ISO/IEC JTC1/SC32 WG3:HEL-nnn
ANSI NCITS H2-2000-440

August 7, 2000

Subject: A Snapshot of XSQL: A Query Language for XML
Author: David Beech
Source: U.S.A.
Status: Discussion paper

References

[SQL/XML]: Melton J.: Subproject proposal for "XML-Related Specifications (SQL/XML)" H2-2000-331 (WG3:HEL-026, SC32 N00510), Aug 4, 2000

[XML Query]: W3C XML Query WG. Home page: http://www.w3.org/XML/Query

Overview

The attached paper "A Snapshot of XSQL: A Query Language for XML" is offered as partial background information for the subproject proposal [SQL/XML].
It is not intended as a specific technical proposal, being only a snapshot of exploratory work in progress in connection with the efforts of the W3C XML Query Working Group [XML Query]. Rather, it is intended to provide timely information that might suggest the value of possible coordination of efforts in this area. Moreover, as indicated in [SQL/XML], the query language is only one possible area of relationship between SQL and XML, and hence this paper provides background only to part of the problem space addressed by the subproject proposal.

XSQL suggests one possible syntax for the data model and algebra being developed by [XML Query]. It has been applied to the solution of a number of examples included in use cases submitted to [XML Query]. Other language syntaxes, not closely related to SQL, have also been proposed and applied to these examples. The requirements approved by [XML Query] allow that one or more syntaxes may be included in the eventual W3C Recommendation.

This paper was presented to the [XML Query] meeting on July 27, 2000, and in the ensuing discussion there was interest in the possible relationship of [XML Query] to any XML-related work that might be initiated in the SQL standards world.

A Snapshot of XSQL:

A Query Language for XML

David Beech
Oracle Corporation

July 11, 2000

"The language designer should be familiar with
many alternative features designed by others...
His task is consolidation, not innovation."
-- Sir Tony Hoare [1]

1. Introduction

This paper introduces the main features of XSQL, a language for querying XML documents [2].

The syntax and semantics of XSQL are envisaged as one desirable realization of the functionality of the XML Query Data Model [3] and XML Query Algebra currently under development by the XML Query Working Group [4]. The XML Query Requirements [5] allow for more than one syntactic realization of a standard XML Query language, and there are certain advantages to having one such syntax be compatible with the accepted SQL-99 standard [6] and the almost complete SQL/MM Full-Text standard [7]:

- SQL is a widely-understood language, and education is simplified if existing functionality does not have to be relearned in a different syntax
- SQL users will often want to query XML data stored in databases together with other data, and to have an SQL query span various kinds of data uniformly
- SQL-99 and SQL/MM Full-text already encompass much of the functionality required to query the information content of XML documents
- Semantic compatibility with the SQL kernel of XSQL allows highly scalable, robust implementations of the XML Query language to be built on the base of existing DBMS engines, rather than from the ground up.

Other syntaxes will be defined for XML Query, but the case being presented here is that one syntax should be strongly SQL-related, both for usability reasons and to ensure quality support at the high-volume, high-performance end of the spectrum for XML data on large web servers.

For XSQL, some extensions are proposed beyond SQL-99 and SQL/MM Full-Text, together with substantial subsetting of those standards. For purely XML purposes, XSQL alone may be used. In an SQL environment, the XSQL extensions may be used together with the full power of the existing SQL and Full-Text support to query both XML and non-XML data in a database.

Since the XML Query Data Model exists only in its first Working Draft, and the XML Query Algebra has not yet appeared, this paper necessarily presents a snapshot of work in progress. The design is likely to need modification and completion as those specifications assume their final shape.

Two good reasons for exploring an SQL-related syntax at this stage are that

- a number of XML Query Use Cases have been published [8], and solutions in other proposed syntaxes have been developed within the XML Query WG, so that it seemed valuable to offer XSQL solutions for comparison
- the development of the Query Algebra and the Query Language Syntaxes should be part of an iterative process, where the theoretical algebra can suggest strongly-based semantics to be expressed in the syntaxes, and the application of the syntaxes to the solution of examples falling within the scope of the Query Requirements can suggest appropriate operators needed in the algebra.

The relationship between any new query language and an accepted standard query language like SQL deserves to be studied for a number of reasons, both conceptual and pragmatic.

On the conceptual front, we will first take a simple example that illustrates the essential similarities between the SQL and XML data models. This will show that although SQL-99 already treats some aspects of collections of values and of objects with identity, a more general treatment of collections in XSQL would be fruitful in dealing with XML and its collections of recurring elements and attributes.

On the pragmatic front, we will note the immediate likelihood of XPath [9] being used as a sublanguage applied to XML documents stored as strings in SQL database systems. This leads to the suggestion that a better long-term solution in an SQL-like syntax would be to merge similar functionality into the path expressions aleady in SQL-99.

The generalization of collection facilities, and the adoption of XPath functionality, are the two major areas of extension beyond SQL-99 currently included in XSQL. In addition, we adopt the Contains(...) functionality of SQL/MM Full-text searching, expressed in the form of a function applicable to any XML string.

The body of the paper will deal with these three areas in turn in sections 2, 3, and 4..

1.1 Similarities of the SQL and XML Query Data Models

At first sight, XML documents and relational tables might appear to have little in common, so that the chances of sharing a common query language would be quite slim.

An XML document has a textual representation, marked up with properly nested pairs of start-tags and end-tags in angle brackets, and the mark-up can be quite irregular in its structure. An SQL table, on the other hand, conjures up a picture of rectangular regularity, of rows and columns of data values without any mark-up other than names and types at the heads of the columns.

