IR is only concerned with partitioning the collection into relevant and non-relevant documents. It can be implemented by means of the ordering approach and the application of a threshold below which documents are regarded as not relevant.

A standard approach for weighting terms is the term frequency-inverse document frequency (tf-idf) approach [12]. Tf-idf assigns higher weights to term occurrences with high in-document and low overall document frequencies, i.e., few documents that contain the term. Term distributions may vary widely throughout a document collection. Hence, the significance of a term occurrence and thus, its relevance, highly depends on the context. In most existing retrieval approaches, the context is always the complete document collection.

3 XQuery Rank Operator

In traditional IR, the result of a usually stand-alone query is a total order or partitioning of documents that are the single unit within a collection. The result is directly presented to the user. When integrating IR into an XML query language, this raises two major questions:

1. What is the appropriate equivalent for documents and document collections within a single database-like XML document source (or set of such)?

2. How and where can IR be of use within a query? In particular, can it make sense to consider IR not only as a final operation, but one that does something meaningful in an intermediate query step?

XML queries extract document fragments (DFs) from XML sources. DFs are the only data unit suitable to replace the notion of document from standard IR. In XQuery in particular, all results are sequences of DFs (DFSs). It is only natural to modify the order of DFs within a DFS by means of an ordering IR approach, very similar to the sort operator in XQuery. This can be accomplished by a single operator. DFSs replace the notion of document collections. The order is transparent to subsequent queries that do not rely on any order, but can be exploited by other sub-queries or shown as an end result. The relevance-based partitioning can then be implemented on top of the ordering IR as discussed in Section 2.3.

The above observations let us derive the following requirements for integrating an IR operator into an XML query language, in particular XQuery.

1. **Total order.** The operator should be able to order a sequence of DFs based on relevance.

2. **Local context.** The partitioning flavor of IR should be supported to establish a local context for subsequent queries.

3. **Closure.** The operator should be closed within XQuery and thus, be applicable within arbitrary XQuery expressions.
**Transparency.** The operator should not affect queries in ways other than changing the DFS-internal order or eliminating elements of the DFS.

**Exchangeability.** The IR weighting algorithm underlying the operator should be exchangeable. A user should have means to choose the weighting algorithm for a query.

There is an additional requirement we impose, because it appears to make the IR-style ranking within the host language even more useful:

**Visibility.** The operator should assign visible ranking weights to DFs, but without causing side-effects to the embedding query.

The exchangeability property of the operator allows for different IR algorithms to be plugged into the query engine. Although we have not introduced any limitations here, we identify a certain class of algorithms called dynamic ranking algorithms as required to establish a real local context for sub-queries. These algorithms will be subject of the next section.

In the following, we introduce the syntax and semantics of an operator that meets the above requirements. The elegant simplicity of the XQuery extension should allow to easily understand the operator’s functionality, even though space limitations prevent us from going into all the details.

### 3.1 Syntax

We propose to add a single new operator called `rank` to the XQuery language. The operator is used within a Rankby expression that extends the set of base XQuery expressions. A Rankby expression is very similar to XQuery’s Sortby expression [2, Section 2.4].

**Definition 3.1 (Rankby expression)**

\[
\text{RankBy}::= \text{Expr} \
\quad \text{"rankby"} \\text{("QuerySpecList")} \\
\quad \text{("ascending" | "descending")?} \\
\quad \text{["basedon" "(" TargetSpecList ")"]} \\
\quad \text{["limit" n ["%"]]} \\
\quad \text{["using" MethodFunctCall]} \\
\]

\[
\text{QuerySpecList ::= Expr ("," TargetSpecList)?} \\
\text{TargetSpecList ::= PathExpr ("," TargetSpecList)?} \\
\]

QuerySpecList is a list of strings (constant DFs) or expressions that return DFs that can be interpreted as strings in XQuery as well. TargetSpecList is a list of context node-dependent path expressions. MethodFunctCall refers to an XQuery function.

The syntax and semantics of XQuery functions (FunctionCall in the specification) are not yet fully specified by the W3C. MethodFunctCall is more of symbolic nature. It represents an IR algorithm that may or may not be implemented as some kind of stored procedure.

### 3.2 Semantics

Assume the set \( L(F) \) of document fragments (Section 2.1). Let `weight` be a special element name in \( E \).

**Definition 3.2 (Weighting algorithm)**

Assume a set of IR queries \( Q = L(F) \) which includes the DFs consisting of only a string from \( T \), in particular single terms. Let \( W \) be the set of operators

\[
W: L(F) \times Q \rightarrow L(F), \\
W(S,Q) = S' 
\]

such that \( S' \) is equal to \( S \) except that each DF in \( S' \) has an additional element named `weight` at the root. The `weight` element has a single number string as child representing the relative relevance weight of the respective DF within \( S \). Then a \( W \in W \) is called a weighting algorithm.

Within the Rankby expression, the MethodFunctCall part represents a certain exchangeable weighting algorithm. Related to the QuerySpecList, queries are DFs that can be interpreted as query text. The actual query interpretation and limitations to the allowed query types depend on the weighting algorithm \( W \). In particular, typical IR term queries like (“sun”, “moon”) can be regarded as DFs with only a single text string node. More advanced weighting algorithms may make use of the structure of query DFs. In the following, we define the actual ranking operator in terms of a weighting algorithm and XQuery’s existing sort operator.

**Definition 3.3 (Rank operator)**

The `rank` operator is an operator:

\[
\text{rank} : L(F) \times N \times Q \times W \rightarrow L(F) \\
\text{rank}(S,k,Q,W) = \text{First}_k(\text{sort}_\text{weight}(W(S,Q))), 
\]

where \( N \) is the set of natural numbers and \( Q \) defined as above. `sort_weight` refers to the sorting of DFs within \( S \) based on the weight element. `First_k` eliminates all but the first \( k \) elements from \( S \) (`First_\infty(S) = S`).

If there is a non-empty TargetSpecList in the Rankby expression, for each DF in \( S \), the input to `rank` are the
fragments selected by the TargetSpecList’s path expression, otherwise the complete DFs. The elimination of all but the first \( k \) DFs relates to the limit clause of the Rankby expression. As an extension, the limit could also be based on a certain percentage of the DFS’s total weight to be kept after the elimination.

The weighting algorithm \( W \), exchangeable within \( \text{rank} \), implements the total order as called for in the requirements. It adds a weight element to a DF making the ranking visible, a further requirement. Note that in XML, we envision an attribute to be added. An attribute is much more transparent to the rest of a query, but not an explicit part of our simplified data model. Full transparency of the ranking, however, can only be achieved through introducing a reserved name for \( \text{weight} \) in XQuery. XQuery’s closure properties are kept as DFSs remain the only type of in- and output.

We call the XQuery query language extended by the \( \text{rank} \) operator \( \text{XQuery/IR} \). Due to space limitations we have to illustrate the expressiveness of \( \text{XQuery/IR} \) by means of the following examples in favor of a more complete discussion.

### 3.3 Examples

Example 2.1 (simplified) can be extended to retrieve a maximum of 100 paragraphs with relevant information about New York as follows:

```
document("news.xml")//article//paragraph
rankby ("New York") limit 100
```

The result might look like:

```
<paragraph weight=0.96>
The New Yorker fire fighters . . .</paragraph>
```

```
<paragraph weight=0.81>
Weekend weather in New York promises to . . .</paragraph>
```

```
<paragraph weight=0.79>
. . .</paragraph>
```

In Example 2.2, instead of sorting the result by author, a relevance-based ranking only based on the articles' abstracts can be obtained through:

```
FOR $a IN document("bib.xml")//article
WHERE $a/year < 1996
RETURN
<early_paper>
  <fstAuth> {$a/authors/author[1]/text()} </fstAuth>
  {$a/title}
</early_paper>
RANKBY ("Albert", "Einstein") BASEDON (abstract)
```

Note that \( \text{abstract} \) as argument to \( \text{BASEDON} \) is an XPath expression relative to the context node \( $a \). The terms “Albert” and “Einstein” could be replaced by a subquery that selects text to be used as query from another document like:

```
document("authors.xml")//author[./name="Einstein"]/accomplishments
```

A modification to the earlier Example 2.3 extracts only the first 10 articles in each category. Here, the rank

```
<news_by_category>
{FOR $c IN document("newsmeta.xml")//category
RETURN
  <category>
    <name> {$c/name/text()} </name>
    {FOR $a IN document("news.xml")
      //article[@cid = $c/@id]
      RETURN
        <title aid= {$a/@id}> {$a/title/text() } </title>
      RANKBY ({$c/keywords}) LIMIT 10
    </category>
  </category>
</news_by_category>
```

king occurs in the inner loop based on the actual relevance for the category an article is classified under.

### 4 Implementation Aspects

Weighting algorithms underlying the new operator have some novel properties. Furthermore, \( \text{rank} \) has some implications for a system implementation, in particular index structures and query optimization. In this section, we briefly address both issues.

#### 4.1 Dynamic Ranking

In standard IR, queries are always evaluated in the context of the complete document collection, a notion that we have replaced by the concept of DFS. DFSs are not static anymore but dynamically established by a (sub-)query. Nothing prevents us from still using static, pre-computed term weights. However, most of the power of the integrated query approach with its local contexts is lost in that case.

For instance, it is unsatisfactory to always use the same, fixed inverse document frequency for terms when using an adapted \( \text{tf-idf} \) scheme, because the sub-query result to which the weighting is applied might not even once contain the respective term. This will lead to unexpected and even useless ranking results. Consequently, more general term counting statistics have to be kept. For each ranking operation, term statistics for the current intermediate result have to be derived. Then, a scheme like, e.g., \( \text{tf-idf} \) could be used as before, or an extension thereof that utilizes the additional information encoded in the structure of DFs.
We call this general principle underlying IR embedded into a data retrieval query language dynamic ranking principle. An approach following this principle allows us to detect the local significance of an otherwise rather common term, considered as irrelevant in a more global context.

4.2 Indexes and Query Optimization

For an IR index to be useful for hierarchically structured XML data, it needs to capture term occurrences within single document fragments. This is obviously more expensive than a static index on a collection of documents as single units. However, by definition only leaf nodes contain content. Thus, storing term statistics for these and relying on an index for the document structure as required for structure-based queries anyway can be sufficient to derive statistics for arbitrary DFs.

Furthermore, the query engine needs to be extended to keep track of where parts of DFs within an intermediate result originate from. This is made even more difficult as DFs from different contexts may be put together through XQuery’s construction operations. In some cases, e.g., in case of aggregated data, no direct relationships between DFs in a DFS and their origin can be maintained. In this case, either ranking is not possible or term statistics have to be collected from scratch, which might be acceptable for small results.

The expenses associated with a full text index on top of XML data can be compensated by an IR index-aware query engine, an advantage that is ignored when deploying IR outside of a database. For semistructured data without a lot of schema information, optimization via the IR index is especially promising [10]. Not only point term indices directly to the data, but detailed data statistics, e.g., about the variance of certain element values are easily integrated into an IR index.

5 Conclusions and Future Work

We have introduced a new operator into XQuery that naturally extends XML queries by information retrieval capabilities. XML document fragment sequences are intermediate and end results in XQuery. We have identified arbitrary document fragments within such dynamically selected sequences as suitable replacement for the notions of document (collection) in today’s information retrieval. The extended language, which we have dubbed XQuery/IR, is conceptually simple yet more expressive than the sum of its parts. We have identified dynamic ranking, the usage of term statistics in the local context established by a query, as the most important property of an underlying IR algorithm to achieve this expressiveness.

Currently, we are implementing a system able to demonstrate important properties of the presented approach. A hindrance is the non-existence of freely available query engines and large, deeply nested XML databases with diverse types of textual information. Other aspects we are investigating include reformulation and thus, optimization rules for queries involving the rank operator. Questions are, for instance, to what extent ranking and data extraction are commutative.

References

Abstract

As XML becomes a standard for data representation on the internet, there is a growing interest in storing XML using relational database technology. To date, none of these techniques have considered the semantics of XML as expressed by keys and foreign keys. In this paper, we present a storage mapping which preserves not only the content and structure of XML data, but also its semantics.

1 Introduction

Over the past several years, there has been a tremendous interest in using relational databases to store XML documents [1, 2, 3, 4, 5], thus leveraging a well-developed technology for data management and query processing. There have also recently been several proposals for capturing constraints beyond DTDs, in particular keys [6] and foreign keys. Aspects of these proposals are finding their way into XMLSchema [7] where, following the older notions of ID and IDREF, foreign keys are termed “keyrefs”. The question naturally arises as to how to capture XML key and keyref constraints in the relational mapping.

For example, consider a community of biomedial researchers who are performing gene expression experiments and exchange data using the XML standard MAGE-ML [8]. This standard includes a specification of keys and keyrefs, and each group is expected to produce data that is correct with respect to these keys. Upon the recommendation of their bioinformatic experts, they use relational database technology to store their experimental data (e.g. the Stanford Microarray Database [9], which uses Oracle). Since the data is already stored in a relational database, they wish to ensure that the data in relational form is correct with respect to the XML keys using relational technology.

Key and foreign key constraints can always be expressed as queries which produce an empty result when evaluated against a correct instance and a non-empty result otherwise. Thus XML constraints can always be checked using triggered procedures in a relational database. However, triggers are very inefficient compared to key and foreign key constraints in relational databases [10]. To fully leverage database technology for constraint checking, we therefore wish to map XML key and keyref constraints to relational key and foreign key constraints.

In this paper, we address the problem of mapping an XML document together with its constraints into a relational schema so as to check XML key and keyref constraints using key and foreign key constraints. The mapping should preserve three kind of information: 1) the content of the document, i.e. each node of the document should appear in the target database; 2) the structure of the document, i.e. the parent-child relationship of the nodes; and 3) the semantics of the document as captured by XML key and keyref constraints. We will call such a mapping an information preserving mapping, and focus on mappings in which the schema is available.

This work makes the following contributions:

1. We distinguish three categories of information which need to be preserved in an information lossless mapping.
2. We propose a notion of constraint relations which explicitly capture XML key and keyref constraints.
3. We present a mapping from an XML document to a relational instance which extends constraint relations to capture the complete content and structure of the document.
4. XML key/keyref constraints can be checked using key and foreign key constraints in the constraint relations.

The rest of the paper is organized as follows: A definition of XML keys and keyrefs is given in Section 2. After introducing constraint relations in Section 3, we give an information preserving storage mapping algorithm called X2R. We close by reviewing related work and discussing future directions.

\[\text{Note that we are not considering other structure information such as cardinality constraints.}\]
Figure 1: Sample XML document

2 Constraints: Keys and keyrefs

Before giving a formal definition of keys for XML, consider the sample document "restaurants.xml" shown in Figure 1. The document contains information about restaurants and reviews of restaurants by cities. Within a city, the restaurants are grouped by their cuisine. As for the semantics of the document, one might wish to assert that the context in which the key must hold; 2) a set on which we are defining a key; and 3) the values which distinguish each element of the set. Since we are working with hierarchical data, specifying the context, set and values involve path expressions. In what follows, we adopt the syntax of [6] for keys and use the following notation: \( n[P_i] \) denotes the set of node identifiers in the XML tree representation of the document that can be reached by following the path expression \( P \) from the node with identifier \( n \). We also use \( [P_i] \) as an abbreviation for \( r[P_i] \), where \( r \) is the root node.

An XML key can be written as

\[
K : (C^K, (T^K, \{P^K_1, \ldots, P^K_p\}))
\]

where \( K \) is the name of the key, path expressions \( C^K \) is called the context path, \( T^K \) the target path, and \( P^K_1, \ldots, P^K_p \) the key paths for the key. For a context node \( c \in [C^K] \), relative to a target node \( t \in [T^K] \), key path \( P^K_i(i = 1, \ldots, p) \) must identify a single key node (either an element or attribute) whose value must be of a simple type. That is, following XMLSchema we require that each \( P^K_i \) exist and be unique (strong keys, [6]); furthermore, the value at the end of a key path must be of a simple type (rather than an XML tree value as in [6]). The idea is that within the scope of a context node, the key constraint must hold on the set of target nodes.

Definition 2.1: An XML tree \( T \) is said to satisfy a key \( K : (C^K, (T^K, \{P^K_1, \ldots, P^K_p\})) \) if and only if \( \forall c \in [C^K], \forall t_1, t_2 \in [T^K], t_1[P^K_j] = t_2[P^K_j] \Rightarrow t_1 = t_2. \]

In relational databases, a foreign key allows a list of attributes in one table to reference a list of attributes in another table (which must form the key for the referred table). Since XML is hierarchical, we must additionally specify the context in which the references are made. Our notation for a keyref is therefore similar to that of a key:

\[
R : (C^R, (T^R, \{P^R_1, \ldots, P^R_p\})) \text{ KEYREF } K
\]

where \( K \) is the name of the key being referenced, \( R \) is the name of the keyref, \( C^R \) is the context path, \( T^R \) is the target path, and \( P^R_1, \ldots, P^R_p \) are the key paths for the keyref. The concatenation of the context and target paths locates the referencing node. As in the relational case, each key path \( P^K_j \) must match the key path \( P^R_j \) of key \( K \), \( j = 1, \ldots, p \), i.e., although the key path expressions may be different they must have compatible data types. Following XMLSchema, the referencing node and referenced node must be within the same context node, thus the path expression \( C^R \) must be the same as \( C^K \).

Definition 2.2: An XML tree \( T \) is said to satisfy a keyref \( R : (C, (T^R, \{P^R_1, \ldots, P^R_p\})) \text{ KEYREF } K \) if and only if \( \forall c \in [C], \forall t_i \in [T^K], \exists \bar{t} \in [T^K], \bar{t}[P^K_i] = t_i[P^K_i] \Rightarrow t_i = \bar{t}. \]

Adopting XPath notation for paths, let “\( f \)” denote the root or be used to concatenate two path expressions, “\( . \)”
denote the current context, “/” match a sequence of labels, and @ be an attribute name. All paths must end in either a single label or a disjunction of labels. The keys and keyrefs for our sample XML document can be written as:

\[ K_0 : \{ /city, \{ /name, /state \} \} \]

A city is identified by its name and state.

\[ K_1 : \{ /city, \{ /restaurant, \{ /name \} \} \} \]

Within the context of a city, each restaurant is uniquely identified by its name.

\[ K_2 : \{ /restaurant, \{ /appetizer|entree|salad|dessert, \{ /name \} \} \} \]

Within a restaurant, an appetizer, entree, salad or dessert is identified by its name.

\[ R_0 : \{ /city, \{ /reviews|review, \{ /restaurant \} \} \} \]

KEYREF \( K_1 \)

Within a city, each review refers to a restaurant in that city by its name.

Before moving on to storage mapping, we comment on the restriction that the context node of the node being referenced must be the same node as the context node of the referencing node. Although this is done in XMLSchema to simplify the referencing scheme, it is not necessary. In [6], the authors defined a notion of “transitive keys”, which guarantees that the context node of any key can itself be identified by some key value. In this case, the target node of any key can be identified by some path of key values from the root. Using this, the notion of foreign key can be generalized so that the context nodes may differ.

3 An Information Preserving Mapping

We now present an algorithm which, given the schema of an XML document, generates an information preserving relational schema. The XML schema consists of information about the structure of the document (i.e. the DTD) and XML key and keyref information as described in the previous section. The relational schema generated consists of a collection of relations together with their key and foreign key constraints.

3.1 Constraint preservation via constraint relations

At the heart of the schema-generation technique is a set of relations corresponding to the given XML key and keyref constraints. To capture the internal identifier for a node \( n \) in the document, we use the notation \( \text{nodeid}(n) \). We use \( \text{text()} \) to grab the value of an attribute or simple element text.

For each XML key \( K : (C^K, T^K, \{ P^K_1, \ldots, P^K_p \}) \), we create a key relation \( KR(\text{tid}, \text{cid}, P_1, \ldots, P_p) \), where for every \( c \in C^K \) and \( t \in T^K \), a tuple \( (\text{nodeid}(t), \text{nodeid}(c), t/P^K_1.text(), \ldots, t/P^K_p.text()) \) is inserted into \( KR \). We also assert the following functional dependency (key) in \( KR : \text{cid}, P_1, \ldots, P_p \rightarrow \text{tid} \). Each target node is in a single context and hence occurs exactly once in \( KR \).

There are therefore two keys for \( KR : (\text{cid}, P_1, \ldots, P_p) \) and (tid).

Similarly, for each XML keyref \( R : (C^R, T^R, \{ P^R_1, \ldots, P^R_p \}) \) KEYREF \( K \) we create a keyref relation \( RR : (\text{tid}, \text{cid}, P_1, \ldots, P_p) \), where for every \( c \in C^K \) and \( t \in T^K \), a tuple \( (\text{nodeid}(t), \text{nodeid}(c), t/P^K_1.text(), \ldots, t/P^K_p.text()) \) is inserted into \( RR \). We also assert the foreign key \( (\text{cid}, P_1, \ldots, P_p) \) REFERENCES \( KR(\text{cid}, P_1, \ldots, P_p) \). As with the key relation, (tid) is a key for \( RR \). Both the key and key reference mappings rely crucially on the fact that each key path terminates in an attribute or simple element (text). We will refer to these key and key ref relations as constraint relations.

