A Decade of XML Data Management : An Industrial Experience Report from Oracle

Zhen Hua Liu Ravi Murthy

Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, USA
{firstname.lastname}@oracle.com

ABSTRACT

XML and its related technologies have now been in use for almost a decade. There has been considerable amount of effort both from research and industry focusing on XML, XQuery/XPath, XSLT and SQL/XML processing in the database. Many research prototypes and industrial products have been built to satisfy the XML use cases. This paper reviews several use cases where XML databases are leveraged to build real-world XML applications. We discuss the lessons learnt in supporting both data-centric and document-centric XMLDB applications within a single database system and the need for the implementation of different XML storage, index and query optimisation techniques for different XML use cases. We show the value of managing XML in databases, the current challenges and improvements that will hopefully promote future research directions. This paper also provides a timely checkpoint of XML data management from industrial perspective with experience of developing and supporting Oracle XML products.

I. INTRODUCTION

Since the start of XML technology a decade ago, there has been tremendous amount of interest and effort in supporting XML and its associated languages, such as XPath, XQuery, XSLT and SQL/XML, from both research and industry in the database community. When XML became the popular data format for the Internet, there were many research efforts within the database community on approaches to store, query and process XML data. Numerous papers [23,25,27] have researched how to leverage RDBMS to store XML and index over XML without taking advantage of XML structures, such as DTD or schema description of the underlying XML. Paper [24] researched how to store and query XML in RDBMS with the help of XML structure. Paper [30] focuses on building native XMLDB without leveraging RDBMS. Papers [28,29] have addressed how to model and query XML view over relational data so that XML and XQuery languages can integrate variety of data in the mid-tier.

The commercial database industry has developed three major approaches for leveraging and supporting XML in data management system. One is to extend RDBMS with XML capability and the other is to build pure native XML database without direct integration with relational system. The third approach is to position XML and XQuery support as the data integration solution in the mid-tier [31][32].

Similar to the object relational wave in the database community [19], all major relational database vendors have extended their RDBMS with the capability of storing, querying and updating XML data in addition to the traditional relational or object relational data [13][14][15]. The goal of such hybrid XML and RDBMS system is to enable users to manage both relational, object relational and XML data in one platform and to have full interoperability among all of their data. SQL/XML standard [16] has played an important role in "gluing" SQL and XQuery, relational data and XML data, relational schema and XML schema into one single platform. On such platforms, XML views can be created from relational and object relational data whereas relational views can be created from XML data. SQL can be used to query XML data and coversely, XQuery can be used to query relational data.

On the other hand, there are also several "native" XML database vendors [33]. This is similar to the emergence of "native" object databases (and OQL), The goal of such systems is to natively store XML data using a persistent tree data model and use XQuery/XPath as the language for manipulating XML. Since XML has a tree based data model and XQuery/XPath is based on tree model with wild card path matching capability, combined with full text searching capabilities, native XML database are quite attractive for managing document content and facilitate semi-structured data search. Instead of tables and views, document collections and documents are the primary objects in native XMLDB. XQuery is the primary languages to query documents, to extract pieces of documents and to construct new documents from various pieces of documents.

Although there are many convincing use cases for all of these products, there are still some underlying philosophical questions and concerns on the fundamental value of XML within the database community. For

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.18

example, paper [2] questioned that thirty years of relational database research and industrial practices have proven that relational data model is much better than hierarchical data model or network data model, isn't XML with its hierarchical tree based model regressing us back to the hierarchical data model? Is relational model not sufficient i.e. do we need XML? Does XML have to be persisted as a tree to match its logical data model? What are the database application use cases for which XML is best suited? In terms of query languages, report [45] argued that "we have moved from SQL to XQuery, one declarative language to a second declarative language with roughly the same level of expressiveness". So what is the significant value of XQuery compared to SQL?

In this paper, we will attempt to answer these questions based on the analysis of actual customer XML application use cases and their adoption of XML technologies in RDBMS. It is also based on our own development experience of Oracle XML database systems. We find that since the concept of XML is too general, the attempt of finding 'one size fits all' universal XML storage and index solution is not feasible. Instead, it is important to classify XML use cases into different categories and leverage different XML indexing, storage and processing techniques in each category. Failure to do so may cause sub-optimal solutions and raise doubts of the value of XML in DB world. Only when we model XML as an abstract datatype requiring different processing techniques for different XML use cases , then we can fully realize the fundamental value of XML in database community. We discuss these points in section 3.4, 4.5 and 6.

In summary, the main contributions of this paper are

A. We show the importance of separating XMLDB applications into **data centric XMLDB applications** and **document centric XMLDB applications** because the two classes require different XML storage, indexing approaches and query optimisation and processing techniques.
B. We show the idea of modelling XML as an **abstract data type** in database and using an advisor wizard to find the best XML storage, index and processing models for different XML applications.
C. We show that promoting XML data model does not attempt to recycle prior hierarchical database model. Instead, modelling XML as an abstract data type enables us to use XML as different models for different shapes of data. We show that XML acts as a **presentational model** for relational data and enhances the hierarchical expressive power of the relational model. We also show that XML is an adequate **logical data model** for semi-structured data and **physical data model** for managing content repository data.
D. We show how XQuery combines the capabilities of declarative data query, imperative data programming and flexible data transformation paradigms into a single language. Therefore, an optimal implementation of XQuery language requires several different processing techniques and poses many challenging research opportunities.

The rest of this paper is organized as follows. Section 2 discusses various XML use cases to provide concrete examples for A). Section 3 discusses the modelling of XML for different type of data to support B) and C). Section 4 discusses the improvements needed and challenges ahead for XML processing to support B) and D). Section 5 shows the empirical data for A). Section 6 discusses the fundamental lessons learnt to support A). Section 7 concludes the paper.

II. XML Application Use Cases

A. Usecase: XML Data and Report Generation

XML is widely used as a format for data transfer and thus needs to be generated from relational data for business data exchange and report generation. This is one of the most common use cases for XML enabled RDBMS. The generated XML data is hierarchical in nature corresponding to the master-detail-detail hierarchy in the typical OLTP relational model. There are two syntactic ways of generating XML data in RDBMS. One is SQL/XML centric and the other is XQuery centric. The SQL/XML centric approach is to use SQL/XML [16] generation functions, such as XMLElement(), XMLForest(), XMLConcat() and XMLAgg(), that can declaratively generate arbitrarily shaped hierarchical XML data from relational data. The significance of the XML generation functions is that they support generating hierarchical data from relational data inside RDBMS using XMLType as an abstraction. The XQuery centric approach is to use pure XQuery with deeply nested XQuery constructor expressions. The base relational or XML data is accessed as a built-in XQuery function that is able to generate XML from flat relational table. One such example is *ora:view()* function in Oracle XMLDB [13]. Query 1 shows the XQuery example. The XQuery extension function *ora:view()* generates XML from a relational table/view (in this case "*COMMERCE*") or an XMLType table/view (in this case "*ATTR_XMLT*") transparently.