But let us begin with an example heavily weighted in favor of exhibiting the similarity. This, like many other examples in this paper, is taken from the set of Use Cases

[8] published by the XML Query Working Group to give more concrete illustration of some of the requirements being addressed by the WG.

The Use Case "R": Access to Relational Databases takes the example of an on-line auction supported by relational tables, such as the USERS table:

USERS

USERID	NAME	RATING
U01	Tom Jones	B
U02	Mary Doe	A
U03	Dee Linquent	D
U04	Roger Smith	C
U05	Jack Sprat	B
U06	Rip Van Winkle	B

and makes this queriable in the XML world by postulating an "XML view" of the data, in the form:

```
<users>
  <user_tuple> <userid>U01</userid> <name>Tom Jones</name>      <rating>B</rating>
</user_tuple>
  <user_tuple> <userid>U02</userid> <name>Mary Doe</name>       <rating>A</rating>
</user_tuple>
  <user_tuple> <userid>U03</userid> <name>Dee Linquent</name>   <rating>D</rating>
</user_tuple>
  <user_tuple> <userid>U04</userid> <name>Roger Smith</name>    <rating>C</rating>
</user_tuple>
  <user_tuple> <userid>U05</userid> <name>Jack Sprat</name>     <rating>B</rating>
</user_tuple>
  <user_tuple> <userid>U06</userid> <name>Rip Van Winkle</name> <rating>B</rating>
</user_tuple>
</users>
```

Abstracting from both of these, we can take the common data model to be a tree rooted in a node named "users". In the XML view, the immediate child nodes are all named "user_tuple", whereas in the relational model the rows are represented by nameless nodes. Below these, the models are the same again, with named elements or columns respectively having leaf values that are strings.

So, at least in this simple example, we see similarity in the data models. Both SQL and XML standards in fact emphasize abstract data models in their specifications.

In the case of ANSI/ISO SQL [6], the Syntax Rules and General Rules are framed in terms of values and their descriptors, where descriptors may provide the names and tree structure shown above. In the case of XML, there is a a generic XML Information Set [10] that provides a set-oriented equivalent of the tree, with an Element Information Item for each node of the tree which includes information such as the

name of the node and references to its child Element Information Items, if any.

XML Query, in accordance with its Requirements document, takes as input to a query, not XML representation syntax, but Information Sets corresponding to the documents to be queried. Preferably the documents have have been validated by an XML Schema processor that has added information such as the types of all nodes, and has passed on this augmented Post-Schema-Validation Information Set (PSV-Infoset) to the XML Query processor.

So far, we have been concentrating on the input to a query, and emphasizing the use of an abstract data model to describe this. Turning to the output from a query, it is a natural requirement to be able to produce XML representations of query results. But here again it is valuable to distinguish the information content from the representation - just as with relational queries, where the result defined by an SQL SELECT statement is a relation (a computed table), which may then be represented in various ways on printers or displays, or via APIs to programs. So the result of an XML Query is an instance of the XML Query Data Model (which we have pictured as a tree), one of whose representations is in XML syntax.

Regarded in this light, XSQL can use SQL syntax and semantics to map from input instances of the (tree) data model into output instances of the data model, which could then on demand be emitted in XML syntax, or by other means. When using subqueries within a query, it is very natural to deal with their results within the abstract data model. Only at the top level are we forced to consider mapping the results to other forms.

We can illustrate this with the first example of Use Case "R". (The nature of the example queries can probably be understood sufficiently at an outline level without the full background of their document structures and expected results - otherwise please refer to [8].)

(Q1): List the item number and description of all bicycles that currently have an auction in progress.

The SQL query against the underlying items table would read:

```
SELECT i.itemno, i.description
FROM items i
WHERE i.start_date < CURRENT_DATE
AND i.end_date > CURRENT_DATE
AND i.description LIKE '%Bicycle%';
```

In XSQL, the amendments are slight:

```
AS result
SELECT (i.itemno, i.description) AS item_tuple
FOR i IN "items.xml".item_tuple
WHERE i.start_date < CURRENT_DATE
AND i.end_date > CURRENT_DATE
AND i.description LIKE '%Bicycle%';
```

Firt of all, the tree picture of the result of a pure SQL query would always have an anonymous root node, with anonymous child nodes for each row, before getting down to the named nodes for the columns (and in the case of SQL-99, sub-columns) and their values. So XSQL extends the use of the

AS clause, already available in SQL to give user-defined names to result columns, to two additional positions as shown here. "AS result" gives the user-chosen name "result" to the whole query result, and "AS item_tuple" gives a name to each row, which will have child nodes named "itemno" and "description" (simple column names are carried through to the result by default in SQL, so there is no need for AS clauses for them in this example). By using and extending this existing naming mechanism in SQL SELECT, we avoid having to use a separate tagging constructor for results, and the default rules are very convenient when selecting subtrees that do not need to be renamed.

Second, the syntactic form

```
FROM table [[AS] identifier]
```

has been replaced by

```
FOR identifier IN collection.
```

The identifier that serves as a correlation-name, being bound in turn to each row of a table (i.e. to each member of a collection of rows), is mandatory in the FOR clause, and must always be used when referring to parts of a row or member, as in `i.description` above. By disallowing the abbreviated naming (and also the implicit casting of subquery results) permitted within the scope of a FROM clause, the FOR clause enables the SELECT statement to extend smoothly to deal with more general collections, as in OQL [11], rather than just tables that are collections of flat rows. This small syntax change helps provide a user-friendly language for the more general data model by allowing clean left-to-right name resolution to be applied to extended paths such as p.q.r.s.t, and to single identifiers such as p intended to be top-level unqualified names, without affecting the deeper semantics of an SQL query engine

Third, the collection in the FOR clause may be specified by reference to anything that satisifes the extended data model, in this case an XML document named `"items.tuple"` (using SQL double quoting to surround the identifier since it contains a period). The following `.item_tuple` creates a simple form of path expression, evaluating to all immediate subnodes in the tree model with name `item_tuple`. In general, this will produce a collection of such nodes, and the user-chosen identifier `i` will be bound to each in turn.

