# DB2/XML: Designing for Evolution

Kevin Beyer Fatma Özcan Sundar Saiprasad Bert Van der Linden

IBM Almaden Research Center
{kbeyer,fozcan} almaden.ibm.com

IBM Silicon Valley Lab
{saipras,robbert} us.ibm.com

## ABSTRACT

DB2 provides native XML storage, indexing, navigation and query processing through both SQL/XML and XQuery using the XML data type introduced by SQL/XML. In this tutorial we focus on DB2's XML support for schema evolution, especially DB2's schema repository and document-level validation.

## 1. INTRODUCTION

The database research community continues to debate the merits of XML as a storage format. Few question its use as a wire protocol, but a significant number of researchers believe that XML should be shredded into relational tables. On the contrary, we believe that native XML support addresses a major pain-point for the industry, for exactly the same reasons that XML is used for communication: XML enables extensibility and loose coupling through schema variation and evolution.

The changes to most services are carefully planned, and each service is not changed very often. The reorganization, addition, or removal of bits of information is debated among the controlling parties. However, in aggregate, the number of changes faced by an organization is quite large. Moreover, the changes must be trickled out to ends of the system. The more wide-spread the use of the message is, the longer it takes to complete the move to the next standard. Think of how long the U.S. has been transitioning to HDTV, or the Internet to IPv6. Large-scale distributed systems do not have an atomic commit; they *evolve*.

In this tutorial we focus on DB2's XML support for schema evolution, including DB2's schema repository, document-level validation, and document schema detection. We will first give an overview of DB2 XML support in the next section. Then, the tutorial follows a fictitious TeeNee Bank through the evolution of an autonomous Web service that delivers crucial information to its applications.

## 2. DB2 XML SUPPORT OVERVIEW

DB2 provides native XML storage, indexing, navigation and query processing through both SQL/XML [4, 3] and XQuery [10]. DB2 stores native XML data in columns of relational tables. The physical storage format for the XML type preserves all the information in the XQuery data model, thus supporting *XML fidelity* as defined in the tutorial introduction. In addition, DB2 allows XML data to be "shredded" into relational form, supporting *relational fidelity*, or stored in a CLOB column, supporting *textual fidelity*. An important feature of DB2 is that it does not require an XML schema to be associated with an XML column. An XML column can store documents validated according to many different and evolving schemas, as well as non-validated documents, all in the same column. Hence, the association between schemas and XML documents is per document, providing maximum flexibility.

Applications can validate documents during insertion, or later in a query with the `xmlValidate` function. XML nodes in a validated document are annotated with their primitive types and the run-time uses this information for dynamic dispatch of functions during query processing. DB2 provides an XML schema repository, which stores XML schemas natively as XML documents. XML schemas used for validation or query processing need to be registered with this repository.

A DB2 application can access XML data using either SQL/XML or XQuery. The two languages are composable: SQL can be invoked from XQuery, and XQuery can be invoked from SQL. The key to this reciprocal behavior is the new XML data type, which is based on the XQuery data model (QDM) [11]. Supporting arbitrary QDM results from these functions enables the user to seamlessly transition back and forth between SQL and XQuery.

SQL programmers manipulate XML data using XQuery subqueries from SQL's `xmlQuery`, `xmlExists`, and `xmlTable` functions. The XQuery arguments to `xmlTable`, `xmlQuery` and `xmlExists` can be arbitrarily complex, including FLWORs and joins with other tables. SQL also provides `xmlCast` to convert XML data into SQL data, and the XML publishing functions (e.g., `xmlElement`) to create new XML data [4].

DB2 supports a stand-alone XQuery interface and provides access to XML data stored in relational tables through two input functions: `db2-fn:sqlquery` allows XQuery to invoke a SQL subquery that returns XML data as a sequence of XML nodes, and `db2-fn:xmlcolumn` simply imports an entire XML column.