However, both the SQL/XML centric way and the XQuery centric way are algebraically equivalent. Both can be optimized into the same underlying XML extended relational algebra and executed on the underlying SQL/XML engine [17].

```
<counties>
  {for $c in ora:view("COMMERCE")
   let $coc_county := $c/ROW/COC_COUNTY/text(),
       $coc_name := $c/ROW/COC_NAME,
       $coc_phone := $c/ROW/COC_PHONE/text()
```

```
    order by $coc_county
    return
      <county>
        <name>{$coc_county}</name>
        <chamber
phone="{$coc_phone}">{$coc_name/text()}</chamber>
        <attractions>
          {for $a in ora:view("ATTR_XMLT")
           where $coc_county = $a/attraction/county/text()
           return $a
          }
        </attractions>
      </county>}
    </counties>
```

Query 1- XML Report Generation with pure XQuery

B. Usecase: Structured Query Over XML

This use case is the reverse of the previous relational to XML generation case that we discussed above. In this use case, the XML data (potentially generated from a relational store) is received and then (after some preliminary analysis and cleansing) decomposed into a set of relational tables. Even if the XML were generated from a relational store, there is value in using XML as the intermediate transport format. This allows decoupling of the physical schemas of the source and target databases. Finally after decomposing the XML into relational tables, users are able to access these relational tables directly. This use case is typically known as 'XML shredding'. However, the problem with shredding is that users still need to manipulate relational data directly, not XML. One of the key features of an XML enabled RDBMS is to hide the XML shredding (and SQL query) via the XML datatype. Such system enables user to register the XML schema for XML data directly into the system so that the system can generate a set of object types and object relational tables underneath the cover for storing the XML data. Users directly manipulate the XMLType table without worrying about its shredding details. They directly use XPath, XQuery and SQL/XML to query XMLType table without having needing to explicitly refer to the underlying relational storage tables. The RDBMS optimizer is extended to support XML Queries and can rewrite XPath/XQuery to SQL/XML constructs over the storage tables transparently. Query 2 shows an example of creating a XML schema based *purchaseOrder* table. As shown, the SQL/XML query with XMLQuery() and XMLExists() operators and embedded XQuery/XPath is rewritten by XML query optimizer into an equivalent query without XQuery/XPath.

Paper [11] discusses the XML schema based object relational storage and [12] presents XML rewrite in such use cases. Although this type of use case does not show the value of XML in terms of handling semi-structured data, it has the value of showing XML as a good presentation data model for accessing relational data in a hierarchical way. We will discuss further details of this in section 3.1.

Although XML schema registration results in standard shredding of XML data, there are use cases where the default automatic way of shredding data is not adequate. In these use cases, users leverage XMLTable [22] construct to create relational views over XML. Afterwards, all the relational access can be built over this set of base XMLTable views. The XQuery used in the XMLTable construct is complex such that that automatic shredding is not feasible.

```
-- create a table storing all XML document instances for registered
schema 'http://mypo.xsd'
create table purchaseOrder of xmltype schema 'http://mypo.xsd'
```

```
select
xmlcast(xmlquery('$po/PurchaseOrder/@podate' passing value(po) as
"po") AS date),
xmlquery('<ship_order_cities>
            <shipCity>{if ($po/shipAddr/city) then $po/shipAddr/city
else ()}</shipCity>
             <billingCity>{if ($po/billingAddr/city) then
$po/billingAddr/city else ()}</billingCity>
          </ship_order_cities>' passing value(po) as "po")
from purchaseOrder po
where xmlexists('$po/PurchaseOrder/lineItems[price > 34 and quantity
< 5]/Parts[partPrice < 23 and partQty > 4]' passing value(po) as
"po")
Order by xmlcast(xmlquery('$po/PurchaseOrder/@podate' passing
value(po) as "po") AS date)
```

```
select po.podate, xmlelement("ship_order_cities",
                    Xmlforest(po.shipCity as "shipCity",
                              Po.billingCity as "billingCity"))
from purchaseOrder po
where exists (select null
              from lineItem li
              where li.price > 34 and li.quantity < 5
              and li.nid = po.snid and
              exists (select null from part prt
                      where prt.partPrice < 23 and prt.partQty >4
                      and prt.nid = li.snid)
order by po.podate
```

Query 2 – Schema based XML Query and Rewrite

Beyond structured XML, users want to just extract some structured data from the XML for query without the need to fully shred the XML data. Sometimes, it is not even feasible to fully shred the XML data because there is no XML schema or the XML schema has too many optional components and fields and 'any' types so that a straightforward shredding solution may results in creation of many relational tables with sparsely populated content. However, the set of queries to access the XML are pre-determined by the particular XML applications. All the query accesses are well controlled and there are no ad-hoc queries during application run time. Therefore, the index to satisfy these queries can be determined ahead of time. The TPoX benchmark paper [37] illustrates examples of such usage pattern where structured components of XML need to be indexed and queried.

C. Usecase: XML for Content Management

XML documents are heavily used to represent human edited information in content stores and knowledge databases. Owing to its SGML legacy, XML is primarily used in these domains as a simpler and more standard format. XML also goes much beyond HTML in its ability to mark up the content and associate interesting semantic tags to portions of content. The XML markup is then exploited to improve relevance of search, and also for reuse and re-purposing the content within many documents and applications. XML is invaluable for applications such as books-on-demand where relevant content is pulled together dynamically based on user interests, and a customized book is generated on the fly.