The tree for a PSV-Infoset is, of course, more general than that for an SQL relation (table, view, or query result), but SQL-99 has considerably increased the flexibility of its data model in moving from the pure relational model of SQL-92 to an object-relational model. Two major extensions in the SQL-99 data model are to support structured types and arrays. The structured types have "attributes" (akin to Java classes that have "fields"), and may have functions defined as their "methods". The arrays are varying-length (possibly empty) arrays whose elements may be of any type, including structured types. The relevance to the comparison with the XML data model will become clear if we extend our example with a structured type and an array.

Suppose that there is an SQL-99 structured type Address_t with SQL "attributes" (street, city, state, zip), and a type Phone_t defined over the character type to represent phone numbers. Then the USERS table could also be given a column "address" of structured Address_t values, and a column "phones" which was an array of Phone_t values.

The tree picture of the data model for USERS will then have deeper nesting of subtrees, just as is commonly the case in the XML data model.

Structured types can also be used at the top level when creating SQL-99 tables, as in:

```
CREATE TABLE Addresses OF Address_t;
```

in which case each instance of Address_t in the table may be thought of as a row, and each of the SQL "attributes" (street, city, state, zip) for the type gives rise to a column of the same name. There is additionally a "self-referencing column" with a user-defined name to provide a referenceable identity value for each row. We are here on the brink of a more general treatment of collections. Since the rows of a table in general form a multiset, the table Addresses is an instance of a "multiset of Address_t", i.e. a "collection of Address_t", since a multiset is the general form of a collection. More constrained forms of collection would be a "set of Address_t" which excluded duplicate addresses, and an "array (or list or sequence) of Address_t" which maintained an ordering on its members, independently of their values. There might even be utility in an "ordered set of Address_t" that combined both the uniqueness and the ordering properties.

The fact that structured values can be nested, and have their fields named, leads to a similar picture to that of nested XML elements. The picture for an array differs slightly from the natural XML model, in that "phones" labels the whole array whereas in XML it is possible to use a repeated element "phone" with or without a parent "phones". However, we will show later that XSQL can handle these and other forms of collection in a consistent manner.

1.2 Relationship to XPath

Already, XML data is being stored in databases, and there is a need to query it. Of course, an XML query language should not be limited in applicability to database usage, but there is clearly a need to be able to cope with querying XML on large servers supported by database systems. There are currently two broad approaches available: to store the XML document as a character string, or with its elements and attributes spread among column values in the rows of one or more tables. These approaches may be combined in hybrid fashion by storing parts of a document spread across columns, and other parts as strings. The approaches may also be used redundantly, spreading some or all values for easy SQL queriability and retaining the original string value for reproducibility.

Querying of unparsed XML data in strings can be accomplished by making XPath implementations accessible, and indeed XSLT transformations could be invokable to provide flexible construction of XML results of queries.

So why is this easy short-term solution for SQL database systems not also a good long-term solution? The answer lies in the considerable overlap between the query functionality of SQL and that of XPath, e.g. the use of the paths `invoice.customer.name` in SQL-99 and `invoice/customer/name` in XPath. To have continuing duplication of this kind would force many users to learn two similar but slightly different languages, and would force vendors to provide two implementations (or face many tricky semantic differences in attempting to common up the work under the covers). Moreover, if this is accepted as a long-term direction, the picture will not remain static. Both languages would be likely to be extended in overlapping but different ways, with increasing divergence.

In particular, the optimization problem will be very challenging for implementors, to provide efficient and scalable support for a wide range of queries over documents with the flexibility of structure permitted by XML. It is desirable that the work should be done once in a system, on a common data model and set of fundamental algebraic operators, rather than twice over.

A subpart of the functionality where duplication should especially be avoided is in the provision of operators for scanning full-text. Given the work that has already gone into support of full-text in database systems, and the emergence of SQL/MM Full-text after many years of study and negotiation, there does not appear to be a prima facie reason why full-text appearing in XML documents should require reinvention of the wheel.

2. XSQL Collections

2.1 Query Expressions

The most general form of query in XSQL is a <query expression> that allows simpler <query specification>s to be connected by collection operators, or may be a single <query specification>. The result is a collection that is unnamed, unless given a name by a preceding AS clause. Additionally, an optional ORDER BY clause may appear at the end, whereas in SQL-99 a similar construct may only be used when defining a cursor associated with a query expression.

Work is still in progress on some details of the result collection types and their element types, which will be developed in parallel with the refinement of the XML Query Data Model and Algebra in these areas.

(The formal grammar rules may be found collected together in Appendix A "XSQL Grammar".)

The collection operators include UNION, EXCEPT (with equal precedence), and INTERSECT (with higher precedence), for example:

```
SELECT c.phone FOR c IN document1.customer
  UNION
SELECT c.phone FOR c IN document2.customer
```

Each SELECT is a <query specification> producing a collection of named elements in the data model, and the union produces a similar collection. This might be represented in XML syntax in the form

```
<phone>(650)321-7654</phone>
<phone>(831)864-2200</phone>
...
```

as a sequence of elements without the surrounding element that would make the result a true XML document. This is analogous to the node sets that may be the results of XPath expressions.

