Integrating XPath with SQL

I.J. Davis

Department of Computer Science,
University of Waterloo,
Waterloo, Ontario,
Canada N2L 3G1

ABSTRACT

The leading candidate for searching and retrieving fragments of XML text is XQuery. It is proposed that minor extensions to the SQL language might instead be employed to facilitate searching and retrieval of structured texts. This alternative approach has numerous advantages, since it builds upon existing database technology, and allows a tight integration of structured and semi-structured data, under a powerful and well-established query language.

Enhancements to the XPath pattern matching language are proposed, that facilitate the melding of relational database and structural queries, while offering the potential of being rapidly resolved by existing text engines.

1. Introduction

In 1994 pre-competitive research began on how best to integrate structured text with relational and object oriented databases. That research led to the development of a rudimentary structured text pattern matching language, and prototype hybrid query software that served as a proof of concept for proposals submitted to the SQL/MM standards committee [11]. Funding for this research terminated in 1997.

The results of this pre-competitive research were encouraging [6,7], and led to the formation of a company (www.textserver.com) whose objectives were to develop commercial software that allows efficient and effective searching and retrieval of structured text within the existing SQL2 paradigm.

This software can be broadly divided into five modules. First, an XML [5] parser was developed that parsed XML documents and allowed these documents to be suitably indexed. Then appropriate data structures and a core text engine were implemented, that allowed rapid retrieval of regions of text that matched specified criteria as defined in an XPath expression [4]. Then functional extensions to SQL [1] were developed that allowed effective access to structured text. In addition, an efficient SQL query engine was written, capable of supporting all conventional SQL DDL and DML. Finally, OLE/DB interfaces were developed and validated that allowed the resulting database software to be deployed in the desired environments [12].
This paper describes the extensions made to both XPath and SQL to facilitate a tight integration of XPath with SQL.

2. A model for text

In accordance with the XPath model, a structured text is viewed as a collection of nodes, contained within a partially ordered tree. This partially ordered tree corresponds to the structure of the text imposed by the textual markup. Each node is assigned a unique preorder node traversal number computed when the underlying text is parsed. It is this node number that internally identifies a given node [8]. This model is extended by considering individual words, occurring within attribute values, spanned text, comments, etc. to also be represented by logical nodes within the tree structure.

SQL is extended to support a new text datatype. An individual value of this type has three intrinsic properties associated with it.

- It has some provenance, identifying the base document from which this fragment of text is obtained, as well as ancillary information such as the document’s grammar, location, indices derived from the base document, etc.

- It spans some region of text. The region spanned is represented internally by a start and end node number. The text subsumed by a region begins at the text subsumed by the start node number, and includes all text up to and including the text subsumed by the end node number. While it will often be the case that the text thus subsumed will correspond to the text subsumed by a specific node within the text hierarchy, this is not a requirement within our model.

- Each text value may have associated with it a list of zero or more regions of text having the same provenance as this text value. Individual regions of text within this list are ordered in ascending order by the node number at which they start, and within this ordering in descending order by the node number at which they end. Duplicated regions appear as a single region within this list. Each member of this list is referred to as a mark.

3. XPath

The evaluation of an XPath expression is performed in one or more steps. Each step takes as its input a set of regions, and uses these to compute a new resulting set of regions. The input to the first step is a provided region of text, and the input to subsequent steps is the set of regions computed during the prior step. The result of an XPath expression is the set of regions computed during the final step.

At each step, regions are identified which have some structural relationship to the prior set of regions. These relationships are called axes. Thus the sequence of axes specified
in an XPath expression describes the legitimate routes than may be taken from the provided input region, to regions that may be included in the result.

At each step the result may be further constrained so that selected regions span specific types of information, contain specific textual content, and/or satisfy each of zero or more sequentially applied predicates. Such predicates may themselves include expressions within them that refer back to the individual regions of text being filtered by these predicates.

4. XPath limitations

The process of resolving an XPath expression described above may appear to be a straightforward sequential exercise, but in practice there are often optimizations that can be employed to speed this task [3], particularly if text indices have been created for the texts being operated upon.