DB2 supports indexes on XML path expressions, which makes the indexes significantly faster to maintain as compared to indexes on the entire document. These path expressions can contain wildcards, descendant axis navigation, and kind tests. Since XML documents in a column may contain untyped values and these values have polymorphic behavior in XQuery predicates, each index has a specific data type, and all elements and attributes represented in the index are cast into the given data type. Values that cannot be cast into the designated type are ignored, i.e. not represented in the index.

As a semi-structured data model, XML is a bridge between the rigid structural world of relational systems and the free-form world of text documents. Full-text indexing of XML data is required to complete that bridge. Therefore, DB2 is extending the relational text indexing support [5] to include XML data.

Another important design decision, targeted for schema evolution, is not to support schema import or static typing features. As we will explain in this tutorial, these two features are restrictive as they do not allow conflicting schemas. Additional examples and details on the internals of the system are available elsewhere in these proceedings [1].

## 3. TEENEE BANK EXAMPLE

TeeNee Bank created a simple database schema that is designed to evolve with their business. They use XML Web services to interact with autonomous organizations to acquire reference data. TeeNee has many applications and uses many Web services, so the system is always in flux. The tutorial will follow the evolution of a part of TeeNee's database and their `Loan` application as the `Interest` Web service changes.

Every day, TeeNee uses an XML Web service[1] to acquire the prime interest rate from the U.S. Federal Reserve. The `Interest` table stores the current and historical interest rate information. Multiple documents can be simultaneously current, but each current `doc` will conform to a different XML schema. The `best` column marks the `doc` that came from the most recent version of the service on the day the data was retrieved with a 'Y', or 'N' for redundant data from deprecated services.

```
create table Interest(
   id      int not null,      -- unique id
   until   timestamp,         -- not null when historical
   best    char(1) not null,  -- marks latest spec at the time
   doc     xml not null)      -- interest data for one day
```

Notice that the `create table` statement simply specifies "`xml`" as a data type and does not mention any schemas. Hence, an XML column accepts any well-formed XML document. Inserted documents can be optionally validated against an XML Schema [9].

TeeNee registers the schema requirements of every application in the `AppUse` table. The application registry is used to identify the applications that need to be considered whenever a particular schema is evolving.

```
create table AppUse(
   app        varchar(17) not null, -- application id
   service    varchar(17) not null, -- service id
   schemaOid  bigint not null,     -- supported DB2 schema id
   notify     varchar(50) not null) -- email of app owner
```

[1]This service is invented for the tutorial, but this data is currently published in HTML [7])

## 4. EVOLVING WEB SERVICES VIA DB2

### 4.1 The Evolve Protocol

TeeNee and its business parters are members of the Imaginary Web-service Evolution Foundation (IWEF), whose goal is to define protocols that enable highly distributed, autonomous Web services to change peacefully[2]. The IWEF defines the Evolve Web service response header that notifies callers when a service has been deprecated. The message is defined by the `evolve-v1` schema[3]:

```
namespace: bogus://iwef.org/evolve
schema:    bogus://iwef.org/evolve-v1.xsd
evolve
  service    anyURI               -- global service identifier
  deprecated optional             -- marks an expiring service
    expires    dateTime           -- when service terminates
    replacedBy  anyURI optional   -- replacement service
    reason      string repeating  -- machine explanation
    detailsAt   anyURI optional   -- human explanation
```

Every `Evolve` response echoes back the service URI, which includes the version of the service. The main service response message includes the schema of the response the message itself (using the `schemaLocation` attribute). `Evolve` mandates that the URIs change whenever a new version of the service or schema is created. When the service is deprecated, the service notifies its callers by including the `deprecated` tag that includes the details of the change.

The `replacedBy` tag gives the URI of the new version of the service. The `reason` tags provide machine-readable details of the changes to the service using the IWEF's dictionary of terms. We will consider two reasons: `new-required-tags` and `major-revision`. The most critical part of the deprecation message is the `expires` tag that tells the caller how long they have to update to a more recent version. The server runs multiple versions of the service concurrently, but eventually terminates old services after giving its clients time to upgrade their systems.