The distinction between named and unnamed values is an important one that exists in both the current XML Query Data Model and in SQL. In the XML world, incoming values in a document will be named by their containing elements or attributes (except for text in elements with mixed content), but in the course of computation, e.g. in an XPath expression or in a query, unnamed values such as a node set or the sum of two integers, may be produced. A similar situation exists in SQL where "SELECT salary ..."

will select named values from the salary column of a table, and carry their names through into a salary column of the result, unless overridden by "SELECT salary AS base_salary ...". However, as soon as such a value participates in a more general expression, its name is liable to vanish from the result, e.g. in "SELECT salary+commission ...". This part of the result is then a nameless value unless a name is expcitly provided as in "SELECT salary+commission AS total_salary ..." Similar situations may be expected to be frequent in the XML Query Algebra, where the operators or functions will need to specify precisely when names are or are not retained in their results.

To form a "named collection" with a node named (e.g.) "result" as the parent of all the members of the collection, one can write in XSQL

```
AS result
  SELECT c.phone FOR c IN document1.customer
    UNION
  SELECT c.phone FOR c IN document2.customer
```

and then the corresponding XML syntax for the result would be

```
<result>
  <phone>(650)321-7654</phone>
  <phone>(831)864-2200</phone>
  ...
</result>
```

Also at this top level of a query expression, it is possible to use a WITH clause to give a name to the result of evaluating another specified query expression. This may be useful for readability of complex queries, and is especially convenient when the same nested query expression may be used more than once in a query: Moreover, it is essential to the use of recursive queries to be able to give a name to the query result that is being queried recursively until it reaches a

For example, in the Use Case "REF", there is the following fairly ambitious query:

(Q11) List the names of all Joe’s descendants. Show each descendant as an element with the descendant’s name as content, and his or her marital status and number
of children as attributes. Sort the descendants in descending order by number of children, and secondarily in alphabetical order by name.

```
/*  use SQL-99 RECURSIVE query and CASE expression */

AS result
WITH RECURSIVE descendants AS
(  SELECT p.@name, p.@spouse, c.name AS childname
   FOR p IN "census.xml"..person,
       c IN p.person UNION p.@spouse->person
   WHERE p.@name=’Joe’
 UNION
   SELECT p.@name, p.@spouse, c.name AS childname
   FOR d IN descendants,
```

```
       p IN "census.xml"..person,
       c IN p.person UNION p.@spouse->person
   WHERE p IN (d.person UNION d.@spouse->person)
)
SELECT ((CASE WHEN p.@spouse IS NULL THEN 'no' ELSE 'yes') AS @married,
         COUNT(*) AS @kids,
         STRING(p.@name)) AS descendant
FOR d IN descendants
WHERE d.@name!='Joe'
GROUP BY @name, @spouse
ORDER BY @kids DESC, STRING(descendant) ASC
;
```

Since the language description is proceeding top-down in this section, the examples will often use lower-level features not fully described at that point. However, the general intention should be clear, and the details will be dealt with later.

For example, extended path expressions with Xpath functionality also in general return collections, and the expression `p.person UNION p.@spouse->person` above shows that the UNION operator may be applied to any collections with similar members, and is not limited to use in top-level query expressions. This principle of orthogonality is carried through the design for collections - an operator or function is valid in an expression wherever its operands or arguments are of the right type.

Identifers like `@spouse` are permitted, as in abbreviated XPath syntax, to correspond to XML attributes named `spouse` in the XML Query Data Model.

The right arrow `->` may be used between steps of a path when the value to the left is known to be of a reference type, to imply dereferencing before evaluating the next step, as in SQL-99, C, and C++. XSQL extends this notation to apply also where the value to the left satisfies a key reference constraint in XML Schema, which is analogous to a foreign key constraint in SQL.

The WITH clause in the above example uses a recursive query, following the fixpoint semantics specified in SQL-99. A "fixpoint" of the recursive UNION-query definition of `descendants` above is a collection of (@name, @spouse, childname) tuples such that when the definition is evaluated with this value substituted for `descendants`, it will return exactly the same collection. Another way of looking at this is to consider a "naive evaluation" of the definition which starts with `descendants` as an empty collection and substitutes this into the definition. The result is successively substituted to obtain a new collection until no more elements are added, in which case a "fixpoint" value of the collection has been determined. Proofs of monotonicity, termination, and uniqueness are beyond the scope of this paper!

The CASE construct for conditional expressions is as in SQL-99.

The STRING function is defined as in XPath to produce an unnamed string value from a named element or attribute.

2.2 Query Specifications

Moving on to a systematic exposition of the simpler <query specification>s (the SELECT constructs) that can occur within a general <query expression>, these retain the conventional SQL structure beginning with a SELECT clause, followed by a FROM clause and optional WHERE, GROUP BY, and HAVING clauses, with the one signficant change mentioned earlier: the FROM clause is replaced by a FOR clause.

2.2.1 SELECT clause

Beginning with the SELECT clause, the word DISTINCT may be used as in SQL-99 to remove duplicate members from the resulting collection, i.e. to guarantee that the result is a set rather than a multiset, by using the equality test for the type of the members of the collection. E.g., from Use Case REF again:

(Q3) Find parents of athletes.

```
AS result
SELECT DISTINCT p ATTRIBUTES_ONLY
FOR p IN "census.xml"..person,
    child IN p.person UNION p.@spouse->person child
WHERE child.@job='Athlete';
```

This also illustrates the use of the shorthand ATTRIBUTES_ONLY for projecting only the attributes of an element into the result, shedding element children and any other content such as untagged text.

The only other XSQL extension to the SELECT clause is to optionally add a name to the whole parenthesized "row" of a result, as in the example

```
SELECT (i.itemno, i.description) AS item_tuple
...
```

in the Introduction above.