XPath needs to maximize the ability to describe precisely what is to be recovered from given input texts in a non-procedural manner [13], while minimizing the need to procedurally state how this desired data is to be recovered. Unfortunately, XPath as currently formulated does exactly the reverse. If one is to recover textual information according to a variety of related criteria, one must do this by utilizing multiple XPath expressions; explicitly specifying as distinct computations how each relevant piece of information is to be obtained.

For example, if one wished to establish where a particular phrase occurred in the collected works of William Shakespeare, it is very likely that the title of works where this phrase occurs, the alternative title if there was one, and the speeches containing such phrases would each need to be reported, with the relevant phrase somehow highlighted within each such speech. One would also want to report the act and scene numbers associated with each such speech, and the identity of the speaker.

While one could procedurally compute the set of speeches containing the indicated phrase and then for each such speech recover the ancillary information associated with it, this approach is far from optimal. Inefficiencies occur, because the recovery of ancillary information is performed repeatedly for each speech found, rather than in a single operation for all speeches (which may well be possible given the availability of text indices), and because constraints on the set of speeches to be recovered (for example that they must occur in plays rather than in sonnets) are not necessarily fully specified prior to the recovery of such speeches.

What is required are extensions to the XPath language which allow one to provide a compact description of most, if not all of the information that is to be recovered from a text, with the desired results being returned not as a single set of regions, but as a table which preserves the understanding of the relationships between the various types of information obtained. Essentially we wish to extend the XPath language so that the joins
that must be performed in constructing appropriate tables are implicitly specified within the XPath language, rather than mechanically computed outside of the XPath language, in a fashion largely dictated by the requests of a naïve user.

5. Extending XPath to encode relationships

The first type of relationship that we wish to preserve when evaluating an XPath expression is that regions identified during any step are related to specific regions computed during other steps, according to the specified axis (or sequence of axes) between these steps. For example, if speeches to be recovered must occur within scenes, then it is may be desirable to recover the relationship between a speech, and the scene in which it occurs. Often such relationships will be one to many, but more general relationships may exist. For example the following axis induces a many to many relationship, because a given region of text may follow many alternative regions of text computed during the prior step. Even the ancestor/descendant relationship, may have multiple regions selected during a prior step, each of which is a legitimate ancestor of some region selected during the current step.

In order to recover such relations the semantics of how an XPath expression is evaluated is extended. The initial evaluation of an XPath expression remains unchanged. Then for each internal step in the XPath expression the set of regions associated with this step is reduced by eliminating any region which fails to have at least one remaining region in the following step that enjoys the desired relationship with it, as specified by the next step’s axis. This reduction continues until no further reduction is possible.

An optional OUTER attribute may be associated with any step. The OUTER attribute indicates that regions identified by the prior step within the XPath expression should be preserved, even if no region within the current step can be found that has all of the requested properties associated with this step, the desired relationship with subsequent steps, and the appropriate relationship with the prior step. This attribute thus indicates that this and subsequent steps within an XPath expression are optional, and impose no restrictions on the regions recovered during the prior step.

To facilitate the preservation and recovery of the relationships between regions computed in distinct steps, a positive integer attribute may also be associated with any step of an XPath expression. This attribute is ignored when associated with XPath steps occurring within XPath filtering predicates, or in cases where relations are not being computed. The value of this attribute identifies the column within a relational table that is to be employed when returning regions computed during the associated step. This table will have one tuple for every combination of regions obtained from distinct steps, which are related to each other according to the sequence of axes specified in the XPath expression.

Each such tuple will contain text values in the specified columns, having the same provenance as the input text on which the XPath expression operates, spanning the desired text region. By default these text values will contain no marks.
In the case where only one column is to be recovered the set of text values returned within this column correspond to those spanning the set of regions associated with the specified step.

A second type of relationship that often arises in text is that distinct regions of text are related, not because either is directly computed or derived from the other region by using an XPath axis, but because both are derived from some common prior region. For example acts in Shakespeare contain both an act number and a sequence of scenes, but there is no direct containment relationship between scenes and act numbers.