The `evolve-v1` schema is registered with DB2 using the DB2-specific command, `register xmlSchema`. DB2's schema repository stores all the schemas that can be referenced by other DB2 statements in the system catalog tables. DB2 provides `register`, `alter`, and `drop` schema commands for the schema repository, as well as a way to validate documents against the repository. The schema repository provides quick and guaranteed access to the schemas, and protects the system from unexpected changes to the schemas.

```
register xmlSchema 'bogus://iwef.org/evolve-v1.xsd'
 from 'evolve-v1.xsd' as evolveV1 complete
```

Each distinct `Evolve` deprecation message is stored in the `Evolve` table (not shown) to track the evolution of the Web service.

### 4.2 Initial Service

TeeNee acquires the daily interest rate information from the Fed using a Web service. The first version of the service

[2]The W3C and OASIS are real organizations that define WSDL [8] and UDDI [6], real messaging protocols that support versioning. See [2] for more information.

[3]The actual schemas are expressed in XML Schema. To simplify the presentation, we are using our own schema syntax. The indentation denotes the XML hierarchy.

returns a result that conforms to the `interest-v1` schema, as defined by the Fed and registered with DB2:

```
namespace: bogus://fed.gov/interest
schema:    bogus://fed.gov/interest−v1.xsd
interest
  asOf   dateTime
  prime  decimal
```

Every day, TeeNee's `FedUp` application connects to the Fed to update the interest rate information. Two XML documents are returned, an `Evolve` document and an `interest-v1` document. `FedUp` inspects the `Evolve` message for a `deprecated` tag. In this case, the service is not deprecated, so `FedUp` simply marks the previous interest rate as historical and inserts the new interest rate document into the `Interest` table. No changes are required to the schemas.

```
−− The interest rate document from the Fed
<interest
 xmlns="bogus://fed.gov/interest"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
 xsi:schemaLocation="bogus://fed.gov/interest
                     bogus://fed.gov/interest−v1.xsd">
 <asOf>2004−12−13T07:00:00−05:00</asOf>
 <prime>5.501</prime>
</interest>
```

```
−− Mark previous messages as historical.
update Interest
set until = current timestamp
where until is null;

−− Add the latest information.
insert into Interest(id,doc) values
  (nextval for idGenerator, −− Generate unique id
   xmlValidate( ? )) −− XML parameter marker
```

The `insert` statement the `xmlValidate` function to validate the XML document against a schema. `xmlValidate` allows the schema to be specified in several ways. In this tutorial, we identify the schema from the document itself (using the `schemaLocation` attribute). This method has two advantages: the data is self-describing if it identifies the schema to which it conforms, and the documents can evolve to a new schema without changing the `insert` statement. Other forms of `xmlValidate` allow the DB2 identifier or the XML schema location to be specified; these forms allow validation of documents that do not contain an `schemaLocation` attribute, and allow the `insert` statement to ensure the document conforms to a specific schema.

TeeNee's applications that use the `Interest` table are prepared for multiple versions to exist in the table. For example, the Loan application retrieves the current prime interest rate using `xmlTable` in the following SQL query. For each item in the sequence produced by the first XQuery subexpression, `xmlTable` produces a result row. Each item in the sequence is passed to the column expressions to produce the column values.

```
select prime
from Interest,
 xmlTable( −− creates a table from XML data
   −− declare namespaces
   xmlNamespaces('bogus://fed.gov/interest' as "i"),
   '$d/i: interest '       −− one row per item from this XQuery
   passing doc as "d"  −− pass SQL value to XQuery
   columns              −− defines result column(s)
     prime decimal(7,4) path './i:prime'
 ) as t
where until is null −− is current interest data
 −− correct schema version for application
 and xmlXsrObjectId(doc) = appUses('Loan', 'Interest')
```

The `until` column is null for the most recent version. The `xmlXsrObjectId` function returns the identifier of the schema that was used to validate the document. The statement uses the built-in DB2 function `xmlXsrObjectId` to identify the schema used to validate the XML document.