2.2.2 FOR clause

The FOR clause has already been discussed in outline, and the above example (Q3) illustrates some additional points.

The name `"census.xml"` is enclosed in double quotes to form an SQL <delimited identifier> which allows arbitrary characters to be used in an identifier (with the escaping convention that a double quote character in an identifier must be shown as a pair of consecutive double quote characters in this representation).

A deeper consideration at this point is how such names are bound to their environment. The FOR clause is semantically the beginning of a query expression, identifying the source instances of the data model to which the query is to be applied. In the SQL world, an environment of one or more database schemas is assumed, and a data definition language is provided as part of SQL in order that the names appearing in schemas may be defined within the language. The XML Query language does not have such wide

scope, and its queries must therefore assume some extra-lingual means of binding external names used in queries to their referents, whether in a database schema, or a file system, or at aWeb address. It might prove desirable to subdivide the external namespace and use notations like `FILE("census.xml")` and `XPath("http://www.doc.gov/census.xml")` for these external names, and possibly also to allow user-defined functions that could return source instances by means known to themselves. With the use of an XPath, we allow paths that may be either absolute, or relative to a context within which the query is being evaluated. This area is under active investigation, but for the moment we are including in XSQL only simple <identifier>s or <delimited identifier>s to be bound to the external environment.

Once an external name has been bound to a source, it may optionally be followed by other steps in a path, using XPath functionality in XSQL notation. THus we see above the use of `"census.xml"..person` to start from the document identified by `"census.xml"` and proceed via .. (the XSQL equivalent of //) to any descendant-or-self element node named `person`. The correlation name `p` will then iterate over the collection of such `person` nodes in document order.

Just as in the SQL-99 FROM clause, the FOR clause in XSQL allows one or more iterators to be specified. Moreover, XSQL insists that each iterator has an identifier associated with it as a correlation name to be bound to each member of its collection iteratively. The left to right order of the iterators is significant, with the leftmost iterator defining an outer loop, and any following iterators defining nested loops in order. This also means that the correlation name of an iterator may be used in defining any following iterators, as in the definition of `child` in terms of each person `p` in (Q3) above. When the iterators are mutually independent, one may think of the totality of all the bindings of the correlation names as corresponding to the Cartesian product of the members of the input collections. In the general case, one has a kind of "dependent Cartesian product", where for each binding of the first correlation name, only a particular subset of bindings of the next correlation name is considered, and so forth, producing irregular data-dependent iterations.

The extensions beyond SQL-99 that are defined by XSQL may only be used in FOR query specifications, and not FROM query specifications. Moreover, XSQL disallows mixing of FOR clauses and FROM clauses within a query specification and its subqueries, in order to avoid name resolution complexities. For XML purposes, the FOR clause can be used exclusively. In the SQL world, existing FROM queries that make no references to schema objects outside their own lexical scope can be wrapped as views and the view names are then allowed to be used within FOR queries.

2.2.3 WHERE clause

The body of the WHERE clause is a <boolean value expression>, and the main novelty here is in the use of XPath functionality in the expressions (to be described in section 3 below), and in the more general use of collection values. For an example of the latter, we turn again to Use Case REF:

(Q10) Find single parents (people with children but no spouse)

```
AS result
SELECT p ATTRIBUTES_ONLY
FROM "census.xml"..person p
WHERE EXISTS(p.person)
AND NOT EXISTS(p.@spouse);
```

Here the `EXISTS` and `NOT EXISTS` operators are used in their SQL sense, being equivalent to `0 < COUNT(...)` and `0 = COUNT(...)` respectively. However, they are applied, not to subqueries, but to the collections returned by the paths `p.person` and `p.@spouse`.

The SQL <null predicate> of the form `exp IS [NOT] NULL` is available for testing for null markers in the data model. These correspond in the XML world to use of <... xsi:null='true'> markers on elements, as defined by the draft XML Schema specification , part 1 [12].

Other SQL predicates useful in the XML world include string testing operators such as LIKE and SIMILAR. An example of LIKE was shown in section 1.1. above, use case "R" (Q1). There are also several string functions such as SUBSTRING to match a given string or regular expression; UPPER and LOWER for case folding; TRANSLATE for character translation; and TRIM to remove specified leading and/or trailing characters.

2.2.4 GROUP BY and HAVING clauses

The GROUP BY clause in XSQL omits the considerable SQL-99 extensions that offer cubes and other data mining structures, and adopts the slightly improved semantics pioneered by OQL [11] for the simple case. This takes advantage of the fact that the "collection of groups", i.e. a collection of collections, formed temporarily during the evaluation of a GROUP BY query, is a valid construct within the extended data model of XSQL, whereas it could not be expressed in the original relational SQL model.

A simple XSQL example of GROUP BY was shown in (Q11) in section 2.1 above.

The HAVING clause is unchanged, applying its predicate to each group in turn, and only retaining those groups for which the predicate is true.

3. XSQL Paths

XSQL contains the path syntax of SQL-99, which allows the separators '.' for direct containment, and '->' for dereferencing before taking the next step. This is generalized by adding Xpath functionality as follows.