To request recovery of such things as act numbers, scene numbers, titles, alternative titles etc, when identifying speeches containing specific phrases, the syntax for XPath is extended to allow specification of patterns matching not just a single path derived from the original input text through the specification of axes, but trees derived from multiple paths, potentially sharing initial subpaths. Such patterns match trees, rather than paths.

The final step of an XPath expression may be replaced by a bracketed sequence of XPath expressions, each separated from the next by the keyword AND. The semantics of such an expression is that each XPath expression within the bracketed sequence of expressions is to be independently computed using the set of regions computed during the prior step, and that a prior region is to be considered related to the subsequent bracketed expression if and only if it is suitably related to at least one region computed by the first step of each subordinate XPath expression. Such subsequent regions are to be preserved only while they remain in the desired relationship to at least one preserved region in the prior step. The OUTER attribute may be employed to indicate cases where prior regions need match no region within a specific subordinate XPath expression.

For example, to obtain in the format of a relational table, all relevant information for speeches in Shakespeare that contain the phrase ‘friends, romans’ one might employ the following extended XPath expression:

```
//play/(1::id AND /2::title AND /6::outer::title2 AND /act/(/3::actno AND /scene/(/4::sceneno AND /5::speech//contains(‘friends, romans’))))
```

Such an expression would returns the phrase ‘friends, romans’ occurring within a speech, within a scene, within an act, within a play by Shakespeare. In addition to the recovery of such speeches, the scene number, act number, and title of plays containing such speeches would also be returned, together with the identifier of such plays, and any optional second title. The scene numbers returned are those that occur in the same scene as the speech, the act numbers are those that occur in the same act, etc. It is hoped that the notation is straight forward, and the desired result correspondingly clear.

A third type of relationship sometimes arises in which information from differently structured text must be collectively recovered. Consider for example the case where we wish to find phrases occurring either in plays or in sonnets written by Shakespeare. At
some point in such XPath expressions we wish to indicate not that a step must have a relationship with all subsequent expressions as shown above, but with any of the subsequent XPath expressions. This is done in a fashion similar to that shown above, by employing the keyword ‘OR’.

For example, we might use the XPath expression

\[
/\text{work}/(/\text{sonnet}/(/2::title \text{ and } 5:/\text{verse}///\text{contains}('friends,romans')) \text{ OR } /\text{play}/(…)
\]

to augment the information recovered from plays with the corresponding information derived from the sonnets of Shakespeare. The works of interest are those that contain the desired sonnets, and/or the desired plays. The resulting relation is formed by logically performing a ‘union all’ operation on the relations derived from each of the or’d terms.

The degree of the table returned corresponds to the maximum column number specified in the XPath expression used in constructing this table. All unresolved values in the resulting relation are assigned the null value.

6. Extending XPath to mark and return regions of text

The XPath standard indicates that the \text{CONTAINS}, \text{STARTS-WITH}, \text{LIKE} and \text{STARTS-LIKE} node tests behave as predicates that return true if the nodes these functions are applied against contain the required textual content. Rather than treat these functions as predicates we consider them to be functions that compute sets of regions, with each such region being the smallest region containing the desired sequence of words.

Thus: \( /1::\text{speech}//2::\text{contains}('friends, romans') \) produces a relation containing two columns. The first column contains texts whose regions correspond to speeches that contain the word \text{friends} followed by the word \text{romans}, while the second column will contain texts that span the precise region (or regions) which while occurring within this speech have the property that they start with the word \text{friends} and end with the word \text{romans}. Note that the speech will be duplicated multiple times if the contains predicate can be satisfied for this given speech in multiple ways.

A second optional integer parameter may be provided with each of the above functions. This integer value specifies the maximum number of nodes that are allowed to be included within regions containing specified text, while not matching any word of this text. Thus \text{contains}('friends, romans', 0) indicates that node corresponding to the word \text{romans} must immediately follow the node associated with the word \text{friends}.