The Loan application requires its registered version of the Interest schema found using the `appUses` user-defined function, which is defined as follows:

```
create function appUses(aApp varchar(50),
                        aService varchar(50))
 returns bigint return select schemaOid from AppUse
                 where app = aApp and service = aService
```

The `appUses` function reads the `AppUse` configuration table, which records the current schema accepted by a particular application. In this case, the Loan application uses version 1 of the interest schema. The user-defined function `xsrId` uses the schema repository to find the internal identifier for a "schema location" URI.

```
insert into AppUse(app, service, schemaOid, notify)
values('Loan', ' Interest ',
  xsrId('bogus://fed.gov/interest−v1.xsd'), 'it@TeeNee.com')
```

In the previous example, the user started the query in SQL and invoked XQuery. DB2 also allows the user to start in XQuery and invoke SQL. The `db2-fn:sqlquery` function returns an arbitrary QDM to XQuery from a SQL subquery. The following XQuery returns nearly the same result as the previous SQL query:

```
xquery
declare namespace i="bogus://fed.gov/interest";
for $i in
  db2−fn:sqlquery("
    select  doc from Interest
    where until  is  null
       and xmlXsrObjectId(doc) = appUses('Loan','Interest')")
return $i/i:interest/i:prime/data(.)
```

DB2 allows SQL views to contain XML data types and use XML functions like `xmlQuery` and `xmlTable`. The views can be used like any other table. To simplify the evolution of its applications, TeeNee wisely decided to insulate the applications from unnecessary information by using a view. More sophisticated applications can still access the full XML data when necessary. The final query below is how the Loan application will access the current interest rate.

```
create view LoanInterest as (xmlTable SQL query above)

select prime from LoanInterest
```

## 4.3 Compatible Changes

The Fed decided to improve their interest rate service by including the interest rate on Treasury Bills (T-Bills). They started version 2 of the service and deprecated version 1. When TeeNee connected to the deprecated service, the `Evolve` message informed them of the change.

```
<evolve xmlns="bogus://iwef.org/evolve"
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
```

```
 xsi:schemaLocation="bogus://iwef.org/evolve
              bogus://iwef.org/evolve−v1.xsd">
 <service>bogus://fed.gov/interest−v1.php</service>
 <deprecated>
  <expires>2005−03−23T20:59:59−08:00</expires>
<replacedBy>bogus://fed.gov/interest−v2.php</replacedBy>
  <reason>new−required−tags</reason>
  <detailsAt>bogus://fed.gov/new−in−v2.html</detailsAt>
 </deprecated>
 </evolve>
```

When the Evolve message includes a deprecated tag, FedUp looks for a duplicate message in the Evolve table to see if the Evolve message was previously handled. If the message is new, then FedUp inserts the new Evolve message into the Evolve table and notifies the owners of the affected applications by reading the notify column of the AppUse table.

The Evolve message specified that the only change to the schema was that new required elements or attributes were added (new-required-tags). TeeNee requires that all of its applications be insensitive to the addition of new tags. For example, TeeNee prohibits the use of queries like /interest//* when the meaning would change as new tags are added. As far as TeeNee is concerned, the version 2 schema is a perfect replacement for version 1, even though the old data will not validate with the new schema. FedUp detects this case and automatically updates all the applications from version 1 to version 2.

```
−− Auto−update from v1 to v2.
update AppUse
set schemaOid = xsrId('bogus://fed.gov/interest−v2.xsd')
where service='Interest'
  and schemaOid = xsrId('bogus://fed.gov/interest−v1.xsd')
```

FedUp calls the new service to get the new document and schema. The new schema is registered with DB2 and the new document is inserted into the same Interest table.

```
namespace: bogus://fed.gov/interest
schema:    bogus://fed.gov/interest−v2.xsd
interest
  asOf   dateTime
  prime  decimal
  tBill  repeating −− new required name
    months int
    rate   decimal
```

XML has solved the *data evolution* problem: new information can be added to the database without redesigning the database, and the new data can coexist with the old data. Not migrating the old data is particularly important when the data changes frequently, the amount of historical data is large, or when digital signatures are used to verify the data's authenticity.