Note that since we insert a tuple in \( KR(\text{RR}) \) for every target node of every context node, the mapping from the XML document to these constraint relations is complete.

Proposition 3.1: An XML document satisfies the XML key \( K : (C^K, T^K, \{ P^K_1, \ldots, P^K_p \}) \) if and only if the corresponding key relation \( KR(\text{tid}, \text{cid}, P_1, \ldots, P_p) \) satisfies its key \( (\text{cid}, P_1, \ldots, P_p) \).

Sketch of proof: By construction, there is a one to one correspondence between tuples in \( KR \) and matches for the context node, target node, and key path values of \( K \) in the XML document. Since the functional dependency (key) in \( KR \) is equivalent to the following assertion:

\[
\forall t_1, t_2 \in KR, \quad t_1.\text{cid} = t_2.\text{cid} \land t_1.P_1 = t_2.P_1 \land \ldots \land t_1.P_p = t_2.P_p
\]

a violation of the relational key implies a violation of the XML key and vice versa.

Proposition 3.2: An XML document satisfies the XML keyref

\( R : (C^R, T^R, \{ P^R_1, \ldots, P^R_p \}) \) KEYREF \( K \) if and only if the corresponding key relation \( RR(\text{tid}, \text{cid}, P_1, \ldots, P_p) \) satisfies its foreign key \( (\text{cid}, P_1, \ldots, P_p) \) REFERENCES \( KR(\text{cid}, P_1, \ldots, P_p) \).

Sketch of proof: By construction, there is a one to one correspondence between tuples in \( RR \) and matches for the context node, target node, and key path values of \( R \) in the XML document. The foreign key constraint in \( RR \) is equivalent to the following assertion:

\[
\forall t_1 \in RR, \exists t_2 \in KR. \quad t_1.\text{cid} = t_2.\text{cid} \land t_1.P_1 = t_2.P_1 \land \ldots \land t_1.P_p = t_2.P_p
\]

where \( KR \) is the key relation formed from the XML key \( K \). Since the referring and referred nodes are by definition within the same context, \( t_1.\text{cid} = t_2.\text{cid} \) is always true, and a violation of the relational foreign key constraint implies a violation of the XML key reference constraint and vice versa.

It is therefore enough to check key and foreign key constraints in the relational instance to guarantee that the key and keyref constraints are satisfied in the source XML.

Returning to our example of the previous section, the constraint relations to be created are:

1. \text{city}(\text{cid}, \text{name}, \text{state}) for \( K_0 \), with (\text{name},\text{state}) and (\text{cid}) as keys.
The attribute cid stores the identifier of a node tagged with city, and attributes name and state store the value of its name and state child elements, respectively. Since the context node of the key is the root of the document it is omitted in the relation.

2. restaurant(rid,cid,name) is created for K1, with (cid, name) and (rid) as keys.

3. menuitem(iid,rid, d|e|d.name, type) is created for K2, with keys (rid, d|e|d.name) and (iid). Attribute type can have value “appetizer”, “entree”, “salad” or “dessert”.

4. review(rvid, cid, @restaurant) is created for R1, with key (rvid) and foreign key (cid, @restaurant) REFERENCES restaurant(cid,name).

Before we leave the issue of constraint preservation, it is important to consider the effect of updates to the source XML document. Following [12, 10], we assume two forms of updates, insert(content) and delete(child).4 We also assume that the path from the root to the update point is either specified or recoverable (e.g., using a parent relation).

Thus an insertion can be considered as an XML document with the same root as the original document in which the content to be inserted is marked as “new” and the existing content (i.e. the path from the root to the update point) is marked “old”. Using the mapping to constraint relations described earlier, a tuple produced from a node marked “new” causes an insertion. Tuples produced from nodes marked “old” are ignored.

Assuming that the content of the subtree rooted at child is either specified or recoverable (for example, through a content and structure preserving mapping of the document), an analogous technique can be used for deletion: a tuple produced from a node marked “new” causes a deletion, and tuples produced from nodes marked “old” are ignored.

In this way, every insertion (deletion) can be mapped to a set of inserts (deletes) to the constraint relations via the mappings described earlier to populate the constraint relations. Note that since a single XML update may correspond to several tuples in the target, transactions must be used to prevent anomalies.

Let δ be the content to be inserted or the contents of the subtree to be deleted, “+” be the effect of inserting or deleting content, Σ be the XML constraints, Σ′ be the key and foreign key constraints on the constraint relations, and M be the mapping which populates the constraint relations from the XML input.

Proposition 3.3: I + δ satisfies Σ if and only if M(I) + M(δ) satisfies Σ′.

Sketch of proof: According to Propositions 3.1 and 3.2, I + δ satisfies Σ if and only if M(I + δ) satisfies Σ′. By the definition of M, M(I + δ) = M(I) + M(δ). ■

1. Create constraint relations. If any target path ends with a disjunction of tags, then add a new attribute in the relation to record the tag of the target node.

2. Create a DTD graph that represents the structure of a given XML-Schema.

3. For each XML key or key reference constraint, mark the edges at the end of the target path in the schema graph.

4. If a target path ends with a single tag (as opposed to a disjunction), then inline any attribute or non-empty text descendant connected by non-star path as well as the content of the target node (if it exists).

5. If any target path ends with a disjunction tag, inline the common attributes or non-empty text descendents connected by a non-star path as well as the content of the target node (if it exists).

6. If all incoming edges of a node are marked, the non-star edge connecting this node and the inlined node (including that on the key path) are marked. Do this recursively until no more edges need to be marked.

7. If there are unmarked non-star edges in the schema graph, find one whose source vertex connects with its parent by a star edge. We name this node a master node. Create a table for the master node and inline any descendant nodes connected by a non-star path. Mark all incoming edges of this master node.

8. Repeat the last two steps until there are no unmarked non-star edge in the schema graph.

9. Ensure that all parent-child relations are recorded by adding a parent id (and parent code) in each child relation to link with its parent.

Figure 2: The X2R storage mapping algorithm

3.2 Content and Structure Preservation

Since only some of the nodes in the source are involved in a key or key reference relation, we need to consider how the rest of the document can be captured to preserve content, structure and semantic information. We can use arbitrary XML storage system together with the constraint relations to preserve them, but this approach brings redundancy in the generated relational schema. In the remainder of this section, we present an algorithm called X2R (see Figure 2) which generates an information preserving mapping without redundancy.

The algorithm extends an inlining strategy [4] as follows: First, we create a DTD graph (as defined in [4]) as well as the constraint relations defined in the previous subsection. We then map the nodes not captured in the constraint mapping, either by inlining them into the constraint relations or by creating separate relations. Finally, we add any missing parent-child information in the constructed relations.

The salient differences between the X2R algorithm and hybrid-inlining and its variants [4, 5] are as follows: 1) We
start from a set of key and reference relations which capture semantic information; 2) Relations that are separated in hybrid inlining may be coalesced by paths that end with a disjunction; and 3) The key and reference relations may inline ancestors other than the parent, i.e. the context node may be a non-parent ancestor.

**Proposition 3.4:** The $X^2R$ mapping from the source XML schema $X$ to $R$ is information lossless.

**Sketch of proof:** Every node in the source XML document is mapped into some relation (content preservation), the structure information is captured either explicitly in $pid$ attributes or via the DTD (inlining), and the semantics is captured in the constraint relations (proposition 3.1 and 3.2).

### 3.3 An example

We now illustrate how the $X^2R$ mapping algorithm works on our example. The DTD graph assumed is presented in Figure 3 (see the Appendix for the XML schema specification) and the final relations created are as follows:

- **city(cid, name, state)** with keys (name, state) and (cid).
- **cuisine(csid, type, pid)** with key (csid) and foreign key (pid) REFERENCES city(cid).
- **restaurant(rid, cid, name, pid)** with keys (cid, name) and (rid), and foreign key (pid) REFERENCES cuisine(csid).
- **menuitem(iid, rid, d.name, price, type)** with keys (rid, d.name) and (iid), and foreign key (rid) REFERENCES restaurant(rid).
- **spicy(eid, spicy)** with key (eid) and foreign key (eid) REFERENCES menuitem(iid).
- **review(rvid, cid, @restaurant, TEXT, pid)** with key (rvid), and foreign keys (cid, @restaurant) REFERENCES restaurant(cid, name) and (pid) REFERENCES city(cid).

In step 1, we build constraint relations city(cid, name, state), restaurant(rid, cid, name), menuitem(iid,rid, d.name,type), and review(rvid, cid, @restaurant). In step 4 we inline the TEXT information into the review relation, and in step 5 we inline the price information into table menuitem. Relation cuisine(csid,type, pid) and relation spicy(eid, spicy) are created in step 7. All the pid information is inlined in step 9. Note that in some relations (e.g. table menuitem) the parent node coincides with the context node. Details of this construction are deferred to the Appendix.

### 4 Related works and conclusions

This paper proposes a novel XML storage strategy using relational databases which preserves the content, structure and semantic information as expressed in key and foreign key constraints. In contrast, other strategies that have been proposed [1, 2, 3, 4, 5] only preserve the content and structure of XML documents. Our storage mapping is guided by the XML key and keyref constraints, and is based on the notion of constraint relations. Although the storage strategy in this paper is integrated with hybrid-inlining, constraint relations can be combined with virtually any other strategy. A direct benefit of our storage mapping is the ability to efficiently check XML constraints using relational key and foreign key constraints; furthermore, the technique is incremental. In some ways, our approach is analogous to the initial stage of database design based on functional dependencies. In future work, we plan to consider how the conceptual schema design can be refined according to the query workload.

LegoDB [5] considers the issue of finding an optimal relational schema in a space of storage mappings according to the query workload for some cost model. The procedure is analogous to the tuning procedure for physical database design in the relational database [13]. Specifically, it tunes the conceptual schema according to the query and update workload to achieve better performance. In this paper we produce a relational mapping which is optimized for updates (enforcing the constraints efficiently), but do not consider queries. Considering queries would obviously affect the design: For example, if finding the price and spiciness of an entree is a frequent query we may wish to extend the schema of menuitem to include attribute spicy; the attribute is null for appetizers, salads and dessert.

We classify nodes by considering both their tag and their keys (if any), while [1, 2, 3, 4, 5] only consider the former. For example, in our sample XML file, we can define a parent type menuitem for types appetizer,entree,salad and dessert based on keys and then store type menuitem into one table. In other words, we use the semantic information in keys to guide the schema design.
We do not address the issue of mapping queries on the XML document to the relational representation in this paper. Several general mapping algorithms from XML query languages to SQL [14, 15, 16] have been proposed and could be used for the X2R mapping.

Another related issue not addressed in this paper is how to map constraints into a relational schema that has already been created. For example, in the Clio system [17] the target schema is fixed. In general, it is impossible for a fixed target to preserve all the information in the source, especially the structure and constraints information. [18] begins to address the question of mapping constraints to a fixed target schema by giving an algorithm to answer the following question: Given a set of XML keys \( \Sigma \) and a mapping to a fixed relational schema \( \mathcal{R} \), is a functional dependency \( f \) on \( \mathcal{R} \) provable from \( \Sigma \)? [19, 20] focuses on the more general question of how to use the constraints in the target to optimize constraint checking defined in source.

There are several native XML Schema validators[21, 10], to our knowledge no other work has been done on automatically enforcing XML constraints using relational database technologies.

Acknowledgments. We are grateful to Val Tannen, Wang-Chiew Tan, Byron Choi, Wenfei Fan and Carmem Hara for helpful discussions.

References


[8] MAGE-ML.

http://daisy.stanford.edu/MicroArray/SMD.


Types for Correctness of Queries over Semistructured Data

Dario Colazzo  Giorgio Ghelli  Paolo Manghi  Carlo Sartiani

Dipartimento di Informatica - Università di Pisa, Corso Italia 40, Pisa, Italy
e-mail: {colazzo, ghelli, manghi, sartiani}@di.unipi.it

Abstract

A type system for a query language should serve both purposes of verifying whether a query is coherent with what is known about the structure of the database (query correctness) and of giving information about the type of the query result (result analysis). Current proposals for typed query languages for semistructured data are usually focused on result analysis, but perform very few controls, or none at all, of query correctness.

This work presents a type system for a core of XQuery that supports both query correctness and result analysis, and discusses some of the design issues and alternatives.

1 Introduction

In conventional query languages, a type system analyses query correctness and the result type. Query correctness is generally defined as a relation of compatibility between the type of the query input and the query type. The query type represents the type of the data targeted by the query, and is either inferred from the query structure, or directly provided by the programmer. Correctness ensures, at the very minimum, that the query has some hope to find a match in data that respects the input type.

Result analysis is the process of checking whether a query effectively returns data of an expected output type. This check is performed by inferring the output type of a query and by matching it against the expected type. Both query correctness and result analysis are useful tools for the development of complex database applications, where database queries and high-level programming language applications are usually combined forming a complex web of input and output type dependencies.

In the context of semi-structured data (SSD) and XML only a few query and manipulation languages exploit static type information, given the possibly irregular and unstable nature of the data. Current proposals mainly focus on result analysis [3, 6, 5], but perform very few controls, or none at all, of query correctness. Actually, there is no clear agreement, and neither much discussion, about what query correctness means in this context.

This is due to the fact that SSD, especially XML documents, are usually endowed with rather irregular type descriptions, comprising union types, recursive types, and wildcards; coherently, the corresponding languages include operators such as alternative paths, wildcard matching, collection of descendants.

One possible notion of query correctness, adopted by the XDuce language [4], is the full correspondence between the alternative paths in a query and all the possible cases of the union-type that describes the input. This approach seems, however, too restrictive for many SSD specific programming tasks.

At the other extreme, one may flag as non-correct only those queries that are statically deemed to always return an empty collection. This approach has been suggested by the authors of XQuery [1]. However, unless the system is able to flag the specific parts of the query where the problem arises, this policy becomes quite loose, and not informative enough for programmers.

An intermediate notion of correctness is to deem as wrong all and only those paths in the query that have no hope to match the input data.

In summary, each of these approaches is reasonable in some specific application, hence none of them can be regarded as the general purpose solution.

Our contribution This work describes a type system for μXQuery, an abstract core of XQuery. μXQuery’s type system provides a formal framework where different notions of query correctness can be formalized and compared. Specifically, it supports a notion of conformance of data to a type, of result analysis, and is based on a three-levels definition of query correctness, according to which a query can be classified as:

- incorrect, if the structural requirements of the query will not find a match against any instance of the database schema;
- weakly correct, if the structural requirements of the query will find a match in at least one instance of the database schema.
- strongly correct, if the structural requirements will find a match against all possible instances of the database schema.

We believe this characterisation to be particularly suitable in the context of SSDBs, as it is based on a clear semantic characterization but is flexible enough to be compatible with the needs of different applications.
2 Query correctness in XML query languages

In the absence of input type description, query correctness cannot be checked. As a consequence, programmers may interpret an empty result as being due to a structural requirement failure (from clause) or a selection failure (no data satisfied the where clause).

```xml
<!DOCTYPE people[
<!ELEMENT people person+>
<!ELEMENT person (name, phone)>
<!ELEMENT name (frsname, sndname | firstname, secondname)>
<!ELEMENT frsname PCDATA>
<!ELEMENT sndname PCDATA>
<!ELEMENT firstname PCDATA>
<!ELEMENT secondname PCDATA>
<!ELEMENT phone PCDATA>
</ELEMENT person>
</ELEMENT firstname>
</ELEMENT secondname>
</ELEMENT phone>
]
```

Figure 1: A sample DTD.

When schemas are available, instead, some static controls can in principle be performed. However, the irregular nature of SSD types and query languages makes this aim very elusive. In fact, only the language XDuce defines a standard notion of query correctness, but XDuce is quite far from the standard structure of query languages, being a stricter relative to functional languages as ML.

Other approaches, such as XQuery’s and Suciu’s approach [6], are not concerned with the automatic identification of incorrect queries and concentrate on result analysis. Given a query and a schema for the database at hand, the problem is that of checking whether every output of the query conforms to a given expected output type.

Such solutions target different kinds of application scenarios and therefore differ for a number of design choices. However, if we focus on query correctness, XDuce’s approach turns out to be quite restrictive for a query language, while the other approaches are instead poorly informative for the programmer.

XDuce

XDuce is a typed, functional, Turing complete programming language. It is based on an ML-like pattern language that implements a one-match semantics, i.e., every pattern, instead of collecting every matched piece of data (as in standard query languages), only binds the first match. XDuce is nearer to a programming language than to a query language, but we consider it here since it is the only example of typed language for XML that explicitly provides a notion of type correctness.

For example, consider the following XDuce’s function, which returns the list of phone numbers of all people in the document d, which conforms to the schema in Figure 1:

```xml
fun selNums: person* --> (sndname, phone)* =
    person[name[frsname[String], sndname[n:String]],
        phone[p: String]], rest: person*
    --> sndname[n], phone[p], selNums(rest) |
        () --> ()
```  

(XD1)

The function `selNums` can be applied to the element `people` of d. This function is type correct, because XDuce supports a notion of query correctness according to which functions are correct if and only if they specify a matching pattern (a function case) for all possible patterns described by the input type: exhaustive patterns in function definitions are required to ensure the soundness of the type system of XDuce, as stated in [4]. Indeed, the function,

```xml
fun selNums: person* --> (sndname, phone)* =
    persons[name[frsname[String], sndname[n:String]],
        phone[p: String]], rest: person*
    --> sndname[n], phone[p], selNums(rest) |
        () --> ()
```  

(XD2)

is statically judged incorrect and never executed, because the field `persons` is not defined in the schema. This notion of correctness, however, is too restrictive for XML querying purposes. For instance, the function

```xml
fun selNums: person* --> (sndname, phone)* =
    person[name[frsname[String], sndname[n:String]],
        phone[p: String]], rest: person*
    --> sndname[n], phone[p], selNums(rest) |
        () --> ()
```  

(XD3)

is considered incorrect and never executed, although one would expect the query to be run, as instances of the database exist that are matched by the body of the function.

XQuery

XQuery’s type system infers the output type of a query by matching the structural requirements of the query (query type) with the type of the query input [3]. In doing this, the type system does not identify and discard incorrect queries, but simply assigns an empty type to those subparts of the query that cannot find a match in any instance of the data. Coherently, a query over an instance of a union type is assigned an empty type only when none of the members of the union type is relevant to the query. Given the expected type of the query’s output data and the inferred output type of the query, the system can statically detect if the query’s output value has the expected output type.

