Managing Structure in Bits & Pieces: The Killer Use Case for XML

Eric Sedlar
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
eric.sedlar@oracle.com

ABSTRACT

This paper asserts that for databases to manage a significantly greater percentage of the world's data, managing structural information must get significantly easier. XML technologies provide a widely accepted basis for significant advances in managing data structure. Topics include schema design, evolution, and versioning; managing related applications; and application architecture.

1. INTRODUCTION

One of Oracle's primary goals over the entirety of its existence has been to find ways to get more information into Oracle databases. Unfortunately, today most estimates places less than 20% of the data in the world in any relational database, let alone Oracle. Oracle has made advances, adding capabilities for many new data types over the past few releases. Oracle 10g natively supports user-defined objects based on the SQL99 model, multimedia datatypes & indices, multidimensional cubes & analytic operators, and filesystem protocols—capabilities that cover the structures of most information in the world. However, most information today remains stored in filesystems.

The reasons for this are no longer a lack of capabilities in Oracle, or any other relational database for that matter. The primary reason is the cost of utilizing the database's capabilities in a significant way. The cost I am referring to is not the software licensing cost—it is the cost to the organization of building and deploying a relational application.

2. THE COST OF STRUCTURE

The typical database application development process today looks something like the following:

1. Gather requirements
2. Design schema
3. Build application code
    b. If this is the second version, handle schema & application upgrade issues (generally requiring data migration)
4. Load data & deploy application
5. Tune performance
6. Gather feedback based on actual data & usage and go back to step 2 to create the second version.

Most of the languages, tools and techniques developed over the past few decades focused on the cost of application development. Significant progress has been made in reducing the cost of step 3. Much less progress has been made at using technology to decrease the costs of the other steps. Since user requirements directly drive schema design, the problem is not really a technical one. Figuring out what people want before they see the actual thing is an intractable problem of human nature.[1]

Currently, it is our experience that requirements gathering, schema design, and upgrade costs are far more than application development costs. The basic reason for this is that the cost of making a mistake in schema design and correcting it is very high—far higher than any other cause. In many cases, desirable schema changes are impossible without throwing away old data—the structural information required does not exist at all in the older data. Since relational technology doesn't allow for tables with heterogeneous structure, the information is effectively gone. Generally, current applications must try to anticipate all requirements and structural requirements up front.

An added complication to the current application lifecycle is in dealing with multiple applications with overlapping information. In practice, this is the usual case—a solitary application with no overlap with others is rare. The set of related applications may be loosely coupled, exchanging data via techniques such as web services, or they may be tightly coupled, sharing database schema objects (such as the components of the Oracle E-Business Suite, or SAP R-3). Providing high levels of functionality and performance often requires tightly coupling applications, which is why the major application vendors offer a single integrated product suite. However, this means that all of the components must agree on the schema design for shared components in advance. This negotiation between various groups can be quite tedious. A similar process goes on in industry standards bodies that define common semantics for documents commonly exchanged in a particular industry segment. When multiple

[1] Many business problems beyond data management could benefit from improvements in anticipating what people want—deciding what movies to greenlight and Internet dating could probably gain an even greater benefit.

applications have overlapping schemas, the structural design cost grows exponentially.

Various techniques are commonly employed to allow some schema flexibility. In simple cases, object-oriented techniques such as single inheritance can be used. A version 2 object might just inherit from the version 1 object. This can be implemented in the database via object-relational techniques such as the SQL99 support directly in Oracle, or in via middle-tier object mappers such as Oracle TopLink. Another common technique is to use name/value pairs for physical storage of attributes. (This is use in many content management systems).

Unfortunately these techniques have limitations—name/value pair storage often results in unacceptably slow performance and loss of stricter datatyping when desired. Name/value pair storage may not be enough for complicated data structures involving nesting, or those where order is important. Single inheritance often is too restrictive a way to evolve structures—open content models may be required.