We have studied several customer implementations of XML based solutions for content management within Government legislatures, legal firms, etc and summarize them as follows. XML is the native format for representing the human edited information. An XML-aware client or editor is used to create or update the content. Instead of raw tags being exposed to the end-users, subtle semantic markers are used. The XML content that is created via user interactions is stored as XML into the database. The XML schema is typically quite complex with a large number of tags, high degree of variability, optional tags, and recursive structures. The amount of XML content is also typically quite large (hundreds of megabytes is common). XML processing instructions and comments, which are typically alien in data centric XML, are common in this use case. Queries are used to both (a) identify documents matching keywords and structured queries as well as (b) extract matching fragments. Due to the large size of documents and possibly expensive post-filtering, indexes are augmented to store fragment level pointers – facilitating direct fragment extraction.

In content management applications, XQuery and XSLT are the primary languages to provide direct manipulation of document content. Although XQuery is used similar to SQL in the usecases of querying structured XML, XQuery shows its full power when it is used in content management applications because it can declaratively transform, construct and flexibly traverse the document content, far beyond the capabilities of SQL. The (simplified) query example below illustrates some of the typical operations in a content management application. Note the use of descendant axis (//) (and in other queries - wildcard steps). Also the query combines full text search along with other structured conditions. Finally the returned value is constructed from multiple extracted fragments with possible transformation.

In content management applications, document references need to be managed as first class constructs. In particular, intra-document and inter-document references have to be stored, indexed, queried and updated. For example, there are references from the table of contents to all the chapters of a book. Further, a piece of the content may be reused in several places e.g. the Safe Harbour statement is included in all financial statements. This is typically tracked as (a different kind of) reference. Some sample requirements in this scenario are :

- Find all references to a document (link query)
- Which content referenced by this document has been modified in the last week? (link query combined with other query conditions)
- List the most heavily referenced content (link analysis)

```
for $b in fn:collection(''/books')
where $b/author=$a or ft:contains($b/abstract, 'XML Query')
return
<citationinfo>
   <bookinfo>{$b/title}</bookinfo>
    <citations>{$b//citation}</citations>
</citationinfo>
```

Query 3 - Sample Query in Content Management

The current standards such as XML schema, XLink and XInclude do not completely address these requirements. Consequently applications have to implement a large amount of custom code to enforce the link constraints and semantics. [39, 40] has proposed several declarative mechanisms for comprehensive link management and better support for graph data models within XML. The XML schema framework is extended to express constraints on links, namely the type of link target, referential integrity and acyclic constraints. The query language is extended with functions to traverse links in both the forward (*fn:deref*) and reverse (*fn:getInLinks*) directions. A sample query with proposed extensions is shown below. The citation element is a link to another book. All citation links from the given book are traversed to extract their titles. Additionally all links to the given book are also identified and their titles extracted.

```
for $b in fn:collection(''/books')
where $b/author=$a or ft:contains($b/abstract, 'XML Query')
return
<citationinfo>
  <bookinfo>{$b/title}</bookinfo>
  <citedbyme>{fn:deref($b//citation)/title}</citedbyme>
  <citesme>{fn:getInLinks($b, citation)/fn:root(.)/title}</citesme>
</citationinfo>
```

Query 4 - Query using Link Functions

D. Usecase: Heterogenous and Ad-hoc XML Store

The biggest benefit of XML is the possibility of schema-later or schema-never applications, compared to other data models that invariably insist on schema-first approach. In this usecase, XML documents originating

from a number of different disparate data sources are aggregated into a single central repository. The owner of the central repository has little or no authority to impose a single schema on all the data sources. Instead there is a loose understanding (in the best case) of a set of interesting tags that might be used within the documents. Each of the different data sources (applications) has a different realm of control. It may have a local schema that might use some of the interesting tags mixed in with many local tags. Note that the context of the interesting tags (i.e. hierarchical nesting) may be quite different in the different sources. The goal of the application is to store and query the documents in the centralized repository in some meaningful fashion.

The common queries in this scenario involve XPath wildcards and/or descendant axes. This is understandable given that there is no universally accepted schema. Instead users search for interesting tags (in any context) using the descendant axes. The main indexing strategy here is the universal XML Index [13] that indexes all tags, paths and values in the XML documents. This provides an efficient mechanism to lookup any tag and/or value without apriori schemas. We also see full text search within a tag context being very useful in this scenario.

```
for $r in fn:collection('/accidentreports')
where $r//vehicle = 'Ford' and ft:contains($r//reason, 'overturn')
return $r
```

Query 5 - Query against Ad-hoc XML

The basic wildcard query returns all results in a flat list. A better result strategy is to categorize the results based on their context i.e. return the interesting path contexts and the number of results in those contexts. The categorized results provide a succinct but very useful understanding of the result set and allows the user to drill down (or refine) the specific context of interest – thereby iterating from a loose search with no context to successively better contexts. This process is similar to faceted or parametric search but the set of facets or parameters is not predefined. Instead it is calculated dynamically from the query results.

```
(: Initial query:)
for $r in fn:collection('/reports')
where $r//vehicle = 'Ford'
return $r//vehicle
```

```
(: Faceted search results :)
120 results found
/reports/accident/vehicle (95)
/reports/accident/reason/impact/vehicle (18)
/reports/witness/vehicle (7)
```

```
(: Refined query :)
for $r in fn:collection('/reports')
where $r/accident//vehicle = 'Ford'
return $r/accident
```

Query 6 – Query Refinement

Although the initial set of documents in the collection have no apriori schema, it is still very useful to derive partial schemas from the instances themselves. This is similar to structural summaries or Data Guide work presented in [10]. One difference is that the inferred schemas also have typing information such as datatypes for scalar elements, facets such as min and max values, etc in addition the structural summary. Another difference is that it may be sufficient to derive a good enough schema instead of a comprehensive summary. This will prevent exception cases from polluting the common case. For example, if the field <Age> appears as a integer in most of the documents but as a string in a few instances, the inferred schema may still declare it as a integer field and deem the exception cases as validation errors. Much of this behavior can be configured via threshold parameters. The inferred schema can then be used for formulating better queries.

III. XML Model for different types of Data

Logical and physical data design independence is a well-known principle. Database system is divided into logical data model and physical data model. The **logical data model** describes the conceptual layout of the data on which a declarative language is defined so that the language has well defined semantics under the model. **Physical data model** defines how the data is represented in memory or disk, how the data is indexed and partitioned to speed up queries. Changes to the physical model should not impact the language functionality but may impact its performance. However, applications often demand various **presentation data model**s that are more natural and native to the application objects. There can be many presentation data models for one logical data model and there can be multiple physical data models to support one logical data model. In this section, we will show that XML plays an important role for modelling different types of data.