The function (XD1) can be encoded in the following XQuery’s query:

```
This policy is looser than XDuce’s, where a query is accepted only if it searches for all the members of a union type, and more suitable for navigating arbitrarily irregular SSDBs.
for $p$ in d/person,
  $n$ in op:union($p/name/sndname,
  $p/name/secondname),
  $ph$ in $p$/phone
return <sndname> data($n) </sndname>,
  $ph$

(XQ1)

for which the type system statically infers the following output type,

(element sndname {xsd:string},
 element phone {xsd:string})*

Function (XD3) corresponds to the query,

for $p$ in d/person,
  $n$ in $p/name/sndname,
  $ph$ in $p$/phone
return <sndname> data($n) </sndname>,
  $ph$

(XQ3)

The type system infers the same output type of (XQ1).

Result analysis states that both (XQ1) and (XQ3) are correct as their output type matches (is a subtype of) the expected type of the output. The function (XD2) becomes instead,

for $p$ in d/persons,
  $n$ in op:union($p/name/sndname,
  $p/name/secondname),
  $ph$ in $p$/phone
return <sndname> data($n) </sndname>,
  $ph$

(XQ2)

The system pinpoints the error, as the programmer was expecting a different type.

Essentially, XQuery provides programmers with powerful result analysis tools, which are, in some situations, also useful for detecting errors before execution. In particular, the authors observe that, since queries are assigned an empty output type, the program expects the output type as output. This notion of incorrectness, however, is rather incomplete, as many incorrect queries do not necessarily return an empty type. Consider for example the following query:

for $p$ in d/person,
  $n$ in op:union($p/name/sndname,
  $p/name/secondname),
  $ph$ in $p$/phone
return <sndname> data($n) </sndname>,
  $ph,$p/age

(XQ4)

Although the schema of d contains no age field, the type system infers exactly the same output type inferred for the query (XQ1). The same happens with the following query, although the schema of d contains no secondname field.

for $p$ in d/person,
  $n$ in op:union($p/name/sndname,
  $p/name/secondname),
  $ph$ in $p$/phone
return <sndname> data($n) </sndname>,
  $ph$

(XQ5)

Suciu’s proposal Dan Suciu et al. focus on the development of a formal framework for the definition of result analysis tools [6]. They view queries as transformation programs, i.e. applications transforming an original data source into an XML database that conforms to a given type.

They explore a backward type inference mechanism, which takes as inputs the query, the query input type, the expected output type, and checks that every database that is the result of the query applied to an instance of the input type, conforms to the output type. Again, the methodology fully addresses the issues of result analysis, but totally disregards a notion of query correctness.

3 μXQuery

μXQuery is an abstract version of the FLWR core of XQuery. The main difference between μXQuery and XQuery is the lack of support for function definitions, and for if − then − else and typeswitch expression. Moreover, μXQuery features only copy-semantics return clauses, hence discarding reference-semantics element construction.

The novelty of μXQuery’s type system is that it has been specifically designed for both result analysis and query correctness checking. The type system infers the output type of a query, as in XQuery, but makes a distinction between correct and incorrect queries, as in XDuce. In particular, it supports a three-levels definition of correct queries, distinguishing between weakly correct and strongly correct queries. We shall briefly discuss the advantages of this approach.

3.1 Grammar

Queries are defined by the following grammar:

\[
Q ::= ( ) \mid v_B \mid (l) Q \langle /l \rangle \mid Q.Q \mid x \mid Q p \\
\mid \text{let} \ x := Q \ \text{return} \ Q \mid \text{for} \ x \ \text{in} \ Q \ \text{return} \ Q
\]

Data are represented as ordered forests (f) of trees (t), as defined by the following grammar:

\[
f :: ( ) \mid t \mid f.f \quad t ::= v_B \mid (l) f \langle /l \rangle
\]
where $v_B$ is a leaf value of type $B$, ‘,$ is associative, and ($), f = f, ($) = f.$

Paths are defined by the following grammar:

\[
p \ ::= \ nil \mid /ls \mid //ls \mid p \ p \mid p + p' \\
ls \ ::= l \mid * 
\]

### 3.2 Query semantics

A query $Q$ is evaluated according to an environment $\rho$ which associates a forest with each free variable occurring in $Q$, and the result is denoted by $[Q]_\rho$. Informally, $[Q]_\rho$ yields the pair $\langle f, S \rangle$, where $f$ is the forest returned by $Q$ with respect to the substitution $\rho$, and $S$ is a status variable that captures a notion of correct execution (formal details can be found in [2]). $S$ ranges over the set $\{C, F\}$, respectively representing the correct or faulty status of execution. Specifically, $\langle f, C \rangle$ states that $f$ is correctly returned by a query, while $\langle f, F \rangle$ states that $f$ is faultily returned by a query.

In detail, a query $Q$ correctly returns a forest $f$ in an environment $\rho$ $([Q]_\rho = \langle f, C \rangle)$, if, for all path selections $Q'p$ in $Q$, the path $p$ finds a match with the forest returned by $Q'$. In particular, for path selections of the form $Q'(p_1 + p_2)p$ it is only required that either $p_1p$ or $p_2p$ finds a match in the forest returned by $Q'$. A query $Q$ faultily returns a forest $f$ in an environment $\rho$ $([Q]_\rho = \langle f, F \rangle)$, if there exists a path selection $Q'p$ in $Q$ for which either $Q'$ faultily returns a forest $f'$ or $p$ cannot find a match in $f'$.

Because of union types, two well-typed input values may exist such that the same query may correctly return a result on one and faultily return a result on the other one. For this reason, we adopt a three-levels classification of the semantic correctness of a query with respect to an input type: strongly correct if it correctly returns a forest for any well-typed input, weakly correct if it correctly returns a forest for some well-typed input, incorrect if it faultily returns a forest for any well-typed input (Definition 3.1).

Consider for example the databases $d_1$ and $d_2$ in Figure 2, which conform to the schema in Figure 1. The query $(XQ4)$ on $d_1$ faultily returns the forest:

\[
\langle \text{name} \rangle \text{Sartiani} \langle \text{phone} \rangle 123456 \langle /\text{phone} \rangle
\]
as the path selection $(\$\text{phone})$ does not match the data; for the same reason, $(XQ4)$ execution would be faulty over $d_2$ or any $d$ that conforms to the same schema. The same is true for the query $(XQ2)$, which faultily returns the empty forest because the $/\text{persons}$ path will never match the data. These queries are incorrect.

On the other side, a query $d/\text{person}/\text{name}/\text{secondname}$ would faultily return an empty forest when applied to $d_1$, since it finds no match, but would correctly return a result when applied to $d_2$. Hence this query is weakly correct.

For path selections $Q(p_1 + p_2)p$, our notion of correct execution is rather permissive, in the sense that, as we already said, matching is required for at least one alternative. The query $(XQ1)$, for example, is correctly executed over $d_1$, and returns the same forest $f$ above with $S = C$, since one of its alternative paths finds a match in $d$. $(XQ1)$ is actually strongly correct, since, for any well-typed content of the database, at least one of its alternative paths finds a match. The query $(XQ5)$, when applied to $d_1$, returns the same forest as $(XQ1)$, thanks to the disjunct $/\text{name}/\text{secondname}$, hence this execution is correct, despite the presence of the wrong path $/\text{name}/\text{seconname}$. However, $(XQ5)$ is actually weakly correct since, over $d_2$, its execution is faulty.

Hence, our semantics defines a three-level notion of the correctness of a query with respect to an input type. Now, our aim is to define a type system that is able to infer, for each query, a reasonable approximation of its semantic correctness with respect to a given input type.

### 3.3 Type system

In this Section we introduce $\mu$XQuery’s type system. We first introduce the syntax of types, then give an informal characterisation of the semantics of types, and give a characterisation of type correctness in terms of the semantics of queries. Formal definitions, as well as type rules, can be found in [2].

#### 3.3.1 Type language

The type language we consider is essentially XDuce’s type language, and is defined by the following grammar.

\[
T ::= () \mid B \mid T , U \mid T + U \mid l[T] \mid X
\]

where $B$ represents atomic types. The empty type $( )$ only contains the empty forest $( )$. The type constructor $l[T]$ represents the set of trees rooted as $l$ and containing a forest of type $T$. Concatenation $T , U$ represents the set of forests $f, f'$, where $f$ and $f'$ are forests in $T$ and $U$ respectively. The untagged union type constructor $T + U$ represents the set of forests $f$ which belong to either $T$ or $U$.

Type variables are defined by an environment $E$, which consists of a set of potentially mutual recursive type definitions of the following form:

\[
d_1 = <\text{people}><\text{person}>
\langle \text{name} \rangle \\
\langle \text{fname} \rangle \text{Carlo} \langle /\text{fname} \rangle \\
\langle \text{name} \rangle \text{Sartiani} \langle \text{name} \rangle \\
\langle /\text{name} \rangle 123456 \langle /\text{name} \rangle
\]

\[
d_2 = <\text{people}><\text{person}>
\langle \text{name} \rangle \\
\langle \text{fname} \rangle \text{Dario} \langle /\text{fname} \rangle \\
\langle \text{name2} \rangle \text{Colazzo} \langle /\text{name2} \rangle \\
\langle /\text{name2} \rangle 654321 \langle /\text{name2} \rangle
\]

Figure 2: Two db’s conforming to the DTD in Figure 1.
\[ E ::= ( ) \quad \text{empty environment} \]
\[ X = T, E \quad \text{type variable definition} \]
\[ x : T, E \quad \text{query variable declaration} \]

Note that environments also contain query variable type declarations \( x : T. \) These are used in the typing rules given in [2]. Moreover, observe that regular expressions types, such as repetition and optional types, can be defined by combining recursive and union types as follows:

\[ T* \equiv X \text{ with } X = ( ) + (T, X) \quad T? \equiv ( ) + T \]

For instance, the DTD given in Figure 1 corresponds to the following \(\mu\)XQuery’s type environment,

\[
\begin{align*}
\text{PEOPLE} & = \text{people[PERSON *]} \\
\text{PERSON} & = \text{person[NNAME, PHONE]} \\
\text{NNAME} & = \text{name[(FIRSTNAME, Surname) * (FIRSTNAME, SECONDNAME)]} \\
\text{FIRSTNAME} & = \text{firstname[String]} \\
\text{SECONDNAME} & = \text{secondname[String]} \\
\text{PHONE} & = \text{phone[String]} \\
\end{align*}
\]

While query correctness in XDue and XQuery is based on subtyping, in \(\mu\)XQuery it is based on a relation of coherence between query paths and query input types.

### 3.3.2 Semantics of types

We interpret a type as the set of all forests that have that type. In the style of [4], we define the semantics of types by means of a set of deduction rules over judgements of the form \(E \vdash f : T\), which state that \(f\) conforms to \(T\) with respect to \(E\). Informally, we write

\[ [T]_E = \{ f \mid E \vdash f : T \}. \]

#### 3.3.3 Query correctness

To define query correctness in \(\mu\)XQuery, we denote as \([Q]_E\) the set of all possible results, i.e., pairs \((f, S)\), returned by \(Q\), for each assignment to \(Q\)’s variables that respects the type definitions in \(E\).

**Definition 3.1 (Correctness)** Given a query \(Q\) and an environment \(E\) of type definitions for free variables in \(Q\), we say that \(Q\) is

- **strongly correct** \(\) if \(\forall (f, S) \in [Q]_E \cdot S = C\)
- **weakly correct** \(\) if \(\exists (f, S) \in [Q]_E \cdot S = C\)
- **incorrect** \(\) if \(\forall (f, S) \in [Q]_E \cdot S = F\)

In [2] we have defined a set of algorithmic rules which reflect this characterisation, by returning the inferred correctness of a query (the relationship between the correctness as inferred by the type rules and the actual correctness is discussed below). Observe that query correctness is strictly related with type inference, as in the presence of nested queries, correctness of outer queries depends on the type inferred for inner queries. As a consequence, the rules also return the type query thereby enabling result analysis techniques.

Table 1 shows the inferred correctness returned by the rules for path selections \(Q \rightarrow \) given the type inferred for \(Q\) (the symbol \(\_\) denotes any label) and the path \(p\).

<table>
<thead>
<tr>
<th>Path (p)</th>
<th>Type of (Q)</th>
<th>Strong</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (_ + b)</td>
<td>(\Gamma)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2 (_ + a + b)</td>
<td>(\Gamma)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3 (_ + a)</td>
<td>(\Gamma)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>4 (_ + b)</td>
<td>(\Gamma)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>5 (_ + a + b)</td>
<td>(\Gamma)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>6 (_ + b + a)</td>
<td>(\Gamma)</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Inferred correctness.

As illustrated by the rows 1 and 2, \(Q \rightarrow \) is strongly type correct if \(p\) finds a match in each instance of the type of \(Q\). Accordingly, when checking correctness of a path \(p\) with an input union type, the rules state that \(Q \rightarrow \) is strongly correct only if all members of the union type are matched by \(p\) (data covering). However, we do not require here the path covering property, i.e., that every disjunctive member of \(p\) is matched (or may be matched) by a piece of data, hence row 2 is strongly correct as well. Data covering and path covering are, in a sense, independent issues, both of them relevant, though we focus here on the first one only; we will come back to this in Section 4.

When not all members of the type, but at least one, are matched by \(p\), the query is weakly correct, as exemplified in rows 1, 3, 4, and 6. In this case, programmers may exploit this information, deciding either to make their queries strongly correct, by adding the missing alternatives in the path, or run them anyway when they are not interested in matching this information, deciding either to make their queries strongly correct, by adding the missing alternatives in the path, or run them anyway when they are not interested in matching this information, deciding either to make their queries strongly correct, by adding the missing alternatives in the path, or run them anyway when they are not interested in matching this information, deciding either to make their queries strongly correct, by adding the missing alternatives in the path, or run them anyway when they are not interested in

The rules, given a query \(Q\) and an environment \(E\), infer a pair \((\Gamma, A)\), where \(\Gamma\) is the output type of \(Q\) and \(A\) is a variable that ranges over the set \(\{a, w, 1\}\). The correct definition of the rules is far from trivial. Consider the following query over a database of type \(\text{root}[\text{int}] + b[\text{int}]\), bound to the variable \$y, \(\{\$y/a, \$y/b\}\).

The query is semantically incorrect, as \$y\) either matches \(\_a\) or \(\_b\), but cannot match both of them. However, a standard inductive type rule such as,

\(E \vdash Q : a, E \vdash Q' : b \Rightarrow E \vdash Q + Q' : a\)

does not suffice for the example above, as both \$y/a\) and \$y/b\) are weakly correct with respect to the type of \$y\).

Similarly, but more subtly, the following query

\(\{\$y/a, \$y/b\}\)
where $y$ is of type $\text{root}[X]$ with $X = a[X] + b[X] + \text{int}$, is semantically incorrect. Indeed, the product query $\text{sy}/a/a$, $\text{sy}/a/b$ is incorrect as each level of a unary tree either contains a label $a$ or a label $b$.

In [2] we give a solution to these problems by means of complex algorithmic type rules, which keep track of which members of union types are matched by the paths in the query, and return a correctness status which depends on this information.

The following proposition claims soundness of the type rules with respect to the characterisation of query correctness, although the rules are not complete (a strongly correct query may be flagged as $\omega$).

**Proposition 3.2 (Soundness)** For each query $Q$ and environment $E$, if the judgement $E \vdash Q : (T; A)$ holds, then,

- if $A = s$ then $Q$ is strongly correct;
- if $A = w$ then $Q$ is weakly or strongly correct;
- if $A = i$ then $Q$ is incorrect;

Observe that the type rules always infer an output type. By doing so, in the style of XQuery, the type system also provides programmers with result analysis tools. For example, for queries (XQ1) to (XQ4), our type rules infer the same output types as XQuery’s. However, we are also able to identify (XQ1) as strongly correct, (XQ3) and (XQ5) as weakly correct, and (XQ2) and (XQ4) as incorrect.

### 4 Path covering

To simplify the discussion, imagine, for a moment, that every query is just a sum of paths, and every input type is just a union type. Then, the system we described up to now is geared towards the prevention of problems that arise (informally) because one branch of the input-data union-type is not covered by any path (lack of strong correctness), or even no branch of the union type is covered by any path (lack of weak correctness).

This is already complex enough, but only captures those errors that show up as ‘too few paths in the query’. Errors may also show up as ‘too many paths’, as in rows 2, 4, and 6, in Table 1, or in query (XQ5), where we have paths that are not covered by any branch of the union type.

Typical programming errors, like path misspelling, tend to generate both path and data coverage problems, as in (XQ5) or in row 4, hence suggesting that path coverage may be ignored. On the other side, consider row 6 in Table 1. Here the programmer misspelled an /m into an /o, was expecting a ‘weak correctness’ result, and gets it from the type-checker; hence, the misspelling problem does not show up in the type.

Moreover, the errors generated by a data-covering based analysis can only be reported in terms of type-branches that have not been considered by a subpart of the query, while a path-covering based analysis would pinpoint a wrong path. Going back to line 4 if the table, the data-covering error is ‘branch $m[U]$ in the type is not covered’, while the path-covering one is ‘subpath /n is irrelevant’. The second message helps the programmer better.

For these reasons, path coverage should be considered by a correctness-checking type system. However, while we gave a precise semantic characterization of the correctness of a query with respect to data covering, this is not easy when path covering is considered. As an example, the path expression $(/a+/b+/c),(/a+/b+/c),(/a+/b+/c)$ should be equivalent to its expansion $/a/a/a+/a/a+b+/a/a+c+/b/a/...$. However, the first one should probably be considered wrong only when one of the nine atoms /a is useless, while the second one is suspect as soon as any one of the twenty-seven addends is useless.

In [2] we discuss some concrete notions of path-covering correctness and type rules; here we can only point out that the problem is relevant and difficult.

### 5 Conclusions and future issues

This work presents a type system in which both result analysis and query correctness analysis of XML queries can be conveniently expressed. Specifically, queries can be classified as strongly correct, weakly correct, or incorrect. We have seen that such classification describes an intuitive spectrum of query correctness characterisations.

Our type system, beside being the first to try correctness analysis for XQuery-like languages, also provides a framework in which different notions of correctness can be formally identified and studied, as exemplified in the discussion of Section 4. We are currently working on the design and comparison of such alternative notions.

Finally, we plan to augment the type language with other type operators, such as non-ordered sequences, ID and IDREFs types, so as to study our results in the context of a system closer to XML Schema.

### References


The Query Language TQL

Giovanni Conforti  Giorgio Ghelli
Antonio Albano  Dario Colazzo  Paolo Manghi  Carlo Sartiani
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
May 1, 2002

Abstract
This work presents the query language TQL, a query language for semistructured data, that can be used to query XML files. TQL substitutes the standard path-based pattern-matching mechanism with a logic-based mechanism, where the programmer specifies the properties of the pieces of data she is trying to extract. As a result, TQL queries are more ‘declarative’, or less ‘operational’, than queries in comparable languages. This feature makes some queries easier to express, and should allow the adoption of better optimization techniques. Through a set of examples, we show that the range of queries that can be declaratively expressed in TQL is quite wide. The implementation of TQL binding mechanism requires the adoption of non-standard techniques, and some of its aspects are still open. In this paper we implicitly report about the current status of the implementation by writing all queries using the version of TQL that has been implemented, and that can be freely downloaded from //tql.di.unipi.it/tql.

1 Introduction
The Tree Query Language [1] (TQL) is a query language for tree-shaped semistructured data. The language is based on the set comprehension (match-filter-construct) paradigm, in the tradition of SQL, StruQL, Lorel, Quilt, XQuery (among many others). However, the match-filter operation is expressed in TQL using a variant of the ambient logic [2], a logic defined to describe process structure and behaviour. TQL adopts a subset of that logic, for its ability to describe trees.

TQL logics is used to express the binding (match-filter) part of a query. The same logic can be exploited to describe those properties of the data that are usually expressed through types and constraints. This implies that:

- TQL queries can be exploited in order to check whether a data source has a type, or satisfies a constraint;
- whenever a type or a constraint $C$ is known to hold for a data source, the binder $B$ of TQL queries is equivalent to its refinement $B \wedge C$, which is a legitimate TQL binder, as we will see. This refinement opens the way for new optimizations, or even for the static declaration of an empty result, if the unsatisfiability of $B \wedge C$ can be detected.

We will exemplify later these two properties.

The promise of combining the expression of types, constraints, and queries in just one language, and to use this synergy for optimization and error-checking purposes, is the kernel of the TQL project. But the language is also worth studying for its ability to express complex queries by declaring the properties of what one is looking for, instead of describing a path to arrive there. While most interesting properties, as we will show through examples, are heavily path-based, others involve negation, implication, universal quantification, and are expressed in other languages, such as XQuery, by resorting to external functions or by operational means, which makes optimization and formal reasoning on the queries quite harder. While many programmers are perfectly comfortable with operational-oriented programming and reasoning, others find declarative expression easier, and there is at least a pattern, exemplified in Section 5.3, where the TQL style clearly pays off. In TQL, whenever you are able to describe a property of a specific tag (e.g., $title$ is a key for each $article$), by substituting the constant with a variable you obtain the query that finds all tags with the same property (e.g., find all pairs $x$, $y$ such that tag $x$ is key for $y$).

This feature is reminiscent of prolog-like languages. However, TQL does not share datalog problems with negation, partly because TQL is born with negation, and mostly because we restrict ourselves to a monotone form of recursion.

In the rest of the paper we present the expressive power of TQL and some of the properties we discussed here, through a succession of examples, all tested on the current TQL implementation. In some cases we also present an XQuery equivalent query, for the sake of comparison, and also to clarify our usage of the terms ‘declarative’ and ‘operational’ expression of queries.

The current version of TQL data model is unordered. This makes TQL unusable in document-oriented applications, but this lack of order is very important, in terms of allowed optimization, in database-like applications. Dealing with order is left as a future extension.
2 Related work

Many query languages for semistructured data and XML have been designed in the past years: StruQL, Lorel, XQL, XML-QL, YATL, etc. Building on this research, W3C is designing XQuery [3], a standard query language for XML data, which subsumes many concepts coming from these languages. XQuery (still a work in progress) is a typed, Turing-complete query language that can be used in both XML-enabled database systems and native XML systems.

While TQL and XQuery are based on the same bind-filter-reconstruct paradigm, they differ in many aspects.

First of all, TQL, by design, is based on a logic that can express types, constraints, and queries, and is tailored for formal, and automated, manipulation. On the other side, XQuery is designed as an industrial-strength language, aimed at both database-oriented and document-oriented applications. As a consequence, TQL has a very sharp semantic definition, that can be completely defined in one page of formulae, while XQuery semantics is much more complex. On the other side, XQuery data model supports order and oid-like information, which are not dealt with in the current TQL version.

Second, even though XQuery expressive power is greater than TQL’s (the former is Turing-complete), some queries can be more easily expressed in TQL, thanks to the greater expressive power of the tree-logic with respect to a pure matching mechanism.

Finally, XQuery features powerful vertical navigational facilities, while it lacks corresponding horizontal operators; TQL, instead, makes no difference between horizontal and vertical navigation, hence allowing the user to easily impose horizontal constraints on documents.

3 The Simplest Queries

3.1 The Input Data

We begin with some standard queries, borrowed from the W3C XMP Use Case [4]. These queries operate over the XML document available at //tql.di.unipi.it/bib.xml, which we assume to be bound to the variable $Bib$ in the global environment (the TQL system allows any document on the web to be bound to a variable). The document contains bibliography entries, whose structure is described by the following DTD:

```xml
<!ELEMENT bib (book*)>
<!ELEMENT book (title, (author+ | editor+), publisher, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first)>
<!ELEMENT editor (last, first, affiliation)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
```

The DTD specifies that a book element contains a title, one or more author elements or one or more editor elements, one publisher element and one price element; it also has a year attribute. An author contains a last and a first name elements. An editor element also contains an affiliation. Finally, title, last, first, publisher, and price elements contain string values.

In this paper we present the XML file using its more compact TQL-syntax representation, which looks as follows (the implemented system allows both XML and TQL visual presentations):

```xml
bib[
    book[year[1992]]
    | title[FoundationsDatabases]
    | author[ first[Serge] | last[Abiteboul] ]
    | author[ first[Richard] | last[Hull] ]
    | author[ first[Victor] | last[Vianu] ]
    | publisher[Addison]
    | price[60]
] 
| book[year[1990]]
    | title[SistemiOperativi]
    | author[ first[Piero] | last[Maestrini] ]
    | publisher[McGrawHill]
    | price[38]
] ...
```

In this format, bib[C] stands for an element tagged bib whose content is C, while C1 | C2 is the concatenation of two elements, or, more generally, of two sets of elements. We use this non-XML notation because TQL is born as a language to query semistructured data in general, i.e. unordered forests with labeled nodes, and not just XML. XML is just one way to construct such forests, using tagged elements (and attributes) to build labeled nodes.

3.2 The Formal Presentation of TQL Data Model

More formally, TQL data model is defined by the following syntax and equations. The syntax specifies that a forest is either a leaf, or an empty forest, or a node containing two elements, or, more generally, of two sets of elements. We use this non-XML notation because TQL is born as a language to query semistructured data in general, i.e. unordered forests with labeled nodes, and not just XML. XML is just one way to construct such forests, using tagged elements (and attributes) to build labeled nodes.

TQL forests

```
forest ::= 'tag' | 0 | tag[forest] | forest | forest
```

The formal definition of the data model is completed by the equations that specify that | is commutative and

---

TQL node-labeled forests can be equivalently described as edge-labeled trees, as done in [1].
associative, and $0$ is its neutral element:
\[ t \mid t' = t' \mid t \quad t \mid (t' \mid t'') = (t \mid t') \mid t'' \quad t \mid 0 = t \]

Hereafter we will elide the leaf constructor ‘$\cdot$’, writing $t[d]$ instead of $t['d]',$ unless ambiguity arises; the same abbreviation is supported in the implemented system.

### 3.3 Matching and Binding

The basic TQL query is from $Q = A$ select $Q'$, where $Q$ is the subject (or data source) to be matched against the formula $A$, and $Q'$ is the result expression. The matching of $Q$ and $A$ returns a set of bindings for the variables that are free in $A$. $Q'$ is evaluated once for each of these bindings, and the concatenation of the results of all these evaluations is the query result.

For example, consider the following TQL query, that returns the titles of all books written in 1991, and is evaluated in an environment where $\$Bib$ is bound as specified above.