3. SHARED SCHEMA OBJECTS

The largest cause of structural complexity is multiple application components with shared schema objects. Each of these components may have different assumptions about the data structure. I refer not just to the case of large applications like the Oracle E-Business Suite with many modules. Different versions of the same application may be considered to be different applications as well. In either case, the applications have some common knowledge about the data structures. This common data structure could be defined by a industry-specific XML standard, it could be defined by lots of internal design meetings between application development teams, or it could be defined by the shared source code between different application versions.

The most common way to manage shared data structures is to use optional attributes (or attributes with default values), which can be ignored by those applications that aren't interested in them. This allows any application to create an instance of the shared data structure. WebDAV properties and HTML markup are common examples where unknown structure is ignored.

If each application only needs to read & update instances from the other applications (and not create them), this requirement can be relaxed somewhat. The collection of objects in question can be a set with related schemas sharing those common elements—they don't all need to be defined using the same schema. The only thing that is required is that there is some common set of structural elements with common semantics to query on.

Schema versioning is just one particular scenario involving collections of related schemas. In this case, the common structure is generally that defined by the first version of the schema, as opposed to being defined by a standard vocabulary.

Application integration is another scenario with related problems. In the loosely coupled application case, documents from one application may be exchanged via web services, and transformed into the schema of a second application. Transformation is generally necessary if the schemas for related objects were developed completely independently. However, if the applications are exchanging documents extending a standard, there may not be a need for transformations before loading data.

4. XML STRUCTURAL DEFINITION

XML has a different method of structural definition than relational or object-oriented techniques. The schema design phase in those techniques is separated into two phases in XML. Phase 1 identifies a vocabulary of discrete granules of information that are of interest on their own. The granules must be named, and assigned some semantic meaning. Phase 2 defines relationships between the granules, their datatypes, and constraints on the contents. Phase 1 requires the XML 1.x and Namespace specs, while phase 2 requires XML schema (XSD, RELAX, etc.).

The phase 1 structural design is only somewhat easier in XML. We still must figure out which parts of the information are interesting. However, XML does provide for a continuum between unstructured and structured content, by allowing for mixed text, where structural tags can be introduced into unstructured text without disturbing the flow of the text. This allows for a greater degree of freedom to identify different granules of interest.

One the vocabulary of tags has been defined, XML provides a marked advantage in structural definitions. It allows for an application lifecycle that allows for the phase 2 structure to be determined even data is loaded. In some applications, it is not even possible to know much of the data structure without looking at instances, since the instance data may not be directly input via an application user interface, but exchanged via another application. Oracle has technology today in Warehouse Builder that helps determine data typing and constraint information from data that is already loaded. This approach is currently limited by the SQL type system (the data is typically in a VARCHAR column). However, the technique could easily be applied to schemaless XML data as well, to help identify interesting structural properties.

With XML, documents with a common vocabulary can be exchanged without defining a schema. The schema could be added later if the document is validated. The XML schema is also much more flexible than relational or object-oriented structural defintion. XML adds "fuzzy" schema concepts, such as open content (undefined components), mixed structured & unstructured content, and flexible substitutions of datatypes.

5. STRUCTURAL CHANGE IN XML

To decrease the cost of schema design and requirements analysis, we must allow designers more scope for schema design mistakes without imposing a large penalty. A comprehensive schema design shouldn't be required to start loading data. Once actual data is loaded and an application is available, more requirements will always arrive, changing structural needs. Allowing for more iterations of structural design should decrease costs, as hindsight is a valuable tool for defining structure.

In general, schema complexity is related to the number of object types and attributes in each object. So, simpler schemas will generally err on the side of too little structure rather than too much structure. More structure allows for more precise identification of the information we need. Over time, we may want users to answer more questions that we didn't think to ask before. (In effect, a form that evolves from essay questions to multiple-choice is one that is gaining structure.) So, the general tendency of data is to increase in structure over time.

This is not to say that it is impossible to have too much structure—it is just less common. If a particular question we ask users generally elicits nonsense replies, or the answers to the question are all the same, a question may not be worth asking. The problems of too much structure and too little structure must both be considered.