A. XML as hierarchical presentation data model for relational data

"Making Database Systems Usable" paper [1] argued that "To make database system usable, it is required to support various presentation data models to match user's logic view of the system. This is beyond just creating an attractive user interface to the database system, but rather creating and supporting conceptually different presentation data models than the logical and physical data model in the underlying database system.

From this perspective, XML has its value because its hierarchical tree based data model serves as an excellent presentation data model on top of underlying flat relational data model in RDBMS. RDBMS with XML extensions supports the intuitive hierarchical data model that end users are familiar with. With hierarchical XML data model, users can use path-like languages, such as XPath to traverse hierarchy without worrying about join operations. In fact, simple path expressions from XPath [20] are essentially projection and selection operations

without any explicit join operations. This is aligned with the argument from [1] that a good intuitive query language should only have selection and projection without explicit joins.

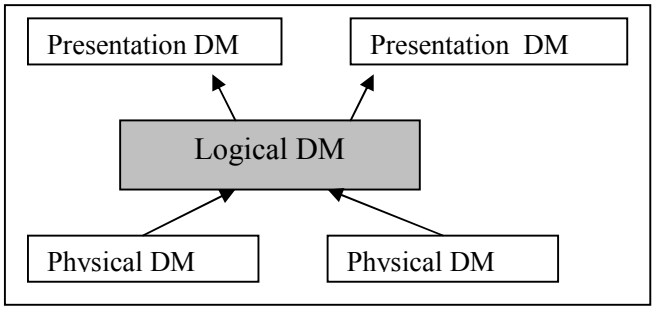

Figure 1- Presentation, Logical, Physical Data Model Relationship

In current RDBMS, users have to understand the explicit primary/foreign key join relationship before writing SQL queries. Although creation of high-level relational views is able to hide the details of the join relationship, the result of relational view is still a flat relation. But users want to manipulate high-level hierarchical objects inside RDBMS directly! Consequently, the impedance mismatch between the hierarchical view of data from user's perspective and the flat physical relational view from database system perspective has to be resolved outside RDBMS resulting in many object relational mapping tools today.

Supporting XML as presentation data model in RDBMS with a set of functions and operators that directly manipulate XML has precisely solved the impedance mismatch problem. SQL/XML [16] has defined a set of XML generation functions to construct hierarchical XML data from flat relational data. For example, XMLElement(), XMLForest() makes singular object whereas XMLAgg() makes collection objects. XMLConcat() glues different objects together. XPath embedded functions, such as XMLQuery() and XMLExists() make path driven query over XML very easy for users. The XMLTable construct, conceptually an inverse operator for XMLAgg(), is able to decompose XML and cast it back to relational data. These inverse algebraic relationships among these functions and operators make XML query optimization and rewrite over relational data feasible [11][12]. The benefits of having algebraic operators for presentation data model is further backed up by "The painless future" section of paper [1] which states that "We must develop an algebra of operations in the presentation data model such that the basic needs of most users are met by a very small number of operators, thus reducing the barrier to adoption".

However, having argued that XML is a good presentation data model, then why not use XML as the actual physical data model for hierarchical data? Why store hierarchical data using relational model? Our answer is that for highly structured XML data, relational data model provides a great physical data model and there is no *reason* to switch it to physically persisted XML trees. Doing so would inevitably regress both the theoretical advantages and practical success of relational data model that has successfully replaced the old hierarchical and network data models [2]. We observe the following key points when we analyze physical tree storage models for structured XML:

First, the flat relational data model gives tremendous freedom for the optimizer to decide how to search for the data from any relations in the hierarchy instead of forcing the top-down search direction directly supported by the physical hierarchy. Any secondary indexing strategies on the semi-structured data to facilitate the top-down, bottom-up, hybrid query plans [46] is essentially the same as the relational optimizer determining the optimal join order.

Secondly, when XML data is highly structured and can be described by a pre-defined relational schema, the overhead of run time structural search can be completely eliminated by pre-determined relational schema selection during query compile time - so only data value search is needed during run time. That is, although answering XPath query conceptually involves both structure and data search, the structure search in XPath can be pre-computed during query compile time for structured XML data.

Thirdly, when XML data has well defined structured hierarchy, then it is much more efficient to traverse the hierarchical relationship among nodes using master-detail primary-foreign key join from the relational model than using ordered key of nodes. *In fact, even if XML were persisted as a physical hierarchical tree, indexing its highly structured components using relational model provides much better query performance when querying the structured components of XML, especially for searching the scalar property data of XML.* Relational like XMLTableIndex [18] has shown that indexing highly structured XML data using relational model is much more efficient to answer structured query than the universal structure/path/value index approach.

B. XML as logical data model for semi-structured data

While XML serves as a good hierarchical abstraction over relational data, XML shows its real strength in managing semi-structured data where there is insufficient schema or structure to describe the data so that it can be fitted into tabular form. This is what "Making Database Systems Usable" paper [1] described as "Birthing Pain" – hence a "schema-later" style of heterogeneous database

design is needed. Supporting both structure and value search during run time allows users to be able to still query their data while the schema of the data is not fully known. Paper [3] has outlined some of the key aspects of querying semi-structured data.

Although relational database system can be leveraged to support both structure and data search when all attributes of an entity are not known in advance or there are too many nullable attributes resulting in sparsely populated table [7][8], we argue that various techniques to support these use cases, such as vertical schema approach [4][7], Entity Attribute-Value model [8], interpreted storage format for sparse datasets [5][6], essentially advocate the same underlying idea that we need a data model to support both structure and data query at run time because data and structure cannot be separated out cleanly. XML is the appropriate logical data model for these use cases. The interpreted storage model [5] is very similar to binary XML format presented in [13]. *In both approaches, data and structure are interpreted together at run time.*

Beyond the flexibility of representing sparse data as XML, the wild card query construct and child-or-descendant query construct expressed in XPath steps essentially embrace the concept of searching structures without knowing all the details of the structure in advance. This implies the underlying data model has to be able to co-locate both the structure and data so that they can be searched together at run time instead of storing structure separately into meta-data catalog in the relational model. Although such style of query may not be as efficient and precise as highly structured XML data, it is still the best approach for querying XML without knowing structures in advance. Furthermore, the use of keyword search within XPath/XQuery [21] is a unique strength of XML. Traditional keyword search can only search for keywords within a document. With the hierarchical model of XML, we can also find the *context* of keyword occurrences and also find the hierarchical relationships among these contexts. In [9], although not explicitly using XML data model, the idea of combining structure and keyword search for heterogeneous data is promoted.