```tql
from $\$Bib$ |= .bib.[year[1991] And .title[$t$]]
select title[$t$]
```

The formula:

```tql
```

is an ambient logic formula, which should be read as: “there is a path $.bib[ ]$ that reaches a place that matches $.year[1991] And .title[$t$]$, i.e. a place where you find both a path $.year[]$ leading to 1991 and a path $.title[]$ leading to something, that you will call $T$.

The formula $.tag[A]$, read “there exists an element tag whose content satisfies $A$”, is the most useful operator, but is actually defined in terms of three more basic operators: truth $T$, vertical splitting $A' \mid A''$, and element matching $tag[A]$.

The element formula $tag[A]$ only matches a one-element document: while $.t[A]$ matches both forests $t [D]$ and $t [D] \mid t2[D2] \mid \ldots$ (provided that $A$ matches $D$), the formula $t[A]$ only matches the first one. The truth formula $T$ matches every forest. Finally, the formula $A_1 \mid A_2$ matches $D$ iff $D$ is equal, modulo reordering, to $D_1 \mid D_2$, with $A_1$ matching $D_1$. For example, the following pairs match, provided that $\$a$ is bound to Date:

```tql
<table>
<thead>
<tr>
<th>Title of book: Foundations of Databases</th>
<th>Author: Serge Abiteboul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title of book: Interpreters</td>
<td>Author: Vincent Aho</td>
</tr>
</tbody>
</table>
```

The third formula can be read as: there is an author $\$a$ and something else, hence is equivalent to $\$author[\$a]$; the fourth pair matches as well, since the empty forest matches $T$. Hence, $m[A] \mid T$ is equivalent to $m[A]$; this is actually the official definition of the semantics of $m[A]$.

While in this example we matched $\$t$ with a leaf, a TQL variable can be matched against any forest, or against a tag.

For example, the following query returns any tag inside a book whose content matches $.first[Serge] \mid .a.b[A]$ abbreviates $\$a[\$b[A]]$.

```tql
from $\$Bib$ |= .bib.book.$tag.first[Serge]
select SergeTag[$\$tag$]
```

Finally, the following query matches the formula $year[1992] \mid $EveryThingElse against any book, hence it returns, for any book whose year is 1992, everything but the year:

```tql
from $\$Bib$ |= .bib.book[year[1992] | $\$EveryThingElse
select BookOf1992[$\$EveryThingElse$]
```

Since we have two books of 1992, there are two possible bindings for $\$EveryThingElse$, each corresponding to the whole content of a 1992 book without its year subtree; hence the result is:

```tql
BookOf1992[
  title[Foundations of Databases]
  | author[ first[Serge] | last[Abiteboul] ]
  ...
]
BookOf1992[
  title[Interpreters]
  | author[ first[Vincent] | last[Aho] ]
  ...
]
```

Hereafter, as a convention, we use lowercase initials for variables that are bound to tags and uppercase initials for variables that are bound to forests.

### 3.4 Matching and Logic

TQL logic allows the programmer to combine matching and logical operators. For example, the condition in the following query combines the request for the existence of a title field, of a $\$x$ field containing Springer, and of either an author.last or an editor.last path leading to Buneman.

```tql
from $\$Bib$ |=
  .bib.book [ .title[$\$t$]
  And Exists $\$x$.\$x[Springer]
  And (.author.last[Buneman] Or .editor.last[Buneman])
  ]
select title[$\$t$]
```
The pattern \texttt{Exists $\mathbf{x}$. $\mathbf{x}[A]$} is common enough to
deserve the abbreviation $\%[A]$, that we will use hereafter
(see [1] for the exact definition of this abbreviation).

Conjunction, disjunction, and universal quantification
are operators that can be found in many match-based
languages. TQL, however, has the full power of first-order
logic, hence we can express universal quantification and
negation of arbitrary formulas. This will be exemplified
later.

4 Restructuring the Data Source

In TQL syntax, a subquery can appear wherever a forest
expression is expected, as expressed by the following
syntax:

\textbf{TQL Queries}

\[
Q ::= \text{from } Q \text{=} A \text{ select } Q \mid \text{tag} \mid 0 \mid \text{tag}[Q] \mid Q|Q
\]

This freedom of nesting is a feature of most modern
query languages, and is typically exploited to use the nesting
structure of the query in order to describe the nesting
structure of the result. For example, in our data source
there is an entry for each book, containing the list of its
author. We can restructure it to obtain an entry for each
author, containing the list of its books. The structure of the
result can be visualized as follows, where \((A)\)* indicates
an arbitrary repetition of the A structure:

\[
\text{author}[\text{authorname}[\ldots] \mid (\text{book}[\ldots]\ast)\ast]\ast
\]

Observe how this structure is reflected by the structure of
the following query, with a from-select for each *.

\[
\text{from } \text{$Bib$} \text{=} .\text{bib.book.author}\{A\}
\text{select author[authorname\{A\}]
\text{from } \text{$Bib$} \text{=} .\text{bib.book[author\{A\}]
\text{select book\{OtherFields\}]
\]
\]
\]

This query performs a nested loop. For each binding of \{A\} to a different author, it returns a forest
result[author\{A\}] | book[\ldots]|...[book[\ldots]],
where book[\ldots]|...[book[\ldots]] is the result of the inner
query, i.e. it contains one book element for each book
whose author is \{A\}. As in a previous example, we extract,
from the input book, all the fields but the author.

5 Schema-less XML data

As XML documents are not necessarily to come with
a DTD, query languages should provide mechanisms for
querying data regardless of the structure.

Alternatively, when schema information is fundamental
for writing sensible queries, schema inference mechanisms
are very useful. For example, one may be interested in
finding the exact structure of the data, or in finding the
mandatory elements in the data. Property checking tools
may also be useful, so as to prove the validity of given as-
sertions about the data. For instance, checking whether a
certain set of tags is a primary key, or if a tag is mandatory
in a specified path.

TQL provides all these mechanisms by simply combin-
ing \textit{tag variables} (as in [6] and [7]) and ambient logic, as
shown in the following sections.

5.1 Querying in absence of schema

We consider an XML document, bound to $\text{\$Bib2}$ in the
global environment, which is similar to the $\text{\$Bib}$ file, but
features some extra-elements with a title (i.e. article, phd,
etc.), whose labels are not known a priori.

The following query selects the title of all elements,
whatever the label, and wherever they are, that contain
an element whose value is Suciu; the * operator iterates
a path an arbitrary number of times (may be zero); $\%*$
must be read as $(\%\ast)$* and corresponds, roughly, to the
XPath operator $//$. 

\[
\text{bib[\text{from } \text{\$Bib2}|= .\%\ast.\%[\text{Suciu}|\text{\$Rest }]
\text{select } \text{\$B[ Suciu}\{\%\}\text{\$Rest }]
\
\]
\]

This query constructs a Suciu’s personal bibliography doc-
ument, selecting all elements in $\text{\$Bib2}$ where he appears
and inverting the tag with the content. The remaining
information present in the elements involving Suciu are
inserted in the result using the $\text{\$Rest}$ variable.

This query clearly reveals some of the differences be-
tween TQL and XQuery, in which it would be expressed
as follows,

\[
\langle \text{\$bib}\rangle
\text{for } \text{\$b in } \text{\$Bib2}//*,
\text{let } \text{\$xx } := \text{\$b/*,}
\text{for } \text{\$y in } \text{\$xx}
\text{where } \text{\$y/data() } = \text{"Suciu"
return } <\text{xf:name}(\text{\$b})>
\langle\text{Suciu}\rangle
\text{xf:local-name}(\text{\$y})
\langle\text{Suciu}\rangle,
\text{\{ op:except($\text{\%xx}$,$\text{\%y}$)}
\langle\text{xf:name}(\text{\$b})\rangle
\langle\text{\$bib}\rangle
\]

Observe how TQL’s binding mechanism and horizontal
navigation are more declarative than XQuery’s, which
adopts instead operational techniques:

- the definition of each binding to a variable requires a
  corresponding nested loop (\text{for} or \text{let}), while in TQL
all free variables are bound in one single \text{from-select}
clause;
- horizontal constraints are dealt with an external op-
erator \text{op:except}, while in TQL these are expressed
with the logic horizontal navigation operator $\text{|}$.
5.2 Checking Properties

In this section we show how tree logic formulae can be used to express properties of XML data. When a formula \(A\) expresses a property, we can check if \(A\) holds with the query \(\text{from } Q |= A \text{ select success:} \) this query returns the leaf \(\text{success}\) if \(A\) holds over \(Q\), and an empty forest otherwise.

As a first example we consider a query that verifies if the tag \(\text{title}\) is mandatory for \(\text{book}\) elements in the \(\$\text{Bib}\) document.

\[
\text{from } \$\text{Bib } |= \text{bib[Not .book[Not .title[T]]] select title_is_mandatory}
\]

The formula \(\text{Not .book[Not .title[T]]}\) means: it is not the case that there exists a book whose content does not contain any title, i.e. each book contains a title. TQL actually features an operator \(!a[A]\) defined as \(\text{Not a[Not A]}\) which we can directly use, as in the following query. Here \(\text{!book.title[T]}\) is an abbreviation for \(\text{!book[.title[T]]}\), hence means: for every book there is a title.

\[
\text{from } \$\text{Bib } |= \text{bib[ !book.title[T] ] select title_is_mandatory}
\]

The formula \(\text{ta[A]}\) is dual to \(a[A]\) in the same sense as \(\forall x.A\) is dual to \(\exists x.A\), or \(\land\) is dual to \(\lor\). In TQL, every primitive operator has a derived dual; this implies that negation can always be pushed inside any operator, hence you can write any query with no use of negation. Actually, when negation appears in a query, in most cases the TQL optimizer pushes it down to the query leaves (variables, expression of the content of a leaf, comparisons), since negation is quite expensive. This is the reason why, although we claim that unlimited negation is an important feature of TQL, you will see very little explicit use of negation in our examples.

The next query verifies that \(\text{title}\) never appears twice in a field.

\[
\text{from } \$\text{Bib } |= \text{Not bib[.book[ .title[T] | .title[T] ] ] select title_never_appears_twice}
\]

Another interesting property to verify is whether a given tag is a primary key. There are many possible generalizations of the relational notion of key to the semistructured case. The statement below, for example, says that \(\text{title}\) is a mandatory field, and that you cannot find two separate books with the same title (more precisely, with one title in common).

\[
\text{from } \$\text{Bib } |= \\
\text{bib[!book[ .title[T] ]} \\
\text{And foreach } X. \text{ Not ( .book[.title[$X$] | .book.title[$X$] ) ] select each_title_is_key}
\]

Of course, if the system knows that \(\$\text{Bib}\) satisfies \(\text{bib[!book[.title[T]]]}\), this knowledge implies that \(\text{bib[!book[.title[T]] And foreach } X. \text{ Not ( .book.title[$X$] | .book.title[$X$] ) ]}\) is equivalent (over \(\$\text{Bib}\)) to \(\text{bib[foreach } X. \text{ Not ( .book.title[$X$] | .book.title[$X$] )]}\).

We do not comment further on this point, since this kind of optimization is out of the reach of the current implementation of TQL.

Our last query checks that the \(\$\text{Bib}\) element contains only elements labeled \(\text{book}\), by asking that each tag inside the outer \(\text{bib}\) is equal to \(\text{book}\).

\[
\text{from } \$\text{Bib } |= \text{bib[foreach } X. \text{ Not ( .book.title[$X$] | .book.title[$X$] ) ] select only_book_inside_bib}
\]

This query can be rewritten using path operators as follows:

\[
\text{from } \$\text{Bib } |= \text{bib[Not (.Not book[T])]} \\
\text{select only_book_inside_bib}
\]

Here \(\text{Not book}\) is a tag-expression that stands for any tag different from \(\text{book}\). Hence, \(.\text{Not book[T]}\) means: there exists a subelement whose tag is different from \(\text{book}\). Hence, \(\text{Not (.Not book[T])}\) means: there exists no subelement whose tag is different from \(\text{book}\).

5.3 Extracting the Tags That Satisfy a Property

Every query \(Q\) in the previous subsection checks a property \(P\) of a tag \(t\). In all such cases, if we substitute, in \(Q\), \(t\) with a tag variable, we obtain a query that finds the set of all tags that satisfy \(P\).

For example, we can extract all keys of books by taking the query that checks whether \(\text{title}\) is a key, and substituting \(\text{title}\) with \(\$k\), as follows:

\[
\text{from } \$\text{Bib } |= \\
\text{bib[!book[.$.k[T]]} \\
\text{And foreach } X. \text{ Not ( .book.$k[.x] | .book.$k[.x] ) ]} \\
\text{select key[$.k]}
\]

It must be highlighted that this is possible because in TQL we can universally quantify even on a formula with other free variables \(\$k\), in this case). The query evaluation algorithm we exploit to this aim is quite sophisticated, and is described in [5].

Generalisation by simple substitution is not possible in XQuery, where variables inserted to replace tags must at least be bound by an outer \(\text{for}\) clause, thus requiring the redesign of the original query.

A similar generalisation can be performed for the queries that check whether a label is mandatory, or occurs only
once, inside another one. We present below a query that almost produces a DTD for any input XML file (modulo ordering) by extracting all the tags in the file and listing, for each of them, all the labels that must or may appear, and distinguishing among them the ones which may be repeated and the ones which only appear once. While it may look frightening, it has just been obtained by a trivial generalization of the simple queries we presented above.

from $parts |= .%*.$tag[.%[T]]
select $tag[ mandatory_subtags
    {from $parts |=
      Not (.%*.tag[Not .$subtag[T]])
      select $subtag[]
    ]
    | optional_subtags
    {from $parts |=
      .%*.tag[ .$subtag[T]]
      And .%*.tag[not .$subtag[T]]
      select $subtag[]
    ]
    | list_subtags
    {from $parts |=
      .%*.tag[ .$subtag[T] | .$subtag[T]
      ]
      select $subtag[]
    ]
    | non_list_subtags
    {from $parts |=
      .%*.tag[ .$subtag[T]]
      And not .%*.tag[ .$subtag[T] | .$subtag[T]
      ]
      select $subtag[]
    ]

\]

6 Recursion

TQL logic also includes two monotonic recursion operators (\textit{rec} and \textit{maxrec}), very similar to the \(\mu\) and \(\nu\) operators (minimal and maximal fix point) of modal logic. These can be used to interpret the Kleene star operator $\text{path}^*$. The Kleene star operator allows for recursive definition of properties. The expression of queries which involve recursion, negation, or universal quantification, has in TQL a clear declarative nature, while other languages are forced to adopt a more operational approach.

All queries presented in this paper are executable in the prototype version of the TQL evaluator, and can be found in the file \texttt{demo.tql} in the standard distribution. The current version of the prototype works by loading all data into main memory, but is already based on a translation into an intermediate TQL Algebra [5], with logical optimizations carried on both at the source and at the algebraic level. The intermediate algebra works on infinite trees, represented in a finite way, and supports such operations as complement, to deal with negation, co-projection, to deal with universal quantification, several kinds of iterators, to implement the \texttt{|} operator, and a recursion operator.

TQL is currently based on an unordered nested multisets data model. The extension of TQL’s data model with ordering is an important open issue.

7 Conclusions

Although the language TQL originates from the study of a logic for mobile ambients, for the simplest queries it turns out to be quite similar, in practice, to other XML query languages. However, the expression of queries which involve recursion, negation, or universal quantification, has in TQL a clear declarative nature, while other languages are forced to adopt a more operational approach.

References


Semantic Caching of XML Databases

Vagelis Hristidis
Computer Science and Engineering Dept.
University of California, San Diego
vagelis@cs.ucsd.edu

Michalis Petropoulos
Computer Science and Engineering Dept.
University of California, San Diego
mpetropo@cs.ucsd.edu

Abstract

We present a novel framework for semantic caching of XML databases. The cached XML data are organized using a modification of the incomplete tree [ASV01], which has many desirable properties, as incremental maintenance, containment decidability and remainder queries generation in PTIME. The modification we propose alleviates the exponential blowup observed in [ASV01] by partitioning the domains of the XML schema nodes in domain ranges. We also provide an upper bound on the total size of the conditional tree type of the modified incomplete tree (MIT), which describes the data stored in the MIT.

XCacher operates on top of XML databases and intercepts the query requests from a web server. We show how the MIT is maintained and how queries are answered by sending a complete and non-redundant set of remainder queries to the XML database. Finally we present a replacement algorithm for the MIT.

1 Introduction