We believe that the majority of schema changes will be the inclusion of additional information. As long as an application does not access the new data, or just passes it along without inspecting it, then the application does not require modification as new data is added.

## 4.4 Incompatible Changes

After some time, the Fed again reported a change in the Interest service. This time, the Evolve deprecation message included a major-revision notice, and the FedUp application could not automatically update the applications to use the new schema, so the IT department was notified.

FedUp registers the version 3 schema with DB2, and begins collecting both version 2 and version 3 messages. Both messages are inserted into to the same Interest table – avoiding DBA intervention – with best set to 'Y' for the version 3 row.

```
namespace: bogus://fed.gov/interest
schema:    bogus://fed.gov/interest−v3.xsd
interest
  asOf   dateTime
  country repeating −− new repeating name
    id     string −− adds new context to data
    prime  decimal −− moved deeper in tree
    tBill  optional repeating −− deeper, optional
      months int
      rate  decimal
```

In this case, the Fed added a new level into the XML tree and started reporting international interest rates. Just as application developers are advised to use "select *" carefully, so should they consider the use of features like the double-slash descendant operator and wildcard name tests (both are perhaps a little too convenient). In version 3, the query //i:prime would still return the U.S. prime interest rate, as in version 2, but it would also return the prime interest rate in other countries, causing TeeNee's loan application to fail.

After examining the version 3 schema, the TeeNee IT department determined that the Loan application needed attention. An easy change to the application-specific view definition and an update to the application registry solved the problem.

```
drop view LoanInterest

create view LoanInterest as (
  select prime
  from Interest, xmlTable(
    xmlNamespaces('bogus://fed.gov/interest' as "i"),
    −− extra level and predicate added
    '$d/i:interest/i:country[i:id="US"]' −− only change
    passing doc as "d" columns
      prime decimal(7,4) path './i:prime') as t)
  where until is null
  and xmlXsrObjectId(doc) = appUses('Loan','Interest'))

−− The loan application now uses interest version 3.
update AppUse
set schemaOid = xsrId('bogus://fed.gov/interest−v3.xsd')
where app='Loan' and service='Interest'
```

TeeNee has many applications using the interest data. Updating all of the applications in lock-step is inconvenient and sometimes impossible. For example, if the new service eliminated some important information, then an application may need to find a new source for the information, or be deprecated itself. This may cascade and cause the users of that application to also evolve. By allowing the applications to evolve independently, the service expiration time can be more effectively exploited. Moreover, the user may wish to create new applications that use the new data before all the old applications are corrected.

## 4.5 Queries over Multiple Versions

Queries that span schema versions are more difficult to evolve than queries that only work on the current version because they must unify all the versions. One way to solve the problem is to create a view per version that maps each version into a "universal document" that can hold all the information from every version. For example, the version 1 and version 2 interest data can be transformed into a version 3 schema. This approach is taken by many integration systems. A disadvantage in the case of XQuery is that node construction is a costly operation and has side-effects – e.g., it changes node identity.