Finally, unlike schema in relational data that is used for defining the table structures for holding the data, the XML schema used in many XML applications is designed for data validation. Therefore, although relational schema is compact and small compared with its data size, XML schema is quite large, evolving and appears to be 'over designed' from the classical entity-relational design in the relational model. Therefore, the meaning of schema in XML world is different enough that we have to separate the notion of the *query schema* from the *validation schema*. The XML schema is used for data validation whereas the query schema is used for indexing the data. In many use cases, the XML schema may be absent, however, the query schema, which is used for indexing the data, can be derived from the data guide. In this way, the indexing size for XML can be compact instead of bloated.

C. XML as physical data model for document repository

Besides being a hierarchical presentation data model for relational data, XML is also an ideal physical data model for document and content repository. In the content world, the basic high level concepts are files, documents, folders and repositories instead of rows, columns, tables in relational model. Folders and documents have a named path and system assigned identifiers. Folders and documents are located through directory path mechanism instead of row ids. The direct manipulation of content data is often associated with protocols, such as FTP, HTTP, NFS, WebDAV associated with drag and drop interface instead of SQL languages. Document versioning and being able to manipulate multiple versions of the same document are more common. *It is more natural to manipulate coarse-grained document fragments instead of fine-grained relational cells*. Having direct support for these high level document concepts in RDBMS via XML makes RDBMS more usable for content users [34].

XQuery and XSLT are the most natural declarative languages to manipulate document content replacing the imperative DOM and SAX APIs and substantially more powerful than SQL for this usecase.

D. XML as an abstract type

ANSI SQL/XML defines XMLType as a built-in datatype in the RDBMS [16]. However, unlike other RDBMS datatypes, XMLType is an abstract datatype that has several standard query languages, such as, XQuery/XPath/XSLT and SQL/XML, to process it. The physical storage model can be changed without affecting application queries which process XML data via XQuery, XPath, XSLT and SQL/XML languages. The change in storage and/or indexing model for XML does not change the functionality but may impact query performance. As we have discussed in all the previous use cases, there is no "one-size-fits-all" storage and index model for XML that can deliver best query performance for all XML use cases. *This is similar to the situation in pure relational database design where the optimal physical data layout is dependent on all the queries used in the application use cases.* For example, traditionally the choice of data normalization and de-normalization has to be considered when designing OLTP and decision support application use cases. Recently, paper [35] argues that column based storage can improve the performance of data warehouse queries significantly compared to the classical relational row storage model.

Using XML as an abstract type allows user to alter the storage model of XML without changing any applications. For example, users may start with object relational storage of XML for highly structured XML data and later on evolve it into hybrid or binary XML storage as the XML data evolves to have more unstructured content. On the other hand, users may start with binary XML storage model and create (relational-like) XMLTable indexes to index structured components of the XML data as the XML structure is discovered. However, regardless of the changes in XML storage and indexing strategies, application queries do not have to be changed at all. That is, XML queries over XMLType abstraction are supposed to be **'write once and run everywhere'**[54]. Figuring out the right storage and index models for XML should be the job of a "storage wizard". Paper [43] describes an approach of building schema advisor for hybrid relational-XML DBMS. Although the advisor technique is appropriate, however, exactly what business data should be stored in a relational tables and what should be stored as XML tree is the choice of physical model. It can be hidden if we model the business data as the logical XMLType. With this approach, when the business data characteristics changes, for example, a performance critical element is later on identified as an element with many variations so that its storage has to move from the relational table storage to XML tree storage, the query over such elements does not change at all. Figure 2 shows XMLType serving as an interface type abstraction between XML languages with its multiple underlying storage and indexing mechanisms.

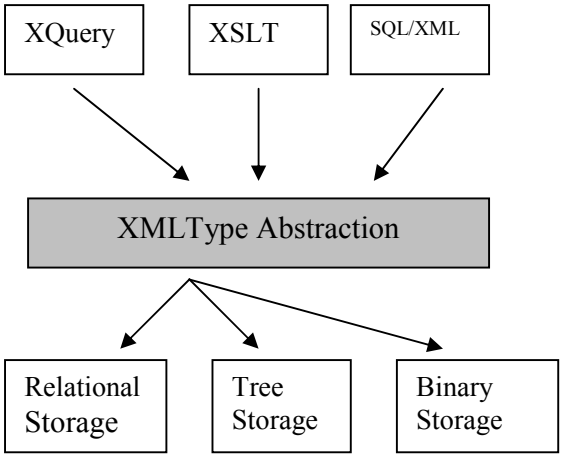

Figure 2 - XMLType Abstraction

Having discussed the value of XML in DBMS, we now discuss the improvements that are needed for XML processing.

IV. Improvements and Challenges for XML Processing

A. The need for light-weight typing system for XQuery

XML schema provides a very rich and complex set of functionality. It is a good mechanism to express validation rules for wide variety of XML data. However, it is too cumbersome to be used as a typing mechanism for XML programming languages, such as XQuery and XQueryP [44]. When using XML programming languages for general purpose computation, users prefer a light-weight mechanism of defining datatypes primarily for scalar values of the leaf nodes of an XML tree instead of having to define a full schema for the entire XML tree. Furthermore, in many common usecases, users prefer to able to define anonymous type structures to group relevant fields without having to define an XML schema and having it imported into the system before it can be used. Therefore, we believe that it would be more useful for languages such as XQueryP[44] to introduce a light-weight and convenient way of defining types for XML instead of relying on XML schema. XML schema is a good validation language for XML. However, a general purpose XML programming language needs a simple typing mechanism. We propose that XQuery and XQueryP take the XML Schema Part 2 definitions for all the basic built-in types, such as *xs:integer, xs:string, xs:date,* etc as the base types and then develop a light-weight mechanism to define structure and class types on these built-in types.