On a more technical level, schema design problems that might be corrected over time include:

- Fail to identify structure typing: the element may be untyped, or a string, when a constrained datatype like a number, enumeration or reference may be more appropriate
- Missing subcomponents of an element
- Wrong level of granularity may be used. For example, a "name" field may need to be broken up into first & last name components
- Missing elements (optional or required) or too many elements
- Datatype too simple (e.g. the value goes from attribute to complex typed element)
- Data too constrained—may be valid corner cases that violate constraints, or we may need to add values to an enumeration
- Ordering may be relevant, and we want to reorder
    - Ordering may be important for rendering
    - May impact performance (e.g. want to stop SAX parsing after a certain point)

5.1 Schema evolution

Current database practice focuses primarily on the question of schema evolution. An evolution is a change to the structure that avoids changes that create backwards incompatibilities, where the old instances no longer conform to the new schema. Examples of backwards-compatible changes include adding optional elements, or adding values to enumerations. Relational databases allow for some of these evolutions by altering a table definition. Some evolutions such as changing an attribute from single-valued to multiple-valued, are often very tedious to impossible. Some types of evolution (such as converting from a numeric datatype to a string datatype) are not allowed, even though they wouldn't break the backward compatibility restriction. This is generally an implementation issue rather than a limitation of the relational model, but is difficult nonetheless. XML allows for a somewhat more complete set of evolutions than relational, without some of these limitations.

Backwards-compatible schema evolution often removes constraints—which means that structure is decreased. Adding optional elements does increase structure, although this may create redundancies if unstructured text fields were available to hold the information in earlier versions. Unfortunately, many of the schema design "mistakes" above break the backwards-compatibility rule, and add structure. So, schema evolution by itself doesn't provide sufficient structural flexibility.

5.2 Schema versioning

If the more common and more natural case driving schema change is to add structure, schema versioning must be used to handle. Schema versioning need not be linear—clearly multiple paths of schema descent are possible (and likely) from a single starting point, be it either v1 of an application schema adapted by various internal development groups in a corporation, or a standard schema defined by some industry consortium.

The main difference between schema versioning and application integration (when similar business documents must be exchanged between totally independent applications) is the existence of a common vocabulary. In the application integration case, data with the same semantic content may use different XML tags to identify it. If the tag names are mapped to a common vocabulary via a transformation, application integration problems look very similar to managing collections of data in the schema versioning case.

5.3 Application Access to Versioned Schemas

When an application must deal with multiple versions of the schema, it must relax its assumptions about the data. Typically, when applications access a data item, they expect to know the exact location of the data, and the return datatype. When accessing XML instances with versioned schemas, this may not always be true. Luckily, XML access technologies such as DOM, XPath and XQuery allow for some structural uncertainty in the data access. In particular, the following cases might occur:

- An element may not be in the expected location
- Unexpected element tags may occur
- Constraints (such as those defined by XML Schema facets) may be violated in a different version

XPath provides techniques for handling some of these cases. For example, to access data regardless of location or nesting level, you could write "//elname", or to ignore extraneous internal tags, you could write "/elname//text()".

Applications are already built to allow for uncertainty of datatyping via polymorphism—you might have to ask an object about its datatype by asking it what interfaces it implements. In general, though, application code will need to anticipate schema variation and be robust enough to deal with whatever variation is allowed. XML clearly doesn't solve all of the problems in this space.

5.4 Schema Versioning Limitations

All this being said, with or without XML, there are still many hard problems to solve in the schema versioning domain. Let's examine a use case involving a defect tracking system. In version 1, the designer simply includes an untyped element called <productDescription>. Any valid XML content, such as XHTML, could appear there. In version 2, <productDescription> becomes a complexType allowing mixed text, but explicitly specifying subelements for <manufacturer> (required), <model>, and <serialNumber> (both optional), and allowing an open content model. Now, to find a defect report where the manufacturer is Dell, in the version 1 schema the best I can do is a text search in <productDescription> for the word "Dell". This is highly likely to give me what I want, but not always. If an instance looks like:

```
<productDescription>The HP LaserJet5 that  I
was shipped along with my Dell Dimension
650</productDescription>
```

A search looking for the word "Dell" in <productDescription> will get a spurious hit, since the defect in question applies to an HP printer. However, the chances of a spurious hit are much less than a full-text search of the entire defect document had no tagging been used.