A considerable amount of work has focused on the problem of semantic caching for database-backed web applications [DFJ+96, LN01, GG99, LC99, Sel88]. These works focus on relational databases. As the number of web applications that are backed by XML databases increases, so does the need to provide efficient caching mechanisms that are suitable for the nature of the XML queries. However, the semantic caching approaches available, focus on specific classes of SQL queries [DFJ+96, LN01], which do not capture the navigational nature of the XML queries. XCacher is a system that facilitates semantic caching of XML databases for a subset of XQuery [W3C01], which is enough to support most reasonable applications.

In particular, XCacher operates on XML trees and supports XQuery queries that do not contain nested FOR-WHERE-RETURN expressions. Each query consists of an extract (FOR-WHERE clauses) and a construct part (RETURN clause). The extract part, which is cached by XCacher, is equivalent to a prefix-navigation query (ps-query), as defined in [ASV01], which selects a prefix of the source XML tree. The cache is organized as a modified incomplete tree (MIT), which is based on the idea of the incomplete tree presented in [ASV01] and is an incomplete copy of the source XML tree. The representation of the cached data as an incomplete tree offers more flexibility and containment opportunities for subsequent queries than the traditional overlapping boxes [DFJ+96] employed so far to organize a semantic cache.

The MIT offers considerable advantages over the incomplete tree, i.e., more efficient maintenance and bounded size. Furthermore, the MIT retains the desirable properties described in [ASV01], i.e., incremental maintenance, containment decidability and remainder queries generation are in PTIME.

This work has the following contributions:

• A novel architecture is presented for performing semantic caching for applications backed by XML databases.

• A modification of the incomplete tree [ASV01] is presented and we show how the cache can be organized as a modified incomplete tree. The main drawback of the incomplete tree [ASV01] is the exponential blow-up of its size. We alleviate this problem by a modification that groups the range conditions of the queries into pre-specified domain ranges as described in Section 3.
• An algorithm is described that creates in PTIME the remainder queries, which are guaranteed not to retrieve any data already in the cache.

• The MIT is stored in main memory, which bounds its size. A replacement algorithm is presented that removes the least recently used piece of data and its corresponding description from the MIT, when it gets full.

The paper is structured as follows. In Section 2 we present the framework and the architecture of XCacher. Section 3 describes the process of refining a MIT when a query arrives to the system. Section 4 explains how XCacher creates the remainder and the refinement queries for a query. An overview of the replacement algorithm is presented in Section 5. Finally in Section 6 we conclude and present future work directions.

2 Framework and Architecture

The architecture of XCacher is shown in Figure 3. XCacher operates on top of an XML database server that exports a view $V$. $V$ is defined as a labeled tree [ASV01], whose structure is described with a schema tree $S$. The ordering of the children of $V$ is unimportant and there is no distinction between attributes and subelements. Each node of $V$ is assumed to have a unique $id$. The schema tree of Figure 1(a), that will be used as a running example, shows the schema of an auction database, where the catalog contains products that have a name, an optional color, a category they belong to, and a list of sellers. Each seller has a name, a price he/she is selling the product for, and a rating according to the customers’ feedback.

XCacher runs as an application in the application server and a web server operates on top of the application server. Each web page $p$ consists of a set of XQuery [W3C01] queries $q_1, \ldots, q_m$ that do not contain any nested FOR-WHERE-RETURN expressions. Each query $q$ consists of an extract part $q^E$ (FOR-WHERE clauses), which is served by the XML database, and a construct part $q^C$ (RETURN clause), which is executed at the query composer. When $q^C$ is executed against the result of $q^E$, it constructs a result equivalent to the result of the initial query $q$, that is, $q^C(q^E(V)) = q(V)$. $q^E$ is a prefix-selection query (ps-query) [ASV01]. A ps-query browses the input tree $V$ down to a certain depth starting from the root, by reading nodes with specified element names and possibly selection conditions on data values. The answer to a ps-query is a
prefix of \( V \). For example the query of Figure 1(b) returns the products with category “cars”, which have a
seller, who sells the product between \( \$7.5K \) and \( \$8K \) and has a rating greater than 3.

XCache is responsible for the efficient execution of
the extract queries \( q^E \). The modules of XCache shown
in Figure 3 are the following: The query decomposer
takes as input an XQuery expression \( q \) and outputs \( q^E \)
and \( q^F \). The query rewriter module takes as input the
extract query \( q^E \) and determines if \( q^E \) can be answered
entirely from the data stored in the cache, i.e., with-
out accessing the XML database. If it can, the query
rewriter outputs a refinement query \( q^F \), which is the
same as \( q^E \) and is executed on the data stored in cache.
Otherwise, a refinement query \( q^F \) and a set \( S^R \) of re-
mainder queries \( q^R \) are created and sent to the query
composer and the XML database respectively, as de-
scribed in Section 4.

The answers to the remainder queries are input to
the query composer and to the replacement controller,
which is the only module that has a “write” privilege on
the cache. If the new piece of XML data does not fit in
the cache, the replacement controller decides, applying
the replacement algorithm described in Section 5, which
XML data fragment(s) to remove from the cache.

The query composer “merges” the results of the re-
mainder queries and \( q^F \), executes \( q^F \) on the result, and
passes the output to the web server, as described in
Section 4.

The cache is organized as a modified incomplete
tree (MIT) \( T \), which consists of a data tree \( T_d \) which is
a prefix of \( V \) and a conditional tree type \( \tau \) which describes the information stored in \( T_d \). In particular, a
conditional tree type \( \tau \) is a specialization of the schema tree \( S \), i.e., a mapping of conditions to the schema
nodes of \( S \), where some schema nodes (which are called donable below) are cloned to denote disjunction
between their conditions. That is, the information stored in \( T_d \) for a node \( s \) of \( S \) is described by the union
of all instances of \( s \) in \( \tau \). The conditions applied to the schema nodes in \( \tau \) are equivalent to those of the
ps-queries. Figure 2 shows (a) the data tree \( T_d \) and (b) the conditional tree type \( \tau \) of the MIT after
executing Q1. The generation of the conditional tree type is performed by the replacement controller module
by applying the Refine MIT algorithm presented in Section 3. The size of \( T \) is bound by the available
main memory to provide fast access times.

The MIT differs from the incomplete tree defined in [ASV01] in the following ways: First, the conditional
tree type of a MIT describes the information contained in \( T_d \), in contrast to the incomplete tree, where the
conditional tree type describes the missing information. This change was made because the em replacement
controller, shown in Figure 3, needs the information of what is currently stored in the MIT to decide what
to replace. Second, the specialized types\(^1\) of an element \( s \) in \( \tau \), i.e., the descriptions of the data of type
\( s \) contained in \( T_d \), are disjoint. That is, there cannot be an element of \( T_d \) conforming to two different
specialized types. This property simplifies the process of generating the remainder queries as it is shown
below. Third, there is no mapping between the elements of the data tree of a MIT and the specialized
types of the conditional tree type, because the specialized types are disjoint and the data tree is considered
a “queryable” data source for the underlying query processor. In contrast, the incomplete tree generates a
unique label \( l(t) \) for each specialized type \( s' \) and all elements in \( T_d \) that conform to \( t \) have the same label
\( l(t) \).

Hence the overhead imposed to maintain this mapping in a caching system with frequent replacements
is avoided. Finally each specialized type \( t \) of \( \tau \) is annotated with a timestamps list of the \( l \) last timestamps,
when a query \( q \), overlapping with \( t \), arrived to the system. The timestamps lists are used by the replacement
algorithm as described in Section 5.

\(^1\)The specialized types are defined formally in Section 3
The MIT is simpler and easier to maintain than the incomplete tree, but the desirable complexity results of the later still hold. In particular, the refine algorithm and the remainder queries' generation for a MIT T are more efficient because they do not access the data tree of T, but only its conditional tree type. Furthermore the size of the MIT is bounded as it is shown below. Also, similarly to the incomplete tree, the following problems are in PTIME: incremental maintenance, containment decidability and remainder queries' generation. A drawback of the MIT is that it does not infer the emptiness of ps-queries with respect to the schema tree in order to avoid their execution. The performance overhead that is imposed though is minor since these queries are executed against the data tree of T that is kept in main memory and not against the XML database server that exports the view V.

3 Refine MIT

The Refine MIT algorithm takes as input the schema tree S, the conditional tree type τ of the MIT T and a ps-query q and outputs a refined conditional tree type τ' that describes q in addition to the queries already described in τ.

Refine MIT proceeds in three steps. The first step constructs the conditional tree type τ_q of q, by applying the conditions of q on the input schema tree S. Figure 1(c) shows τ_q1. In the second step, the range conditions of τ_q are normalized according to the domain partitions, which are partitions of the active domains of the node values into domain ranges. For example, in τ_q1, the range condition on the price has changed from ($7.5K, $8K) to [$7K, $8K), because the domain partition of price has domain ranges of 1K each.

The normalized conditional tree type τ_q^N of τ_q is defined as a conditional tree type, where each condition specifies a continuous set of domain ranges, which contains the range specified in q. According to this definition, the data tree represented by the initial conditional tree type τ_q is always a prefix of the data tree represented by the normalized conditional tree type τ_q^N. Figure 2(b) shows τ_q^N1, which is the same as τ of T, since Q1 is the first query that arrived to the system.

The third step of Refine MIT computes the union of the input conditional tree type τ of T and the normalized conditional tree type τ_q^N of the query q, and outputs the refined conditional tree type τ':

$$τ' = τ ∪ τ_q^N$$ (1)

In order to compactly and efficiently merge the two conditional tree types, we must decide which nodes of τ will be further specialized, i.e., what parts of τ will be cloned to represent the overlap between the conditions of τ and τ_q^N. Specializations occur on clonable nodes, that are defined as a subset of the repeatable nodes of S. For example, in the schema tree of Figure 1(a), product and seller are clonable. A specialized type t is a specialization of a subgraph G of the schema tree S in τ, that is, an annotation of G with conditions. The root of G is a clonable node and the leaves of G are either clonable nodes or leaves of S. For example a seller specialized type is marked in Figure 2(b).

The basic idea of the algorithm that merges the two conditional tree types, which is not described in detail due to lack of space, is the following: We traverse τ and τ_q^N top-down until we find the first clonable node and if there is no specialized type t in τ that contains the corresponding specialized type t_q of τ_q^N, the current node of τ is cloned and annotated with the conditions of t_q. If t_q is contained in a specialized type
$t$ of $\tau$, we continue with the first clonal descendant of $t$. If $t_q$ overlaps with $t$ then the current node of $\tau$ is cloned and annotated with the conditions of $t_q$ such that the new specialized type does not overlap with $t$.

For example, Figure 4(a) shows a query $Q2$ submitted after the query $Q1$ of Figure 1(a). If $\tau$ and $\tau_{Q2}^N$ are the ones in Figures 2(b) and 4(b) respectively, then the ReFine MIT algorithm generates the $\tau'$ that is shown in Figure 4(c). Traversing $\tau_{Q2}^N$ and $\tau$ top-down, and comparing their specialization, no difference is observed until the seller specialized type. The seller specialized type $t_{s,d}$ of $\tau$ overlaps with the seller specialized type $t_{Q2}^{s,d}$ of $\tau_{Q2}^N$, so seller is cloned and a new specialized type describing the difference of $t_{s,d}$ from $t_{Q2}^{s,d}$ is created in $\tau'$. Note that the specialized types of $\tau'$ are non-overlapping.

Finally, the current timestamp is added to the timestamp lists of all specialized types that overlap with the current query $q$. When a specialized type $t$ is cloned, then all the clones inherit the timestamps list of $t$.

If we define a finite domain partition for each node of the schema tree $S$, we can show that the size of the conditional tree type is bounded, in contrast to [ASV01], where no domain partitions are used. Assume that $S$ has $n$ schema nodes and each node $s_i$ is partitioned into $m_i$ domain ranges. Then in the worst case, where there is exactly one repeatable node $s'$ under the root and $s'$ is the only clonal node in $S$, the maximum number of specialized types is $\prod_{i=1}^n m_i$ and the maximum size in number of nodes of the conditional tree type $\tau$ is $n \cdot \prod_{i=1}^n m_i$, since each specialized type will have at most $n$ nodes.

4 Remainder and Refinement Queries

Given a MIT $T$, with conditional tree type $\tau$, and a ps-query $q$, the remainder query $q^R$, which is expressed as a set $S^R$ of local remainder queries, is equivalent to the query $q - \tau$, that is, $q^R$ retrieves from $V$ the data not stored in $T$. The navigation of the XML view $V$ starts from the nodes that correspond to the suitable leaf nodes of the data tree $T_d$ of $T$ and not from the root of $V$, to avoid navigating on nodes already in $T_d$. We call such queries local ps-queries [ASV01], which are ps-queries that operate on the subtree rooted at a node $n$ different than the root of $V$. Local ps-queries are possible because for each node in $T_d$ we keep a pointer to the corresponding node of $V$. For example, consider the MIT of Figure 2 and the normalized query $Q2^N$ corresponding to the conditional tree type $\tau_{Q2}^N$ in Figure 4(b). The remainder query $Q2^R$ is the local ps-query shown in Figure 5.

The local remainder queries are generated in PTIME by the following procedure: First the normalized conditional tree type $\tau_q^N$ is rewritten such that each specialized type refers to single domain ranges, that is, $\tau_q^N$ is split into minimal specialized types. These are either completely contained or disjoint to a specialized type of $\tau$ of $T$. Then, each minimal specialized type of $\tau_q^N$ is checked for containment against $\tau$. The specialized types of $\tau_q^N$, which are not contained in $\tau$, form a minimal and complete set of remainder queries for $q$. Finally, they are merged into a smaller number of local ps-queries and sent to the XML database server.

Due to the normalization process with domain ranges in the ReFine MIT, the number of local queries is small, in contrast to [ASV01], by an analysis similar to the one about the maximum conditional tree type size in Section 3. This property considerably improves the performance because it avoids the overhead of executing many “small” queries on the XML query engine.

The results of the remainder queries arrive at the query composer, where they are filtered to remove any data retrieved due to the normalization of the range conditions. At the same time, the refinement query $q^F$ extracts the data in the data tree $T_d$ that are in the answer of $q$. It is:

$$q^F = q^F \cap \tau$$ (2)
The filtered results $q^{R_j}(V)$ of the remainder queries are merged with the results of $q^F$, and $q^C$ is executed on their result. It is

$$q(V) = q^F(q^F(T_d) \cup q^{R_j}(V))$$  \hspace{1cm} (3)

5 Replacement Algorithm

The replacement algorithm selects one specialized type $t$ of the conditional tree type $\tau$ of the MIT $T$ and removes it along with the data in the data tree $T_d$ of $T$ that conform to $t$. The decision is based on the timestamps lists of the specialized types. We use LRU, although other more sophisticated approaches [RV00] can be employed. Only the leaf specialized types are candidates for removal, to retain the coherence of $T$.

For example, the candidate specialized types for the conditional tree type of Figure 4(c) are the two seller specialized types. Both of them have the latest timestamp in their timestamps lists, since they overlap with $Q2$, so the replacement algorithm selects arbitrarily one of them. Without loss of generality, suppose that the second specialized type is selected. An update extended XQuery [THW01] $q^P$ is generated that deletes the data that are relevant to the selected specialized type, and $q^P$ is executed against $T_d$. In particular, $q^P$ is:

```xml
FOR $p$ IN document("catalog.xml")/catalog/product, $s$ IN $p/seller
WHERE $p/cat = "cars"$ AND $s/price >= 8000$ AND $s/price < 9000$ AND $s/rating > 4$
UPDATE $p$ { DELETE $s$ }
```

If the space emptied is not enough to insert the data of the new query to $T$, another specialized type is selected and removed in the same way.

6 Conclusions and Future Work

We presented a system for the efficient caching of a broad class of XML queries on XML databases, where the cache is organized as a MIT. We are working on implementing the system to measure the improvement in the response time for a query workload and experiment with replacement policies other than LRU. Also, we are working on an algorithm, that will be executed periodically on the incomplete tree, to compact the conditional tree type’s representation. Finally, we work on the pipelining opportunities that the system presents in order to increase its throughput.

7 Acknowledgements

We wish to thank Yannis Papakonstantinou for the ideas he proposed and Victor Vianu for the discussions we made regarding the properties of the incomplete tree.

References


ACE-XQ: A CachE-aware XQuery Answering System

Li Chen and Elke A. Rundensteiner
Department of Computer Science,
Worcester Polytechnic Institute, Worcester, MA 01609
{lichen|rundenst}@cs.wpi.edu

Abstract. Caching popular queries and reusing results of these previously computed queries to speed up query processing is one important query optimization technique for distributed environments such as the Web. However, existing query-based cache systems, based on query containment and rewriting techniques developed for relational queries, are not appropriate for supporting the more powerful XML queries. We hence propose the first solution for XML query processing using cached XQuery views. In particular, we describe in this paper an XQuery-based semantic caching system called ACE-XQ, that we have implemented to realize the proposed containment mapping and query rewriting techniques. Preliminary experiments confirm the feasibility of our approach and also illustrate the performance gains achievable by ACE-XQ over the original XQuery query engine.

1 Introduction

Due to the growing popularity of XML, many Web applications retrieve desired information from multiple XML sources by issuing queries against remote data sources across the Internet. However, delayed data transmissions and lost data packages often hinder query-embedded Web applications from retrieving such desired remote information in an efficient manner. Hence, the idea of query-based caching known for improving performance by orders of magnitude in relational distributed systems [7, 9] could also provide a viable solution for efficient XML query processing in the Web environment.

Related Work. As a fundamental technique underlying view-based query answering (or semantic caching), query containment has been extensively studied for relational conjunctive queries [3, 12, 13]. Even though the topic of semantic caching of web queries has recently gained the attention of researchers, assumptions are usually made that web queries are simple form-based queries. This means they can be either translated into selection SQL queries with simple predicates over form attributes [14] or reduced to a traditional Datalog query evaluation for simple boolean queries [6]. Such an assumption is reasonable when web data sources are stored in back-end relational database systems or when plain web pages are being wrapped to expose limited querying capabilities via a form-based interface.

Other work tackles the query containment problem for semi-structured data [1, 2]. Papakonstantinou et. al. [17] have addressed the query containment and rewriting problem for the Tree Specification Language (TSL). They generalize the relational containment mapping technique [3] to establish mappings between complex object pattern variables in two TSL queries. Florescu et. al. showed in [8] that query containment for a union-free, negation-free subset of StruQL is decidable, and even NP-complete for a significant subset of this language restricting the regular path expressions to contain only * or label constants.

However, no work in view-based query answering for semi-structured data has investigated this issue for state-of-the-art XML query languages such as the recent W3C proposal of XQuery [18], which is capable of expressing complex structural pattern matching, joins of multiple XML data sources and output restructuring. No semantic caching system developed thus far has been targeting XQuery.

XQuery is heavily based on the notation of expression which may be nested with full generality. For example, a nested XQuery expression can interleave result construction with further pattern matchings, which is not trivial to be reduced to a datalog-like language like TSL [17]. Also, the regular expression type forms the foundation for XQuery semantics, whereas TSL does not support regular path expressions. Similar to [8], we only deal with the conjunctive, negation-
free subset of XQuery, where the regular path expressions are restricted to range over tag names and either * or //.

On the other hand, we consider the rewriting of a new XQuery in terms of the restructured view schema of a containing query, whereas [8] does not deal with the restructuring capability of StruQL. Furthermore, [8] focuses on the fundamental decidability of query containment for StruQL. Our focus instead is to provide the first practical framework for a cache-aware XQuery answering system, including XQuery pre-processing by normalization, type-enhanced XQuery containment mapping and rewriting, and cache management. A semantic caching system called ACE-XQ \(^1\) [5] has been implemented to realize the proposed techniques.

2 Motivating Example

Suppose we have the cached XQuery V1 and the new query Q as shown in Figure 1. Both V1 and Q involve a “join” of two XML documents which conform to the two DTD structures shown in Figure 2.

\[
\text{FOR } s_b \text{ IN } \text{bib0/book}, s_a \text{ IN } s_b/"[last="James"] \\
\text{LET } s_w := (\text{FOR } s_b' \text{ IN } s_b/reviews0/book \\
\text{WHERE } s_b'/title=s_b/title \\
\text{RETURN } \langle result \rangle b' (\langle result \rangle)) \\
\text{RETURN } \\
\langle \langle entry \rangle s_b/s_year, s_b/title, s_a, \\
\text{(reviews)} \\
\text{RETURN } \langle s_u/review \rangle \} \\
\langle /reviews \rangle \\
\langle /entry \rangle \\
\text{V1}
\]

\[
\text{FOR } s_b \text{ IN } \text{bib0/book}, s_a \text{ IN } s_b/author \\
\text{WHERE } s_a/last="James" \\
\text{RETURN } s_b/title, s_a. \\
\text{(favorable reviewers)} \\
\text{RETURN } \langle \langle s_r/reviewer \rangle \} \\
\langle /favorable reviewers \rangle \text{ Q}
\]

Figure 1: Cached Query V1 and New User Query Q

V1 retrieves the year, title, author and editor information of the books from bib.xml at a remote site (bib0 denotes the variable bound to the root element of bib.xml) whose author or editor has the last name of “James”, and the review about such books from another remote document reviews.xml (the root variable is reviews). Q retrieves the books that are authored by a person with the last name of “James” and highly rated (rate>4) by the reviewers.

\(^1\)Our system was previously called XCache, but has been renamed since that name has been registered with a different product already

3 The ACE-XQ Approach

The basic idea of our approach is to explore the containment mappings between the variables specified in the matching patterns of two XQueries for query containment decision and for query rewriting. Since an XQuery defines its variables using regular expressions, our containment mapping strategy incorporates a type-related query analysis, in particular type inference and subtyping relationships, for the purpose of matching regular-expression-type-based variables. To facilitate query reasoning for containment mapping, we first propose a normalized form for XQuery queries, which helps to separate the pattern matching semantics of a query from its restructuring semantics.

In this example, we can see that the answer required of Q is totally subsumed in that of V1. In this paper, we sketch out the basics of our XQuery containment and rewriting strategies using this running example.
their referring pattern variables to form variable-specific subqueries. The **Query Pattern Register** uses the pattern variables, their dependency relationships as well as associated conditions and return expressions extracted from the decomposed query to construct the query semantics descriptor. The query's restructuring semantics are also captured separately by nested relational restructuring operators.

For a pair of new and cached queries, the **Query Containment Mapper** explores containment mappings between their pattern variables. It makes the query containment decision depending on whether one-to-one containment mappings can be established. Type inference and subtyping mechanisms supported by a static type checker of XQuery are utilized for this containment mapping between the regular-expression-type-based variables specified in the two queries. Based on the established containment mappings, the **Query Rewriter** rewrites the new query with respect to the possibly severely restructured view structures of the containing cached queries.

The **Replacement Manager** incorporates replacement strategies for purging both complete and partial regions to make space for new queries. For better cache space utilization, the **Region Coalescer** merges and splits views to control region granularity over time.

Due to space limitations, we focus on the description of the query decomposition and containment strategies based on the introduced motivating example. Interested readers are referred to [4] for more details about the containment mapping and rewriting algorithms used in ACE-XQ.

### 4 XQuery Semantics Analysis

**XQuery Pre-processing via Normalization**

To exploit normalization rules for transforming an XQuery arbitrarily composed of FLWR (FOR-LET-WHERE-RETURN) expressions into a form which separates its pattern matching and restructuring semantics and explicitly exposes the interdependencies between those two semantics.

The XQuery normalization has been addressed both in the W3C proposal for XQuery formal semantics [19] and by Manolescu et. al. [15] to obtain an appropriate XQuery form for which the XML-to-XQuery translation can be initiated. The normalization rules proposed by W3C are used for transforming FLWR expressions in the full XQuery syntax into their simplified form in the XQuery Core syntax [19]. Although aiming at different goals, these two normalization techniques do not necessarily exclude each other.

Using the normalization rules provided by [15], the LET bound variables can be eliminated by substituting their occurrences for expression definitions. Also, the FROM expressions that are nested within a FOR or WHERE clause can be unnested and put inside of the RETURN clause. Reversely, a RETURN clause could be unnested as well. For our interest in the XQuery containment problem, we select the unnesting rule for the FOR and WHERE clauses which simplifies them to contain only path expressions while still capturing the FROM hierarchy specified in the original query by nesting FROM expressions only within the RETURN clauses. An FROM expression at any nesting level of such a form can be represented as below:

\[ \text{FOR } \vec{x} \text{ IN } \vec{E}(\vec{y}) \text{ WHERE } C(\vec{y}'') \text{ RETURN } R(\vec{y}''), \]

where the arity of \( \vec{x} \) is the same as that of \( \vec{E}(\vec{y}) \). The definition of \( \vec{x} \) by \( \vec{E}(\vec{y}) \) can be expanded as FOR \( \forall x_1 \text{ IN PE}_1(\vec{e}_{pre}X), \forall x_2 \text{ IN PE}_2(\vec{e}_{pre}X, x_1), \ldots, \forall x_n \text{ IN PE}_n(\vec{e}_{pre}X, x_1, \ldots, x_{n-1}) \), where PEi denotes the path expression used to define variable \( x_i \), and \( \vec{e}_{pre}X \) represents the variables defined in the outer FROM expressions w.r.t. the current FROM expression.

For example, during the normalization process for V1, the LET bound variable $u$ is first eliminated and its occurrence in the nested FOR clause used to define $u$ is substituted for its FROM definition. Then the unnesting rule is applied to derive the required normalized form of V1, as depicted in Figure 4:

![Figure 4: The Normalized Form for V1](image)

**Figure 4: The Normalized Form for V1**

We also provide additional rules to eliminate unnecessary FROM expressions if the defined variables do not occur elsewhere, and to add FOR and WHERE clauses to keep variable definitions free of filter expressions:

\[
\text{FOR } a \text{ IN } b/\text{title} \text{ WHERE } a/\text{last} = "James" \implies \text{RETURN } a/\text{title} = 1
\]

\[
\text{FOR } f \text{ IN } b/\text{authors} \text{ WHERE } f/\text{last} = "James"
\]
Variable Dependency. We use \$bibi0 /book \$b to denote that variable \$b is dependent on \$bibi0 via the navigation path /book. Similarly, \$b \rightarrow \$a, \$b /title \$t and \$reviews0 /book \$b'. The pattern matching semantics of a query are captured by the specified variables and their dependency relationships.

We also break down the WHERE and RETURN clauses and move condition and return expressions close to the FOR clauses where the referring variables are defined. We hence obtain nesting subqueries, each of which clusters all the conditions and return expressions with locally defined variables. Each variable-specific subquery hence encapsulates the selection and projection semantics of local variables.

In ACE-XQ, we utilize the data structures in Figure 5 to represent the semantic descriptor of a normalized query.

\[ V: \text{variables defined in a local subquery} \]
\[ VarECRs: \text{variable specific semantics} \]
\[ VarDep: \text{parent and children variables} \]
\[ V_{top}: \text{referred earlier-defined variables} \]

\[
\begin{array}{|c|c|c|c|}
\hline
\$bibi0 & V_{top} & V & \$a, \$b, \$t \\
\hline
\$b & \{\$bibi0\} /book & \{\$a, \$b, \$t\} & \$bibi0 \{\$b\} \$b \{\$a, \$t\} \\
& & & \\
\hline
\end{array}
\]

Figure 5: Variable Dependencies in V1's subQ1

The original nesting FWR expressions imply the view structure, which may be affected by the movements of the WHERE and RETURN clauses into different nesting levels. For example, assuming V1 returns \$a in the inner FWR expression where \$r bound with the reviewer objects is to be returned. This indicates a “product” of objects bound to \$a with those reviewer objects. We utilize some restructuring operators, such as “product”, “nest”, “unnest” from the nested relational context, to annotate the effect caused by moving \$a up to the outer FWR expression in order to form the \$a-specific subquery.

5 Subtyping Enhanced XQuery Containment Mapping

As a regular expression language, XML processing is closely related to tree automata theory [16, 11]. Tree-automata-based regular expression types can be inferred for the variables specified in the matching pattern of an XQuery and for the return type composed of these variables. XDuce [10] provides core functions for such static type checking of XQuery through a functional programming language, and we thus make use of XDuce in our ACE-XQ system.

Our containment mapping strategy proceeds in a progressive fashion guided by the variable dependencies of the new query Q. It first matches each variable in Q to a variable of the cached query V1 based on their inferred types from the definitions and their subtyping relationships. It then establishes a containment mapping between a matched variable pair by checking their selection and projection semantics, similar to the relational view utility conditions.

We now walk through the subtyping-enhanced containment mapping process using the example queries V1 and Q in Figure 1. V1 and Q both query the same XML documents, bib.xml rooted at $bib0 and reviews.xml rooted at $review0. We illustrate in Figure 6 the graphic presentation of the two queries' variable dependencies as well as the associated conditions and return expressions.

Since variable \$b in V1 (denoted by \$b_{V1}) and \$b in Q (\$b_{Q}) are defined the same using PE1($bibi0/book), we set up a containment mapping from \$b_{V1} to \$b_{Q}. We then check whether any condition specified in the \$b-specific subquery of V1 corresponds to some equally or more restricted condition posed on \$b_{Q}. In this case, there is no condition for either \$b_{V1} or \$b_{Q}. We now check that the document nodes to be returned by the RETURN expression referring to \$b_{Q} are also required in any RETURN expression with the occurrence of \$b_{V1}. In this case, \$r1($b/title) in Q is also required to be returned in V1.
The join condition on Q's R1($b/title) in source bib.xml and R3($b/title) in reviews.xml match the join condition specified on R2($b/title) and R4($b/title) in V1. We hence continue the containment mapping process to check whether V1's $a_{v1}^Q$ defined as PE2($b/*$) matches Q's $a_Q^Q$ defined as PE2($b/author$). The regular expression type inferred for $a_{v1}^Q$ is (Author+Editor+) and the type for $a_Q^Q$ is Author. Based on the subtyping theory for regular expression types, we find that $a_{v1}^Q <: a_Q^Q$, where $:<$ denotes subtypeOf. We thus can set up a containment mapping from $a_{v1}^Q$ to $a_Q^Q$ and annotate their subtyping relation with a type constraint typeswitch($a_{v1}^Q$) as author. If $a_{v1}^Q$ is also returned by V1, which is the case, we rewrite the variable definition of $a_Q^Q$ to obtain the document nodes bound to it from those returned by $a_{v1}^Q$ using such a type constraint. We further compare the strictness of conditions attached to $a_{v1}^Q$ and $a_Q^Q$.

A similar containment mapping process proceeds for all the variables defined in reviews.xml by V1 and those in Q. Although $a_Q^Q$ has no corresponding variable match in V1, it can be specified on R5($b/review$) returned by V1. Therefore, our containment mapping strategy allows for subtyping relations between Q's variables and V1's variables or return expressions, in addition to requiring stricter conditions on Q than on V1. Based on the successful containment mappings between pattern variables of V1 and Q by exploiting their subtyping relations, we determine Q $\subseteq$ V1.

XQuery Rewriting. We now exploit the established containment mappings to obtain a rewriting of Q $Q^{rew}$ (shown in Figure 7) with respect to V1's view structure. In V1's view structure, the association relationship between title and author or editor is preserved inside of a newly constructed element entry. We hence specify a variable $x$ in $Q^{rew}$ by PE($SV1Res0/entry$), where $SV1Res0$ represents the variable for the root element of V1's result document. We then return the title-author pair of each object bound to $x$ as required by Q. For regular reviewers, we specify a variable $y$ in $Q^{rew}$ to be bound to its match in V1, the descendant element <review> of <entry>, restrict its rate with the condition of $>4$ and return the corresponding reviewer.

6 Preliminary Evaluation

Our ACE-XQ system is developed using Java JDK 1.3. The front end of the ACE-XQ system is running on top of Apache Web Server and Tomcat servlet engine. We use the Quilt parser and query engine available at http://cheops.cis.upenn.edu/Kweelt to analyze and evaluate the input XQuery. We also utilize the type inference and subtyping mechanisms provided by the XDuces system [11] for containment mapping.

Our ACE-XQ system checks the correctness of our rewritten queries by comparing their results with those produced by directly evaluating the original query against the remote documents. It also serves as a testbed for further conducting in-depth experimental studies investigating the query performance gain achieved by answering queries using cached views.

In a distributed system like the Internet, we expect that a query-based caching system like ACE-XQ would save the fetching costs for queries answerable by cached ones. In our experiments, we install the Kweelt query engine on two UNIX machines across a Local Area Network (LAN). One also has the ACE-XQ system installed and allows users to input queries, while the other hosts a set of XML documents as a remote data server.

![Figure 8: Comparison of Query Processing Times](image)

In our initial experiments, we test different XML document sizes. We warm up ACE-XQ with several queries and design the new queries to be either totally contained in or overlapping with one of the cached
Table 1: ACE-XQ Processing Time Components

<table>
<thead>
<tr>
<th>XML Doc Size (KB)</th>
<th>Decomposition Time (ms)</th>
<th>Rewriting Time (ms)</th>
<th>Evaluation Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for Contained Case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>0.8</td>
<td>5.2</td>
<td>173.6</td>
</tr>
<tr>
<td>890</td>
<td>0.8</td>
<td>5.4</td>
<td>1068.8</td>
</tr>
<tr>
<td>1800</td>
<td>0.8</td>
<td>5.2</td>
<td>4525.4</td>
</tr>
<tr>
<td>for Overlapping Case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>0.8</td>
<td>1.6</td>
<td>1126.2</td>
</tr>
<tr>
<td>890</td>
<td>0.7</td>
<td>1.6</td>
<td>12778</td>
</tr>
<tr>
<td>1800</td>
<td>0.8</td>
<td>1.7</td>
<td>54660</td>
</tr>
</tbody>
</table>

queries. We compare the query processing time spent when using ACE-XQ versus when by-passing it.

The result in Figure 8 is consistent with our expectation. The query performance improvement for the totally contained case using ACE-XQ is the most significant, by an order of magnitude in a LAN environment. The performance improvement is about 1.8x times for the overlapping cases.

We break down the query processing time used in the ACE-XQ system into three components, i.e., the time used for query decomposition, for query containment mapping and rewriting and for query evaluation respectively. As observed in Table 1, the computational overhead for ACE-XQ is comparatively small with respect to the overall cost.

7 Conclusions

The XQuery-based ACE-XQ query system aims to minimize the fetching cost of XQuery results by answering new queries using cached queries whenever possible. Our preliminary experimental results are promising. Given that our “remote” data sources are on a LAN and hence network delay is minimal, we expect that the performance improvement would be even more dramatic in Web systems over the Internet.

References


On space management in a dynamic edge data cache

Khalil Amiri, Renu Tewari, Sanghyun Park, and Sriram Padmanabhan
IBM Thomas J. Watson Research Center
{amirik,tewarir,sanghyun,srp}@us.ibm.com

Abstract
Emerging web applications are increasingly serving dynamic content generated by querying back-end database servers. The serving of dynamic content can be scaled by offloading applications and caching data at edge servers. In recent work, we proposed a persistent and self-managing edge-of-network data cache that is dynamically populated based on the application query stream and stored locally in a persistent database. In this paper, we discuss the challenges of cache maintenance in such a dynamic environment, focusing on replacement issues in the presence of update-based consistency protocols. We describe how our experience with a real prototype, built using a JDBC driver and DB2, highlights the limitations of traditional approaches. Furthermore, we propose a cache replacement mechanism and policy that address these challenges.

1 Introduction
Data accessed via the Web is increasingly dynamic, generated on-the-fly in response to a user request or customer profile. Examples of such dynamic data include personalized web pages, targeted advertisements or online e-commerce interactions. Dynamic data is served using a 3-tiered architecture consisting of a web server, an application server and a database; data is stored in the database and is accessed on-demand by the application server components and formatted and delivered to the client by the web server. To improve scalability and performance, caching at edge servers has been widely deployed on the web for static HTML pages. For dynamic content, which requires database accesses, caches are typically by-passed by marking the content uncachable. Recent work has targeted extending the static caching concept by storing the result of a dynamic web request as HTML fragments or other formats indexed by the exact URL string or HTTP request header [11, 9]. Consistency and cache space management issues, however, can easily limit the scalability of these schemes.

In more recent architectures, the edge server (which collectively refers to client-side proxies, server-side reverse proxies at the edge of the enterprise, or caches within a content distribution network(CDN) [1]) acts as an application server proxy by offloading application components (e.g., JSPs, servlets, EJBeans) to the edge [7]. Database accesses by these edge application components, however, are still performed across the wide area network. To accelerate edge applications by eliminating wide-area network transfers, we have recently proposed and implemented DBProxy, a database cache that dynamically and adaptively stores data at the edge [2]. Since the edge server is limited in resources of space and processing power, we rely on efficient cache replacement policies and mechanisms to keep only the most beneficial data in the local database cache. Cache replacement for physical page-based caches, static files, and read-only semantic caches has been thoroughly studied in the literature. Yet, it introduces new challenges in the context of a persistent edge cache containing a large number of changing and overlapping “materialized views” of previous query results. This is because locally stored data can be shared by multiple cached “views”, and can be updated by a cache consistency protocol.

In this paper, we review the design of the dynamic edge data cache in Section 2, discuss the limitations of traditional approaches to cache replacement and offer alternative solutions in Section 3, briefly review related work in Section 4, and summarize the paper in Section 5.

2 Background
We assume in this discussion that application components (e.g., servlets) are running on the edge server (e.g., using the IBM WebSphere Edge Server [7]). The edge server receives HTTP client requests and processes them locally; passing requests for dynamic content to application components which in turn access the database through a JDBC driver. The JDBC driver
manages remote connections from the edge server to the back-end database server, and simplifies application data access by buffering result sets, and allowing scrolling and updates to be performed on them.

2.1 DBProxy overview

We designed and implemented an edge data cache, called DBProxy, as a JDBC driver which transparently intercepts the SQL calls issued by application components executed on the edge and determines if they can be satisfied from the local cache (shown in Figure 1). DBProxy is designed to be deployed on edge servers which could number in the tens to hundreds. Consequently, it needs to be self-managing to limit the administrative overheads of a large scale deployment. Furthermore, each edge server could serve a different population (i.e., may observe a different access pattern) and could contain different resource constraints, making manual optimizations impractical. To make DBProxy as self-managing as possible, while leveraging the performance capabilities of mature database management systems, we chose to design DBProxy to be: (i) persistent, so that results are cached across instantiations and crashes of the edge server; (ii) DBMS-based, utilizing a stand-alone database for storage to allow for the efficient execution of complex local queries; (iii) space-efficient, storing query results in common tables to avoid redundancy whenever possible; (iv) dynamically populated, populating the cache based on the application query stream without the need for pre-defined administrator views; and (v) dynamically pruned, adjusting the set of cached queries based on available space and relative benefits of cached queries.

2.2 Common store