B. Document links in XQuery

As we had discussed in section 2.3, XQuery currently does not provide adequate data model and query functions and operators to support *querying link based XML documents*. As real world XML documents are often compound documents with Xlinks to weave different components of documents together, XQuery needs basic function support to traverse XML documents linked through XLink mechanism. Similar to the object reference and de-reference concepts in object relational DBMS [19], XQuery needs to extend its data model and functions with document reference and de-reference concepts so that it can manage a graph model of XML documents instead of the current tree model [40]. Defining, managing and querying links between XML documents shall be the first class citizen construct in XQuery. There are basically two types of relationships among entities in relational model. The first relationship is master-detail, one-to-many hierarchical relationship. The second relationship is many-to-many linking relationship among entities. With XML model, the hierarchical relationship is built-in, however, the many-to-many linking relationship represented by XLink needs to be natively expressed in XQuery.

A common usecase involves evolving the XML representation of an object from being nested within a parent document to becoming a standalone full-fledged document itself. E.g. in the first version of the application, the customer information is nested within Order documents. In a later version of the same application, the customer details are normalized into a separate Customer document. The new Order documents now contain a link to Customer document. To minimize impact to existing applications, it is desirable to shield navigation operations from this evolutionary change. We are evaluating the use of a new XPath axis (child-or-deref) that traverses to the child element (if nested) or traverses the link (if linked). Uniform use of this axis reduces the impact to applications in face of many evolutions.

C. Optimization of general purpose XML languages

There has been much discussion about the "impedance mismatch" problem between the SQL, Java/.NET and XML models. The benefits of a single data model and language across all tiers are generally accepted. However, previous efforts to accomplish them from different directions have not succeeded.

XQuery is more general than SQL and is able to support both data query and data transformation. With the scripting extension of XQuery into XQueryP [44], the XQuery/XQueryP can potentially be the universal programming language to integrate data, logic and presentation into a single platform, XQuery/XQueryP can then be the integrated language for XML data query, transformation and programming. Such integration enables user to write a single **holistic** program without thinking about the (artificial in our opinion) boundary *between declarative query and imperative procedure portions.* Today, separating imperative and declarative logic is mandatory when embedding SQL into host programming languages. Even for the case of SQL PSM (such as Oracle PL/SQL), users have to somewhat demarcate between SQL and procedural logic. Therefore, contrary to the observation from paper [45] that SQL and XQuery have the same level of expressiveness, XQuery/XQueryP can fundamentally blur the boundary between data query and procedural program. However, this does raise a major challenge.

Traditional query languages, such as SQL, are based on iterator lazy evaluation model with index probe and materialized view matching to scale to large size data sets. Traditional programming languages, such as C/Java, are based on imperative eager evaluation model. Transformation languages, such as XSLT, are based on template matching model. As XQuery/XQueryP integrates all of these together, the optimizer needs to determine when eager evaluation should be used and when lazy evaluation strategy should be used and achieve the balance between the two. **Optimizing general purpose XML languages so that the conversion between the declarative and imperative logic is automatically handled in a cost based manner is not a simple exercise.** For example, users may program the logic of child or descendant node search using recursive functions and the language optimiser should consider rewriting such recursive functions into child or descendant XPath expression so that the underlying path index might be leveraged. *These challenges require cooperation between the database and programming language communities* [53].

D. Storage and index advisor for XML

As discussed in section 3.4, XML is a good abstract type to model data, however, the physical storage and index for XML depends on the actual XML usecases and their set of workload queries [26]. Figure 4 shows a quadrant diagram for various XML physical storage and indexing models for data centric XML having embedded structured component and unstructured content, document centric XML having embedded structured component and unstructured content. As evidenced in RDBMS, automatically figuring out the physical designs, such as indexes, materialized views, horizontal partitioning, de-normalization, column store [35] strategies, for a given work load is an interesting challenge. Recent industrial efforts for automatic storage and index advisor in relational applications are discussed in [48, 49, 50]. We expect similar efforts and research opportunities for processing XML workloads. Paper [47] shows that for schema based relational mapping, there is a cost based approach to find the best mapping solution for an XML query workload. With schema and non-schema based XML and universal structure, value, path and relational XMLTable indexing strategies for XML, there are many cost based choices to optimize the physical design of XML for given XML query loads.

Primary Content Vs Embedded Content	*Embedded Structured Component*	*Embedded Unstructured Content*
Data Centric XML	Structured storage	Structured storage with path/value index on unstructured content
Document Centric XML	Tree/binary storage with XMLTable index on structured component	Tree/binary Storage with path/value/text Index

Figure 3 – Quadrant Diagram for XML storage/index model

V. EMPIRICAL DATA FOR XML DB APPLICATIONS

This section presents the empirical data that we have gathered from various XML DB application use cases. The data is aggregated from about 30 different real-world customer applications. Specifically we present the empirical query and schema complexity for the different scenarios.

SQL/XML Generation Function

SQL/XML generation functions are commonly used to generate XML from relational data. This is the usecase that we discussed in section 2.1. Table3 shows the average number of SQL/XML generation functions used and the average depth of hierarchical tree generated in one such query.

Average Number of SQL/XML generation functions (XMLElement, XMLForest, XMLConcat) used in one query	*Average depth of hierarchical tree generated in one query using XMLAgg() with correlated scalar subquery*
28	5

Table 2 – SQL/XML generation function data

Relational View using SQL/XML XMLTable Construct

It is common to query XML relationally by defining multiple XMLTable views. This is for usecase where XMLTable relational view is used to query XML relationally discussed in section 2.2. Table 3 shows the average number of relational columns projected in XMLTable construct and the chaining depth of XMLTable construct. The XMLTable chain corresponds to traversal of the XML hierarchy. The average XMLTable chain depth is 4 shown in table 4 and the hierarchical generation depth is 5 shown in table 3. These two data correlations indicates the average hierarchical depth of XML is 4 to 5.

Average Number of relational columns projected out of XML	*Average number of XMLTable chain depth to iterate over XML hierarchy*
20	4

Table 3 – XMLTable relational view qry data

XQuery

Paper [51] classifies XQuery into six layers: Simple XQBE, Core XQBE, Full XQBE, View XQBE, Arbitrarily nested XQuery, XQuery with user defined functions and types. Paper [51] concludes that the first four XQuery layers covering 80% to 90% user's needs. Table 5 shows the percentage of different XQuery layers used in data centric XML usecases from our observation.