The XPath syntax has the following general form:

```
[1] LocationPath ::=
        RelativeLocationPath
      | AbsoluteLocationPath

[2] AbsoluteLocationPath ::=
        '//' RelativeLocationPath?
      | AbbreviatedAbsoluteLocationPath

[3] RelativeLocationPath ::=
```

```
        Step
      | RelativeLocationPath '//' Step
      | AbbreviatedRelativeLocationPath

[4] Step ::=
        AxisSpecifier NodeTest Predicate*
      | AbbreviatedStep

[5] AxisSpecifier ::=
        AxisName '::'
      | AbbreviatedAxisSpecifier
```

To merge this into XSQL, we replace the '/' character with '.', and omit the use of `Predicate*`. The functionality of the predicates is replaced by use of the WHERE clause in XSQL, and most of the functions defined by XPath for use in predicates are made generally available in XSQL expressions, e.g. in the WHERE clause. We saw an example above in the use of the STRING function in section 2.1, use case REF (Q11). Some XPath functions already have equivalents or similar functions in SQL, and detailed work is in progress to resolve questions of overlap in this area.

The axis specifier syntax is adopted unchanged, with its '::' separator.

Function and axis names are adjusted where necessary to replace '-' by '_', rather than forcing such names to be double quoted in XSQL.

XPath also defines abbreviations as follows:

```
[10] AbbreviatedAbsoluteLocationPath ::=
          '//' RelativeLocationPath

[11] AbbreviatedRelativeLocationPath ::=
          RelativeLocationPath '//' Step

[12] AbbreviatedStep ::=
          '.'
        | '..'

[13] AbbreviatedAxisSpecifier ::=
          '@'?
```

Rules [10] and [11] are adopted into XSQL with '//' as an abbreviation for /descendant-or-self::node()/ replaced by '..'. Rule [13] is adopted unchanged, where the '@' prefix indicates the attribute:: axis, and the absence of any axis specifier indicates the element child:: axis.

The abbreviations in rule [12] for self::node() and parent::node() are currently not supported, so that these steps must be written out in full when needed. Possible alternative abbreviations are under consideration.

Additionally, the SQL '->' step separator mentioned above is extended to apply wherever the

information available in the input data model indicates, for example as a result of DTD or XML Schema processing, that the preceding path has produced a value of type URIReference or IDREF, or has a "keyref" constraint applied to it.

A wider question is how to indicate paths that retain more than the leaf node sets to which they evaluate in XPath and SQL. The path expressions of GORDAS [13] and the filters of Quilt [14] are under study here.

4. XSQL Full-Text

XSQL embodies the functionality of the SQL/MM Full-text Contains method in the form of a Contains function, i.e. what is expressed in SQL/MM in the form

```
FT.Contains(pattern)
```

applied to a value FT of type FullText, appears in XSQL as a function applied to a first argument S of type String"

```
Contains(S, pattern)
```

This has the advantage that the type String for use with XML text can be defined to combine normal SQL string functionality such as concatenation with the FullText functionality of Contains, without the need for casting. But we will begin with a simple example in use case TEXT:

(Q1) Find all news items where the name "Foobar Corporation" appears in the title

```
AS result
SELECT ni
FOR ni IN "input.xml".news_item
WHERE Contains(ni.title, '"Foobar Corporation"')=1 ;
```

The pattern in the second argument to Contains can be any character-string valued expression, and here it is a character-string constant enclosed in single quotes. The value of the pattern is a literal that in the pattern language must be enclosed in double quotes, and in this case it is merely a search for the consecutive words Foobar and Corporation separated by arbitrary whitespace. This illustrates one difference between the word-based searching of Contains, and the simple character string searching of Substring. The Contains pattern specifies only a single space between the words, but can be matched by arbitrary whitespace between them. Moreover, the Contains pattern would not be matched by finding NonFoobar Corporation in the input string since it tests whole words for equality, whereas the Substring function would find a match in that case when searching for Foobar Corporation.

The result of the Contains function is an integer, 1 for true and 0 for false. (SQL/MM also has a similar Rank method that returns a Double Precision result.)

We see the convenience of combining simple string operations and full text capabilities in a more complex example in use case TEXT:

(Q4) Find titles of news items where Foobar Corporation and one or more of its partners are mentioned in the same sentence, but none of its competitors are
mentioned in the news item. (The "." character designates the end of a sentence.)

```
AS result
SELECT ni.title
FOR ni IN "input.xml".news_item
WHERE EXISTS (SELECT part
              FOR coy IN "input.xml".company, part IN
coy.partners.partner, para IN ni..par
              WHERE coy.name='Foobar Corporation'
              AND Contains(para, '"Foobar Corporation" IN SAME SENTENCE AS
"' || part || '"')=1 )
AND NOT EXISTS (SELECT compet
                FOR coy IN "input.xml".company, compet IN
coy.competitors.competitor, para IN ni..par
                WHERE coy.name='Foobar Corporation'
                AND Contains(para, '"' || compet || '"')=1 );
```

The second use of Contains constructs its pattern argument dynamically within the query by concatenating the necessary double quotes around the string value bound to the correlation name "compet" on individual iterations.

The first use of Contains constructs part of its pattern in a similar way, but takes advantage of one of the proximity searching features of the pattern language, IN SAME SENTENCE AS, so that if "part" is bound to Gorilla Corporation on one iteration, the pattern would be:

```
"Foobar Corporation" IN SAME SENTENCE AS "Gorilla Corporation"
```

In fact, the underlying model of text in SQL/MM Full-text is

"any sequence of characters which represents one of the following:

- a single word,
- a sequence of words,
- a single sentence,
- a sequence of sentences,
- a single paragraph,
- a sequence of paragraphs.

A sentence consists of one or more words. A paragraph consists of one or more sentences.

When modeled as a value of the FullText type of this part of ISO/IEC 13249 a text value is associated with
a specific language. The recognition of word, sentence and paragraph boundaries is largely governed by language specific rules, conventions, and heuristics. It is implementation-defined which of these rules, conventions, and heuristics are applied by a given implementation."