Data in a DBProxy edge cache is stored persistently in a local stand-alone database. The contents of the edge cache are described by a cache index containing the list of queries. To achieve space efficiency, data is stored in shared tables whenever possible such that multiple query results share the same physical storage. Queries over the same base table are stored in a single, usually partially populated, cached copy of the base table at the origin server. Join queries with the same join condition and over the same base table list are also stored in the same local table. This scheme not only achieves space efficiency but also simplifies the task of consistency maintenance, as discussed below. When a query is worth caching, a local result table is created (if one does not already exist) with as many columns as selected by the query. The column type and metadata information are retrieved from the back-end server and cached in a local catalog cache. For example, Figure 2 shows a local table cached at the edge. The local item table is created just before inserting the three rows retrieved by query $Q_1$ with the primary key column (id) and the two columns requested by the query (cost and msrp). All queries are rewritten to retrieve the primary key so that identical rows in the cached table are identified. Later, to insert the three rows retrieved by $Q_2$, the table is altered if necessary to add any new columns not already created. Next, new rows fetched by $Q_2$ are inserted (id = 450, 620) and existing rows (id = 340) are updated. Note also that since $Q_2$ did not select the cost column, a NULL value is inserted for that column. The query matching module ensures that the queries executed against the cache do not return any of the “fake” NULL values in local tables. To handle a large and varying set of cached views, the query

![DBProxy architecture diagram](image-url)

Figure 1: DBProxy key components. The query evaluator intercepts queries and parses them. The cache index is invoked to identify previously cached queries that operated on the same table(s) and column(s). A query matching module establishes whether the new query’s results are contained in the union of the data retrieved by previously cached queries. A local database is used to store the cached data.
matching engine of DBProxy must be highly optimized to ensure a fast response time for hits. Cached queries in DBProxy are organized according to a multi-level index of schemas, tables and clauses for this purpose.

2.3 Update propagation

Update transactions in DBProxy are routed to the back-end database without applying them first to the edge cache, and are therefore guaranteed transactional semantics. Read-only queries in DBProxy are satisfied from the cache if the data is locally available. Data consistency is ensured by subscribing to a stream of updates propagated by the back-end server. Specifically, the cache guarantees δ-consistency, that is, the view exported by the cache to a query corresponds to a consistent past database state that is within δ time units. Other important properties like view monotonicity and immediate update visibility are also guaranteed and the protocols that ensure them are described in [2]. It suffices to note here that the edge data cache can be updated along two paths as shown in Figure 3: the first is through the insertion of a new query result upon a query miss, and the second is through the stream of refresh messages propagated by the origin server. Refresh messages contain updates, deletes and inserts which are applied to the edge server’s partially populated cached copies of the origin tables, without first checking if the tuples propagated to the cache satisfy the predicate of any cached query. We found that such a check can induce significant overhead due to the potentially large number of cached queries and the complexity of query predicates. This is illustrated in Figure 2, where the two bottom rows in the table are inserted after being propagated from the origin because they were written by an update transaction. No check is made as to whether these rows match the predicates of queries \(Q_1\) or \(Q_2\). Excess rows inserted in the table must later be “garbage collected” by the cache replacement process.

3 Cache replacement

To limit space overhead and optimize the usage of usually limited edge resources, the cache space has to be managed such that unused data gets evicted safely while preserving data consistency. Specifically, the goal of cache replacement is to maximize the benefit of the cache for a limited amount of available space. The cache replacement component of DBProxy consists of a replacement policy, that determines what to replace, and a replacement mechanism, that determines how to remove the data.

3.1 Replacement policy

The function of the cache replacement policy is to determine the set of queries to replace from the cache. Cache replacement policies have been extensively studied in different areas, from virtual memory and file buffer caching to, more recently, web caching. The policy we use is a combination of previous approaches and is most suited for edge data caching where the query processing costs and sizes are different. Given a space constraint, our policy tries to maximize the benefit of storing the query results locally for the cost of the space used, similar to the traditional knapsack problem of optimizing the cost-benefit. Determining the benefit of a query depends on multiple factors, namely: i) recency of access (that is the factor
used in the LRU policy); ii) frequency of access (that is used in the LFU policy and is useful for skewed access patterns); iii) miss cost versus hit cost (especially since the query execution costs are high and variable); and iv) the frequency of updates (since an update adds to the overhead of caching). We use an estimated access frequency measure that balances the recency of access (using the last access time) and the frequency of access. Combining the above set of factors, the benefit of maintaining an object in the cache is proportional to the estimated access frequency and the differential miss processing cost, and is inversely proportional to the update frequency. The update frequency term is maintained at the level of tables. The benefit of query is then offset by its space overhead.

Borrowing from the greedy heuristic used in the knapsack solution for replacing queries with variable data-set sizes, we order the queries based on the ratio of \( \frac{\text{benefit}}{\text{space}\_\text{overhead}} \). The objects with the smallest ratio are then marked for removal. While the parameters in the benefit computation can be estimated by maintaining various statistics, the space overhead of a query, as we discuss in Section 3.4, is more challenging to compute accurately because queries can have multiple overlapping tuples, i.e., the same row can “belong” to many cached queries.

### 3.2 Replacement mechanisms: Challenges

The replacement mechanism is the process by which the tuples belonging to queries marked for eviction are removed from the local database tables. This process is complicated by several factors including: the shared storage strategy, the containment checking overhead and the consistency policy used.

#### 3.2.1 Shared store

Recall that the tuples brought in by different queries are stored in common tables as far as possible. In contrast to traditional replacement of files and memory pages, the underlying tuples can be shared across multiple queries in the cache. Consider the queries \( Q_1 \) and \( Q_2 \) of Figure 2, where the result set of query \( Q_1 \) contains rows with \( id \in \{5,120,340\} \) and the result set of query \( Q_2 \) contains rows with \( id \in \{340, 450, 620\} \). If query \( Q_2 \) is to be replaced while \( Q_1 \) remains in the cache, then only the rows with \( id \in \{450, 620\} \) can be removed. The space gained will be that of the two rows deleted and not the size of the entire query which was 3 rows. In general the replacement mechanism should support the following property.

**Property 1** When evicting a victim query from the cache, the underlying tuples that belong to the query can be deleted only if no other query that remains in the cache accesses the same tuples.

**Read-only lazy replacement:** If we assume that the back-end database is read-only (i.e., there are no updates), a simple “counter” based mechanism that counts the number of references to a given tuple, can be used to evict queries that share common tuples in the local table. When a new tuple is inserted into the local table, a reference counter for that tuple is incremented. When a query is marked for deletion, the reference counter for a corresponding tuple is decremented. Eventually, a tuple is “lazily” deleted when its reference counter becomes zero, i.e., there are no queries in the cache that access that tuple. The assumption of a read-only database is obviously unrealistic in practice. Next, we relax this assumption and discuss the implications of updates for the replacement policy and mechanism.

#### 3.2.2 Containment checking

When a new query is received by the cache, the query matching module (shown in Figure 1) verifies whether the new query’s predicate is more restrictive than (i.e., is contained in) that of a cached query’s predicate. Furthermore, the query matching module ensures that the cached query has retrieved all the columns required to evaluate the new query over the local cache. The query matching module can check if a new predicate is contained in the union of several cached predicates. In case of a miss, the query is cached and its predicate and other clauses are added to the cache index. A new query can potentially overlap with a large number of cached queries, but no attempt is made to compute or maintain information about this overlap. One approach to eliminate the problem of common tuples between queries is to partition the tuples into non-overlapping sets and index each set separately in the cache [3]. This approach, though theoretically possible, adds to the overhead of the containment checker, and we found that it was not practical when the number of queries was large. Splitting the queries increases the terms in the predicate clauses where for every pair of queries \( Q_1 \) and \( Q_2 \), their “intersection” set \( Q_1 \cap Q_2 \) is indexed as \( Q_1 \ AND \ Q_2 \). Since each query can intersect with multiple queries, the complexity of the clause in the number of terms, grows linearly with the number overlapping queries. The number of non-overlapping sets, on the other hand, grows exponentially with the number of queries. The containment checker is optimized to quickly find a matching query by using an index hierarchy starting with the table names and the column names used by the different clauses. This
quickly narrows the number of queries to check for a full containment. For overlap checking, on the other hand, the set of possible queries is much larger. Apart from the performance issue, another problem with assuming non-overlapping query sets arises because of consistency maintenance when the back-end database is not read-only. Whenever tuples in the cache are updated, their membership in the non-overlapping sets may change.

### 3.2.3 Consistency management

The consistency manager, as described earlier, propagates all changes (UDIs) to the back-end tables that have been cached at the edge server. The update of a tuple in an edge cache may make it match a larger (or smaller) number of cached query predicates. It is possible that an updated tuple ceases to be useful, because it no longer matches any cached predicate. The replacement policy selects queries for removal, and the replacement mechanism must remove the underlying tuples that belong to these queries’ results but do not belong to the results of any queries that are to remain in the cache. Moreover, the replacement mechanism should also evict any rows inserted by the consistency manager that do not match any query’s predicate. For example, the rows in Figure 2 with id \{770,880\}, that were inserted by the consistency manager but which did not belong to either query’s result set, should be evicted. In general, the following property should hold.

**Property 2** The tuples inserted by the consistency manager that do not belong to any of the results of the cached queries are eventually garbage collected.

The following observation affects the choice of the replacement mechanism and the containment checking strategy:

**Observation 1** The set of tuples forming the result of a cached a query can change dynamically due to possible UDI operations at the back-end that are propagated to the local database.

This observation says that the membership of cached tuples in query results can change dynamically, which implies that maintaining explicit information about this membership or its overlap is not desirable. In particular, the lazy removal approach discussed above, which uses a reference counter will not have the correct counter value if the tuples are added and updated by the consistency manager. Creating non-overlapping sets becomes non-trivial when the consistency manager can add or update tuples. On a UDI, the tuple belonging to a query can change; since determining the reverse mapping, whether a tuple belongs to a query’s result set, requires the re-evaluation of the query predicate over the new tuple, we cannot easily maintain accurate information about the membership of a tuple in one of the non-overlapping sets.

### 3.3 Proposed mechanism: Group replacement

We propose a replacement mechanism that proceeds as a background process concurrently with query hit and miss processing. The idea is to perform pro-active cleaning such that the cache space usage never exceeds a maximum threshold. Thus replacement, in our architecture, is not triggered when there is no space to insert the miss results. While a thorough discussion of alternative mechanisms for replacement is the subject of ongoing work, we briefly describe here one particular and promising mechanism, called group replacement, which is simple to implement and adds no overhead on hit, miss, or update propagation. The basic idea of group replacement is to execute the queries that are to remain in the cache against all the locally cached rows, “marking” any rows that match any of the queries’ predicates. A control column, used as “marked” flag, is created in each cached table. This flag is first reset at the beginning of the group replacement cycle, and is set whenever the row is accessed by the cached query. Once all cached queries are executed, any unmarked rows can be safely deleted. Group replacement is used in conjunction with a replacement policy to determine the set of “victim” queries to be deleted. Note that the overhead of group replacement is linear in the number of queries that remain in the cache.

### 3.4 Determining query size

The replacement mechanisms rely on the replacement policy to determine the query or set of queries to replace. As described earlier, one factor used by the replacement policy, when handling varying sized data-sets, is the size of the query. However, due to overlapping tuples between queries, determining the number of tuples that will get replaced is not straightforward. In particular, the following observation holds.

**Observation 2** The actual number of tuples of a query that can be garbage collected depends on the set of queries that remain in the cache.
The above observation highlights the complexity of using the size as a factor to determine the set of queries to be replaced. It results in a circular dependency—the replaceable size depends on what remains in the cache and what remains in the cache depends on the ordering by the ratio of benefit to size. We assume, initially, that the total size of the query (as determined by the number of tuples hit in the last access) is a good estimator of the replaceable size of the query. Another approach is to evaluate each query and determine its count of “exclusive” (non-overlapping) tuples with respect to all the other queries currently in the cache. This is a high overhead operation, requiring a counter to be maintained for each tuple to represent the number of queries that access that tuple. Using this counter, a re-execution of the query can be used to determine the set of “exclusive” tuples that the query accesses, which becomes a measure of the replaceable size. These two approaches—total size and exclusive size—form two ends of the spectrum of heuristics used to determine the replaceable size of a query. An intermediate approach is to order the queries by benefit (without using size as a factor) and then re-executing each query in descending order of benefit while counting the new tuples accessed by a query to represent its replaceable size. We are evaluating this and other approaches in our ongoing work.

4 Related work

Client-server database systems have also addressed replacement issues in page-based and tuple-based client caches [5, 6]. Replacement in a cache indexed by semantic units such as views or query results introduces different challenges, however. Semantic caches have been proposed in client-server database systems [3] but that work addressed only read-only caching and used a different storage implementation. Predicate-based caches [8] addressed similar issues, but opted for a different approach to consistency maintenance. Tuples propagated as a result of UIDs performed at the origin are matched with all cached query predicates to determine if they should be inserted in the cache. A simplified form of semantic caching targeting web workloads and using queries expressed through HTML forms has been recently proposed [11], but this work did not address consistency or replacement. The caching of query results has also been proposed for specific applications, such as high-volume major event websites [4, 10]. The set of queries in such sites is known a priori and the results of such queries are updated and pushed by the origin server whenever the base data changes.

5 Summary

Dynamic caching of data on edge servers based on the application query stream promises to be an adaptive caching solution with a low administrative overhead. We have implemented a prototype of a caching system which maintains previous query results in shared tables whenever possible. In this paper, we discuss the challenges of cache maintenance in such a dynamic environment, focusing on replacement issues in the presence of consistency guarantees. We describe a consistency protocol which propagates UIDs performed on the origin database over cached tables to the edge cache without first checking whether the new tuples match the cached query predicates. A “group replacement” mechanism operates in the background and removes all excess tuples from the cache. We discuss the challenges of devising a cache replacement policy for such a dynamic environment and propose insights into addressing these challenges.

References

What Are Real DTDs Like

Byron Choi
University of Pennsylvania
kkchoi@gradient.cis.upenn.edu

Abstract

DTDs have proved important in a variety of areas: transformations between XML and databases, XML publishing, consistency analysis of XML specifications, typechecking, and optimization of XML queries. Much of this work depends on certain assumptions about DTDs, e.g., the absence of recursion and nondeterminism. With this comes the need to justify these assumptions against DTDs in the real world. This paper surveys a number of DTDs collected from the Web, and provides statistics with respect to a variety of criteria commonly discussed in XML research.

1 Introduction

The first and most widely used method of describing the structure of an XML document is the Document Type Definition (DTD). It is part of the XML standard [3], and more sophisticated tools for structuring XML, such as XML-Schema [21, 1] are based on DTDs. Recently DTDs have been found useful in implementing efficient storage systems for XML [19] and in typechecking programming and query languages for XML [12, 15, 7]. Software such as XML storage systems may only work well when the structure of the DTD has certain properties. For this reason, as well as to see whether DTDs "make sense", we collected a number of DTDs and analyzed their structure. An earlier survey [18] on DTDs discusses some of their limitations but does not study the properties that we analyze in this paper.

Our statistics were collected by the “DTD Inquisitor” [5], a program that reads a DTD, attempts to identify problems with it, and computes a number of graph-theoretic properties of it. An online demo can be found at [5]. The statistics it collects fall into two categories: local – describing the kinds of content models found at individual element declarations, and global – describing the graph-theoretic structure of the DTDs and the documents that conform to them. For example: saying that there are no elements with mixed content is a local property, while saying that the maximum path length allowed by this DTD is 4 is a global property. Clearly these two properties are not entirely independent: a DTD that contains complex content models (a local property) is likely to allow a large number of distinct paths of a given depth (a global property).

1.1 The DTD Sample

The DTDs were recently extracted from the XML.org DTD repository [23]. We attempted to clean up all the DTDs in initial sample by hand; however some were too full of errors to allow us to do this. We were left with a total of 60 DTDs that formed the basis of our survey.

One may expect that the structure of a DTD depends on how it is intended to be used. For this reason the DTDs are divided into three broad categories: app, data and meta. DTDs that are primarily designed for data interchange between applications are called app, e.g. bookmarks exchange [17]. DTDs for data are those one might imagine easily putting in a structured database - baseball statistics [11], for example. Finally DTDs whose function is to describe the structure of document markup, such as that for Shakespeare’s plays [2], are termed meta. The boundaries between these three kinds of DTD are not clear. For example a BIOML DTD [20] which describes an annotation framework for biopolymer sequences might be put in more than one categories. They certainly describe data; they also are designed as a common syntax for exchanging the data between scientists. However, for the most part there was little doubt about the classification. The element and attribute names usually tell how DTDs are used. DTDs with names bold, italic or paragraph are likely to belong to the meta category. One can also tell from the structure of the DTDs – mixed content for example. Of our 60 DTDs, 7 were app, 13 were data and 40 were meta. A complete listing of the DTD sample can be found in [6].

1.2 The Statistics

The statistic are grouped into (1) local and (2) global properties.

- Local properties. In addition to noting the presence of mixed or any content, we extracted a number of
Table 1: The percentage and the number of the content models in the nine classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>app</th>
<th>data</th>
<th>meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdedata</td>
<td>13% (32)</td>
<td>58% (1147)</td>
<td>26% (685)</td>
</tr>
<tr>
<td>e</td>
<td>13% (32)</td>
<td>2% (33)</td>
<td>16% (415)</td>
</tr>
<tr>
<td>any</td>
<td>0% (0)</td>
<td>0% (4)</td>
<td>1% (26)</td>
</tr>
<tr>
<td>Mixed content</td>
<td>4% (9)</td>
<td>1% (21)</td>
<td>19% (488)</td>
</tr>
<tr>
<td>“</td>
<td>” only (not mixed)</td>
<td>22% (55)</td>
<td>2% (32)</td>
</tr>
<tr>
<td>“,” only</td>
<td>14% (34)</td>
<td>28% (557)</td>
<td>11% (296)</td>
</tr>
<tr>
<td>Complex content</td>
<td>19% (48)</td>
<td>7% (132)</td>
<td>7% (188)</td>
</tr>
<tr>
<td>List</td>
<td>13% (33)</td>
<td>2% (31)</td>
<td>6% (161)</td>
</tr>
<tr>
<td>Single</td>
<td>3% (7)</td>
<td>1% (29)</td>
<td>2% (63)</td>
</tr>
</tbody>
</table>

statistics concerning the structure and complexity of the content models involved in the DTDs. We also looked for properties that would affect the parsing of the DTD or its ambiguity in being used as a type [15, 13]

- **Global properties.** One can come up with a never-ending sequence of graph-theoretic properties to analyze. We chose some that might be important in the mapping of the XML into some database format. For example, if the presence of a Kleene star indicates the need for a table or some other collection type [19], one is interested in how deeply nested these can be. One is also interested in what kind of recursion can occur in DTDs. If one is interested in finding structure for optimization or type-checking as in Dataguides [10] one would like to know about the number and length of paths that are allowed by the DTD.

We next define some notation to be used throughout the paper. A DTD [3] can be described as a collection of element declarations of the form \( e \rightarrow \alpha \) where \( e \) is the element name (type) and \( \alpha \) is the content model. The content model is defined by: \( \alpha ::\leq \epsilon \mid \text{pdedata} \mid e \mid \alpha \cdot \alpha \mid \alpha | \alpha \mid \alpha^{*} \mid \alpha^{+} \mid \alpha^{?} \), where \( \epsilon \) denotes the empty content model, \( \text{pdedata} \) denotes string, \( e \) denotes an element name, “,” and “\|” stand for concatenation and union, and “\*\*”, “\*\*\*” and “\?” stand for zero or more, one or more and optional occurrences.

In the following sections we present the results of our analysis. Most of the figures in this paper are as follows unless otherwise specified. DTDs are grouped by their categories. Two vertical lines partition the visualization into three areas. The leftmost, middle and rightmost areas show the result from app DTDs, data DTDs and meta DTDs respectively. Each bar represents a DTD. The bars are sorted within each area for presentation purpose.

2 Local Properties

2.1 Content Model Classification

We classified the content model of the three categories. The classification of the content models is follow: (1) \( \text{pdedata} \), (2) \( e \), (3) \( \text{any} \), (4) \( \text{mixed content} \), (5) “\|” only but not mixed content, (6) “,” only content, (7) complex content, (8) list and (9) single. The first two classes are string and empty content. The class \( \text{any} \) does not impose any restriction on the subelements. Mixed contents are mixture of text and elements. For example, \( p \rightarrow (\text{bold}|\text{italic}|\text{pdedata})^{*} \) means a paragraph contains a mixture of normal, bold and italic text. The fifth class is union of elements, e.g., \( \text{dna} \rightarrow \text{address}\?, \text{city}\?, \text{country}\?. \) Complex content is a content model with both “,”s and “\|”s. List is the content model with one element name followed by a Kleene star or a plus. Single is the content model with one element name followed by an optional “\?”.

The breakdown is shown in table 1. The structure of the DTDs is closely related to their category. DTDs in the app category have diverse structures. However, we see a large number of data string and the tuple-like structures in data DTD. The meta DTDs are dominated by the data string and the mixed content.

Note that the presence of \( \text{any} \) content makes it impossible to measure some graph-theoretic properties in DTD. Thus we ignored such content in our analysis.

2.2 Syntactic Complexity

The syntax for XML DTD content model allows one to write arbitrarily complex expressions. We investigate the complexity of content model in this section. A depth function is defined as a rough measure of the content model complexity.

**Definition 1.** The inductive definition of the depth of the content model:

\( \text{depth}(\epsilon) = 0; \text{depth}(e) = 1; \)
2.4 Ambiguity

We follow the definition of ambiguity in [4]. An expression $R$ is ambiguous if and only if there exists some string $s$ in $L(R)$ such that there can be distinct ways to parse string $s$. It is straightforward to show that any ambiguous content model is non-deterministic.

Very often, a data source exports its data in XML format and the receiver maps the XML into some objects [17]. Ambiguity in DTDs means that the mapping is not unique. An ambiguous content model from a DTD for open application [16] is shown below:

$$\text{partner} \rightarrow (\text{name}^?, \text{onetime}^?, \text{partnrid}^?, \text{partntype}^?, \text{syncind}^?, \text{active}^?, \text{currency}^?, \text{descriptn}^?, \text{dunsnrnum}^?, \text{glentitys}^?, \text{name}^*, \text{parentid}^?, \text{partnridz}^?, \text{partnrateq}^?, \text{partnrrole}^?, \text{paymethod}^?, \text{taxexempt}^?, \text{taxid}^?, \text{termid}^?, \text{userarea}^*, \text{address}^*, \text{contacts}^*)$$

Consider the underlined element names. When the XML document is streamed over a network, the partner element arrived followed by a name element. The receiver cannot decide if it is the first optional name or the sequence of names. Mapping it to either name can form a parse. The Inquisitor detected 2 ambiguous content models.

3 Global Properties

3.1 Reachability

The definition of reachable element name is given below.