Rather than extract those phrases within a speech that correspond to a given pattern, we may instead elect to include such phrases as marked text within the selected speeches. A \text{MARK} attribute, may be associated with any XPath step, and indicates how marks are to be formed. Any extracted text contains as marks those regions of text derived from later XPath steps, that are in the correct relation to the region spanned by this text, and which are derived by an XPath step having the \text{MARK} attribute.
Thus \(/1::\text{speech}/2::\text{mark}::\text{contains}('friends, romans')/\)

produces the same table as the previous expression, but texts in the first column have regions corresponding to selected speeches and marks within them corresponding to all of the smallest sub regions beginning with friends and ending with romans. Marks may be counted, manipulated, or employed to highlight for display purposes those regions of text that resulted in specific speeches being recovered.

To allow marks occurring within an input text to be of relevance in an XPath expression a new \(\text{mark}()\) node test is defined. This test when applied to a given set of regions occurring within a single XPath step preserves those regions that were explicitly marked in the input text, and discards all other regions.

Thus \(/1::\text{speech}/self::\text{mark}()\) retrieves those speeches marked in the input text.

Because our text model considers each word to be a node, a \(\text{word}()\) node test is introduced. Given a set of regions matching a given XPath step, the step will preserve those regions that span a single word. Depending on the XPath context these words will be words extracted from the textual content of a document, an attribute value, a comment, or whatever. The \(\text{text}()\) nodetest is somewhat similar but preserves those regions which span the maximal number of words of the desired type.

Thus \(/1::\text{speech}/(2::@\text{speaker}/3::\text{text()} and /4::\text{text()}\) produces a table in which the first column contains texts spanning speeches, the second the speaker attribute associated with this speech, the third the value of this speaker attribute, and the fourth the entire sequence of words associated with this speech..

**7. Set operations within XPath**

The set of regions to be computed in a single step of an XPath expression may also be formed by using \(\text{UNION, INTERSECT, and EXCEPT}\) operators. This is achieved by extending the node tests to include the following new node tests. These are:

- \(\text{UNION}(\text{Path}, \text{Path})\)
- \(\text{INTERSECT}(\text{Path}, \text{Path})\)
- \(\text{EXCEPT}(\text{Path}, \text{Path})\)
- \(\text{NODE}(\text{Path})\)

The \textit{path} expressions occurring in such expressions are computed before resolving the step in which the set operator occurs, and are resolved relative to the same prior step as the step that the set operator applies to. These path expressions evaluate to the set of regions identified in the first step of each such expression, which have the property that they are indeed relevant to all subsequent steps in the XPath expression, according to the earlier explained rules. The specified set operations (identity in the case of NODE) are
then applied to the computed region sets associated with each such path expression, producing a new region set. The step containing such a node test constraint is finally computed according to the XPath standard, but constrained to contain only regions that also occur in the previously computed region set specified by the set expression.

8. Integrating XPath with SQL

To integrate the above concepts cleanly into SQL our underlying SQL engine supports both the correlated join operation [10] and the ability to employ a function that returns a relation wherever a table may occur.

A correlated join operates on a left and right table. Each row produced by the left table, has values that may be employed in computing specific instances of the right hand table. Each row in such a left hand table is joined with all rows that are produced by the corresponding right hand table. Thus as many right hand tables will be computed as there are rows in the left hand table. The result of such a join is the collection of all rows thus produced. A correlated join is implicitly understood to occur whenever computation of a table depends on values derived from a table against which it is to be joined.

Text values may be introduced into SQL expressions in two ways. If a text is to be permanently available, externally indexed, and potentially have grants associated with it then this text should be accessed via a specially constructed table having a single column of type text that is related with some previously indexed external text using DDL such as that shown below:

```
cREATE TABLE ws_text(ws TEXT) IN "/texts/Shakespeare.qsm"
```

The created table is initialized with a single row containing a single text value. This text value has as its provenance the externally indexed text, and spans the entire external text. The text value contains no marks.