The functionality provided by patterns is summarized as follows:

"Like text, patterns are sequences of characters, representing one of the following:

- a single word (patterns of the form <word>),
- a set of words (patterns of the form <word> with wild card characters, patterns of the form <token list>,
- patterns of the form <stemmed word>, patterns of the form <expansion function invocation>, or certain patterns of the form <text literal list>),
- a phrase, i.e. a representation of a sequence of words (patterns of the form <phrase>),
- a set of phrases (patterns of the form <phrase> with wild card characters, patterns of the form <stemmed phrase>, patterns of the form <expansion function invocation>, or certain patterns of the form <text literal list>),
- a set of words and/or phrases (patterns of the form <text literal list> or patterns of the form <expansion function invocation>),
- sets of two or more patterns, each either consisting of a single word or phrase, or a set composed of context patterns (patterns of the form <Proximity expansion>, or patterns of the form <context condition>),
- patterns formed by patterns and Boolean operators for negation, conjunction, or disjunction (patterns of the form <search expression> | <search term>, patterns of the form <search term> & <search factor>, or patterns of the form NOT <search primary>).

Each word pattern and single phrase pattern is either explicitly or implicitly associated with a specific language."

The category <expansion function invocation> provides considerable flexibility in searching for terms which are:

- similar in sound to the generating term,
- broader terms for the generating term,
- narrower terms for the generating term,
- synonyms of the generating term,
- preferred terms for the generating term,
- related to the generating term,
- top terms of the generating term.

The SQL/MM Full-text functionality is appropriate to many languages, but of course not to all. However, it is somewhat representative of the state of the art, being the outcome of several years of work by experts in the field, in attempting to specify an acceptable international (ISO) standard. It does not by any means solve all internationalization problems, and many of its results are implementation-defined. Even so, it provides a standard way to frame certain questions, even where the answers are not always determined by the standard.

5. Conclusion

Exploiting the similarities between the SQL-99 data model and a subset of the XML Query data model, XSQL extends the syntax and semantics of a subset of SQL-99 in three main ways:

- collections are treated more generally;
- paths are generalized by adopting XPath functionality in a slightly modified syntax;
- SQL/MM Full-text pattern searching is provided in the form of a Contains function.

This snapshot of the design of XSQL has shown the language poised between the reasonably successful XSQL solutions to the XML Query Use Cases encountered so far, and the forthcoming interaction with the XML Query Algebra as it is developed. The latter interaction should preferably be a two-way influence between the algebra and various possible syntaxes that may eventually be adopted as user-friendly and efficiently-implementable realizations of the algebra.

References

[1] Hoare, C.A.R.: Hints on programming language design. Stanford, Dec. 1973. ftp://elib.stanford.edu/pub/reports/cs/tr/73/403/CS-TR-73-403.pdf

[2] Bray, T., Paoli, J., and Sperberg-McQueen, C.M. (eds.): Extensible Markup Language (XML) 1.0. W3C Recommmendation. Feb. 1998.
http://www.w3.org/TR/1998/REC-xml-19980210

[3] Fernandez, M., and Robie, J. (eds.): XML Query Data Model. May 2000. http://www.w3.org/TR/query-datamodel/

[4] W3C Architecture Domain: XML Query. http://www.w3.org/XML/Query.html

[5] Fankhauser, P., Marchiori, M., and Robie, J. (eds.): XML Query Requirements. W3C Working Draft. Jan. 2000.
http://www.w3.org/TR/2000/WD-xmlquery-req-20000131

[6] ISO/IEC: Database Language SQL -Part 2: Foundation (SQL/Foundation). 1999. ISO/IEC IS 9075-2:1999 (E)

[7] ISO/IEC: SQL Multimedia and Application Packages - Part 2: Full-Text. Feb. 2000. ISO/IEC FDIS 13249-2:2000 (E)

[8] XML Query Use Cases. June 2000. (Currently accessible to W3C XML members only at http://www.w3.org/XML/Group/xmlquery/usecases/)

[9] Clark, J., and DeRose, S. (eds.): XPath. W3C Recommendation. Nov. 1999. http://www.w3.org/TR/xpath

[10] Cowan, J., and Megginson, D.. (eds.): XML Information Set. W3C Working Draft. Dec.1999.

http://www.w3.org/TR/xml-infoset

[11] Cattell, R.G.G., and Barry, D.K. (eds.): The Object Database Standard: ODMG 2.0. Morgan Kaufmann, San Francisco. 1997.

[12] Thompson, H.S., Beech, D., Maloney, M., and Mendelsohn (eds.): XML Schema.Part 1: Structures. W3C Working Draft. Apr. 2000..
http://www.w3.org/TR/xmlschema-1/

[13] Elmasri, R.: On the Design, Use, and Integration of Data Models. Ph.D. thesis, Stanford, May 1980. STAN-CS-80-801.

[14] Robie, J., Chamberlin, D., and Florescu, D.: Quilt: an XML Query Language. Mar. 2000.
http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

Appendix A. XSQL Grammar

[Draft: Work in Progress]

Extensions to SQL-99 are shown in bold. Restrictions to SQL-99 are shown by strikethrough.

The approach taken in formulating this grammar is to show the extensions and restrictions as simply as possible for purposes of exposition. This may not necessarily be the best approach for particular eventual standards specifications, or for implementation purposes. For example, where an extension is desired to some deeply nested category in the grammar where the extension is only valid in certain contexts (e.g. within the scope of a FOR clause rather than a FROM clause), it is assumed that there is a general statement that these extensions are not available within the scope of a FROM clause, rather than duplicating the structure of many existing rules in order to enforce the constraints within the grammar.