To allow a v2 application to get a "manufacturer", I could write a utility to go through v1 instances and tag any of the well-known manufacturer names from a list by scanning productDescription. If one and only one manufacturer matched, I could wrap the manufacturer name with XML tags, and mark the instance as upgraded to a v2 schema. However, some instances would not be upgradable, such as the example above.

In this case, a v1 instance simply doesn't have enough information to answer all of the questions a v2 application might want the answers to. The v2 application would simply have to handle both cases. For example, in a query-by-example screen, the v2 application could gray out v2 query fields based on a date range search, if the minimum date was earlier than the v2 application upgrade date. Various manual techniques would have to be employed if that was unacceptable. For example, the application could keep an upgrade list to send to users to upgrade instances they originally created, or require that the instance be upgraded before update.

In general, schema design errors may result in instances with insufficient information to meet requirements of later versions of an application, because the designer didn't think to ask all of the right questions when the data was input. The key thing that XML allows, though, is managing all of the instances, even those with only some of the answers.

6. IS XML PERFORMANCE SUFFICIENT?

Any data management system architecture is typically driven by performance considerations as much as anything else. So, to provide a technology allowing for more flexible structural management, one must also have competitive performance with existing object-oriented and relational technologies. Some of the key metrics that XML technologies must match include:

- Query performance based on known structure should be comparable to relational.
- Read performance should be comparable to file read
- Write performance should be at minimum within 2-3x of a file write (given the index update cost)
- Partial update (of a single element) should be comparable to relational row/column update
- Access of in-memory data structures based on XPath should be comparable to the cost of a hash table access

Based on the work at Oracle currently under development, all these goals appear to be achievable.

To go through the example of the query case, the XML lifecycle still allows for all of the fundamental techniques that provide relational performance. Once XML data is loaded and schemas have been defined, indexes can be built on the XML data in the same manner as on relational data.

High performance queries generally are driven by indexes, so there is no reason that query performance on indexed XML storage should be significantly slower than relational, even if the XML was originally stored without a schema. The index structures will be the same—only the cost to scan a row would be different (Storing XML in a binary format could also address that cost). The interesting parts of the data structure, from the point of view of the relational query engine, could all be in the indexes, not in the table data.

Indexes have the lifecycle properties we desire—they can be built after the data is loaded, without causing significant disruption to the running application. The data types being indexed can also be determined at index creation time, by building functional indices coercing data into the desired type from untyped string data. It is possible to build generic name/value pair indices on XML data (actually this is typically a path/value index, where Xpath-like node identifiers take on the role of the name in a typical relational name/value table). In addition, specific value indices can be built just as in the relational case using functional indexes, including concatenated key indexes, and still have the indexes get picked up by the query optimizer.

7. CONCLUSION

To be a compelling technology platform for more than 20% of the world's information, databases must go beyond the rigid structural capabilities typical in applications today. The answer is not simply to remove all structure and use LOBs, since a large part of the value of databases is lost. We believe that allowing small amounts of structure to be added cheaply and manageably, without requiring all the overhead of typical relational design processes, provides a more compelling solution. The cheaper the structure is to add, the more likely an iterative schema design approach can be employed. If the first iteration is cheap, more content will be loaded into a DBMS.

XML technologies don't solve the entire problem of managing structural change, but they do provide a significant advance over previous techniques such as relational or object-oriented data management. XML also allows for structural definitions that aren't possible in object-oriented or relational systems. While XML also provides benefits for document management systems, and as a common file format, our experience is that neither of these generates as much impact for our customers as the structural flexibility of the XML stack. This perception of XML utility is likely to be instrumental in driving technology directions in the Oracle database for some time to come.