XQuery Layer	*Percentage*
Simple XQBE	37%
Core XQBE	12%
Full XQBE	11%
View XQBE	19%
Arbitrarily nested XQuery	12%
XQuery with user defined functions & types	9%

Table 4 - Usage of XQuery Layer Data

Our data is consistent with paper [51] conclusion. We also have the first four XQuery layers covering 80% of XQuery usecases with the first layer *simple XQBE* be the highest percentage among all layers and the sixth layer of *supporting user defined function and types* be the lowest percentage. The *View XQBE* is also quite popular. This makes sense because *let* clause is very handy to define variables that can be referenced multiple times in the subsequent query. However, this observation is only true for *XQuery over data centric XML*. For *document centric XQuery applications*, usage of XQuery user defined functions and sequence data type operations are very common. Document centric XQuery are usually associated with XQuery modules and user defined functions. For document centric XML applications, average number of lines for XQuery user defined functions is *21 lines* with average *124 user defined functions* defined within an average of *22 modules*. Therefore, it only makes sense to divide XQuery into layers for data centric XML applications. For document centric XML applications, XQuery has been leveraged to its full strength and capability.

Paper [51] asserts that XPath as the basic query language is not sufficient due to the lack of support for tag construction. However, we find that with SQL/XML generation functions, it is common for users to use XPath only in SQL/XML XMLExists() to qualify XML documents or document fragments, XMLQuery() to extract document fragments and then use SQL/XML generation functions to construct XML results instead of using XQuery constructor functions. However, again such XQuery usage pattern is more applicable to data centric XML applications. For document centric XML applications, using pure XQuery without SQL/XML dominates the usecases.

VI. LESSONS LEARNT

Our experience in developing XML DB systems and studying customer XML applications as discussed in section 2 clearly show that there are two classes of XML database applications: data centric and document centric XMLDB applications. Although both kinds of XML share the same logical tree data model, their physical data model ought to be different for efficient query performance and compact data storage. There is no 'one-size-fit-all' solution for XML database based applications. The application developer needs to be aware of the

differences so that they can use the appropriate XML storage and index techniques.

In data centric XMLDB applications, the structure of the XML is relatively static so that it can be statically separated from the data into a relational schema. E-R model [52] is powerful enough to express the hierarchical tree-based data model. Therefore, the storage of the structure and data does not have to be physically clustered. Separating structure into relational schema that is shared by all the data leads to best data storage compactness. XPath/XQuery over data centric XML can be optimised as SQL query over the data in the relational schema. There is no need to index the structure, only data needs to be indexed. Query over data centric XML only needs to search data during run time because the structure is treated as metadata and "searched" during query compilation time. This leads to the best query performance. For such data centric XMLDB applications, defining a hierarchical XML view over relational DB model makes sense and all the mature relational DB technologies can be fully leveraged.

In document centric XMLDB applications, the structure of XML is dynamic so that it is not possible to separate structure cleanly from the data. Therefore, structure and data for document centric XML has to be physically clustered. Both structure and data has to be indexed. XPath/XQuery over document centric XML has to search both structure and data during run time. This includes matching ad-hoc XPath element names and sequence types during run time. For such applications, a native tree or binary physical storage of XML documents with path-value text index makes sense. The path index allows users to do wildcard search, descendant search efficiently. Value index allows user to discover data values without knowing its containment structures. Full text index over XML allows user to do classical full text search without knowing its structure. Furthermore, sparse structure within document centric XML can be indexed by XMLTable style of XMLIndex [18].

VII. CONCLUSIONS

In this paper, we have demonstrated the value of XML in database system by reviewing various XML use cases in XML data management systems. As paper [2] reviews the data model evolution history in the database community and urges people not to repeat history by recycling data models, it is evident that people tend to equate the presentation data model with the underlying physical data model despite the fact that logical and physical data model independence is well known principle in RDBMS. We believe that the temptation of making the presentation data model the same as the physical data model is the source of the problem. We argue that with the introduction of the concept of presentation data model and the clear separation and independence among presentation, logical and physical data models, XML data model is not regressing the relational data model but rather complementing relational data model with its hierarchical expressive power for structured data. Relational data model remains a simple but powerful building block for many other high-level presentational data models. Hierarchical and network data models can be built on top of the relational model. However, in some cases, *materializing the presentation data model as aggregated physical data model has its merits when the retrieval and manipulation of the aggregated data model as a unit is required*. This is similar to the field of Organic Chemistry that although all chemical compounds can be decomposed into atoms, working with high level chemical compounds, such as sugar, protein, lipid, as an abstraction are very useful in life science. *Our lesson from XML is not to repeat history by positioning XML data model as a replacement of relational data model but rather as a presentation data model for structured relational data.*

Furthermore, we show that XQuery has more expressive power than SQL. For structured XML that can be modelled as hierarchical view over relational data, XQuery with native XPath construct gives more hierarchical expressive power than SQL. For semi-structured data, content centric document data, making XML data model as the logical data model has its merits. XQuery/XQueryP languages are superior to SQL in coping with semi-structured data: it allows both structure and data search simultaneously while the document structure is unknown in advance. The concept of being able to do search without defining schema is a major milestone compared with the rigid schema-first relational world. Being able to integrate declarative data query, imperative programming and transformation logic using a single language is a major improvement over current solutions of separating query and procedural logic artificially. We are living in an interesting time for data management. The value of XML in data management system is not to repeat the data model history but rather address problems that challenge the limits of the current state of art relational solution. Our product experience shows an optimistic view of XML applications in database community. However, *understanding and classifying XML application into different categories so as to use the proper XML storage, index and processing techniques is crucial to deliver the value of XML in database management.*

VIII. REFERENCES

[1] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, C. Yu: "Making Database Systems Usable". SIGMOD 2007.

[2] M. Stonebraker, J.M. Hellerstein: "What Goes Around Comes Around." http://mitpress.mit.edu/books/chapters/0262693143chapm1.pdf

[3] S. Abiteboul: Querying Semi-Structured Data. ICDT, 1997, 1996.