**Definition 2.** An element name $e'$ is reachable from $e$, denoted by $e \Rightarrow e'$, if either $e \rightarrow \alpha$ and $e'$ occurs in $\alpha$, or $e \Rightarrow e''$ and $e'' \Rightarrow e'$.

Similarly, a content model $\alpha$ is derivable from an element name $e$, denoted by $e \Rightarrow \alpha$, if either $e \rightarrow \alpha$, or $e \Rightarrow \alpha''$, and $\alpha = \alpha'[e'/\alpha'']$, where $\alpha'[e'/\alpha'']$ denotes the content model obtained by substituting $\alpha''$ for all occurrences of $e'$ in $\alpha$.

**Definition 3.** An element name $e$ is reachable if $r \Rightarrow e$, where $r$ is the name of the root element. Otherwise element name $e$ is called unreachable or useless.

We assume that the root of the sample DTD is not known, i.e., all element names can be the root element name. Figure 2 shows the number of unreachable element names in DTDs. These unreachable element names are either the root element name or useless. The mode of the number of unreachable element name is 1. Thus DTDs very often have a clear notion of the root element. An app DTD with 20 unreachable element name encodes an application, and different kinds of small request and response messages in XML syntax. Each of these small messages are not reachable from all other element names.

2.3 Determinism

The XML Standard [3] defines deterministic content models to be those do not require look ahead when parsing. Non-deterministic content model is not allowed in XML DTDs [3]. Certain typed XML query language implementation [7] assumes determinism. An example of non-deterministic content model is $(a, b | a, c)$ and that of deterministic content model is $(a, (b | c))$. The Inquisitor detects 5 non-deterministic content models and they are found in 4 DTDs. For example, the following non-deterministic content model is found in the DTD from workflow management coalition [22], $WF\_XML \rightarrow (\text{request} | (\text{request}, \text{response})).$
Separating the unreachable parts in DTD into smaller DTDs appear to be a better design.

3.2 Recursions

We define recursive DTDs and two kinds of recursions as follow.

Definition 4. A DTD is recursive if and only if it has an element name e such that e ⇒ e and e is reachable.

Definition 5. A DTD is linear recursive if and only if it is recursive and for any reachable element name e and any e ⇒ α, e occurs at most once in α and the occurrence is not enclosed in "*" or "+". A DTD is said to be non-linear recursive if it is recursive but is not linear recursive.

Consider an example of a non-linear recursive element DTD [17]. folder → title?, info?, desc?, (bookmark|folder|alias|separator)*. A folder has some optional information and, among other things, a list of folders. folder does not occur in bookmark, alias and separator. However, folder is enclosed in a "*".

No linear recursive DTD is found in our sample. The sample contains 7, 2 and 26 non-linear recursive DTDs in the app, data and meta category respectively. The analysis shows that there are a few (25) non-recursive DTDs in our sample. data DTDs are usually non-recursive.

3.3 Simple Path and Simple Cycle

For the non-recursive DTDs, we investigate their longest simple path. The result is shown in figure 3. The leftmost area is empty since all DTDs in the first category are recursive. The result shows that the longest path in such DTDs is not too long (mostly less than 8). The DTD for exchanging financial data [14] allows the longest simple path with the length 20. The structure in the DTD is dominated by tuples and strings. We do not have an explanation on permitting such a long path in this DTD.

We also investigate the number of simple paths in non-recursive DTDs and the number of simple cycles in recursive DTDs. Simple cycle is a path in the form e₁, e₂, ... ek, e₁, where e₁, e₂, ... ek are distinct element names. The range of the number of simple paths in the DTD sample is wide. For the number is larger than 512, we treat it as "too many" and do not show the entire bar. Figure 4 shows the number of simple path allowed by non-recursive DTDs. DTDs appear either allowing few or a large number of simple paths.

The number of simple cycle allowed by recursive DTDs is shown in figure 5. For the number is larger than 50, we do not show the entire bar but the actual value is shown on top of it. Most DTDs with large number of simple cycles are designed for document markup. The short bars in meta category are the DTDs for marking up small messages. Recursive data and app DTDs have small number of simple cycles, with some exceptions.

3.4 Chain of Stars

A real example [9] of a chain of 2 stars is follow: entity → name*, contact*, location*, phone*, fax*, email* and location → address*, city?, region?, postal?, country?. A business entity has a list of locations. Each location has some address lines. A star/plus in a cycle leads to a chain of stars with infinite length. Such chains are not included in this analysis. The result of this analysis is shown in figure 6. The longest chain of stars is often
small (the mode is 3). One star-free non-recursive DTD is detected. The extreme case is found in the DTD for exchanging financial data mentioned in section 3.3.

3.5 Hubs

Fan-in of an element name, e, is the cardinality of the set \( \{ e' | e' \rightarrow \alpha' \text{ and } e \text{ occurs in } \alpha' \} \). An element name with a large fan-in value is called hub. We present the fan-in of the elements in data and meta in figure 7 and figure 8 respectively. We skip the result for the DTDs for app DTDs for space constraint. Each line in the figure represents a DTD. A cardinal number (element name ID) is assigned to each element name. The fan-in values are sorted in descending order (all lines are monotonically decreasing). For DTD with few element names, the line is short. The figure shows the largest 52 fan-in values. Our survey shows that hubs exist in DTDs in all categories. In some cases, documents contain many elements whose name is a hub. Fast access to hubs can be in high priority.

An observation is that many content models are exactly the same, a prefix or a suffix of another. These content models cause some horizontal lines at high fan-in values.

4 Conclusion

Some XML storage system, typechecking and query language depend on certain assumptions on DTDs. We collected real DTDs and analyzed the structures that may be assumed in XML research. The paper provides statistics on some structures of real DTDs.

Acknowledgement. Thanks to Peter Buneman and Wenfei Fan for their invaluable comment and encouragement. The author would also like to thank many members of Penn Database Research Group for suggestions on this paper.

References


ToXgene: An extensible template-based data generator for XML*

Denilson Barbosa¹  Alberto O. Mendelzon¹  John Keenleyside²  Kelly Lyons²

¹ Department of Computer Science
University of Toronto
{dmb,mendel}@db.toronto.edu

² IBM Toronto Lab
{keenley,klyons}@ca.ibm.com

Abstract. Synthetic collections of XML documents are useful in many applications, such as benchmarking (e.g., Xmark), and algorithm testing and evaluation. We present ToXgene, a template-based generator for large, consistent collections of synthetic XML documents. Templates are annotated XML Schema specifications describing both the structure and the content of the data to be generated. Our tool was designed to be declarative, and general enough to generate complex XML content and to capture most common requirements, such as those embodied in current benchmarks. In the paper, we give an overview of the ToXgene template specification language and the extensibility of our tool; we also report preliminary experiments with ToXgene carried out at the IBM Toronto Lab, which show that our tool can closely reproduce the data sets of the TPC-H and Xmark benchmarks.

1 Introduction

Synthetic collections of XML documents have many applications in benchmarking and testing algorithms, tools and systems. Moreover, different applications require documents with different complexities, sizes, etc. For example, a benchmark for data-intensive applications might be a large and relatively homogeneous document, with many references among elements, while a test suite for a parser might be an heterogeneous collection with thousand of documents. Given the complexity of writing and/or customizing hard-coded data generators for specific scenarios, we believe a declarative tool for generating synthetic XML documents will prove useful.

We present ToXgene, a template-based tool for generating large, consistent synthetic collections of complex XML documents. ToXgene’s template specification language is a subset of the XML Schema [5] notation augmented with annotations for specifying other properties of the intended data, such as probability of occurrences of elements, the vocabulary of CDATA content, etc. This work is part of the ToX project [2], recently started at the University of Toronto.

1.1 Related work

A general purpose generator for synthetic XML documents is presented in [1]. Our work differs from that in the following ways. First, our data generation process is centered on a conceptual description of the desired data (a template). ToXgene gives the user total control over the data to be generated, unlike the method in [1], where both the structure and the content of the documents are randomly generated. We note that our tool allows some controlled randomness in its output and can generate documents with irregular structure, as shown in Section 2.4. Second, our tool generates more complex XML content, including elements with mixed content; attributes; non-gibberish text; and different numerical and date values. Finally, ToXgene supports different probability distributions.

Another template-based XML data generator is the IBM XML Generator [7]; its inputs are annotated Document Type Definitions (DTDs) specifying the structure and the characteristics of the data. There are many limitations to that tool, however. For instance, it allows one to limit the maximum depth of the document tree, or the number of ID and IDREF attributes in the documents, but it does not allow one to specify actual values for these properties. Moreover, unlike ToXgene, that tool does not support different probabilities of occurrence on a per element basis, nor the generation of CDATA content of different datatypes (e.g., strings, dates, etc.). On the other hand, the IBM generator deals with XML constructs that are not addressed in ToXgene, such as ENTITY declarations and processing instructions [4].

Furthermore, ToXgene differs from both approaches above in the following ways. First, ToXgene supports element sharing, i.e., the values of some elements (or attributes) can be shared among different elements (or attributes) in the same (or in different) XML documents. This allows the generation of collections of correlated documents (i.e., documents that can be joined by value). Second, ToXgene can produce data conforming to user-specified integrity constraints, which allows the generation of consistent references within and across documents. Third, ToXgene allows the use of existing data when generating documents; therefore, one can grow an existing collection of documents while maintaining its consistency, or mix real and

*This work is supported by the National Science and Engineering Research Council of Canada, Bell University Laboratories, and the IBM Centre for Advanced Studies.
tox-distribution name="c1" type="exponential" minInclusive="5" maxInclusive="100" mean="35">

(a) Specification of an exponential probability distribution.

<complexType name="my_book">
  <attribute name="isbn" type="isbn_type"/>
  <element name="title" type="string">
    <tox-string type="text" maxLength="30"/>
  </element>
  <element name="author" type="my_author" minOccurs="1" maxOccurs="5"/>
  <element name="price">
    <complexType mixed="true">
      <attribute name="currency" type="currency">  
        <simpleType>
          <restriction base="string">
            <tox-expr value="'CDN'"/>
          </restriction>
        </simpleType>
        <tox-number minInclusive="5" maxInclusive="100"/>
      </complexType>
    </element>
  </complexType>
</complexType>

(b) Specification of a complexType.

Figure 1: Notation for specifying types, genes and probability distributions in ToXgene.

2 ToXgene template specification language

Among the various languages for specifying XML content, we chose XML Schema as the basis for our template language for two reasons: it is a W3C standard, thus it is expected to become familiar to XML practitioners; and it allows a more detailed description of XML content than DTDs. In particular, it allows the specification of types, for describing literals (i.e., CDATA content), elements and attributes. However, we note that having an XML Schema specification alone is not enough for generating useful synthetic data; at the very least, we need to annotate the schema for specifying which type should be used as the type for the root element of the documents to be generated.

We also define annotations for other purposes, such as specifying probability distributions of occurrences of elements and attributes; defining element sharing; defining integrity constraints over XML elements; and providing some controlled randomness in the structure of the data to be generated. More details on the ToXgene template language can be found in [9].

Types and genes. A type is a specification of some valid XML content, and can be either a simpleType or a complexType [5]; instances of a simpleType are CDATA literals, while instances of a complexType are either XML elements, CDATA literals, or a mix of both.

A gene is a specification of either an element (an element gene) or an attribute (an attribute gene), and contains a name and a type; an instance of an element gene with name n and type t is an element whose opening and closing tags are labeled with n and whose content is an instance of t; similarly, an instance of an attribute gene with name n and type t is an attribute whose name is n and whose content is an instance of t. An attribute gene can only be declared within a complexType, and its type must be a simpleType.

2.1 Specifying probability distributions, types and genes

Figure 1 contains the examples we use to illustrate the discussion in this section. All ToXgene-specific annotations are prefixed by the keyword tox and presented in a different font.

Specifying probability distributions. A probability distribution is declared using the tox-distribution annotation, as shown in Figure 1(a), and referenced via its name, as shown in Figure 1(b). A single template might specify multiple probability distributions, which can be used to determine, for example, the number of occurrence of elements and attributes, the length of string literals, and instances of numerical types, as in Figure 1(b). ToXgene currently supports the uniform, normal, exponential and log-normal distributions, as well as arbitrary discrete distributions, where the user provides all possible outcomes together with their respective probabilities of occurrence. In order to allow the static checking of the consistency of the templates,
we require each probability distribution to have a maximum and a minimum value; all values outside this interval have null probability of occurrence.

**Specifying simpleTypes.** A simpleType is a specialization of a base type, e.g., string, integer, etc.; the domain of a simpleType is a subset of the domain of the base type it is built upon; and an instance of a simpleType is an element chosen from its domain. For example, the isbn type, specified in Figure 1(b), is a specialization of the string base type; its domain is the set of strings that conform to the given pattern; and 1234567890 is one of its instances. simpleTypes are the basic content specification tools for defining genes.

The tox-number, tox-string and tox-date annotations are used to refine a SimpleType definition (e.g., to specify that instances of my_float in Figure 1(b) obey the probability distribution c1, and that the content of instances of the title element in Figure 1(c) are non-gibberish strings\(^1\), no more than 30 characters long), or to specify the generation of elements with mixed content (e.g., the price element in Figure 1(c)). Constants can be specified using the tox-expr annotation.

**Specifying complexTypes and genes.** A complexType specification (see Figure 1(c)) contains definitions of element genes, attribute genes, or annotations defining CDATA literals, required for defining elements of mixed content. Both named and anonymous types are allowed in ToXgene, as shown in the figure. Our tool supports all element content models defined in [4]: character data, as in the isbn element; elements, as in the book element; and mixed, as in the price element.

### 2.2 Specifying element sharing and integrity constraints in ToXgene

ToXgene allows element sharing both within and across documents; i.e., different elements (or attributes), in the same or in different documents, can have the same CDATA content. This allows the generation of collections of correlated documents (i.e., documents that can be joined by value). In our tool, element sharing is achieved by generating all shared content prior to generating any documents. The shared content is kept in what we call tox-lists; such lists are queried at document generation time. Integrity constraints over the contents of a list, such as uniqueness of certain values, can be specified, thus ensuring the consistency of the collections.

**Specifying lists.** Lists are declared by tox-list annotations. Each list has a unique name, an element gene that defines its contents, and, optionally, some integrity constraints. Figure 2(a) shows the specification of a list of books, each containing an ISBN and up to 5 author_id values, sampled (without repetition) from a list of authors, declared elsewhere.

**Querying lists.** ToXgene has a language for specifying expressions which are declared using tox-expr annotations, and are evaluated at content generation time. This language allows arithmetic and string operations; operands can be the results of queries, instances of simpleTypes generated “on-the-fly”, or constants. All expressions are typed; type checking and casting mechanisms are implemented. Expressions are useful for defining conditional instantiation of genes (see Section 2.4), and for specifying where clauses for lists and selection conditions on cursors (see below). Due to space limitations, we describe expressions defining queries only; for details, see [9].

The contents of a list are retrieved using cursors and queries, both of which are specified using path expressions. A path

---

\(^1\)Our tool supports two built-in string types: gibberish, and text, as defined in [10]. However, ToXgene can produce strings using a different vocabulary, as discussed in Section 4.
ToXgene also supports the generation of recursive XML content, determined by three parameters: the total number of (recursive) elements to be generated, the number of children per element, and the number of levels in the recursion. Any of the supported probability distributions can be used for specifying these parameters.

2.4 Specifying irregular structures and recursive elements

Templates can specify control statements that determine, at content generation time, which genes are instantiated. Each control statement contains one or more blocks of genes; at processing time, at most one such block is chosen, and only the genes in that block are instantiated. ToXgene provides the customary IF-THEN-ELSE statement and also a “lottery” statement, in which a block is randomly chosen according to a probability distribution. Control statements can be nested arbitrarily. Also, the blocks in a control statement might contain completely different genes; note that this provides a way of specifying some controlled randomness in the structure of the XML documents produced by ToXgene.

Recursion in element genes. ToXgene also supports the generation of recursive XML content, determined by three parameters: the total number of (recursive) elements to be generated, the number of children per element, and the number of levels in the recursion. Any of the supported probability distributions can be used for specifying these parameters.

---

Note that ISBN values are unique in book_list.
was validated against the DTD provided by the benchmark authors without problems. For Xmark, we generated a 100Mb document (i.e., we used a scaling factor of 1). Our document size, corresponding to 400Mb of XML data. These documents were loaded into a relational database in DB2 using 4 Experimental results

Table 1: Comparison between ToXgene and other data generators. The metric used for part (a) is number of tuples in the result set; the metric for part (b) is the length (in lines) of Kweelt’s output. All ratios are computed by dividing the measurement with ToXgene’s data by the corresponding measurement with the data from other data generator.

3 Extending ToXgene

ToXgene is a general purpose tool, and as such provides built-in tools for generating XML content conforming to the most common datatypes. Although preliminary experimentation with our tool shows it can easily reproduce the synthetic data used in complex benchmarks, using a combination of queries and string operations, ToXgene was designed to support user-defined CDATA generators (e.g., a generator for pseudo-random DNA sequences).

ToXgene was implemented in Java 2, and its architecture was designed in a way that Java classes implementing CDATA generators could be added with little effort: all that is required is registering the new code in the Gene Factory module; of course, this new code has to implement the interface defined in our code, which consists of a single method. In principle, the code that produces instances of simpleTypes is not supposed to produce XML elements (i.e., strings containing element tags). However, there is nothing in our code that enforces this behavior. Thus, if one needed XML elements with random tag names, it would suffice to encapsulate a data generator such as [1] as a new simpleType in our tool. ToXgene can also be coupled with external tools, since it has the ability of storing and reading lists from files.

4 Experimental results

Our goal with ToXgene was the generation of “useful” synthetic XML documents. To test our tool, we tried to reproduce the data produced by the generators of known benchmarks; we chose the generators of TPC-H [10] (dbgen) and Xmark [8] (xmlgen). We decided to use TPC-H because it is a widely used relational benchmark and because it defines many non-trivial integrity constraints over its database. Xmark was chosen because it was designed specifically for XML, and specifies a complex document, with different levels of nesting, and thousands of references among its elements.

Experimental setting. For the TPC-H data set, we generated data for a 100Mb relational database (i.e., we used a scaling factor of 0.1), corresponding to 400Mb of XML data. These documents were loaded into a relational database in DB2 using IBM’s XML Extender [6]; the schema of this database was identical to the one populated by dbgen. No integrity constraint was violated by our data. For Xmark, we generated a 100Mb document (i.e., we used a scaling factor of 1). Our document size was validated against the DTD provided by the benchmark authors without problems.

For each benchmarking data set, we ran both queries from the benchmark workload and some ad-hoc queries to test specific details of the data generated. We used Kweelt\(^3\) to run these queries on the Xmark data sets.

All experiments were run on a 4-way 500 Mhz Pentium III machine with 3Gb of RAM and 18Gb of disk storage. The data generation takes about 1 hour for TPC-H, and about 25 minutes for Xmark. The memory required for storing the lists are 750Mb and 250Mb of RAM, respectively.

Experimental results. Table 1 shows the results of some of the queries selected from the workloads of the benchmarks. We show the size of the result sets of the queries, and the relative times to execute these queries on each data set. We show queries of different result set sizes, both with fixed (e.g., “find the best n customers”) or variable result set sizes. As one can see, the results are comparable.

The ad-hoc queries were designed to compare the distributions of values between the data sets. Table 2 shows the result of some of these queries. All data values in TPC-H are generated according to uniform distributions; this is not the case for Xmark: for instance, the average prices of items in closed auctions obey an exponential distribution, while the number of items that can be bought with a credit card obey a uniform distribution. Again, the results are comparable.

Generating Xmark text. Most of the textual content in Xmark is generated by sampling from the 17000 most common words in Shakespeare’s plays, according to their frequency of occurrence. We obtain similar content by loading the list of words used by xmlgen into a list, which is sampled accordingly. Sentences are formed by concatenating these words. We generate the final text by copying these sentences into emph, bold and keyword elements in a recursive fashion, so that we can have bold sentences nested within emph sentences, etc.

\(^3\)Available at http://http://kweelt.sourceforge.net/.

<table>
<thead>
<tr>
<th>Query</th>
<th>sizes</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToXgene</td>
<td>dbgen</td>
<td>size</td>
</tr>
<tr>
<td>q14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>q3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>q2</td>
<td>54</td>
<td>44</td>
</tr>
<tr>
<td>q16</td>
<td>2789</td>
<td>2762</td>
</tr>
</tbody>
</table>

(a) TPC-H results.

<table>
<thead>
<tr>
<th>Query</th>
<th>sizes</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToXgene</td>
<td>xmlgen</td>
<td>size</td>
</tr>
<tr>
<td>q1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>q11</td>
<td>7661</td>
<td>7661</td>
</tr>
<tr>
<td>q12</td>
<td>1358</td>
<td>1436</td>
</tr>
<tr>
<td>q17</td>
<td>25595</td>
<td>25535</td>
</tr>
</tbody>
</table>

(b) Xmark results.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Target</th>
<th>ToXgene</th>
<th>dbgen</th>
</tr>
</thead>
<tbody>
<tr>
<td>average balance on customer accounts</td>
<td>4500</td>
<td>4510.06</td>
<td>4470.50</td>
</tr>
<tr>
<td>average size of parts</td>
<td>25</td>
<td>25.00</td>
<td>25.00</td>
</tr>
<tr>
<td>average supply cost of parts</td>
<td>5000</td>
<td>499.74</td>
<td>499.69</td>
</tr>
<tr>
<td>average tax per item in each order</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
</tbody>
</table>

(a) TPC-H results.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Target</th>
<th>ToXgene</th>
<th>xmlgen</th>
</tr>
</thead>
<tbody>
<tr>
<td>average price in closed auctions</td>
<td>100.00</td>
<td>100.58</td>
<td>96.83</td>
</tr>
<tr>
<td>average “happiness” of customers with closed auctions</td>
<td>5.50</td>
<td>5.48</td>
<td>5.43</td>
</tr>
<tr>
<td>percentage of items in Europe that have “United States” as location</td>
<td>0.75</td>
<td>0.7553</td>
<td>0.7447</td>
</tr>
<tr>
<td>percentage of items in Asia that can be purchased with credit card</td>
<td>0.50</td>
<td>0.5020</td>
<td>0.4945</td>
</tr>
</tbody>
</table>

(b) Xmark results.

Table 2: Comparison between ToXgene and the other data generators, using the ad-hoc queries. The target value on the tables is the expected value for the corresponding feature, given the probability distribution specified by the benchmark.

5 Conclusions and future work

In this paper we introduced ToXgene: an extensible template-based generator for consistent collections of correlated synthetic XML documents. We presented an overview of our template specification language; we discussed some of the novel features of our tool, such as element sharing, and described how ToXgene can be extended or coupled with other tools. Finally, we reported on preliminary experiments we conducted with our tool.

Development of ToXgene is continuing. We intend to provide a more general mechanism to allow the generation of text according to different vocabularies, grammars, and character encoding schemes. This would be of capital importance for generating testing data for text-intensive applications. We are also working on improving ToXgene’s performance; in particular, we are interested in exploiting possible parallelism in the data generation.

As future work, we identify the extraction of ToXgene templates from existing documents as an interesting problem. The obvious application of this research would be generating synthetic data closely reproducing the characteristics of real documents. This would be especially important when inspecting the data is unfeasible or impossible (e.g., for security reasons). Also, templates capture considerable information about the data they describe, and thus are valuable metadata in themselves.

References