A second approach that avoids construction is to deal with the differing versions as necessary. For example, the following XQuery finds yearly interest rate statistics over all three versions of the interest schema. Most of the query is common to all versions because the `asOf` tag is in the same place in every document. The `prime` tag moved between versions 2 and 3, so two different path expressions are used to find this data.

```
-- Find the minimum, average, and maximum U.S. prime
-- interest rate for each year.
xquery
declare default element namespace "bogus://fed.gov/interest";
-- Get the best version of each daily interest data.
let $in := db2-fn:sqlquery("
      select doc from Interest where best = 'Y'
  ")/interest
-- Iterate over the different years.
for $year in distinct-values(
                    year-from-dateTime($in/asOf))
-- Get all the documents for this year.
let $group := $in[year-from-dateTime(asOf) = $year]
-- Get the prime interest rate from each document.
-- This is the only version-specific part
let $prime := (
      $group/prime,                       -- version 1 and 2
      $group/country[id = "US"]/prime) -- version 3
-- Sort by the year.
order by $year
-- Aggregate the yearly prime interest rate
return <yearly-interest-summary>
          <year>{ $year }</year>
          <min>{ min($prime) }</min>
          <avg>{ avg($prime) }</avg>
          <max>{ max($prime) }</max>
       </yearly-interest-summary>
```

## 5. XML INDEXES AND EVOLUTION

Indexes are interesting in the presence of multiple, conflicting schemas. Consider a numeric index on `//i:prime` on the `Interest.doc` column. For versions 1 and 2, this index works as expected. The index is also useful for interest version 3 because of the "//", although it is now indexing significantly more information. DB2 also handles more complicated scenarios. For example, consider schema version 3b, an alternative to version 3:

```
interest ...
  prime   repeating -- changed to complex type
    country  string -- country now below prime tag
    value    decimal -- relabeled, deeper in tree
  ...
```

Now the index on `//i:prime` no longer finds numbers for this pattern in version 3b, but rather a complex type. The XQuery specification says that a comparison of `//i:prime` with a number will result in an error because `prime` has a complex type in version 3b. To support schema evolution, DB2's indexes ignore values that do not comply with the index's target type, which effectively treats the predicate that caused an error as false when using the index. This is important for performance and desirable when the query has another predicate to select the appropriate schema.

## 6. CONCLUSIONS

The design of database systems for evolution, and complete application stacks in general, is still evolving. This is a continuous theme in computer science; abstract data types, SQL views, software components, XML, and Web services all have evolution at their core. Looking forward in XML databases, we see a need for more support for versioned schemas, especially in the query languages. We need systematic application dependency tracking, beyond the do-it-yourself approach of TeeNee.

We're database researchers; we like atomic commits and referential integrity. But we need to open our database systems to the real, dirty world. The *X* in XML stands for *eXtensible*; XML database systems need to grow to support this extensibility. Welcome to the primordial data soup.

**Acknowledgments**

This paper would not be possible without the amazing effort by the entire IBM DB2 XML Research and Development team. This paper represents the effort of many people who contributed to the DB2 XML support from the parser to serialization and all code in between. In particular, we would like to acknowledge David Bertoni, Seeling Cheung, Michelle Jou, and Brian Vickery for their design and implementation work that is highlighted in this paper. We also wish to thank Don Chamberlin, Bobbie Cochrane, Latha Colby, and Hamid Pirahesh for their thoughts on evolution and comments on this paper.

## 7. REFERENCES

[1] K. Beyer et al. System RX: One part relational, one part XML. In *Proc. of ACM SIGMOD*, 2005.

[2] K. Brown and M. Ellis. Best practices for web services versioning. See `http://www-106.ibm.com/developerworks/webservices/library/ws-version/`.

[3] A. Eisenberg and J. Melton. Advancements in SQL/XML. *Sigmod Record*, 33(3):79–86, 2004.

[4] I. O. for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML).

[5] A. Maier and D. Simmen. DB2 optimization in support of full text search. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[6] OASIS. UDDI. See `http://www.uddi.org/`.

[7] F. R. S. Release. See `http://www.federalreserve.gov/releases/h15/update/`.

[8] W3C. WSDL. See `http://www.w3.org/TR/wsdl20/`.

[9] W3C. *XML Schema*, February 2002. See `http://www.w3.org/XML/Schema`.

[10] W3C XML Query Working Group. See http://www.w3.org/XML/Query.

[11] *XQuery 1.0 and XPath 2.0 Data Model*, February 2005. W3C Working Draft, See `http://www.w3.org/TR/xpath-datamodel`.