[4] G. P. Copeland, S. Koshafian. A decomposition storage model. SIGMOD 268-279, 1985

[5] J. L. Beckmann, A. Halverson, R. Krishnamurthy, J. F. Naughton: Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format

[6] E. Chu, J. Beckmann, J. Naughton: The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets

[7] R. Agrawal, A. Somani, Y. Xu: Storage and Querying of E-Commerce Data. VLDB 2001

[8] T. J. Eggebraaten, J. W. Tenner, and J. C. Dubbels: A health-care data model based on the HL7 Reference Information Model. IBM System Journal, Vol 46, No 1, 2007

[9] X.Dong, A. Halevy: Indexing Dataspaces. SIGMOD 2007

[10] LORE: http://infolab.stanford.edu/lore/

[11] R. Murthy, S. Banerjee: XML Schemas in Oracle XML DB. VLDB 2003

[12] M. Krishnaprasad, Z. Hua Liu, A. Manikutty, J. Warner, V. Arora, S. Kotsovolos: Query Rewrite for XML in Oracle XML DB,VLDB 2004

[13] R. Murthy, Z. Hua Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, V. Krishnamurthy: Towards An Enterprise XML Architecture , SIGMOD 2005

[14] F. Ozcan, R. Cochrane , H. Pirahesh, J. Kleewein, K. Beyer, V. Josifovski , C. Zhang: System RX: One Part Relational, One Part XML, SIGMDO 2005

[15] M. Rys: XML and relational database management systems: inside Microsoft SQL Server 2005.

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

[17] Z. Hua Liu, M. Krishnaprasad, V. Arora: Native XQuery Processing in Oracle XML DB. SIGMOD 2005

[18] Z. Hua Liu, M. Krishnaprasad, Hui J. Chang, V. Arora: XMLTable Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007

[19] M. Stonebraker, P. Brown, D. Moore: Object-Relational DBMSs, Second Edition Morgan Kaufmann 1998

[20] PathExpression http://www.w3.org/TR/xpath20/#id-path-expressions

[21] XQuery Full Text: http://www.w3.org/TR/xquery-full-text/

[22] F Zemke, M. Rys, K. Kulkarni, J. Michels, B. Reinwald, F. Oczan, Z. Hua Liu, I. Davis, K. Hare, "XMLTable" , ISO/IEC JTC1/SC32 WG3:SIA-051 ANSI NCITS H2 2004-039 http://www.wiscorp.com/H2-2004-039-xmltable.pdf

[23] D. Florescu, D. Kossmann: Storing and Querying XML data using an RDBMS. IEEE Data Eng Bull 22(3):27-34 (1999).

[24] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton: Relational Databases for Querying XML documents: Limitations and Opportunities, VLDB 1999

[25] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita: Storing and Querying Ordered XML Using a Relational Database System: SIGMOD 2002

[26] F. Tian, D. DeWitt, J. Chen, C. Zhang: The Design and Performance Evaluation of Alternatives of Storage Strategies: SIGMOD Record, Vol 31, No 1, Mar 2002.

[27] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura: Xrel: A Path-Based Approach to Storage and Retrieval of XML documents Using Relational Databases

[28] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk: "Querying XML Views of Relational Data". VLDB 2001.

[29] M. Fernandez, A. Morishima, D. Suciu: "Efficient Evaluation of XML Middleware Queries", SIGMOD Conf., May 2001

[30] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M.Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A Native XML Database," VLDB Journal 11, No. 1, 274–291 (2002), http://www.eecs.umich.edu/db/timber/files/timber.pdf.

[31] M. J. Carey: Data delivery in a service-oriented world: the BEA aquaLogic data services platform. SIGMOD Conference 2006: 695-705

[32] V. R. Borkar, M. J. Carey, D. Lychagin, T. Westmann, D. Engovatov, N. Onose: Query Processing in the AquaLogic Data Services Platform. VLDB 2006: 1037-1048

[33] MarkLogic: http://www.marklogic.com/

[34] V. Krishnamurthy : Oracle XML DB Repository, SIGMOD 2003: 635

[35] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S.Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, S. B. Zdonik: C-Store: A Column-oriented DBMS. VLDB 2005: 553-564

[36] M. Krishnaprasad, Z. Hua Liu, A. Manikutty, J. Warner, V. Arora: Towards an industrial strength SQL/XML infrastructure, ICDE 2005.

[37] M. Nicola, I. Kogan, B. Schiefer: "An XML Transaction Processing Benchmark", SIGMOD 2007.

[38] XBRL: http://www.xbrl.org/Home/

[39] R.Murthy. Managing Complex Relationships in Oracle XML DB. XML2006 Conference, Boston, 2006.

[40] R.Murthy. From Trees to Graphs : Evolving XML for Enterprise Applications. XTech 2007 Conference, Paris, 2007.

[41] A. Balmin, K. S. Beyer, F. Ozcan, M. Nicola: On the Path to Efficient XML Queries. VLDB 2006

[42] M. Stonebraker: Implementation of Integrity Constraints and Views by Query Modification. SIGMOD Conference 1975: 65-78

[43] M.M. Moro, L. Lim, Y.Chang: "Schema Advisor for Hybrid Relational-XML DBMS". SIGMOD 2007

[44] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, J. Robie : XQueryP: Programming with XQuery, XIME-P 2006

[45] S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, S. Ceri, W. B. Croft, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, D. Gawlick, J. Gray, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, M. L. Kersten, M. J. Pazzani, M. Lesk, D. Maier, J. F. Naughton, H. Schek, T. K. Sellis, A. Silberschatz, M. Stonebraker, R. T. Snodgrass, J. D. Ullman, G. Weikum, J. Widom, S. B. Zdonik: The Lowell database research self-assessment. Commun. ACM 48(5): 111-118 (2005)

[46] J.McHugh, J. Widom, S. Abiteboul, Q. Luo, A. Rajaraman: Indexing Semistructured Data. http://www-db.stanford.edu/lore

[47] P.Bohannon, J. Freire, P. Roy, J. Siméon: From XML Schema to Relations: A Cost-Based Approach to XML Storage. ICDE 2002: 64

[48] D.C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, S. Fadden: DB2 Design Advisor: Integrated Automatic Physical Database Design. VLDB 2004

[49] S. Agrawal, S.Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, M. Syamala: Database Tuning Advisor for Microsoft SQL Server 2005

[50] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, Mo. Ziaduddin: Automatic SQL Tuning in Oracle 10g. VLDB 2004.

[51] D. Braga, A. Campi, S. Ceri, P. Spoletini: XQuery Layers, SIGMOD Record, Mar 2007

[52] P. P. Chen: The Entity-Relational Model – Toward a Unified View of Data, ACM Transactions on Database Systems, Vol 1, No 1, March 1976, page 9-36

[53] A. Novoselsky, Z. Hua Liu: XVM - A Hybrid Sequential-Query Virtual Machine for Processing XML Languages. PLAN-X 2008

[54] Z. Hua. Liu, T. Baby, S. Chandrasekar, Hui Chang: Towards a Physical XML independent XQuery/SQL/XML Engine , VLDB 2008