If however a text must be dynamically indexed, this may be achieved by using either the `string_to_text(string)` function which dynamically parses the provided XML string, or the `document(string)` function, which when provided with an external filename, or URL, parses the contents of the named document. In either case, a text value is returned having as its provenance a memory resident text which is suitably indexed, and whose lifetime corresponds to that of the value.

The `xpath_table(text / XPath)` function takes two arguments, the first a value of type text, and the second an XPath expression. These two arguments are separated by a slash. The function returns the previously described table computed by applying the XPath expression to the input text.
The **xpath_value**(*text / XPath*) function returns a text value having the same provenance as the input text, and spanning the same region, but containing marks as specified by the XPath expression.

A host of additional functions exist which allow the properties of a given text value and/or its marks to be extracted, manipulated and suitably formatted for presentation [17].

**9. Integrating XPath predicates with SQL**

In order to harmonize the syntax employed within an XPath predicate with the syntax employed when expressing an SQL predicate, an arbitrary SQL predicate may occur wherever an XPath predicate is permitted. Since such SQL predicates may themselves employ tables and text values derived from XPath expressions, this relationship between SQL and XPath may become highly nested.

In compliance with the XPath standard the trivial predicates `true()`, `false()`, and `unknown()` are added to our implementation of SQL2, and may occur anywhere that a predicate is expected. They return the result corresponding to their name. The `mod` function is implemented as specified in the XPath standard, and the keyword `div`, may be employed as an alternative method of specifying division. Generic XPath functions such as `SUBSTRING-BEFORE` and `SUBSTRING-AFTER`, are treated as conventional SQL functions, with some care being taken to correctly parse the hyphen.

The following functions are meaningful only when employed within the scope of an XPath predicate, and will produce syntax errors if appearing outside such a context.

- `context()`
- `position()`
- `last()`

The context is the region currently being tested against the inner most XPath predicate, expressed as a value of type text. The `position()` and `last()` functions behave as specified by XPath. These functions may be employed as required, and within the scope of XPath predicate evaluation behave like any other outer variable.

XPath predicates employ various syntactic shortcuts, primarily to improve the external readability of such predicates. Permitting such syntactic shortcuts to be preserved, while allowing an arbitrary SQL predicate to be employed in place of an XPath predicate is non-trivial, and more reasonably might be forbidden.

In accordance with the XPath standard an XPath predicate may be either an arbitrary SQL predicate or an arbitrary SQL value expression v. In the second case this is understood to be shorthand for the predicate ‘`position() = v`’.
For consistency with SQL, the XPath keyword RANGE is removed, and the BETWEEN clause used instead. If such a clause lacks an initial operand, this is understood to be shorthand for ‘position() [NOT] BETWEEN ...’.

As specified in the XPath standard, an XPath expression may occur anywhere within the context of an XPath predicate. Such an XPath expression, when used as a predicate returns true if it describes some set of regions, relative to the current XPath context. Otherwise it returns false. XPath node tests may also be employed as predicates, which return true if they hold for the current XPath context.

10. Support for aggregation

The aggregate_strings([set qualifier] string [, separator ] [ order by ...] ) function concatenates the strings that occur within a grouping, separating each from the next by the optional specified separator. Duplicate strings may be preserved or eliminated, in accordance with the optional set qualifier. The optional order by clause specifies a list of values to be used in computing the desired sort order for strings thus concatenated.

The aggregate_marks( text ) function operates on groups of texts having the same provenance, and produces a single text having the same provenance containing all of the marks present in any text within this group. When these texts span differing regions, the nearest common ancestor of all input regions is the region associated with the resulting text.

The isolate_subtexts( text ) function is very loosely the inverse of the aggregate_marks() function. Given an input text it produces an output table containing a single column. The input text is reproduced in each output row. Successive rows in this output table have texts containing successive marks within the input text. This function therefore allows marks within a text to be independently manipulated, kept or rejected, and if desired for the preserved marks to be re-aggregated back into a single text.