QUERY EXPRESSION

```
<query expression> ::=
    [ <with clause> ] <query expression body> [ <order by clause> ]
  |  <XSQL result clause> [ <with clause> ] <query primary> [ <order by clause> ]

***check positioning of parens

<XSQL result clause> ::= AS <XSQL result name>

<XSQL result name> ::= <identifier>

<with clause> ::= WITH [ RECURSIVE ] <with list>

<with list> ::=
  <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
```

```
  <query name>
***mark omission of column names
  AS <left paren> <query expression> <right paren>
  [ <search or cycle clause> ]

<query expression body> ::=
    <non-join query expression>
  | <joined table>

<non-join query expression> ::=
    <non-join query term>
  | <query expression body> UNION [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query term>
  | <query expression body> EXCEPT [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query term>

*** add CONCAT with some precedence

<query term> ::=
    <non-join query term>
  | <joined table>

<non-join query term> ::=
    <non-join query primary>
  | <query term> INTERSECT [ ALL | DISTINCT ]
    [ <corresponding spec> ] <query primary>

<query primary> ::=
    <non-join query primary>
  | <joined table>

<non-join query primary> ::=
    <simple table>
  | <left paren> <non-join query expression> <right paren>

<simple table> ::=
    <query specification>
  | <table value constructor>
  | <explicit table>
```

QUERY SPECIFICATION

```
<query specification> ::=
  SELECT [ <set quantifier> ] <select list>
    <table expression>

<select list> ::=
```

```
    <asterisk> |
    <select sublist> [ { <comma> <select sublist> }... ]
  | <left paren> <select sublist> [ { <comma> <select sublist> }... ] <right paren>
[ <as clause> ]

<select sublist> ::=
    <derived column>
  | <qualified asterisk>

<derived column> ::=
  <value expression> [ ATTRIBUTES_ONLY ] [ <as clause> ]
```

TABLE EXPRESSION

```
<table expression> ::=
  <from clause> | <for clause>
  [ <where clause> ]
  [ <group by clause> ]
  [ <having clause> ]

<for clause> ::=
  FOR <iterator list>

<iterator list> ::=
  <collection reference> [ { <comma> <collection reference> }... ]

<iterator> ::=
    <correlation name> IN <table or query name>
  | <correlation name> IN <derived table>
  | <correlation name> IN <lateral derived table>
  | <correlation name> IN <collection derived table>
  | <correlation name> IN <only spec>
  | <left paren> <joined table> <right paren>

<only spec> ::=
  ONLY <left paren> <table or query name> <right paren>

<lateral derived table> ::=
  LATERAL <left paren> <query expression> <right paren>

<collection derived table> ::=
  UNNEST <left paren> <collection value expression> <right paren>
  [ WITH ORDINALITY ]

<derived table> ::= <table subquery>

*** extend to more general coll val exp
```

```
<table or query name> ::=
    <table name>
  | <query name>

<derived column list> ::= <column name list>

<column name list> ::=
  <column name> [ { <comma> <column name> }... ]

<table subquery> ::=


TABLE VALUE CONSTRUCTOR

<table value constructor> ::=


WHERE CLAUSE

<where clause> ::= WHERE <search condition>

<search condition> ::= <boolean value expression>


GROUP BY CLAUSE

<group by clause> ::=
  GROUP BY <grouping specification>

<grouping specification> ::=
    <grouping column reference>
  | <rollup list>
  | <cube list>
  | <grouping sets list>
  | <grand total>
  | <concatenated grouping>

<grouping column reference> ::=
  <column reference> [ <collate clause> ]

<column reference> ::=

<collate clause> ::=


HAVING CLAUSE

<having clause> ::= HAVING <search condition>
```

VALUE EXPRESSION

```
<value expression> ::=
*** add paths (and -> for foreign keys) and Contains()

<boolean value expression> ::=
```

SEARCH OR CYCLE CLAUSE

```
<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
  | <search clause> <cycle clause>

<search clause> ::=
  SEARCH <recursive search order> SET <sequence column>

<recursive search order> ::=
    DEPTH FIRST BY <sort specification list>
  | BREADTH FIRST BY <sort specification list>

<sequence column> ::= <column name>

<cycle clause> ::=
  CYCLE <cycle column list>
  SET <cycle mark column> TO <cycle mark value>
  DEFAULT <non-cycle mark value>
  USING <path column>

<cycle column list> ::=
  <cycle column> [ { <comma> <cycle column> }... ]

<cycle column> ::= <column name>

<cycle mark column> ::= <column name>

<path column> ::= <column name>

<cycle mark value> ::= <value expression>

<non-cycle mark value> ::= <value expression>
```

ORDER BY CLAUSE

```
<order by clause> ::=
  ORDER BY <sort specification list>

<sort specification list> ::=
  <sort specification> [ { <comma> <sort specification> }... ]

<sort specification> ::=
  <sort key> [ <collate clause> ] [ <ordering specification> ]

<sort key> ::=
  <value expression>

<ordering specification> ::= ASC | DESC

<collate clause> ::= COLLATE <collation name>

<column name> ::=
  [<at sign>] <identifier>

<at sign> ::= @
```

Rules as in SQL-99

```
<identifier> ::=

<comma> ::= ,

<query name> ::=

<left paren> ::=

<right paren> ::=

<explicit table> ::= TABLE <table name>

<corresponding spec> ::=
  CORRESPONDING [ BY <left paren> <corresponding column list> <right paren> ]

<corresponding column list> ::= <column name list>

<column name list> ::=

<column name> ::=

<collation name> ::= <identifier>

<set quantifier> ::=
```

```
    DISTINCT
  | ALL

<period> ::=

<as clause> ::= [ AS ] <column name>
```