11. Comparisons with XQuery

All of the case studies [20] provided with XQuery [19], were implemented using Textserver [17]. The proposed XPath extensions prove valuable in a large number of these cases. The ability to apply the full power of SQL to text was often useful. Textserver was able to efficiently and effectively handle the vast majority of the proposed use cases.

Significant problems were encountered only with queries that involved recursion, since SQL2 does not support recursion. In case study 2.6, the proposed XQuery solution employed a user constructed recursive function that provided a textual template specifying how text is to be dynamically restructured, which could not be duplicated using SQL. This limitation may be addressed by augmenting Textserver with a powerful
recursive text transduction language such as XSLT [21]. In case studies 8 and 9, recursion was also required, but here we were able to use the ADO scripting language, to implement the necessary recursion needed to resolve the queries. Extending our SQL implementation will circumvent these limitations, so that it directly supports recursive construction of result sets [14].

12. Performance considerations

It is important that any proposed means of querying large semi-structured texts be capable of being performed efficiently, through the use of auxiliary text indices. To prove that the proposed solutions were capable of being implemented efficiently, the TPC-D benchmark was used [18]. The tables required by this benchmark were stored in an Oracle database, and each separately encoded within an XML document. Views of the TPC-D tables were then derived from these XML documents by using the proposed XPath extensions. Oracle and Textserver were then accessed through identical interfaces and performance results compared, on all seventeen TPC-D queries.

The results obtained from both sources were consistent, as one would expect [17]. Oracle is currently somewhat faster in most cases, but had performance in the same order of magnitude as Textserver on most queries. Some degradation in optimal performance can be attributed to poor join order selection on the part of Textserver while some is inherent, since any product that recovers relations from an XML encoded text, has to compute not simply the location of tuples, but the location of every value contained within each tuple, according to a sophisticated collection of rules. This overhead in the context of semi-structured text, is more than offset by the ability of the proposed XPath query language, to dynamically and efficiently generate user specified views of an arbitrary text, without the need to store this text within a relational database system, and to directly return arbitrary regions of text.

Performance results are currently somewhat preliminary. Additional effort is needed to determine how best to resolve SQL queries having embedded XPath expressions, that themselves recursively employ arbitrary SQL expressions. There is also further research needed into the various strategies for indexing text. Experiments must also be conducted on significantly larger data sets.

13. Design issues

Unlike several other proposals [15] we avoid decomposing XML documents into relational tables, preferring instead to dynamically create relational views of relevant components of an XML document, from a given extended XPath specification. This approach has many advantages. Preserving the source document in its original format makes queries against it more direct and easier to integrate with other tools that operate on the original document. It also allows the conceptual decomposition of such documents to be done in a variety of ways that are optimal with respect to the specific queries performed on the text. The expensive task of converting texts into relational data...
and later reconstructing desired regions of the original text from this relational data is entirely avoided. The construction of indices entirely separate from the relational database gives total freedom in deciding how such indices can best be implemented and used. The problem of allowing documents to be concurrently updated [16], while being queried, becomes at least tractable, since the queries are being performed against the document being updated. Since XML source texts are never stored, these may be widely distributed, as for example occurs on the web, and dynamically indexed when first accessed [9] or when such cached indices are lost. Text indices may also be widely distributed in a manner similar to the distribution of the source texts they support [2], and may be dynamically updated without the need for supporting relational database software to be present. The text files, indices and database catalogs may all be stored on read only devices, since all can be used without requiring updates to be performed.

14. Conclusions

Designing, implementing, and deploying software that employs novel but largely unproven database languages is no more desirable than reinventing the wheel. It is much more cost effective to extend existing well-established database concepts and standards, so that these established standards provide the additional capabilities needed in order to query, extract, and restructure information in XML texts.

A viable strategy for extending the SQL language has been presented that is competitive with XQuery. This strategy greatly simplifies the problems associated with optimizing queries applied against semi structured text, by encapsulating the core specification of a textual query in an enhanced non-procedural XPath language, that returns relations.

This product is further described and demonstrated at www.textserver.com [17].
References


